

**Universidad Nacional de Córdoba.**  
**Facultad de Ciencias Exactas, Físicas y Naturales.**



Arquitectura de Computadoras

---

**Trabajo Práctico 2:**  
***UART***

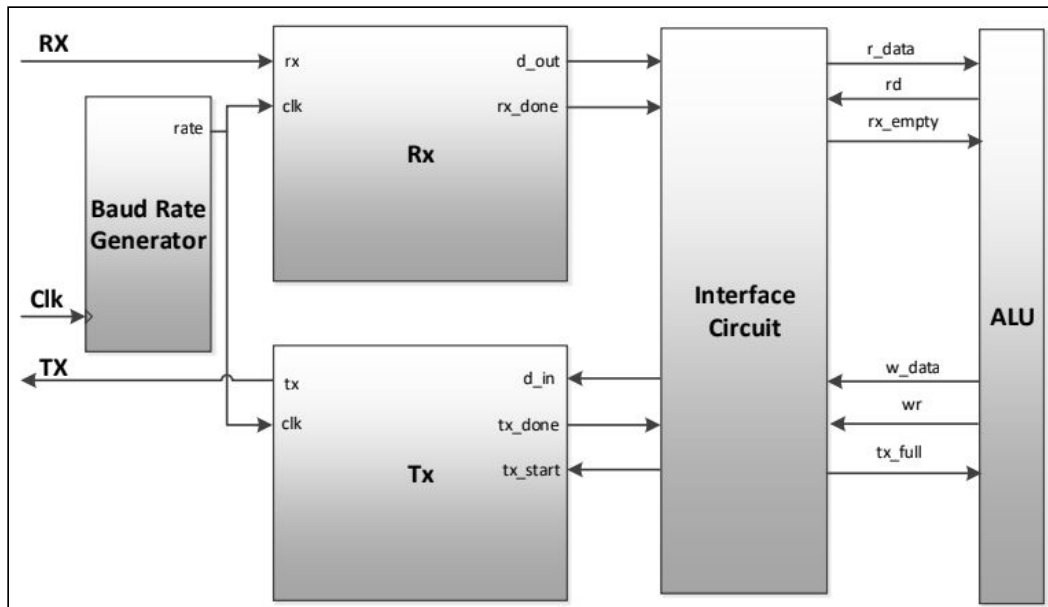
Integrantes:

- Drudi, Leandro.
- Goldman, Nicolás.

Github repository: <https://github.com/nicolasgoldman07/TPs-Arquitectura>

## CONSIGNA:

Para el segundo trabajo de la materia se pide desarrollar un módulo de UART transmisor y receptor, sus respectivos circuitos de interfaz y baud rate generator y complementarlo con la ALU del TP1 para poder intercambiar y trabajar datos. El circuito completo es el siguiente:



## DESARROLLO:

Para la implementación del trabajo, se realizaron 5 módulos además de los correspondientes a la ALU realizados para el TP1. Ellos son: **Top Level UART**, **Circuitos RX y TX**, **mod-m-counter** (Baud rate generator) y **flag-buf** (cómo circuito de interfaz).

### UART - SISTEMA DE RECEPCIÓN:

Para el desarrollo de los módulos de recepción se utilizó una técnica de oversampling, haciendo que cada bit sea muestreado 16 veces y estimar el punto medio de cada uno de ellos.

