TRABAJO PRACTICO 2
Arquitectura de computadoras

Recepción y Transmisión Asincrónica Universal - UART

Integrantes:

- Sosa Ludueña Gabriel
- Cazajous Miguel

2016

Introducción

La UART[1] es el dispositivo o circuito encargado de controlar los puertos y dispositivos serie. Se encuentra integrado en la placa base o en la tarjeta adaptadora del dispositivo.

El controlador del UART es el componente clave del subsistema de comunicaciones series de una computadora. El UART toma bytes de datos y transmite los bits individuales de forma secuencial (módulo transmisor de la UART). En el destino, un segundo UART reensambla los bits en bytes completos (módulo receptor de la UART). La transmisión serie de la información digital (bits) a través de un cable único u otros medios es mucho más efectiva en cuanto a costo que la transmisión en paralelo a través de múltiples cables. Se utiliza un UART para convertir la información transmitida entre su forma secuencial y paralela en cada terminal de enlace. Cada UART contiene un registro de desplazamiento que es el método fundamental de conversión entre las forma secuencial y paralela.

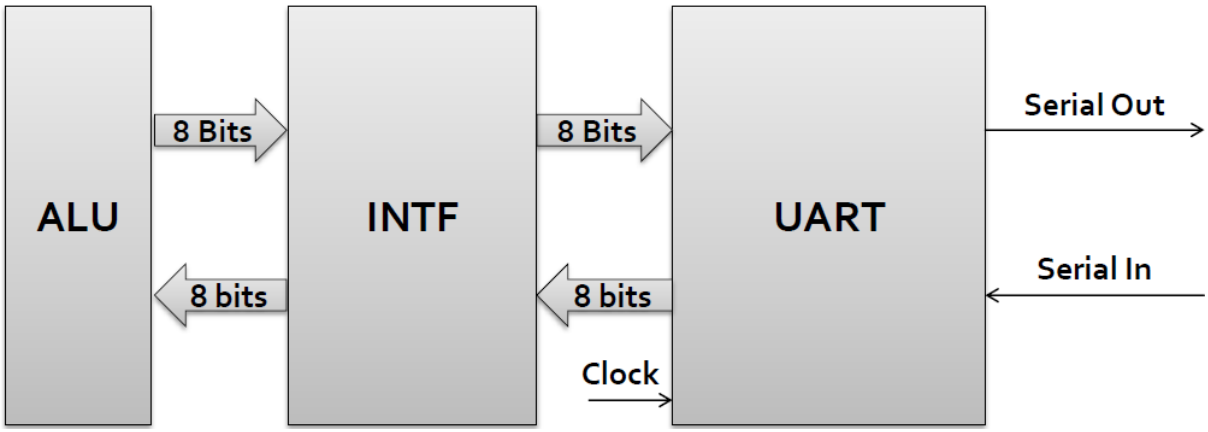
El UART normalmente no genera directamente o recibe las señales externas entre los diferentes módulos del equipo. Usualmente se usan dispositivos de interfaz separados para convertir las señales de nivel lógico del UART hacia y desde los niveles de señalización externos.

Las señales externas pueden ser de variada índole. Ejemplos de estándares para señalización por voltaje son RS-232, RS-422 y RS-485 de la EIA. Históricamente se usó la presencia o ausencia de corriente en circuitos telegráficos.

Algunos esquemas de señalización no usan cables eléctricos; ejemplo de esto son la fibra óptica, infrarrojo (inalámbrico) y Bluetooth (inalámbrico). Algunos esquemas de señalización emplean una modulación de señal portadora (con o sin cables); por ejemplo, la modulación de señales de audio con módems de línea telefónica, la modulación en radio frecuencia (RF) en radios de datos y la DC-LIN para la comunicación de línea eléctrica.

Consigna de trabajo

Se pide establecer una comunicación entre una PC y una placa de desarrollo (FPGA) a través de un módulo UART para lograr manipular una ALU (Unidad Aritmética Lógica) de la placa FPGA desde la PC. La PC se deberá adaptar a una interfaz que se encuentra en la FPGA para vincular la información recibida por el módulo UART (datos en código ASCII) y lograr su adaptación a los datos y códigos de operaciones (datos binarios asociados a la ALU), de manera que con dicha interfaz se logra vincular la información recibida para ser almacenada en la ALU que por último obteniendo el resultado la interfaz se encargará de enviar dicho resultado a la PC a través del módulo UART.

