UART Lab Report

Matthew Crump A01841001
Nicholas Williams A02057223

Objectives

Review concepts of digital design, including Finite State Machines, FIFOs, PLLs, serial communication, ASCII, VHDL, and learn how to use the Intel Pin Planner. All of these skills will be combined to create a UART to change the case of the letter received and then send it back.

Procedure

We used the system Builder tool to generate the initial project with the clock inputs, push buttons, and GPIO header. This was then translated from Verilog to VHDL. We knew that the UART would be used in future labs, so we made it a separate module while the part that would translate the character case would just be in the top module. We reviewed the UART timing diagrams given on the lecture slides. From those timing diagrams we derived the state machines for both the RX and TX side of the UART. A generic was used to easily configure the BAUD rate of our UART. This number (clks_per_bit) was calculated by taking the FPGA clock frequency divided by the BAUD rate. This number would tell us how many FPGA clocks would occur per bit of serial data. This generic will eventually be changed to a register that can be configured by a CPU to change the BAUD rate while running.

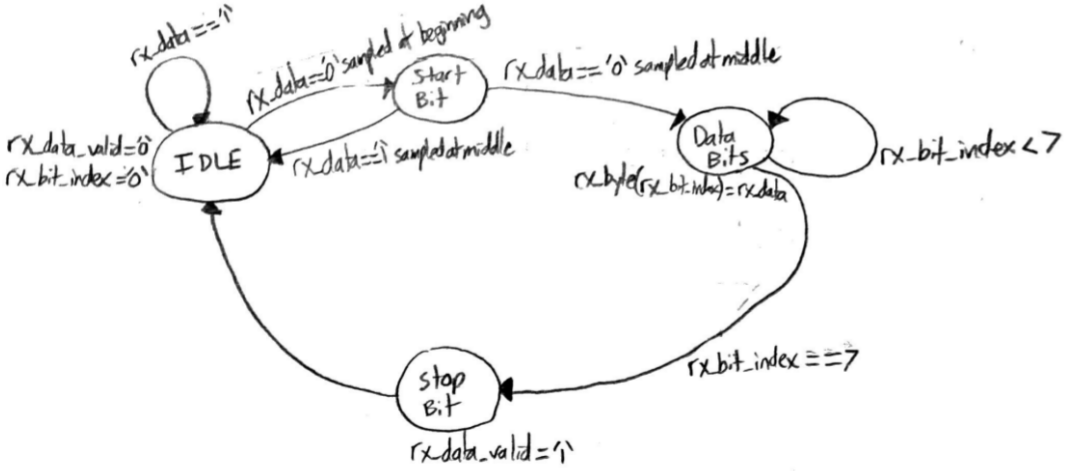


Figure 1. RX State Machine

For both the RX and TX state machine, we used 4 states each. The first state waits for the start bit and upon receiving the startbit, the machine changes state. The next state waits for half of the bit rate to align the sampling with the center of the data bit. After this time, the state machine transitions to the third state. Here, the state machine waits for the counter to expire and shifts one data bit into the receive buffer. After receiving 8 data bits, the state machine switches to the last state to verify that a stop bit has been sent and if so, raises the data_valid flag and goes back to the idle state. Similar steps were done for the transmit side of the UART.

A state machine in the top module waits for the rx data valid signal. Upon receiving this signal, it looks at the received byte and if it is within the range of a capital letter it adds 32 or if it is within the range of a lower case letter it subtracts 32. This result is then saved into the TX send byte and the TX data valid flag is raised. If the received byte is not an alphabetical letter, then an ‘E’ is saved into the TX send byte.

Results

For the first iteration, we just made a simple loopback configuration. This was done to ensure that the data received could be sent back unaltered. This first test was successful, and created a good foundation to build off of. We cut the loopback and then put our state machine inline to translate the received bytes. This also worked as expected since each letter case that was sent was flipped and any non-alphabetical characters were sent back as an “E”. The TX light on the board also lit up on each transmit for debugging purposes to easily verify that the state machines were working.

