Lab 1 - UART

Nate Herbst
A02307138
Nathan Walker
A02364124

Introduction

This project required configuring a DE10-Lite development board to take a serial transmission at *19200* baud from a USB in a laptop and send the data back to the usb stick and laptop. This required using a terminal such as Putty running on the laptop to transmit the serial data from the keyboard. The FPGA then translated lowercase letters to upper case letters and vice versa. If a non letter character was transmitted the board would send back 'E' for error.

Procedure

Project Setup:

- The DE10-Lite board was configured in Quartus to initialize the Arduino header to get access to the boards RX and TX built in pins.

Configuring the *inout* pins:

- The Arduino pins featured on the pins are *inout* pins. This took some research online on how to ensure the pins were treated only as input or output pins. After some looking we found the solution of setting the inputs to 'Z' (high impedance) and the outputs we were free to set to whatever the data was.

Challenges and Solutions:

- Syntax errors: The main issue during this lab was recalling how to use VHDL. remembering key words and how they are used such as case, process, if then, and more. However, documentation from previous work and the internet proved useful to help relearn VHDL in a short time.

Results

The final implementation allowed the users to type in characters from the keyboard and in real-time see the board output the correct character. There was no unexpected behaviors observed in the final implementation and all the components functioned according to the specifications. Refer to appendix for simulation waveforms for the receiver and transmitter UART modules

Figures

1. **Wiring of the UART Connection** (Figure 1): Shows connections between the USB - Serial module (CP2102) and the DE10-Lite Board: [Pin 0 - RX | Pin 1 - TX]

