آزمایشگاه سیستم های دیجیتال

آزمایش هفتم – UART

امیرحسین علمدار 400105144

محمدپیام تائبی 400105867

علیرضا سلیمیان 400105036

بهار 1402

شرح آزمایش

در این آزمایش قصد داریم تا یک Universal Asynchronous Receiver Transmitter طراحی کنیم.

هدف ارسال داده بین دو دستگاه دیجیتال است که شناختی از یکدیگر ندارند.
پکت داده به اینصورت تعریف شده است که ابتدا یک بیت **start** و سپس یک بیت **parity** و بعد 7 بیت داده و در نهایت یک بیت **stop** ارسال کنیم و گیرنده آن ها را دریافت کند.

ابتدا به سراغ طراحی **ماژول فرستنده** می رویم:

ورودی و خروجی های این ماژول عبارتند از:

7 بیت به ماژول می دهیم و انتظار داریم آن هارا بصورت سریال ارسال کند.

سیگنال خروجی نیز باید همواره یک باشد و به محض صفر شدن آن، مقصد متوجه ارسال داده می شود و داده ها را از طریق همین سیگنال می فرستیم و وقتی یک بماند مقصد متوجه اتمام می شود.

```
module TX # (
    parameter START_SIG = 0
) (
    input           rstN,
    input           clk,
    input           start,
    input   [6:0]   data_in,
    output  reg     s_out = 1,
    output  reg     sent
);

```

این ماژول 4 استیت دارد، آماده به شروع، فرستادن **parity**، فرستادن 7 بیت داده، و حالت پایان که باید همچیز را به استیت آماده به شروع در بیارد.

همانطور که می بینید به هر حالت یک عدد نسبت دادیم و یک رجیستر دو بیتی برای آدرس دهی خانه های آرایه داده ها استفاده کرده ایم.

همچنین در همین بخش، **xor** داده ها را به **parity_sig** اساین کردیم.

```
localparam S_START      = 0;
localparam S_PARITY     = 1;
localparam S_SEND       = 2;
localparam S_STOP       = 3;
reg stop = 0;
reg [1:0]   state = 0;
reg [6:0]   data;
reg [2:0]   data_index;

wire parity_sig;
assign parity_sig = ^data;
```

حال بلاک اصلی این ماژول را می نویسیم.

بخش ریست که در آزمایش های قبل نیز زیاد داشتیم و عملکرد آن مشخص است.

برای استیت های ارسال به اینصورت عمل می کنیم که تا وقتی سیگنال ورودی start که دست کاربر است فعال نشده باشد، ارسالی انجام نخواهد شد. همچنین اگر یک ارسال انجام شده باشد، باید حداقل یکبار start صفر و سپس یک شود تا بتوان دوباره ارسال انجام داد. این بخش را برای کارایی بهتر اضافه کردیم زیرا سرعت کلاک بسیار بالا است و هنگام تست بارها و بارها یک کد یکسان ارسال میشد تا قبل از اینکه کد جدیدی وارد کنیم.

هنگامی که به حالت شروع وارد شدیم، طبق چیزی که خواسته شده بود، start را low می کنیم تا مقصد متوجه ارسال دیتا شود. همچنین استیت را به حالت ارسال پریتی تغییر می دهیم.

در کلاک بعد، پریتی را می فرستیم و استیت را جلو می بریم.

سپس داده ها را در کلاک ها متوالی تا جایی که ایندکسشان اجازه دهد ارسال میکنیکم.

و درنهایت به استیت stop می رویم و تمام مقادیر را به حالت اولیه باز می گردانیم.

```
always @(posedge clk or negedge rstN) begin
    if (~rstN) begin
        state <= S_START;
        data_index <= 0;
        sent <= 0;
        stop <= 0;
        s_out <= ~START_SIG;
    end
    else begin
        case (state)
            S_START: begin
                if (start && !stop) begin
                    s_out <= START_SIG;
                    data_index <= 0;
                    data <= data_in;
                    state <= S_PARITY;
                    sent <= 0;
                end else if(!start)
                    stop <= 0;
            end
            S_PARITY: begin
                s_out <= parity_sig;
                state <= S_SEND;
            end
            S_SEND: begin
                s_out <= data[data_index];
                if (data_index == 6)
                    state <= S_STOP;
                data_index <= data_index + 1;
            end
            S_STOP: begin
                s_out <= !START_SIG;
                state <= S_START;
                sent <= 1;
                stop <= 1;
            end
            default: state <= S_START;
        endcase
    end
end
```

