Axelou Olympia, 2161
oaxelou@uth.gr
ce430: Spartan 3 FPGA
16/11/2018

Lab2: UART Implementation

The purpose of this assignment is the implementation of a serial communication system which will use the UART (Universal Asynchronous Receiver Transmitter) protocol. The system is composed of a transmitter and a receiver and they communicate through a 1-way,1-bit channel (a simple signal). This system is later used to transfer a series of 8-bit symbols AA, 55, CC, 89) that are displayed on the 7-segment display of the Spartan3 FPGA.

The assignment is divided into 4 parts:

A. The baud rate controller: The calculation of the period when a given baud rate is selected. I have calculated the error as well.
B. The UART transmitter
C. The UART receiver
D. The complete circuit (a driver for the transmitter is used):
    1. The transmission synchronized by the Tx_WR & Tx_BUSY signals
    2. The circuit implemented on the Spartan3 (using switches & the 7segment display)

I successfully implemented the four parts of the assignment. In the following paragraphs, there is a detailed explanation of every part.

Part A: The baud controller

The purpose of the baud_controller circuit is to provide the transmitter and the receiver with the proper sampling signal, according to the chosen Baud Rate. The sampling signal is active-high and remains active for one cycle. Also, its frequency is 16 times faster than the Baud Rate so that the receiver later on will be able to lock on the center of the signal for more accurate sampling.

For the baud rate implementation, I use a counter of 20ns period cycles. The max value of the counter is computed by the following formula:

1. I multiple the selected baud rate (e.g. the 300 bits/sec) with 16.
2. Then, I flip it to 1/x to find the period and multiple it with 10^9 to convert it to ns.
3. I divide it with 20 (the period of the FPGA clock in ns) and round the number to the nearest integer and the result is the maximum value of the counter.

Then to calculate the error, I take the absolute value of ($T_{theoretical}$ – (20ns) * MaxCounterValue). So, for each of the possible baud rate values:

a. 300 bits/sec: 16 * 300 = 4800bits/sec. So, T = 10^9/4800 = 208,333.333ns.
208,333.33 / 20 = 10,146.66 = 10,417 (max counter value) & Error = | 208,333.33 – 20*10,417 | = 6.66ns

b. 1200 bits/sec: 16*1200 = 19,200bits/sec. So, T = 10^9/19,200 = 52,083.33ns.
52,083.33/20 = 2,604.166 = 2,604 (max counter value) & Error = | 52,083.33 – 20*2,604 | = 3.33ns

c. 4800bits/sec: 16*4800 = 76,800bits/sec. So, T = 10^9/76,800 = 13,020.833ns.
13,020.833/20 = 651 (max counter value) & Error = | 13,020.833 – 651*20 | = 0.833ns

d. 9600bits/sec: 16*9600 = 153,600bits/sec. So, T = 10^9/153,600 = 6,510.4166ns.
6,510.4166/20 = 326 (max counter value) & Error = | 6,510.4166 – 20*326 | = 9.5833ns

e. 19200bits/sec: 16*19200 = 307,200bits/sec. So, T = 10^9/307,200 = 3,255.2083ns.
3,255.2083/20 = 163 (max counter value) & Error = | 3,255.2083 – 20*163| = 4.79166ns

f. 38400bits/sec: 16*38400 = 614,400bits/sec. So, T = 10^9/614,400 = 1,627.604166ns.
1,627.604166/20 = 81 (max counter value) & Error = |1,627.694166 – 20 * 81 | = 7.604166ns.

g. 57600bits/sec: 16*57600 = 921,600bits/sec. So, T = 10^9/921,600 = 1,085.0694ns.
1,085.0694/20 = 54(max counter value) & Error = |1,085.0694 – 20 * 54 | = 5.0694ns.

h. 115200bits/sec: 16*115200 = 1,843,200. So, T = 10^9/1,843,200 = 542.53472ns.
542.53472/20 = 27 (max counter value) & Error = | 542.53472 – 20 * 27 | = 2.53472ns.

In each of the cases above, as we expected, the error is less or equal to 10ns (half of the 20ns period). In real time, this error is so small that can be considered negligible.

Implementation

This circuit takes as input the selected baud rate as a 3bit code gives as output the sampling enable signal, which is 1 for one cycle every max-counter-value cycles (the one we calculated above). It is composed by 3 sub-circuits:

- 1 combinational logic always block which sets the max value of the counter (based on the input)
- 1 sequential logic always block which changes the value of the counter
- 1 combinational logic always block which calculates the next state (whether to enable the sampling signal)

Verification

In the testbench I set the 3bit code of the baud rate to each of the possible values. The delay between the assigns is calculated so that we can see at least 2 rises of the sampling signal to measure the distance between them. For example, given the baud_rate = 3'b111, the time we should measure is 540ns.