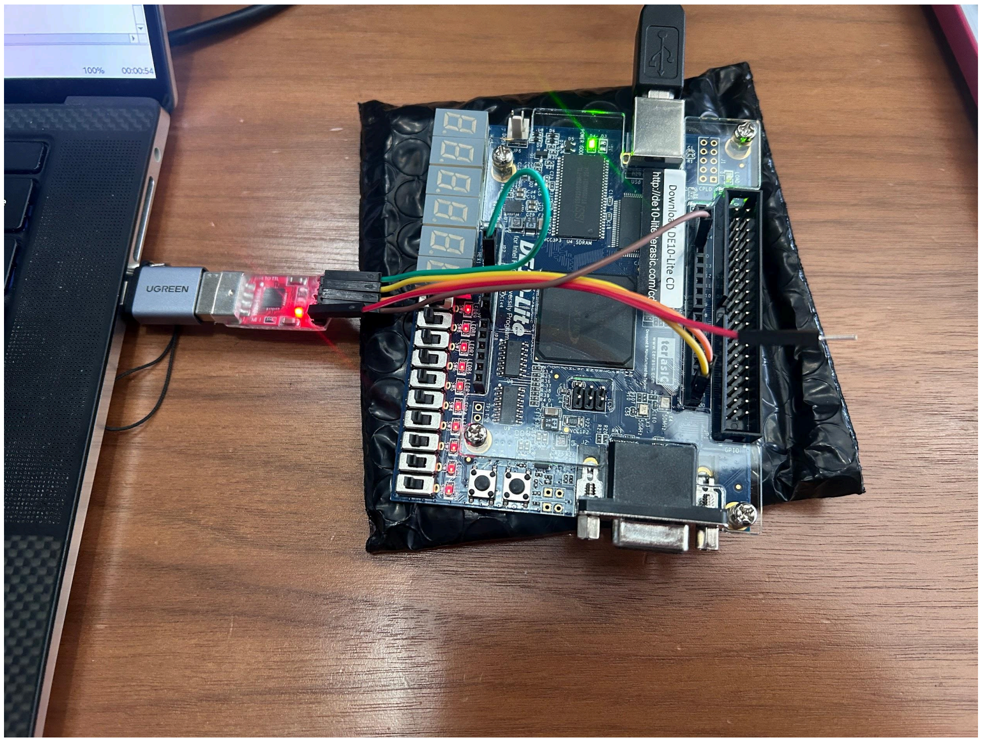


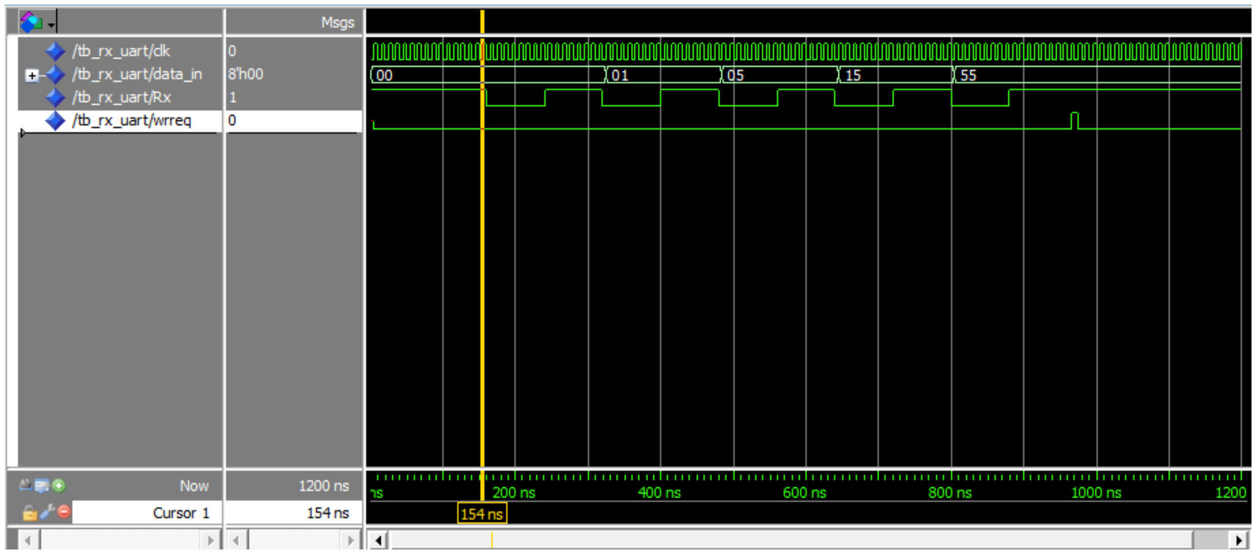
Figure 1: Wiring Connection for UART

Conclusion

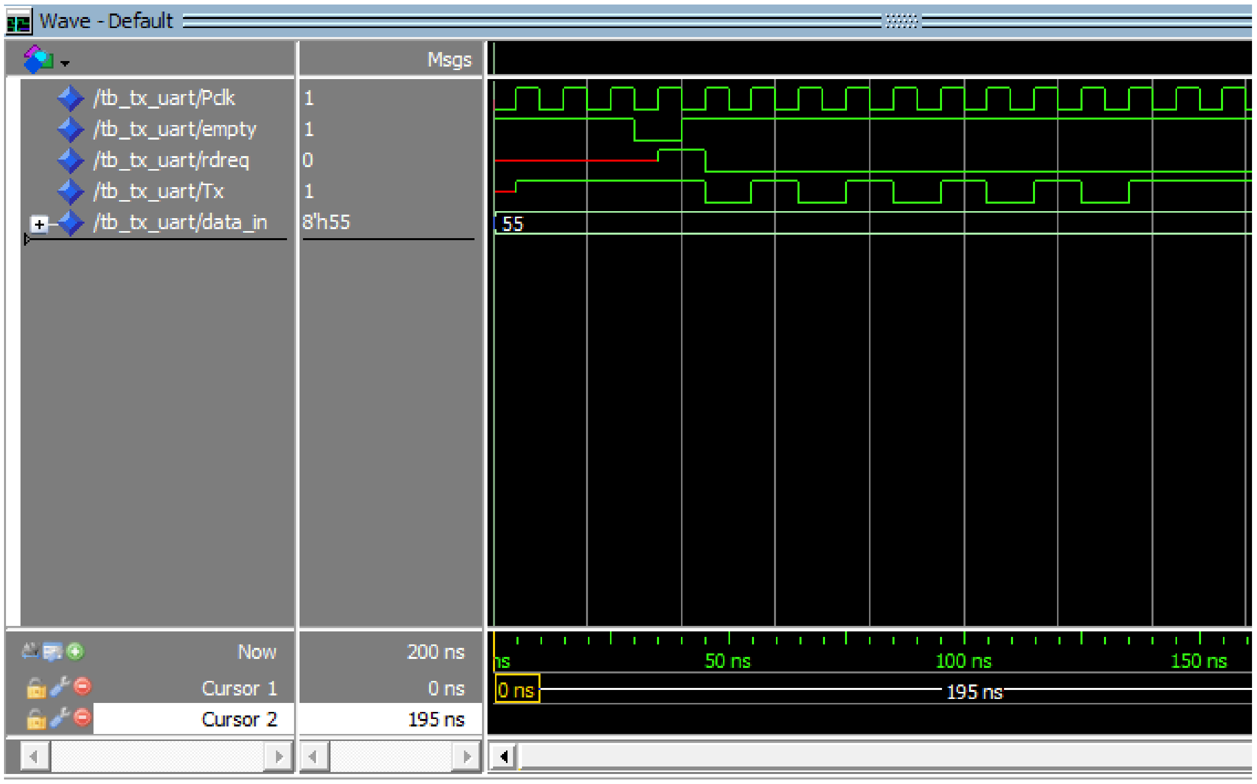
This lab provided valuable insights into FPGA configuration, ADC interfacing, and VHDL troubleshooting. Resolving issues such as the inverted reset pin and syntax errors helped refine our approach to debugging and code management, reinforcing best practices for FPGA development.

Appendix

Simulation tb_RX_UART



Simulation tb_tX_UART



Lab1_UART.vhd (top level file)

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

entity Lab1_UART is
   port (

   -- Clock
   ADC_CLK_10 : in std_logic;
   MAX10_CLK1_50 : in std_logic;
   MAX10_CLK2_50 : in std_logic;

   -- Arduino Header
   -- I/O 0 - Rx
   -- I/O 1 - Tx

   ARDUINO_IO      : inout std_logic_vector(15 downto 0);
   ARDUINO_RESET_N : inout std_logic

   );
end Lab1_UART;

architecture component_list of Lab1_UART is

   -- Component Declarations

   component PLL_UART
       PORT
       (
           inclk0      : IN STD_LOGIC  := '0';
           c0      : OUT STD_LOGIC ;
           c1      : OUT STD_LOGIC
       );
   end component;

   component RX_UART
       port
       (
           clk : in std_logic;
           Rx: in std_logic;
           wrreq: out std_logic;
           data_out: out std_logic_vector(7 downto 0)
       );
   end component;

   component TX_UART
       port
       (
           Pclk : in std_logic;
           Tx: out std_logic;
           rdreq: out std_logic;
           empty: in std_logic;
           data_in: in std_logic_vector(7 downto 0)
       );
   end component;

   component CASE_TRANSLATOR
       port
       (
           wrclk : in std_logic;
           rdclk : in std_logic;
           data_in: in std_logic_vector(7 downto 0);
           translate_req: in std_logic;
           read_req: in std_logic;
           data_out: out std_logic_vector(7 downto 0);
           empty: out std_logic
       );
   end component;

   -- Signals
   signal Rx : std_logic;
   signal Tx : std_logic;

   -- Clock Signals
   signal clk_rx_8x : std_logic; -- 153.6 kHz (8 * 19200)
   signal clk_tx_1x : std_logic; -- 19.2 kHz

