UMONS
Université de Mons

Faculté Polytechnique

Hardware/Software Platforms

G4-DE1-Serial-RS232

Tutorial technical document

Hugo BOHY, Bastien VAN ESBEEN

POLYTECH MONS

Under the supervision of the
Professor C. VALDERRAMA SAKUYAMA

Academic year 2019-2020

pôle hainuyer

Contents

Introduction

This tutorial aims to understand the main principles of a serial communication using RS232. We are going to observe how it works via simulations using two different modules: write and read. These modules correspond to the emitter and the receptor sides of the communication and their behaviour will be explained, as well as the waveforms. For each module, a test-bench was created, we explain how to use it and how to make the simulations. Moreover, an application is added on the read part (a 7-segments display), in order to give a concrete aspect to the simulations.

Fig. 1 shows the global schema block of the project. Each part, inputs and outputs will be explained in the following sections.

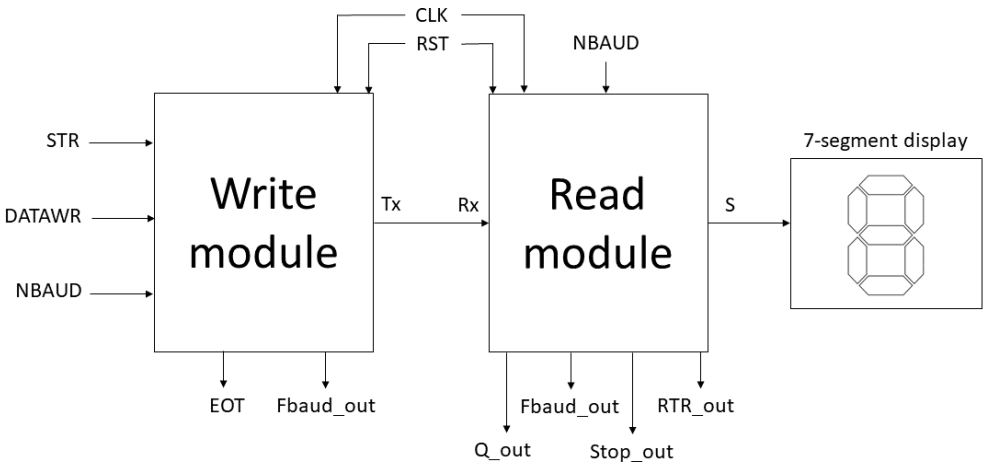

Figure 1: Project opening

1 Getting Started

1.1 Download and installation

In order to use and simulate the RS232 project, you have to go to this GITHUB link (https://github.com/HuBohy/RS232) and download the **RS232_AlteraProject.zip** file. Once downloaded, extract it in a specific folder (let's call it *RS232folder* for clarity). The ZIP file contains both **Write** and **Read** modules in separated folders. Then launch **Quartus Prime**, select "Open Project" (cfr. Figure 1.1) and enter the folder of the module you want to simulate (Project files are *.qpf*).