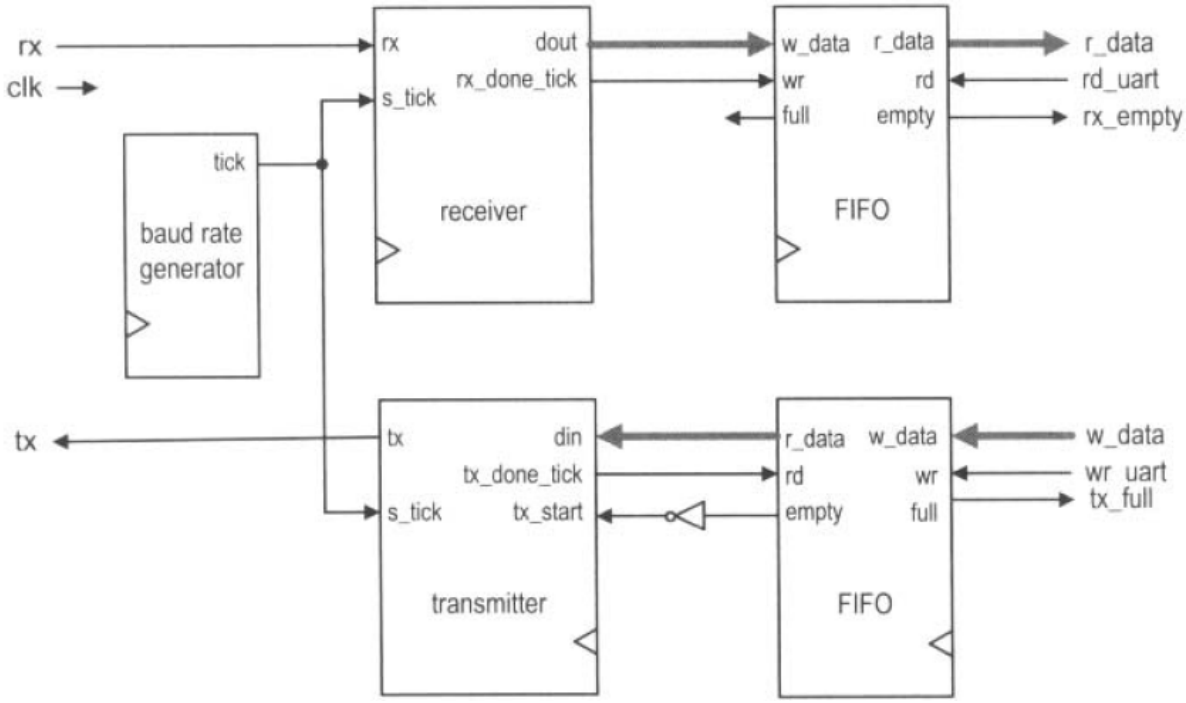


Para llevar esto a cabo se debe desarrollar un bloque ALU, un bloque UART y un bloque INTF (interfaz) que vincule los dos anteriores.

Por el lado del bloque ALU se deberá utilizar el mismo bloque ALU realizado en el TP1 de la materia (enlace externo).

Para el bloque UART se deberá hacer uso del concepto de máquinas de estado para la codificación de los módulos que lo componen y poder establecer una comunicación serial con la PC.

Y por último se deberá implementar un bloque interfaz INTF que será el encargado de vincular el bloque UART con el bloque ALU según la transmisión que realice la PC.



El bloque UART será el encargado de recibir los valores que se envían desde la PC y este por medio de un bloque de interfaz le comunicará esta información al bloque ALU que realizará las operaciones convenientes y devolverá un resultado que será devuelto a la PC en respuesta a las órdenes recibidas.

Desarrollo

Módulo Baut Rate Generator

Debido a que no hay un reloj que permita decidir cuando el dato (bit) está disponible a la entrada, el receptor usará un sistema de sobremuestreo (o oversampling) que permite estimar el punto medio del bit transmitido y adquirirlo en ese instante. Este módulo es el encargado de generar las señales para el procedimiento de sobremuestreo.
Su función será generar un “tick” 16 veces por baud rate.
Como sabemos de la teoría, si el baud rate es de 19.200 ciclos por segundo, la frecuencia de muestreo debe ser 19.200 *16= 307.200 ticks por segundo. Si el reloj de la placa (en spartan) es de 50 Mhz, entonces hay que generar un tick cada 163 ciclos de reloj.

Clock/(BaudRate * 16) ≈ 163

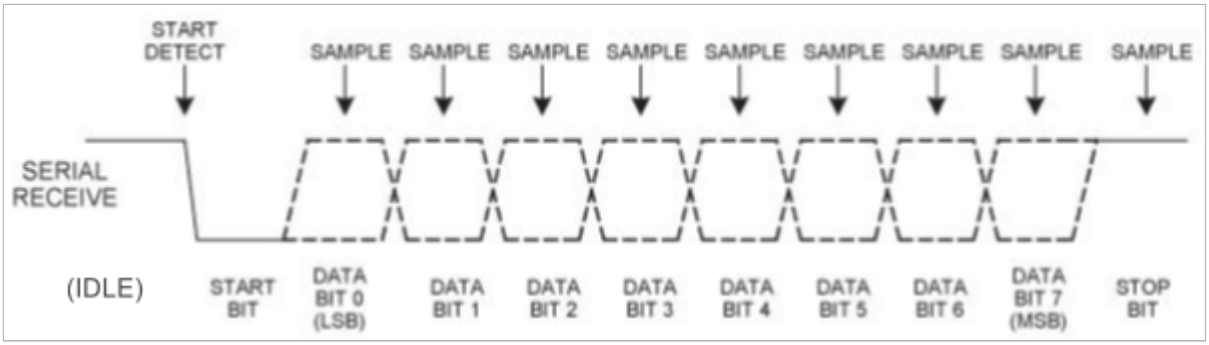
Por lo que concluimos en que que el Baut Rate Generator es un contador módulo 163.
El módulo en verilog para la FPGA quedó de la siguiente manera

```
1.  module Baut_Rate_Generator
2.  (clock, tick);
3.
4.     // Entrada
5.     input clock;
6.
7.     // Salida
8.     output reg tick = 0;
9.
10.    // Variables internas
11.    reg [8:0] cont=0;
12.
13.    // Lógica secuencial
14.    always@(posedge clock)
15.    begin
16.       if(cont == 163) begin
17.          tick = 1; // se cambia estado de pulso, y reinicia cont
18.          cont = 0; end
19.       else begin
20.          cont = cont + 1; // sigue contando
21.          tick = 0;end
22.    end
23.
24.
25. endmodule
```

Módulo Receptor Rx[3]

De manera resumida el receptor Rx, realiza un proceso inverso al del transmisor, es decir que recibirá los bit de la trama UART uno a uno y detecta los bits de datos para reunirlos nuevamente en el dato completo de 8 bits.