   -- Interconnect Signals
```

```
   signal rx_data_byte : std_logic_vector(7 downto 0);
   signal rx_data_valid : std_logic;

   signal tx_data_byte : std_logic_vector(7 downto 0);
   signal tx_read_req : std_logic;
   signal fifo_empty : std_logic;

begin

   -- UART IO Assignments
   Rx <= ARDUINO_IO(0);           -- Read from IO0
   ARDUINO_IO(0) <= 'Z';          -- Tri-state IO0 so it can be used as
input

   ARDUINO_IO(1) <= Tx;           -- Write internal Tx to IO1

   ARDUINO_IO(15 downto 2) <= (others => 'Z'); -- Set unused pins to
high-Z


   -- PLL Instantiation
   pll_inst : PLL_UART
   PORT MAP (
       inclk0 => MAX10_CLK1_50,
       c0     => clk_rx_8x,
       c1     => clk_tx_1x
   );

   -- RX UART Instantiation
   rx_inst : RX_UART
   PORT MAP (
       clk      => clk_rx_8x,
       Rx       => Rx,
       wrreq    => rx_data_valid,
       data_out => rx_data_byte
   );

   -- Casing Translator & FIFO Instantiation
   translator_inst : CASE_TRANSLATOR
   PORT MAP (
       wrclk          => clk_rx_8x,
       rdclk          => clk_tx_1x,
       data_in        => rx_data_byte,
       translate_req  => rx_data_valid,
       read_req       => tx_read_req,
       data_out       => tx_data_byte,
       empty          => fifo_empty
   );

   -- TX UART Instantiation
   tx_inst : TX_UART
   PORT MAP (
       Pclk    => clk_tx_1x,
       Tx      => Tx,
       rdreq   => tx_read_req,
       empty   => fifo_empty,
       data_in => tx_data_byte
   );