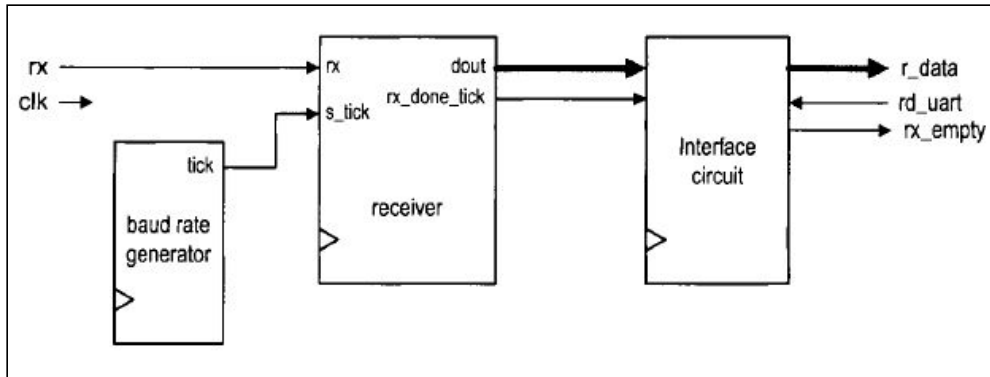
El esquema para dicho sobremuestreo funciona de la siguiente manera:

1. Espero que la señal se haga 0 y comienzo el contador de ticks (1 tick = 1 muestra).
2. Cuando el contador llega a 7, quiere decir que llegamos a la mitad del start bit, entonces reinicio el contador.
3. Cuando llegue nuestro contador a 15, nos encontraremos parado en el medio del sampleo del primer bit de data. Ahora resta tomar su valor y asignarlo a algún registro y reiniciar el contador.
4. Repetimos este último paso 7 veces y obtenemos el valor de todos los bits de data.

5. Repetimos el paso una vez más para obtener el bit de stop.

Este esquema funciona básicamente como una señal de clock, con la diferencia que usa los ticks de sampleo (en vez de usar los flancos de subida y bajada) para estimar el punto medio de cada bit.

Una pequeña representación del modelo de recepción es la siguiente:



En la que el UART receiver será el encargado de re-ensamblar los bits entrantes en palabras mediante el sobremuestreo explicado anteriormente, el baud rate generator será el módulo que nos permita generar los ticks de sampleo y por último el interface circuit, capaz de proveer un buffer y status entre el módulo UART y el módulo que se encuentre utilizando a este último.

### mod\_m\_counter MODULE (pág 128):

Este módulo es el baud rate generator, es decir, el encargado de generar los ticks que usaremos en nuestro modelo de sobremuestreo. Debe realizarlos a una frecuencia de 16 veces el baud rate.

Para una baud rate de 19200, la tasa de sampleo debe ser de 307200 ticks por segundo.

Usando el clock a **100MHz**, necesitamos un contador de  $\frac{100MHz}{307200}$  ticks en el que 1 “tick” se genera cada 326 pulsos de reloj.

```
1 `timescale 1ns / 1ps
2
3 module mod_m_counter
4   #(
5     parameter M = 326, //mod-M163
6     parameter N = 9    //8 bits in counter
7   )
8
9   (
10    input wire clk,
11    input wire reset,
12
13    output wire max_tick,
14    output wire [N-1:0] q
15  );
16
17  reg [N-1:0] r_reg;
18  wire [N-1:0] r_next;
```

```
19
20  always@(posedge clk)
21  begin
22    if (reset) begin
23      r_reg <= 0;
24    end else begin
25      r_reg <= r_next;
26    end
27  end
28
29  //next-state logic
30  assign r_next = (r_reg==(M-1)) ? 0 : r_reg + 1;
31
32  //output
33  assign q = r_reg;
34  assign max_tick = (r_reg==(M-1)) ? 1'b1 : 1'b0;
35
36
37 endmodule
```

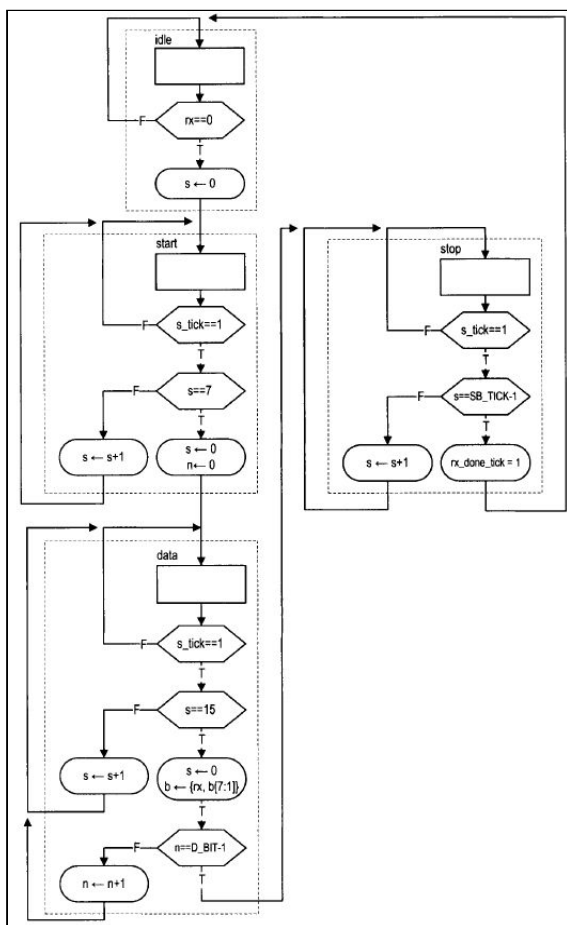
En el código de arriba se puede ver el módulo del generador de ticks, el cual tiene un `always` encargado de asignarnos el valor de `r_next` a `r_reg` en cada uno de los pulsos de reloj. La lógica funciona simplemente con una sentencia condicional en la que si `r_reg` es 325, en reiniciamos el `r_next` a cero, sino lo aumentamos en 1.

A la salida tendremos `q`, contenedor del valor en el `r_reg` y `max_tick` que nos lleva registro de si hemos alcanzado el contador y tenemos nuestro tick completo, o si todavía faltan pulsos de reloj.

## RX MODULE:

El receptor es aquel módulo que debe encargarse de tomar los datos enviados bit a bit por el Tx y encargarse de re-ensamblarlos para obtener la palabra enviada.

Para hacerlos, se realiza una máquina de estados, que será la encargada de movernos entre los bits de **start**, **data** y **stop**. Se muestra la misma en la siguiente imagen:



En la máquina, podemos notar claramente los 4 estados que disponemos.

Primero un estado de **idle** en el que nos encontramos esperando por el start bit en cero.

Cuando esto ocurre, ponemos el contador de ticks “s” en 0 y avanzamos al estado de **start**.

Ya en este estado, como se explicó con el esquema de sobremuestreo, cuando llegamos a 7 en el contador, nos encontramos parados en el punto medio del bit de comienzo. Sucedido esto, reiniciamos el contador, guardamos el valor en **n** y nos vamos al estado de **data**. Una vez que lleguemos al contador en 15, nos encontramos parados en la mitad del primer bit de data, por lo que reiniciamos el contador de ticks y guardamos el valor en el MSB de **b** concatenándolo con los 7 MSB del mismo para movernos en la ventana. Si repetimos esto 7 veces más, tendremos la

recepción de los 8 bits de data y pasaremos al estado de **stop** en el que cuando lo hayamos recibido completo, nos moveremos nuevamente a idle con la bandera de **rx\_done** en 1.

```

1  `timescale 1ns/1ps
2
3  module uart_rx
4  #(
5      parameter DBIT = 8,      // # N data bits
6      parameter SB_TICK = 16   // # ticks for stop bits
7  )
8  (
9      input wire      clk, reset,
10     input wire      rx, s_tick,
11     output reg       rx_done_tick,
12     output wire [7:0] dout
13 );
14
15 // symbolic state declaration
16 localparam [1:0] idle = 2'b00,
17 localparam [1:0] start = 2'b01,
18 localparam [1:0] data = 2'b10,
19 localparam [1:0] stop = 2'b11;
20
21 // signal declaration
22 reg [1:0] state_reg, state_next;
23 reg [3:0] s_reg, s_next;      // Sampling t
24 reg [2:0] n_reg, n_next;      // Number of d
25 reg [7:0] b_reg, b_next;      // Retrieved b
26
27 // body
28 // FSMD state & data registers
29 always @(posedge clk, posedge reset)
30 begin
31     if (reset)
32     begin
33         state_reg <= idle;
34         s_reg <= 0;
35         n_reg <= 0;
36         b_reg <= 0;
37     end
38     else
39     begin
40         state_reg <= state_next;
41         s_reg <= s_next;
42         n_reg <= n_next;
43         b_reg <= b_next;
44     end
45 end
46
47 // FSMD next_state logic
48 always @(*)
49 begin
50     state_next = state_reg;
51
52     rx_done_tick = 1'b0;
53     s_next = s_reg;
54     n_next = n_reg;
55     b_next = b_reg;
56     case (state_reg)
57         idle :
58             if (~rx)
59             begin
60                 state_next = start;
61                 s_next = 0;
62             end
63             start :
64                 if (s_tick)
65                 if (s_reg==7)
66                 begin
67                     state_next = data;
68                     s_next = 0;
69                     n_next = 0;
70                 end
71                 else
72                     s_next = s_reg + 1;
73             data :
74                 if (s_tick)
75                 if (s_reg==15)
76                 begin
77                     s_next = 0;
78                     b_next = {rx, b_reg [7:1]};
79                     if (n_reg==(DBIT-1))
80                     state_next = stop ;
81                     else
82                         n_next = n_reg + 1;
83                 end
84                 else
85                     s_next = s_reg + 1;
86             stop:
87                 if (s_tick)
88                 if (s_reg==(SB_TICK-1))
89                 begin
90                     state_next = idle;
91                     rx_done_tick =1'b1;
92                 end
93                 else
94                     s_next = s_reg + 1;
95             endcase
96         end
97 // output
98 assign dout = b_reg;
99
100 endmodule

```

En el código de la imagen de arriba, se muestra la máquina de estados con su respectiva lógica capaz de modelar un sistema de transmisión de UART.

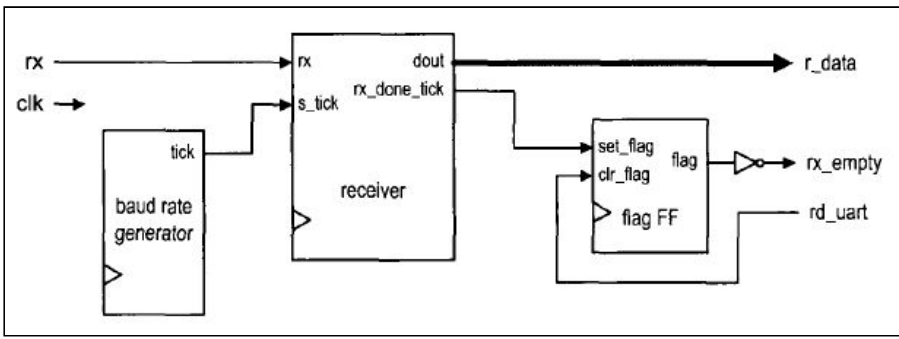
### flag\_buf MODULE:

Este es nuestro módulo nos permitirá señalar la disponibilidad para poder recibir una nueva palabra, prevenir de realizar la recepción de una misma palabra múltiples veces y en algunos casos, proveernos un buffer entre el receptor y el sistema principal (modelos con one-word buffer o FIFO).

Si revisamos la explicación del receptor, tenemos un **rx\_ready\_tick** que se pone en 1 cuando recibimos el último bit de todos (stop bit). Este está conectado al **set\_flag** de nuestro interface circuit para setear la bandera del arribo de una nueva palabra.

