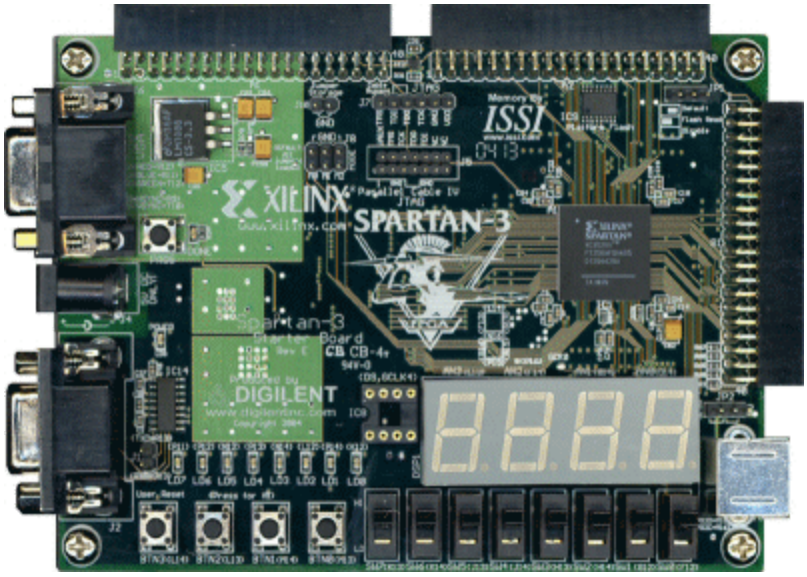
Τμήμα Ηλεκτρολόγων Μηχανικών & Μηχανικών Υπολογιστών
Πανεπιστήμιο Θεσσαλίας

CE430
DIGITAL CIRCUIT LAB



EXERCISE 2:
UART TRANSMITTER-RECEIVER SYSTEM

PATSIANOTAKIS CHARALAMPOS
2116

INTRODUCTION

In this exercise, a message on the 7 segment display monitor is going to get driven by the fpga. The exercise is separated in 4 parts:

- Part A: Decoding the binary value of digit, to driving values for the monitor.
- Part B: Displaying a static message on the monitor.
    - Changing the clock period
    - Filtering the reset signal (which comes from external environment.
    - Driving the AN and the outputs of decoder to monitor.
- Part C: Rotating the message, when a button is pushed.
    - Saving the message in an array of flip flops.
    - Filtering the change_message signal from external environment.
- Part D: Rotating the message with standard delay.
    - Make a FSM for this reason, generating the change_message signal.

All the parts are succesed.

PART A

Baud Controller

Implementation

First of all, the receiver and the transmitter have to agree in a baud rate to sample, in order not to lose data in connection. In other words, the order the transmitter sends the data and the receiver samples, must be known between the 2 parts. The baud rate is defined by the baud select which is set by higher level controller.

To control this, the baud_controller module is implemented. This module has inputs the clk, reset and baud_select signals and output the sample_enable signal. In order to activate the sample_enable signal in correct time, a counter is implemented inside, whose maximum value is relative to baud_select input.

To find the correct Maximum Counter Value first of all, the period of sampling (T_sc) must be known. Because the sample_enable signal must be activated 16 times faster than the protocol defines, the function of T_sc is *T_sc = 10^9 / (BR * 16)*. The T_sc value is the desired sampling period in ns.

After this the MCV (Maximum Counter Value) is easy to get found. As the clock period is 20 ns, the MCV value is the rounded to the nearest integer of *MCV = (T_sc / 20)*.

The error value is the absolute difference of desired T_sc an the real time of sample_enable period, the circuit implements. The Error value is calculated as *Error = abs (T_sc - 20*MCV)*.

Finally, the counter value is going to get saved in a register table. The size of the register table. The table's size is up integer of log2 of the maximum value between the MCV of baud_select values.

In the following table are the results of these calculations:

BS	BR	T_sc	MCV	Error
000	300	208333	10417	7
001	1200	52083	2604	3
010	4800	13021	651	1
011	9600	6510	326	10
100	19200	3255	163	5
101	38400	1628	81	8
110	57600	1085	54	5
111	115200	543	27	3

BS: Baud Select.

BR: Baud Rate (desired) (in bit/sec).

T_sc: Sampling Period (in ns).

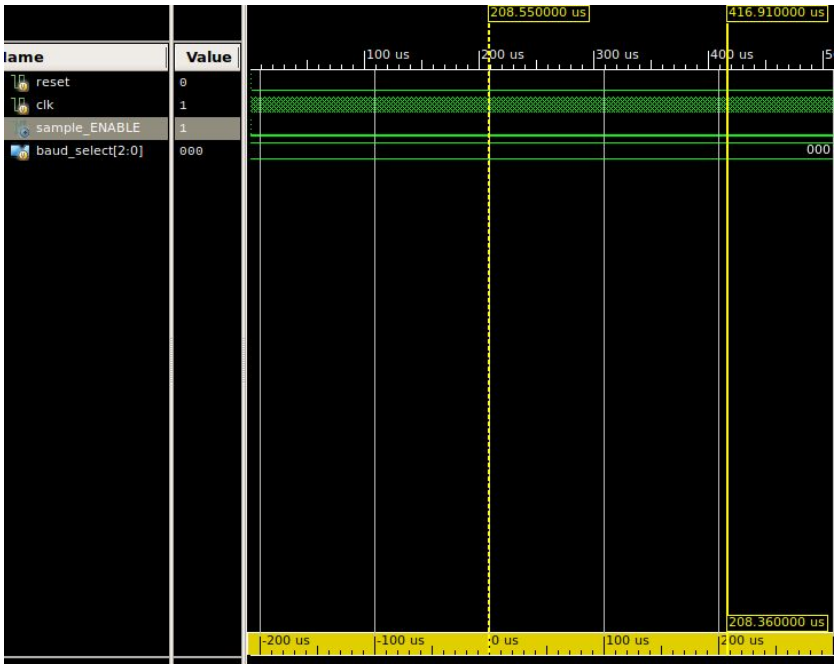
MCV: Max Counter Value.