end component_list;
```

RX_UART.vhd

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

entity RX_UART is
   port
   (
       --CLK input (8x Baud Rate)--
       clk : in std_logic;
       --input bit--
       Rx: in std_logic;
       --output bits--
       wrreq: out std_logic;
       data_out: out std_logic_vector(7 downto 0)
   );
end entity RX_UART;
```

```
architecture component_list of RX_UART is

   type bits is (start, data, stop);
   type oversample is (idle, throw_away_start, acquire, throw_away_end);

   -- Initialize state machines start states
   signal bit_state : bits := start;
   signal sample_state : oversample := idle;

   signal oversample_counter : integer range 0 to 7 := 0;
   signal one_count : integer range 0 to 8 := 0;
   signal zero_count : integer range 0 to 8 := 0;
   signal data_count : integer range 0 to 8 := 0;

   signal true_data : std_logic := '0';
   signal sample_rdy : std_logic := '0';

   signal rx_data_buffer : std_logic_vector(7 downto 0) := (others =>
'0');

begin

   data_out <= rx_data_buffer;

   --state machines--
   process (clk)
   begin
       if rising_edge(clk) then

           -- default one-cycle signals
           wrreq <= '0';

           -- ==========================================
           -- Bit State Machine (Tracks Frame Progress)
           -- ==========================================
           case bit_state is
               when start =>
                   -- Wait for the Oversample machine to tell us it found
a bit
```

```
                    if sample_rdy = '1' then
                        -- We expect a '0' for a valid Start bit
                        if true_data = '0' then
                            bit_state <= data;
                            data_count <= 0;
                        end if;
                        -- If true_data is '1', it was a glitch, stay in
start (and Oversample machine will go back to IDLE)
                    end if;

                when data =>
                    if sample_rdy = '1' then
                        rx_data_buffer(data_count) <= true_data; -- Shift
in LSB first? Standard is LSB. Array index matches.

                        if data_count = 7 then
                            bit_state <= stop;
                        else
                            data_count <= data_count + 1;
                        end if;
                    end if;

                when stop =>
                    if sample_rdy = '1' then
                        -- We expect a '1' for a valid Stop bit
                        if true_data = '1' then
                            wrreq <= '1'; -- Good valid byte!
                        end if;
                        -- Regardless of whether stop bit was valid frame
or error, we reset to look for next start
                        bit_state <= start;
                    end if;
            end case;

            -- ==========================================
            -- Oversample State Machine (Samples the Line)
            -- 8x Oversampling (8 Clocks per bit)
            -- ==========================================
            case sample_state is
                when idle =>
```

```
                   sample_rdy <= '0';
                   oversample_counter <= 0;
                   zero_count <= 0;
                   one_count <= 0;

                   -- Synchronize to falling edge of Start Bit
                   -- Only start if we are expecting a start bit
(bit_state = start)
                   -- or if we are already in the middle of a frame
(bit_state != start)
                   -- But actually, loop control is handled in
throw_away_end.

                   if Rx = '0' and bit_state = start then
                       sample_state <= throw_away_start;
                   end if;

               when throw_away_start =>
                   -- Skip first 2 ticks (Ticks 0, 1)
                   if oversample_counter < 1 then
                       oversample_counter <= oversample_counter + 1;
                   else
                       oversample_counter <= 0;
                       sample_state <= acquire;
                       zero_count <= 0;
                       one_count <= 0;
                   end if;

               when acquire =>
                   -- Vote on middle 4 ticks (Ticks 2, 3, 4, 5)
                   if oversample_counter < 3 then
                       oversample_counter <= oversample_counter + 1;
                       if Rx = '0' then
                           zero_count <= zero_count + 1;
                       else
                           one_count <= one_count + 1;
                       end if;
                   else
                       -- Final vote tally
                       if Rx = '0' then
```

```
                            zero_count <= zero_count + 1;
                        else
                            one_count <= one_count + 1;
                        end if;

                        oversample_counter <= 0;
                        sample_state <= throw_away_end;
                    end if;

                when throw_away_end =>
                    -- Decision Time (happens immediately upon entering
state)
                    -- Note: zero_count/one_count updated at end of
acquire, so values are valid now.
                    if zero_count > one_count then
                        true_data <= '0';
                    else
                        true_data <= '1';
                    end if;
                    sample_rdy <= '1'; -- Signal valid data for one cycle

                    -- Skip last 2 ticks (Ticks 6, 7)
                    if oversample_counter < 1 then
                        oversample_counter <= oversample_counter + 1;
                    else
                        oversample_counter <= 0;
                        sample_rdy <= '0'; -- Clear ready flag

                        -- Where do we go next?
                        if bit_state = stop then
                            -- We just finished the Stop bit. Go to IDLE to
wait for next Start bit edge.
                            sample_state <= idle;
                        elsif bit_state = start and true_data = '1' then
                            -- We thought we saw a Start bit, but the
voting said it was '1' (Glitch). Reset.
                            sample_state <= idle;
                        else
                            -- We are in the middle of data bits (or moving
Start->Data),
                            -- jump straight to next bit sampling without
waiting for edge.
                            sample_state <= throw_away_start;
                        end if;
                    end if;
            end case;

        end if;
    end process;

end component_list;
```

TX_UART.vhd

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

entity TX_UART is
   port
   (
       --CLK input (19200 Hz)--
       Pclk : in std_logic;
       --output bit--
       Tx: out std_logic;
       rdreq: out std_logic;
       --input bits--
       empty: in std_logic;                    -- Signal to start
transmission
       data_in: in std_logic_vector(7 downto 0) -- Byte to transmit
   );
end entity TX_UART;

architecture component_list of TX_UART is

   type state_type is (idle, start_bit, data_bits, stop_bit);
   signal state : state_type := idle;

   -- Index to track which bit (0-7) is being sent
```

```vhdl
    signal bit_index : integer range 0 to 7 := 0;

    -- Register to latch the data
    signal tx_data_reg : std_logic_vector(7 downto 0) := (others => '0');

begin

    process (Pclk)
    begin
        if rising_edge(Pclk) then

            case state is
                when idle =>
                    Tx <= '1'; -- UART Idle line is High
                    bit_index <= 0;

                    -- When write request is received, latch data and start
                    if empty = '0' then
                        rdreq <= '1';
                        tx_data_reg <= data_in;
                        state <= start_bit;
                    end if;

                when start_bit =>
                    rdreq <= '0';
                    Tx <= '0'; -- Start bit is Low
                    state <= data_bits;

                when data_bits =>
                    Tx <= tx_data_reg(bit_index); -- Send LSB first

                    -- Move to next bit or stop bit
                    if bit_index < 7 then
                        bit_index <= bit_index + 1;
                    else
                        bit_index <= 0;
                        state <= stop_bit;
                    end if;

                when stop_bit =>
```

```
                    Tx <= '1'; -- Stop bit is High
                    state <= idle; -- Transmission complete

            end case;
        end if;
    end process;

end component_list;
```

CASE_TRANSLATOR.vhd

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

-- This module translates lowercase ASCII letters to uppercase, or
uppercase to lowercase.
-- If there are non-alphabetic characters, an "E" is passed through.

-- There is a FiFo internally instantiated in this module, so when data is
the data is translated, it then stored in the FiFo. When data is available
in the FIFO, empty is set low and can be read out.

entity CASE_TRANSLATOR is
    port
    (
        --input bits--
        wrclk : in std_logic;
        rdclk : in std_logic;

        data_in: in std_logic_vector(7 downto 0); -- Data to be translated

        translate_req: in std_logic;  -- Signal to start translation when
RX_UART's wrreq is high
        read_req: in std_logic;       -- Signal to read from FIFO coming
form TX_UART

        --output bits--
        data_out: out std_logic_vector(7 downto 0); -- Translated data
```

```
       empty: out std_logic -- Signal to indicate if FIFO is empty
   );
end entity CASE_TRANSLATOR;

architecture component_list of CASE_TRANSLATOR is

   component FIFO
       PORT
       (
           data         : IN STD_LOGIC_VECTOR (7 DOWNTO 0);
           rdclk        : IN STD_LOGIC ;
           rdreq        : IN STD_LOGIC ;
           wrclk        : IN STD_LOGIC ;
           wrreq        : IN STD_LOGIC ;
           q        : OUT STD_LOGIC_VECTOR (7 DOWNTO 0);
           rdempty      : OUT STD_LOGIC
       );
   end COMPONENT;

   signal translated_data : std_logic_vector(7 downto 0);
   signal fifo_write_req : std_logic := '0';

begin
   -- Translation Logic
   process(wrclk)
       variable char_code : integer range 0 to 255;
   begin
       if rising_edge(wrclk) then
           -- Default values
           fifo_write_req <= '0';

           if translate_req = '1' then

               char_code := to_integer(unsigned(data_in));

               if (char_code >= 97 and char_code <= 122) then -- Lowercase
'a' to 'z'
                   -- Convert to Uppercase
                   translated_data <= 
std_logic_vector(to_unsigned(char_code - 32, 8));
               elsif (char_code >= 65 and char_code <= 90) then -- 
Uppercase 'A' to 'Z'
                   -- Convert to Lowercase
                   translated_data <= 
std_logic_vector(to_unsigned(char_code + 32, 8));
               else
                   translated_data <= x"45"; -- 'E'
               end if;

               fifo_write_req <= '1';
           end if;
       end if;
   end process;

   -- FIFO Instantiation
   fifo_inst : FIFO
   PORT MAP
   (
       data         => translated_data,
       rdclk        => rdclk,
       rdreq        => read_req,
       wrclk        => wrclk,
       wrreq        => fifo_write_req,
       q        => data_out,
       rdempty      => empty
   );

end architecture component_list;
```