Para la máquina de estado FSM del receptor tenemos en cuenta el diagrama de estado de la misma, el cual está asociado a la trama UART lo que nos permite diferenciar cada uno sus estados. A continuación vemos un ejemplo de cómo se recibiría la trama UART y cual seria cada uno de los estados a tener en cuenta



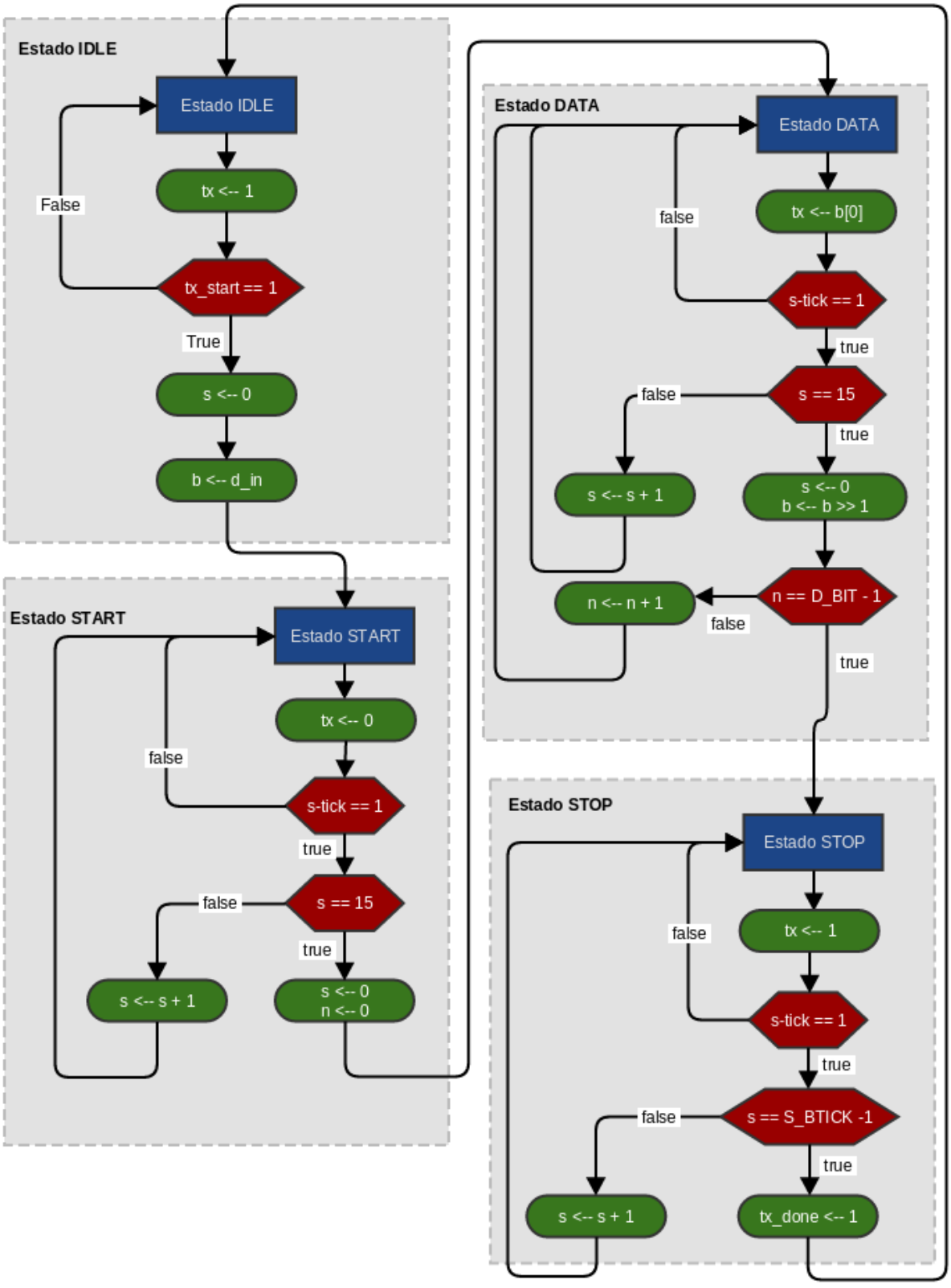
Como se observa inicialmente la la FSM del receptor debe tener un estado de espera o desocupado (IDLE) hasta detectar el bit START para luego saber que siguen los bits de datos a receptar, que para nuestro caso serán 8 bits de datos, y por último se encuentra el bit de STOP omitiendo el bit de PARIDAD ya que es opcional. Por lo cual la FSM Rx contiene 4 estados posible que son

- **IDLE:** estado en el que el receptor se queda desocupado o a la espera de la detección del bit de START de la trama UART en la recepción.
- **START:** estado en el que se está recibiendo el bit START, de la trama UART, para el inicio de una nueva recepción de datos.
- **DATA:** estado en el que se van recibiendo y almacenando cada uno de los 8 bits de dato de la trama UART.
- **STOP:** estado en el que se recibe el bit de finalización (o STOP), de la trama UART, para la recepción de datos.

El comportamiento de cada estado de la FSM Rx respeta el diagrama de flujo y estados de Rx.

Diagrama de flujo y estados en Rx[3]

Para la representación de máquinas de estado finitas FSM se puede utilizar el diagrama de estado o también el método de máquina de estado algorítmica (ASMD)[2].



La explicación al diagrama del módulo Rx consiste en, dado un baud rate de transmisión, se genera una frecuencia que es 16 veces el baud rate, y se sigue el siguiente procedimiento:

- Esperar que la señal se ponga a “0”, el principio del bit de start, e iniciar el conteo.
- Cuando el conteo llega a 7, quiere decir que estamos en la mitad del start bit, se resetea el contador a cero.
- Cuando el contador llega a 15, quiere decir que estamos en la mitad del primer bit de datos. Se obtiene el bit y se resetea el contador.
- Repetir el paso 3 por cada bit de datos.
- Si hay bit de paridad, repetir el paso 3.
- Repetir el paso 3 para el bit de Stop.

