UNIVERSITÁ DI PISA
ENGEGNERIA DELL'INFORMAZIONE

Project Report
UART Receiver

Bruno Augusto Casu Pereira de Sousa

Electronics Systems a.a. 2022/2023

Pisa, 2023

Contents

1. Introduction

The project presented in this document is the implementation of the Receiver part of a Universal Asynchronous Receiver/Transmitter (UART) peripheral. The UART is a simple protocol for bi-directional serial data transmission, generally used for microcontroller applications. The peripheral device provides a hardware interface for fast serial-to-parallel data conversion from external devices sending to the CPU, and for parallel-to-serial data conversion when receiving from the CPU. The main portion of the UART hardware interface consists in two data lines, *tx* and *rx*:

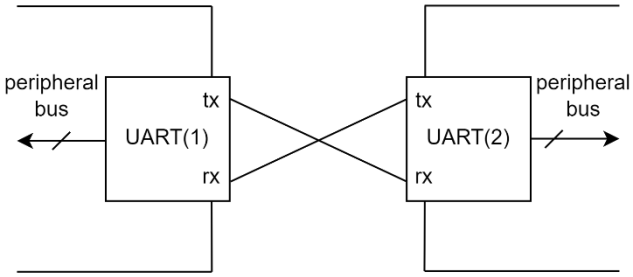


Figure 1 – UART hardware interface

For the transmission of data, the UART protocol defines that the data lines must be kept in high (‘1’) to represent the idle state. To start a new transmission, the line must be set to GND (‘0’) for one period, indicating the start bit. After the start of the frame, the word bits are transmitted, followed by a parity bit (optional). To complete the transmission, a defined number of stop bits must be transmitted, keeping the UART lines at high and thus leaving it in the idle state at the end of the frame. An example with different configurations is shown in Figure 2:

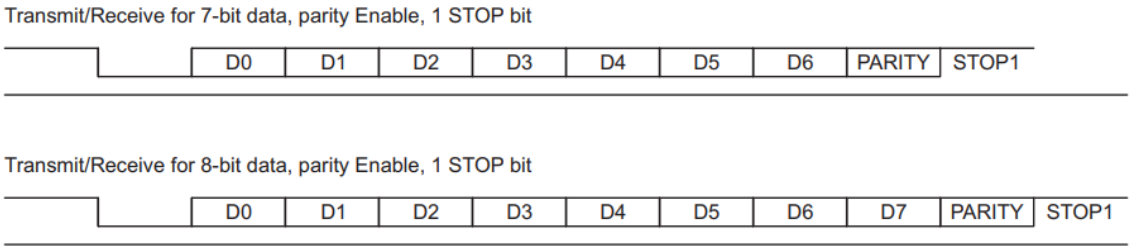


Figure 2 – UART protocol formats (Source: Texas Instruments)

A common use of the UART is to provide an interface for character transmission from CPU to peripheral devices, such as sensors in an IoT application, to control and obtain the measurements from the devices. Another possible usage is to connect the CPU to an external serial device, working as a converter, for example, to an USB port connected. By using this implementation, the UART can reduce the process demand from the CPU, as it uses a much-simplified protocol and an easy to build dedicated hardware.

To build a peripheral UART in a microcontroller architecture, the design must implement efficient ways to read and write the data from the serial lines to and to perform the serial-to-parallel and parallel-to-serial operations. A possible way to do so is by using a shift register connected at the input and output ports of the UART, combined with a buffer and a time and control logic. In addition to the data reading, the peripheral must provide error and control signals for the system interrupt control.

An example of hardware implemented UART peripheral by Texas Instruments was studied to guide the final design of the module developed, and to understand the overall behavior of the peripheral. The block diagram presented highlights the receiver part of the UART peripheral found in the TI device:

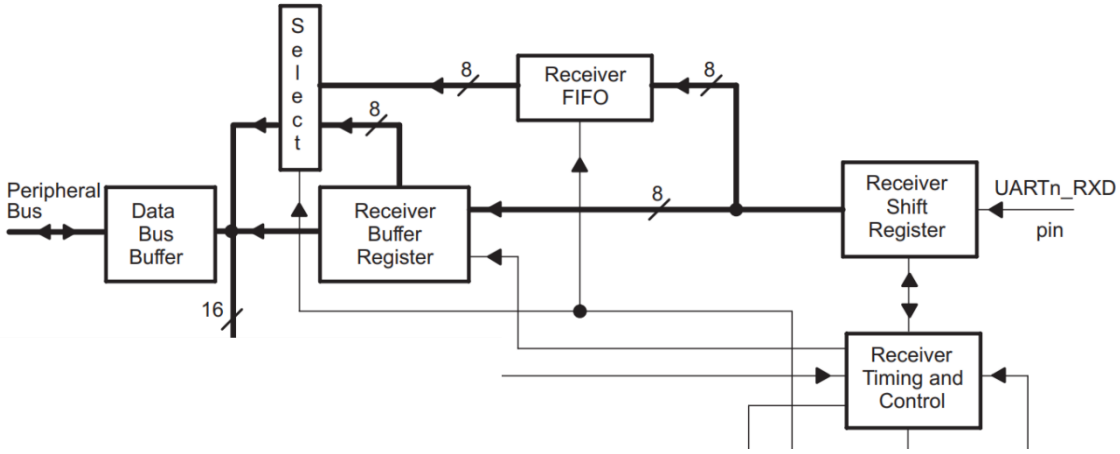


Figure 3 – KeyStone UART partial block diagram (Source: Texas Instruments)

By focusing on the key elements of the receiver, this section can be divided in four main components:

1. **Receiver Timing and Control**: will track the start and end of the transmission according to the settings and protocol specification (will produce a Frame Error in case of transmitter frequency mismatch).
2. **Receiver Shift Register**: connected to the rx data line, will accumulate the word data and parity bit, until it is ready to be moved to the buffer.
3. **Receiver Buffer Register**: buffer for the UART output data.
4. **Receiver FIFO**: alternative mode for receiving the frames. It implements a FIFO type buffer (memory array) used to store multiple frames received (The FIFO mode will not be present in the hardware developed for the project)

Considering the project requirements, some adaptations will be necessary for this overall scheme of the UART receiver. A specific validation signal must be produced at the output of the module; therefore, a custom control block must be used, checking for error conditions during the reception. The signal is then asserted for one UART clock cycle, informing that the data at the buffer is ok to be read.

A simplified diagram of the initial idea for the UART receiver is shown in Figure 4. On the Architecture description section, the final implementation will be discussed.

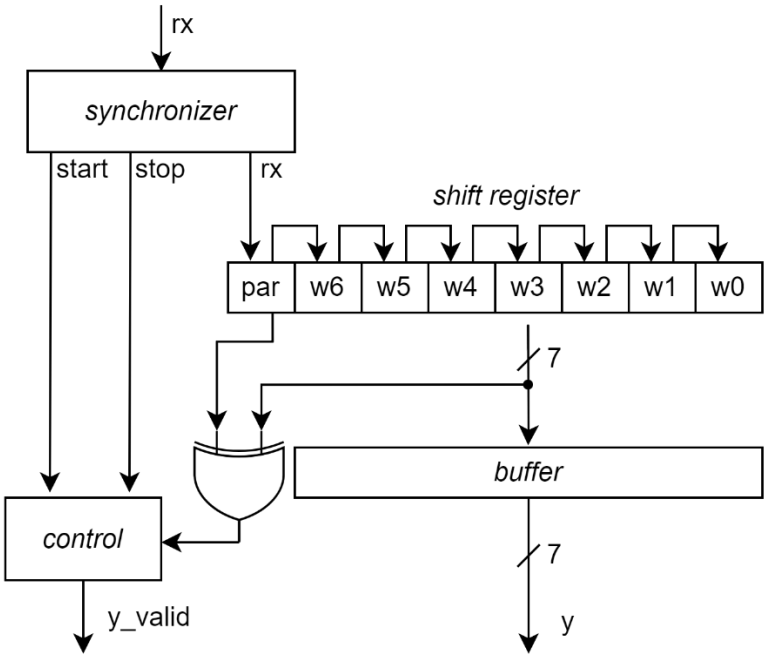


Figure 4 – UART receiver initial concept

2. Architecture description

With the UART peripheral example studied, aligned with the project requirements, a model was developed to perform the necessary operation of the protocol. The required settings for this receiver were:

1. Word size (W) = 7 bits
2. Baud rate (B) = 115200
3. Parity (P): even
4. Number of Stop bits (S) = 2
5. Oversampling rate (OS_RATE) = 8 (UART clock period ~ 1085 ns)

Within this UART module, three main components were designed to handle the protocol operations, as well as an addition independent block, used for a particular situation of the protocol (break condition). The modules will be detailed in the next sections, and the overall UART peripheral structure and internal connection were defined as:

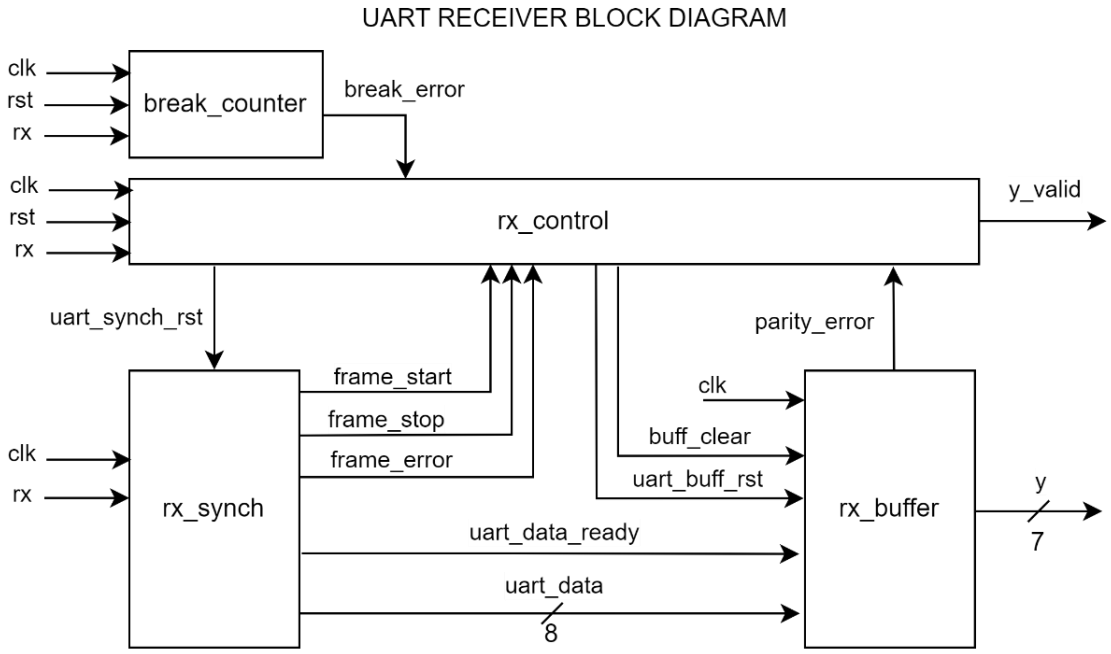


Figure 5 – UART receiver internal signals block diagram (implemented)

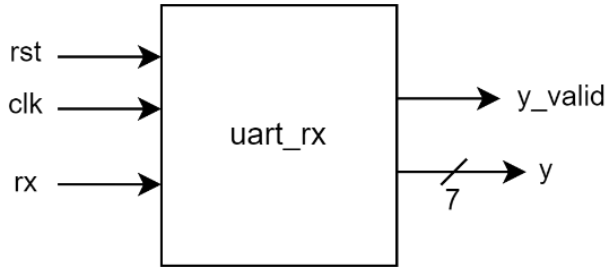


Figure 6 – UART receiver block diagram

2.1 Synchronization component (rx_synch)

The main function of the synchronization component is to read and accumulate the data from the serial line and to provide the frame synchronization signals for the UART protocol status control.

For the data reception, an internal counter based on the UART internal clock will be used to synchronize the bits at the receiver. From the design specifications, the UART clock is eight times faster than the Baud rate of the UART transmitter, therefore, the rx line value will be read at the 4^{th} cycle of every bit and copied to the output. The expected behavior in time for this process is as shown:

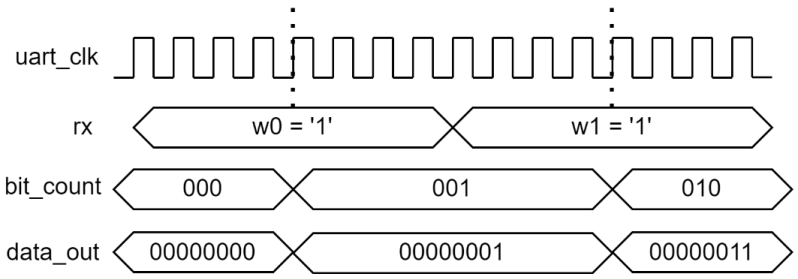


Figure 7 – UART rx data reading and accumulation

A second counter (bit count) based on the number of bits read will then be used to identify the end of the word (including the parity bit) and the stop bits.