Error: Time difference between real time of baud rate circuit implements and desired Baud Rate (in ns).

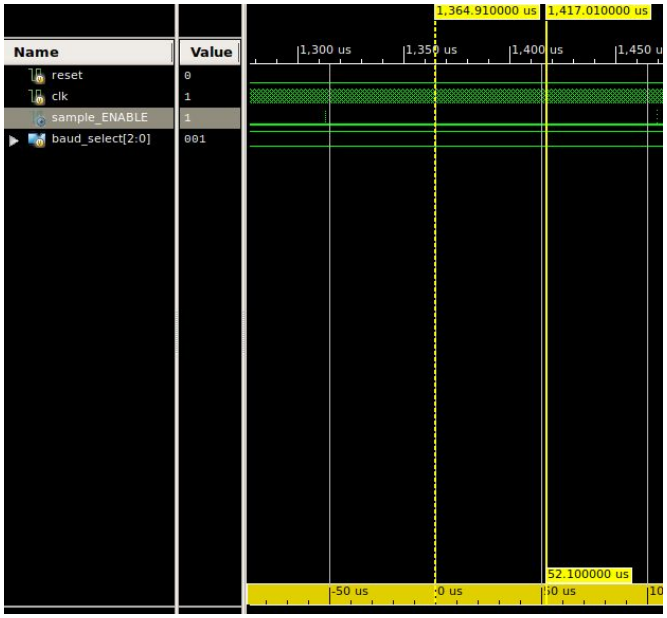
Verification

To verificate the functionality of this module, the time difference of sample_enable is checked for each baud_select mode.