حال میخواهیم ماژول گیرنده را طراحی کنیم.
ورودی ها و خروجی ها عبارتند از:

```
module RX # (
    parameter START_SIG = 0
) (
    input               rstN,
    input               clk,
    input               s_in,
    output  reg         received,
    output  reg         parity,
    output  reg [6:0]   data
);
```

تفاوت این ماژول در ورودی ها و خروجی ها چنین است که

بیت ها را سریالی دریافت می کنیم و بصورت موازی آن ها را به کاربر می دهیم.

با تمام شدن عملیات، received یک می شود.

```
localparam S_IDLE       = 0;
localparam S_PARITY     = 1;
localparam S_RECEIVE    = 2;
localparam S_STOP       = 3;

reg [1:0]   state;
reg [2:0]   data_index;
reg last_s_in = 1;
```

این مآژول نیز 4 حالت دارد که در عکس مشاهده می کنید.

مانند مآژول فرستنده، آرایه ای برای رفتن به خانه های آرایه داده ها در نظر گرفتیم و یک متغیر last_s_in داریم.

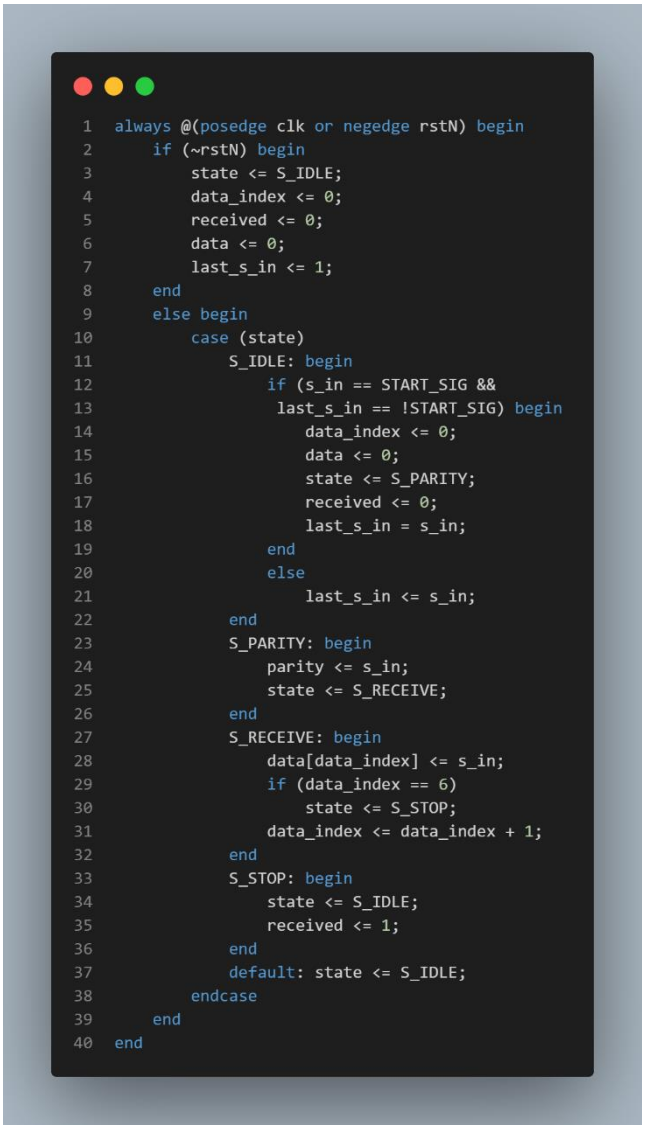
کاربرد این متغیر برای متوجه شدن تغییر ورودی از 1 به 0

است تا بفهمیم فرستنده درحال شروع به ارسال داده است.

بخش ریست مانند ماژول قبلی است.