This method of handling the input data used on the synchronizer component resembles more the operation of a demultiplexing device, rather than a proper shift register, by accumulating the data and then issuing a ready signal once the final bit of the word is received. A dedicated demux component was not introduced in the synchronizer, though the process used on the VHDL code will work similarly as one. As a reference, the demux block diagram would be like the diagram presented:

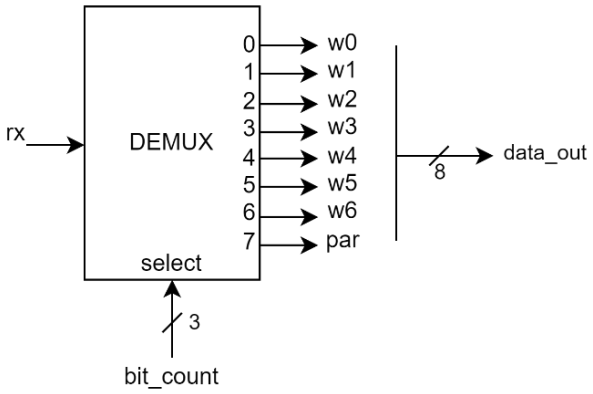


Figure 8 – Demux block diagram

This design choice is then an alternative way of converting the transmitted bits from the serial line to a parallel output, instead of the shift register approach used on the Texas Instruments device shown.

No performance benefit was observed when implementing this component as the serial-to-parallel converter. However, a noticeable difference that can be discussed is that the individual outputs of this block are kept constant during the reception of the word (after the first bit is set to the demux output, that value is no longer altered during the frame transmission, and so on for the other bits).

To complete the serial-to-parallel operation, a data ready signal then is raised once the bit counter reaches the word size plus parity bit, allowing the connected buffer to read the data output values from the component.

In addition to the data ready signal, the synchronizer component will also generate three external control signals, informing the UART control component the current state of the protocol transaction or the occurrence of a frame error. Those signals are:

1. **frame_start**: indicates the start of a frame, it is set to ‘1’ after the rx line goes down, and is kept at ‘0’ after 4 cycles (oversampling rate/2).
2. **frame_stop**: indicates when the last stop bit is detected.
3. **frame_error**: indicates when the start or stop conditions of the UART protocol are violated (usually frame errors occurs when there is a mismatch between transmitter and receiver baud rate values).

To set the established status signals a Finite State Machine was developed for the synchronization component, by using a total of 8 states, shown in the FSM diagram and block diagram:

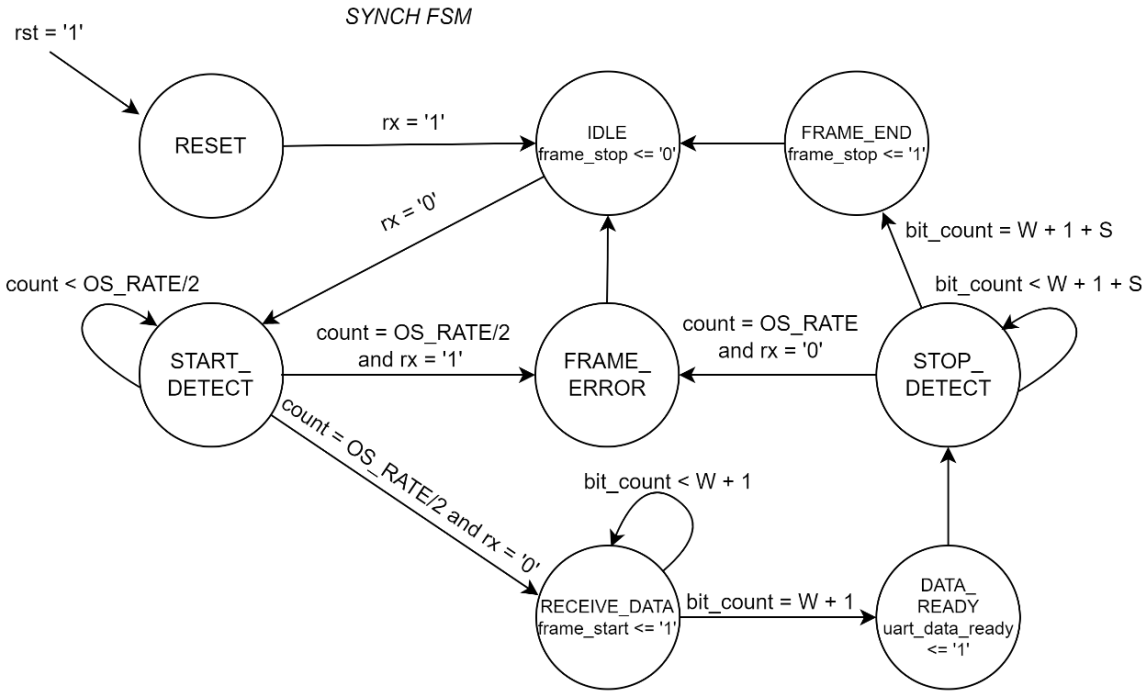


Figure 9 – Synchronizer FSM

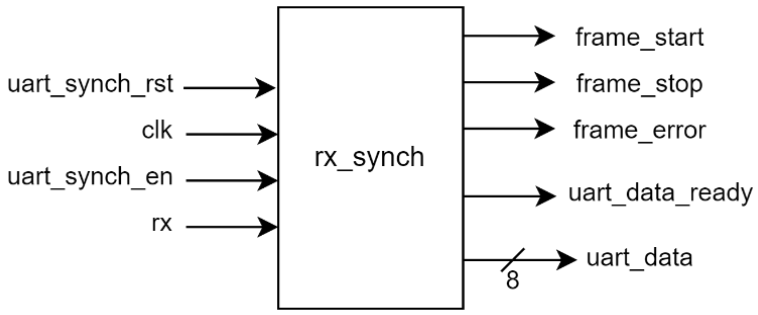


Figure 10 – Synchronizer block diagram

2.2 Buffer component (rx_buff)

As the synchronizer component converts the serial data from the tx line to a parallel output, a buffer was implemented to hold the word data as the frame is transmitted. The buffer then reads the output data of the synch block, and performs the

parity check on the received word. After the data is copied, the buffer can be read from the CPU (output ‘y’ of the UART receiver peripheral).

To inform the buffer that new data can be obtained (completing the cycle for one transmitted frame), the component was design to respond to a clear signal issued by the control module. This signal returns the buffer to a standby state, waiting for a new data ready signal from the synch block. The internal states described of the buffer component are illustrated in the FSM diagram:

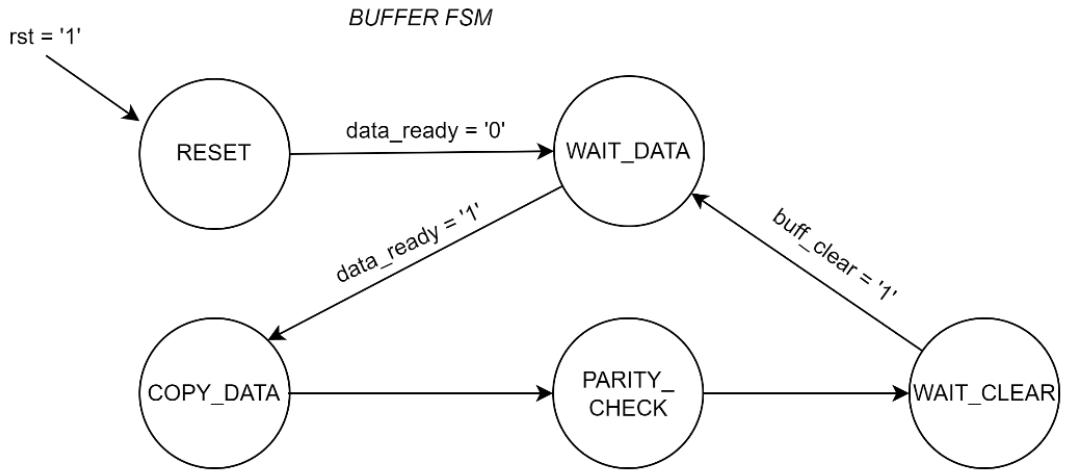


Figure 11 – Buffer FSM diagram

The envelope for this component is then represented by the block diagram:

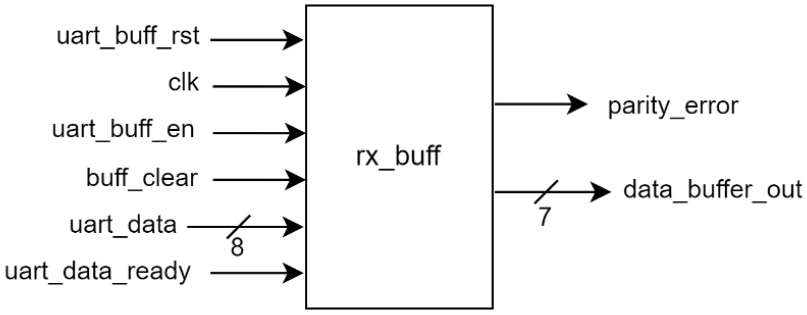


Figure 12 – Buffer block diagram

Considering that this component then must provide the parity check, a dedicated XOR gate was implemented inside this module. This auxiliary component was configured for the settings configured on this UART receiver, and has a total size of 8 bits of input (W = 7 + 1 parity). The buffer component will raise a parity error flag if the result for the XOR operation does not match the even parity UART configuration (the result must be ‘0’). The block diagram for XOR gate developed is:

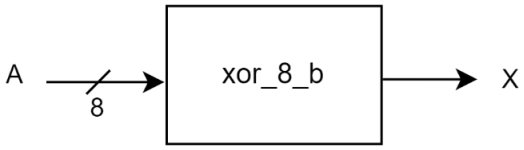


Figure 13 – 8 bit XOR gate block diagram

2.3 Control component (rx_control)

To handle the state changes and to provide the frame validation signal, a dedicated control block was implemented in the UART receiver peripheral. This component then obtains the status and error signals from the synchronizer and buffer components. Once the conditions, defined by the UART settings (matching baud rate, word size, parity, and number of stop bits) are satisfied on the received frame, the control component will set the ‘y_valid’ signal to represent that the data at the buffer is OK to be read.

Also, this control module can directly reset the internal components when an error situation occurs.

An important mention on the implemented architecture is that the control operations could also be implemented as a Hardware Abstraction Layer, when combining the peripheral to a microcontroller platform. However, for this type of external control the UART module must expose a set of error signals (as well as extra communication control signals such as Ready to Send and Clear to Send, in a master slave operation) to the external hardware. On the Appendix section, the schematics for the complete UART peripheral implementation from TI are included for reference.

For the current project requirements, the only external control signal defined was the data validation, and so the modules were developed accordingly, keeping the error and status signals internal to the UART receiver block. The implemented FSM for the described state control and the component block diagram are as following:

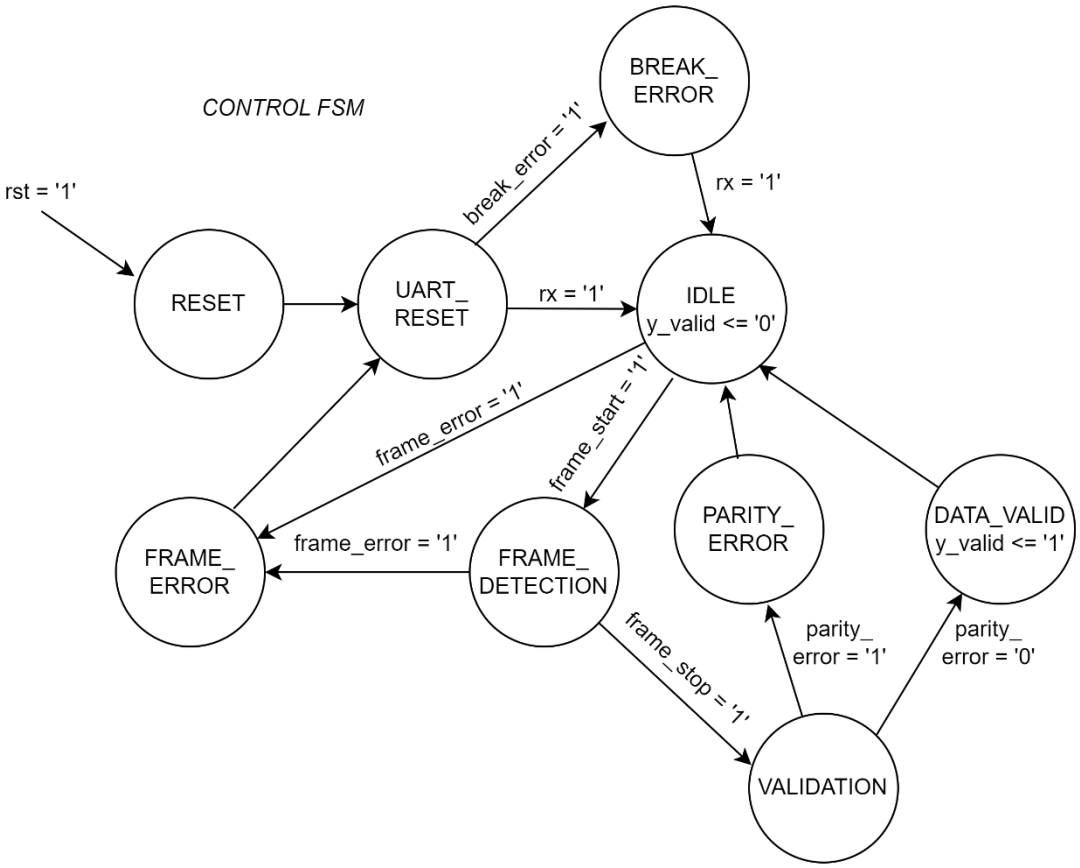


Figure 14 – Control FSM

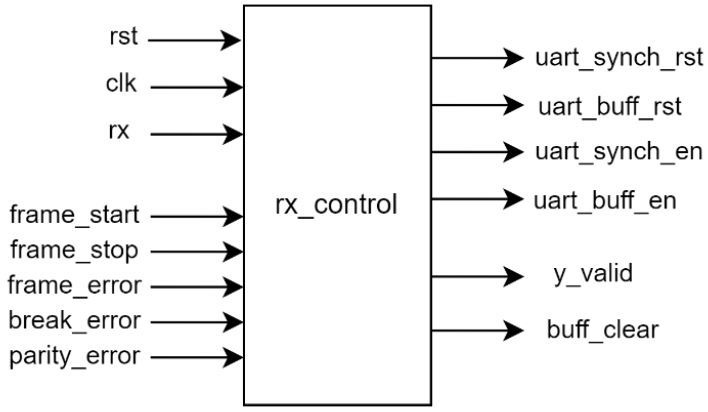


Figure 15 – Control block diagram

Noticeably, a break condition is also managed by the control module, however this state currently has no effect on the behavior of the UART receiver. More details on the break counter component will be discussed in the next session.

2.4 Break counter (break_counter)

The UART break is a special condition defined in the protocol suite, used to trigger a specific response on the peripheral, and could be used to start a new communication or reset the slave UART peripheral. Considering the project requirements, this component was introduced in the design to complement the available functions.

In spite of being added to the control FSM, the break condition does not have an impact on the current component behavior, and it is only to represent and identify the reception of the break character, which is defined by: “A Break character consists of all zeros and must persist for a minimum of 11 bit times before the next character is received.” (Source: Microchip).

The break counter, used to identify the described condition, was implemented in a dedicated component, and follows a similar behavior of the synchronizer used in this UART receiver. The internal counter then performs a bit count respecting the oversampling rate defined (8 UART clock cycles for each bit) and will trigger once the 11 consecutive zero is detected. If a logic ‘1’ bit is detected, then the counter is automatically sent back to the IDLE state.

If the break condition is met, then the component will raise a break error signal, informing the control block, and it will automatically reset its internal counters (handled In the AUTO_RESET state). The FSM for this module and its block diagram is defined as:

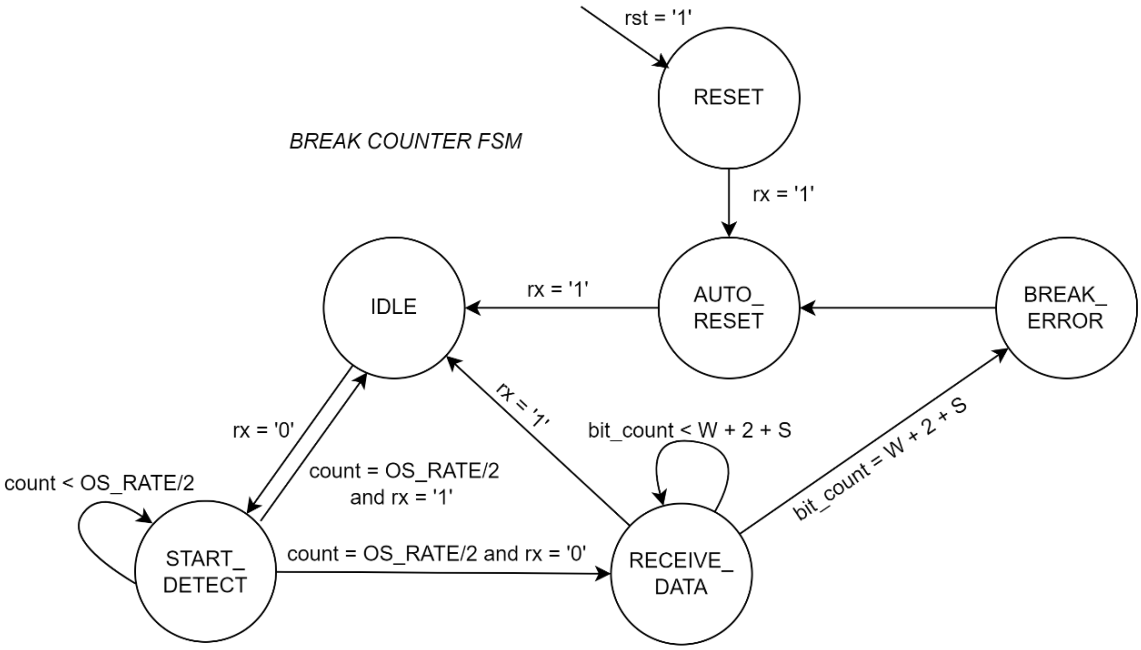


Figure 16 – Break counter FSM

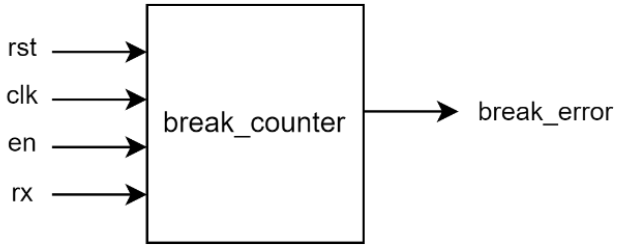


Figure 17 – Break counter block diagram

3. Testing and verification

To test and verify the components correct behavior, a testbench VHDL component was designed for each block, as well as for the complete UART receiver. In this section, the results from the Modelsim simulations of the modules will be used to demonstrate the proper operation and state change on every module. With this demonstration, it will be possible to confirm the configurated settings and design specifications of the developed peripheral.

The testbench configurations will also provide scenarios for the error handling situations, demonstrating the detection and responses for each error condition. Those conditions, followed the UART protocol definitions, and were classified as:

1. **Frame error**: inconsistency detected at the start of end of the frame (issued by the synch block).
2. **Parity error**: the parity of the transmitted word is not even (issued by the buffer block).
3. **Break error**: special condition detected and issued by the break counter, when a total of 11 zeros are sent in the rx line.

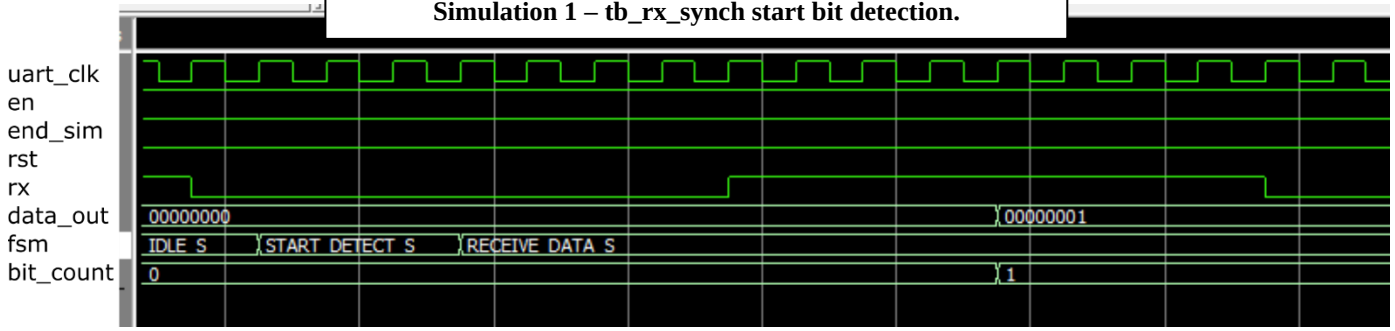
3.1 Synchronizer testbench (tb_rx_synch)

The testbench for the synchronizer component will be used to demonstrate three main conditions and transition. The first test shown (label: Simulation 1) shows when the start bit is sent, initiating the frame transmission. The simulated rx line then goes from high to low, triggering the FSM of the synch module to start the internal counter. After the 4^{th} cycle, the input is checked again and if a zero is still detected, the state is changed to RECEIVE_DATA. At this stage, the component will start to set its output, according to the values detected at the input at every 8 counts.

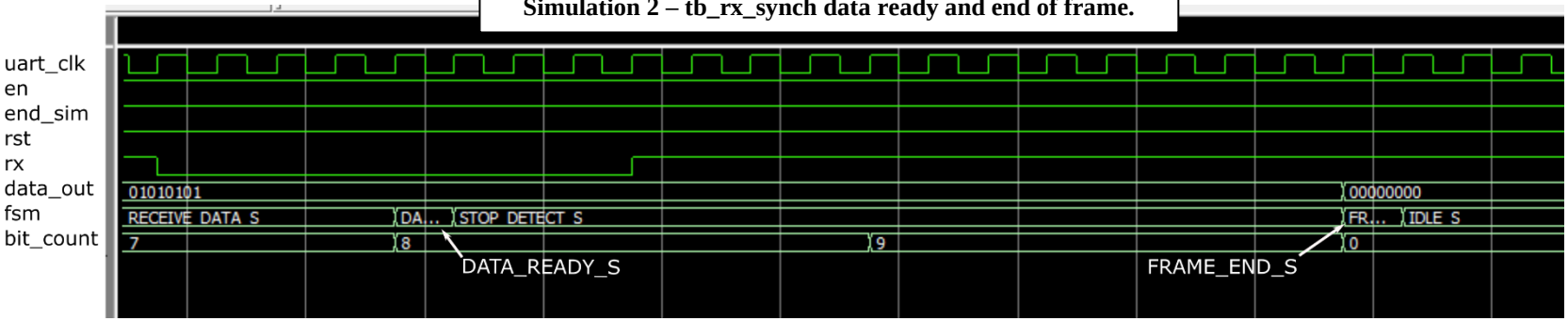
The second demonstration (label: Simulation 2) shows the end of the transmitted word and the stop bit counting. In the test, after the bit count reaches W + 1, the synchronizer rises the data_ready signal, and start the stop bit detection. When the last stop bit is detected, the FSM return to IDLE state.

The third test (label: Simulation 3) will then show a scenario where the rx line goes to zero at the end of the frame, resulting in a Frame error condition. At this point the FSM of the synch block is locked in the FRAME_ERROR state until a reset is performed.

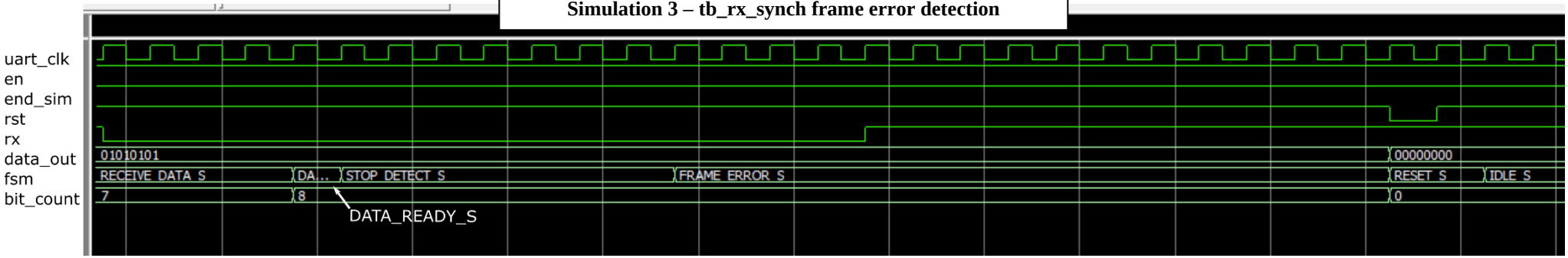
Simulation 1 – tb_rx_synch start bit detection.



Simulation 2 – tb_rx_synch data ready and end of frame.



Simulation 3 – tb_rx_synch frame error detection

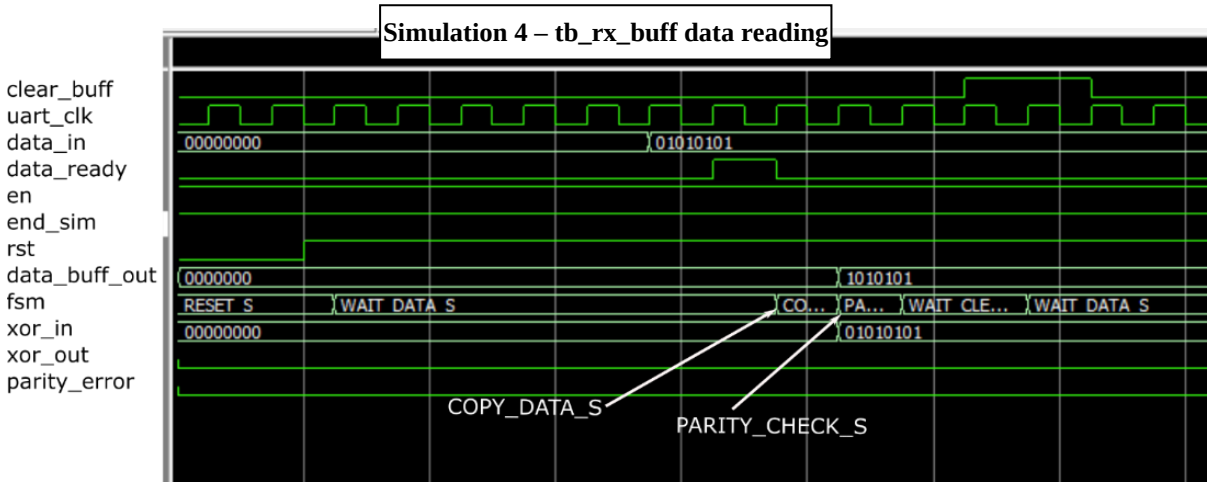


3.2 Buffer testbench (tb_rx_buff)

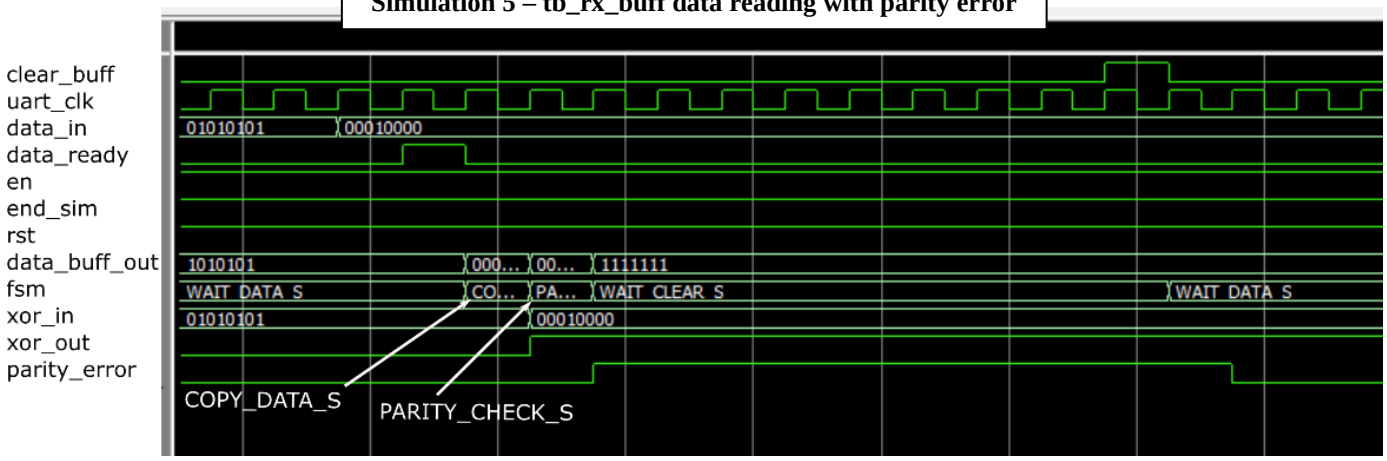
For the buffer component, the testbench developed demonstrates the overall simple operation that this module should perform. From the initial waiting state, the data ready triggers the internal FSM to move to the copy state, where the data from the output of the synchronizer block is copied to the buffer and inserted at the input of the XOR gate. The parity assertion is then done, and if it does not match the even parity configuration, the buffer block then raises the parity error signal.

After the parity check, the component will wait to be cleared by the control module, returning to the waiting state where the data ready signal is at low. The two scenarios shown in the simulations then demonstrates the complete process of the buffer component, one when there is no parity error (label: Simulation 4), and one where the word data is inconsistent (label: Simulation 5). When a parity error is detected, the output of the buffer will be set to all '1' values (0x7f).

Simulation 4 – tb_rx_buff data reading



Simulation 5 – tb_rx_buff data reading with parity error



3.3 UART testbench (tb_rx_uart)

The last testbench presented will show the complete operation of the UART receiver developed. In this simulation, the state transitions of the control module will be highlighted, as for the other blocks, the individual tests already demonstrate their individual behavior.

On Simulation 6 and Simulation 7 it is shown the operation of the peripheral at the beginning and at the end of the transmitted frame, respectively. With the rx line in idle, the first state change is noticed as the line goes to GND and the synch module detects it. After the 4^{th} cycle, the start bit is detected, informing the control block with the start fame signal. With this the FSM of the control component is set to FRAME_DETECTION.

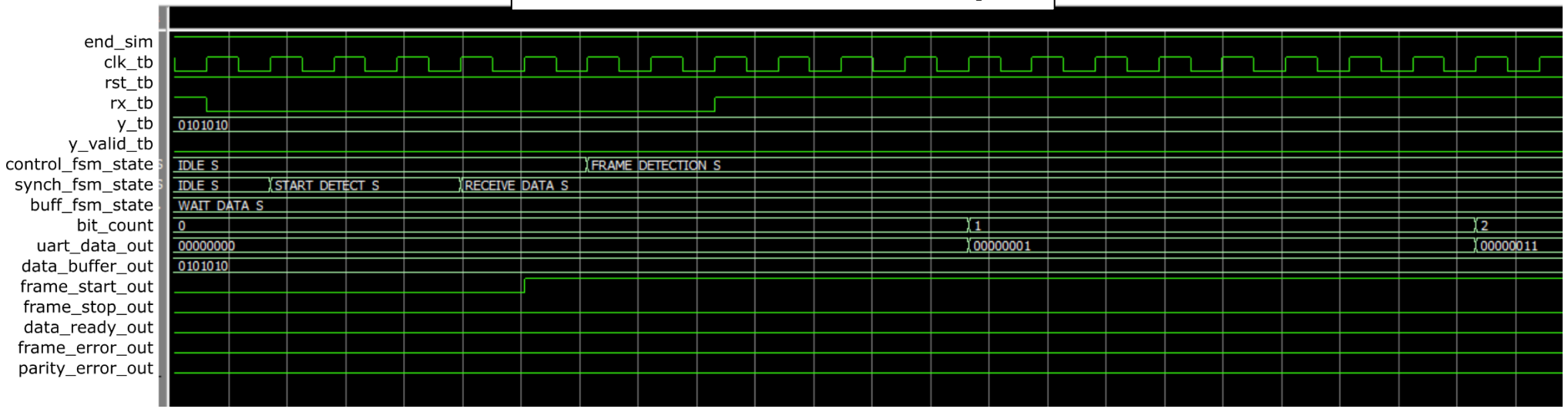
Meanwhile, the synch block is accumulating the data from the serial line, until the last bit of the word is received. When the last data bit is read, the signal data ready is raised (and the start signal is cleared), allowing the buffer to copy the parallel data. With this, the synch module goes to the end frame state, now reading the stop bits. At the same time, the buffer will check the data parity (on the example, the 7 bit word sent is 0x55, with an accurate parity bit with value '0'). As the correct value for the parity bit is set, there is no parity error raised.

Finally, the synch module counts the 2 stop bits sent on the serial line, raising the frame stop signal. With this transition, the control FSM then is triggered, and changes to the validation state. As no error flag is raised, the transaction is completed and the 'y_valid' signal is raised for 1UART clock cycle. With the valid signal raised, the control also raises the buffer clear signal, making the component available for the next reading. After the end of the frame all components return to the IDLE state.

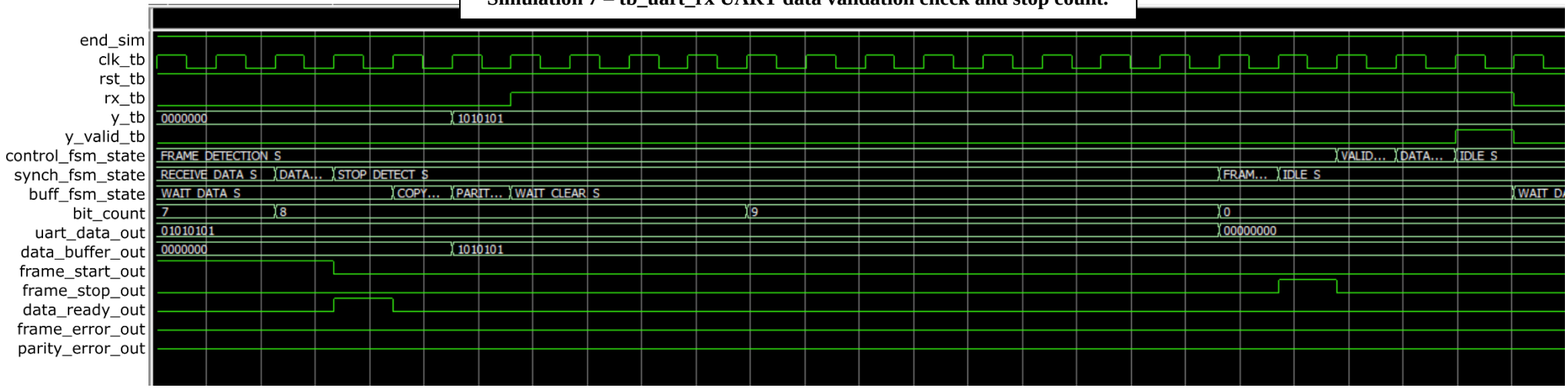
To complement the tests and to demonstrate the error handling behavior, four scenarios will be shown, with the following settings:

1. **Simulation 8**: the word sent contains a wrong value for the parity bit (W = 0x55 with parity bit = '1').
2. **Simulation 9**: two characters are transmitted in the serial line with a baud rate higher than 115200 (the bit period is equivalent to 6 UART clock cycles, B = 153600).
3. **Simulation 10**: two characters are transmitted in the serial line with a baud rate lower than 115200 (the bit period is equivalent to 10 UART clock cycles, B = 92160).
4. **Simulation 11**: a sequence of 11 zeros is transmitted on the line (break character).

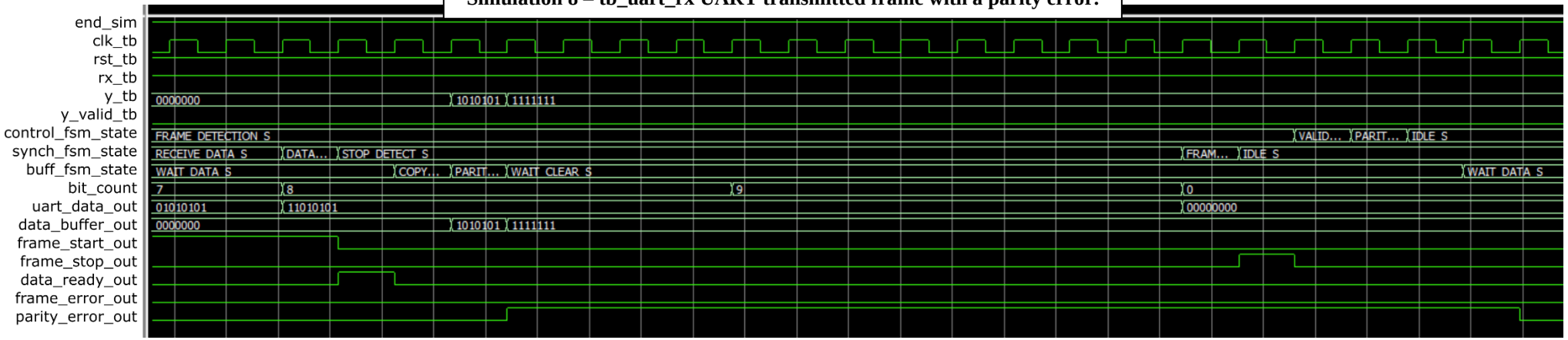
Simulation 6 – tb_uart_rx UART frame start reception.



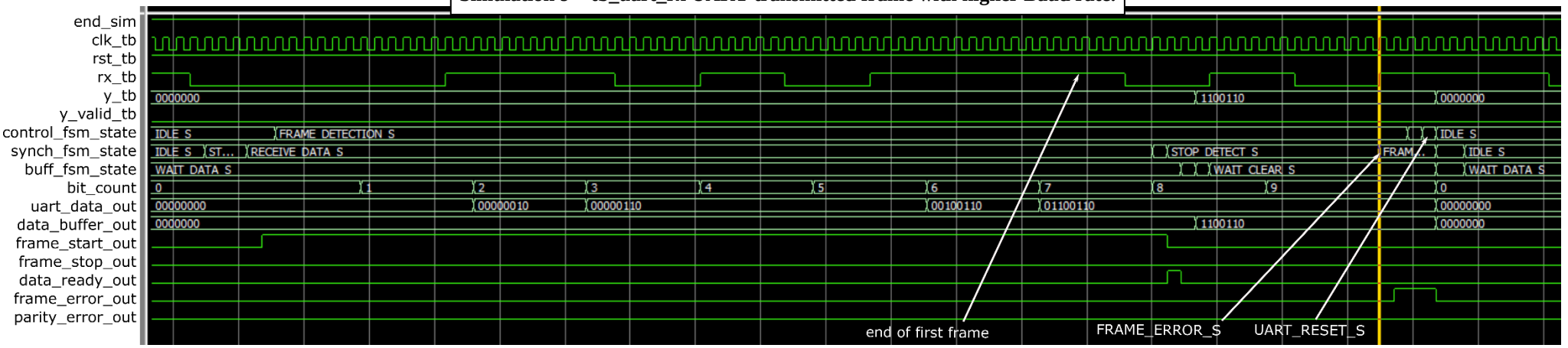
Simulation 7 – tb_uart_rx UART data validation check and stop count.



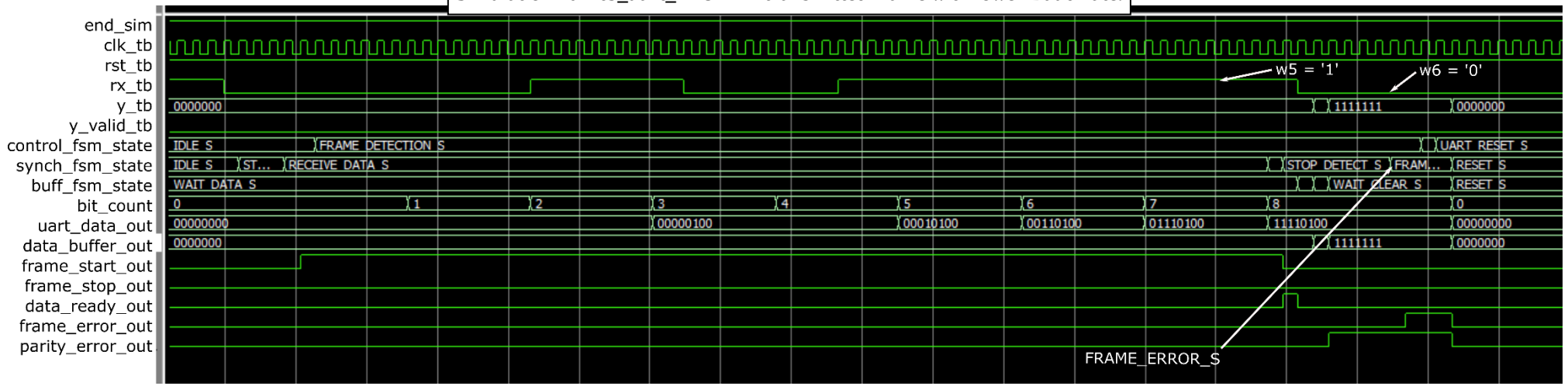
Simulation 8 – tb_uart_rx UART transmitted frame with a parity error.



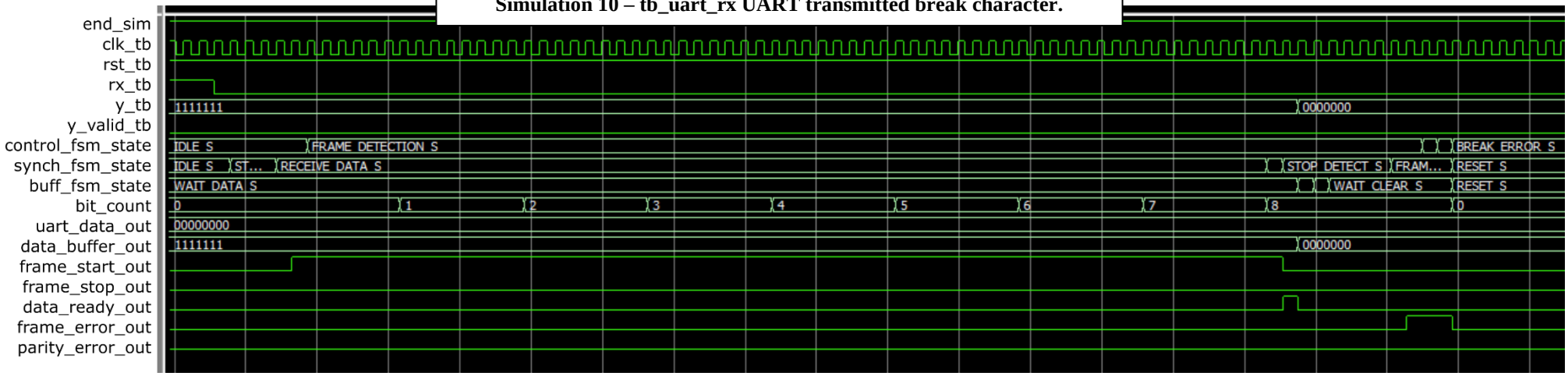
Simulation 9 – tb_uart_rx UART transmitted frame with higher Baud rate.



Simulation 10 – tb_uart_rx UART transmitted frame with lower Baud rate.



Simulation 10 – tb_uart_rx UART transmitted break character.



4. Synthesis and implementation

To synthesize the developed peripheral component in the target device (FPGA based board, Zynq 7000 APSoC), the Xilinx Vivado tool will be used. This will allow the evaluation of the component when implement in hardware, considering then the resource usage, timing conditions and power consumption.

With the source files added to the Vivado project, an initial synthesis design was produced. No critical errors were found in the design, however one warning was produced, the information was: "[Synth 8-327] inferring latch for variable" for the **synch_enable_out** signal. That specific signal was the enable control for the synchronizer block, and it is an output of the control module. This warning was produced as on the control component code, the internal process was design with the following portion of VHDL code (rx_control.vhd – deprecated version):

```
p_rx_control_fsm : process(clk, rst)
   begin
       if rst = '0' then -- reset active low
           rx_control_fsm_state <= RESET_S;
           y_valid_out      <= '0';
           synch_enable_out <= '1';
           synch_reset_out  <= '0';
           buff_enable_out  <= '1';
           buff_reset_out   <= '0';
           buff_clear_out   <= '0';

       elsif rising_edge(clk) then
           case rx_control_fsm_state is
               when RESET_S =>
                 …
```

Noticeably, the signal referred at the warning was being set at the asynchronous reset section of the control block. However, this signal is connected to the **en** input at the synchronizer component, and its assertion is clock triggered, as shown in the portion of VHDL code (rx_synch.vhd):

```
p_rx_synch_fsm : process(clk, rst)
variable count : integer := 0;
begin
    if rst = '0' then -- reset active low
  …
    elsif rising_edge(clk) then
        if en = '1' then
            case rx_synch_fsm_state is
                when RESET_S =>
                  …
```

By moving the signal from the asynchronous reset portion of the control process to the IDLE state of the clock triggered FSM, the warning was cleared. With this the component was successfully synthesized, and the schematic produced by Vivado is as follows:

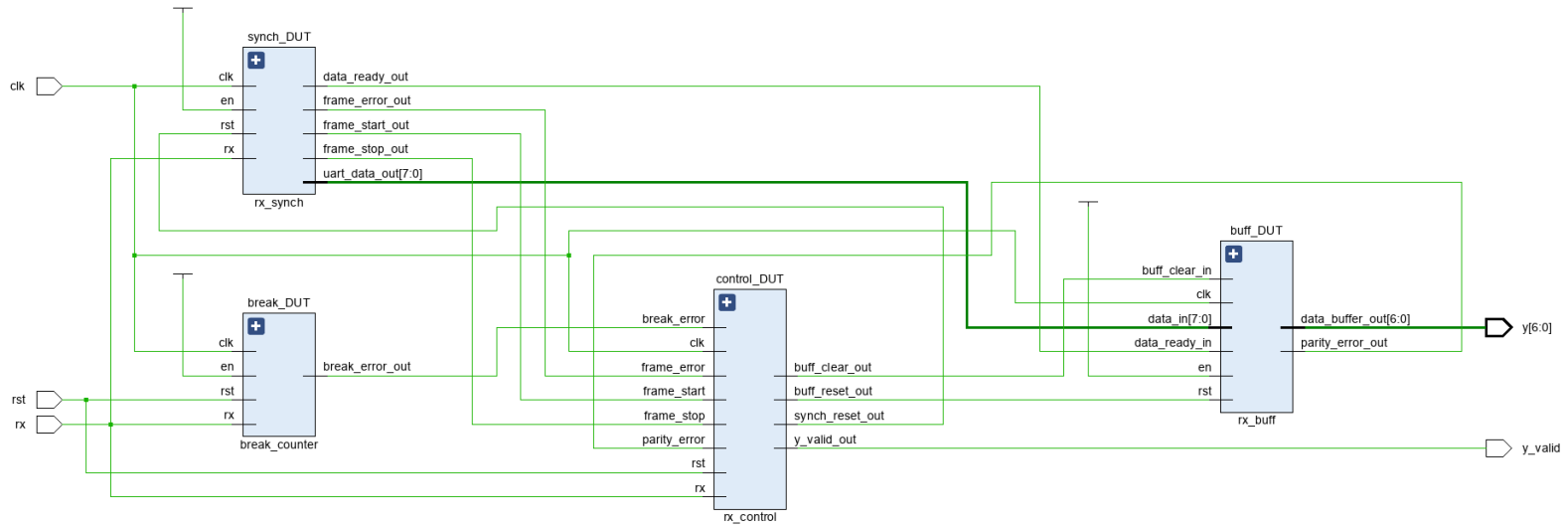


Figure 18 – Vivado synthesis schematic

For the timing and power evaluation, a constraint file was create for this design to set the value for the UART input clock. The period used for the standard design followed the project requirements for the stipulated data rate of 115200 bits/s. As the oversampling rate has a value of 8, the clock period for the input clock is then set to 1085,07 ns (duty cycle at 50%). The results for Utilization, Timing were as follows:

Utilization Post-Synthesis | **Post-Implementation**

Graph | **Table**

Resource	Utilization	Available	Utilization %
LUT	146	17600	0.83
FF	117	35200	0.33
IO	11	100	11.00
BUFG	1	32	3.13

Figure 19 – Vivado Implementation: utilization report (B = 115200)

Design Timing Summary

Setup		Hold		Pulse Width	
Worst Negative Slack (WNS):	1077.945 ns	Worst Hold Slack (WHS):	0.139 ns	Worst Pulse Width Slack (WPWS):	542.035 ns
Total Negative Slack (TNS):	0.000 ns	Total Hold Slack (THS):	0.000 ns	Total Pulse Width Negative Slack (TPWS):	0.000 ns
Number of Failing Endpoints:	0	Number of Failing Endpoints:	0	Number of Failing Endpoints:	0
Total Number of Endpoints:	279	Total Number of Endpoints:	279	Total Number of Endpoints:	118

All user specified timing constraints are met.

Figure 20 – Vivado Implementation: timing summary (B = 115200)

From the implemented design, the worst path estimated was sufficient to provide a correct signal assertion. The clock period used in this UART peripheral was approximately 1085 ns and the necessary time to a particular logic to be completed in this design is much less than the period, as the Slack equation shows:

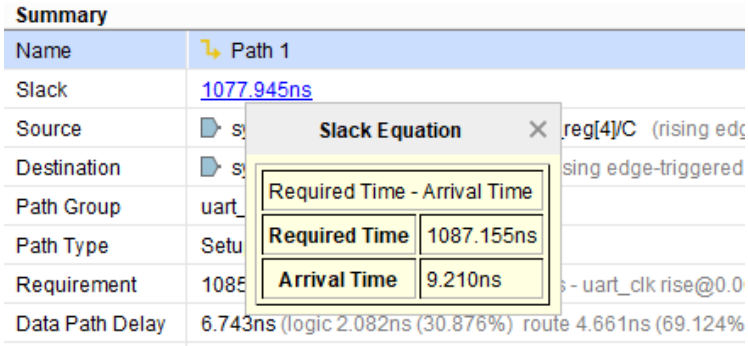
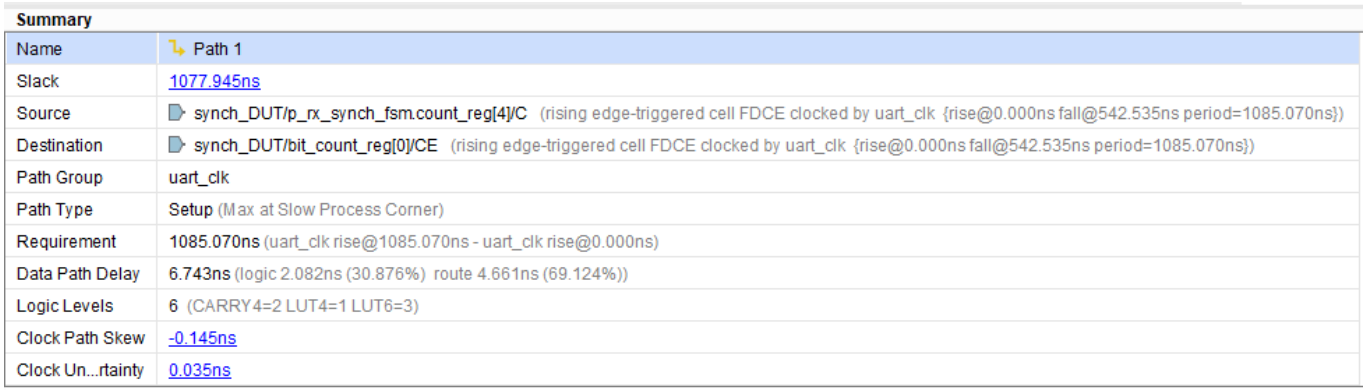
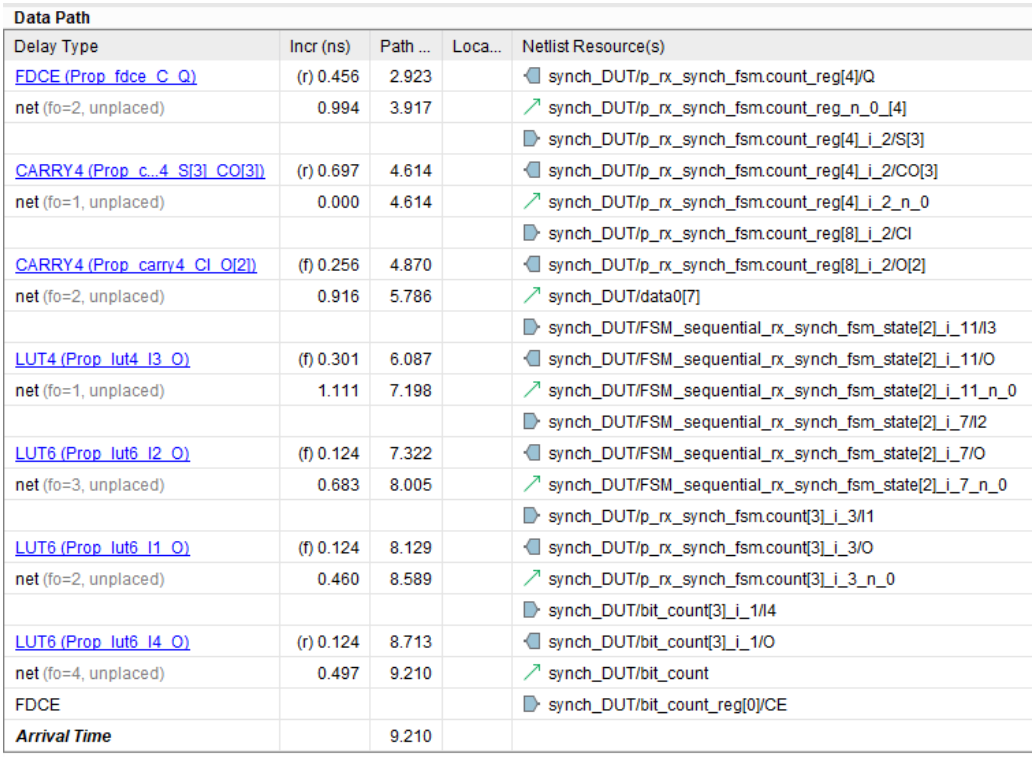


Figure 21 – Vivado slack calculation at worst path (B = 115200)



Summary	
Name	Path 1
Slack	1077.945ns
Source	synch_DUT/p_rx_synch_fsm.count_reg[4]/C (rising edge-triggered cell FDCE clocked by uart_clk {rise@0.000ns fall@542.535ns period=1085.070ns})
Destination	synch_DUT/bit_count_reg[0]/CE (rising edge-triggered cell FDCE clocked by uart_clk {rise@0.000ns fall@542.535ns period=1085.070ns})
Path Group	uart_clk
Path Type	Setup (Max at Slow Process Corner)
Requirement	1085.070ns (uart_clk rise@1085.070ns - uart_clk rise@0.000ns)
Data Path Delay	6.743ns (logic 2.082ns (30.876%) route 4.661ns (69.124%))
Logic Levels	6 (CARRY4=2 LUT4=1 LUT6=3)
Clock Path Skew	-0.145ns
Clock Un...rtainty	0.035ns

Figure 22 – Vivado Worst negative slack path (B = 115200)



Data Path				
Delay Type	Incr (ns)	Path ...	Loca...	Netlist Resource(s)
FDCE (Prop_fdce_C_Q)	(r) 0.456	2.923		synch_DUT/p_rx_synch_fsm.count_reg[4]/Q
net (fo=2, unplaced)	0.994	3.917		synch_DUT/p_rx_synch_fsm.count_reg_n_0_[4]
				synch_DUT/p_rx_synch_fsm.count_reg[4]_i_2/S[3]
CARRY4 (Prop_c...4_S[3]_CO[3])	(r) 0.697	4.614		synch_DUT/p_rx_synch_fsm.count_reg[4]_i_2/CO[3]
net (fo=1, unplaced)	0.000	4.614		synch_DUT/p_rx_synch_fsm.count_reg[4]_i_2_n_0
				synch_DUT/p_rx_synch_fsm.count_reg[8]_i_2/CI
CARRY4 (Prop_carry4_CI_O[2])	(f) 0.256	4.870		synch_DUT/p_rx_synch_fsm.count_reg[8]_i_2/O[2]
net (fo=2, unplaced)	0.916	5.786		synch_DUT/data0[7]
				synch_DUT/FSM_sequential_rx_synch_fsm_state[2]_i_11/I3
LUT4 (Prop_lut4_I3_O)	(f) 0.301	6.087		synch_DUT/FSM_sequential_rx_synch_fsm_state[2]_i_11/O
net (fo=1, unplaced)	1.111	7.198		synch_DUT/FSM_sequential_rx_synch_fsm_state[2]_i_11_n_0
				synch_DUT/FSM_sequential_rx_synch_fsm_state[2]_i_7/I2
LUT6 (Prop_lut6_I2_O)	(f) 0.124	7.322		synch_DUT/FSM_sequential_rx_synch_fsm_state[2]_i_7/O
net (fo=3, unplaced)	0.683	8.005		synch_DUT/FSM_sequential_rx_synch_fsm_state[2]_i_7_n_0
				synch_DUT/p_rx_synch_fsm.count[3]_i_3/I1
LUT6 (Prop_lut6_I1_O)	(f) 0.124	8.129		synch_DUT/p_rx_synch_fsm.count[3]_i_3/O
net (fo=2, unplaced)	0.460	8.589		synch_DUT/p_rx_synch_fsm.count[3]_i_3_n_0
				synch_DUT/bit_count[3]_i_1/I4
LUT6 (Prop_lut6_I4_O)	(r) 0.124	8.713		synch_DUT/bit_count[3]_i_1/O
net (fo=4, unplaced)	0.497	9.210		synch_DUT/bit_count
FDCE				synch_DUT/bit_count_reg[0]/CE
Arrival Time		9.210		

Figure 23 – Vivado data path at worst negative slack (B = 115200)

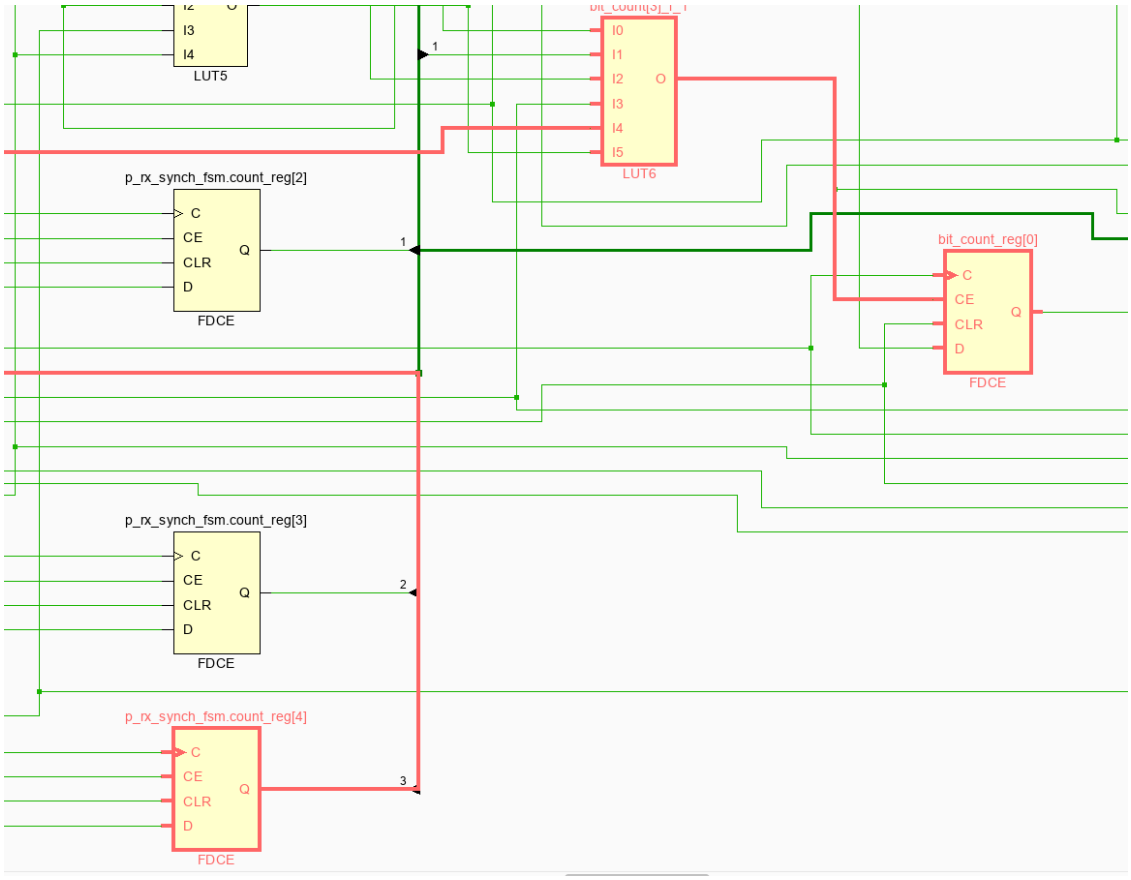


Figure 24 – Vivado zoom in WNS data path at synchronizer schematic (B = 115200)

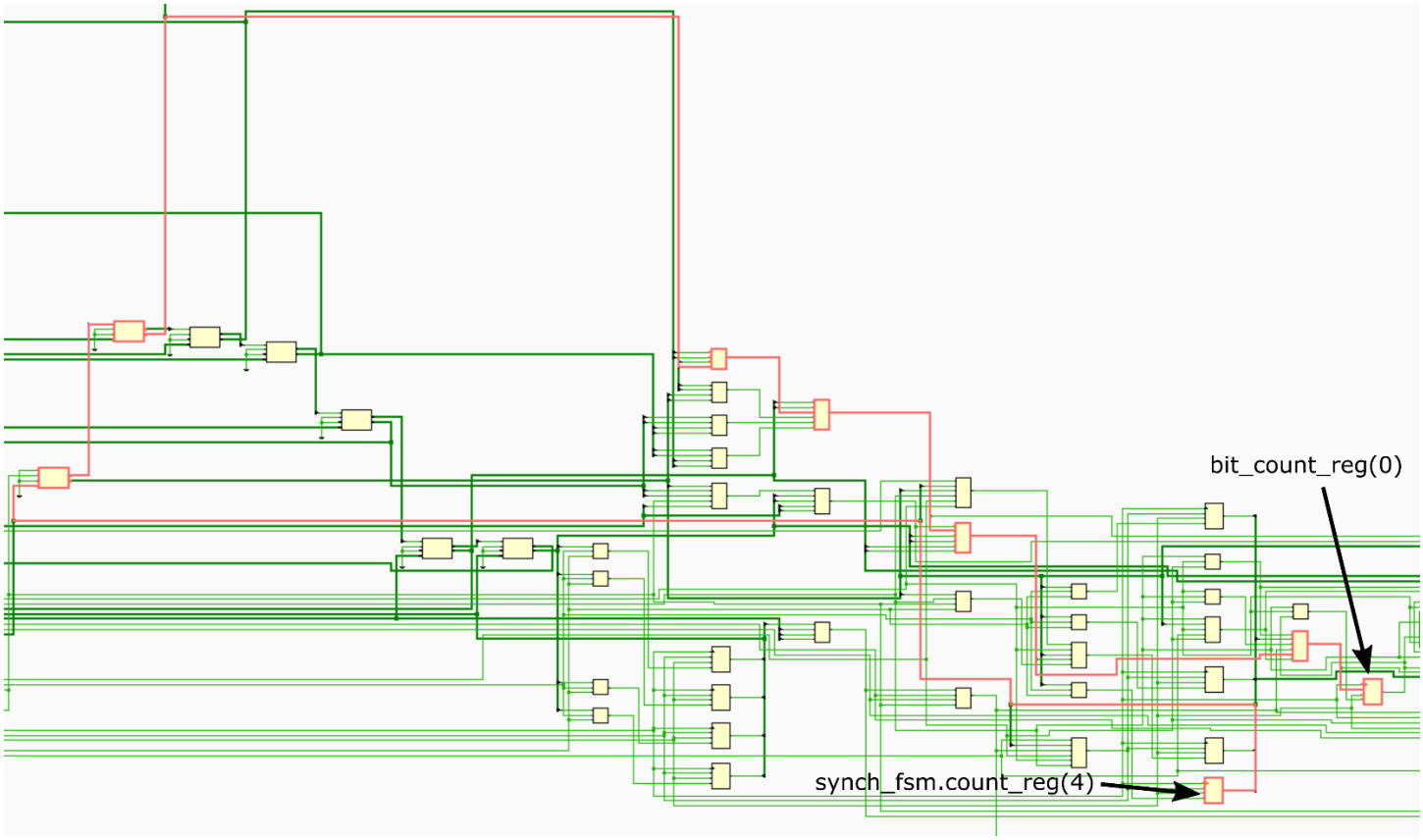


Figure 25 – Vivado WNS data path highlighted at synchronizer schematic (B = 115200)

The Hold time worst path was also evaluated in the design. No constraints were violated in the final implementation, as the report shows:

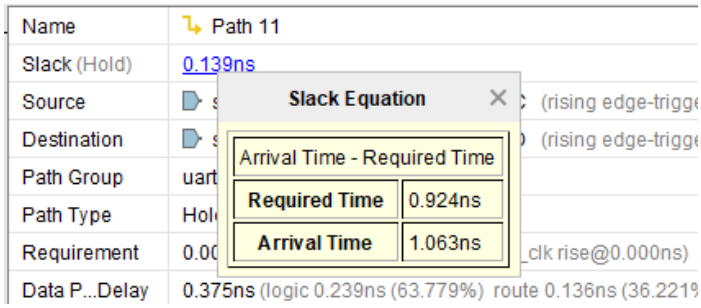
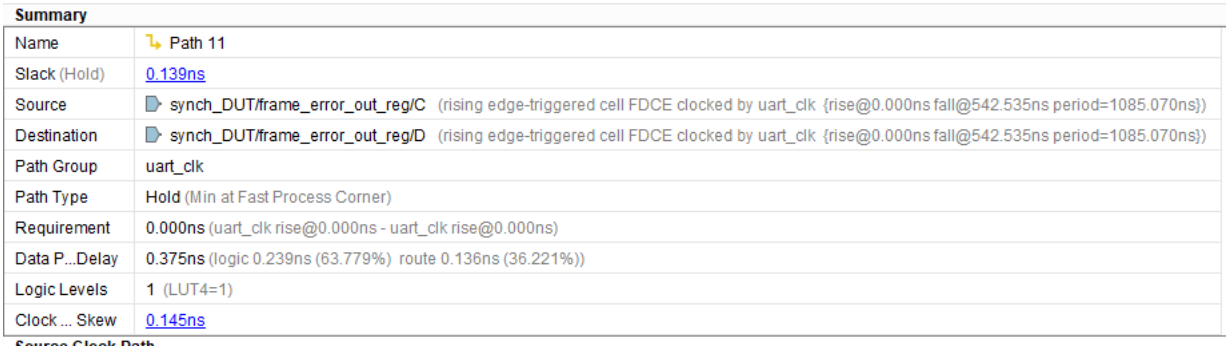


Figure 25 – Vivado slack (Hold time) calculation at worst path (B = 115200)



Summary	
Name	Path 11
Slack (Hold)	0.139ns
Source	synch_DUT/frame_error_out_reg/C (rising edge-triggered cell FDCE clocked by uart_clk {rise@0.000ns fall@542.535ns period=1085.070ns})
Destination	synch_DUT/frame_error_out_reg/D (rising edge-triggered cell FDCE clocked by uart_clk {rise@0.000ns fall@542.535ns period=1085.070ns})
Path Group	uart_clk
Path Type	Hold (Min at Fast Process Corner)
Requirement	0.000ns (uart_clk rise@0.000ns - uart_clk rise@0.000ns)
Data P...Delay	0.375ns (logic 0.239ns (63.779%) route 0.136ns (36.221%))
Logic Levels	1 (LUT4=1)
Clock ... Skew	0.145ns

Figure 26 – Vivado slack (Hold time) calculation at worst path (B = 115200)

For the power consumption report, the device produced has a relatively low dissipated power (91 mW), as most of the energy used is considered as “Device static”.

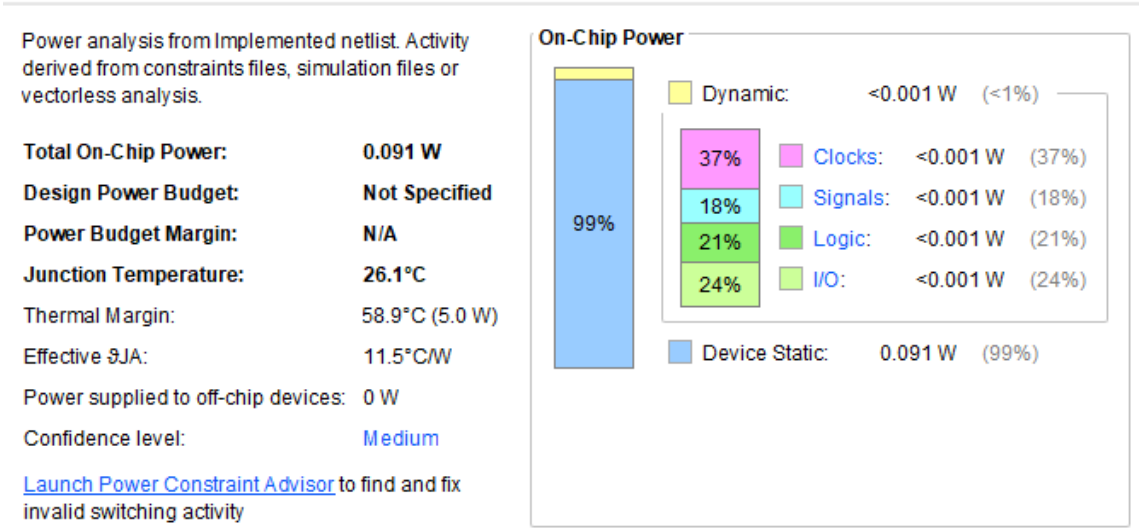


Figure 27 – Vivado Implementation: power report (B = 115200)

The final test for the UART peripheral implemented was done by increasing the input clock value, allowing an evaluation for higher data rates. The FTDI chip was used as a reference for a possible use for this UART peripheral. This component, widely used as a USB converter to UART, operates at a maximum Baud rate of 3Mbps in the UART TX port. The UART receiver developed was then fed with an 80 MHz clock, allowing for at reception over 3 times higher than the maximum of the FTDI transmission capacity (considering an oversampling rate of 8 cycles). The results showed that even with these conditions, the timing requirements were satisfied, as the reports show:

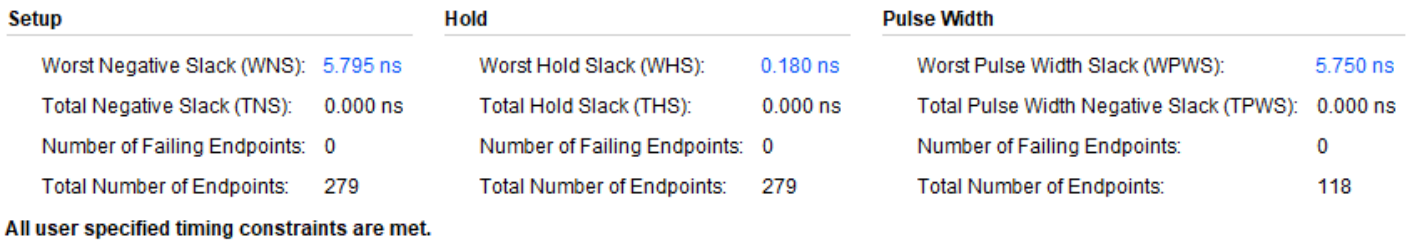


Figure 28 – Vivado Implementation: timing summary (B = 10Mbps)

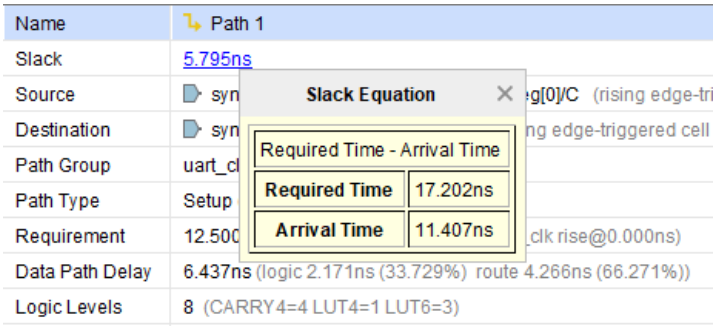


Figure 29 – Vivado slack calculation at worst path (B = 10Mbps)

The power consumption report showed a slight increase in the dissipated power on these conditions (94 mW):