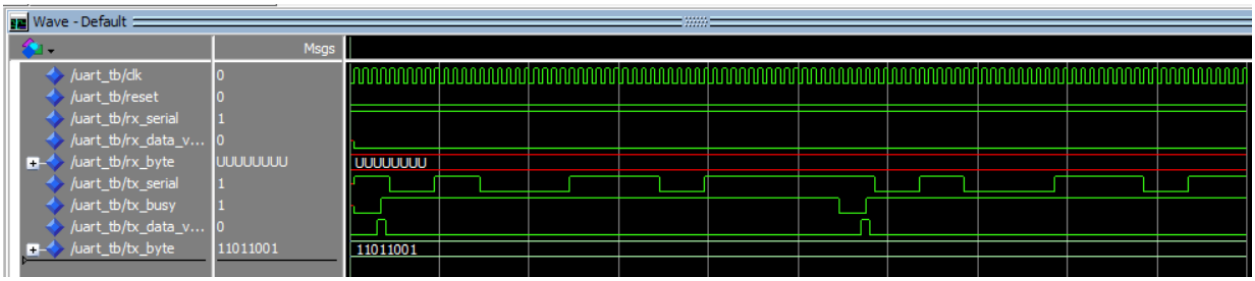
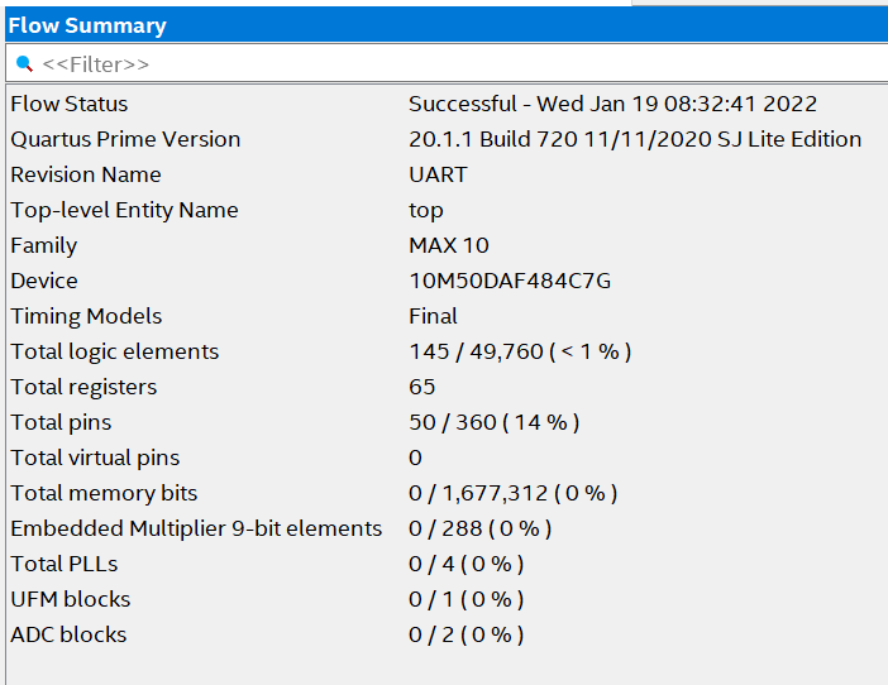


Figure 2. Simulation Results

Figure 2 shows a wave diagram that resulted from simulating the transmit state machine. As the tx_data_valid line goes high the tx_busy line goes high to show that it is transmitting the data. The tx_serial line begins to transmit the data starting with a start bit of zero and ending with a stop bit of one.

Everything seemed to work with this lab. We realized that we forgot to add reset logic and so we quickly added that to the top module's state machine. The reset just sets the state machine back to the idle state. In the idle state each of the values are reset to 0. Overall, everything worked. Figure 3 shows the Flow Summary. Everything is to be expected in the flow summary. If we had not used the system builder to set up the pin header for us then we could have used less pins.