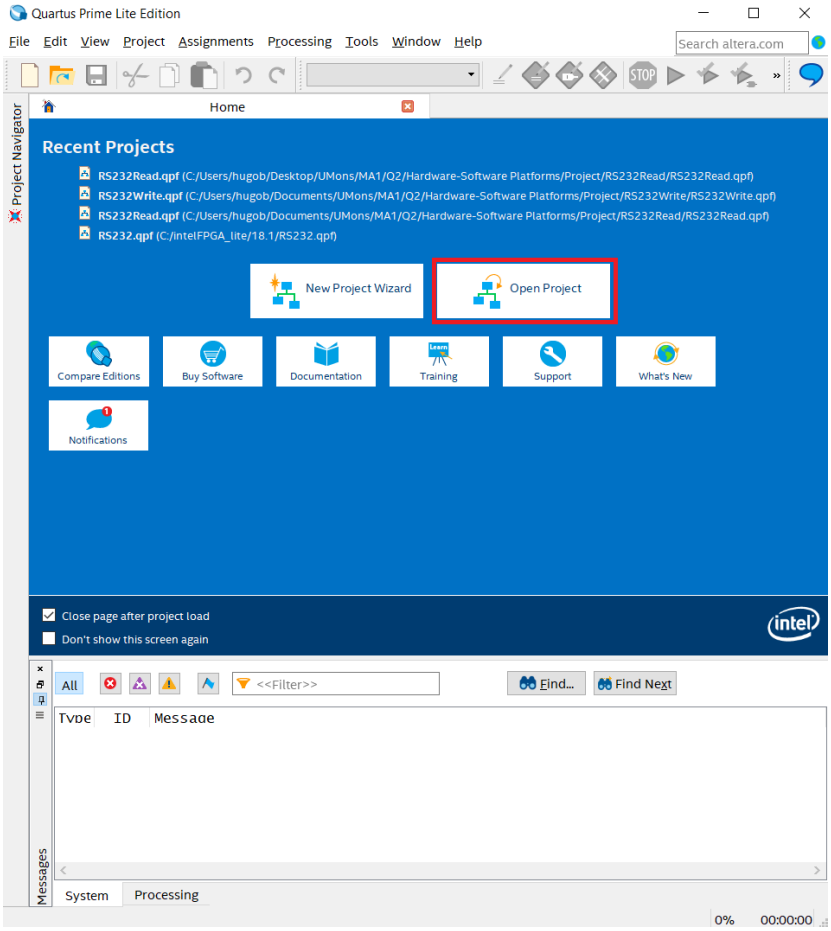

Figure 1.1: Project opening

1.2 Run a simulation

To launch a simulation, go to **Tools/Run Simulation Tool/RTL Simulation**. If ModelSim is installed correctly, a new window will open. The project already contains the required configurations to launch the test bench simulation, but you have to stop it and launch it again (cfr. Figure 1.2).

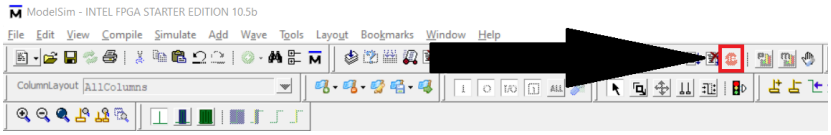

Figure 1.2: Stop the simulation

2 RS232 - Write

2.1 Source code

The **Write** module contains four source code files and one test bench file. The source code includes *BaudRate.vhd*, *FsmWrite.vhd*, *RightShift.vhd* and their top level file *RS232Write.vhd*.

RS232Write.vhd is the top level file of the module. It shows what are the inputs (RST, CLK, STR, DATAWR, NBAUD) and outputs (FBaud_out, EOT, Tx) of the driver, it links each component (BaudRate, FSMWrite, RightShift) to each other by connecting the corresponding ports (e.g. "CTRL" output from FSMWrite to "CTRL" input fromRightShift). It also creates the internal signal of the project, used and controlled by the components:

- FBaud, goes to '1' when a new bit is written (i.e. every N clock periods, see explanations below), else '0' ,
- CTRL, 4-bit index of the current state for the FSM.

BaudRate.vhd manages the Baud Rate (BR) of the communication (the number of symbols transmitted per second, one symbol = one bit in this case) for a clock of 50 MHz. A bit will be emitted every N periods of clock. So, by choosing N, we fix BR.

$$BR = \frac{F_{clock}}{N} \tag{2.1}$$

We can choose N among fourteen different values, corresponding to baudrates from 110 bits/s to 256 kbits/s. For example, if we want BR equal to 256 kbits/s, we choose $N = 196$. As N must be an integer, the actual value of BR, found by (2.1), equals more precisely 255.102 kbits/s.

