Report on LAB 3 : UART

(Topics in Adv Comp Eng: Embedded Systems Hardware EE 493)

NAME: Prince K. Bose
NET ID: pkb44
RUID: 186008149

DATE: 3/28/19

Purpose

The main purpose of this lab was to understand the basics of Finite State Machines and its use in Universal Asynchronous Rx Tx. Finite State Machines are a crucial part of any processor/ embedded circuit design. FSM can be used to define and program different states that a circuit/ system might go through. You can switch between the states on the basis of some pre defined inputs.

There are 4 main components to the circuit of Lab 3:
1. Clock Divider
2. Debounce Switch (Reset button, Trigger to send)
3. UART
4. Sender

PRE LAB

For UART transmission, we use the following waveform to design the FSM.

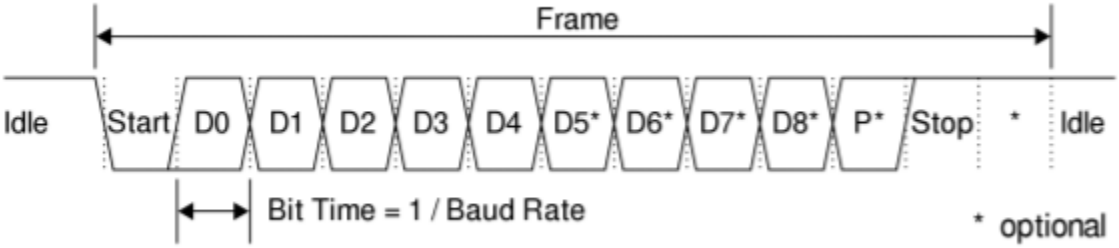

Fig. 1: UART WAVEFORM

Hence, there are 3 states, IDLE (no transmission happens here), START (data fetching begins), DATA (Actual transmission takes place till the data is transmitted, then goes back to IDLE)

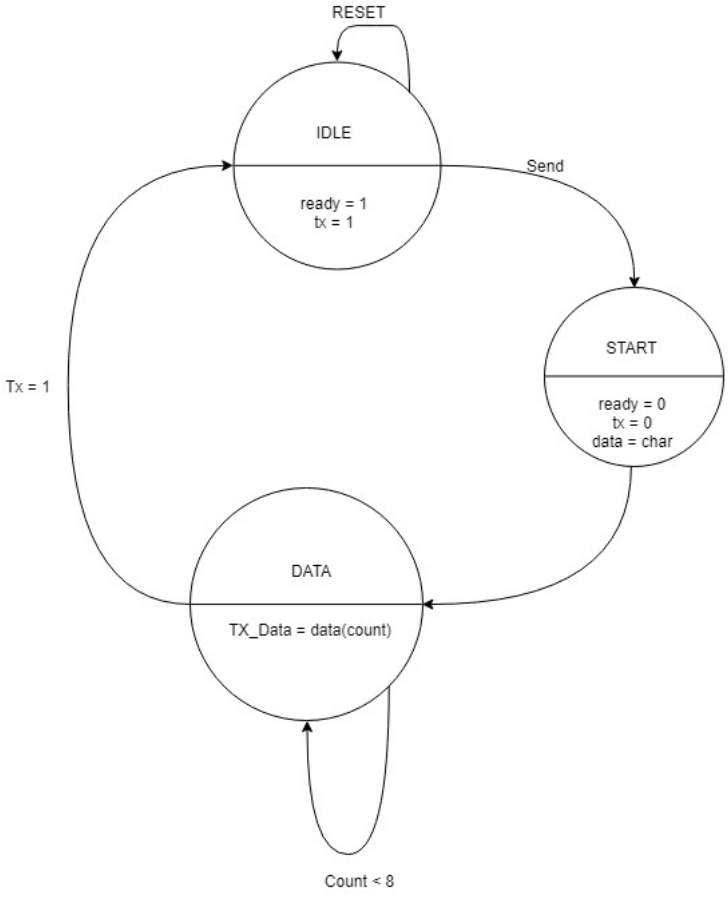

Fig. 2: Pre-Lab – FSM for UART

1. Clock Divider

The Zybo board has an inbuilt clock frequency of 125Mhz which has a time period of 8 nanoseconds. As we know, a clock signal is the heart of all FPGA systems and all subsystems on a synchronous circuit rely heavily on having the exact same clock signal as it helps the subsystems synchronize and communicate efficiently.