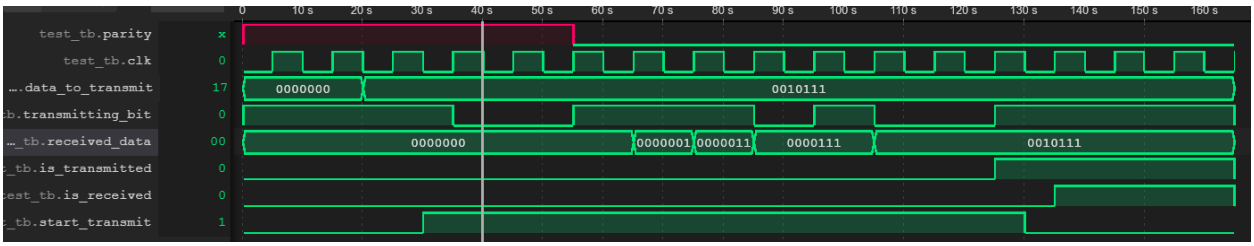
اگر ورودی از 1 به 0 تغییر دهد، با لبه بالا رونده کلاک به داخل شرط اول می رویم و متوجه ارسال داده خواهیم شد پس از حالت IDLE در می آییم و به حال خواندن بیت توازن می رویم.
بعد از خواندن بیت توازن، به حالت گرفتن داده ها می رویم و بعد از خواندن 7 بیت داده، به حالت پایانی می رویم و استیت را به حالت IDLE برخواهیم گرداند.

همچنین سیگنال received که نشان دهنده اتمام دریافت است، فعال خواهد شد. تا وقتی شروع به دریافت دوباره کنیم یا دستگاه را ریست کنیم، این سیگنال روشن خواهد بود.



```
always @(posedge clk or negedge rstN) begin
    if (~rstN) begin
        state <= S_IDLE;
        data_index <= 0;
        received <= 0;
        data <= 0;
        last_s_in <= 1;
    end
    else begin
        case (state)
            S_IDLE: begin
                if (s_in == START_SIG &&
                 last_s_in == !START_SIG) begin
                    data_index <= 0;
                    data <= 0;
                    state <= S_PARITY;
                    received <= 0;
                    last_s_in = s_in;
                end
                else
                    last_s_in <= s_in;
            end
            S_PARITY: begin
                parity <= s_in;
                state <= S_RECEIVE;
            end
            S_RECEIVE: begin
                data[data_index] <= s_in;
                if (data_index == 6)
                    state <= S_STOP;
                data_index <= data_index + 1;
            end
            S_STOP: begin
                state <= S_IDLE;
                received <= 1;
            end
            default: state <= S_IDLE;
        endcase
    end
end
```

ابتدا کارکرد همین دو قطعه را بررسی می کنیم و سپس در بخش بعد، با اضافه شدن چند بخش دیگر، کل سیستم را تست میکنیم:



با کمی دقت مشاهده میکنیم که در لحظه 20، مقداری برای ارسال مشخص شده اما هنوز کاربر(در اینجا تست بنچ) سیگنال شروع به ارسال را فعال نکرده است. با فعال شدن این سیگنال، در کلاک های بعدی transmitting_bitبه ترتیب از lsb تا msb خواهد بود. همچنین مشاهده میکنیم که پریتی ناشناخته است تا زمانی که ارسال شود.

در نهایت با اتمام ارسال، سیگنال is_transmitted یک خواهد شد و یک کلاک بعد آن دریافت تمام می شود و is_received نیز یک می شود.

(به علت ابتدایی بودن تست بالا، کد تست بنچ آن ضمیمه نشد)

حال باید این دو ماژول را طوری کنار هم قرار دهیم که هم ارتباط دو طرفه برقرار باشد، هم با Baud rate مقصد که 115200 بیت برثانیه است، هماهنگ باشد.

فرکانس کلاک fpga، 50 مگاهرتز است. برای فرستادن یا گرفتن داده، باید با فرکانس طرف دیگر که 115200 است هماهنگ شویم. دستگاه ما هر ثانیه 50 میلیون بار وارد always می شود و باید یک پالس را 115200*2 بار در ثانیه نات کنیم تا به فرکانس مورد نظر برسیم. پس کافیست هر $\frac{50,000,000}{2*115200}$ بار که وارد always شدیم، یکبار پالس را نات کنیم. این مقدار در با کمی تقریب 217 است.

پس یک مآژول برای ساختن پالس های مصنوعی از کلاک fpga تعریف میکنیم.

کافیت یک counter قرار دهیم و هنگامی که به 217 رسید، پالس را نات کنیم.

سپس از پالس خروجی این ماژول برای کلاک دستگاه های فرستانده و گیرنده بهره ببریم.