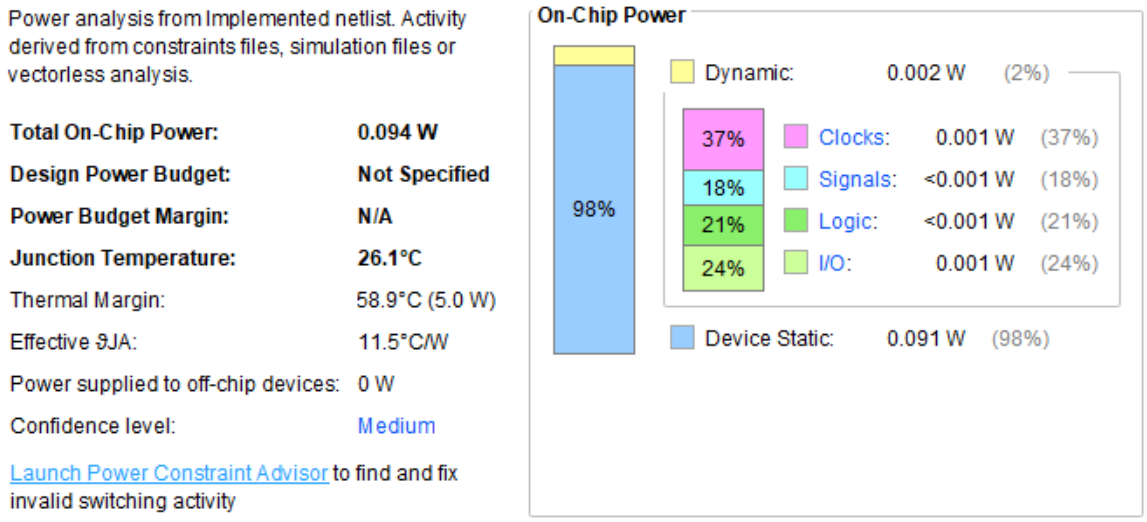


Figure 30 – Vivado Implementation: power report (B = 10Mbps)

5. Conclusions

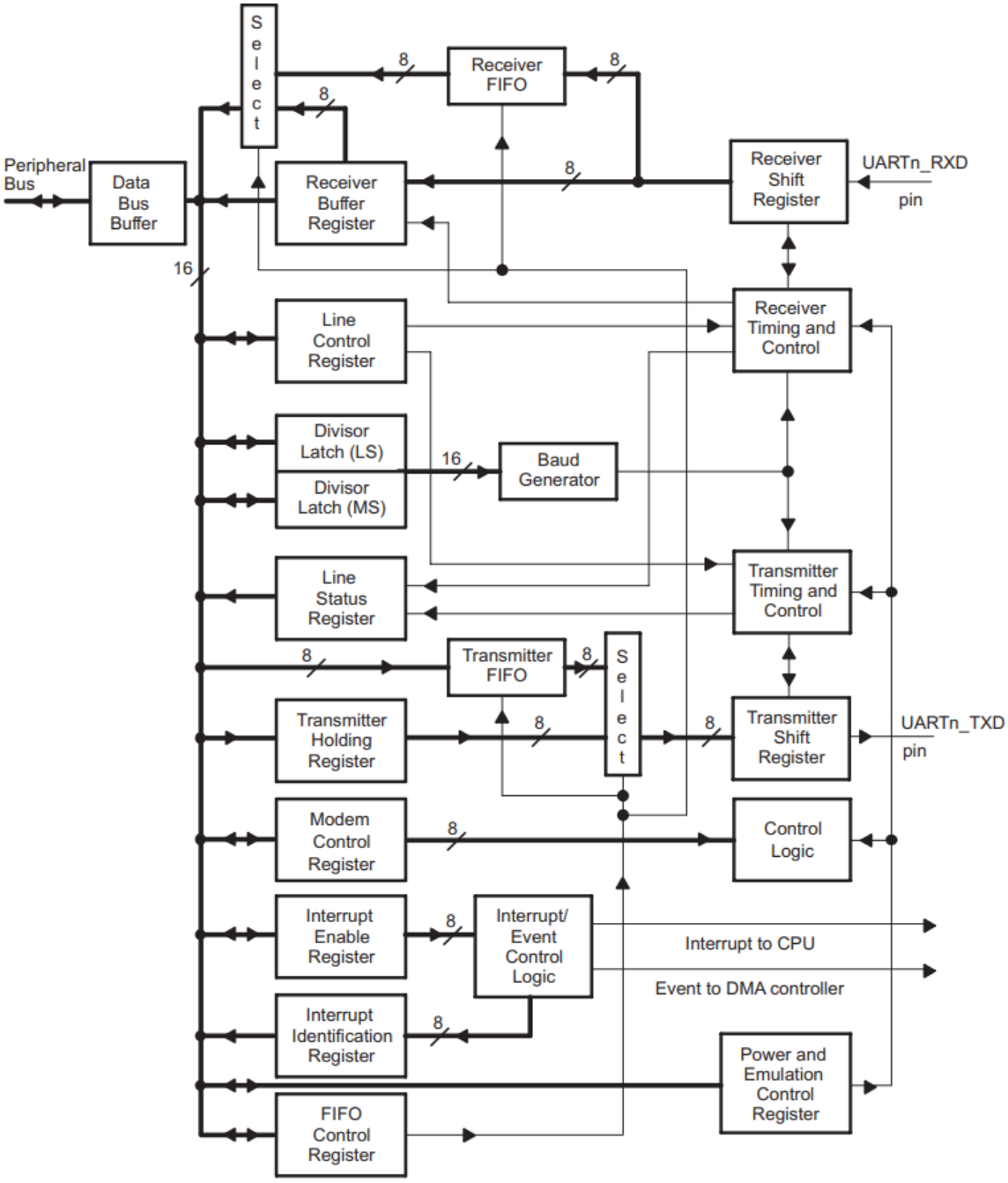
With the results from the Modelsim simulations, complemented by the synthesis and implementation design using Vivado, it is possible to conclude that the UART receiver peripheral develop was operation with success. The overall architecture followed an established device as reference (KeyStone UART peripheral), with the proper adaptation to fulfill the Project requirements.

For a more sophisticated implementation, the error signals could be exposed (usually those outputs are connected to the interruption signals of the MCU), as well as a configuration register. This would allow for the complete external control of the internal UART protocol settings, such as the word size, the number of stop bits and even the data rate (a PLL component could be inserted in the design to match the sampling frequency with the desired data rate). By exposing the error signals, the component would also be more flexible, as the information of the protocol would be passed directly to the microcontroller, allowing an HAL to manage the operation of the peripheral (custom policies and timeout conditions could be used for handling the error conditions, as it is generally the user who decides what to do in those situations).

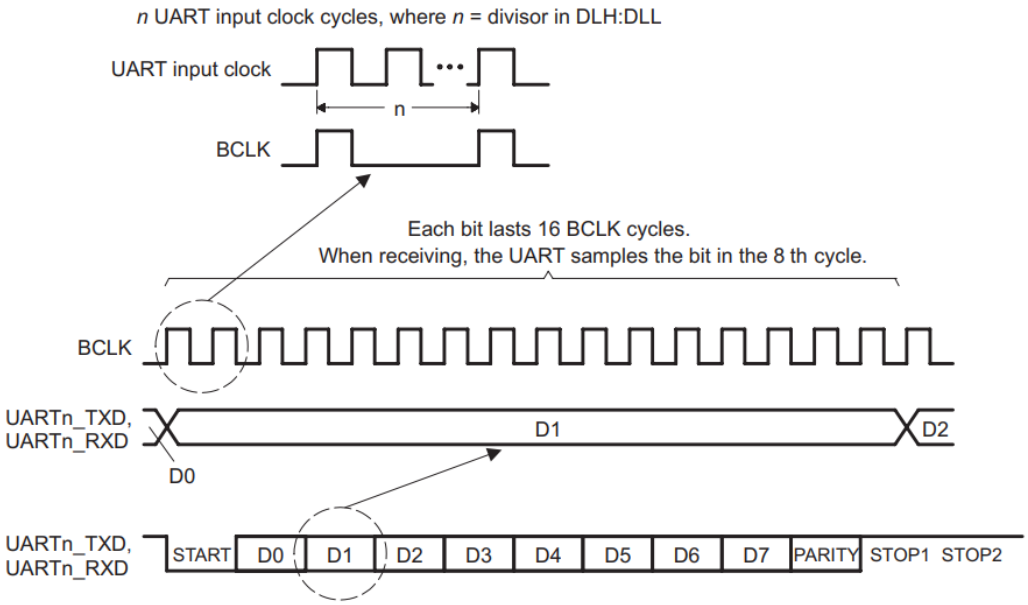
The final steps for the actual deployment and testing on the FPGA board would require a Pin planning, as well as the bitstream file generation for the component. Also, an addition module could be implemented in the FPGA, simulating a source device that would transmit the characters to the receiver.

Appendix

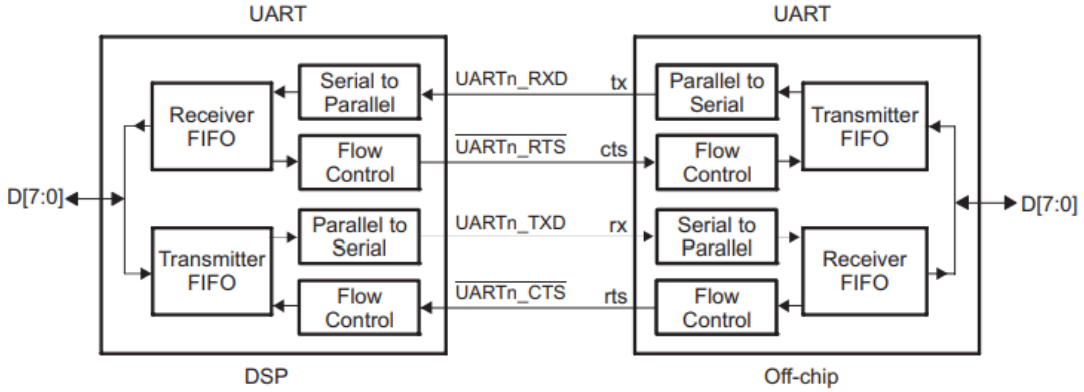
TI Keystone UART Peripheral: Functional Block Diagram:



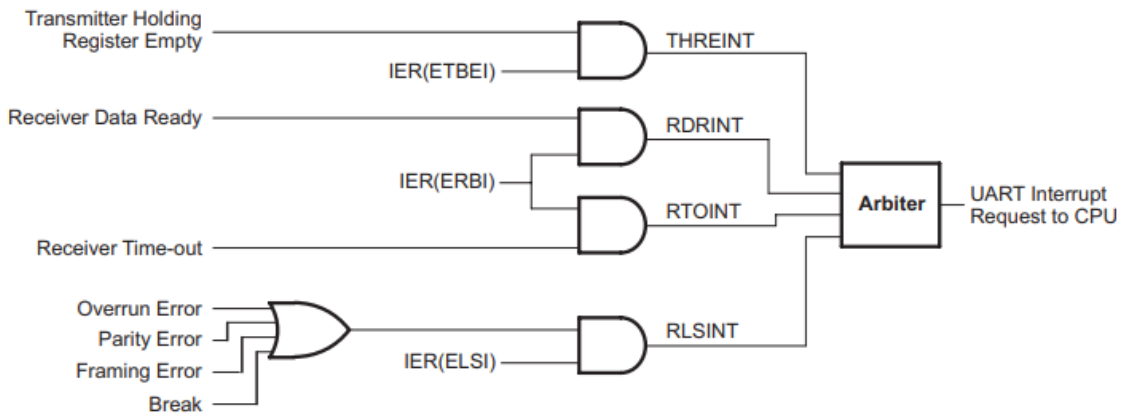
TI Keystone UART Peripheral: Relationships Between Data Bit, BCLK, and UART Input Clock:



TI Keystone UART Peripheral: Autoflow Control – Request to Send (RTS) and Clear to Send (CTS) signals:

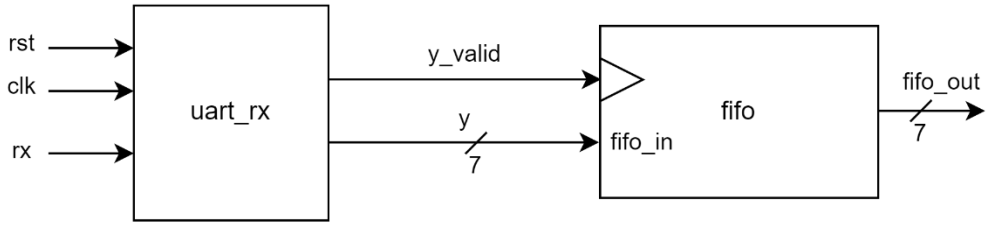


TI Keystone UART Peripheral: UART Interrupt Request Enable Paths:



UART receiver integration test with the FIFO component developed at the Electronics Systems Laboratory lessons:

Block diagram of the FIFO integration test:



Simulation result of the FIFO integration test (FIFO depth = 6, data rate B = 115200):