It is impossible for the human eye to notice an LED blinking at the rate of 125Mhz. Imagine an LED remaining ON for 4 ns and OFF for 4 ns. That means 250000000 switches between ON and OFF state per second. You cannot notice that without a Super Slomo camera.

In order to tackle this problem we have circuits called clock divider circuits, that essentially derive lower frequency clocks from higher frequency Master clocks. This is done using counters. We keep adding a 1 to a count register till a specific value, then we switch the output clock. This helps us derive a clock that has a greater clock cycle duration than the original clock.

In my clock divider code, I divide the clock frequency of the board (125 MHz) into 115200 (Baud rate) parts by counting up to 124999999/115200 before generating a pulse.

VHDL CODE

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

entity clock_div is
port (
  clk : in std_logic;
  div : out std_logic
);
end clock_div;

architecture clk_div of clock_div is
  signal count : std_logic_vector (25 downto 0) := (others => '0');
begin
  process(clk)
  begin
    if rising_edge(clk) then
        if(unsigned(count) < 124999999/115200) then
        count <= std_logic_vector( unsigned(count) + 1 );
        div<='0';
        else
        count <= (others => '0');
        div<= '1';
        end if;
    end if;
  end process;
end clk_div;
```

2. Debounce Switch

Due to being mechanical in nature, they are often the bane of existence for digital designers when not dealt with correctly. The problem lies in that mechanical nature. When the button is pushed or released the spring inside causes the contacts to behave like a damped oscillator, which creates spikes in the signal.

This means, in between a swap from ‘0’ to ‘1’ or LOW to HIGH or ‘1’ to ‘0’ or HIGH to LOW, the switch goes through a few meta stable stages in between. During these meta stable stages, the output value may fluctuate.

It looks something like this :

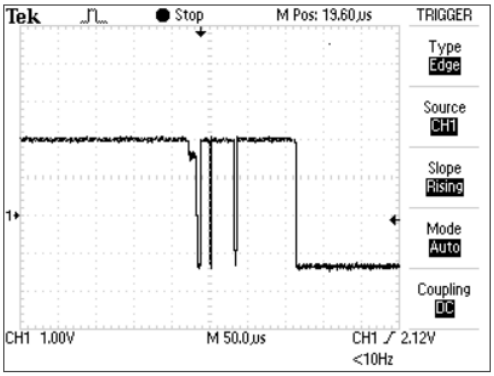

Fig 3. Switch Debouncing

DESIGN

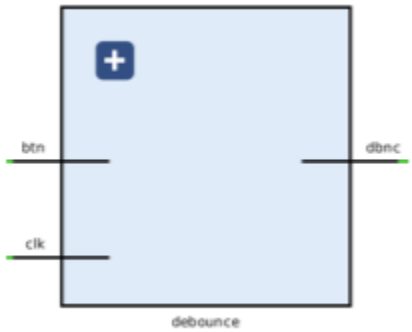

Fig. 4 Block Diagram of a Debounce Circuit

When a state changes from ‘0’ to ‘1’, the circuit waits for 20 ms for the switch to come to a stable state, before finally changing the output of the debounce at ‘1’.

In order to do this, we set the count value to 2499999.

VHDL Code

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


entity debounce is
    Port( clk : in STD_LOGIC;
          btn : in STD_LOGIC;
          dbnc : out STD_LOGIC);
end debounce;

architecture Behavioral of debounce is
signal sh : std_logic_vector(1 downto 0) := "00";
signal cnt_value : std_logic_vector(22 downto 0):=(others => '0');
begin
process(clk,btn)
begin

if (rising_edge(clk)) then
    sh(1) <= sh(0);
    sh(0) <= btn;
    if (unsigned(cnt_value) < 2499999) then
        dbnc <= '0';
        if (sh(1) = '1') then
            cnt_value <= std_logic_vector(unsigned(cnt_value)+1);
        else
            cnt_value <= (others => '0');
        end if;
    else
        dbnc <= '1';
        if(btn = '0') then
            dbnc <= '0';
            cnt_value <= (others => '0');
        end if;
    end if;
end if;
end process;
end Behavioral;
```

3. UART (Universal Asynchronous Receiver Transmitter)

UART stands for Universal Asynchronous Receiver/Transmitter. It's not a communication protocol like SPI and I2C, but a physical circuit in a microcontroller, or a stand-alone IC. A UART's main purpose is to transmit and receive serial data.
One of the best things about UART is that it only uses two wires to transmit data between devices.

UARTs transmit data *asynchronously*, which means there is no clock signal to synchronize the output of bits from the transmitting UART to the sampling of bits by the receiving UART. Instead of a clock signal, the transmitting UART adds start and stop bits to the data packet being transferred. These bits define the beginning and end of the data packet so the receiving UART knows when to start reading the bits.
When the receiving UART detects a start bit, it starts to read the incoming bits at a specific frequency known as the *baud rate.* Baud rate is a measure of the speed of data transfer, expressed in bits per second (bps). Both UARTs must operate at about the same baud rate.

VHDL Code - UART

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

entity uart is
   port (

   clk, en, send, rx, rst      : in std_logic;
   charSend                    : in std_logic_vector (7 downto 0);
   ready, tx, newChar          : out std_logic;
   charRec                     : out std_logic_vector (7 downto 0)

);
end uart;

architecture structural of uart is
   component uart_tx port
   (
      clk, en, send, rst  : in std_logic;
      char                : in std_logic_vector (7 downto 0);
      ready, tx           : out std_logic
   );
   end component;

   component uart_rx port
   (
```

```
        clk, en, rx, rst    : in std_logic;
        newChar             : out std_logic;
        char                : out std_logic_vector (7 downto 0)
    );
    end component;

begin

    r_x: uart_rx port map(
        clk => clk,
        en => en,
        rx => rx,
        rst => rst,
        newChar => newChar,
        char => charRec);

    t_x: uart_tx port map(
        clk => clk,
        en => en,
        send => send,
        rst => rst,
        char => charSend,
        ready => ready,
        tx => tx);

end structural;
```

VHDL Code – UART Tx

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

entity uart_tx is
    Port ( clk,en,send,rst : in STD_LOGIC;
           char : in STD_LOGIC_VECTOR (7 downto 0);
           ready,tx : out STD_LOGIC);
end uart_tx;

architecture Behavioral of uart_tx is
type state is (idle,start,data);
signal current_state : state := idle;
signal TempData  : std_logic_vector(7 downto 0) := X"00";
begin
process(clk)
variable count : natural := 0;
```

```
begin

   if(rising_edge (clk)) then
      if rst = '1' then
         TempData <= X"00";
         current_state <= idle;
      end if;

      if en = '1' then
         case current_state is

            when idle =>
               ready <= '1'; tx <= '1';
               if send = '1' then
                  current_state <= start;
               end if;

            when start =>
               ready <= '0'; tx <= '0';
               TempData <= char;
               count := 0;
               current_state <= data;

            when data =>
               if count < 8 then
                  tx <= TempData(count);
                  count := count + 1;
               else
                  tx <= '1';
                  current_state <= idle;
               end if;
            end case;
      end if;
   end if;
end process;

end Behavioral;
```

VHDL Code – UART Rx

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
entity uart_rx is
    port (
    clk, en, rx, rst    : in std_logic;
    newChar             : out std_logic;
    char                : out std_logic_vector (7 downto 0)
);
end uart_rx;
architecture fsm of uart_rx is
    -- state type enumeration and state variable
    type state is (idle, start, data);
    signal curr : state := idle;
    -- shift register to read data in
    signal d : std_logic_vector (7 downto 0) := (others => '0');
    -- counter for data state
    signal count : std_logic_vector(2 downto 0) := (others => '0');
    -- double flop rx plus 2 extra samples to take majority vote of 3
    -- majority vote of samples helps mitigate noise on line
    signal inshift : std_logic_vector(3 downto 0) := (others => '0');
    signal maj : std_logic := '0';
begin
    -- double flop input to fix potential metastability
    -- plus 2 extra samples to take majority vote of 3 inputs (oversampling)
    -- majority vote of samples helps mitigate noise on line
    process(clk) begin
        if rising_edge(clk) then
            inshift <= inshift(2 downto 0) & rx;
        end if;
    end process;