The following screenshot shows a simulation of the circuit. To measure the period of the the sample_ENABLE signal (the output of the circuit) when baud_rate > 011 we have to zoom in.

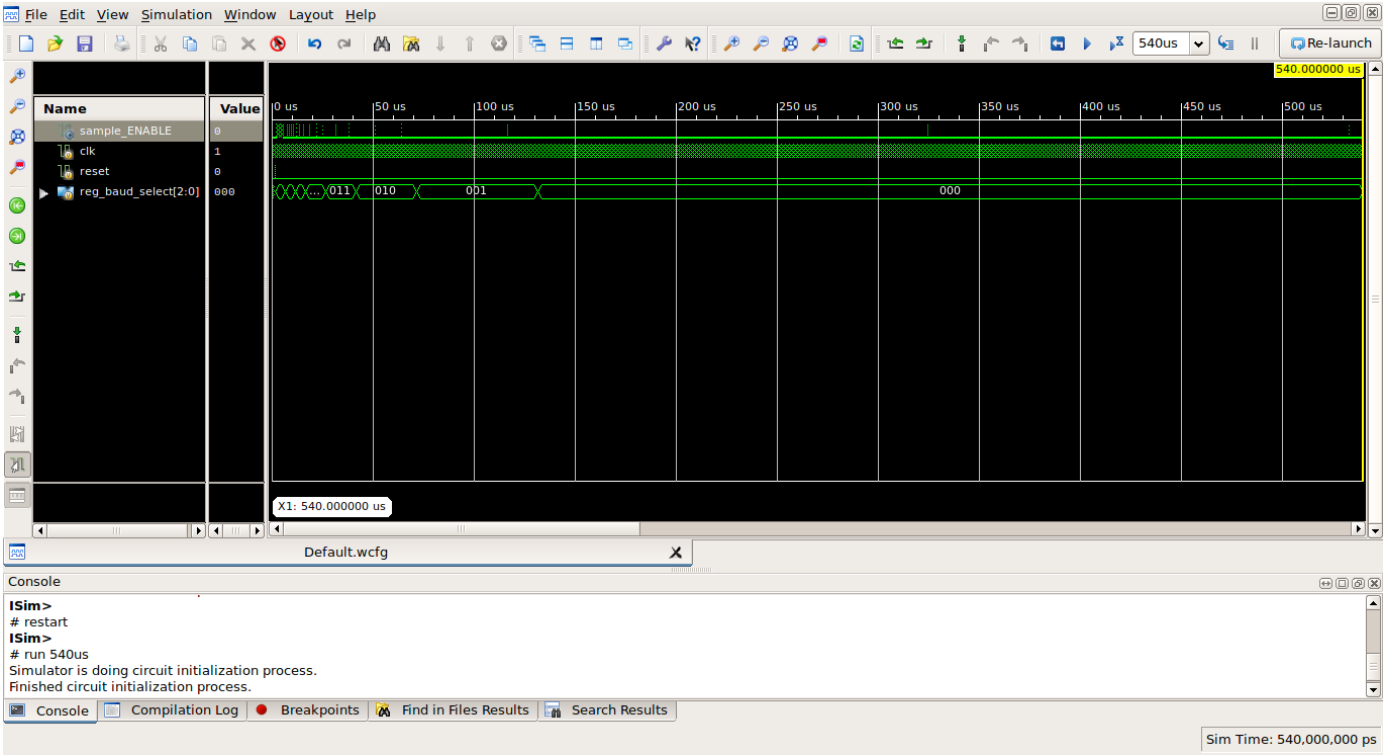


Figure 1: Part A simulation

Experiment / Final implementation

The first part was not tested on the FPGA.

Part B: The transmitter

Implementation

For this part, the circuit is composed of the following sub circuits – modules:

- The top level module (uart.v) which includes the instantiations of the sub-modules (the reset synchronizer and the uart_transmitter) and the wired connection between them and berween the inputs and the outputs.
- The reset synchronizer (synchronizer.v)
- The uart transmitter circuit (uart_transmitter.v)
- The baud rate circuit (baud_controller.v)

The main part of the above is the transmitter. It is composed of the following always blocks – sub-circuits:

- The circuit for the 16-cycles counter which calculates when to transmit
- The control unit, which sends the enable signals to the proper circuit
- The circuit that sends the BUSY signal when data from the system have arrived. It's also used after the transmission to stop sending the busy signal.
- The circuit that stores the data from the user in a buffer
- The transmission unit: increments the data_counter and drives the correct bit of the data buffer to the output

To enable-disable the sub-circuits, I use a 2bit signal and 3 1bit signals. The first one is altered in the control unit and is the one that triggers the circuits and the others become 1 when the circuit has finished its job, e.g. in the unit that stores the data of the user, the data_assigned signal becomes 1 when the transmitter has successfully received all of the 8 bits and has stored them in a buffer. This is when the control unit enables the transmission unit.

The control unit and the busy-signal-sending unit are in the clock domain of the system (so that the transmitter enables the busy signal on time and there won't be any loss of data). The 16-cycles counter is enabled every Tx_sample_ENABLE cycles (which is the output of the baud_controller) and produces the transmit_ENABLE signal. The data_assign and the transmission unit are in the transmit_ENABLE clock domain.

The initialization of each sub-circuit is done when another sub-circuit is enabled. So, for example, in the always block of the transmission unit, its main work is performed when the circuit_enabled 2bit signal has the transmission_ENABLE value and initializes the circuit when it has the data_assign_ENABLE value (this method for the initialization is used in the receiver as well).

Verification

To test the circuit, I use the proposed series of symbols ("aa", "55", "cc", "89"). These inputs cover both cases of the parity bit being 1 and 0. Every 250us (+ 20ns for the Tx_WR), it ensures that the busy signal is down and then assigns the symbols and enables the transmitter by turning to 1 the Tx_WR signal for 1 cycle.

The following screenshot shows the correct transmission of the symbols (the waveform of TxD, the channel).

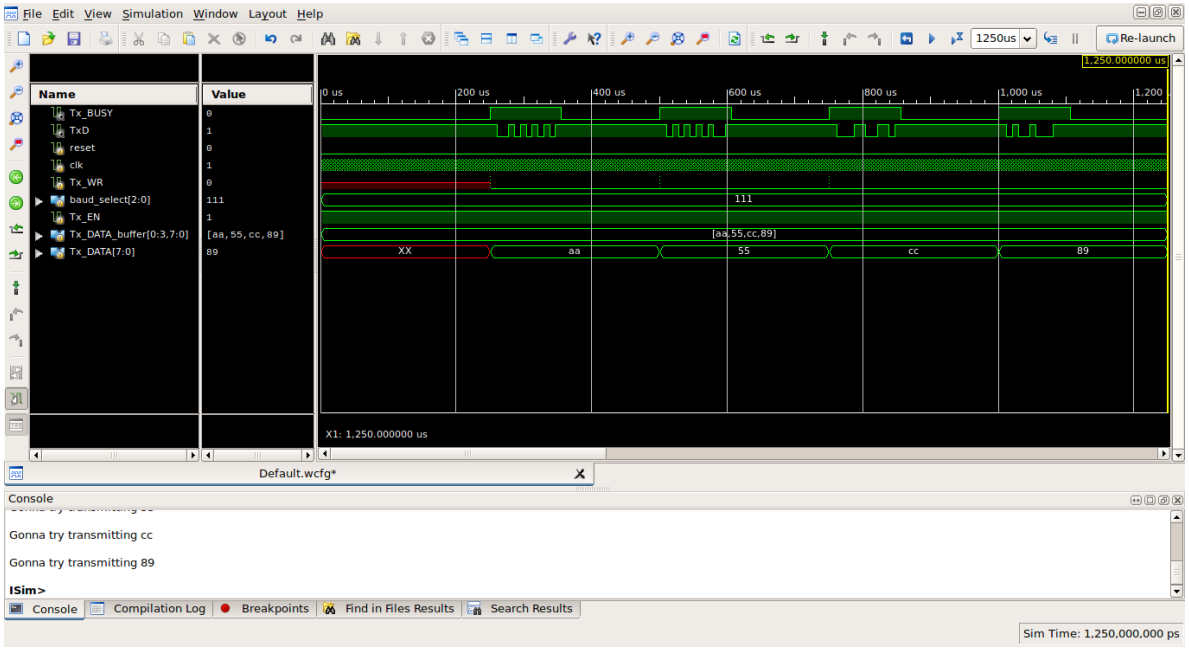


Figure 2: Part B simulation

The following screenshot shows the correct function of the 16-cycles counter for the transmission. It's the transmission of the parity bit and shows that there are 16 rises of the Tx_sample_ENABLE in a one-bit transmission.

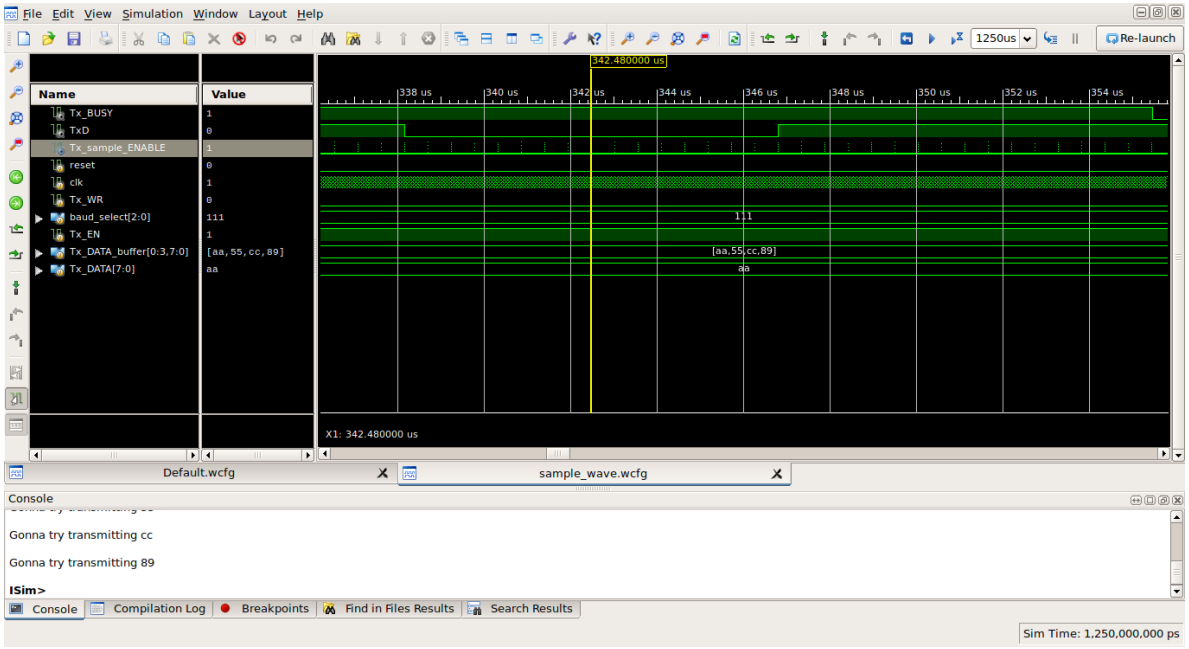


Figure 3: Part B – Zoomed in the parity bit of the "AA" symbol transmission

Experiment / Final implementation

This part was not tested on the FPGA.

Part C: The receiver

Implementation

For this part, the circuit is composed of the following sub circuits – modules:

- The top level module (uart.v) which includes the instantiations of the sub-modules (the reset synchronizer, the uart_receiver) and the wired connection between them and berween the inputs and the outputs.
- The reset synchronizer (synchronizer.v)
- The channel synchronizer (channel_syncrhonizer.v)
- The uart receiver circuit (uart_receiver.v)
- The baud rate circuit (baud_controller.v)

The module of the receiver is divided into 6 sub-circuits – always blocks:

- The control unit. It synchronizes all the other sub-circuits. As in the transmitter, it uses “start” and “finished” signals to enable and disable the circuits.
- The circuit that searches for the start bit
- The circuit with the counter. It is used for both aligning the sampling process to the center of the input signal and for counting the cycles for the next input data.
- The data collector unit. It saves in a temp buffer the 8 bits that it gets from the channel and also at the end of the communication, it is the circuit that sends to the system (the user) the symbols it received.
- The circuit that checks the parity bit (parity error)
- The circuit that checks the end bit (frame error)

In total, there are 6 sub-circuits (so 6 phases of the receiving process).

As in the transmitter, I use a 3bit signal that controls which circuit is enabled and 6 signals that inform the control unit that the corresponding circuit has finished its job. Every sub-circuit is initialized for the next set of data when another is working.

Inside the circuit of the receiver, there are 2 instantiations: the one is for the baud_rate and the other is an instance of a synchronizer which synchronizes the channel with the receiver’s sampling rate. I don’t use the same synchronizer as the one for the reset signal because the channel has to be initialized when the reset is pressed. So, in the flip-flops, I use the reset signal to set the channel to its equilibrium state, to 1.

If a parity or framing error is found, the output data (Rx_DATA) is not changed. So, it keeps the last valid data sent though the channel and if after the reset only parity and framing errors have been found, then the output will have zeros (because in reset I initialize with zeros the output).

The control unit, the circuit that searches for the start bit and the counter circuit are in the Rx_sample_ENABLE clock domain and the other three (sampling, parity bit checker and frame checker) are in the sample_ENABLE clock domain (the signal that the counter produces after locking in the center of the bits for the sampling).

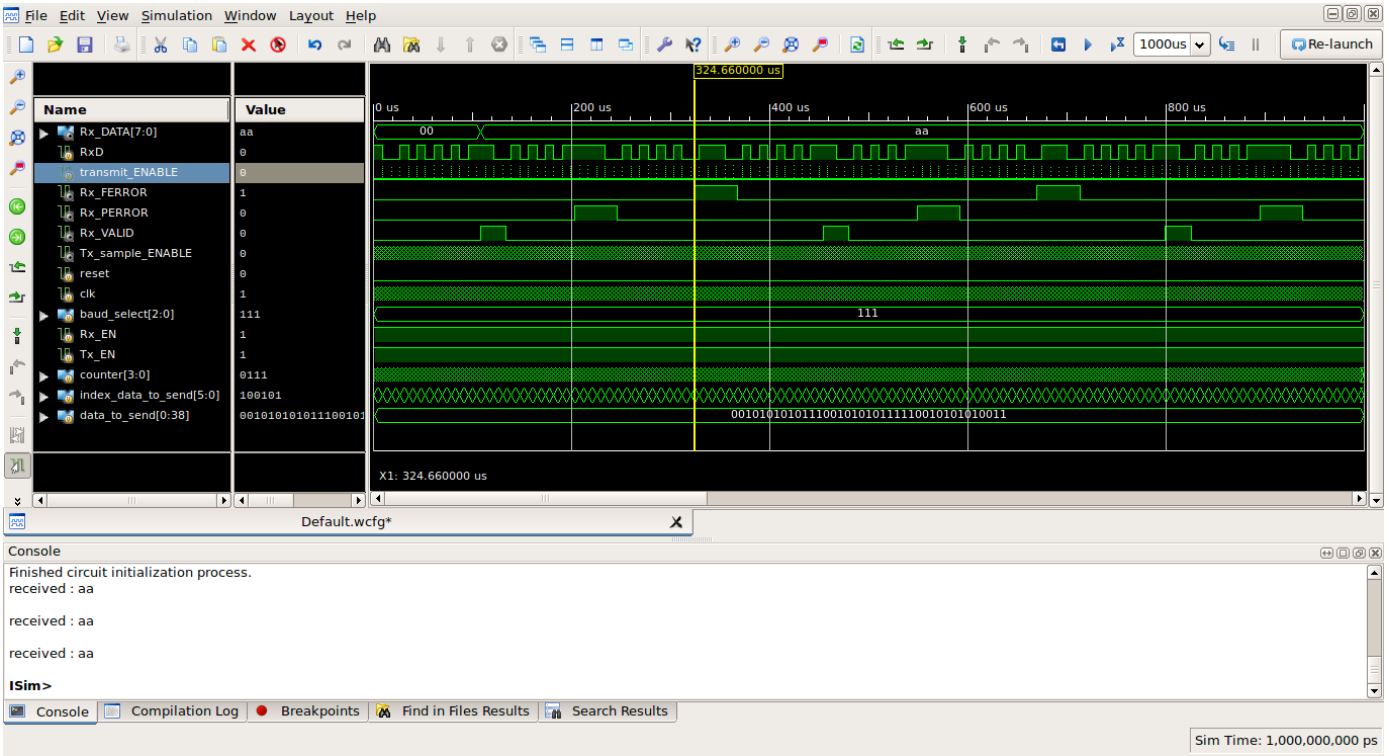
Verification

To test the circuit, I took parts of the transmitter (like the always block where it calculates when to transmit) and created a simple transmitter in the testbench. I added an always block that decides which part of the input vector to transmit and another that takes the output of the receiver and checks if a valid set of data has been received (plus, it displays in the monitor that it received valid bits). The always block that takes bit-by-bit the input doesn't end: it resets the counter when it reaches 39 and restarts.

The input vector consists of 3 inputs (11bit series) parted by 2 bits to delay the transmission. So,

1. [00:10]: The first input. It is a correct series of bits and it sends the symbols "AA".
2. [11:12]: Fillers to delay the transmission process.
3. [13:23]: The second input. The same as the first input with the only exception that the parity bit is wrong. We expect PERROR here.
4. [24:25]: Fillers to delay the transmission process.
5. [26:36]: The third input. Again, the same as the first one but this time the error lies in the end bit: It is a zero. We expect FERROR here.
6. [37:38]: Fillers to delay the transmission process.

The following screenshot shows the correct interaction of the receiver circuit with the input above. In the monitor, we can see the output of the $display: it has received three correct inputs (out of the 8 that we transmitted).



Experiment / Final implementation

This part was not tested on the FPGA.

Part D: The completed circuit – UART System

Implementation

For this part, the circuit is composed of the following sub circuits – modules:

- The top level module (uart.v) which includes the instantiations of the sub-modules (the reset synchronizer, thre transmitter driver, the transmitter, the uart_receiver) and the wired connection between them and berween the inputs and the outputs.
- The reset synchronizer (synchronizer.v)
- The channel synchronizer (channel_synchronizer.v)
- The transmitter driver circuit (uart_transmitter_driver.v)
- The uart transmitter circuit (uart_transmitter.v)
- The uart receiver circuit (uart_receiver.v)
- The baud rate circuit (baud_controller.v)

What differs from part B and C is that the driving of the input is not done in the testbench but in a specific circuit (the transmitter driver) and the testbench is only for initializing the enable signals, baud rate and reset.

In particular, the transmitter synchronizer is composed of a sequential logic always block and it's in the clock domain of the FPGA clock (20ns). It stores in a buffer the 4 sets of symbols ("aa", "55", "cc", "89") and in every positive edge of the clock, it checks if the BUSY signal is down. If this is the case, it drives the corresponding symbol (it uses a word counter) and sets the WR signal to 1. Then, in the next positive edge of the clock, it sets it back to zero.

The rest of circuits are the same as in the previous parts.

Verification

The circuit is automated, so the only handling in the testbench is for the baud rate, the reset and the enable signals. As for baud rate, I alter it after a period of time to check that there are any glitches when it changes.

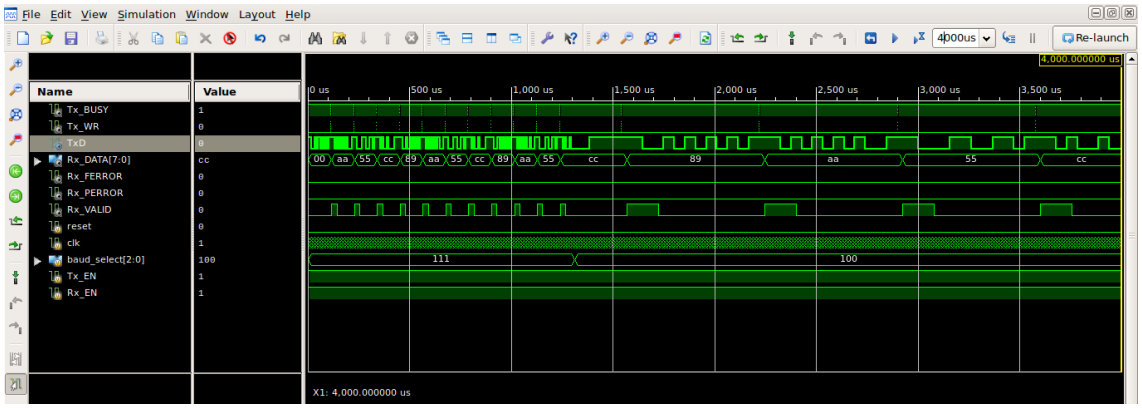


Figure 4: part D - Simulation

Experiment / Final implementation: This part was not tested on the FPGA.

Part D: UART System on the FPGA

Implementation

For this part, the circuit is composed of the following sub circuits – modules:

- The top level module (uart.v) which includes the instantiations of the sub-modules (the reset synchronizer, thre transmitter driver, the transmitter, the uart_receiver, the 7segment driver, the LEDdecoder and the DCM) and the wired connection between them and berween the inputs and the outputs.
- The reset synchronizer (synchronizer.v)
- The channel synchronizer (channel_synchronizer.v)
- The transmitter driver circuit (uart_transmitter_driver.v)
- The uart transmitter circuit (uart_transmitter.v)
- The uart receiver circuit (uart_receiver.v)
- The baud rate circuit (baud_controller.v)
- The seven_segment_driver (seven_segment_driver.v)
- The LEDdeocder (LEDdecoder.v)

This is the only part of the assignment that is synthesizable. As a result, only here we need a .ucf file. The inputs and the outputs of the top level module are the following:

- The clock (20ns)
- The reset signal, controlled by the first button (L14)
- The baud select code (3bit), controlled by the first 3 switches (K13, K14, J13)
- The Tx_EN & Rx_EN signals, controlled by the 4th and the 5th switches (J14, H13)
  - The 3rd and the 4th digit of the seven-segment display (an1 & an0)
  - The 7(out of 8) segments to display the characters

Transmitter driver:

As I noticed while testing the circuit on the FPGA, even the slowest baud rate was too fast. To fix it, I added a 3bit counter so that every set of symbols would be transmitted 8 times before moving on to the next one. The result is to display each set for a period 8 times longer than before this change.

Seven-segment driver module:

It uses 2 flip-flops to store the message to display. If it receives a PERROR or FERROR signal, it assigns to the flip-flops the messages “PE” and “FE” respectively. If it receives the VALID signal then takes the data from Rx_DATA. The rest of the circuit is the same from the first assignment (with the only exception that we only care about an1 and an0). The circuit is in the clock domain of the divided 20ns period: 320ns.

LEDdecoder module:

The only difference from the first assignment is that I have mapped the ‘P’ character on the ‘1’ character, as we don’t need the latter (the message is fixed).

Verification

The testbench I used for this part doesn't differ much from the other one of part D – I just don't change the baud rate. The way the output is displayed is of course different. What we expect to see is:

- The an3, an2 & dp constantly at 1.
- A pattern of the 4 different inputs:
    1. The "AA": 00010001 (for each of the 8 segments)
    2. The "55": 01001001
    3. The "CC": 11100101
    4. The "89": 00000001 & 00011001 changing as the anodes open and close

In the beginning, the circuit I had created in part C (the receiver's and the transmitter's circuit) was not synthesizable because of registers that were changed in multiple drivers and after I fixed this issue, I changed the circuits to the final form (the one I explained above). Apart from this error, there wasn't anything else I noticed in the simulation process.

The following screenshot is from Post-Route Simulation.

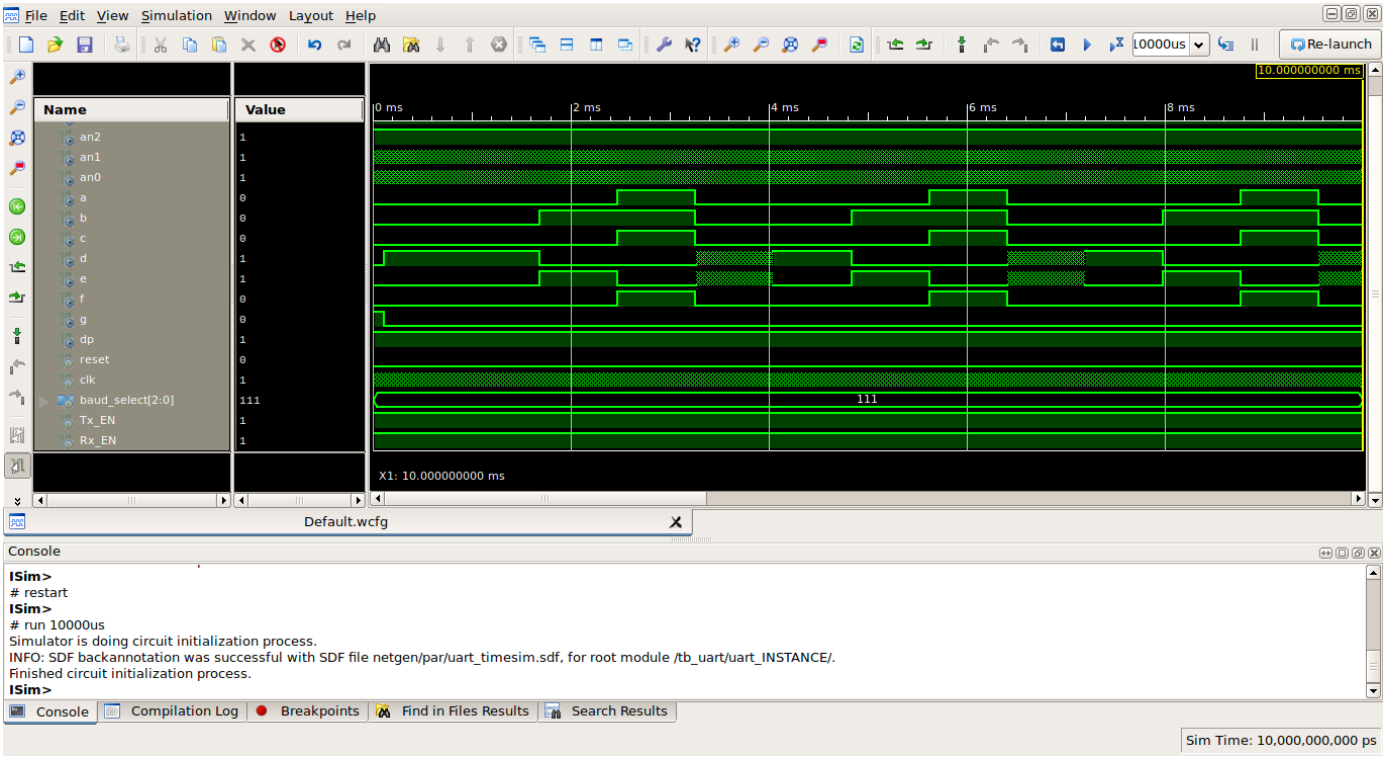


Figure 5: part D – Post Route Simulation

Experiment / Final implementation

In the lab, after we checked the waveforms, I made the Bit File and downloaded it on the FPGA. The reset button was working properly. However, I had forgotten to drive the an3 and the an2 to 1 so all of the 4 digits of the 7segment display were open and it didn't display exactly what we expected. Furthermore, it was changing very fast.

I fixed the anodes and added the counter in the transmitter driver module that I mentioned. We tried again and it was working as it should. So, in total there was one change after the testing of the circuit on the FPGA.

We tested the switches for the baud rate and noticed how it accelerated and we enabled and disabled the receiver and the transmitter.

We did a testing in which we only disable the transmitter and we noticed the "FE" message (frame error). This means that the communication was interrupted before it ended, and particularly it ended with zero. The error message was still displayed until we re-enabled the transmitter or pressed reset button. This makes sense as according to the implementation (this is mentioned above in the receiver – part C), the Rx_DATA is only changed when a valid set of symbols is received, or the reset signal is sent.

In the following photo (taken in the lab while checking the circuit on the FPGA), I froze the communication by disabling both receiver and transmitter (the 4th and 5th switches are down).

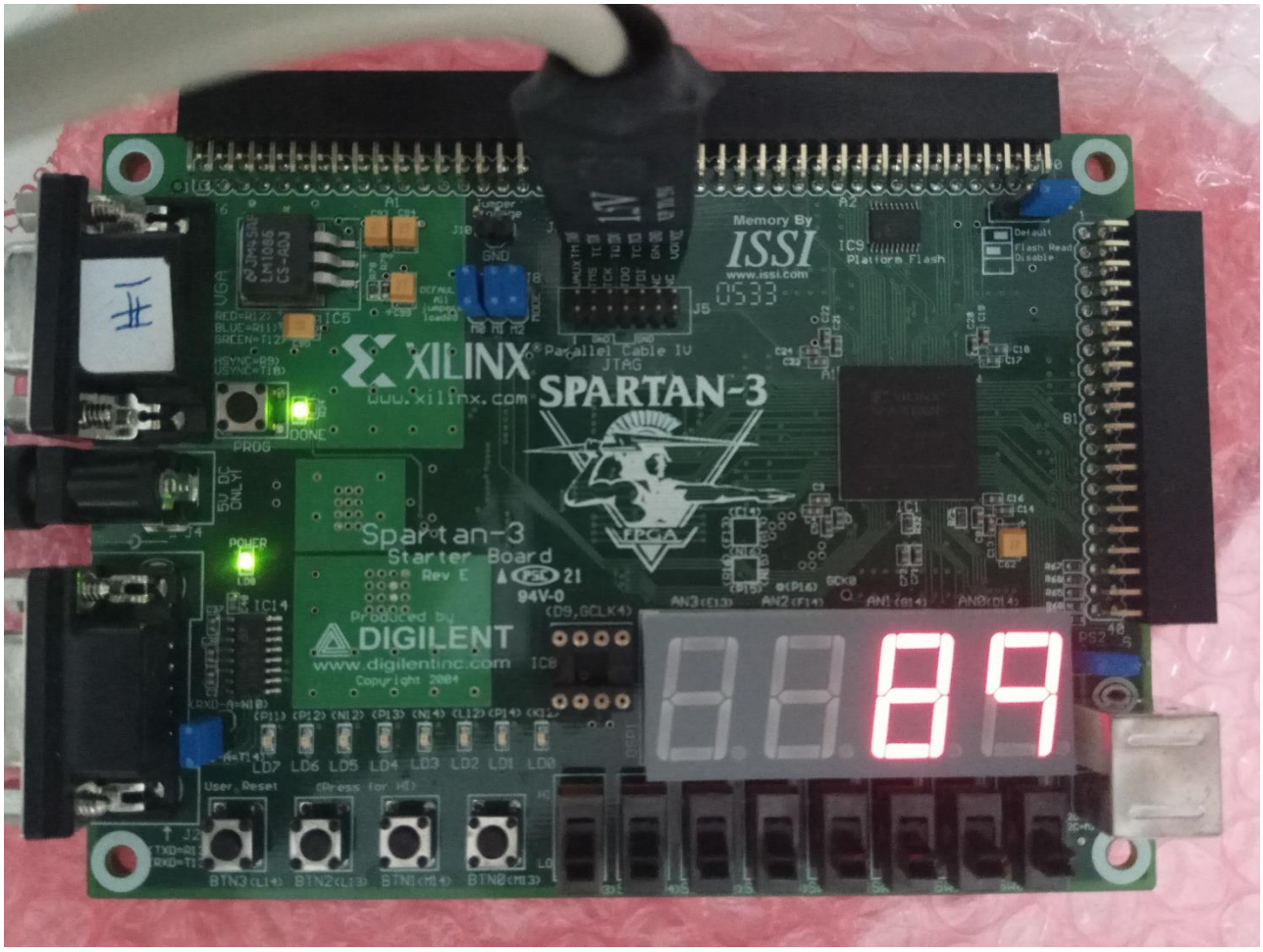


Figure 6: Testing part D on the FPGA board

Conclusion

Overall, I didn't encounter any major problems while developing the circuits for each part.

The first 3 parts (baud controller, transmitter, receiver) were implemented quite easily. The part that seemed to me quite tricky was when I had to make sure that what I created was synthesizable. That was when I changed the circuit of the receiver and part of the transmitter (as these two are closely associated).

Another part where I spent some time was the initialization of the sub-circuits for the next set of data, especially of the ones that are at the end of the communication process. For example, in the receiver, the initialization of the frame checking has to be done in a previous step of the *current* communication and not in the current communication for the *next* set of data (in opposition to the data collector circuit which is reset in the last part of the communication: when we send to the system the symbols we received).

~