El sistema que se encuentre utilizando la UART se encargará de revisar esta salida para darse cuenta cuando recibe una palabra y reasignar la salida con **flag** (cero) un pulso de clock después.

El siguiente esquema muestra una representación del módulo de interfaz entre UART y el main system.



### UART - SISTEMA DE TRANSMISIÓN:

El módulo transmisor es esencialmente como una especie de shift register que carga data en paralelo y las envía hacia afuera, bit por bit, a una determinada tasa de envío.

La transmisión comienza con un **start bit**, que es un 0 seguido de todos los **bits de data** y finaliza con el o los **stop bits**, en 1. La siguiente imagen, nos muestra un modelo de dicha transmisión:

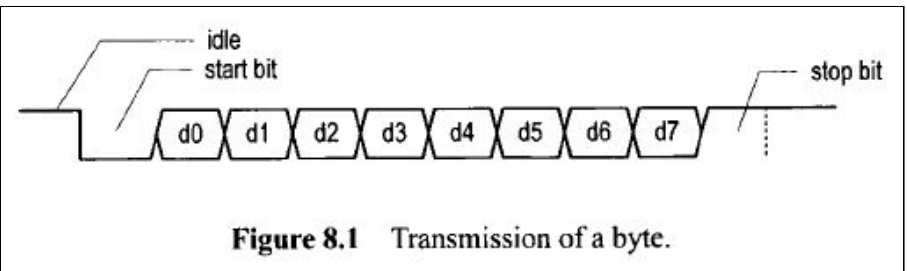


Figure 8.1 Transmission of a byte.

Antes de que comience la transmisión, los módulos de *tx* y *rx* deber acordar con trabajar siguiendo algunos parámetros, entre ellos, la tasa de baudios (bits por segundo) que se utilizará, cantidad de *bits de dato* y de *stop* y el uso del *bit de paridad* (opcional, para la detección de errores). Nuestro transmisor cuenta con 8 bits de data, 1 stop bit, no tiene bit de paridad y tiene una tasa de baudios de 19200.

### uart\_tx MODULE:

El sistema de transmisión en UART es muy parecido al de recepción. Consiste en un módulo transmisor, un generador de ticks y la correspondiente interfaz.

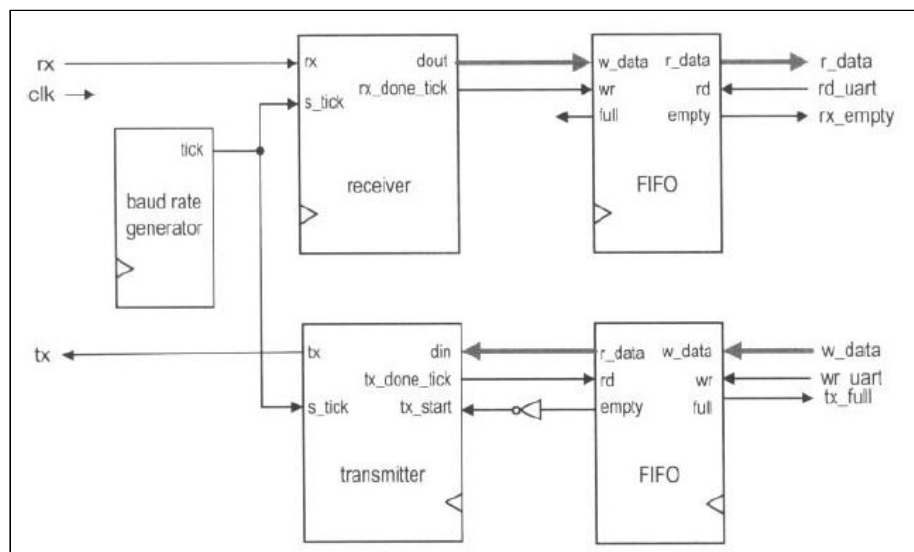
En cuanto al módulo de interfaz, es prácticamente igual al del receptor, con la diferencia de que el **set** lo hace el main system y el **clr** lo hace el módulo UART.