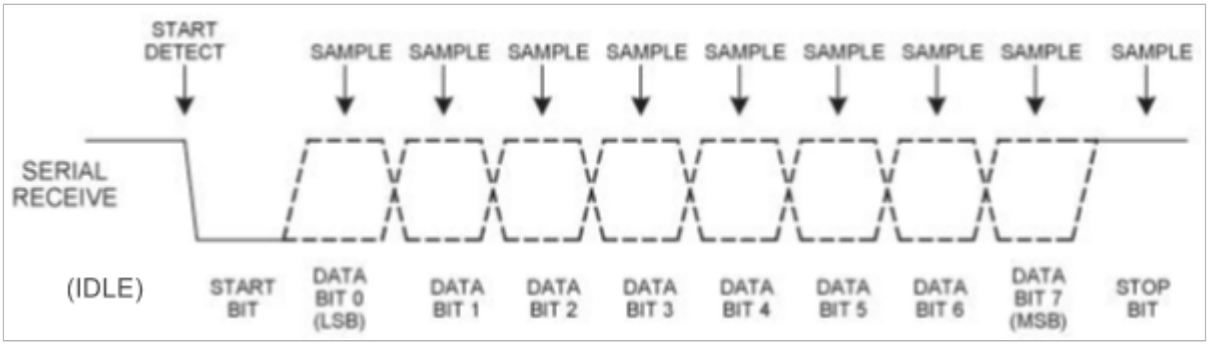
El código verilog para la FPGA de este módulo es el siguiente

```
1.  module Rx
2.  #(parameter N = 8,    // # DBIT: n° bits de dato
3.            M = 1  // # SB_TICK: n° de ticks de bit stop
4.    )
5.  (clk, reset,rx, s_tick_clk,rx_done_tick,dout);// E/S - I/O
6.
7.    // Entradas
8.     input wire clk, reset; // clk: clock de FPGA - reset: señal para resetear FSM
9.     input wire rx, s_tick_clk; // rx: recepcion - s_tick_clk: clock de baud rate gen.
10.
11.     // Salidas
12.    output reg rx_done_tick; // bandera de recepcion
13.    output wire [7:0] dout; // salida de bit de datos recibidos
14.
15.    // Declaracion de estados
16.   localparam [3:0]
17.     idle  = 4'b0001,// desocupado - esperando
18.     start = 4'b0010,// inicio
19.     data  = 4'b0100,// dato
20.     stop  = 4'b1000;// detenido
21.
22.   // Declaracion de señales con inicializacion en cada una
23.   reg [3:0] state, state_next=idle;// estados de 4 bits
24.   reg [3:0] s_reg, s_next=0; //s_reg: Tick counter -> cuenta los ticks desde el inicio,
    maxima cuenta 15(4 bits)
25.   reg [2:0] n_reg, n_next=0; //n_reg: data counter -> cuenta los bits de datos leidos,
    maxima cuenta 7(3 bits)
26.   reg [7:0] b_reg, b_next=0; //b_reg: registro de bits de datos (8 bits de datos)
27.
28.   // Estados de la FSM & Resgistros de datos
29.   always @(posedge clk, posedge reset)
30.     if (reset)
31.       begin
32.         state <= idle;
33.         s_reg <= 0;
34.         n_reg <= 0;
35.         b_reg <= 0;
36.       end
37.     else
38.       begin
39.         state <= state_next;
40.         s_reg <= s_next;
41.         n_reg <= n_next;
42.         b_reg <= b_next;
43.       end
44.
```

```
45.  // Logica de siguiente estado
46.  always @*
47.  begin
48.    state_next = state;
49.    rx_done_tick = 1'b0;
50.    s_next = s_reg;
51.    n_next = n_reg;
52.    b_next = b_reg;
53.    case (state)
54.      idle:
55.        if (~rx)
56.          begin
57.            state_next = start; // estado sig -> start
58.            s_next = 0; // tick counter inicia en cero
59.          end
60.      start:
61.        if (s_tick_clk)
62.          if (s_reg==7)
63.            begin
64.              state_next = data;// estado sig -> data
65.              s_next = 0;// se reinicia tick counter
66.              n_next = 0;// inicia en cero contador de bit de datos
67.            end
68.          else
69.            s_next = s_reg + 1;// aumenta en uno tick counter
70.      data:
71.        if (s_tick_clk)
72.          if (s_reg==15)
73.            begin
74.              s_next = 0;// se reinicia tick counter por cada bit
75.              b_next = {rx, b_reg[7:1]};
76.              if (n_reg==(N-1))
77.                state_next = stop ;// estado sig -> stop
78.               else
79.                n_next = n_reg + 1;// aumenta en uno contador de bit de datos
80.             end
81.          else
82.            s_next = s_reg + 1;// aumenta en uno tick counter
83.      stop:
84.        if (s_tick_clk)
85.          if (s_reg==(M-1))
86.            begin
87.              state_next = idle;
88.              rx_done_tick =1'b1;
89.            end
90.          else
91.            s_next = s_reg + 1;
92.     endcase
93.  end
94.
95.   // Logica de Salida
96.  assign dout = b_reg;
97.
98.
99. endmodule
```

Módulo Transmisor Tx[3]

De manera resumida el transmisor Tx es básicamente un shift register que carga los datos en paralelo a trasmitir y luego los shiftea (desplaza) uno a uno a la velocidad dada por el *baud rate generator* comenzando desde el bit LSB.
Para la FSM del transmisor Tx tenemos en cuenta la trama UART que se analizó en el receptor Rx pero suponiendo que de esa manera debe ser enviada dicha trama, entonces se deberán respetar de nuevo los mismos estados que se tienen en el receptor ya que parten de la trama UART.



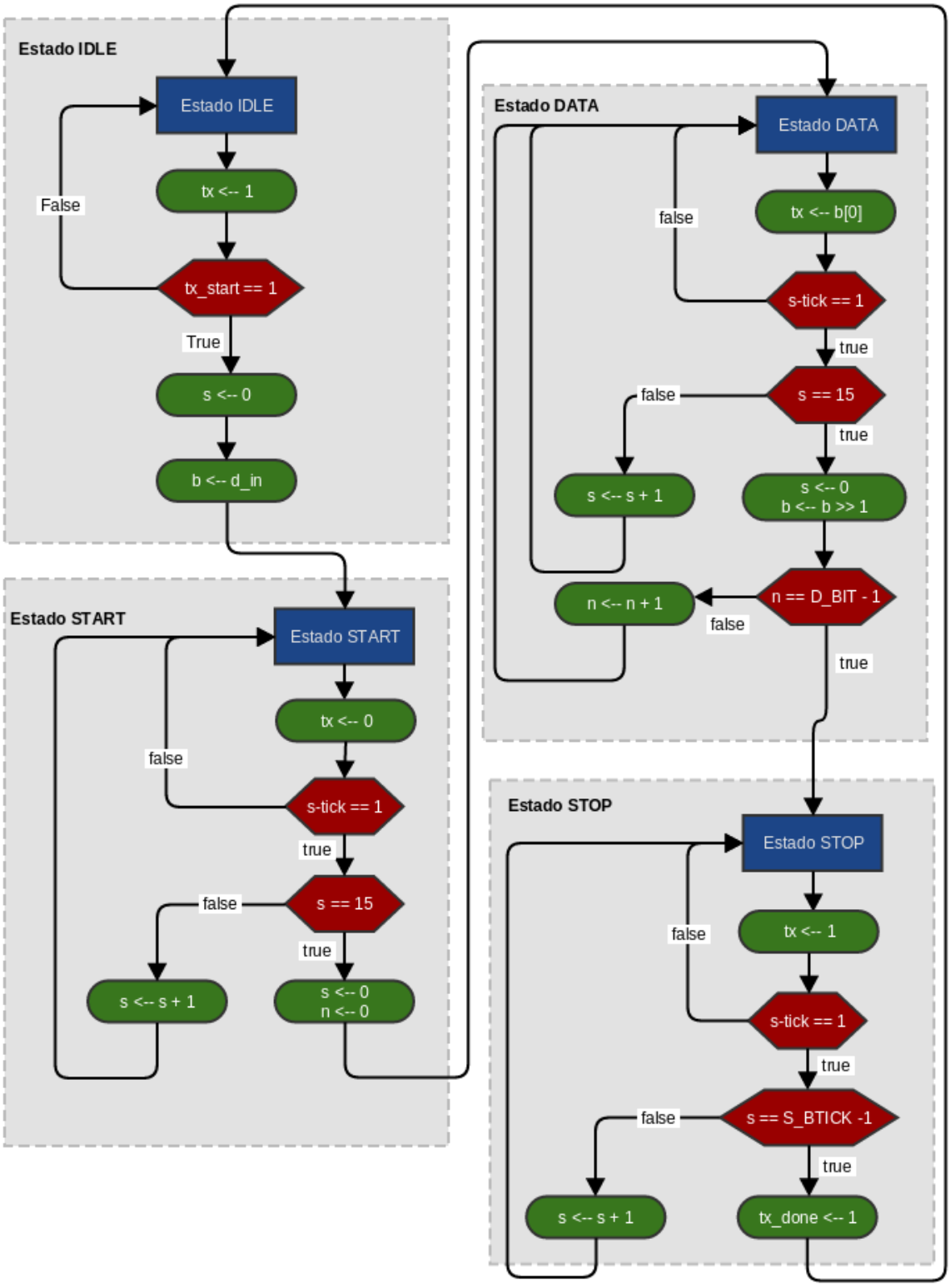
Como se observa inicialmente la la FSM del transmisor Tx debe tener un estado de espera o desocupado (IDLE) hasta detectar el bit START para luego saber que siguen los bits de datos a transmitir, que para nuestro caso serán 8 bits de datos, y por último se encuentra el bit de STOP omitiendo el bit de PARIDAD ya que es opcional. Por lo cual la FSM Rx contiene 4 estados posible que son

- **IDLE:** estado en el que el transmisor se queda desocupado o a la espera de la detección del bit de START en transmisión.
- **START:** estado en el que se transmite el bit de inicio para la transmisión de los bits de la trama UART.
- **DATA:** estado en el que se van transmitiendo cada uno de los 8 bits de dato de la trama UART.
- **STOP:** estado en el que se transmite el bit de finalización de la trama UART.

El comportamiento de cada estado de la FSM Tx respeta el diagrama de flujo y estados de Tx que se forma a partir de los estados que se deben contemplar en la trama UART.

Diagrama de flujo y estado del Tx[3]

Para la representación de la FSM del transmisor Tx se utilizó ASMD[2], que es el mismo método que se utilizó para el receptor Rx.



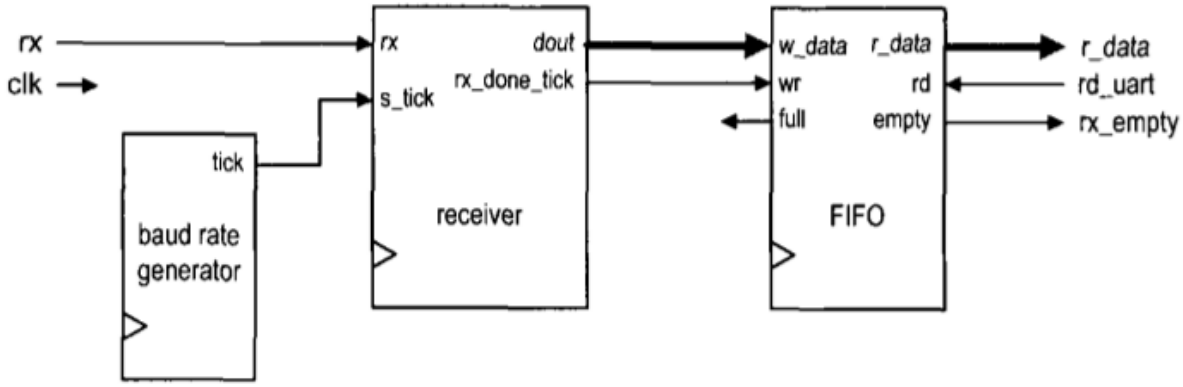
El código verilog para la FPGA de este módulo es el siguiente

```
1.  module Tx
2.  #(parameter N = 8,    // # DBIT: n° bits de dato
3.            M = 1  // # SB_TICK: n° de ticks de bit stop
4.    )
5.  (clk, reset, tx_start, s_tick, din, tx_done_tick, tx);
6.
7.     // Entradas
8.     input wire clk, reset; // clk: clock de FPGA - reset: señal para resetear FSM
9.    input wire tx_start, s_tick;// tx_start: señal bit de inicio - s_tick_clk: clock de baud rate
    gen.
10.   input wire [7:0] din; // din: bits de datos recibidos en paralelo
11.
12.    // Salidas
13.    output reg tx_done_tick;
14.   output wire tx;
15.
16.   // Declaracion de estados
17.   localparam [3:0]
18.     idle  = 4'b0001,// esperando señal de inicio (START)
19.     start = 4'b0010,// transmitiendo bit de inicio (START)
20.     data  = 4'b0100,// trasmitiendo bits de datos
21.     stop  = 4'b1000;// trasmitiendo bit de STOP
22.
23.   // Declaracion de señales con inicializacion en cada una
24.    reg [3:0] state, state_next=idle;// estados de 4 bits
25.   reg [3:0] s_reg, s_next=0; //s_reg: Tick counter -> cuenta los ticks desde el inicio,
    maxima cuenta 15(4 bits)
26.   reg [2:0] n_reg, n_next=0; //n_reg: data counter -> cuenta los bits de datos leidos,
    maxima cuenta 7(3 bits)
27.   reg [7:0] b_reg, b_next=0; //b_reg: registro de bits de datos (8 bits de datos)
28.   reg tx_reg, tx_next=1; // tx_reg: reg de señal de bit a trasmitir
29.
30.   // Estados de la FSM & Resgistros de datos
31.   always @(posedge clk, posedge reset)
32.     if (reset)
33.       begin
34.         state <= idle;
35.         s_reg <= 0;
36.         n_reg <= 0;
37.         b_reg <= 0;
38.         tx_reg <= 1'b1;
39.       end
40.     else
41.       begin
42.         state <= state_next;
43.         s_reg <= s_next;
44.         n_reg <= n_next;
45.         b_reg <= b_next;
46.         tx_reg <= tx_next;
47.       end
48.
49.  // Logica de siguiente estado
50.   always @*
51.   begin
52.     state_next = state;
53.     tx_done_tick = 1'b0;
54.     s_next = s_reg;
55.     n_next = n_reg;
56.     b_next = b_reg;
57.     tx_next = tx_reg ;
58.     case (state)
59.       idle:
60.         begin
```

```
61.           tx_next = 1'b1;
62.           if (tx_start)
63.             begin
64.               state_next = start;
65.               s_next = 0;
66.               b_next = din;
67.             end
68.         end
69.       start:
70.         begin
71.           tx_next = 1'b0;
72.           if (s_tick)
73.             if (s_reg==15)
74.               begin
75.                 state_next = data;
76.                 s_next = 0;
77.                 n_next = 0;
78.               end
79.             else
80.               s_next = s_reg + 1;
81.         end
82.       data:
83.         begin
84.           tx_next = b_reg[0];
85.           if (s_tick)
86.             if (s_reg==15)
87.               begin
88.                 s_next = 0;
89.                 b_next = b_reg >> 1;
90.                 if (n_reg==(N-1))
91.                   state_next = stop ;
92.                 else
93.                   n_next = n_reg + 1;
94.               end
95.             else
96.               s_next = s_reg + 1;
97.         end
98.       stop:
99.         begin
100.              tx_next = 1'b1;
101.              if (s_tick)
102.                if (s_reg==(M-1))
103.                  begin
104.                    state_next = idle;
105.                    tx_done_tick = 1'b1;
106.                  end
107.                else
108.                  s_next = s_reg + 1;
109.            end
110.        endcase
111.      end
112.
113.      // Logica de Salidas
114.      assign tx = tx_reg;
115.
116.
117.  endmodule
```

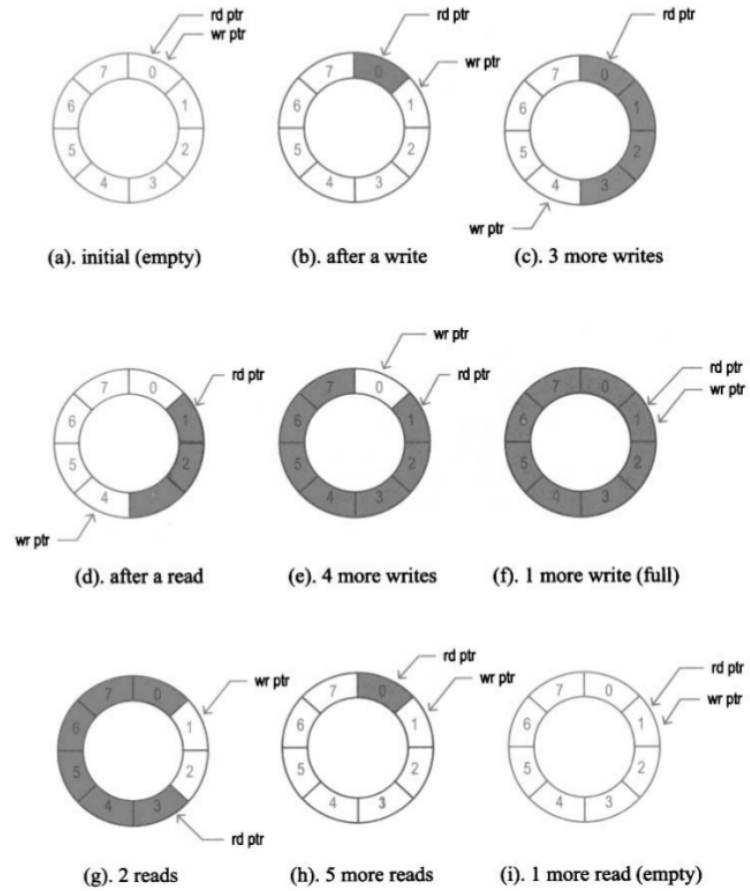
Interfaz fifo[3]

En un sistema grande, UART es usualmente un circuito periférico para la transmisión serial de datos. El sistema principal comprueba su estado periódicamente para obtener y procesar el dato recibido. La interfaz del receptor provee un mecanismo para señalar la disponibilidad de un nuevo dato y prevé que el dato recibido sea obtenido múltiples veces.
Además provee un buffer entre el receptor y el sistema principal, del cual hay varios esquemas usados comúnmente. El implementado en este práctico es el buffer FIFO
El búfer FIFO ofrece más espacio de almacenamiento en búfer y además reduce el riesgo de desbordamiento de datos. Podemos ajustar el número deseado de palabras en FIFO para dar cabida a la necesidad de procesamiento del sistema principal.
El diagrama de bloques detallado se muestra a continuación.