FsmWrite.vhd is a Finite State Machine (FSM) that manages the transition between each sent bit. The FSM starts at a state that holds the transmitter ready to send (or write) a bit, but not sending yet. When the signal STR (that means "start") is '1', the transmitter is set on "busy" because the transmission has started. Those two first states are called "synchronisation states". Next states correspond to every data bit, from bit 0 to bit 7. The end of a byte is reported by going back to the first state "Hold" and the value '1' for "EOT" (End of Transmission). The transmitter keeps EOT = '1' if STR is '0', else the transmission starts for the next byte.

RightShift.vhd gives the output bit to write in the communication channel. For the two first states of the FSM, considered as the synchronisation states, the outputs are always '1' followed by '0', as a way to show that significant data are coming. The next outputs are the eight bits of the data we want to write.

2.2 Test bench

RS232folder/simulation/modelsim/rs232_write_tb.vhd describes the test bench of the module. It sets values allocated to each input of the module:

- CLK (clock): 50 MHz (20 ns period),
- RST (reset): '1' for one clock period, then '0',
- NBaud (baud rate index): "1101", corresponding to 256 kbits/s,
- STR (start) to '1' every time "RST" is '1',
- DATAWR: the data byte to be sent. The default value is 0x00 (0x means the value is expressed in hexadecimal). Each time a byte is sent, the value of DATAWR changes and it corresponds to a list of 4 successive bytes: 0x00, 0x77, 0x2A and 0xB7. Theses three last values are here arbitrary chosen for the simulations. Once all the bytes are sent, STR goes to '1'.

2.3 Simulation

The following figures show a 160 μs simulation according to the test bench. The way to read this is as follows:

- RST (after 20 ns), CLK and NBaud are constant during the whole simulation,
- DATAWR shows the current byte that is written in the channel,
- FBaud_out is a wire that allows to see the value of internal signal FBaud,
- EOT is '1' when a byte is fully written,
- Tx is the channel of transmission.

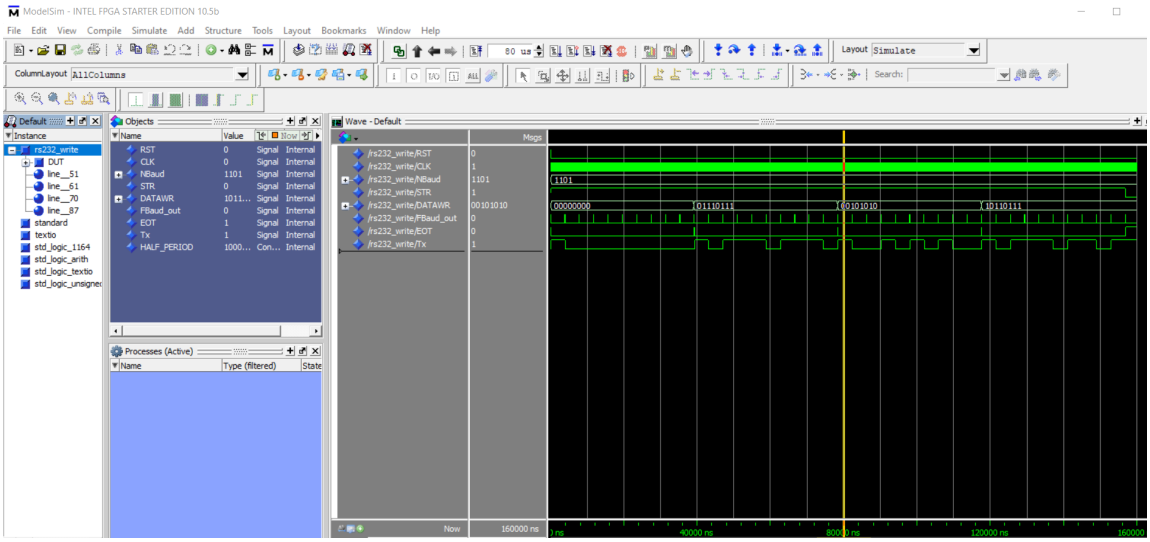

Figure 2.1: 160 μs of simulation

As mentioned in section 2.1, every new byte transmitted is preceded by '1' then '0', and it is confirmed in Figures 2.1 and 2.2. The first two bits correspond to this synchronisation, and the following bits are the data byte. For example, the transmission of "0x77" (= "01110111", Figure 2.2) gives Tx the values in this order : '1', '0', '1', '1', '1', '0', '1', '1', '1', '0'.