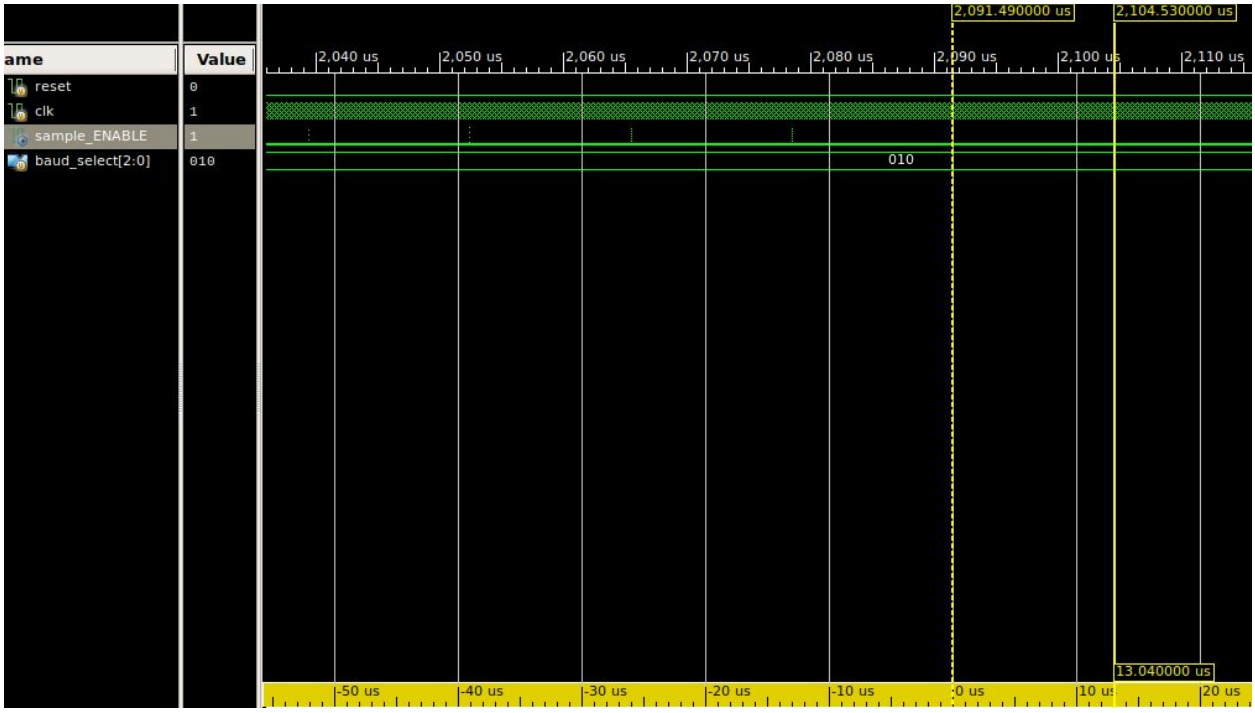
- *For baud_select 000:*



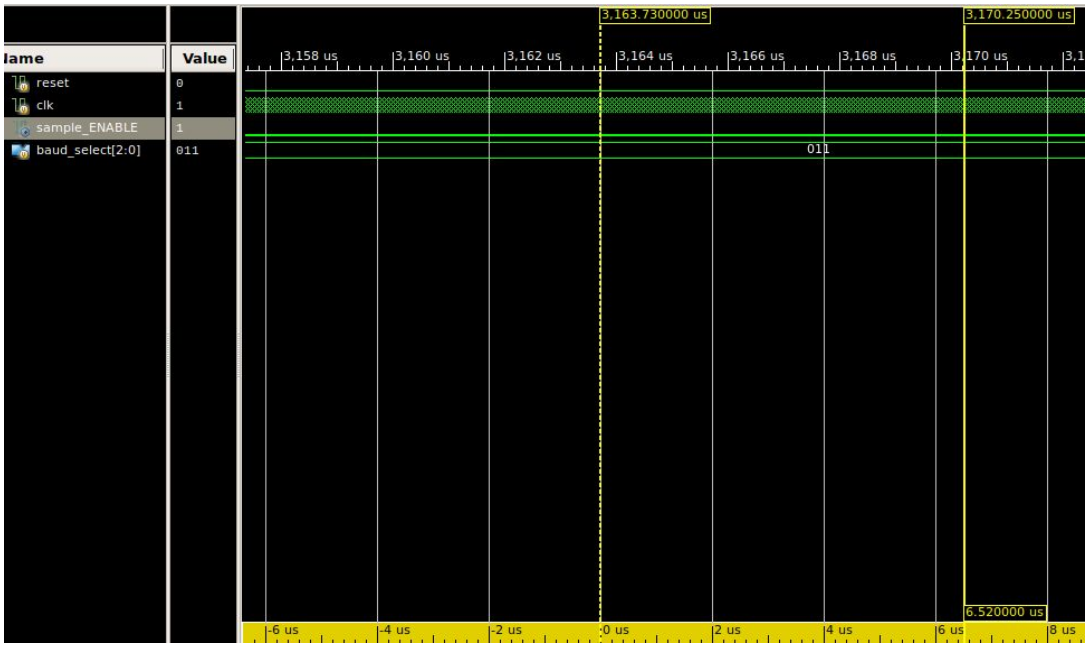
- *For baud_select 001:*



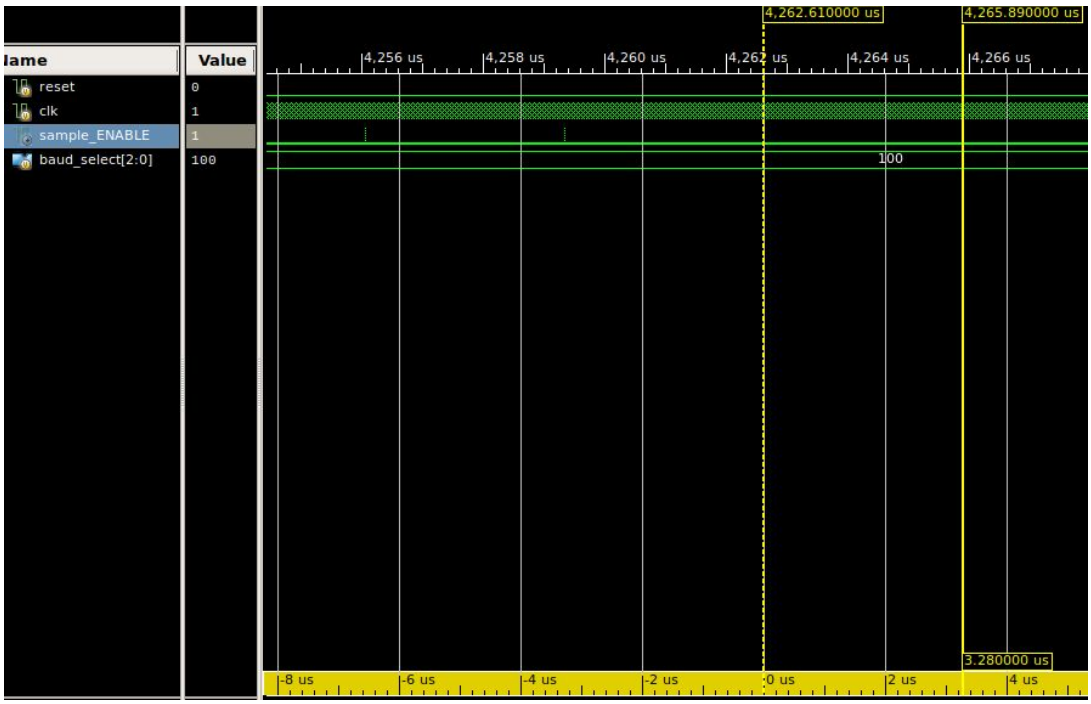
- *For baud_select 010:*



- *For baud_select 011:*

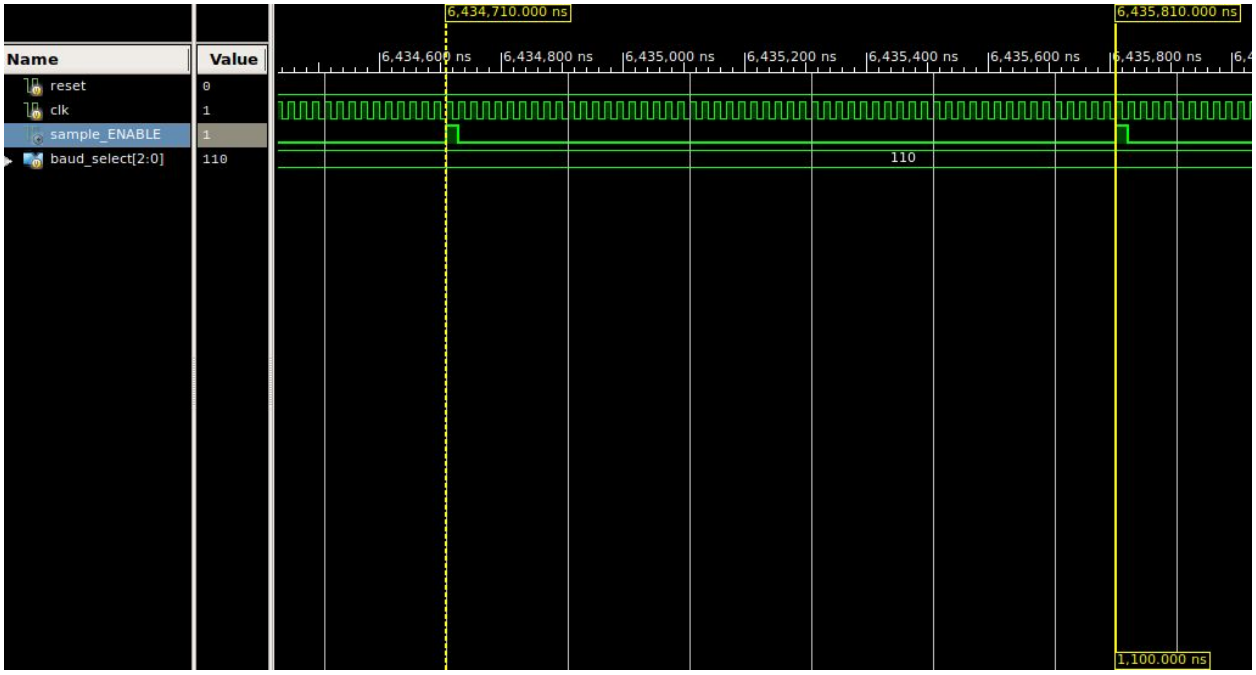


- *For baud_select 100:*

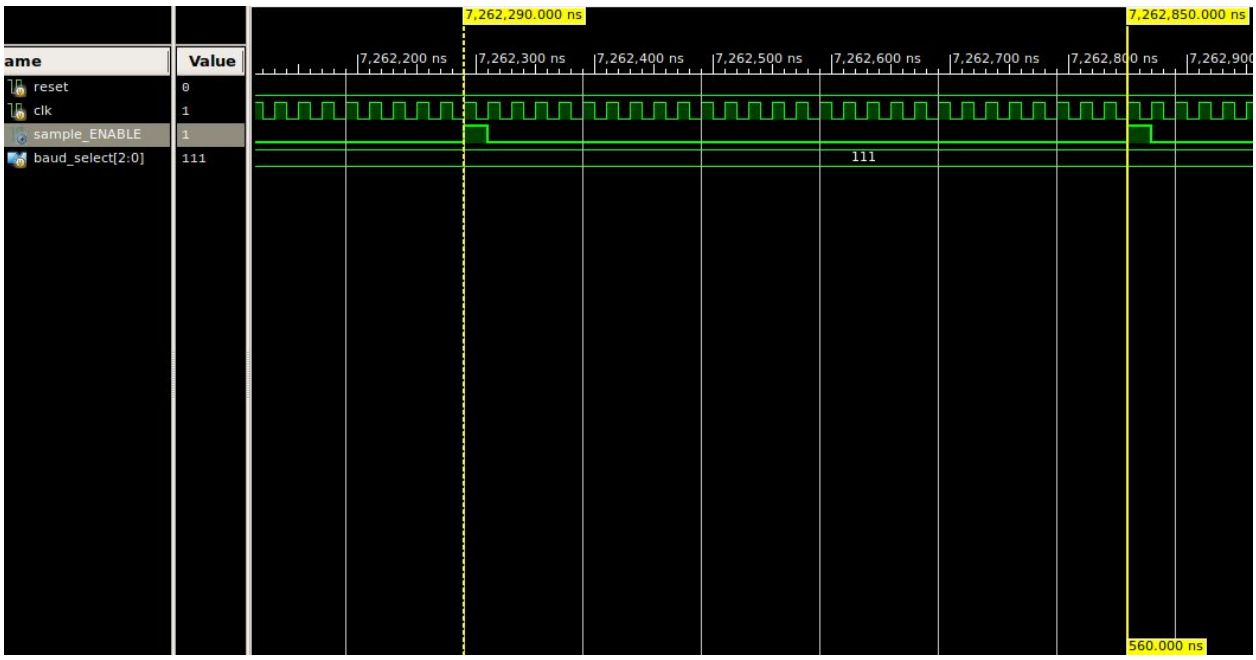


- *For baud_select 101:*

- *For baud_select 110:*



- *For baud_select 111:*



PART B

Uart Transmitter

Implementation

The uart transmitter is responsible to send the data from controller to the channel. It has to send a 11-bit message, which is the 8-bit data attached with a start bit, stop bit and parity bit (to control the validity of data).