La señal *rx-done-tick* está conectada a la señal *wr* de la FIFO. Cuando se recibe una nueva palabra de datos, la señal *wr* se afirma en un ciclo de reloj y los datos correspondientes se escriben en la FIFO. El principal sistema obtiene los datos desde el puerto de lectura de la FIFO. Después de recuperar una palabra, se afirma la señal *rd* del ciclo de reloj FIFO para quitar el elemento correspondiente. La señal de vacío de la FIFO se puede utilizar para indicar si una palabra de datos recibida está disponible. Un error de datos-invadido ocurre cuando una nueva palabra de datos llega y la FIFO está llena.

El buffer FIFO es un almacenamiento variable entre dos subsistemas, que posee dos señales de control “wr” y “rd” para la escritura y lectura. Cuando wr está en 1 el dato de entrada es escrito dentro del buffer. La señal “rd” actúa como una señal de borrado, cuando está en 1 el primer item del buffer es removido y pone disponible el siguiente.
Una implementación optimizada puede ser bastante compleja, lo que se usa aquí es una simplificación basado en una cola circular.
En la imagen se muestra un ejemplo basado en una cola de 8 elementos de capacidad.



El puntero “wr” apunta al primer elemento de la cola y el “rd” al último. El buffer FIFO tiene dos registros de estado conocidos como “full” (donde no se puede escribir) y “empty” (donde no se puede escribir). Las condiciones ocurren cuando los punteros son iguales. La dificultad está en saber cuál de las condiciones es la que se está cumpliendo. Una alternativa es usar 2 banderas para seguir los estados de “full” y “empty”. Se inicializan en 1 y 0 y se van modificando en cada ciclo de reloj de acuerdo a los valores de “wr” y “rd”. A continuación se muestra el código verilog para este módulo.

```
1.  module fifo
2.  #(parameter N=8, // # DBIT: n° bits de dato
3.            W=2  // numero de bits de direcciones en la fifo
4.  )
5.  (clk, reset,rd, wr,w_data,empty, full,r_data);
6.
7.     // Entradas modulo buf FIFO
8.     input wire clk, reset;
9.    input wire rd; // rd: es la entrada de rd_uart del Rx y tx_done_tick de Tx
10.    input wire wr; // wr: es la salida rx_done_tick del Rx y wr_uart de Tx
11.   input wire [N-1:0] w_data; // entrada a transmitir
12.
13.    // Salidas modulo buf FIFO
14.    output wire empty; // bandera que indica buffer vacio, cuando no hay mas nada que
    leer en el buff
15.                       // empty: indica si el buff de recepcion esta vacia (1) o no (0)
16.    output wire full;  // full: bandera que indica cuando el buff esta lleno y no se puede
    escribir hasta que se vacie
17.                       // full: indica si el buff fifo de Tx esta lleno (1) o no (0)
18.   output wire [N-1:0] r_data; // registro de dato recepcion
```

```
19.
20.  // Declaracion de señales con inicializacion en cada una
21.   reg [N-1:0] array_reg [2**W-1:0];  // lista de registros
22.  reg [W-1:0] w_ptr_reg, w_ptr_next=0, w_ptr_succ;
23.  reg [W-1:0] r_ptr_reg, r_ptr_next=0, r_ptr_succ;
24.   reg empty_reg, empty_next = 1;// buff de recepcion Rx inicia vacio
25.   reg full_reg, full_next=0; // buff de transmision Tx inicia vacio
26.  wire wr_en;
27.
28.  // Estados de la FSM & Resgistros de datos
29.  // Operacion de escritura de fila de reg
30.  always @(posedge clk)
31.    if (wr_en)
32.      array_reg[w_ptr_reg] <= w_data;
33.
34.   // Operacion de lectura de fila de lista registro
35.  assign r_data = array_reg[r_ptr_reg];
36.
37.   // Se habilita la escritura solo cuando la fifo no esta llena
38.  assign wr_en = wr & ~full_reg;
39.
40.  // Logica de control de cola fifo
41.  // registros para punteros de lectura y escritura
42.  always @(posedge clk, posedge reset)
43.    if (reset)
44.      begin
45.        w_ptr_reg <= 0;
46.        r_ptr_reg <= 0;
47.        full_reg <= 1'b0;
48.        empty_reg <= 1'b1;
49.      end
50.    else
51.      begin
52.        w_ptr_reg <= w_ptr_next;
53.        r_ptr_reg <= r_ptr_next;
54.        full_reg <= full_next;
55.        empty_reg <= empty_next;
56.      end
57.
58.  // Logica de siguiente estado para la escritur y lectura
59.  always @*
60.  begin
61.    // Valores de punteros sucesivos
62.    w_ptr_succ = w_ptr_reg + 1;
63.    r_ptr_succ = r_ptr_reg + 1;
64.    // default: mantenimiento de valores anteriores
65.    w_ptr_next = w_ptr_reg;
66.    r_ptr_next = r_ptr_reg;
67.    full_next = full_reg;
68.    empty_next = empty_reg;
69.    case ({wr, rd})
70.      // 2'b00:  no op
71.      2'b01: // read
72.        if (~empty_reg) // not empty
73.          begin
74.            r_ptr_next = r_ptr_succ;
75.            full_next = 1'b0;
76.            if (r_ptr_succ==w_ptr_reg)
77.              empty_next = 1'b1;
78.          end
79.      2'b10: // write
80.        if (~full_reg) // not full
81.          begin
82.            w_ptr_next = w_ptr_succ;
83.            empty_next = 1'b0;
```