    -- majority vote of 3 samples (oversampling)
    -- majority vote of samples helps mitigate noise on line
    process(inshift)
    begin
        if (inshift(3) = '1' and inshift(2) = '1' and inshift(1) = '1') or
           (inshift(3) = '1' and inshift(2) = '1') or
           (inshift(2) = '1' and inshift(1) = '1') or
           (inshift(3) = '1' and inshift(1) = '1') then
               maj <= '1';
        else
            maj <= '0';
        end if;
```

```
    end process;
    -- FSM process (single process implementation)
    process(clk) begin
    if rising_edge(clk) then
        -- resets the state machine and its outputs
        if rst = '1' then
            curr <= idle;
            d <= (others => '0');
            count <= (others => '0');
            newChar <= '0';
        -- usual operation
        elsif en = '1' then
            case curr is
                when idle =>
                    newChar <= '0';
                    if maj = '0' then
                        curr <= start;
                    end if;
                when start =>
                    d <= maj & d(7 downto 1);
                    count <= (others => '0');
                    curr <= data;
                when data =>
                    if unsigned(count) < 7 then
                        d <= maj & d(7 downto 1);
                        count <= std_logic_vector(unsigned(count) + 1);
                    elsif maj <= '1' then
                        curr <= idle;
                        newChar <= '1';
                        char <= d;
                    else
                        curr <= idle;
                    end if;
                when others =>
                    curr <= idle;
            end case;
        end if;
    end if;
    end process;
end fsm;
```

4. Sender

Sender module actually communicates with the serial terminal of the PC. It declares and defines the netID and has 4 states: Idle, BusyA, BusyB, BusyC.

IDLE: If ready and trigger are 1, increment i, we extract a character (at position i) from the word and send it to BusyA.

BusyA: Go to BusyB always.

BusyB: Make send = 0, and go to state Busy C.

BusyC: Make ready 1 and trigger 0, and change the state to IDLE.

VHDL Code

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

entity sender is
    Port ( clk : in STD_LOGIC;
           clk_en : in STD_LOGIC;
           rst : in STD_LOGIC;
           trigger : in STD_LOGIC;
           ready: in STD_LOGIC;
           send:  out STD_LOGIC;
           char : out STD_LOGIC_VECTOR (7 downto 0));
end sender;

architecture Behavioral of sender is

type word is array (0 to 4) of STD_LOGIC_VECTOR(7 downto 0);
type state is (idle,busyA,busyB,busyC);
signal current_state : state := idle;
signal netID : word := (X"70",X"6b",X"62",X"34",X"34");
signal i : STD_LOGIC_VECTOR(2 downto 0) := "000";

begin

process(clk)
begin
    if (rising_edge(clk) and (clk_en = '1')) then
        if rst = '1' then
            send <= '0';
```

```
            char <= X"00";
            i <= "000";
            current_state <= idle;
        end if;
        case current_state is
            when idle =>
                if ready ='1' and trigger = '1' then
                    if unsigned(i) < 5 then
                        send <= '1';
                        char <= netID(natural(to_integer(unsigned(i))));
                        i <= STD_LOGIC_VECTOR(unsigned(i)+1);
                        current_state <= busyA;
                    else
                        i <= "000";
                    end if;
                end if;
            when busyA =>
                current_state <= busyB;
            when busyB =>
                send <= '0';
                current_state <= busyC;
            when busyC =>
                if ready ='1' and trigger = '0' then
                    current_state <= idle;
                end if;
        end case;
    end if;
end process;

end Behavioral;
```

5. Sender Top

Sender Top just brings all the above modules together by creating one instance for each and then just port mapping to complete the wiring.

VHDL Code

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

entity sender_TOP is
    Port ( TXD : in STD_LOGIC;
           btn : in STD_LOGIC_VECTOR (1 downto 0);
           clk : in STD_LOGIC;
           RXD : out STD_LOGIC;
           CTS : out STD_LOGIC;
           RTS : out STD_LOGIC);
end sender_TOP;

architecture Behavioral of sender_TOP is

component uart
 port (

   clk, en, send, rx, rst      : in std_logic;
   charSend                  : in std_logic_vector (7 downto 0);
   ready, tx, newChar          : out std_logic;
   charRec                   : out std_logic_vector (7 downto 0)