The module has inputs (all from controller):

- Reset, clk
- 8-bit Tx_DATA
- The baud_select
- Tx_EN
- Tx_WR

And outputs:

- TxD (to channel)
- TxBUSY (to controller)

The module is implemented as an FSM machine. In order to work, the reset signal must be down and TxEN must be up. Otherwise the state is constantly IDLE, TxD at 1 and TxBUSY at 0. The FSM (always loop) is activated at each sample_send signal. The sample_send signal is the Tx_sample_send signal of baud_controller module, with 16 times longer period (In order the receiver is able to sample 16 times while transmitter sends one bit).

The states of the FSM are:

- **STATE_IDLE**: Waits for activation of sample_Tx_Wr signal. This signal is activated when Tx_WR signal is activated until the sample_send signal is activated (actually saves the activation of

Tx_WR, which duration is one clk cycle, until the always block is activated).
When the sample_Tx_Wr signal is activated, it makes Tx_BUSY output 1, calculates the parity bit (xor of the sum of Tx_DATA) and changes state to STATE_START_BIT.

- **STATE_START_BIT**: Sends a zero bit to TxD to activate receiver. At next sample_send activation, the state moves to STATE_D0.

- **STATE_D0**: Sends to TxD the value of Tx_DATA[0]. Makes next state STATE_D1.

- **STATE_D1**: Sends to TxD the value of Tx_DATA[1]. Makes next state STATE_D2.

- **STATE_D2**: Sends to TxD the value of Tx_DATA[2]. Makes next state STATE_D3.

- **STATE_D3**: Sends to TxD the value of Tx_DATA[3]. Makes next state STATE_D4.

- **STATE_D4**: Sends to TxD the value of Tx_DATA[4]. Makes next state STATE_D5.

- **STATE_D5**: Sends to TxD the value of Tx_DATA[5]. Makes next state STATE_D6.

- **STATE_D6**: Sends to TxD the value of Tx_DATA[6]. Makes next state STATE_D7.

- **STATE_D7**: Sends to TxD the value of Tx_DATA[7]. Makes next state STATE_PARITY_BIT.

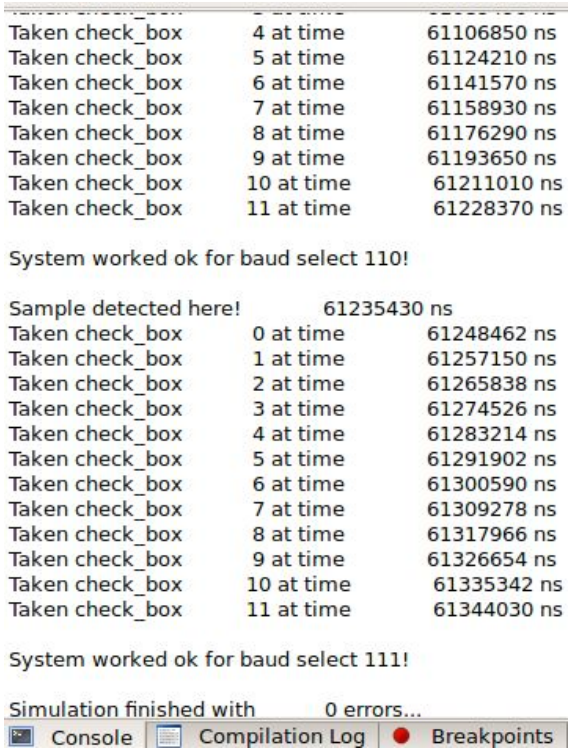
- **STATE_PARITY_BIT**: Sends to TxD the value of parity bit which was calculated in STATE_IDLE. Makes next state STATE_STOP_BIT.

- **STATE_STOP_BIT**: Sends to TxD the a bit with value 1 to conclude the message. Makes the Tx_BUSY bit 0, to let the controller to send again DATA. Returns to STATE_IDLE.

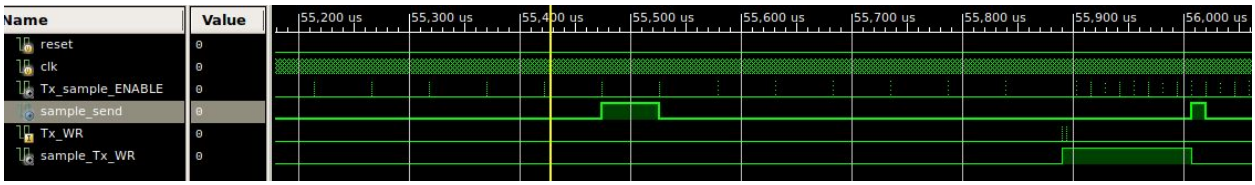
Verification

To verificate the transmitter, a testbench is created, which sends for each baud select the message 10101010. It samples itself the TxD wire and if it is false increases the errors value and shows the sum it samples and what should be to help in debugging.

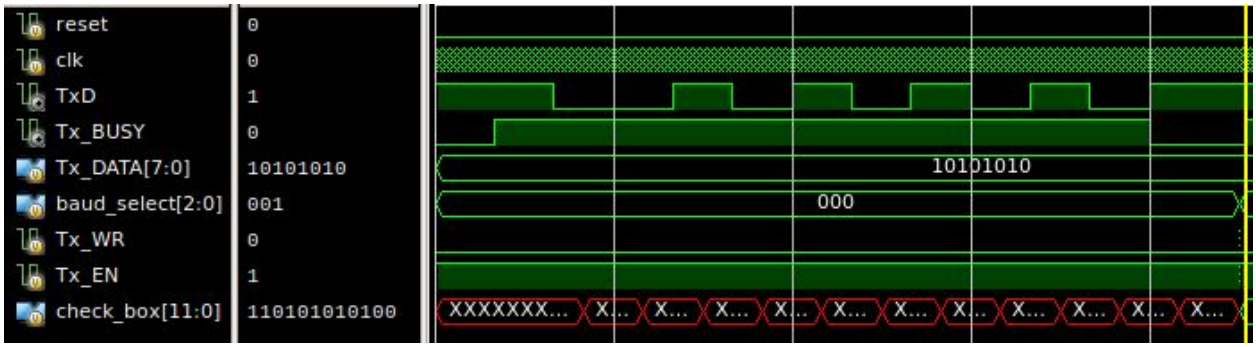
First of all a screenshot of ISE konsole to desribe how about the testbench samples and make decisions (No errors in this case, began from 43 the debugging):

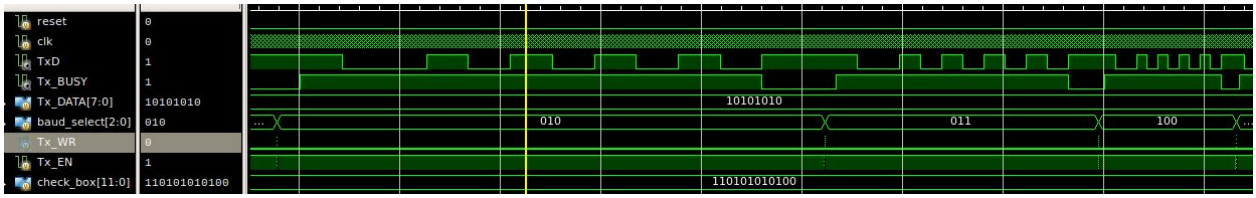


After, a waveform to approve the sample_enable and sample_Tx_Wr signals are working ok.



And two waveforms of TxD regarding to Tx_DATA, Tx_EN, Tx_WR and Tx_BUSY and check_box (of testbench). The waves are correct and the DATA atr the same constantly, so the value of check_box, does not change.





PART C

Uart Receiver

Implementation

The receiver is responsible to receive the data from channel and send them to the controller. It receives the 11 bit message, checks for the validation and sends the 8 bit data which the transmission worked for.

The module has inputs:

- Reset, clk (from controller)
- The baud_select (from controller)
- Rx_EN (from controller)
- Rx_D (from channel)

And outputs (all to controller):

- Rx_DATA table
- Tx_VALID, Tx_PERROR, Tx_FERROR (to assess the input message)

Receiver is also implemented as an FSM machine. It is activated each time the Rx_sample_ENABLE of baud_controller is enabled. In order to know at which part of the 16 parts of bit duration a counter is used.

If reset signal is up, state is constantly IDLE an VALID, PERROR, FERROR signals down. If Rx_EN is down the FSM gets frozen. Else the FSM works normally with the states:

- **STATE_IDLE**: Waits until the RxD signal get 0. If it gets, it changes to STATE_START_BIT.
- **STATE_START_BIT**: Waits until the value of counter get 7 (to middle of start bit sampling). If the value of RxD is still 0, the state changes to STATE_D0, changing the Rx_VALID, Rx_PERROR, Rx_FERROR

and counter values to zero. If the RxD value has changes to 1, the state returns to STATE_IDLE.

- **STATE_D0**: Waits until counter gets to value 15 (to middle of D0 sampling) and moves the value of RxD to Rx_DATA[0] and to parity_bit. Set next state STATE_D1.
- **STATE_D1**: Waits until counter gets to value 15 and moves the value of RxD to Rx_DATA[1]. Get the parity_bit value with the RxD current value to a xor gate and sets the result to parity_bit. Set next state STATE_D2.
- **STATE_D2**: Waits until counter gets to value 15 and moves the value of RxD to Rx_DATA[2]. Get the parity_bit value with the RxD current value to a xor gate and sets the result to parity_bit. Set next state STATE_D3.
- **STATE_D3**: Waits until counter gets to value 15 and moves the value of RxD to Rx_DATA[3]. Get the parity_bit value with the RxD current value to a xor gate and sets the result to parity_bit. Set next state STATE_D4.
- **STATE_D4**: Waits until counter gets to value 15 and moves the value of RxD to Rx_DATA[4]. Get the parity_bit value with the RxD current value to a xor gate and sets the result to parity_bit. Set next state STATE_D5.
- **STATE_D5**: Waits until counter gets to value 15 and moves the value of RxD to Rx_DATA[5]. Get the parity_bit value with the RxD current value to a xor gate and sets the result to parity_bit. Set next state STATE_D5.
- **STATE_D6**: Waits until counter gets to value 15 and moves the value of RxD to Rx_DATA[6]. Get the parity_bit value with the RxD current value to a xor gate and sets the result to parity_bit. Set next state STATE_D7.
- **STATE_D7**: Waits until counter gets to value 15 and moves the value of RxD to Rx_DATA[7]. Get the parity_bit value with the RxD current value to a xor gate and sets the result to parity_bit. Set next state STATE_PARITY_BIT.

