Electronics and Communications Systems Project

UART Receiver

Carlo Mazzanti

Sommario

Introduction

General description

The Universal Asynchronous Receiver-Transmitter (UART) is a device that permits serial data transmission with configurable rate and data format. It's called asynchronous because there is no clock signal to synchronize transmitter and receiver.

UART is usually employed on computers or peripheral device serial port and often one or more are integrated in microcontrollers.

The UART Transmitter will receive the data to transmit in parallel form, then it proceeds to the serialization where each bit will be sent one by one.

The UART Receiver will sample the line to get the transmitted bits, and, at the end of the transmission, the data is provided in output in parallel form.

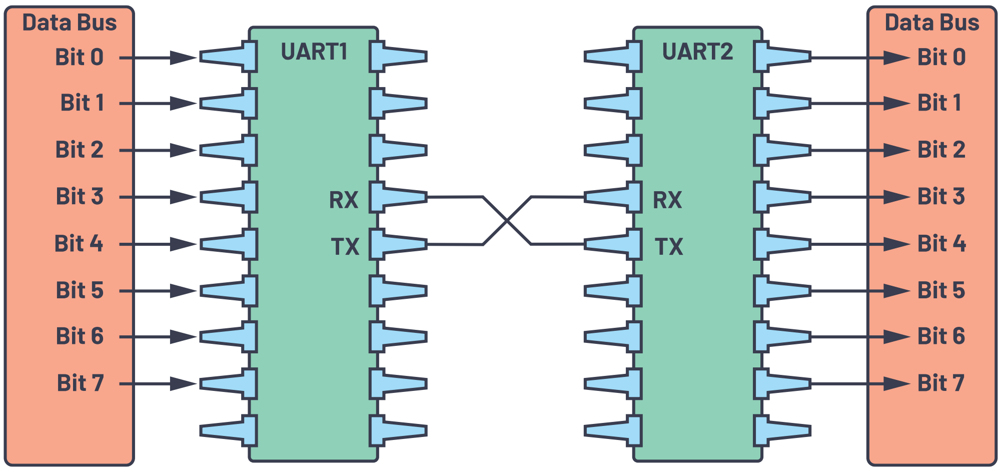


Figure 1. Interconnection of two UART devices
source: https://www.analog.com/en/resources/analog-dialogue/articles/uart-a-hardware-communication-protocol.html

The baud rate (in this case is intended as the number of bits transferred per second) of the transmission is configurable and needs to be the same on both the transmitter and receiver.

Transmission format

The UART protocol is configurable in terms of number of data bits, baud rate, type of parity used and number of stop bits.

The UART packet is composed by four parts:

- Start Bit (1 bit): the data line assumes the logic value 1 when in idle state, by transmitting the start bit with logical value 0 the transmitter signals the start of the transmission to the receiver
- Data Frame (5 to 9 bits): it contains the actual data to transfer. The length is configurable, but it must be known to both transmitter and receiver.
- Parity bits (0 to 1 bit): bit computed by the transmitter and used by the receiver to detect errors in the received data. This bit can be omitted, so no parity is checked.
  The parity can be even or odd: the parity bit will be set so that the total amount of 1 bits is even or odd (start and stop bits are not included).
- Stop bits (1 to 2 bits): they are set to 1 to signal the end of transmission and give enough time to the receiver to process data before the next transmission

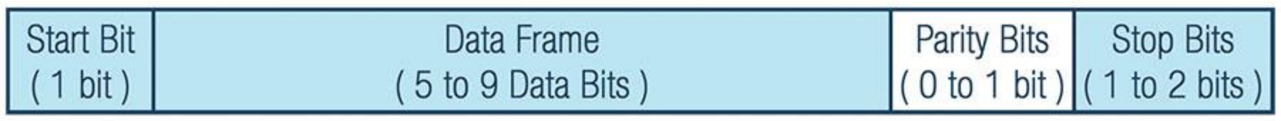

Start Bit (1 bit)
Data Frame (5 to 9 Data Bits)
Parity Bits (0 to 1 bit)
Stop Bits (1 to 2 bits)

Figure 2. UART packet format
source: https://www.analog.com/en/resources/analog-dialogue/articles/uart-a-hardware-communication-protocol.html

The receiver clock frequency will be a multiple of the transmitter clock since we want to gather multiple samples of the same transmitted bit.

Architecture

For understanding the parameters employed in this project, we define the following variables:

- W Number of data bits
- B baud rate
- P parity type
- S number of stop bits

This project implements and tests an UART receiver with these parameters:

- $W = 7$
- $B = 115200$
- $P = Even$
- $S = 2$

The interface of the receiver will be the one specified in figure 3 with an input for the communication line and two outputs for the received data and the assertion of data validity.

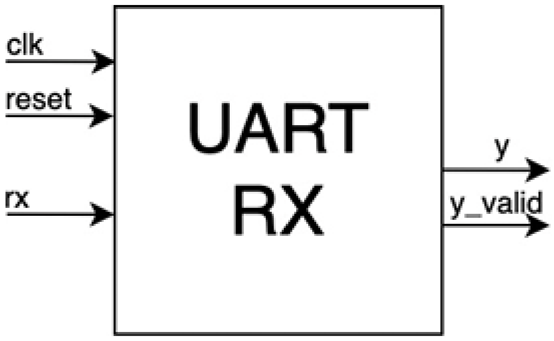


Figure 3. UART Receiver interface

The VHDL code defining the UART Rx entity is:

```vhdl
entity UART_Rx is
    generic(
        W : positive := 7;          -- number of bits per word
        P : positive := 2;          -- 1 odd parity, 2 even parity, 3 no parity
        OVERSAMPLING : positive := 8
    );
    port(
        clk : in std_logic;
        reset : in std_logic;
        rx : in std_logic;
        y : out std_logic_vector(W - 1 downto 0);
        y_valid : out std_logic
    );
end entity;
```

Here we have the parameters W and P that we discussed before. $OVERSAMPLING$ defines how many samples the receiver takes inside a bit time, that for our case will be 8 since the receiver clock frequency is 8 times higher than the transmitter clock.

Since by project specification the clock has to be external to the receiver, it has no meaning the baud rate to be defined since it strongly depends on the clock period, so it will be employed only on the testbench for generating the correct clock for receiving 115200 baud with an oversampling factor of 8. This aspect will be discussed further in the testbench section.

Now each component of the project will be discussed with the relative architecture.

In the block diagrams will be neglected the connections for the clock and reset to enhance the readability. The reset connection will be specified only for particular cases.

Full Adder

The full adder will be the basic block for the implementation of the ripple carry adder.

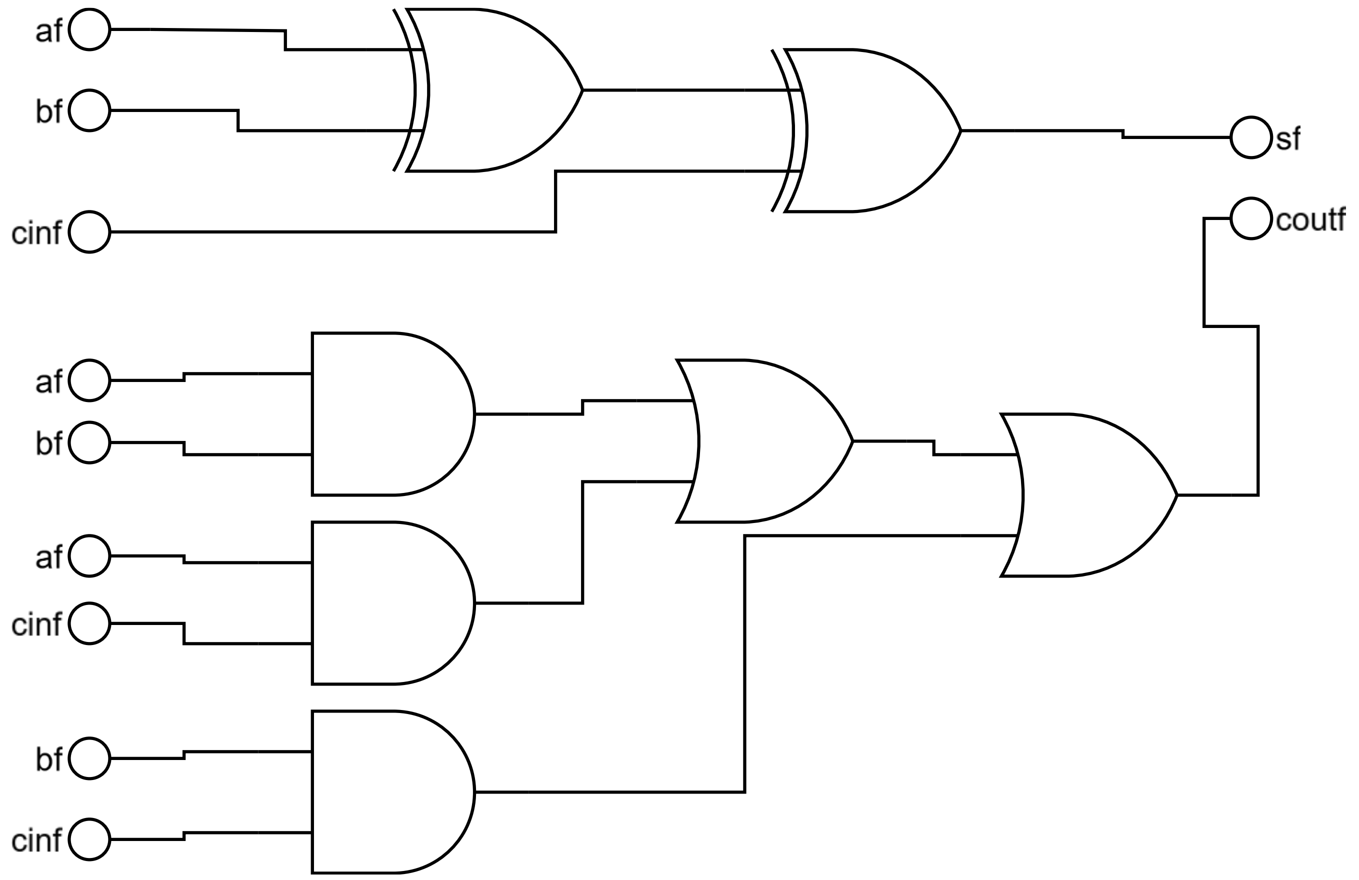


Figure 4. Full Adder architecture

Ripple Carry Adder

The ripple carry adder is used for implementing the addition operation. The number of bits of the inputs (and output) are generalized with the parameter Nbit. In figure 5 we have the architecture when $Nbit = 3$.

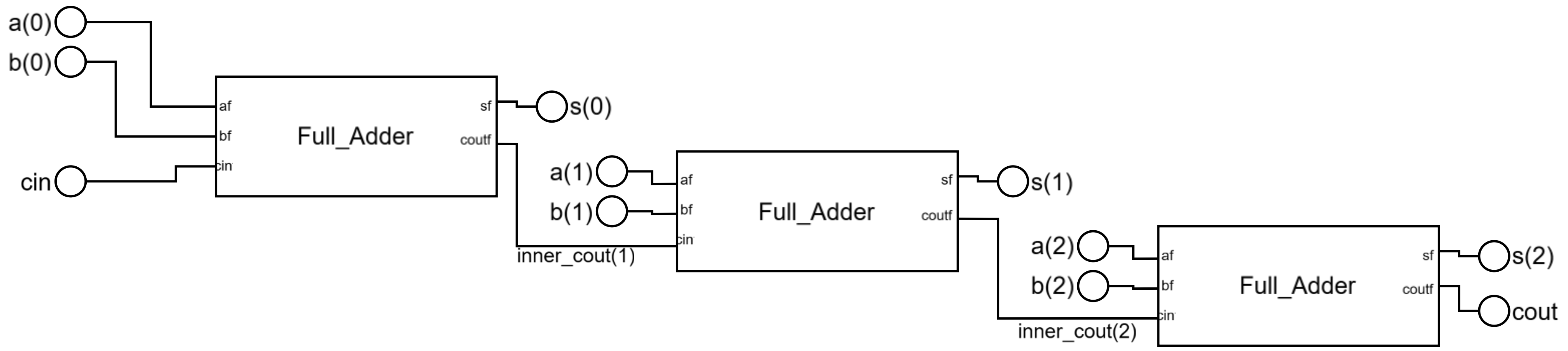


Figure 5. Ripple Carry Adder architecture for the 3 bits case

The implementation is done using a for loop for generating the Full Adders: all Full Adders are connected with each other through the carry in and the carry out signals with the exception of the first and the last.
The first Full Adder will have the carry in connected to the input cin of the Ripple Carry Adder.
The last Full Adder will have the carry out connected to the output cout of the Ripple Carry Adder.

An array of Nbit – 1 internal signals (inner_cout) is employed for propagating the carry between Full Adders.

Multiplexer

The multiplexer will be essential for implementing the Accumulator. It has two data inputs (x0_mu and x1_mu) on a generalized number of bits expressed by the parameter MU_SIZE. The output (y_mu) will assume one of the given data inputs depending on the value of the select input (sel_mu).

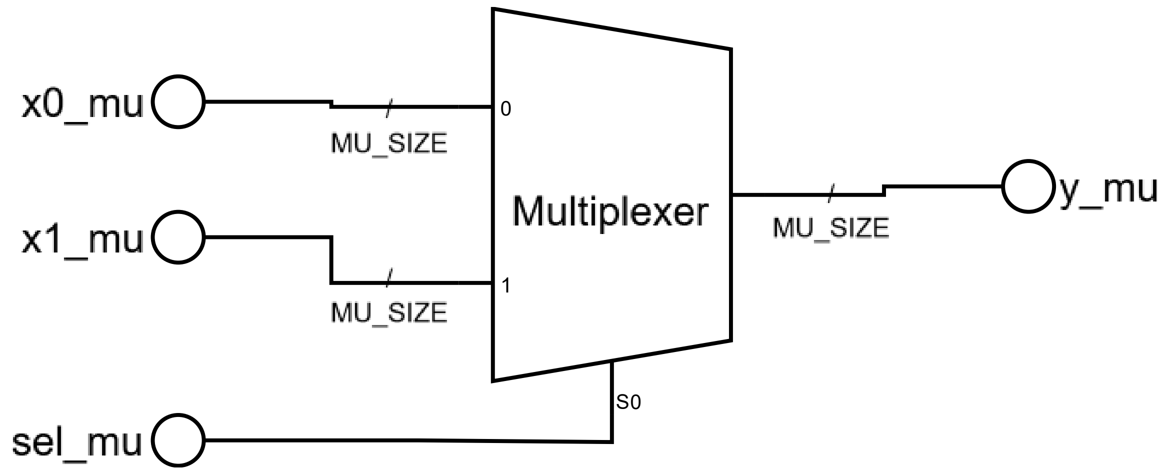


Figure 6. Multiplexer interface

The implementation is done via a process behaving like a combinatorial logic.

DFF

The register used in this project is a D positive edge triggered Flip Flop. The reset mechanism employed is the synchronous one for reasons that will be clear in the UART Rx section.

There are two types of DFF: the regular one at reset assumes the value 0, the modified one (DFF_Rx) at reset assumes the value 1. The purpose for the modified one lies on the wrapper for the UART receiver: since Vivado will evaluate the register logic register paths, we need to include a register at the rx input. For maintaining the correct behavior of the receiver, at reset rx will assume the value 1 so that the line appears idle.

The DFF has a generalized size expressed by the parameter SIZE.

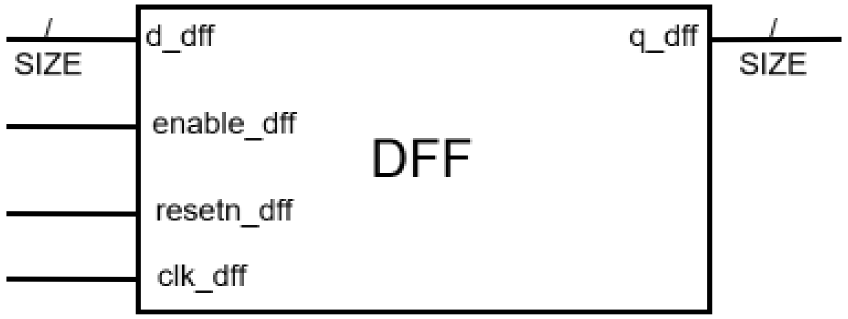


Figure 7. DFF interface

Accumulator

The accumulator is different from the traditional one because it includes two operational modes:

1) Normal mode where the data input will be added to the value held in the DFF
2) Memory mode where the data input goes straight to the DFF input, so it will be memorized

The parameter ACC_SIZE will express how many bits for the data input.

The interface is the following:

```vhdl
entity Accumulator is
    generic(
        ACC_SIZE : positive := 8                                -- amount of bits for input and output
    );
    port(
        d_acc : in std_logic_vector(ACC_SIZE - 1 downto 0);     -- data input
        cin_acc : in std_logic;                                 -- cin of adder
        mode_acc : in std_logic;                                -- operational mode
        enable_acc : in std_logic;
        resetn_acc : in std_logic;
        clk_acc : in std_logic;
        out_acc : out std_logic_vector(ACC_SIZE - 1 downto 0)   -- data output
    );
```

```
end entity;
```

With mode_acc we can specify what mode to use: 0 for normal mode and 1 for memory mode. This function is implemented by the multiplexer that will redirect the correct signal to the DFF input.

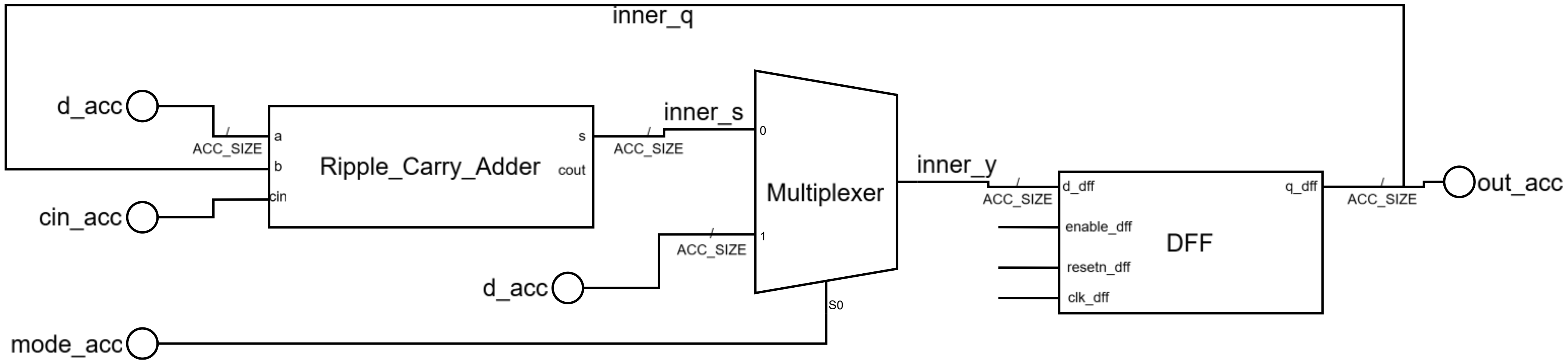


Figure 8. Accumulator architecture

The implementation is simply done by connecting the components like in figure 8.

The carry in input of the Ripple Carry Adder is also used as input for the Accumulator so that we can simply use it as a counter without necessarily using ACC_SIZE bits for the input.

Comparator

Given two inputs on COM_SIZE bits, the comparator will compare the two inputs interpreting them as two positive integers. The interface is the following:

```
entity Comparator is
    generic(
        COM_SIZE : positive := 8
    );
    port(
        a_co : in std_logic_vector(COM_SIZE - 1 downto 0);
        b_co : in std_logic_vector(COM_SIZE - 1 downto 0);
        a_less_b_co : out std_logic;
        a_equal_b_co : out std_logic
    );
end entity;
```

with a_co and b_co the positive integers to compare, a_less_b_co the output that expresses if a_co is less than b_co, a_equal_b_co the output that expresses if a_co equals b_co.

This component is described by a process that behaves as a combinatorial circuit:

```
p_comp: process(a_co, b_co)
    begin
        if a_co = b_co then
            a_equal_b_co <= '1';
        else
            a_equal_b_co <= '0';
        end if;

        if unsigned(a_co) < unsigned(b_co) then
            a_less_b_co <= '1';
        else
            a_less_b_co <= '0';
        end if;
    end process;
```

It is crucial to convert both inputs in unsigned since if they were signed integers the logic would be different.

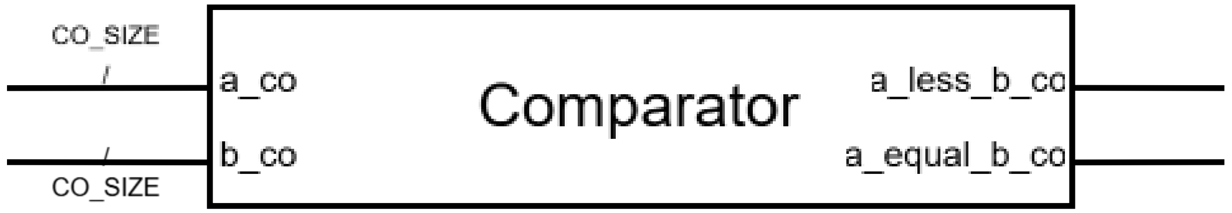


Figure 9. Comparator interface

This component will be essential to elaborate the output of the accumulator.