);
end component;

component debounce
    Port( clk : in STD_LOGIC;
       btn : in STD_LOGIC;
       dbnc : out STD_LOGIC);
end component;

component clock_div
port (
  clk : in std_logic;
  div : out std_logic
);
end component;
```

```
component sender
    Port ( clk : in STD_LOGIC;
       clk_en : in STD_LOGIC;
       rst : in STD_LOGIC;
       trigger : in STD_LOGIC;
       ready: in STD_LOGIC;
       send:  out STD_LOGIC;
       char : out STD_LOGIC_VECTOR (7 downto 0));
end component;
signal RESET_BTN, Trigger : STD_LOGIC;
signal div,SEND,READY : STD_LOGIC;
signal char: STD_LOGIC_VECTOR(7 downto 0);
begin
rts<='0';
cts<='0';
Clock_Divider: clock_div
port map( clk => clk,
          div => div);
Debounce_Reset: debounce
port map(clk => clk,
       btn => btn(0),
       dbnc => RESET_BTN);
Debounce_Trigger: debounce
port map(clk => clk,
         btn => btn(1),
         dbnc => Trigger);
SenderDeclaration: sender
port map( clk => clk,
          trigger => Trigger,
          clk_en => div,
          ready => READY,
          rst => RESET_BTN,
          send => SEND,
          char => char);
uut: uart
port map(    clk=> clk,
             en => div,
             send => SEND,
             rx => TXD,
             rst => RESET_BTN,
             charSend => char,
             ready => READY,
             tx => RXD);