Flow Summary

<<Filter>>

Flow Status	Successful - Wed Jan 19 08:32:41 2022
Quartus Prime Version	20.1.1 Build 720 11/11/2020 SJ Lite Edition
Revision Name	UART
Top-level Entity Name	top
Family	MAX 10
Device	10M50DAF484C7G
Timing Models	Final
Total logic elements	145 / 49,760 (< 1 %)
Total registers	65
Total pins	50 / 360 (14 %)
Total virtual pins	0
Total memory bits	0 / 1,677,312 (0 %)
Embedded Multiplier 9-bit elements	0 / 288 (0 %)
Total PLLs	0 / 4 (0 %)
UFM blocks	0 / 1 (0 %)
ADC blocks	0 / 2 (0 %)

Figure 3. Flow Summary

Conclusion

We successfully designed a UART onto our DE10-Lite board that can interface with a serial terminal on a computer. When a lower-case letter is pressed on the computer's keyboard then an upper-case letter appears inside the terminal. Likewise, when an upper-case letter is pressed then a lower-case letter appears inside the terminal. Any non-alphabetical letter that is pressed results in an "E" being printed to the serial terminal. We learned how to properly setup pins inside of the qsf file. Overall, we fulfilled the requirements for this lab and met the learning objectives along the way.

Appendix A

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

entity top is

    port (
        ADC_CLK_10 : in std_logic;
        MAX10_CLK1_50 : in std_logic;
        MAX10_CLK2_50 : in std_logic;
        rstn : in std_logic;
        GPIO : inout std_logic_vector(35 downto 0);
        LEDR : out std_logic_vector(9 downto 0)
    );
end top;

architecture behave of top is
    type state_type is (s_IDLE, s_SEND);
    signal convert_state : state_type;
    signal rx_pin : std_logic;
    signal tx_pin : std_logic;
    signal rx_data_valid : std_logic;
    signal tx_data_valid : std_logic;
    signal tx_busy : std_logic;
    signal saved_byte : std_logic_vector(7 downto 0);
    signal received_byte : std_logic_vector(7 downto 0);

begin

    UART1: entity work.uart(rtl)
        generic map (
            -- clk_freq / BAUD = clks_per_bit
            -- 50MHz / 19200 = 2604
            clks_per_bit => 2604
        )
        port map (
            clk => MAX10_CLK1_50,
            -- rx
            rx_serial => rx_pin,
            rx_data_valid => rx_data_valid,
            rx_byte => received_byte,
            -- tx
            tx_serial => tx_pin,
            tx_data_valid =>tx_data_valid,
            tx_byte => saved_byte,
            tx_busy => tx_busy
        );

    p_TRANSLATE : process (MAX10_CLK1_50)
    begin
        if rstn = '0' then
            convert_state <= s_IDLE;
        elsif rising_edge(MAX10_CLK1_50) then
            case convert_state is
                when s_IDLE =>
                    if rx_data_valid = '1' then
                        convert_state <= s_SEND;
                        tx_data_valid <= '1';

                        if received_byte >= "01000001" and received_byte <= "01011010" then
                            saved_byte <= received_byte + "00100000";
                        elsif received_byte >= "01100001" and received_byte <= "01111010" then
```

```
                        saved_byte <= received_byte - "00100000";
                    else
                        saved_byte <= "01000101";
                    end if;
                else
                    tx_data_valid <= '0';
                end if;

            when s_SEND =>
                tx_data_valid <= '0';
                if tx_busy = '0' then
                    convert_state <= s_IDLE;
                end if;
        end case;
    end if;
end process;

rx_pin <= GPIO(0);
GPIO(1) <= tx_pin;
LEDR(0) <= tx_busy;
LEDR(9 downto 1) <= (others => '0');