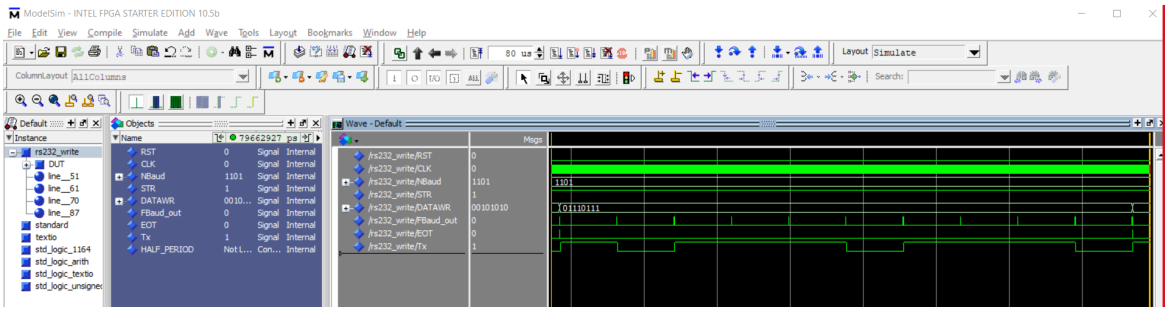

Figure 2.2: Zoom on one byte (0x77)

As the actual BR is 255.102 kbits/s, the period of one bit is 3920 ns (see Fig. 2.3).

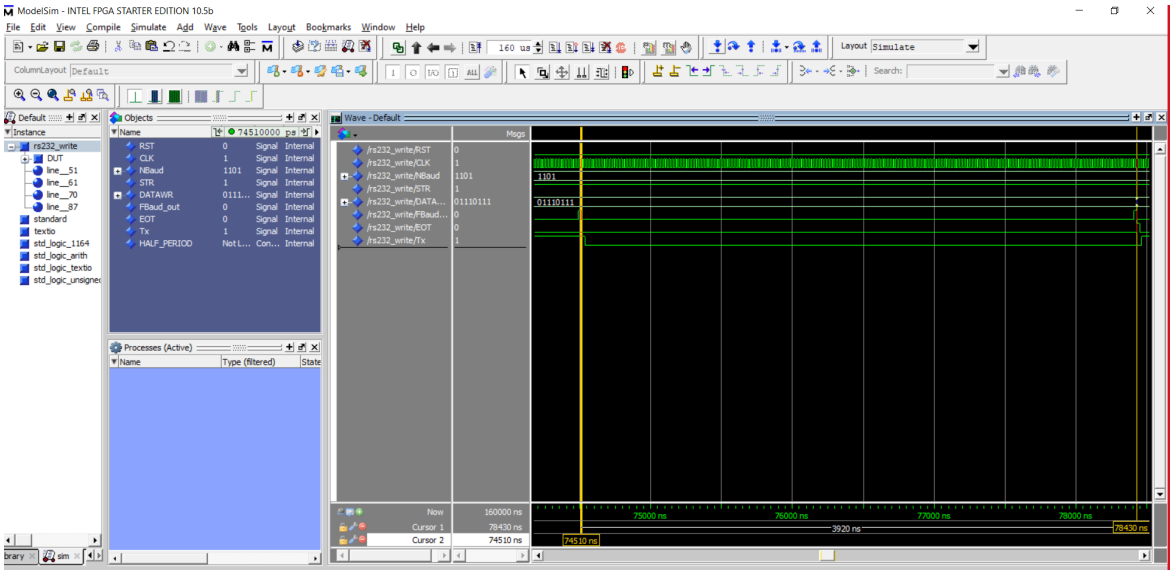

Figure 2.3: Zoom on the transmission of one bit

3 RS232 - Read

3.1 Source code

The **Read** module includes both the driver and the application (7-segments display) at the same time. It contains five source code files and one test bench file. The source code includes *baudraterd.vhd*, *FsmRead.vhd*, *regserpar.vhd*, *hex_ 7seg.vhd* and their top level file *RS232Read.vhd*.

RS232Read.vhd is the top level file of the module. It shows what are the input (RST, CLK, Rx, NBaud) and output (RTR_out, Stop_out, FBaud_out, Q_out, S) pins of the driver, it links each component (baudraterd, FsmRead, regserpar, hex_7seg) to each other by connecting the corresponding ports (e.g. "RTR" output from FsmRead to "RTR" input from regserpar). It also creates the internal signal of the project, used and controlled by the components:

- FBaud, goes to '1' when the counter of N clock periods is over, else '0' ,
- DTR (Data Terminal Ready), indicates if the Terminal is ready to read '1', else '0' and the counter is reset,
- RTR (Ready To Receive), indicates when the receiver can read a bit,
- Stop, which stops the reading of the receiver,
- Q, the buffer for the received byte (8-bit vector).

baudraterd.vhd manages the Baud Rate (BR) of the communication in the same way as *BaudRate.vhd* of the **Write** module (see section 2.1).

FsmRead.vhd is a Finite State Machine (FSM) that manages the transition between each read bit. The FSM starts at a state that stops the receiver from reading data by setting internal signals DTR and RTR to '0'. When the counter is done (internal signal FBaud to '1'), the receiver starts reading the bits (DTR to '1'). While DTR is '1', the receiver can read bits every time the counter is over (making the value RTR go to '1'). Once every of the eight bits are "read", the receiver is put on hold again and the process can restart from the beginning, with new values incoming.

regserpar.vhd acts as the buffer of the **Read** module. It can contain 8 bits. Each time a bit is received, the buffer is shifted and the last bit of the buffer is transmitted to the input buffer Q.

hex_ 7seg.vhd is the component related to the application of the module: display the read values on a 7-segments. It outputs on the S pin the bit values correspond to the hexadecimal representation of the data. The value displayed is updated every time the whole byte is read and received.

3.2 Test bench

RS232folder/simulation/modelsim/rs232_ read_ tb.vhd describes the test bench of the module. It sets values allocated to each input of the module:

- CLK: 50 MHz (20 ns period),
- RST: '1' for one clock period, then '0',
- baud rate index: "1101", corresponding to 256 kbits/s,
- Rx: reads the last bit of a buffer (detailed in the simulation section). The buffer first contains an arbitrary byte (the same as for the write part) and is then shifted bit per bit. Once the byte is read, the buffer takes the next byte to be read. As a reminder, the three bytes to read are 0x77, 0x2A and 0xB7 (and 0x00 the default value).

3.3 Simulation

The following figures show a 160 μs simulation according to the test bench. The way to read this is as follows:

- RST (after 20 ns), CLK and NBaud are constant during the whole simulation,
- Rx shows the current bit received at the receiver,
- Sent is the buffer containing the next bits to put in the channel,
- NSent is the 8-bit value we want to read at the end of the byte transmission,
- FBaud_out is a wire that allows to see the value of the corresponding internal signal, as well as for RTR_out, Stop_out and Q_out),
- S is the 7-bit representation of what the 7-segments displays.

Fig. 3.1 shows the 160 μs of simulation. We can observe the different read bytes in Nsent: 0x77, 0x2A, 0xB7, 0x00, etc.