- **STATE_PARITY_BIT**: Waits also for counter to get value of 15. After this it checks if the value of parity_bit is same of Rx_D. If not (which means a data changed during transmission) makes Rx_PERROR output 1. After this it gets to STATE_STOP_BIT.
- **STATE_STOP_BIT**: This bit should be the conclusion of transmission. If the value (after counter is 15) is 0, which means something got really wrong, the value of Rx_FERROR gets up. Else, if the Rx_PERROR value is zero, the Rx_VALID value gets up. After this it returns to STATE_IDLE.

Verification

To verificate all the cases of inputs, for each baud rate, the testbench sends 5 messages which should have the effects:

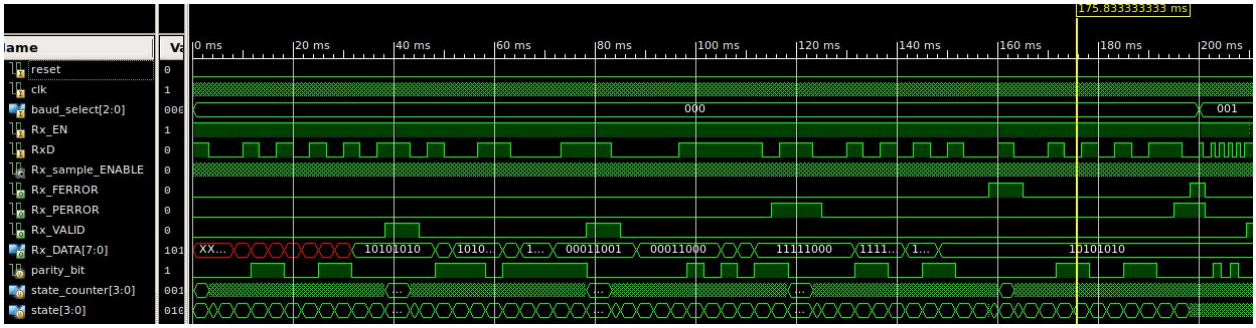
MESSAGE	Rx_VALID	Rx_PERROR	Rx_FERROR
10101010100	1	0	0
11000110010	1	0	0
10111110000	0	1	0
00101010100	0	0	1
01101010100	0	1	1

After this the testbench samples the Rx_DATA table and compares it with expected result. The screenshots are after bugs are fixed.

First of all, the konsole:

```
Console
Sent data          0 at time          274877586 ns
Sent data          1 at time          274886274 ns
Sent data          2 at time          274894962 ns
Sent data          3 at time          274903650 ns
Sent data          4 at time          274912338 ns
Sent data          5 at time          274921026 ns
Sent data          6 at time          274929714 ns
Sent data          7 at time          274938402 ns
Sent data          8 at time          274947090 ns
Sent data          9 at time          274955778 ns
Sent data         10 at time           274964466 ns
Sent data         11 at time           274973154 ns
Sent data          0 at time          274983882 ns
Sent data          1 at time          274992570 ns
Sent data          2 at time          275001258 ns
Sent data          3 at time          275009946 ns
Sent data          4 at time          275018634 ns
Sent data          5 at time          275027322 ns
Sent data          6 at time          275036010 ns
Sent data          7 at time          275044698 ns
Sent data          8 at time          275053386 ns
Sent data          9 at time          275062074 ns
Sent data         10 at time           275070762 ns
Sent data         11 at time           275079450 ns
Simulation finished with        0 errors...
```

And a look of major signals in case of baud select 000:

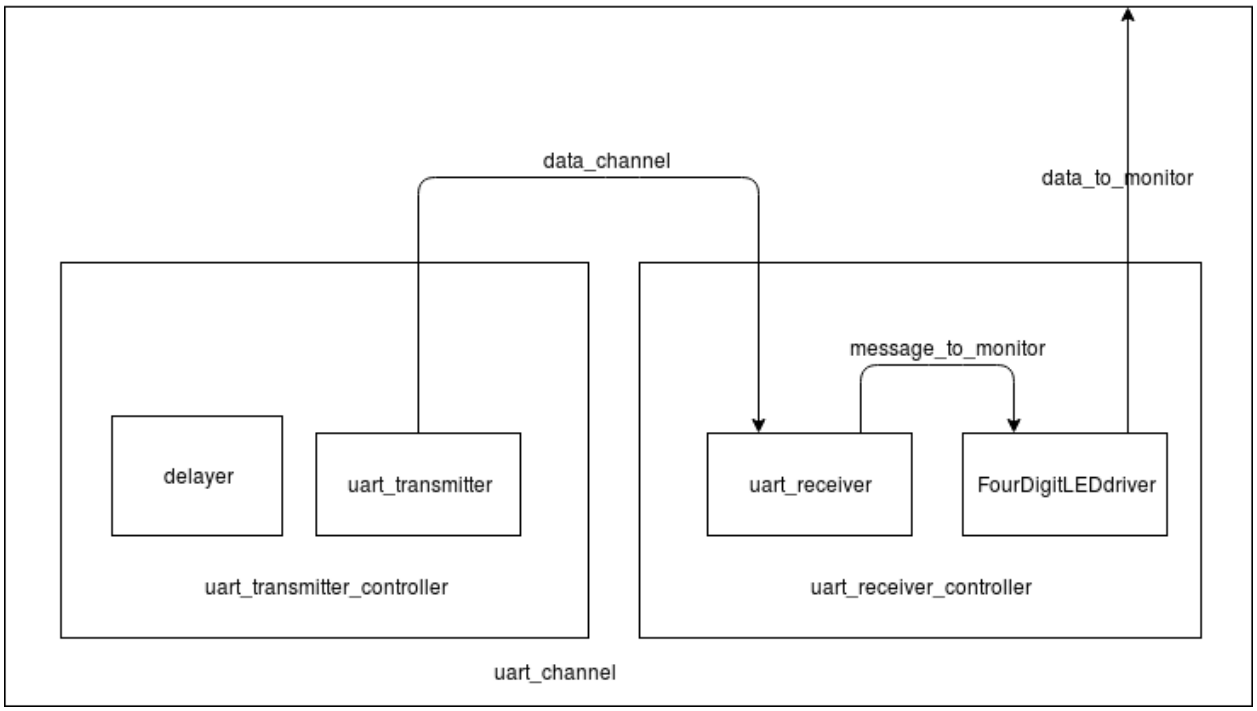

175.833333333 ms
0 ms
20 ms
40 ms
60 ms
80 ms
100 ms
120 ms
140 ms
160 ms
180 ms
200 ms
reset
clk
baud_select[2:0]
Rx_EN
RxD
Rx_sample_ENABLE
Rx_FERROR
Rx_PERROR
Rx_VALID
Rx_DATA[7:0]
parity_bit
state_counter[3:0]
state[3:0]
000
001
XX...
10101010
1010..
1...
00011001
00011000
11111000
1111..
1...
10101010

PART D

Both connection and test on FPGA

Implementation

First of all, in order to control the transmitter and receiver, a controller for each one was created. Then the controllers are instantiated in a top level module, which represents the channel.



Transmitter_controller:

The transmitter controller is responsible to drive the data to transmitter. The message is saved in a 16X8 memory of flip-flops which is instantiated when reset signal is activated. At each second (so that human eye can see message on display) it sends the next 8 bits of the table.

In order to calculate one second, a delayer module is instansiated which activates its output, one second of the last negedge of input. Like the

baud_controller it uses a counter, but of 26 bits. The counter counts only if the state of counter is COUNTING, which state is activated at the posedge of input and deactivated at reaching of 50000000 (Max Counter Value). The counter mises at zero each time the input is activated.

The transmitter controller is an FSM. It has the states:

- **STATE_TRANS_BEFORE**: Sets Tx_WR = 1, so that the transmitter begin the transmission of data counter (of memory) shows. Sets next state as STATE_TRANS_START
- **STATE_TRANS_START**: Sets Tx_WR = 0 and waits until Tx_BUSY is activated. Sets next state as STATE_TRANS_BUSY.
- **STATE_TRANS_BUSY**: Waits until Tx_BUSY is deactivated. Set one input of delayer. Sets next state as STATE_TRANS_AFTER.
- **STATE_TRANS_AFTER**: Set zero input of delayer to wait one second. After delayer output is activated, set counter of memory at next value (++) and state at STATE_TRANS_BEFORE.

Receiver_controller

Get Rx_DATA table of receiver, only when Rx_VALID is one. Drives with the new data the modified FourDigitLEDdriver of lab1 to show the message on monitor.

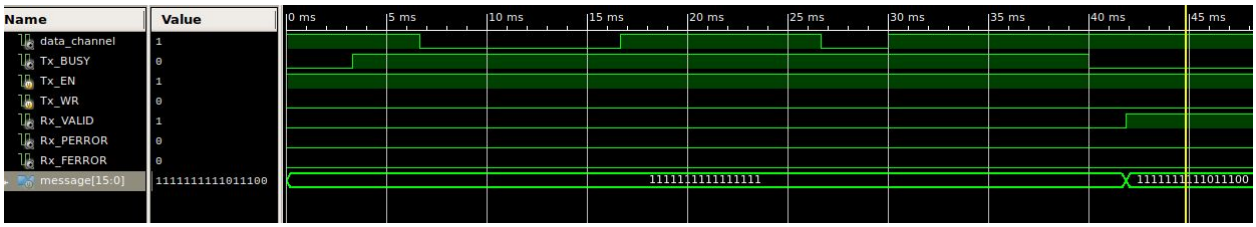
In the FourDigitLEDdriver module is included the anCounter, DCM and LEDdecoder. In place of change_message signal, a valid_message_to_monitor is replaced, and shifts the output message 8 bits left (the 8 right bits now are the new receiver controller got).

Uart_channel

The uart channel includes the transmitter controller and receiver controller, and is actually responsible of data channel which is connected to TxD of transmitter and RxD of receiver.

Verification

Waveform of transmission of message 11011100:



Outputs to monitor regarding to message and an ascent.