Demux

This is a normal Demultiplexer on one bit that will be useful for the sampling mechanism. It hasn't been generalized because it wasn't necessary.

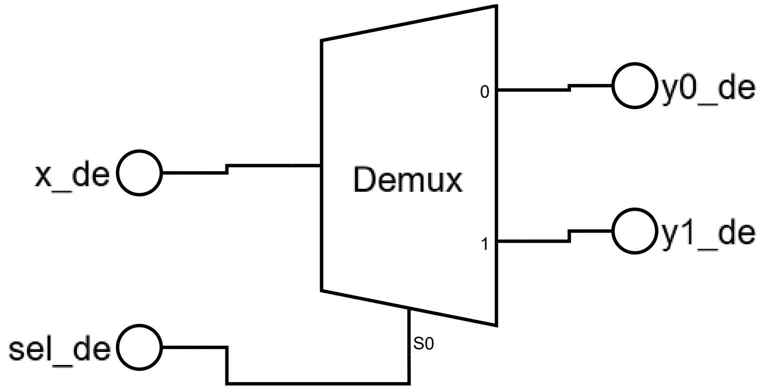


Figure 10. Demux interface

Its implementation is done using directly the AND and NOT logic gates as specified in the code:

```
architecture default of Demux is
begin
    y0_de <= x_de and (not sel_de);
    y1_de <= x_de and sel_de;
end architecture;
```

Sampler

The sampler is the component in charge of handling the oversampling of the line. It relies on the parameter OVERSAMPLING_FACTOR that expresses how many samples it needs to take for each bit time. The idea behind the functioning is: it will count how many times the value 0 and 1 appear in the sample set relative to one bit time. At the end of the bit time it will decide that the transmitted bit will be the one that appears more often. In case of parity than an error will be asserted in input. The interface is the following:

```vhdl
entity Sampler is
    generic(
        OVERSAMPLING_FACTOR : positive := 8      -- number of samples per bit time
    );
    port(
        line_sa : in std_logic;                  -- line to sample
        enable_sa : in std_logic;
        clk_sa : in std_logic;
        resetn_sa : in std_logic;
        data_out_sa : out std_logic;             -- bit received from line_sa
        data_valid_sa : out std_logic;           -- output validity
        error_sa : out std_logic                 -- error in trying to decide the received bit value
    );
end entity;
```

In figure 11 there is the architecture that soon will be discussed.

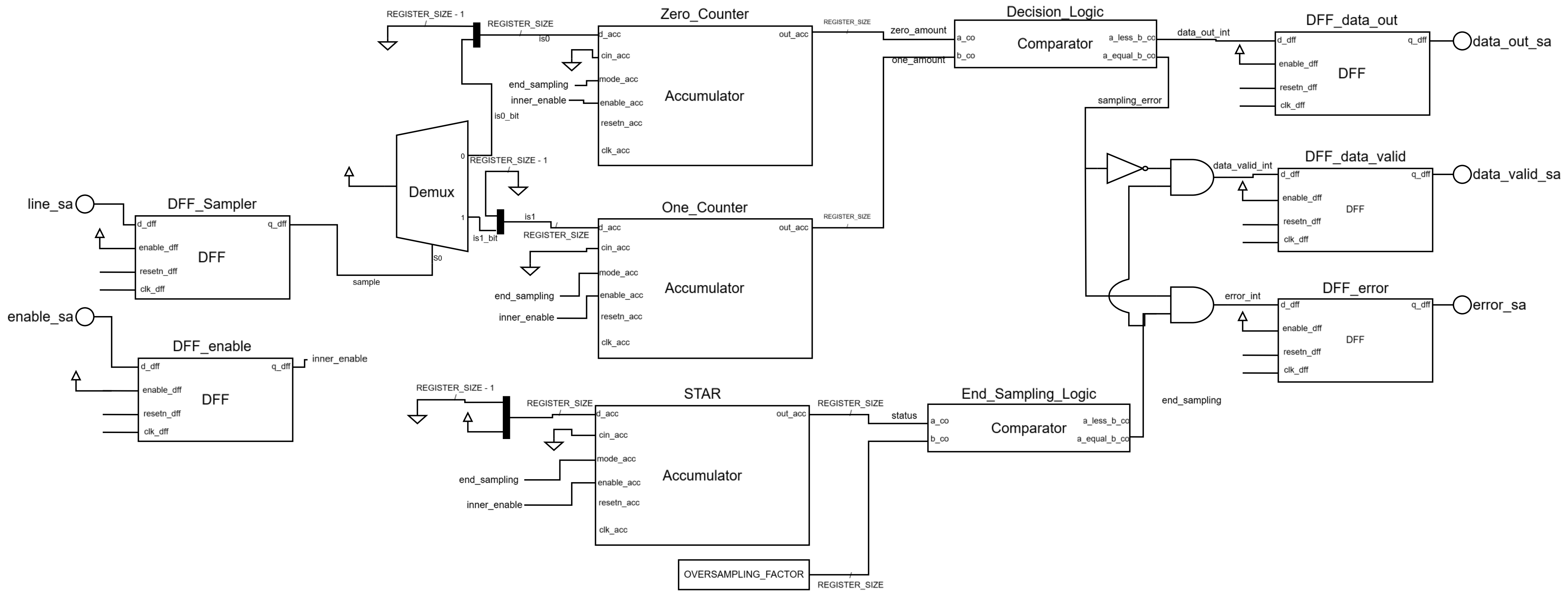


Figure 11. Sampler architecture

Each component has a unique name which is the same as in the VHDL description.

This design requires a correct sizing for the accumulators register that is represented by the constant REGISTER_SIZE. This constant is computed based on the parameter OVERSAMPLING_FACTOR: since the sampler has to count how many times each value appears in the samples; each accumulator will assume at most the value OVERSAMPLING_FACTOR (equal to the total amount of samples for each bit time). This value will certainly be contained in $\log_2(OVERSAMPLING\_FACTOR) + 1$ rounded to the next integer, so on the VHDL description is defined in the following way:

```
constant REGISTER_SIZE : positive := integer(ceil(log2(real(OVERSAMPLING_FACTOR)))) + 1;
```

Sample processing part

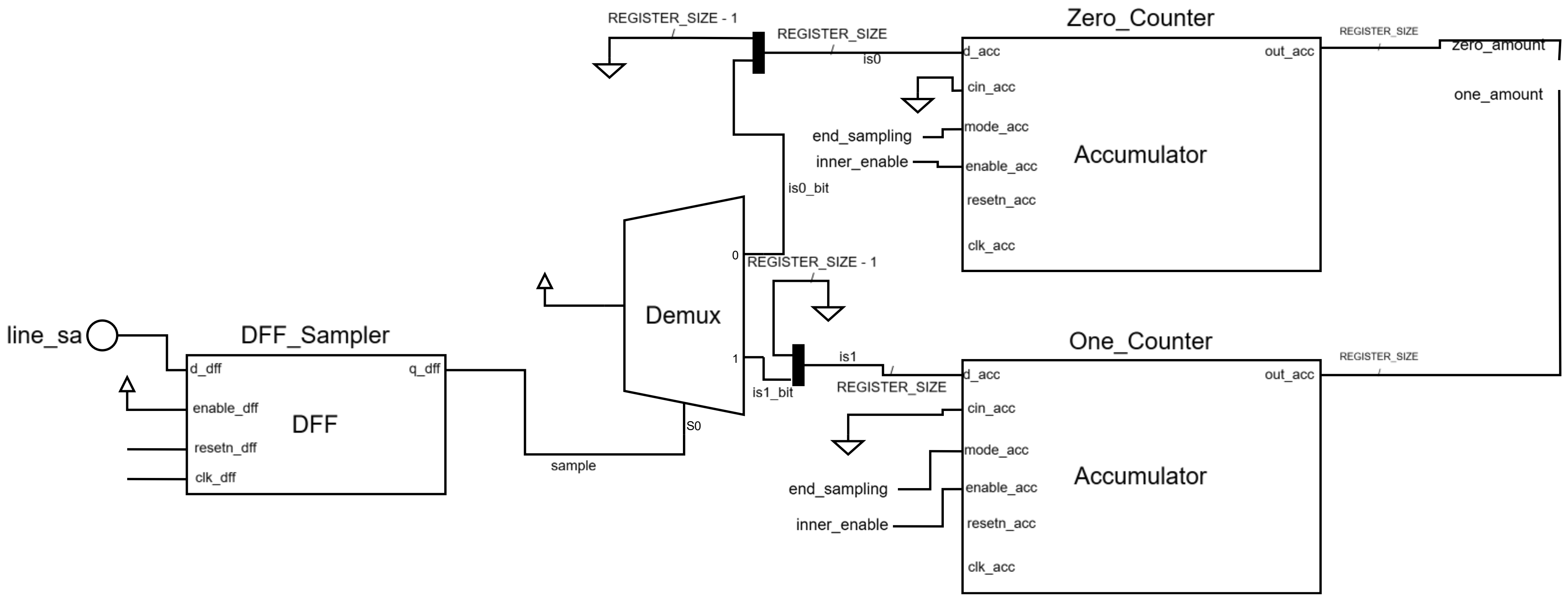


Figure 12. sample processing part of the Sampler architecture

DFF_Sampler will constantly sample the input line and it will be always enabled. Assuming that the circuit is enabled (enable_sa = 1), the gathered sample will select the correct output of the Demux: if 0 is sampled then the logical value 1 will be driven to the output 0, otherwise to the output 1 so that the correct Accumulator value will be increased.

Zero_Counter and One_Counter are the accumulators that will count respectively how many times the values 0 and 1 appear in each sample. The outputs of the Demux are converted on REGISTER_SIZE bits so that the value can be connected to the inputs d_acc. The reason why the outputs of the demux are not connected to the cin_acc inputs is because when a sampling cycle ends, the accumulators need to be reset so that the samples of the next bit time can be processed, but they can’t both just assume the value 0 because we will lose the first sample of the next bit time. We can avoid this by changing mode at the end of the sampling cycle: if the first sample of the next bit time is 0 then Zero_Counter will have d_acc = 1 and One_Counter will have d_acc = 0. Since mode_acc = 1 the input value will be memorized. By doing this each accumulator will be reset considering also the value of the current sample.

Control part

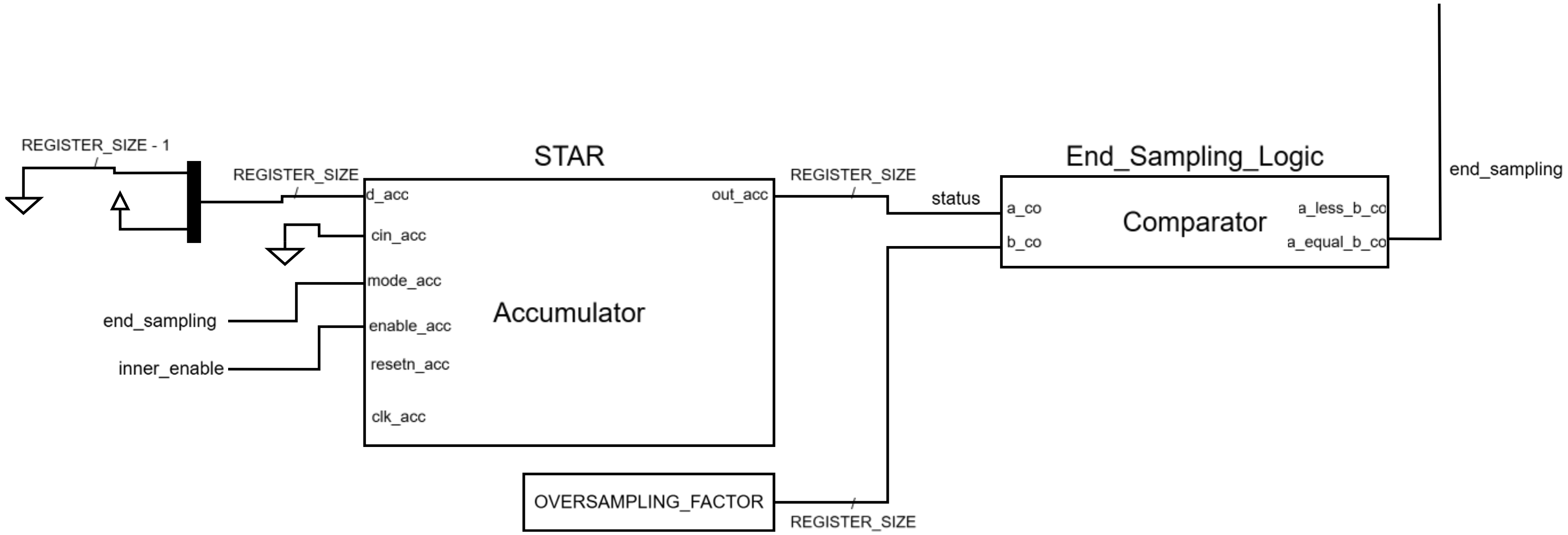


Figure 13. Control part of the Sampler architecture

This part of the Sampler will keep track of the number of samples taken so far. Here is generated the signal end_sampling that will control the operational mode of all Accumulators. It will also contribute to the generation of the valid or error signal.

For each clock, the content of STAR will be incremented by one using the same mechanism of the previous accumulators. The reason behind this decision is that when a sampling cycle ends, the device already took the first sample, so STAR can't be reset at 0.

When the number of samples taken reaches OVERSAMPLING_FACTOR then we concluded a cycle of sampling and end_sampling = 1.

Output part

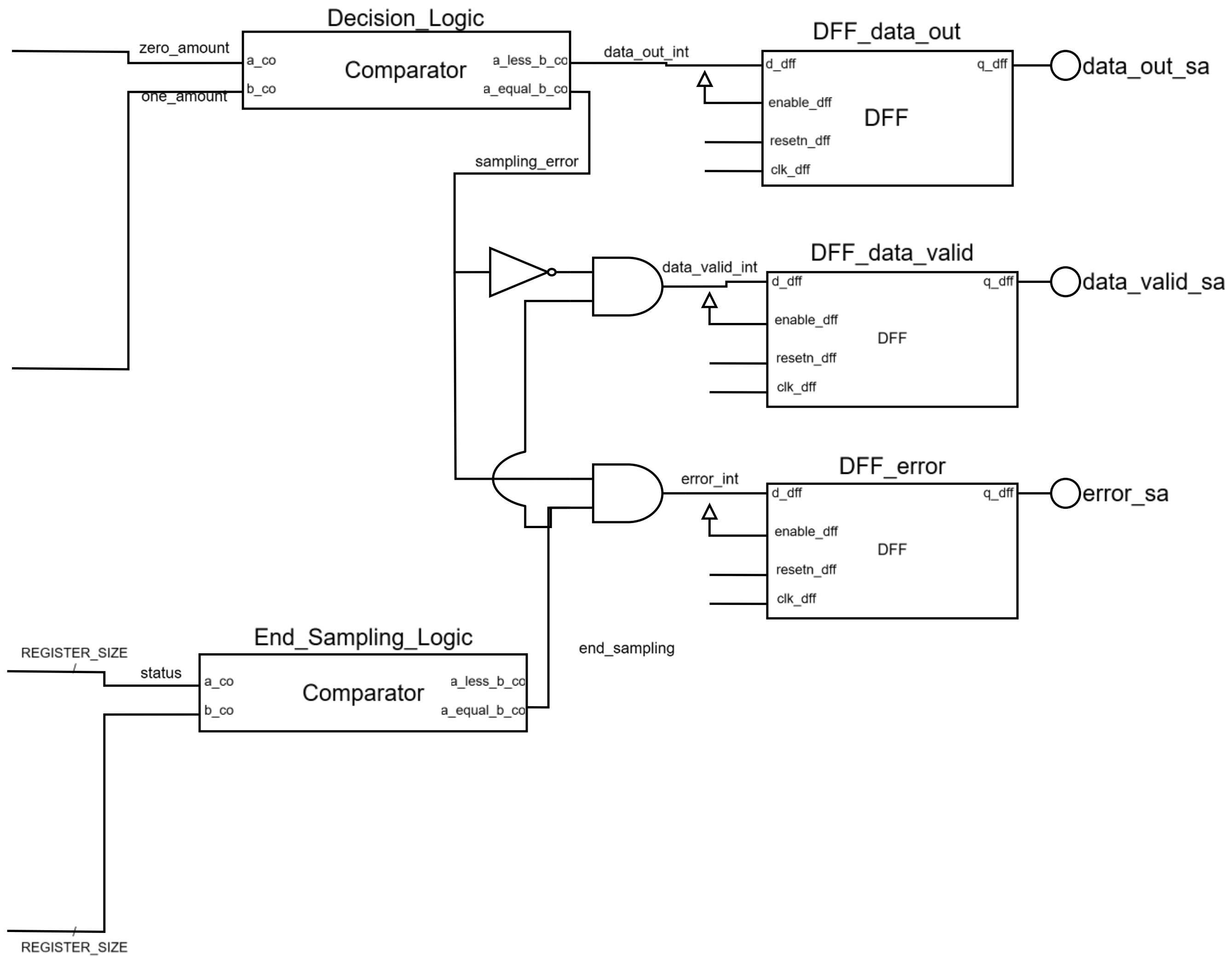


Figure 14. Output part of the Sampler architecture

Since the Sampler will always output data, there are two outputs that will signal when the data is valid and when an error occurs. The decision logic is the following: if the number of zero samples is less than the number of one sample, then we need to output 1, otherwise 0. Since the output a_less_b_co assumes the same values of what the decision logic does, it can be directly output the result of the comparator.

If the number of both types of samples is the same, we are in an error situation that will be asserted in output only when the sampling cycle ends (so that intermediate results are not exposed). Data will be valid only if there are no errors when the end of the sampling cycle occurs.

SRFF

During the data reception period, the UART receiver will use the sampler to manage the oversampling operation, but it needs to know when the line assumes the value 0 for the first time (start bit) and when the sampler encounters an error for the first time.

For these reasons we can’t use a DFF because its value will follow the one in the input. With the set and reset mechanism of the SRFF we have no problems since if we want to know when the line goes to zero for the first time it’s enough to connect the negated line to the set input. Same thing for the sampler error: if we connect the error output to the set input the SRFF will preserve if an error took place for the entire duration of the receiving process.

The SRFF behavior is described through the following process:

```vhdl
p_srff: process(clk_srff)
    begin
        if rising_edge(clk_srff) then
            if s_srff = '1' and r_srff = '0' then
                q_srff <= '1';
            elsif r_srff = '1' then
                q_srff <= '0';
            end if;
        end if;
    end process;
```

This process will implement the SRFF prioritizing the reset signal over the set signal. This is because in the case where the SRFF monitors the line, the reset signal must prevail whatever the value of the line is, otherwise the receiver will fall into an inconsistent state.

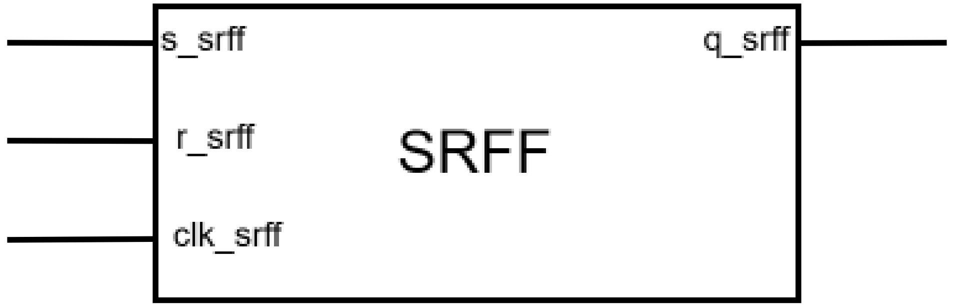


Figure 15. SRFF interface

Parity Checker

The parity checker will contain the logic for verifying the validity of the received bits. The interface is the following:

```
generic(
        PC_TYPE : positive := 2;    -- 1 odd parity, 2 even parity, 3 no parity
        PC_SIZE : positive := 8
    );
    port(
        x_pc : in std_logic_vector(PC_SIZE - 1 downto 0);
        parity_pc : in std_logic;
        valid_pc : out std_logic
    );
```

It has two parameters PC_TYPE that defines the parity type that it implements and PC_SIZE the number of bits on which the parity is computed. The input x_pc represents the data bits on which the parity is computed, parity_pc represents the received parity bit. On valid_pc will be asserted whenever the given inputs are valid.