end Behavioral;
```

Elaborated Design of the entire circuit

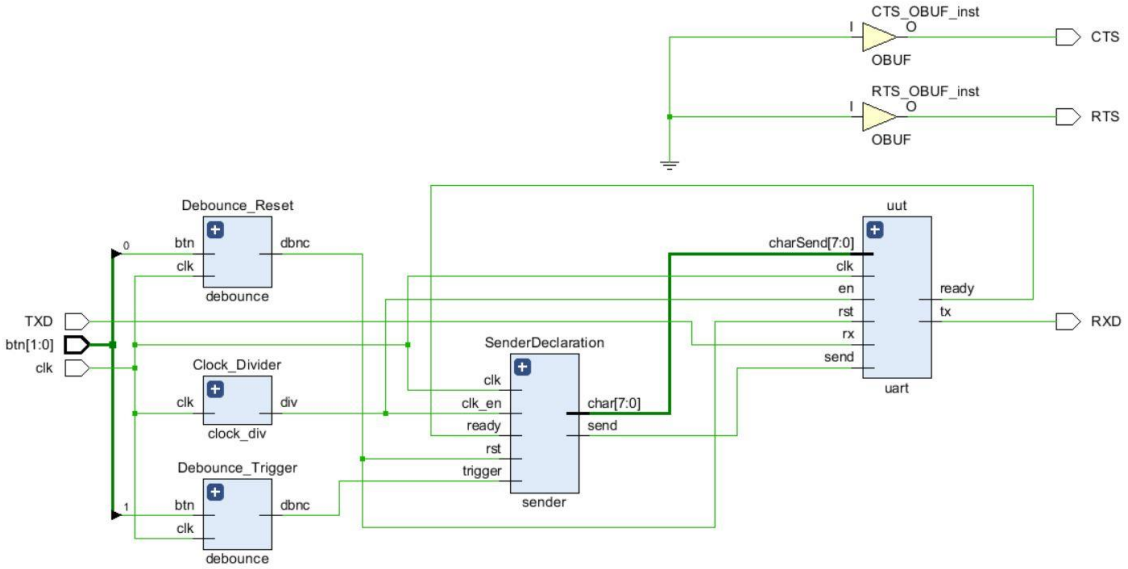

Fig 5: Elaborated Design

Post Synthesis Schematic

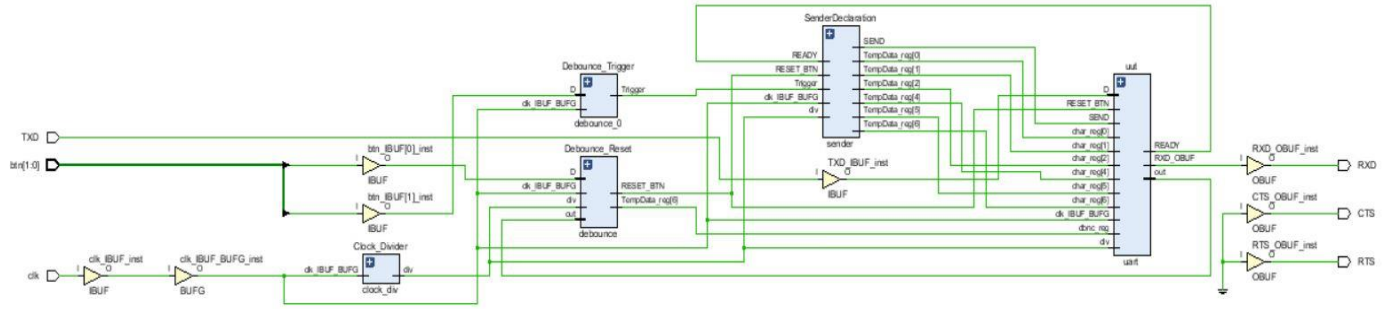

Fig 6: Post Synth Schematic

Implemented Netlist Design

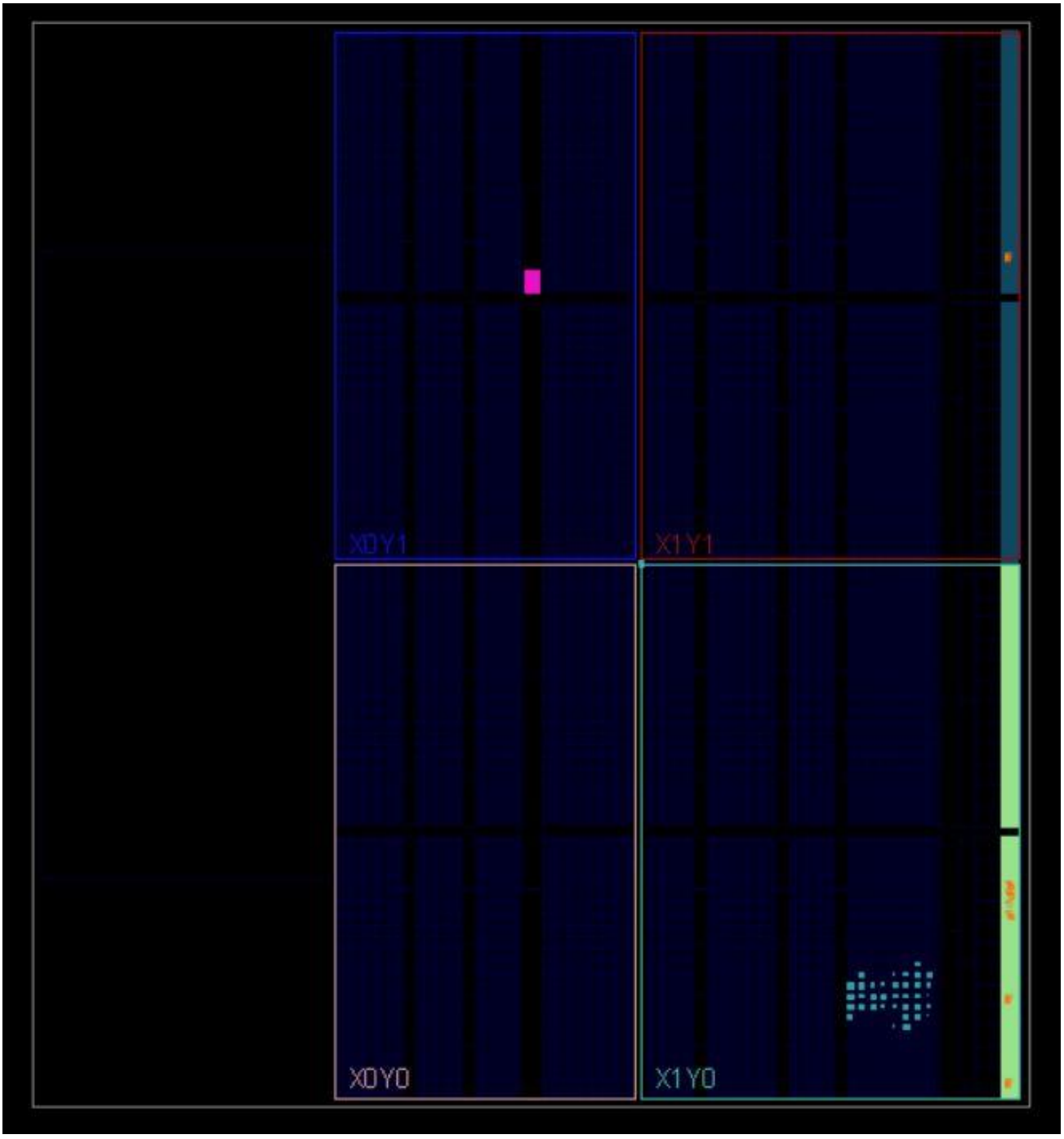

Fig 7: Netlist Design

Utilization Report

Name	Slice LUTs (17600)	Slice Registers (35200)	Slice (4400)	LUT as Logic (17600)	LUT Flip Flop Pairs (17600)	Bonded IOB (100)	BUFGCTRL (32)
N sender_TOP	62	88	36	62	27	6	1
Clock_Divider (clock_d...	10	12	4	10	7	0	0
Debounce_Reset (deb...	12	25	12	12	2	0	0
Debounce_Trigger (de...	11	25	11	11	2	0	0
SenderDeclaration (se...	16	12	6	16	8	0	0