Para no tener que crear su propio módulo de baud rate generator, utiliza el del receptor creando sus propios contadores. Se desplaza un bit hacia afuera cada 16 ticks.

El módulo en sí, es muy parecido al del receptor. Ante la recepción del bit de **tx\_start**, nos movemos a los estados de **start**, **data** y **stop** colocándolos en el medio de cada uno de los correspondientes bits y enviándolo mediante la utilización del **tx\_reg**.

### MÓDULO TOP LEVEL UART:

El diagrama del módulo del core de UART es el siguiente, reemplazando el sistema de interfaz de FF con buffer por uno simplemente de flags FF:



Para el desarrollo del módulo en Verilog, basta con inicializar cada uno de los módulos explicados previamente además del encargado de manejar la interfaz entre el módulo de UART y el de la ALU realizado para el primer práctico de la materia. Por último, en este módulo asignamos la salida de la interfaz a los leds de la FPGA para poder visualizar alguno de los resultados en la práctica.

El módulo que actúa como interfaz es el que más *complejidad* tiene en su interior. Cuenta con dos máquinas de estado, una para poder movernos entre los estados posibles (cuando nos llega el dato de A, B, OP y mientras se realiza el cálculo de la operación); y la segunda para la lógica de cada uno de ellos.

Se dejan las imágenes de ambas máquinas de estado.

La encargada de **movernos entre los diferentes estados** no es muy compleja. Si estamos en el primer estado, quiere decir que estamos esperando el valor de A por uart. Cuando llega, guarda dicho valor en una variable temporal para evitarnos cualquier tipo de problemas de sobreescritura. Así sucesivamente con los demás estados.

La **máquina con lógica** comprueba primero que el rx\_done\_tick este en 1 para darnos cuenta que nos llegó un valor y guarda el valor de rx\_data correspondiente al ingresante de UART.

Para el caso del cálculo, pone en 1 el transmisor y vuelva a quedarse esperando por A nuevamente.

```
// Maquina de estados
always @ (posedge clk)
begin
    if (reset)
        begin
            reg_a <= 8'b00000000;
            reg_b <= 8'b00000000;
            reg_op <= 8'b00000000;
            state_reg <= first_state;
            //state_next <= first_state;
        end
    else
        begin
            case(state_reg)
                first_state :
                    reg_a <= reg_a_tmp;
                second_state :
                    reg_b <= reg_b_tmp;
                op_state :
                    reg_op <= reg_op_tmp;
                calc_state :
                    reg_tx_data <= reg_tx_data_tmp;
            endcase
            state_reg <= state_next;
            reg_tx_start <= reg_tx_start_t;
        end
    end
end
```

```
// Logica de cambio de estados
always @ (*)
begin
    state_next = state_reg;
    case(state_reg)
        first_state :
            begin
                reg_tx_start_t = 1'b0;
                if(rx_done_tick)
                    begin
                        reg_a_tmp = rx_data;
                        state_next = second_state;
                    end
                else
                    begin
                        state_next = first_state;
                    end
            end
        second_state :
            begin
                reg_tx_start_t = 1'b0;
                if(rx_done_tick)
                    begin
                        reg_b_tmp = rx_data;
                        state_next = op_state;
                    end
                else
                    begin
                        state_next = second_state;
                    end
            end
        op_state :
            begin
                reg_tx_start_t = 1'b0;
                if(rx_done_tick)
                    begin
                        state_next = calc_state;
                        reg_op_tmp = rx_data;
                    end
                else
                    begin
                        state_next = op_state;
                    end
            end
        calc_state :
            begin
                reg_tx_data_tmp = calc_data;
                state_next = first_state;
                reg_tx_start_t = 1'b1;
            end
    endcase
end
```