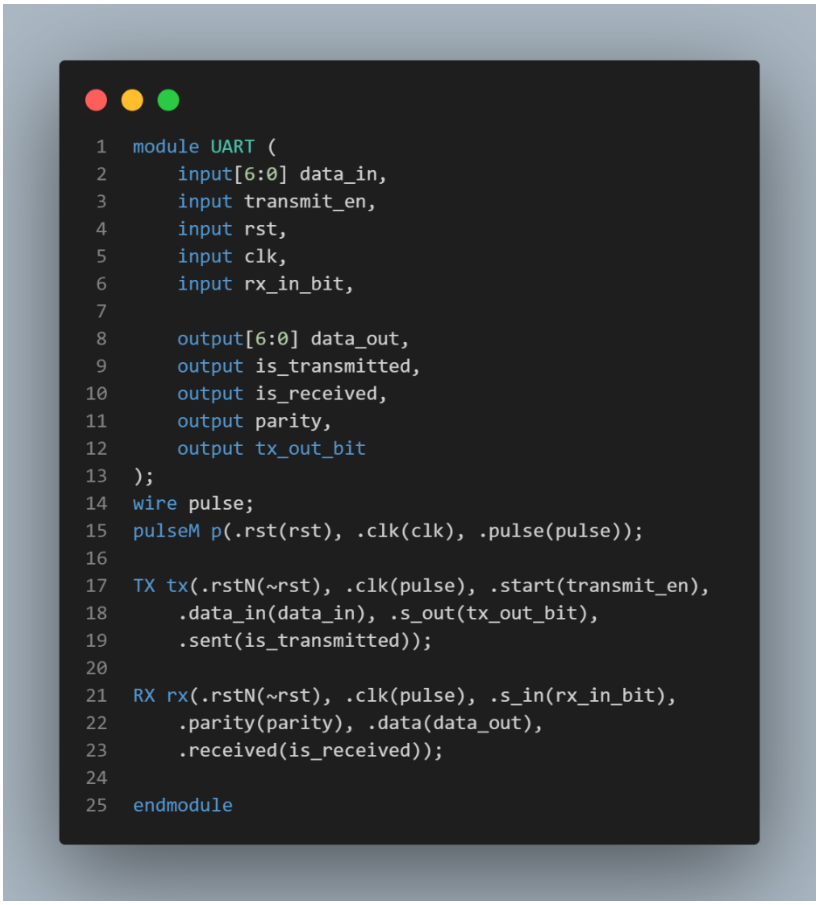
```
module pulseM(
input       rst,
input       clk,
output reg  pulse
);

localparam r = 217;
integer counter = 1;
always @(posedge clk) begin
    if(rst) begin
        counter = 1;
        pulse <= 0;
    end else begin
        if(counter == r) begin
            counter = 1;
            pulse <= !pulse;
        end else begin
            counter = counter + 1;
        end
    end
end

endmodule
```

حال یک ماژول کلی تعریف میکنیم و تمام دیگر اجزا را کنار هم می آوریم.

نکته مهم این است که ماژول نهایی باید بتواند همزمان هم دریافت و هم ارسال کند.



```
module UART (
    input[6:0] data_in,
    input transmit_en,
    input rst,
    input clk,
    input rx_in_bit,

    output[6:0] data_out,
    output is_transmitted,
    output is_received,
    output parity,
    output tx_out_bit
);
wire pulse;
pulseM p(.rst(rst), .clk(clk), .pulse(pulse));

TX tx(.rstN(~rst), .clk(pulse), .start(transmit_en),
    .data_in(data_in), .s_out(tx_out_bit),
    .sent(is_transmitted));

RX rx(.rstN(~rst), .clk(pulse), .s_in(rx_in_bit),
    .parity(parity), .data(data_out),
    .received(is_received));

endmodule
```

با استفاده از ماژول pulseM همانطور که گفتیم کلاک را تنظیم میکنیم.

همانطور که می بینم با سیگنال rst، تمام قطعه بصورت کلی ریست می شود. (می شد قطعه قطعه ریست کرد اما صرفا پیچیدگی و عمدم بهینگی می آورد)

برای حالت ارسال:

از TX نمونه گیری شده است و مکمل rst(اکتیو لو است) و کلاک را می دهیم.

یک سیگنال transmit_en قرار می دهیم برای هنگامی که بخواهیم داده ارسال کنیم.

