UART Simple

Author: Giorgi Solomnishvili

10/15/2020

Contents

Task Description

For Lab 6, we were asked to write Verilog modules for simple UART. Transmitter and Receiver were required to work in parallel. The program would be able to work in full duplex. We were instructed to use two pins Tx and Rx for transmitting and receiving a byte. Bits of the byte should be transmitted and received in series. Alongside with data-byte, the transmitter should send corresponding START_BIT, PARITY_BIT, STOP_BIT. START_BIT is 0 and STOP_BIT is 11. Overall, the transmitter should send 12 bits. Figure 1 shows a layout of the transmitted bits.

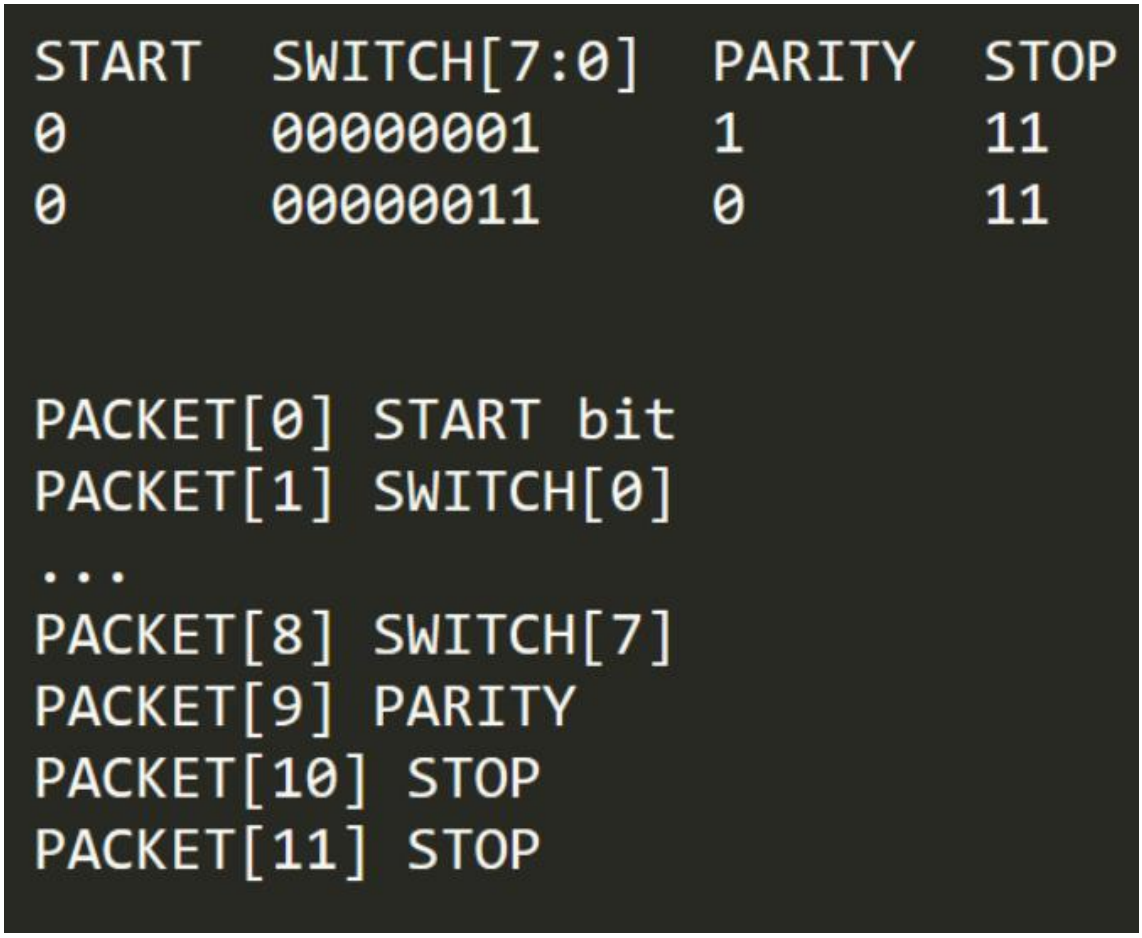


Figure 1

We were required to write separate modules for receiver, transmitter and clock divider. Transmitter and receiver should work on 9600 Hz clock which would enable us to have UART communication with 9600 baud rates. This 9600 Hz clock should be obtained by the clock divider module. The latter is parametrized module. The parameter helps us obtain a clock with desired frequency out of 50 MHz clock signal.

Moreover, we should use three push buttons. Key[0] should be a reset button. When key[1] is pressed, the "program" should take data from SWITCHES and store it in register. When Key[2] is pressed, the "program" should transmit data on TX output pin, with proper START_BIT, DATA_BYTE, PARITY_BIT, STOP_BIT. When the data is received on Rx pin, it should be displayed on LEDs.

Solution

In order to accomplish the given task, I wrote three modules – Transmitter, Receiver, Clock_divide, and instantiated them in a top module – UART. Those modules will be explained in detail.

UART

This is the top module. UART has four inputs – 50 MHz clock, three buttons ([2:0] KEY), eight switches ([7:0] SW), Rx pin, and three outputs – Tx pin, seven LEDs ([7:0] LED), check_parity. A data byte is entered using eight switches. KEY[0] represents a reset button. The press of KEY[1] causes the value of switches to get stored in transmitter register. The press of KEY[2] causes the value of transmitter register to be transmitted. Check_parity signal is 1 if the error is not detected.

We were instructed to make transmitter and receiver parametrized modules, so that we would be able to specify baud rate of transmitter/receiver separately with parameter. UART module has two parameters - counter_ceil_tr and counter_ceil_rec. The former is used to specify baud rate for transmitter and the latter – for receiver.

I have four instantiations in UART module. Module clock_divider is instantiated twice. The first instantiation creates clock for transmitter based on specified baud rate for transmitter. The second instantiation creates clock for receiver based on specified baud rate for receiver. In our case, the baud rate is 9600 which means that 9600 bits can be receiver or sent in a second. See figure 2. The values for counter_ceil_tr/rev are going to be explained later.

```
parameter integer counter_ceil_tr = 2604;
parameter integer counter_ceil_rec = 2604;

clock_divider #(counter_ceil_tr) clock_divider_for_transmitter(
   .clk_in(CLOCK_125_p),.rst(KEY[0]),.clk_out(clock_for_Transmitter)
);

clock_divider #(counter_ceil_rec) clock_divider_for_receiver(
   .clk_in(CLOCK_125_p),.rst(KEY[0]),.clk_out(clock_for_Receiver)
);
```

Figure 2

The third instance is an instance of module called transmitter. This module has three inputs – clock, KEYs, switches, and one output – Tx pin. See Figure 3. Transmitter module is going to be explained later.

```
Transmitter Transmitter00(
   .CLOCK_125_p(clock_for_Transmitter),
   .KEY(KEY),.Tx(Tx),.SW(SW));
```

Figure 3

The final instance is an instance of module called receiver. This module has two inputs – clock, Rx pin, and two output – LEDR and check_parity. See Figure 4. Receiver module is going to be explained in later

```
Receiver Receiver00(
    .CLOCK_125_p(clock_for_Receiver),
    .Rx(Rx),.LEDR(LEDR),.check_parity(check_parity)
);
```

Figure 4

CLOCK_DIVIDER

This module changes frequency of the input clock. The module has a parameter counter_ceil. The module decreases the frequency of the input clock by 2*(1+ counter_ceil). The module has a reset button that resets counter and the output clock to 0. At positive edge of the input clock, the counter is incremented. Whenever the counter reaches counter_ceil, the counter becomes 0 and the output clock toggles. This generates a clock that has 2*(1+ counter_ceil) times more period than the input clock. See Figure 5.

```
always @(posedge clk_in, negedge rst) begin
if(~rst) begin
   counter <= 0;
   clock <= 0;
end
else begin
   counter <= counter + 1;
   if(counter==counter_ceil) begin
      counter <= 0;
      clock <= ~clock;
   end
end
end

endmodule
```

Figure 5

In our case, the input clock has 50 MHz frequency. I want to decrease it to 9600 Hz. Therefore, I need to reduce frequency by 50MHz/9600 which is approximately 5208. If I want to decrease the clock frequency by 5208, the counter_ceil should satisfy the following equation:

$$5208 = 2 \cdot (1 + \ counter\_ceil)$$

Counter_ceil = 2603

Transmitter

The transmitter sends 12-bit long package. The 0^{th} bit is called the start bit. When the transmitter is not transmitting, the 0^{th} bit is 1. Whenever the 0^{th} bit becomes 0, the transmitter starts transmitting. [8:1] bits of the package correspond to data-byte. The 9^{th} bit is a parity bit. The parity bit is calculated by XORing the data-byte's bits with each other. If the data-byte has even number of 1s, the parity bit is 0. Otherwise, the parity bit is 1. The last two bits are called stop bits and they are set to 2'b11.

I have button press detection always blocks in Transmitter module. They detect if the KEY[1]/KEY[2]/Key[0] was pressed and generate an impulse. The press of KEY[0] clears data-byte part of the package, sets 0^{th}, 10^{th}, and 9^{th} bits to 1. Hence, KEY[0] resets data-byte and stops transmitting. See Figure 5.

```
always @(posedge CLOCK_125_p, negedge KEY[0]) begin
if(~KEY[0]) begin
Register_for_input_data[9:1] <= 0; // set parity bit and data byte to 0
Register_for_input_data[0] <= 1; // set start bit to 1 - do not send
Register_for_input_data[11:10] <= 2'b11; // set stop bits to 11
counter <= 0;
sent_data <= 1; // do not send anything
end
```

Figure 5

After KEY[1] is pressed, the value of the switch is stored in Register_for_input_data[8:1] and ^SW value – parity bit, is stored in Register_for_input_data[9]. See Figure 6

```
else if(check_btn1) begin // set data byte to switch value and parity bit to xor SW
Register_for_input_data[8:1] <= SW;
Register_for_input_data[9] <= ^SW;
end
```

Figure 6

Whenever KEY[2] is pressed, the 0^{th} bit of Register_for_input_data becomes 0, which means that the transmission is about to start. See figure 7

```
else if(check_btn2) begin
Register_for_input_data[0] <= 0;
Register_for_input_data[11:10] <= 2'b11;
end
else if(Register_for_input_data[0]==0) begin
sent_data <= Register_for_input_data[counter];
counter <= counter+1;
   if(counter==11)
   Register_for_input_data[0] <= 1;
end
else begin
counter <= 0;
```

Figure 7

Sent_data is assigned to the output Tx.

Receiver

The receiver module has two inputs – a clock and Rx pin, and two outputs – parity_check and LEDs. The module also has a counter which changes its value from 0 to 12. Whenever the package is not being received, the counter is 0 and the last two received bits on Rx pin are ones. Whenever the received bit becomes 0, the counter is 0, and the last two received bits are ones, the package is sent from the transmitter and a flag is set to 1 (a pulse is generated). See Figure 8

```
always @(posedge CLOCK_125_p) begin
tmp[0] <= Rx;
tmp[1] <= tmp[0];
end

assign check = (tmp==2'b11)&(Rx==0)&(counter==0);
```

Figure 8

Generation of the pulse called check causes receive_enable to become 1. Next, data-byte is received and stored into register called sent_data. Once the counter becomes 8, the whole data-byte has already been received and the recent arrive bit is a parity bit. Hence, whenever the counter is 8, I compare the received parity bit with ^sent_data. The result is stored in the output check_parity. If there is no error, check_parity is 1. See Figure 9.

```
always @(posedge CLOCK_125_p, posedge check) begin
if(check) begin
sent_data <= 0;
counter <=0;
receive_enable <= 1;
end
else if(receive_enable) begin
counter <= counter+1;
   if(counter<8)
      sent_data[counter] <= Rx;
   else if(counter==8)
      check_parity = ((^sent_data)==Rx);
   else begin
      counter <= 0;
      receive_enable <= 0;
   end
end
```

Figure 9

After the parity bit is checked, the check_parity and the received byte is outputted on LEDs.

Simulation & Verification

I wrote 4 different testbenches to check the correct functionality of my modules and the whole UART unit.

Testbench For Clock_Divider

I simulated an input clock with 2ps period. If I want to decrease frequency by 4, the value for counter_ceil should be 4/2 -1 = 1. Figure 10 proves that this is correct.

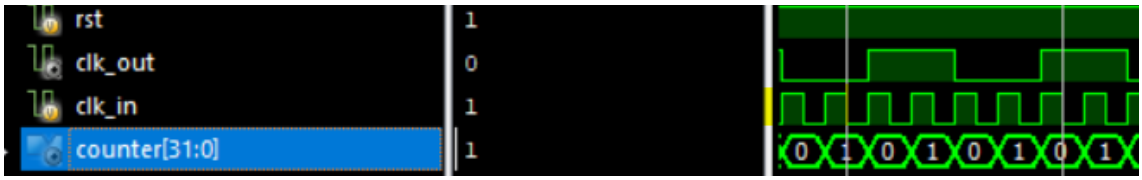


Figure 10

The clock_divider module works properly. Since I have 50 MHz input clock, I should decrease its frequency by 5208 times. Hence, the value for counter_ceil is 2603.

Testbench For Transmitter & Receiver

In this testbench, I instantiated transmitter and the receiver modules. Next, I connected Tx output of Transmitter to Rx input of the Receiver. See Figure 11

```
// Instantiate the Unit Under Test (UUT)
Transmitter uut (
    .CLOCK_125_p(CLOCK_125_p),
    .KEY(KEY),
    .Tx(Tx),
    .SW(SW)
);

// Outputs
wire [7:0] LEDR;
wire check_parity;

// Instantiate the Unit Under Test (UUT)
Receiver uut1 (
    .CLOCK_125_p(CLOCK_125_p),
    .LEDR(LEDR),
    .check_parity(check_parity),
    .Rx(Tx)
);
```

Figure 11

Next, I set switches to SW = 8'b 10110011; and run the simulation. Figure 12 shows that reset button – KEY[0], resets data-byte part of Register_for_input_data. The press of btn1 causes the value of switches to be stored in data-byte part of Register_for_input_data. And, the press of btn2 causes the 0^{th} bit of Register_for_input_data to become 0, which marks the beginning of data transmission.

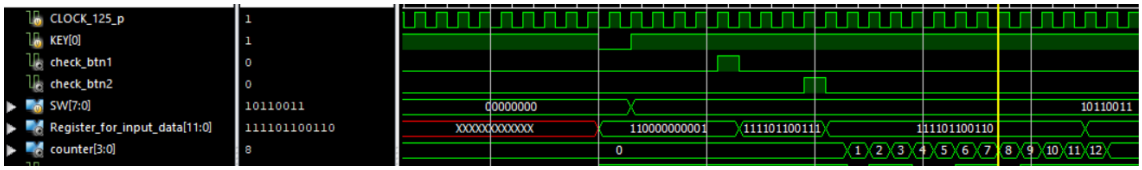


Figure 12

The transmitter should send 0, two 1s, two 0s, two 1s, 0, 11, and the stop bits 11. Figure 13 shows that those bits are correctly sent and received.

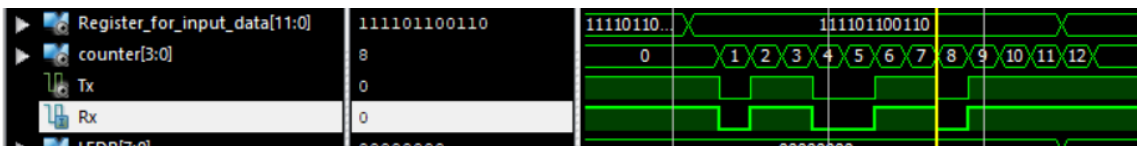


Figure 13

Figure 14 shows that after the last bit has been received, the data-byte and check_parity is outputted on LEDs.

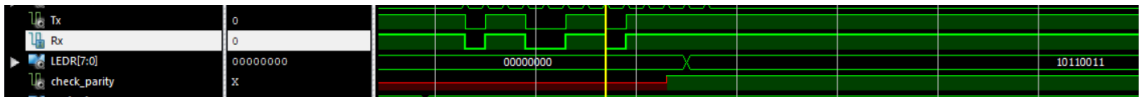


Figure 14

Testbench For UART

I instantiated module UART in testbench and I connected its output Tx to its input Rx. This is what we are going to do on FPGA. Figure 15 shows that the UART can send and receive bits at the same time

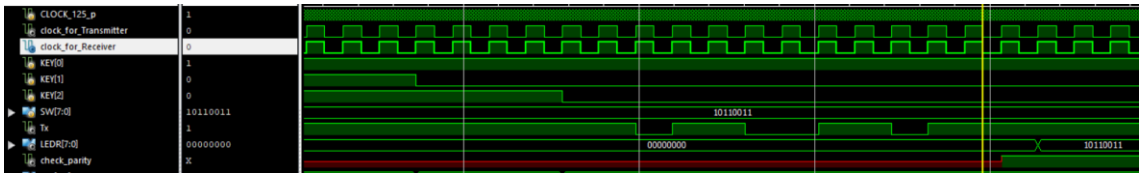


Figure 15

Moreover, we were instructed to write a single testbench module, instantiate two modules of UART, and connect Tx_1 to Rx_2 and Tx_2 to Rx_1. Hence, both instances of UART send and receives package at the same time. Figure 16 shows those instantiations.

```
    // Instantiate the Unit Under Test (UUT)
    UART #(1,1) uut_1 (
        .CLOCK_125_p(CLOCK_125_p),
        .KEY(KEY),
        .SW(SW_1),
        .Rx(Tx_2),
        .Tx(Tx_1),
        .LEDR(LEDR_1),
        .check_parity(check_parity_1)
    );

    UART #(1,1) uut_2 (
        .CLOCK_125_p(CLOCK_125_p),
        .KEY(KEY),
        .SW(SW_2),
        .Rx(Tx_1),
        .Tx(Tx_2),
        .LEDR(LEDR_2),
        .check_parity(check_parity_2)
    );
```

Figure 16

I set SW_1 and SW_2 to the following values:

```
SW_1 = 8'b00001111; // set data byte
SW_2 = 8'b11110000;
```

The figure 17 shows that UART_1 sends SW_1 and receives SW_2. Similarly, UART_2 sends SW_2 and receives SW_1.

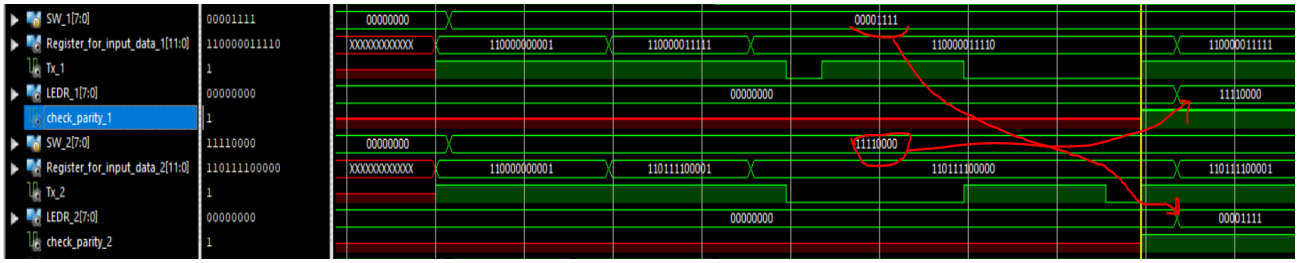


Figure 17

Comparison

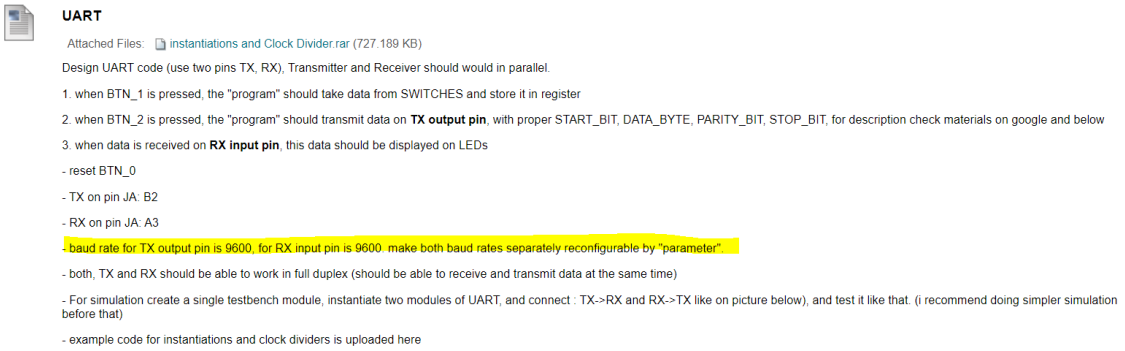
I used Xilinx synthesis tool to synthesize the UART and run simulations. The synthesis tool shows two warnings, which I believe needs to be ignored. The first warning says that: “Signal <Register_for_input_data> is never used or assigned. This unconnected signal will be trimmed during the optimization process.” Figure 18 shows signal <Register_for_input_data>

```
wire [11:0] Register_for_input_data;
assign Register_for_input_data = Transmitter00.Register_for_input_data;
```

Figure 18

I created that signal, so that it would be accessible in the testbench. I do not use Register_for_input_data in uart module. However, it is crucial for transmitter module, because that is the package transmitted by the module.

The second warning says that: “Unit <UART>: instances <clock_divider_for_transmitter>, <clock_divider_for_receiver> of unit <clock_divider> are equivalent, second instance is removed”. In our case those two instances are equivalent because they both produce clock with 9600 frequency. However, we were instructed to make both baud rates separately reconfigurable wit “parameter” (See Figure 19). Having two instances of clock_divide is necessary if I want to meet requirements.



UART

Attached Files: instantiations and Clock Divider.rar (727.189 KB)

Design UART code (use two pins TX, RX), Transmitter and Receiver should would in parallel.

1. when BTN_1 is pressed, the "program" should take data from SWITCHES and store it in register

2. when BTN_2 is pressed, the "program" should transmit data on **TX output pin**, with proper START_BIT, DATA_BYTE, PARITY_BIT, STOP_BIT, for description check materials on google and below

3. when data is received on **RX input pin**, this data should be displayed on LEDs

- reset BTN_0

- TX on pin JA: B2

- RX on pin JA: A3

- baud rate for TX output pin is 9600, for RX input pin is 9600. make both baud rates separately reconfigurable by "parameter".

- both, TX and RX should be able to work in full duplex (should be able to receive and transmit data at the same time)

- For simulation create a single testbench module, instantiate two modules of UART, and connect : TX->RX and RX->TX like on picture below), and test it like that. (i recommend doing simpler simulation before that)

- example code for instantiations and clock dividers is uploaded here

Figure 19

Overall, the synthesis report did not show any errors or important, non-negligible warnings.

Device utilization and Timing analysis summaries are the following:

For clock_divider

```
Device utilization summary:
---------------------------

Selected Device : 3s100ecp132-4

 Number of Slices:                        0  out of     960     0%
 Number of IOs:                           3
 Number of bonded IOBs:                   1  out of      83     1%

---------------------------
Partition Resource Summary:
---------------------------

  No Partitions were found in this design.
```

```
---------------
Speed Grade: -4

   Minimum period: No path found
   Minimum input arrival time before clock: No path found
   Maximum output required time after clock: No path found
   Maximum combinational path delay: No path found

Timing Detail:
--------------
All values displayed in nanoseconds (ns)

=========================================================


Total REAL time to Xst completion: 6.00 secs
Total CPU time to Xst completion: 6.59 secs

-->

Total memory usage is 4503176 kilobytes

Number of errors   :    0 (   0 filtered)
Number of warnings :    0 (   0 filtered)
Number of infos    :    0 (   0 filtered)
```

For Transmitter

```
Device utilization summary:
---------------------------

Selected Device : 3s100ecp132-4

 Number of Slices:                       14  out of    960     1%
 Number of Slice Flip Flops:             17  out of   1920     0%
 Number of 4 input LUTs:                 24  out of   1920     1%
 Number of IOs:                          13
 Number of bonded IOBs:                  13  out of     83    15%
 Number of GCLKs:                         1  out of     24     4%

---------------------------
Partition Resource Summary:
---------------------------

  No Partitions were found in this design.
```

```
Timing Summary:
---------------
Speed Grade: -4

   Minimum period: 4.360ns (Maximum Frequency: 229.358MHz)
   Minimum input arrival time before clock: 4.958ns
   Maximum output required time after clock: 4.283ns
   Maximum combinational path delay: No path found

Timing Detail:
--------------
All values displayed in nanoseconds (ns)
```

For Receiver

```
Device utilization summary:
---------------------------

Selected Device : 3s100ecp132-4

 Number of Slices:                       17  out of    960     1%
 Number of Slice Flip Flops:             16  out of   1920     0%
 Number of 4 input LUTs:                 30  out of   1920     1%
 Number of IOs:                          11
 Number of bonded IOBs:                  11  out of     83    13%
 Number of GCLKs:                         1  out of     24     4%

---------------------------
Partition Resource Summary:
---------------------------

  No Partitions were found in this design.

---------------------------

Timing Summary:
---------------
Speed Grade: -4

   Minimum period: 4.931ns (Maximum Frequency: 202.799MHz)
   Minimum input arrival time before clock: 4.221ns
   Maximum output required time after clock: 7.648ns
   Maximum combinational path delay: No path found

Timing Detail:
--------------
All values displayed in nanoseconds (ns)
```

For UART

```
Device utilization summary:
---------------------------

Selected Device : 3s100ecp132-4

 Number of Slices:                       61  out of    960     6%
 Number of Slice Flip Flops:             60  out of   1920     3%
 Number of 4 input LUTs:                113  out of   1920     5%
 Number of IOs:                          23
 Number of bonded IOBs:                  23  out of     83    27%
 Number of GCLKs:                         2  out of     24     8%

---------------------------
```

```
Timing Summary:
---------------
Speed Grade: -4

   Minimum period: 6.053ns (Maximum Frequency: 165.207MHz)
   Minimum input arrival time before clock: 4.793ns
   Maximum output required time after clock: 7.692ns
   Maximum combinational path delay: No path found

Timing Detail:
--------------
All values displayed in nanoseconds (ns)
```

Conclusion

This lab helped me better understand the way parallel to serial and serial to parallel transmission works. Moreover, I better understand how to design clock divider modules and how to instantiate separate modules in top module. The demonstration of my designs can be seen in the following link:

https://www.youtube.com/watch?v=-Qfc3vc7SL4&list=PL_5kwJbpj5rcMSO9ox0xO1xDytxqcwzdF&index=1

Video is divided into two parts.