This component behaves like combinational logic and is implemented with a XOR cascade and a process:

```
begin
    g_xor: for i in 1 to PC_SIZE - 1 generate
        g_first: if i = 1 generate
```

```
            inner_xor(1) <= x_pc(0) xor x_pc(1);
        end generate;

        g_other: if i > 1 generate
            inner_xor(i) <= x_pc(i) xor inner_xor(i - 1);
        end generate;
    end generate;

    p_pc: process(x_pc, parity_pc, inner_xor)
    begin
        if PC_TYPE = 3 then
            valid_pc <= '1';
        elsif PC_TYPE = 1 and parity_pc = not inner_xor(PC_SIZE - 1) then
            valid_pc <= '1';
        elsif PC_TYPE = 2 and parity_pc = inner_xor(PC_SIZE - 1) then
            valid_pc <= '1';
        else
            valid_pc <= '0';
        end if;
    end process;
```

The XOR cascade will compute the sum of the given bits of data without the carry: if the sum is 0 then the number of bits set to 1 are even, otherwise are odd. The process will set the output based on the parity selected and the parity bit provided: if the number of bits set to 1 are even then the parity bit should be 0 (1 in case of odd parity), otherwise the parity bit should be 1 (0 for odd parity).

If no parity is selected then every output will be valid, so the output will be always 1.

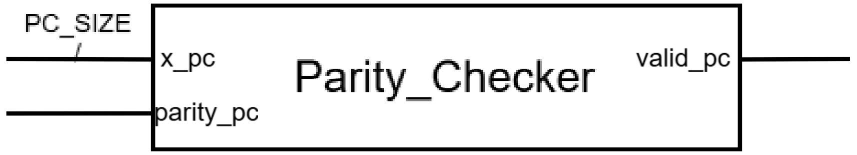


Figure 16. Parity Checker interface

Shift Register

This is a simple shift register with one bit input and an output on SR_DEPTH bits. This will be useful for storing the received data from the Sampler bit by bit.

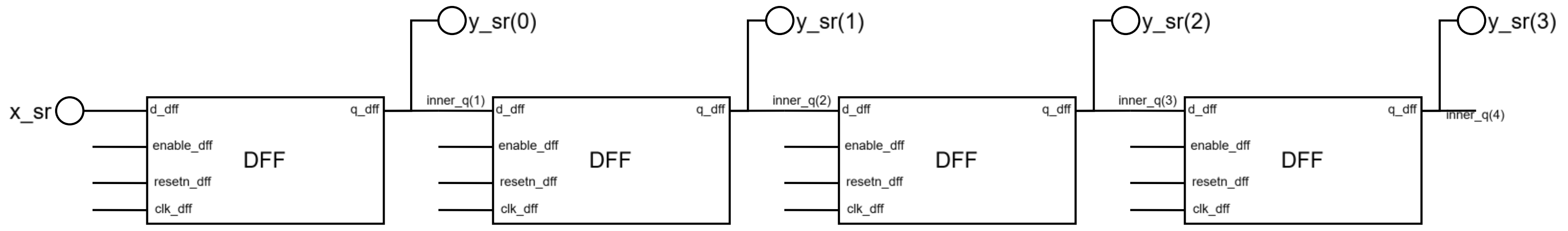


Figure 17. Shift Register architecture for SR_DEPTH = 4

The component is implemented simply interconnecting SR_DEPTH DFFs.

UART Rx

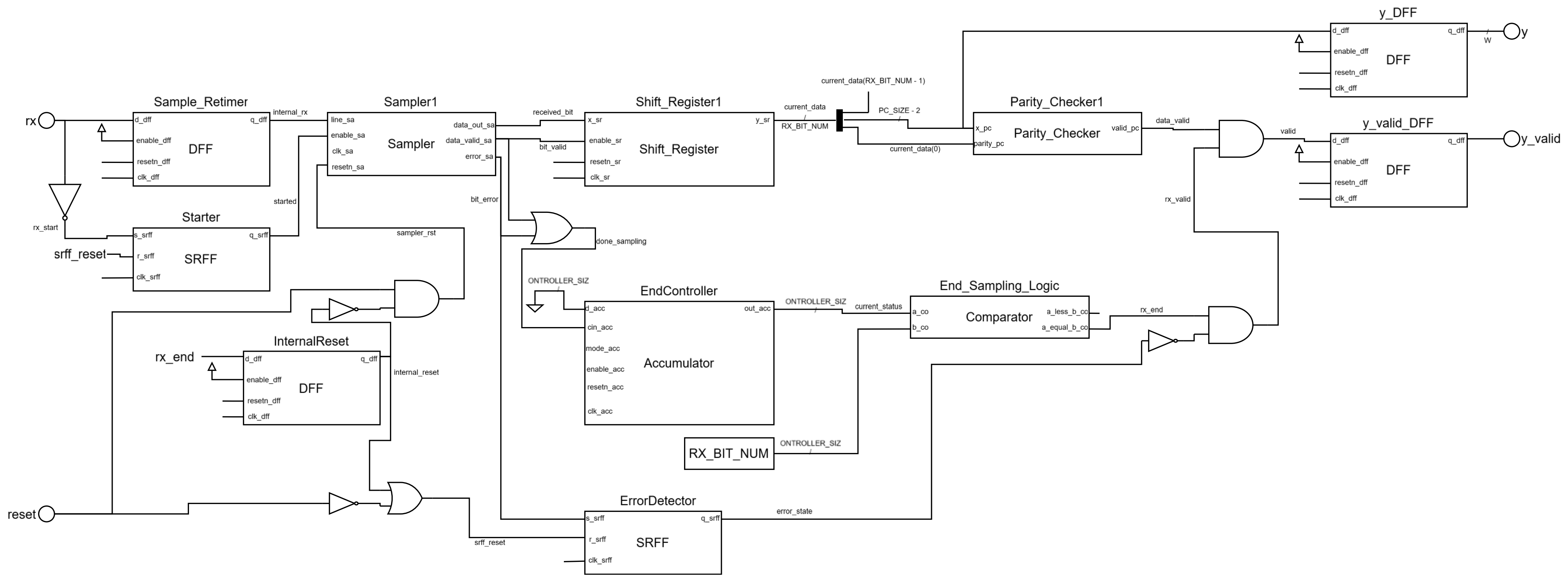


Figure 18. UART Rx architecture

This is the component that implements the UART Receiver requested by the project specifications.

This implementation doesn't directly consider the amount of stop bits: they are only considered as a driving rule for the line since they will give time to the receiver to process the data between one transmission and the other. If this rule is not respected, some samples will be lost since the receiver will restart the receiving process as soon as it finishes the processing. It's important to respect this rule at least most of the time because it will cause more desynchronization as the number of consecutive times the rule is broken increases.

Bits receiving part

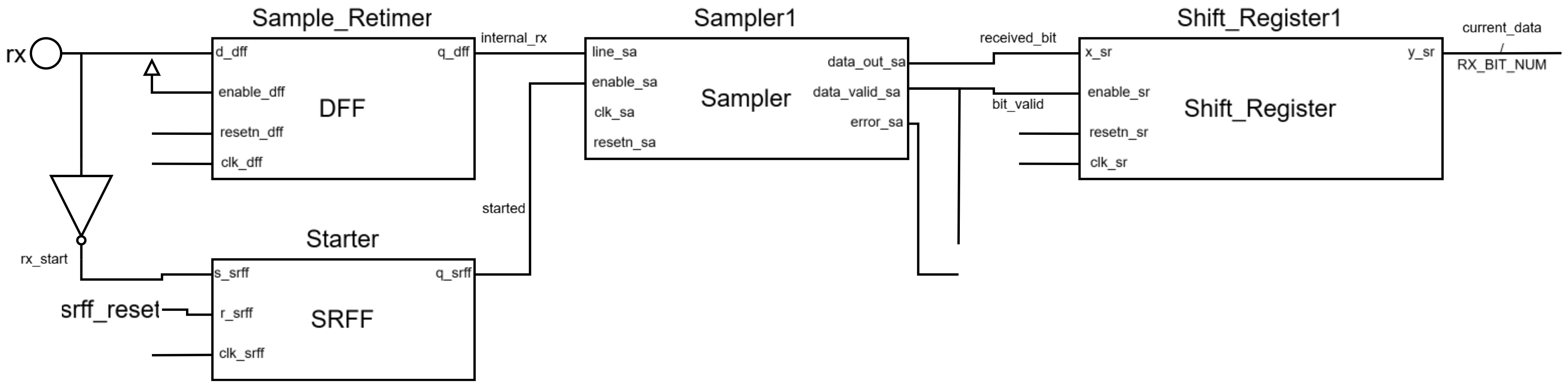


Figure 19. Bits receiving part of the UART Rx

Starter will register when the line goes to zero for the first time, so that the Sampler can be enabled. To avoid the loss of the first sample by the time that the enable signal propagates, Sample_Retimer will preserve the sample for one clock period.

Shift_Register1 will store all valid bits produced by the Sampler: the shift register is enabled only when data_valid_sa = 1 so that it can store only the valid bits.

Since the Sampler starts sampling as soon as the line goes to zero, in the samples set there will be also the start bit, so RX_BIT_NUM is dimensioned so that the shift register can store the start bit, the data bits and the eventual parity bit. The start bit will be discarded later.

Parity check part

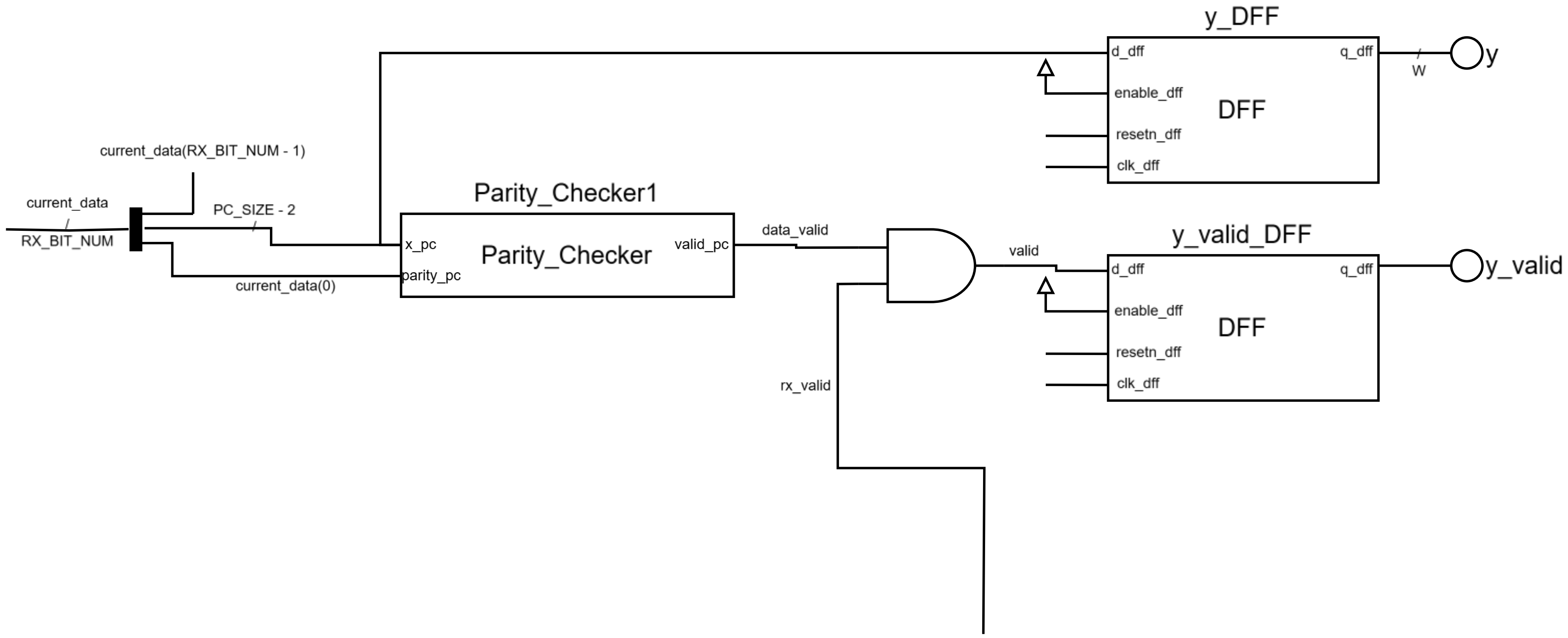


Figure 20. Parity check part of the UART Rx architecture for the case where the parity is implemented

At shift register output we will have RX_BIT_NUM bits. The most significant one is discarded because contains the start bit, the least significant one goes in parity_pc because it contains the parity bit, the others are data bits.

If no parity is implemented, then also the least significant bit is set in input of x_pc because it will contain a data bit. The connection to parity_pc is preserved in both cases because it won't influence the parity check in any case. This aspect is implemented into the declaration of Parity_Checker1:

```
-- for validating the parity
    Parity_Checker1: Parity_Checker
        generic map(
            PC_TYPE => P,
            PC_SIZE => W
        )
        port map(
            x_pc => current_data(RX_BIT_NUM - 2 downto PARITY_BIT_NUM),     -- always remove the start bit, remove the
parity bit when exists
            parity_pc => current_data(0),                                   -- if parity bit exists then goes here,
otherwise it will always be valid
            valid_pc => data_valid
        );
```

PARITY_BIT_NUM is set to 1 if the parity is used, 0 otherwise.

The validity of the data will be asserted if the receiving process has ended and no errors have occurred (rx_valid signal, it will be discussed later).

Control part

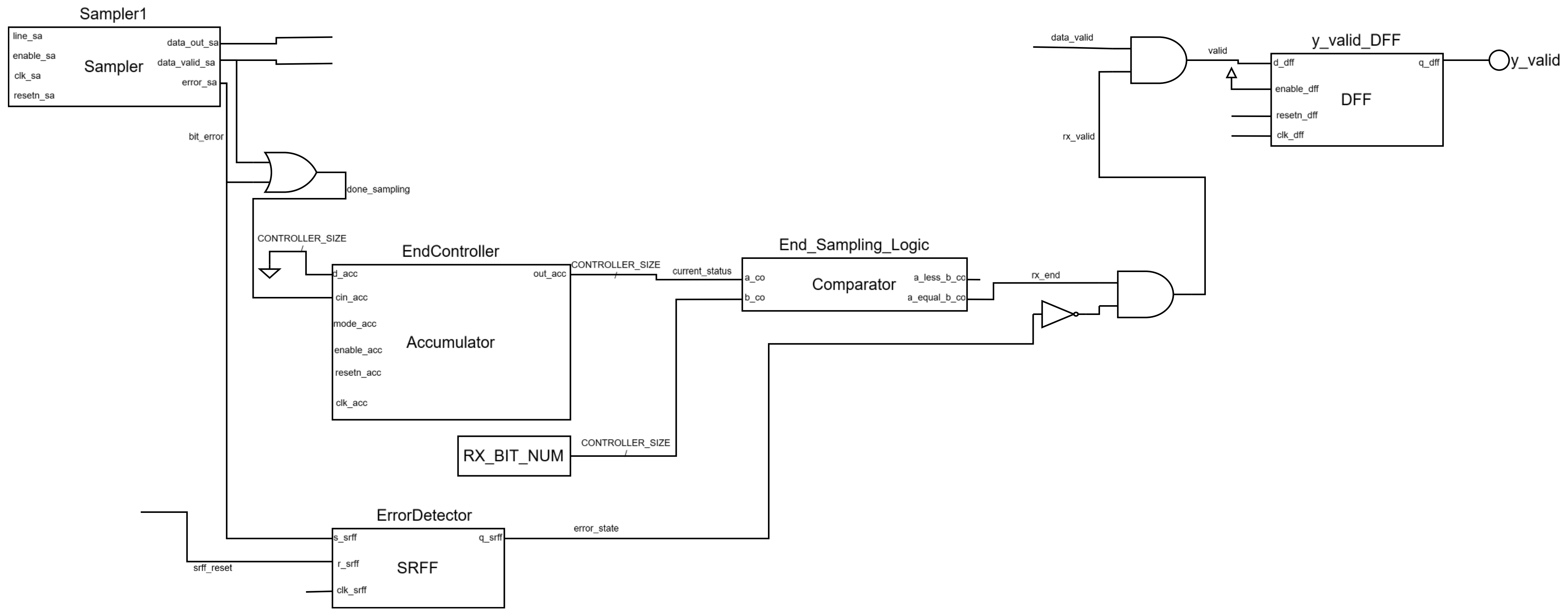


Figure 21. Control part of the UART Rx architecture

EndController will keep track of the amount of sampling cycles performed by the Sampler. A sampling cycle is completed when either the data valid or the error is asserted at Sampler output. When RX_BIT_NUM cycles are completed, we have gathered all the needed bits and the signal rx_end is set to 1.

ErrorDetector will monitor if a sampling error has taken place. If error_sa is raised, the whole receiving process is compromised so the data can't be valid.

Since EndController must hold values up to RX_BIT_NUM, $CONTROLLER\_SIZE = \log_2 RX\_BIT\_NUM + 1$, rounded to the next integer.

When when rx_end is 1, the parity is valid (data_valid) and there were no sampling errors (error_state), then the data set to output can be considered valid.

Internal reset management part

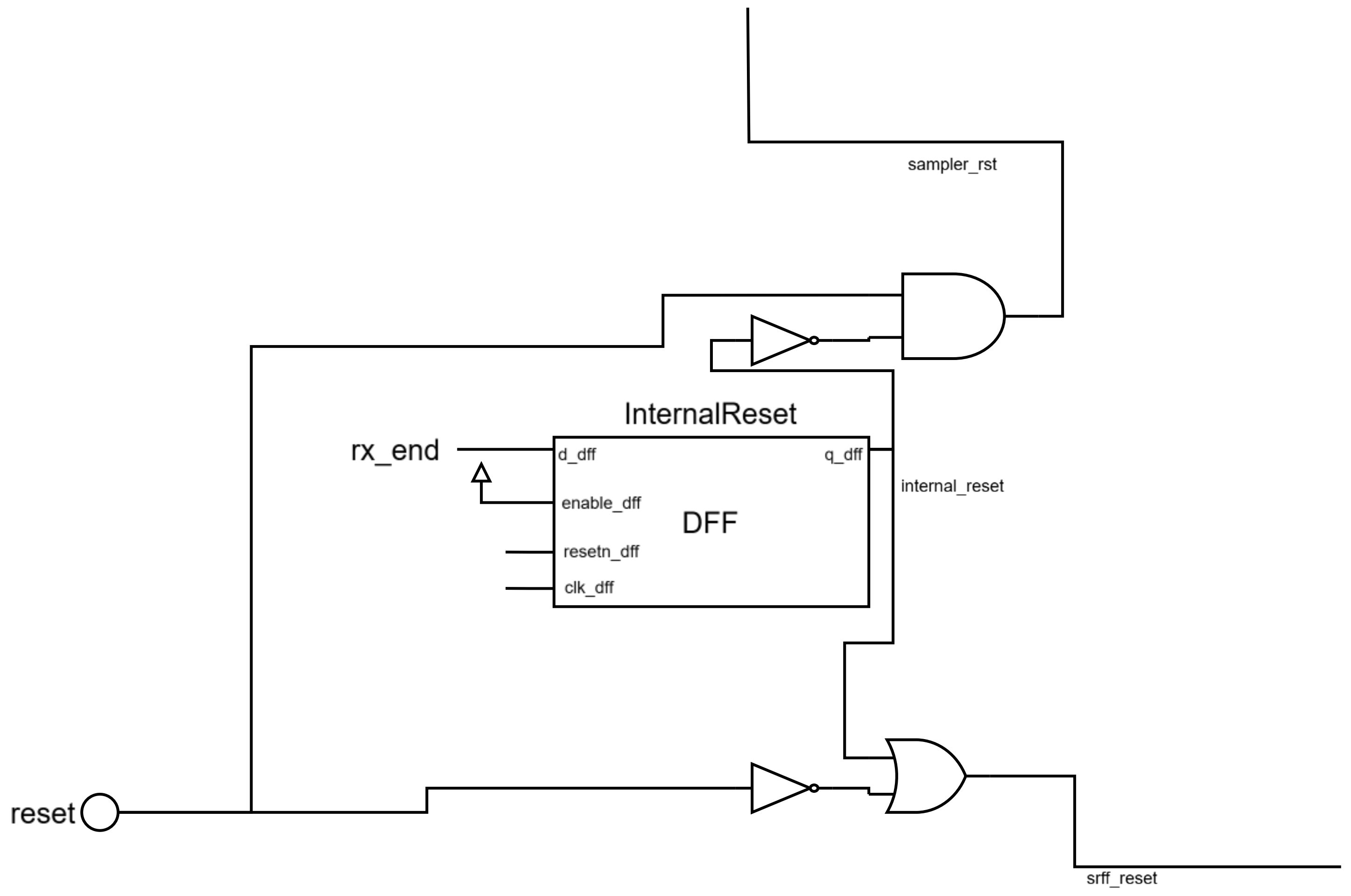


Figure 22. Internal reset management part of the UART Rx architecture

This architecture requires the usage of an internal reset signal for two reasons:

- At the end of the transmission, the whole receiver will take some clock cycles to output data and to lower the enable signal for the sampler. This will cause a temporary sampling of the idle line that influences the next data transmission
- The SRFFs need to be reset after the receiving process

So, an internal reset signal is implemented to reset the sampler between data transmissions and the SRFFs reset. For this reason, it’s crucial a synchronous reset for all flip flops: since the internal reset signal and the external reset signal now need to go through some logic gates, the glitch phenomena may occur that can lead to an unwanted reset of the components in case of asynchronous reset. When enough bits are received, the rx_end signal is raised, and the internal reset is issued. The internal reset is combined with the external reset to create a reset signal for the Sampler (sampler_rst) and a reset signal for all SRFFs (srff_reset). This is because the reset signal for the Sampler is active low and the one for the SRFFs is active high and we want the external reset signal to coexist with the internal one.

Test plan

Each component is provided with a test plan, but this discussion will cover only the testbench of the UART receiver.

The clock period is dimensioned so that the receiver can handle a baud rate of 115200. Since we adopted an oversampling factor equal to 8, we also need a clock frequency 8 times higher than the clock of the transmitter, hence, a clock period 8 times lower than the bit time of the line.

Given a baud rate expressed in bit per second, we have that:

$$BIT\_TIME = \frac{10^9}{BAUD}\ [ns]$$

So, the receiver clock period will be:

$$CLK\_PERIOD\ = \frac{BIT\_TIME}{OVERSAMPLING\_FACTOR}\ [ns]$$

The test results are automatically checked with the assert functionality provided by VHDL.

Four types of tests are employed:

1) Reset test
2) Test of normal operation
3) Test of transmission of data with wrong parity
4) Test of the transmission of inconsistent bits

Reset test

A reset signal is issued to the receiver. It must provide at output the value 0 for both the data output y and the data valid output y_valid.

```
-- test reset --
            reset_ext <= '0';
            wait until rising_edge(clk_ext);
            wait until falling_edge(clk_ext);
            assert y_ext = "0000000" report "wrong y at reset" severity FAILURE;
            assert y_valid_ext = '0' report "wrong y_valid at reset" severity
FAILURE;
            reset_ext <= '1';
            --------------------
```

Test of normal operation

Six data packets are sent to the receiver, all of them will have the correct parity bit and each bit value is held for a BIT_TIME duration.

After the data bits, the line is set to 1 to simulate the end bits. The line is held to 1 for 2 bit time or for the time needed to the receiver to output data. At the end of each transmission the value of both y and y_valid outputs is checked. The valid data signal is asserted for one clock cycle, after that the receiver returns to the idle state waiting for the next start bit. The following is one of the tests performed of this type.

```
 -- test transmission of 1010101/0 --
            rx_ext <= '0';
            wait for (bit_time * 1);                            -- start bit

            rx_ext <= '1';
            wait for (bit_time);
            rx_ext <= '0';
            wait for (bit_time);
            rx_ext <= '1';
            wait for (bit_time);
            rx_ext <= '0';
            wait for (bit_time);
            rx_ext <= '1';
            wait for (bit_time);
            rx_ext <= '0';
            wait for (bit_time);
            rx_ext <= '1';
            wait for (bit_time);
            rx_ext <= '0';
            wait for (bit_time);


            rx_ext <= '1';
            wait until y_valid_ext = '1' for (bit_time * S);    -- stop bits or
result
            assert y_ext = "1010101" report "wrong y" severity FAILURE;
            assert y_valid_ext = '1' report "wrong y_valid" severity FAILURE;
            wait until rising_edge(clk_ext);
            wait until falling_edge(clk_ext);
            --------------------
```

Test of transmission of data with wrong parity

A similar test to the one specified before is performed but with a wrong parity bit. At the end is checked if the y_valid signal rises. It must stay at 0 because no valid data is received.

```
-- test transmission of 1010101/1 (wrong parity)--
            rx_ext <= '0';
            wait for (bit_time * 1);                            -- start bit

            rx_ext <= '1';
            wait for (bit_time);
            rx_ext <= '0';
            wait for (bit_time);
            rx_ext <= '1';
            wait for (bit_time);
            rx_ext <= '0';
            wait for (bit_time);
            rx_ext <= '1';
            wait for (bit_time);
            rx_ext <= '0';
            wait for (bit_time);
            rx_ext <= '1';
            wait for (bit_time);
            rx_ext <= '1';
            wait for (bit_time);


            rx_ext <= '1';
            wait until y_valid_ext = '1' for (bit_time * S);    -- stop bits or
result
            assert y_valid_ext = '0' report "wrong y_valid" severity FAILURE;
            wait until rising_edge(clk_ext);
            wait until falling_edge(clk_ext);
            --------------------
```

This case will send the word 1010101. Since the number of bits set to 1 is 4, the parity bit should be 0, but instead 1 is sent to test the parity check of the receiver.

Test of the transmission of inconsistent bits

This test consists of not holding the same value of the line inside the same bit time. This will test the functionality of the Sampler decision logic for the limit case when it can’t decide what value to output.

The first bit of the transmission (after the start bit) will oscillate and at the transmission end is expected to not have a raised data valid signal since the error for a sampling cycle will lead to an error of the whole transmission.

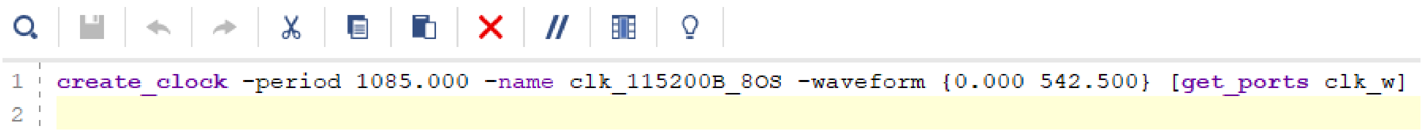
```
-- inconsistent bit inside a bit time
            rx_ext <= '0';
            wait until rising_edge(clk_ext);
            wait until falling_edge(clk_ext);
            rx_ext <= '1';
            wait until rising_edge(clk_ext);
            wait until falling_edge(clk_ext);
            rx_ext <= '0';
            wait until rising_edge(clk_ext);
            wait until falling_edge(clk_ext);
            rx_ext <= '1';
            wait until rising_edge(clk_ext);
            wait until falling_edge(clk_ext);
            rx_ext <= '0';
            wait until rising_edge(clk_ext);
            wait until falling_edge(clk_ext);
            rx_ext <= '1';
            wait until rising_edge(clk_ext);
            wait until falling_edge(clk_ext);
            rx_ext <= '0';
            wait until rising_edge(clk_ext);
            wait until falling_edge(clk_ext);
            rx_ext <= '1';
            wait until rising_edge(clk_ext);
            wait until falling_edge(clk_ext);

            -- consistent bits
            rx_ext <= '1';
            wait for (bit_time);
            rx_ext <= '0';
            wait for (bit_time);
            rx_ext <= '1';
            wait for (bit_time);
            rx_ext <= '0';
            wait for (bit_time);
            rx_ext <= '1';
            wait for (bit_time);
            rx_ext <= '0';
            wait for (bit_time);
            rx_ext <= '0';
            wait for (bit_time);


            rx_ext <= '1';
            wait until y_valid_ext = '1' for (bit_time * S);    -- stop bits or
result
            assert y_valid_ext = '0' report "wrong y_valid" severity FAILURE;
            wait until rising_edge(clk_ext);
            wait until falling_edge(clk_ext);
            --------------------
```

In this case the parity won't matter because even if the received data seems correct, the sampling error will invalidate everything.

Automatic Synthesis and Implementation

A UART Rx wrapper was created for the automatic synthesis and implementation on Vivado, so that it can evaluate all register-logic-register paths.

The synthesis and implementation are performed in out of context mode, so there are no constraints caused by the pin positioning of the FPGA. The only constraint is the one imposed on the clock period to achieve the wanted baud rate.



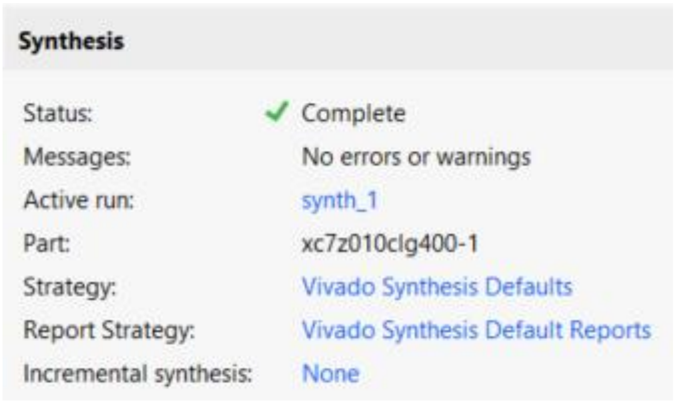
```
create_clock -period 1085.000 -name clk_115200B_8OS -waveform {0.000 542.500} [get_ports clk_w]
```

Figure 23. Content of the clock contraint file

The period of the clock is computed in the way specified in the testbench section with 115200 baud and an oversampling factor of 8.

Synthesis

The result of the Synthesis process is shown in figure 24.



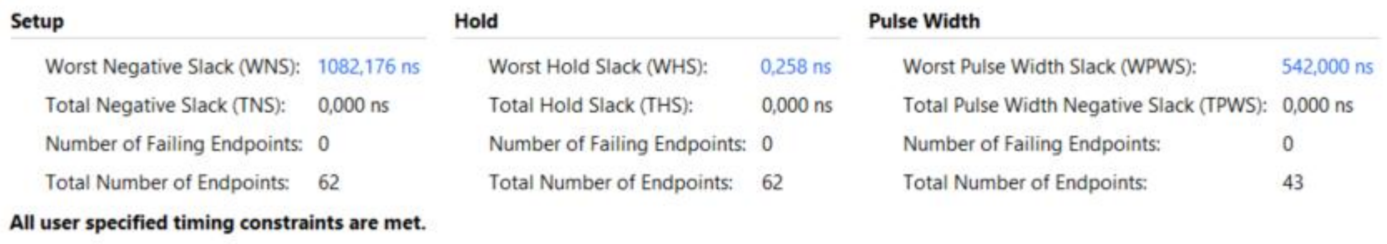
Synthesis

Status:	✓ Complete
Messages:	No errors or warnings
Active run:	synth_1
Part:	xc7z010clg400-1
Strategy:	Vivado Synthesis Defaults
Report Strategy:	Vivado Synthesis Default Reports
Incremental synthesis:	None

Figure 24. result of the synthesis process

This design at synthesis time will take 44 Lookup Tables and 43 Flip Flops, leading to a utilization of the 0.25% of the available lookup tables and 0.12% of the available flip flops.

The results of the timing summary are shown in figure 25.



Setup

Worst Negative Slack (WNS):	1082,176 ns
Total Negative Slack (TNS):	0,000 ns
Number of Failing Endpoints:	0
Total Number of Endpoints:	62

Hold

Worst Hold Slack (WHS):	0,258 ns
Total Hold Slack (THS):	0,000 ns
Number of Failing Endpoints:	0
Total Number of Endpoints:	62

Pulse Width

Worst Pulse Width Slack (WPWS):	542,000 ns
Total Pulse Width Negative Slack (TPWS):	0,000 ns
Number of Failing Endpoints:	0
Total Number of Endpoints:	43

All user specified timing constraints are met.