(در این حالت همزمان داده فرستاده می شود و امکان دریافت داده نیز وجود دارد)

Data_in نیز ورودی فرستنده است. و با اتمام هر ارسالی، is_transmitted یک می شود.

حالت دریافت داده:

این حال همواره فعال است و به enable نیازی ندارد. ورودی های سریالی از طریق rx_in_bit دریافت شده و به این ماژول داده می شوند. این ماژول علاوه بر data_out که داده ها به صورت موازی است، بیت توازن و اینکه آیا دریافت کامل شده است را نیز نشان می دهد.

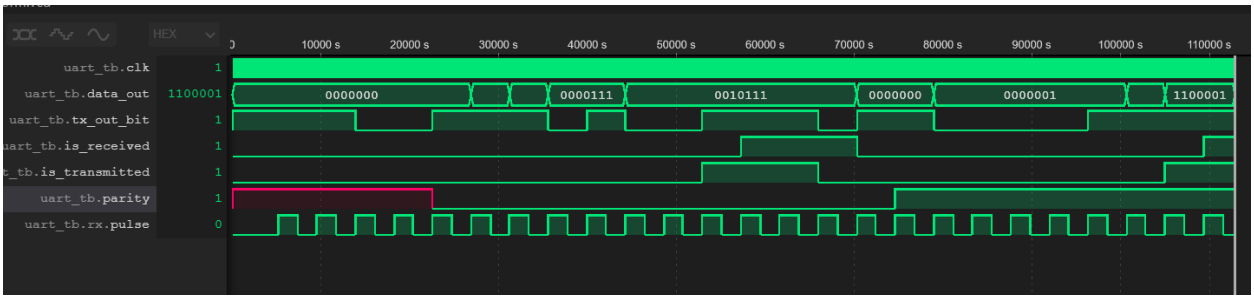
حال باید کل سیستم را تست کنیم:

در این تست، بازه کلاک 10 ثانیه است و به همین نسبت بازه تاخیر های مورد نیاز تا رخ دادن یک چرخه از پالس، 4340 ثانیه است.$(10 * \frac{50,000,000}{115200})$

پس از نمونه گیری و اختصاص سیگنال ها تست را می نویسیم. (کد تست بنچ ضمیمه شده است و اختصاص سیگنال ها در بالا توضیح داده شده است و نکته خاصی ندارد)

```
initial begin
    $dumpfile("waveform.vcd");
    $dumpvars(0, uart_tb);
    rst = 1;
    #3000
    rst = 0;
    #1340
    #4340
    data_in = 7'b0010111;
    #4340
    transmit_en = 1;
    #4340
    #4340
    #4340
    #4340
    #4340
    #4340
    #4340
    #4340
    #4340
    #4340
    #4340
    transmit_en = 0;
    #4340
    transmit_en = 1;
    data_in = 7'b1100001;
    #4340
    #4340
    #4340
    #4340
    #4340
    #4340
    #4340
    #4340
    #4340
    #4340
    #4340
    #5
    $finish;
end
```

ابتدا مقدار 0010111 را می فرستیم ولی چیزی دریافت نیم شود، transmit_en را فعال می کنیم و مشاهده می کنیم داده ها فرستاده می شوند. آنقدر کلاک می زنیم تا ارسال تمام شود.
یک بار دیگر نیز این روند را دنبال می کنیم تا مقدار 1100001 را منتقل کنیم.



کلاک را مشاهده کنید، این تاثیر کار با پالس ساختگیست که هرچرخه آن معادل 434 چرخه از کلاک اصلی است.

شکل نشان می دهد که هردو ارسال و دریافت به درستی صورت گرفته اند.

دقت کنید که سیگنال tx_out_bit، همان خروجی فرستنده است که ورودی rx_in_bit داده شده است.

در این تست، دو نمونه از UART گرفته شد و یکی به دیگری داده ارسال کرد.

حال با یک UART به خودش داده ارسال میکنیم تا ببینیم امکان همزمانی دریافت و ارسال برای این قطعه برقرار است:

```
UART tx(.clk(clk), .rst(rst), .data_in(data_in), .transmit_en(transmit_en),
 .is_transmitted(is_transmitted), .tx_out_bit(tx_out_bit), .data_out(data_out),
  .is_received(is_received), .rx_in_bit(tx_out_bit), .parity(parity));

```

بیت خروجی سریالی، به بیت ورودی سریالی وصل است.

بلاک initial را تغییری نمی دهیم و مانند بالا است: