5/29/2021

Digital Design

Jacobs University Bremen

Priontu Chowdhury
MATRICULATION: 30004005

Introduction

In this lab, we learned how to design from a specification and a block diagram. In the process of developing a UART transmitter we received insight on how to properly clock something at a fraction of the clock speed, develop a state machine, a counter and a multiplexer in VHDL, and instance one entity of the model inside another. We also had an introduction to the multi-segment coding style, which is very useful in compartmentalizing and organizing code, and also makes debugging much easier.

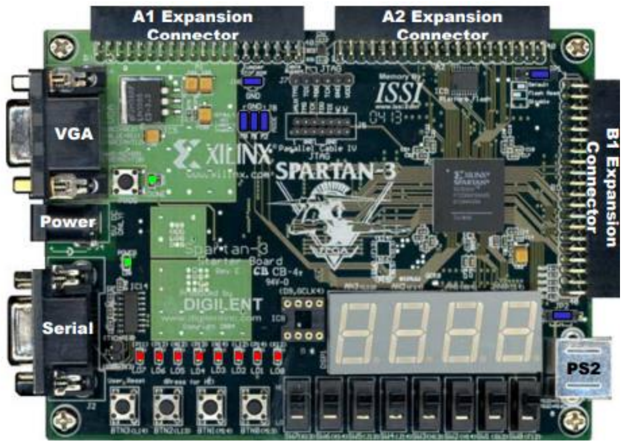
Theory

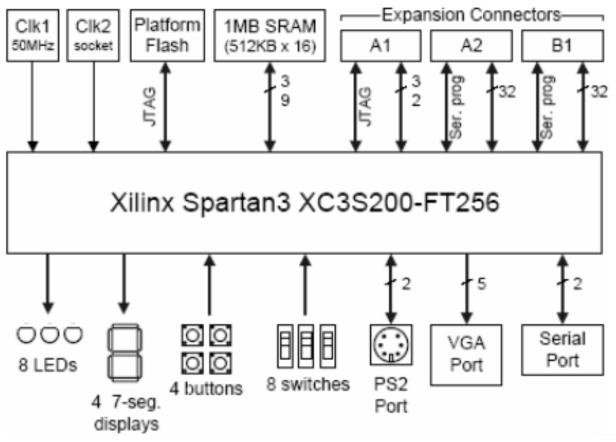
UART is a communication protocol. In UART communication, two devices communicate with each other using the UART communication protocol. The UART device that is transmitting data converts parallel data from a controlling device into serial form, and then transmits it in serial to the receiving UART device. Two wires are required to transmit data between the two devices. Data flow is directed from the Tx pin of the transmitting device to the Rx pin of the receiving device.

Transmission of data in UART transmission is achieved asynchronously – no clock signal is used to synchronize the output bits. This is accomplished by adding a start and stop but to the data packet being transferred. These bits define the beginning and end of the data packet, through which the UART device recognizes when to start reading bits and when to stop.

Hardware Specification:

We use a Spartan-3 FPGA Board in order to develop the UART Transmitter.





Design objectives

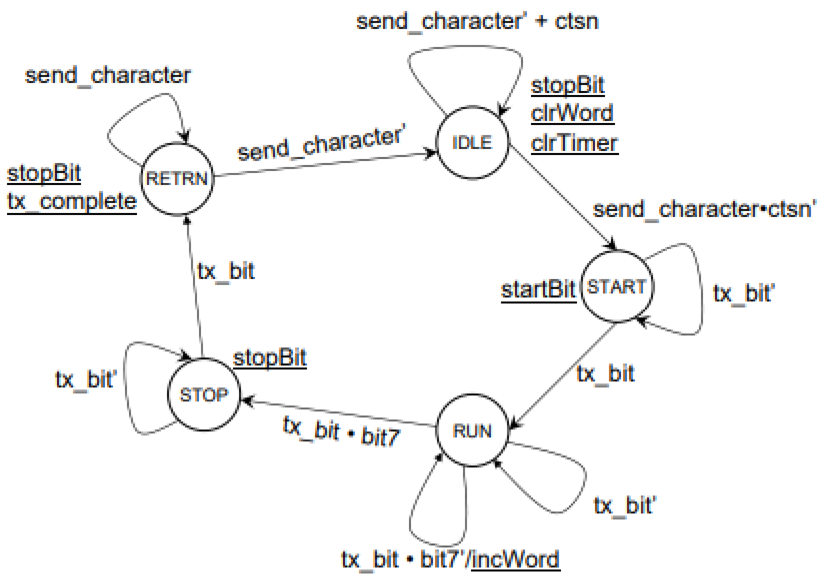
Based on the design specifications provided, the following steps need to be fulfilled in order to meet our objectives:

1. The transmitting device receives data in parallel from the controlling device through the data bus.
2. “Start”, “parity” and “stop” bits are added to the data’s bit-array.
3. The data packet is then sent serially from the transmitting UART device to the receiving UART device. The data-line is sampled by the receiving UART device at a configured baud-rate.
4. The additional “start”, “parity” and “stop” bits are discarded by the receiving UART device.
5. The receiver converts the serial data back into parallel and transfers it to the data bus on the receiving side.

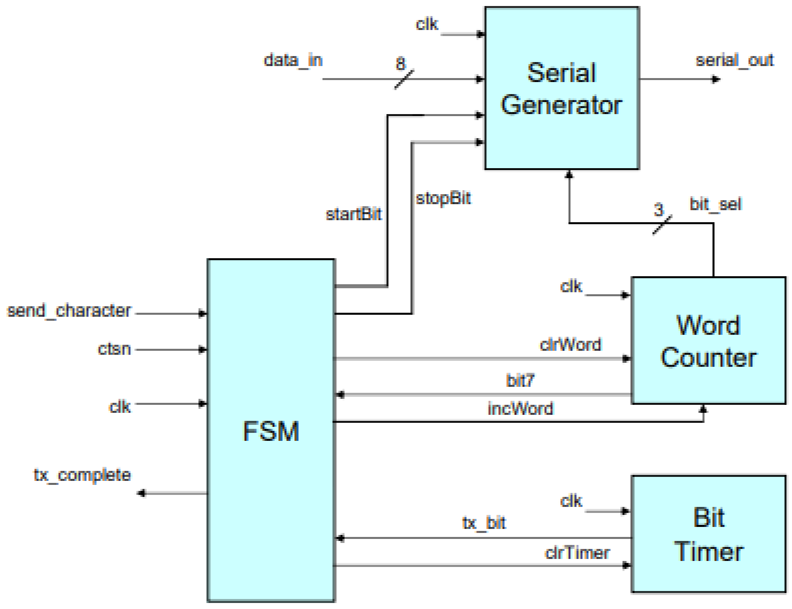
Design Specification

We are provided with the following design specifications in the UART Labs Documentation:

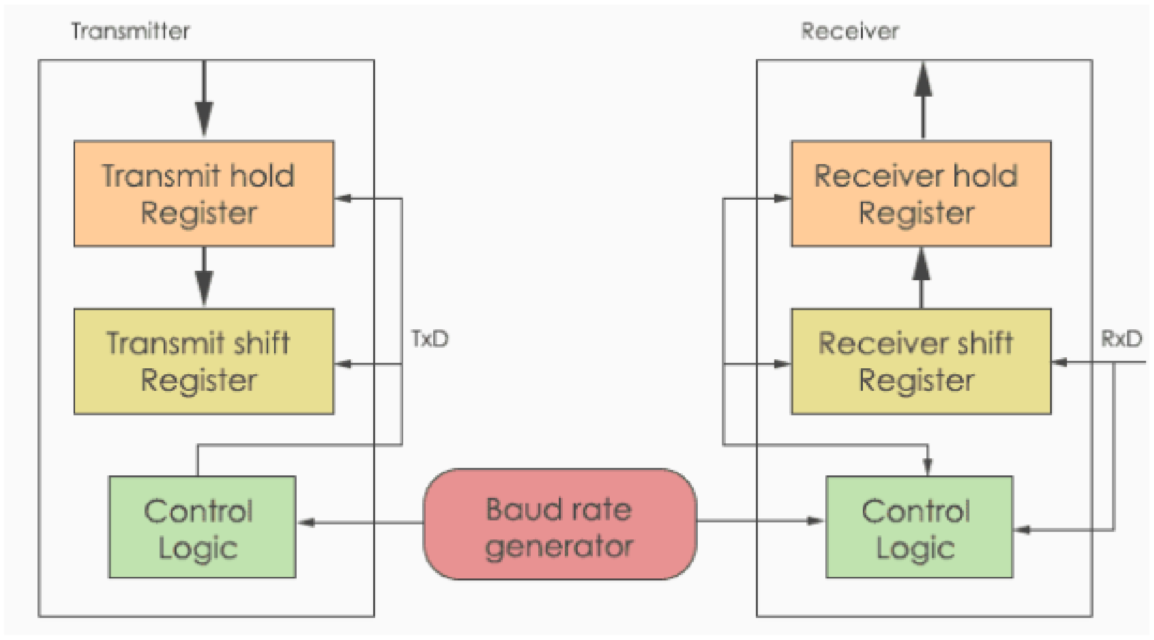
FSM Design:



Transmitter Block Diagram:



Conceptual Diagram:



Explanation on how the FSM and block diagrams are working in the UART Transmitter are provided in the code in the form of comments.

Design File

The following top design was provided:

tx_testbench.vhd

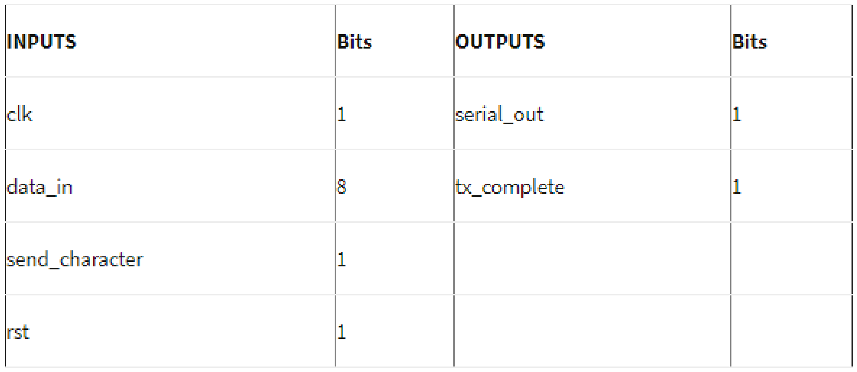
Source Code:

```
------------------------
--Stephen West
--Transmitter Test Bench
--sept 2007
------------------------
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity tx_test is
	port(
		clk,rst,button: in std_logic;
		switches:in std_logic_vector(7 downto 0);
		txd,done:out std_logic
	);
end entity;

architecture tx_test of tx_test is
	component tx is
		port(
			clk,send_character,rst:in std_logic;
			data_in:in std_logic_vector(7 downto 0);
			serial_out,tx_complete:out std_logic
		);
	end component;
	signal send_character,debounce1,debounce2,timer1,timer2:std_logic;
	signal timer:unsigned(19 downto 0);
begin
	tx_component:tx port map(
		clk=>clk,send_character=>send_character,rst=>rst,
		data_in=>switches,
		serial_out=>txd,tx_complete=>done
	);

	debounce:process(clk,button)
	begin
		if clk'event and clk='1' then
```

```
                        if rst='1' then
                                debounce1<='0';
                                debounce2<='0';
                                timer1<='0';
                                timer2<='0';
                                timer<=(others=>'0');
                        else
                                debounce1<=button;
                                if timer1='1' and timer2='0' then
                                   debounce2<=debounce1;
                                end if;
                                if (debounce1 ='1' and debounce2='0') or (debounce1='0' and
debounce2='1') then
                                        timer<=timer+1;
                                else
                                        timer<=(others=>'0');
                                end if;
                                timer1<=timer(13);--1ms
                                timer2<=timer1;
                        end if;
                end if;

                send_character<= debounce2;
        end process;
end tx_test;
```

Objectives

Create a VHDL source file with the following *I/O specification:*



INPUTS	Bits	OUTPUTS	Bits
clk	1	serial_out	1
data_in	8	tx_complete	1
send_character	1		
rst	1		

tx Inputs:

- clk is the system 50 MHz clock
- data_in is the byte that you want to send

- send_character is a control signal that is high when the byte is ready to be sent
- rst is the reset generated by the clock/reset generator that pulses high on power on.

tx Outputs:

- serial_out is the serial data to be transmitted. (Connected to txd of the Serial Port interface).
- tx_complete goes high to signal that the byte has been sent. This should cause send_character to be de-asserted externally.

Implementation

Source Code:

tx.vhd

```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
USE IEEE.NUMERIC_STD.ALL;

   --variables named according to specification
   --in order to improve readability

ENTITY tx_EXECUTABLE IS
   port(
      clk,rst: in std_logic; --Global Clock and Reset
      send_character: in std_logic; --
      data_in: in std_logic_vector(7 downto 0); --input data
                                    --(data to be transmitted)

      --Transmission detection parameters:
      serial_out, tx_complete: out std_logic --serial_out: start bit
                                  --tx_complete: stop bit
   );
END ENTITY;

ARCHITECTURE tx_BEHAVIORAL OF tx_EXECUTABLE IS
   --Finite State Machine Design specific state declaration:
   type FSM_STATE is (IDLE,START,RUN,STOP,RETRN);

   --state registers for management of FSM states:
   signal s_reg: FSM_STATE;
   signal s_next: FSM_STATE;

   --FSM Design Spec requirements:
   signal startBit, stopBit, bit7, incWord, tx_bit, clrTimer, clrWord: std_logic;

   --bit counter and bit selection requirements:
   signal bit_sel: std_logic_vector(2 downto 0);
   signal bit_sel_next: unsigned(2 downto 0);
   signal bit_count_reg: unsigned(11 downto 0);
   signal bit_count_next: unsigned(11 downto 0);

   --serial output:
   signal serial_out_next: std_logic;

begin
--------Execution of Bit Timer(Baud-rate = 2603)--------
   process(clk,rst)
   begin
      if(rst = '1') then                    --if reset is on
          bit_count_reg <= (others => '0'); --clear bit count register
      elsif(clk'event and clk = '1') then
          bit_count_reg <= bit_count_next;  --otherwise, update
                                            --bit count register
      end if;
   end process;

   --Bit Timer logic:
   --Until bit_count_next reaches 2603, keep incrementing.
   --When bit_count_next = 2603, reset it to 0 and change tx_bit to 1.
   process(bit_count_reg, clrTimer)
   begin
      tx_bit <= '0';
      if(clrTimer = '1') then  --
          bit_count_next <= (others => '0');
      --binary(2603) -->  101000101011
      elsif(bit_count_reg = "101000101011") then
          bit_count_next <= (others => '0');
          tx_bit <= '1';
      else
          bit_count_next <= bit_count_reg + 1;
      end if;
   end process;

----------------Word Counter Execution----------------
   process(rst,clk)
   begin
```

```
        if (rst='1') then               --when reset is on
            bit_sel <= (others => '0'); --reset bit selector
        elsif (clk'event and clk = '1') then  --or, at rising edge of clock
            bit_sel <= std_logic_vector(bit_sel_next); --update bit selector
        end if;
    end process;

    --Bit Selector Logic:
        --Using the 3-bit Bit Selector, the system determines
        --which data bit should be sent next.
        --Implementated as follows:
    process(bit_sel,clrWord,incWord)
    begin
        bit7 <= '0';
        bit_sel_next <= unsigned(bit_sel);
        if (clrWord = '1') then             --when clrWord = '1'
            bit_sel_next <= (others => '0'); --bit_sel_next updated to 0
        elsif (bit_sel = "111") then         --when bit_sel = 7
            bit7 <= '1'; --update bit7 to 1,
                         --which means data
                         --transmission is complete
        elsif (incWord = '1') then
            bit_sel_next <= unsigned(bit_sel) + 1; --or, when incWord is 1
                                                   --increment bit_sel_next
        else
            bit_sel_next <= unsigned(bit_sel);  --otherwise, assign bit_sel
                                                --to bit_sel_next
        end if;
    end process;

        --Using this implementation, we can count the number
        --of transmitted bits. When the current count (bit_sel)
        --equals 7, the FSM detects that the data transmission
        --is complete through the bit7 variable update.
        --



---------------Finite State Machine Execution---------------
    --we put together all the components we developed in
    --the FSM. The FSM reacts to the word counter and
    --bit timer to shift in between states and execute
    --the transmission through the UART protocol.
```

```
process(rst,clk)
begin
    if (rst='1') then
        s_reg <= IDLE;
    elsif (clk'event and clk='1') then
        s_reg <= s_next;
    end if;
end process;

process(s_reg, send_character, tx_bit, bit7)
begin
    incWord <= '0';
    startBit <= '0';
    stopBit <= '0';
    clrTimer <= '0';
    clrWord <= '0';
    tx_complete <= '0';
    s_next <= s_reg;
    case s_reg is
        when IDLE => --when in IDLE state, clear bit timer
                     --and update next state
            stopBit <= '1';
            clrWord <= '1';
            clrTimer <= '1';
            if (send_character='1') then
                s_next <= START;
            end if;
        when START => --when in START state, set startBit to 1
                      --and update next state
            startBit <= '1';
            if (tx_bit='1') then
                s_next <= RUN;
            else
                s_next <= START;
            end if;
        when RUN => --when in RUN state
                    --update next state and incWord based on
                    --tx_bit and bit7
            if (tx_bit='1' and bit7='0') then
                s_next <= RUN;
                incWord <= '1';
            elsif (tx_bit='1' and bit7='1') then
                s_next <= STOP;
            elsif (tx_bit='0') then
               s_next <= RUN;
            end if;
         when STOP => --when in STOP state, set stopBit to 1
                    --then update next state based on tx_bit
            stopBit <= '1';
            if (tx_bit='1') then
               s_next <= RETRN;
            else
               s_next <= STOP;
            end if;
         when RETRN =>   --when in RETRN state, transmission is
                      --complete. This is indicated by setting
                      --tx_complete and stopBit to 1.
            stopBit <= '1';
            tx_complete <= '1';
            if (send_character='1') then
               s_next <= RETRN;
            else
               s_next <= IDLE;
            end if;
         when others =>
            s_next <= IDLE;
      end case;
   end process;

---------------Serial Generator Execution---------------
   --The output of the combinational logic needs to be
   --tracked and registered in order to make sure that
   --a clean serial output is transmitted. This is
   --accomplished by executing a Serial Generator.

   process(rst,clk)
   begin
      if (clk'event and clk='1') then
         serial_out <= serial_out_next;
      end if;
   end process;

   serial_out_next <= '0' when (startBit='1') else
                 '1' when (stopBit='1') else
                 data_in(to_integer(unsigned(bit_sel)));
   --serial_out_next is low when startBit is high
```

```
    --serial_out_next is high when stopBit is high
    --otherwise, serial_out is a bit from the input

    --using this implementation we execute the boundary
    --protocols for start and stop of transmission, through
    --which the devices communicate with each other
    --regarding the beginning and end of transmission

    --and

    --also manage to find a way to send the data
    --without manipulating the original input.

END tx_BEHAVIORAL;
```

Constraint file:

tx.ucf

```
NET "switches<0>" LOC="F12"| IOSTANDARD = LVTTL ;
NET "switches<1>" LOC="G12" | IOSTANDARD = LVTTL ;
NET "switches<2>" LOC="H14" | IOSTANDARD = LVTTL ;
NET "switches<3>" LOC="H13" | IOSTANDARD = LVTTL ;
NET "switches<4>" LOC="J14" | IOSTANDARD = LVTTL ;
NET "switches<5>" LOC="J13" | IOSTANDARD = LVTTL ;
NET "switches<6>" LOC="K14" | IOSTANDARD = LVTTL ;
NET "switches<7>" LOC="K13"| IOSTANDARD = LVTTL ;


NET "button" LOC="M13"| IOSTANDARD = LVTTL ;
NET "clk" LOC="T9"| IOSTANDARD = LVTTL ;


NET "rst" LOC="L14"| IOSTANDARD = LVTTL ;
NET "done" LOC="P11"| IOSTANDARD = LVCMOS33 ;
NET "txd" LOC="R13"| IOSTANDARD = LVTTL ;
```

Baud Rate Calculation

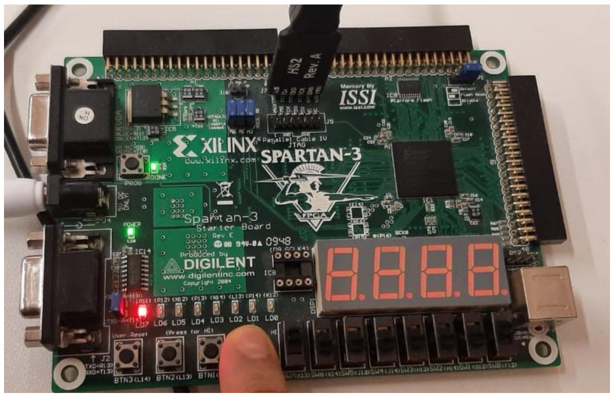
Everything id clocked on the global clock, which has a frequency of 50MHz. However, the Bit Timer needs to create a timing pulse of 19,200 Hz – this is out Baud rate. Therefore, we have to divide by the following factor:

Factor = 50,000,000/19,200 = 2604.1666

I have used a factor of 2063.

Pictures of FPGA Test

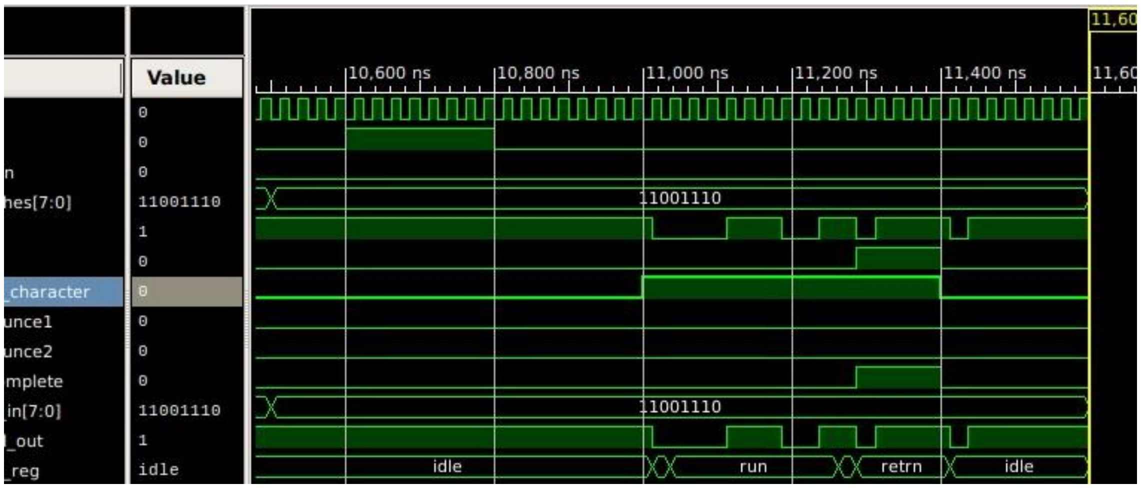
The following pictures were taken after downloading the code onto the board.



We could not run further tests because we were unable to obtain a serial cable on account of the lab being closed.

Simulation Pictures

The simulation for the handshake is provided below:



Conclusion

In this lab, we learned how to implement a UART Transmitter on an FPGA board. We learned how to engineer a UART Transmitter based on design specifications. We also learned how to manipulate Baud Rate using a Bit Timer, and debug the simulation in VHDL. We coded a Serial Generator, the Bit Timer mentioned before and a Word Counter, and connected them together into a Finite State Machine in order to obtain the UART Transmitter. Furthermore, we learned how to separately build components of the circuit and link them together in Xilinx. Using multi-segment programming style also helped us segment and organize our code better, and made debugging the code easier. Building the UART Transmitter provided us with an understanding of how development of more complex logic circuits work. We learned how to build and program different components of the circuit and then put them all together to obtain a complete circuit that accomplishes a specific task.

References:

https://www.circuitbasics.com/basics-uart-communication/

http://fpga-fhu.user.jacobs-university.de/wp-content/uploads/2014/10/uartNotes.pdf

http://fpga-fhu.user.jacobs-university.de/?page_id=38

https://www.codrey.com/embedded-systems/uart-serial-communication-rs232/