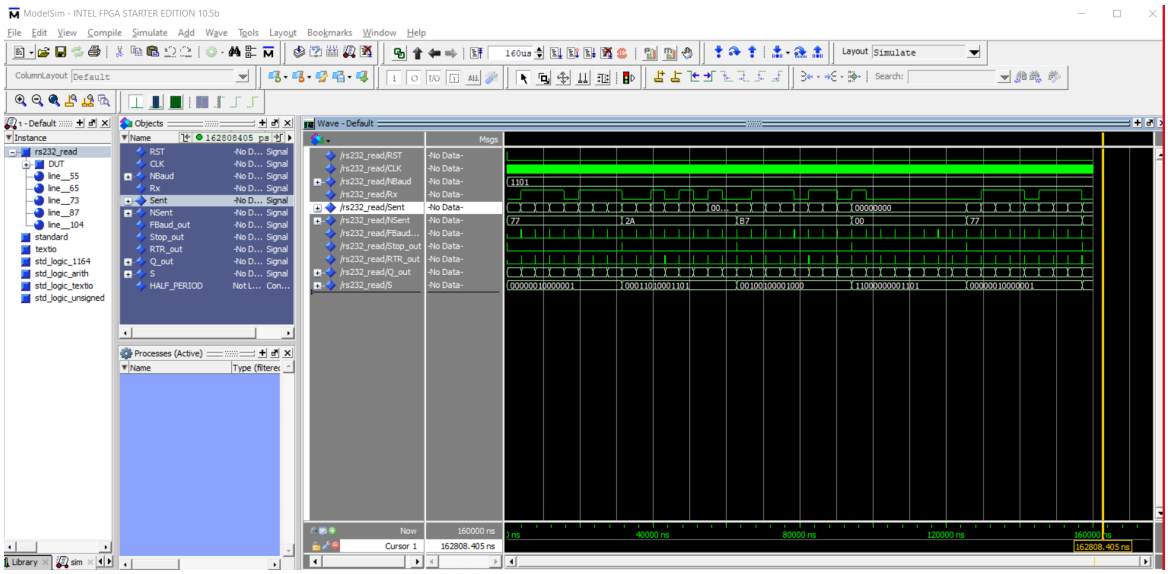

Figure 3.1: 160 μs of simulation

To understand the values in Q_out, let us focus on the reception of the byte 0x77 (Fig. 3.2). The buffer is initially 0x00. Table 3.1 shows the evolution of Q_out, bit per bit. As the next byte to read is 0x77 ("01110111"), the bits will be entered one by one to obtain the wanted value.

Binary	Hexadecimal
00000000	0x00
10000000	0x80
11000000	0xC0
11100000	0xE0
01110000	0x70
10111000	0xB8
11011100	0xDC
11101110	0xEE
01110111	0x77

Table 3.1: Evolution of Q_out bit per bit

In Rx, we can clearly see "01110111", which corresponds to byte we need to read.