Figure 25. report timing summary at synthesis time

With the given worst negative slack of about 1082 nanoseconds, with this design we could decrease the clock period to approximately 3 nanoseconds, creating a clock frequency of approximately 330 MHz. This will lead to a receiver that could handle a transmission of approximately $41 \cdot 10^6$ baud.

The worst path estimated at synthesis time is in the Sampler component and is the one connecting Zero_Counter to the input of DFF_data_valid, as shown in figure 26.

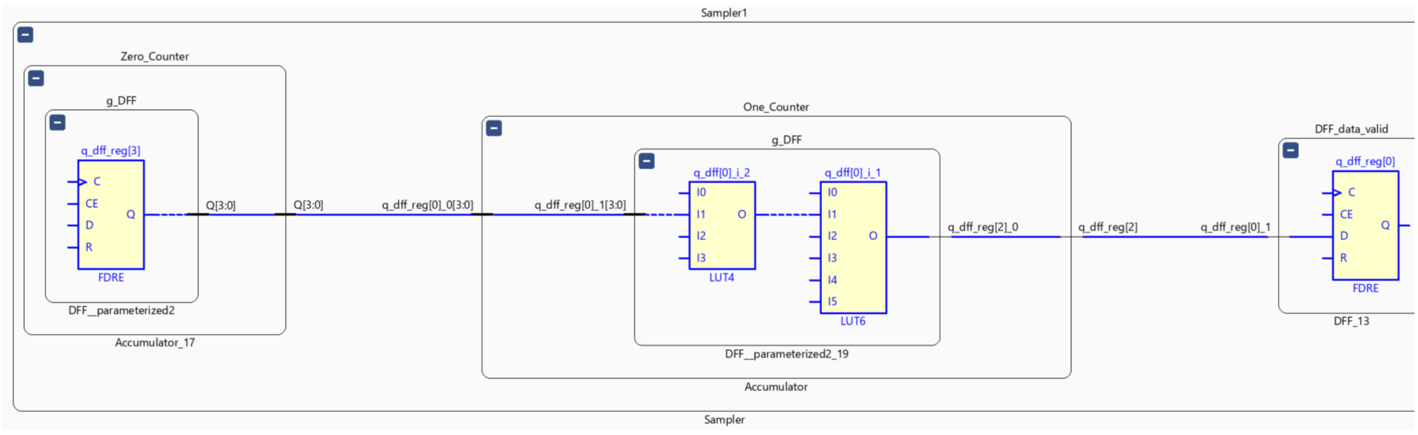
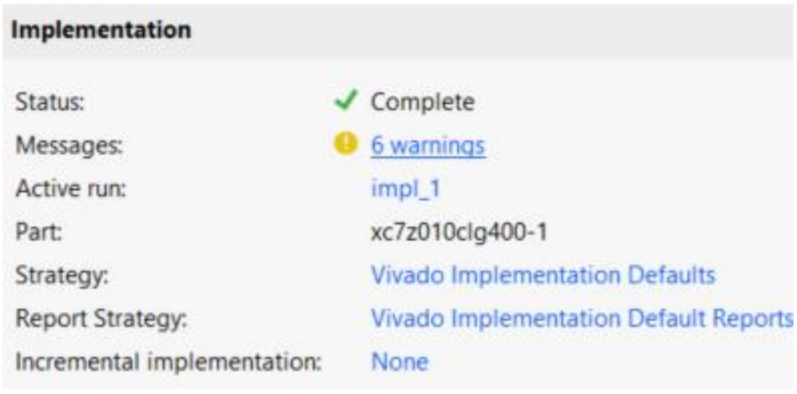


Figure 26. worst path at synthesis time

The cause is that in the path, the signal has to go through two lookup tables.

Implementation

The results of the implementation are shown in figure 27.



Implementation

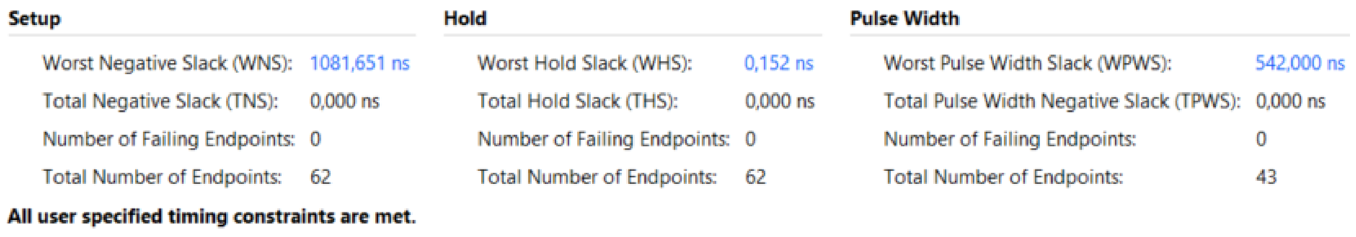
Status:	✓ Complete
Messages:	6 warnings
Active run:	impl_1
Part:	xc7z010clg400-1
Strategy:	Vivado Implementation Defaults
Report Strategy:	Vivado Implementation Default Reports
Incremental implementation:	None

Figure 27. results of the implementation process

6 warnings are generated because of the out of context mode: since no placement constraints are specified, Vivado couldn't estimate any delay of the inputs and the clock skew, so it advises that an accurate timing estimate couldn't be done.

The used resources slightly decrease since the number of required lookup tables goes down to 40 leading to the usage of 0.23% of the available lookup tables.

The results of the timing summary are shown in figure 28.



Setup		Hold		Pulse Width	
Worst Negative Slack (WNS):	1081,651 ns	Worst Hold Slack (WHS):	0,152 ns	Worst Pulse Width Slack (WPWS):	542,000 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:	62	Total Number of Endpoints:	62	Total Number of Endpoints:	43

All user specified timing constraints are met.

Figure 28. report timing summary at implementation time

The worst negative slack slightly decreases, so we can still approximately affirm the same conclusions of the synthesis phase without changing the design.
This change in the slack is caused by a new worst path from InternalReset in UART Rx to DFF_error in Sampler, as shown in figure 29.

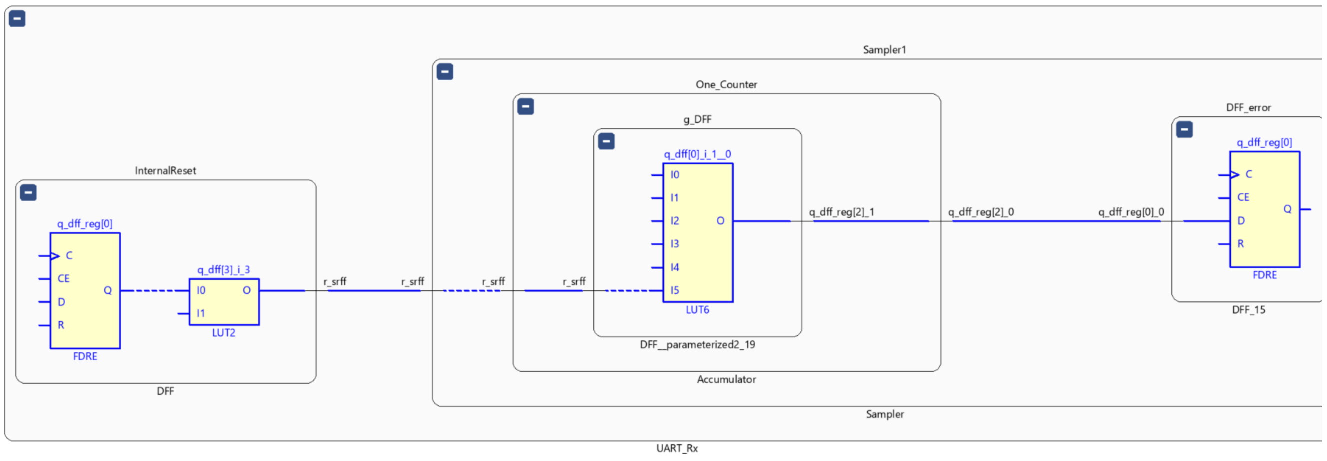


Figure 29. worst path at implementation time

The reason is the same as the synthesis phase.

The total power consumption on chip is 0.09 W distributed as shown in figure 30.

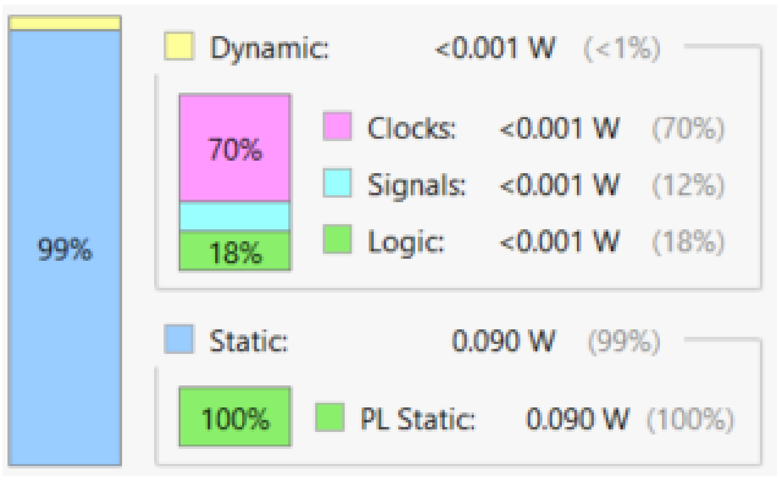


Figure 30. power consumption distribution

All the power consumption is practically due to static power consumption.

Conclusion

The implementation of a UART Receiver involves many components with most of them customizable This is important due to the fact that the UART protocol is itself configurable. With this design, personalization can be made via parameters or by regulating the clock frequency to be able to receive a given baud rate.

Depending on the strategy employed for the oversampling, the number of data bits and the parity, the design can become more complex and cause a lower maximum baud rate.

Relatively to the reliability of the data transmission, the even (odd) parity can ensure a good noise resistance as long as the number of flipped bits is odd (even). For a more reliability is necessary to implement a new strategy, but, since this type of communication is mostly employed between computers and peripherals, it can be enough.

A greater oversampling factor will ensure a better synchronization error resistance since more samples are taken and the decision strategy becomes more effective, but this also leads to a greater clock frequency requirement, so a tradeoff must be made.
In this architecture, the benefits of having an higher oversampling factor are evident because it will be less likely to have the same number of samples for both possible values of bits, so it will be less likely to cause a sampling error.

The functionality of the receiver has been tested in different scenarios such as the normal operational and the limit cases where bits are not stable for all the bit time.