end behave;
```

Appendix B

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

entity uart is
    generic (
        -- clk_freq / BAUD = clks_per_bit
        clks_per_bit : integer := 11
    );
    port (
        clk : in std_logic;
        -- rx
        rx_serial : in std_logic;
        rx_data_valid : out std_logic;
        rx_byte : out std_logic_vector(7 downto 0);
        -- tx
        tx_serial : out std_logic;
        tx_busy : out std_logic;
        tx_data_valid : in std_logic;
        tx_byte : in std_logic_vector(7 downto 0)
    );
end uart;

architecture rtl of uart is
    type state_type is (s_IDLE, s_START_BIT, s_DATA_BITS, s_STOP_BIT);
    signal tx_state : state_type;
    signal rx_state : state_type;
    -- rx
    signal rx_data_raw : std_logic;
    signal rx_data : std_logic;
    signal rx_clock_count : integer range 0 to clks_per_bit-1 := 0;
    signal rx_bit_index : integer range 0 to 7 := 0;
    -- tx
    signal tx_data : std_logic_vector(7 downto 0);
    signal tx_clock_count : integer range 0 to clks_per_bit-1 := 0;
    signal tx_bit_index : integer range 0 to 7 := 0;
begin

    -- double-register incoming data to synchronize it to this clock domain
    -- removes metastability
    p_SYNC : process (clk)
    begin
        if rising_edge(clk) then
            rx_data_raw <= rx_serial;
            rx_data <= rx_data_raw;
        end if;
    end process;

    ------------------------------
    -- RX PROCESS
    ------------------------------

    p_RX : process (clk)
    begin
        if rising_edge(clk) then
            case rx_state is
                when s_IDLE =>
                    rx_data_valid <= '0';
                    rx_clock_count <= 0;
                    rx_bit_index <= 0;

                    if rx_data = '0' then -- start bit detected
                        rx_state <= s_START_BIT;
```

```
                    end if;

                when s_START_BIT =>
                    if rx_clock_count = ((clks_per_bit-1)/2) then -- count to the middle of
the start bit
                        if rx_data = '0' then -- still low = valid start bit
                            rx_clock_count <= 0;
                            rx_state <= s_DATA_BITS;
                        else -- bad start bit go back to IDLE
                            rx_state <= s_IDLE;
                        end if;
                    else
                        rx_clock_count <= rx_clock_count + 1;
                    end if;

                when s_DATA_BITS =>
                    if rx_clock_count < clks_per_bit - 1 then
                        rx_clock_count <= rx_clock_count + 1;
                    else
                        rx_clock_count <= 0;
                        rx_byte(rx_bit_index) <= rx_data; -- save the bit

                        if rx_bit_index < 7 then -- check how many bits have been saved
                            rx_bit_index <= rx_bit_index + 1;
                        else
                            rx_bit_index <= 0;
                            rx_state <= s_STOP_BIT;
                        end if;
                    end if;

                when s_STOP_BIT =>
                    if rx_clock_count < clks_per_bit - 1 then
                        rx_clock_count <= rx_clock_count + 1;
                    else
                        rx_data_valid <= '1';
                        rx_state <= s_IDLE;
                    end if;

                when others =>
                        rx_state <= s_IDLE;

            end case;
        end if;
    end process;

    ------------------------------
    -- TX PROCESS
    ------------------------------

    p_TX : process (clk)
    begin
        if rising_edge(clk) then
            case tx_state is
                when s_IDLE =>
                    tx_serial <= '1';
                    tx_busy <= '0';
                    tx_clock_count <= 0;
                    tx_bit_index <= 0;

                    if tx_data_valid = '1' then
                        tx_busy <= '1';
                        tx_data <= tx_byte;
                        tx_state <= s_START_BIT;
                    end if;

                when s_START_BIT =>
                    tx_serial <= '0';
```

```
                if tx_clock_count < clks_per_bit - 1 then
                    tx_clock_count <= tx_clock_count + 1;
                else
                    tx_clock_count <= 0;
                    tx_state <= s_DATA_BITS;
                end if;

            when s_DATA_BITS =>
                tx_serial <= tx_data(tx_bit_index);

                if tx_clock_count < clks_per_bit - 1 then
                    tx_clock_count <= tx_clock_count + 1;
                else
                    tx_clock_count <= 0;

                    if tx_bit_index < 7 then
                        tx_bit_index <= tx_bit_index + 1;
                    else
                        tx_bit_index <= 0;
                        tx_state <= s_STOP_BIT;
                    end if;
                end if;

            when s_STOP_BIT =>
                tx_serial <= '1';

                if tx_clock_count < clks_per_bit - 1 then
                    tx_clock_count <= tx_clock_count + 1;
                else
                    tx_state <= s_IDLE;
                end if;

            when others =>
                tx_state <= s_IDLE;

        end case;
    end if;
end process;
end rtl;
```