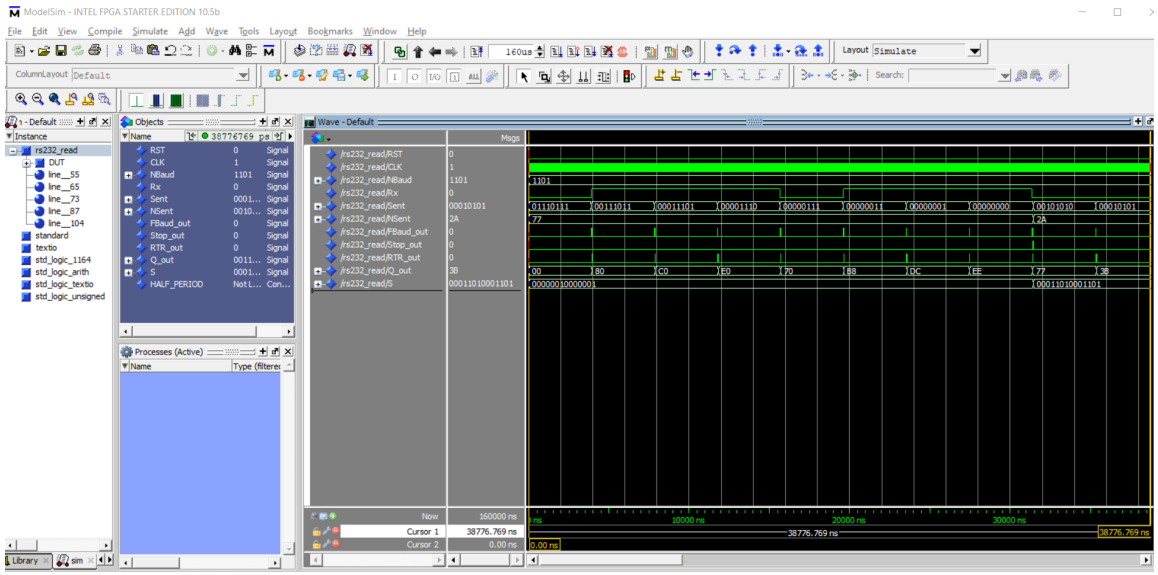

Figure 3.2: Zoom on one byte (0x77)

As the actual BR is 255.102 kbits/s, the period of one bit is 3920 ns (see Fig. 3.3). It corresponds to the period of the emitter side.

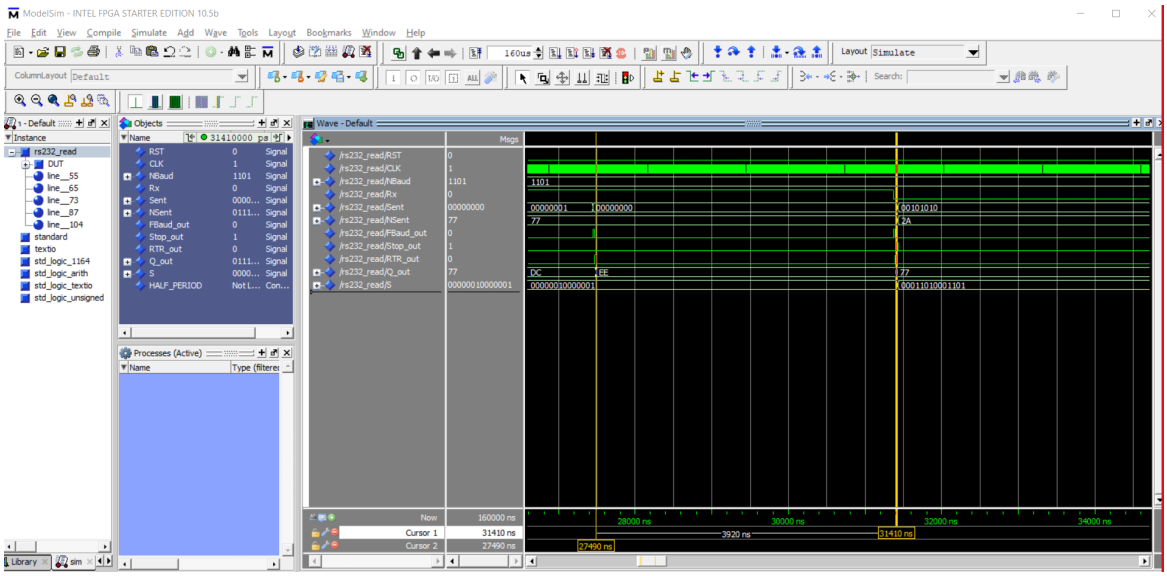

Figure 3.3: Zoom on the reception of one bit

Conclusion

With this tutorial, we were able to analyse a serial RS232 communication. After having precisely examined the read and write modules, we can conclude that both work properly and act coherently with each other. The simulations showed logical results in the frame a RS232 communication.