FH Vorarlberg
University of Applied Sciences

Assignment 05 – UART RX

Documentation

Assignment Protocol

Fachhochschule Vorarlberg
ET-Dual

HDL – Hardware Description Layer

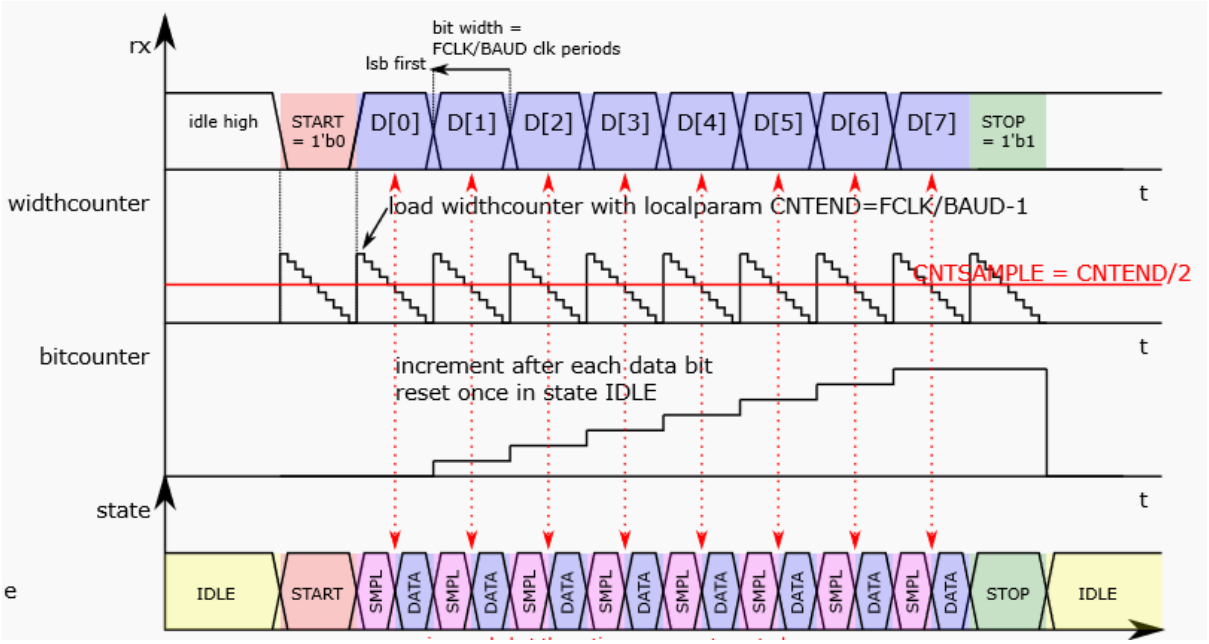

Figure 1: Function of UART RX (Src.: Mitterbacher)

Author:
Stefan Duenser

Dornbirn, 09/12/2021

1. Task description

UART is a digital serial interface that is often used for data exchange between CPU and external peripheries.

After implementing the UART Transmit interface during the lecture, the goal of this 5th assignment was to implement the corresponding Receive interface. This way data can not only be sent.

Data should be received as follows:

- from an idle state the state transitions to the start state via a start bit.
- then 8 data bits follow
- followed by a stop bit before the state changes back to the IDLE state.

The data bits are handled in such a way that the data is sampled after half of the data bit width has elapsed.

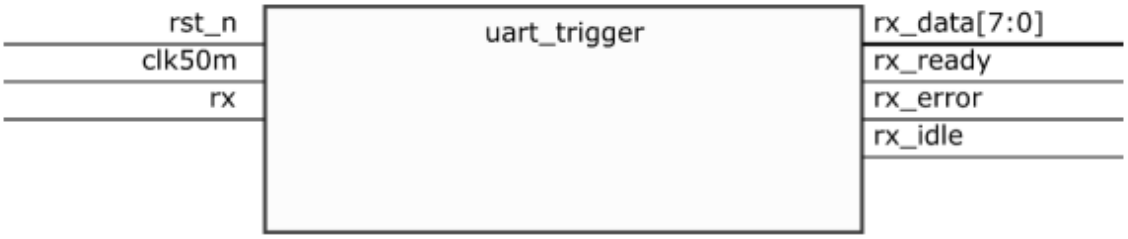

Figure 2: UART RX input and outputs (Src.: Mitterbacher)

The block diagram in Fig. 3 clearly shows which stages the UART RX must pass through to successfully receive data. Fig. 1 also shows how the communication is structured.

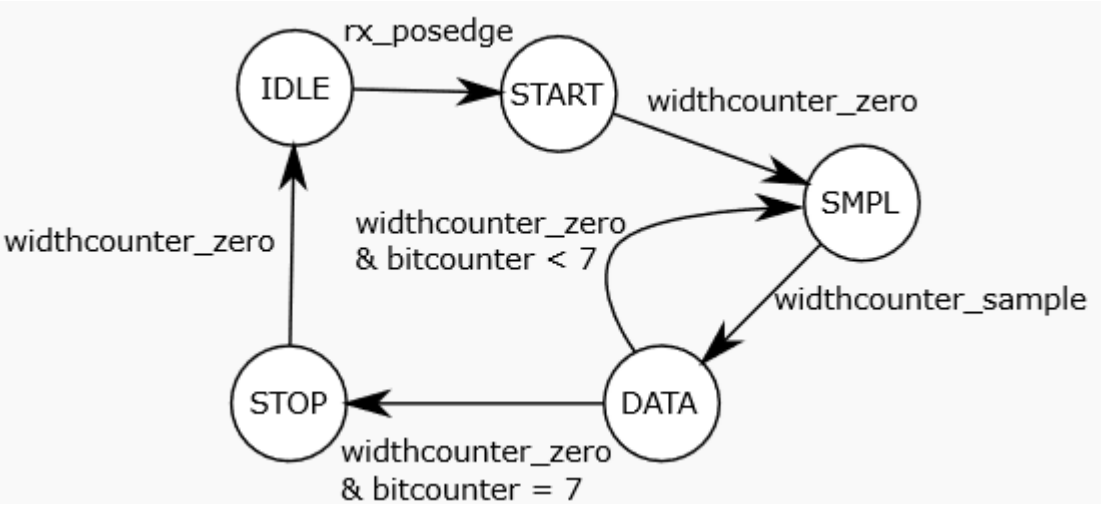

Figure 3: Block diagram of UART RX (Src.: Mitterbacher)

2. Source code

The UART RX is implemented via a finite state machine (FSM). The width of the bits as well as the number (counter) of the bits is solved via memory blocks in an always_ff. The states rx_ready (ready to receive data) and rx_error (framing error) are also stored. Therefore, logic temporary variables were needed to set the information in the combinatorial part (Fig. 6) so that it could be stored late. To save memory space, the logic was implemented in an always_comb (Fig. 6).

In comparison to the UART TX, and also to separate the states cleanly, the state SMPL (sample) was introduced for the UART RX. Fig. 1 shows when the data is sampled. The block diagram (Fig. 3) shows the switching condition for the next state.

```
8   module uart_rx
9   #(
10      parameter WIDTH = 8,
11      parameter FCLK  = 50000000,    // 50 MHz clock
12      parameter FBAUD = 115200
13  )
14  (
15      input logic                 rst_n,
16      input logic                 clk50m,
17      input logic                 rx,
18
19      output logic [WIDTH-1:0]    rx_data,
20      output logic                rx_ready,
21      output logic                rx_idle,
22      output logic                rx_error
23  );
24
25
26  // --- localparameters ---
27  localparam WIDTHCNT_INIT    = ((FCLK / FBAUD) -1);
28  localparam WIDTHCNT_WIDTH   = $clog2(WIDTHCNT_INIT);
29  localparam SAMPLECNT_INIT   = WIDTHCNT_INIT / 2;
30
31
32  // --- local signals ---
33  logic                           widthcnt_zero;
34  logic                           widthcnt_load;
35  logic [WIDTHCNT_WIDTH-1:0]      widthcnt;
36  logic                           bitcnt_inc;
37  logic                           bitcnt_init;
38  logic [2:0]                     bitcnt;
39  logic                           widthcnt_sample;
40
41  logic                           receive_ready;
42  logic                           receive_error;
43
44
45  // --- FSM ---
46  enum logic [2:0] {IDLE, START, SMPL, DATA, STOP}    state, state_next;
47  assign widthcnt_sample = (widthcnt == SAMPLECNT_INIT);
```

Figure 4: UART RX inputs and claculation of UART bit widthcount

```
50  always_ff @(negedge rst_n or posedge clk50m) begin : fsm_seq
51      if(~rst_n) begin
52          state <= IDLE;
53      end
54      else begin
55          state <= state_next;
56      end
57  end
58
59
60
61  // --- WIDTHCNT counter ---
62  always_ff @(negedge rst_n or posedge clk50m) begin : widthcnt_counter
63      if(~rst_n) begin
64          widthcnt <= '0;
65      end
66      else if(widthcnt_load) begin
67          widthcnt <= WIDTHCNT_INIT;
68      end
69      else if(~widthcnt_zero) begin
70          widthcnt <= widthcnt - 1'b1;
71      end
72  end
73  assign widthcnt_zero = (widthcnt == '0);
74
75
76  // --- BITCNT counter ---
77  always_ff @(negedge rst_n or posedge clk50m) begin : bitcnt_counter
78      if(~rst_n) begin
79          bitcnt <= '0;
80      end
81      else if(bitcnt_init) begin
82          bitcnt <= '0;
83      end
84      else if(bitcnt_inc) begin
85          bitcnt <= bitcnt + 1'b1;
86      end
87  end
88
89
90  // --- FF to save flag rx_ready ---
91  always_ff @(negedge rst_n or posedge clk50m) begin : rx_ready_flag
92      if(~rst_n || ~receive_ready) begin
93          rx_ready <= 1'b0;
94      end
95      else if(receive_ready) begin
96          rx_ready <= 1'b1;
97      end
98      else begin
99          rx_ready <= rx_ready;
100     end
101 end
102
103
104 // --- FF to save flag rx_error ---
105 always_ff @(negedge rst_n or posedge clk50m) begin : rx_error_flag
106     if(~rst_n || ~receive_error) begin
107         rx_error <= 1'b0;
108     end
109     else if(receive_error) begin
110         rx_error <= 1'b1;
111     end
112     else begin
113         rx_error <= rx_error;
114     end
115 end
```

Figure 5: Definition of finite state machine (FSM) of the UART RX

```
119   always_comb begin : fsm_comb
120       // Default values
121       state_next      = state;
122       rx_idle         = 1'b0;     // Flag is just one if idel state is present
123       widthcnt_load   = 1'b0;
124       bitcnt_init     = 1'b0;
125       bitcnt_inc      = 1'b0;
126       case(state)
127           IDLE: begin
128               rx_idle         = 1'b1;
129               bitcnt_init     = 1'b1;
130               if(rx == 1'b0) begin
131                   state_next     = START;
132                   receive_error  = '0;
133                   widthcnt_load  = 1'b1;
134               end
135           end
136           START: begin
137               receive_error    = 1'b0;     // Cleared as soon as a new frame startes
138               receive_ready    = 1'b0;     // Cleared as soon as a new frame startes
139               if(widthcnt_zero) begin
140                   state_next      = SMPL;
141                   widthcnt_load   = 1'b1;
142                   bitcnt_init     = 1'b1;
143               end
144           end
145           SMPL: begin
146               if(widthcnt_sample) begin
147                   state_next = DATA;
148                   rx_data[bitcnt] = rx;
149               end
150           end
151           DATA: begin
152               if((widthcnt_zero) && (bitcnt < 3'd7)) begin
153                   state_next      = SMPL;
154                   widthcnt_load   = 1'b1;
155                   bitcnt_inc      = 1'b1;
156               end
157               else if((widthcnt_zero) && (bitcnt >= 3'd7)) begin
158                   state_next      = STOP;
159                   widthcnt_load   = 1'b1;
160               end
161               else if (widthcnt_sample == SAMPLECNT_INIT) begin
162                   state_next      = SMPL;
163               end
164           end
165           STOP: begin
166               if(rx == 1'b0) begin
167                   receive_error   = 1'b1;
168               end
169              if(widthcnt_zero) begin
170                 state_next       = IDLE;
171                 receive_ready    = 1'b1;
172              end
173           end
174           default: begin
175               state_next = IDLE;
176           end
177       endcase
178   end
179
180   endmodule
```

Figure 6: Implementation of block diagram

3. Testbench

The UART RX was tested in a corresponding testbench. The UART TX, already described in the lecture, was used to send constant default values (default assignment). The function check_uart_rx serves to simplify the general queries whether the reception of the data worked as expected or not.

New in contrast to the previous assignments was that combinatorial logic was also required in the testbench. The purpose of this was to provoke a framing error in the testbench. In Fig. 8 you can see that the combinatorial logic checks whether an error is present or not. When checking the correctness of the data reception, it can then be reacted to.

The test should always be terminated when a byte has been successfully received. This is the case when the state changes back to IDLE after the stop bit as recognition for the end of the reception. Afterwards a waiting time was inserted, in order to be able to keep the individual test bytes better apart.

```
module tb_uart_rx
();

// (1) DUT wiring
localparam WIDTH_TB     = 8;
localparam FCLK_TB      = 50000000;
localparam FBAUD_TB     = 115200;


// Input and output parameters for the UART RX
logic                   rst_n;
logic                   clk50m;
logic                   rx;
logic [WIDTH_TB-1:0]    rx_data;
logic                   rx_ready;
logic                   rx_idle;
logic                   rx_error;

// Input and output parameters for the UART TX - just additional parameters
logic [WIDTH_TB-1:0]    tx_data;
logic                   tx_start;
logic                   tx;
logic                   tx_idle;



// (2) DUT instance
uart_rx #(.WIDTH (WIDTH_TB), .FCLK (FCLK_TB), .FBAUD (FBAUD_TB)) uart_rx (.*);

uart_tx #(.WIDTH (WIDTH_TB), .FCLK (FCLK_TB), .BAUD (FBAUD_TB)) uart_tx (
    .rst_n_i    (rst_n),
    .clk_i      (clk50m),
    .data_i     (tx_data),
    .tx_start_i (tx_start),
    .tx_o       (tx),
    .idle_o     (tx_idle)
);

```

Figure 7: Modulation of the testbench for the UART RX

```
// (3) DUT stimulation
// --- local paramter ---
int error_cnt           = 0;
int bitwidth            = 0;
logic run_sim           = 1'b1;
logic rx_error_active   = 1'b0;
logic error             = 1'b0;


// Define the system clock for the testbench
initial begin
    clk50m = 1'b0;
    while (run_sim == 1'b1) begin
        #10ns;
        clk50m = ~clk50m;
    end
end

// Check if an rx error is currently active
always_comb begin : error_check
    if(rx_error_active == 1'b1) begin
        rx = error;
    end
    else begin
        rx = tx;
    end
end


// Function to check the UART RX
function int check_uart_rx(int error_cnt, logic [WIDTH_TB-1:0] rx_data, lo
    int errorCount;
    errorCount = error_cnt;
    assert(rx_data == tx_data) begin
        $display("Received UART message equals Sent UART message");
        $display("UART RX: %h", rx_data);
        $display("UART TX: %h", tx_data);
        $display("Flag ready: %b", rx_ready);
        $display("Flag error: %b", rx_error);
        $display("Flag idle:  %b", rx_idle);
    end
    else begin
        $error("Error during receiving UART message occured");
        $display("UART RX: %h", rx_data);
        $display("UART TX: %h", tx_data);
        $display("Flag ready: %b", rx_ready);
        $display("Flag error: %b", rx_error);
        $display("Flag idle:  %b", rx_idle);
        errorCount++;
    end
    return errorCount;
endfunction
```

Figure 8: Clock definition, frame error manipulation and check function

```
130     $display("------------------------------------------------------");
131     $display("Check for 0x5A");
132     @(negedge clk50m);
133     tx_data     = 8'h5A;
134     tx_start    = 1'b1;
135     #100ns;
136     tx_start    = 1'b0;
137     @(posedge rx_idle);
138     @(negedge clk50m);
139     error_cnt = check_uart_rx(error_cnt, rx_data, tx_data, rx_ready, rx_idle, rx_error);
140     #10us;
141
142     $display("------------------------------------------------------");
143     $display("Check for 0x00");
144     @(negedge clk50m);
145     tx_data     = 8'h00;
146     tx_start    = 1'b1;
147     #100ns;
148     tx_start    = 1'b0;
149     @(posedge rx_idle);
150     @(negedge clk50m);
151     error_cnt = check_uart_rx(error_cnt, rx_data, tx_data, rx_ready, rx_idle, rx_error);
152     #10us;
153
154     $display("------------------------------------------------------");
155     $display("Check for 0xFF");
156     @(negedge clk50m);
157     tx_data     = 8'hFF;
158     tx_start    = 1'b1;
159     #100ns;
160     tx_start    = 1'b0;
161     @(posedge rx_idle);
162     @(negedge clk50m);
163     error_cnt = check_uart_rx(error_cnt, rx_data, tx_data, rx_ready, rx_idle, rx_error);
164     #10us;
165
166     $display("------------------------------------------------------");
167     $display("Check for a forced framing fault");
168     @(negedge clk50m);
169     tx_data     = 8'h00;
170     tx_start    = 1'b1;
171     #100ns;
172     tx_start    = 1'b0;
173
174     #85us;
175     rx_error_active = 1'b1;
176     error = 1'b0;
177     #1us;
178     rx_error_active = 1'b0;
179     @(posedge rx_idle);
180     @(negedge clk50m);
181     error_cnt = check_uart_rx(error_cnt, rx_data, tx_data, rx_ready, rx_idle, rx_error);
182     #10ns;
183
184     $display("------------------------------------------------------");
185     $display("Check if the fault is cleared when a new frame starts");
186     @(negedge clk50m);
187     tx_data     = 8'hFF;
188     tx_start    = 1'b1;
189     #100ns
190     tx_start    = 1'b0;
191     @(posedge rx_idle);
192     @(negedge clk50m);
193     error_cnt = check_uart_rx(error_cnt, rx_data, tx_data, rx_ready, rx_idle, rx_error);
194
```

Figure 9: Test of the UART RX

4. Verification

In the Wave Window you can see very well that the counters for the bit width and the bit counter are displayed in analog form. This makes it easier to recognize the exact intersection points of the individual signal entries.

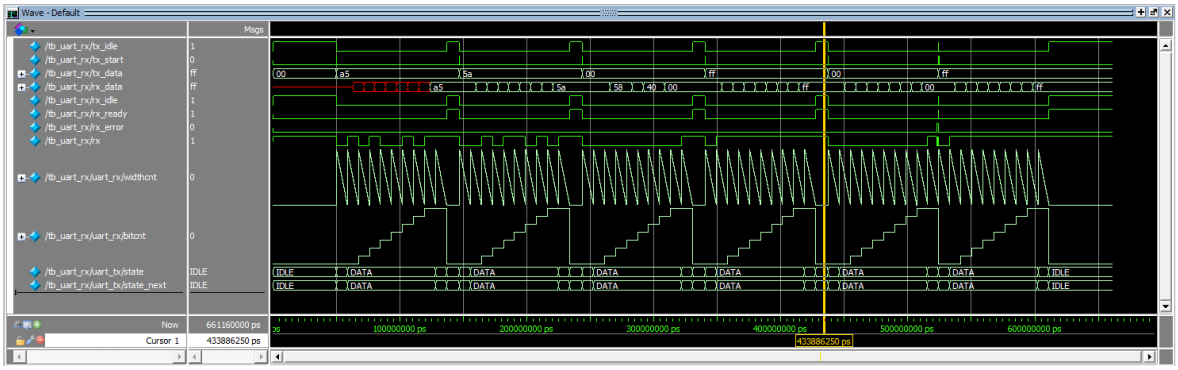

Figure 10: Wave form of the UART RX

```
# ********************************************************************************
# Welcome to the testbench for an UART Reseive (tb_uart_rx)
# ------------------------------------------------------
# ------------------------------------------------------
# Start with a reset
# ------------------------------------------------------
# Check for 0xA5
# Received UART message equals Sent UART message
# UART RX: a5
# UART TX: a5
# Flag ready: 1
# Flag error: 0
# Flag idle:  1
# ------------------------------------------------------
# Check for 0x5A
# Received UART message equals Sent UART message
# UART RX: 5a
# UART TX: 5a
# Flag ready: 1
# Flag error: 0
# Flag idle:  1
# ------------------------------------------------------
# Check for 0x00
# Received UART message equals Sent UART message
# UART RX: 00
# UART TX: 00
# Flag ready: 1
# Flag error: 0
# Flag idle:  1
# ------------------------------------------------------
# Check for 0xFF
# Received UART message equals Sent UART message
# UART RX: ff
# UART TX: ff
# Flag ready: 1
# Flag error: 0
# Flag idle:  1
# ------------------------------------------------------
# Check for a forced framing fault
# Received UART message equals Sent UART message
# UART RX: 00
# UART TX: 00
# Flag ready: 1
# Flag error: 1
# Flag idle:  1
# ------------------------------------------------------
# Check if the fault is cleared when a new frame starts
# Received UART message equals Sent UART message
# UART RX: ff
# UART TX: ff
# Flag ready: 1
# Flag error: 0
# Flag idle:  1
# ------------------------------------------------------
# Errors occoured during the testing:           0
# ------------------------------------------------------
# Testbench for the UART Receive finished (tb_uart_rx)
# ********************************************************************************
```

Figure 11: Log file of the UART RX