uut (uart)	13	14	6	13	8	0	0

Fig 8: Netlist Design

Power Report

Summary

Power analysis from Implemented netlist. Activity derived from constraints files, simulation files or vectorless analysis.

Total On-Chip Power:	**0.095 W**
Design Power Budget:	**Not Specified**
Power Budget Margin:	**N/A**
Junction Temperature:	**26.1°C**
Thermal Margin:	58.9°C (5.0 W)
Effective ϑJA:	11.5°C/W
Power supplied to off-chip devices:	0 W
Confidence level:	Medium

Launch Power Constraint Advisor to find and fix invalid switching activity

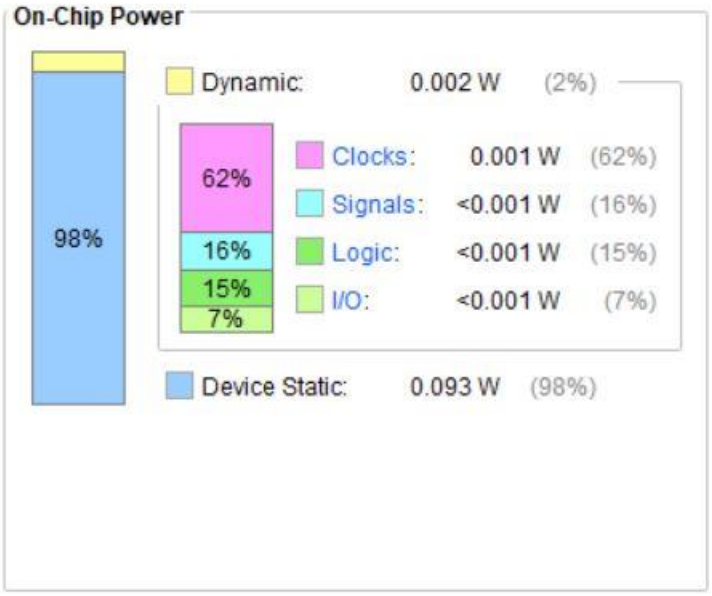

Fig 9: Netlist Design

Changes to the Zybo_Master.xdc Constraints File:

```
##Clock signal
set_property -dict { PACKAGE_PIN L16   IOSTANDARD LVCMOS33 } [get_ports { clk }]; #IO_L11P_T1_SRCC_35 Sch=sysclk
create_clock -add -name sys_clk_pin -period 8.00 -waveform {0 4} [get_ports { clk }];

###Buttons
set_property -dict { PACKAGE_PIN R18   IOSTANDARD LVCMOS33 } [get_ports { btn[0] }]; #IO_L20N_T3_34 Sch=BTN0
set_property -dict { PACKAGE_PIN P16   IOSTANDARD LVCMOS33 } [get_ports { btn[1] }]; #IO_L24N_T3_34 Sch=BTN1

###Pmod Header JB
set_property -dict { PACKAGE_PIN T20   IOSTANDARD LVCMOS33 } [get_ports { RTS }]; #IO_L15P_T2_DQS_34 Sch=JB1_p
set_property -dict { PACKAGE_PIN U20   IOSTANDARD LVCMOS33 } [get_ports { RXD }]; #IO_L15N_T2_DQS_34 Sch=JB1_N
set_property -dict { PACKAGE_PIN V20   IOSTANDARD LVCMOS33 } [get_ports { TXD }]; #IO_L16P_T2_34 Sch=JB2_P
set_property -dict { PACKAGE_PIN W20   IOSTANDARD LVCMOS33 } [get_ports { CTS }]; #IO_L16N_T2_34 Sch=JB2_N
```

Discussion

Final Output

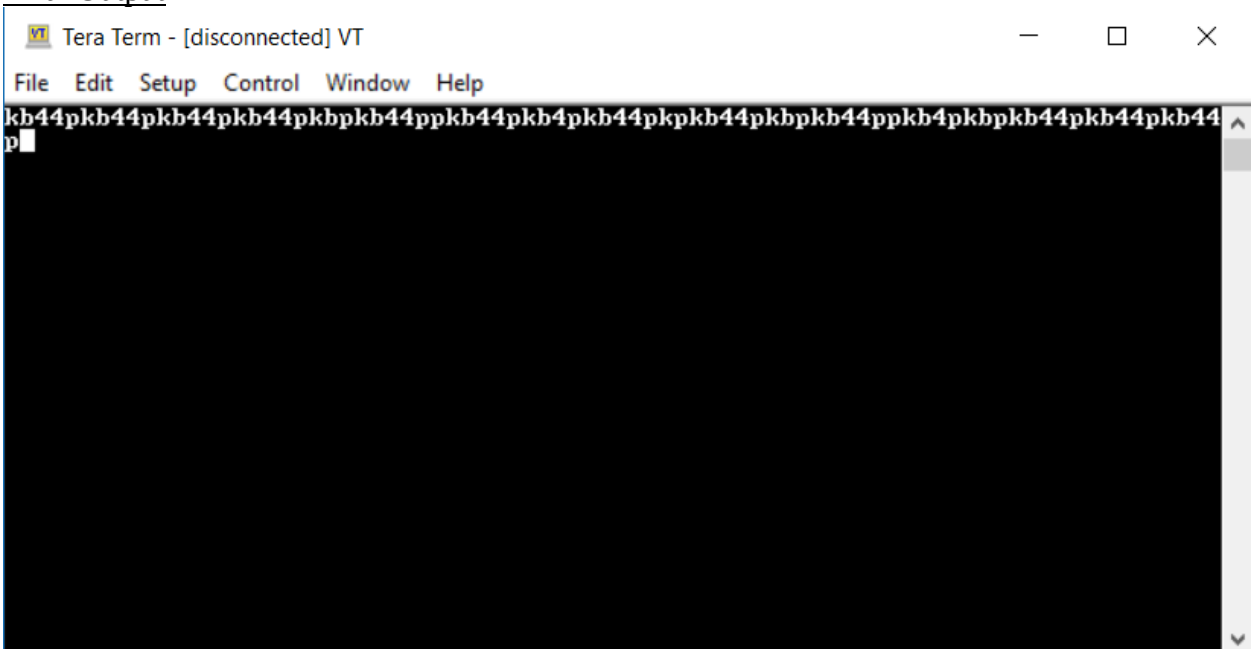

Fig. 10: Serial Monitor Output

Observations

1. In the XDC file, while using multiple buttons we use '[' to denote each bit. While in the VHDL source files, we use '('.
2. A conditional in the code is converted into a MUX.
3. Any register that stores a value is represented as a D Flip Flop.
4. All inputs and outputs have buffers.
5. Overall Power Utilization for my design is 0.095 Watts.
6. A total of 62 LUTs out of a 17600 are used for this design.
7. CTS and RTS were grounded for the UART communication.
8. A baud rate of 115200 was used for the UART, since there is no clock and it is asynchronous

Questions/ Follow Up:

Concepts Understood:

1. UART Tx and Rx
2. Buttons
3. Debouncing
4. Instantiation of components

Concepts Unsure of:

1. Why do we not debounce a switch when it is pulled down?
2. How is UART different from I2C and SPI protocols?