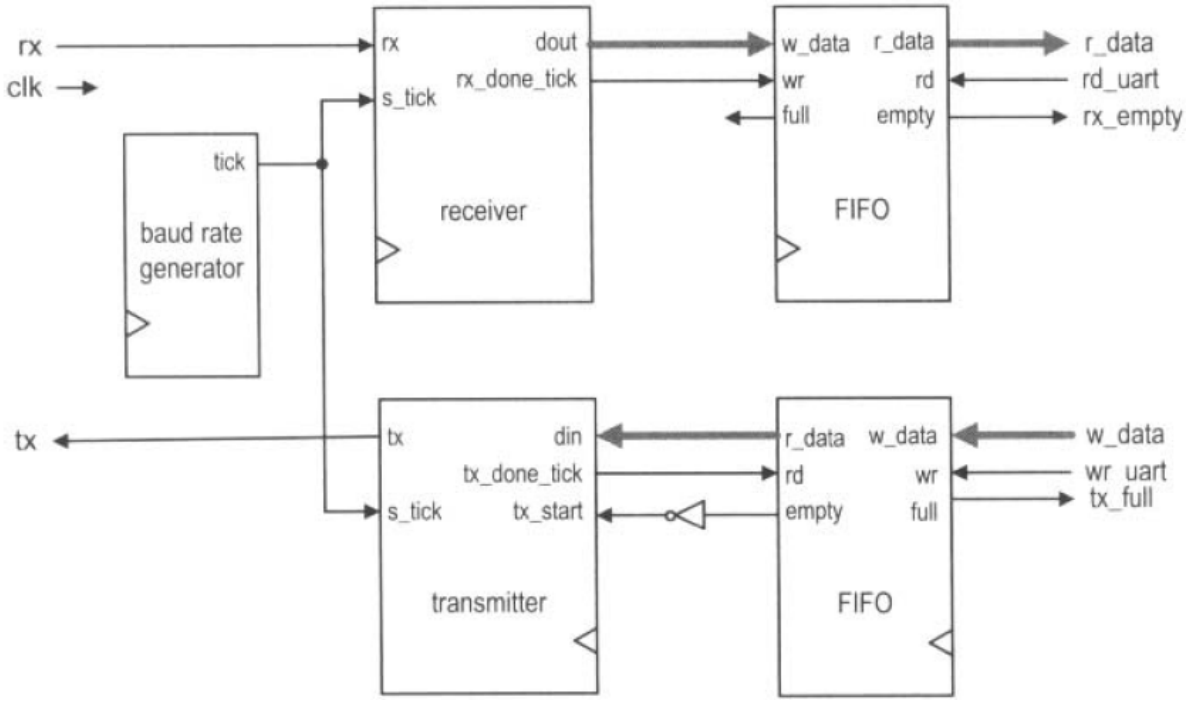
```
84.             if (w_ptr_succ==r_ptr_reg)
85.               full_next = 1'b1;
86.           end
87.       2'b11: // write and read
88.         begin
89.           w_ptr_next = w_ptr_succ;
90.           r_ptr_next = r_ptr_succ;
91.         end
92.     endcase
93.   end
94.
95.   // Logica de salida de moduLo buff FIFO
96.   assign full = full_reg;
97.   assign empty = empty_reg;
```

El código contiene una parte que puede ser interpretada como un controlador y dispone de dos punteros y dos banderas. Su lógica siguiente-estado examina los punteros “wr” y “rd” y toma acciones de acuerdo a esos valores.
La bandera de estado se asegura primero que el buffer no está lleno, de ser verdadera se incrementa el puntero “wr” y se limpia la bandera de estado.

Esta implementación se usa tanto en la recepción y la transmisión con la misma lógica y consideraciones.

Módulo uart completo[3]

El bloque completo se consigue combinando los módulos de transmisión y recepción antes detallados.

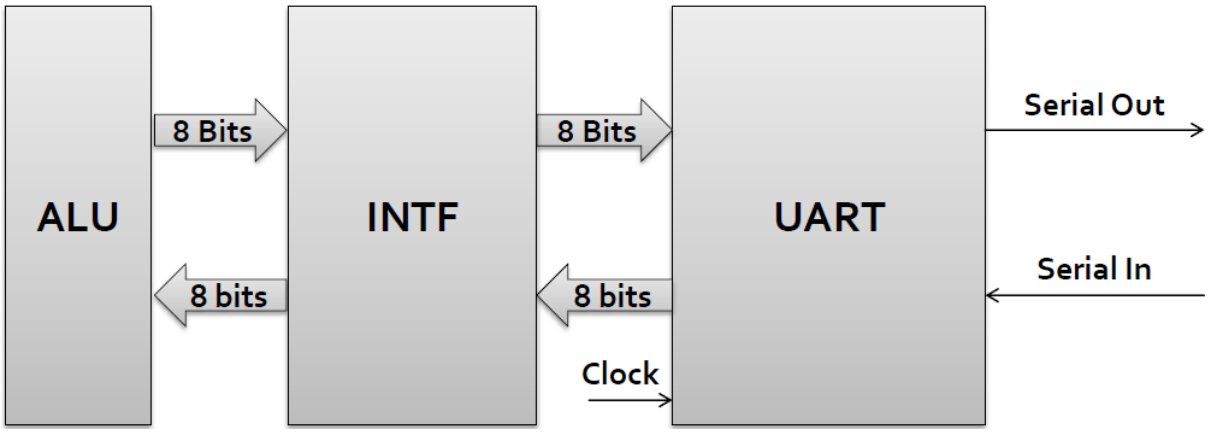


El código verilog del módulo es el siguiente.

```
module uart
#(//Default setting :
// 1 9 , 2 0 0 b a u d , 8 data b i t s
    parameter DBIT = 8,
              SB_TICK = 16, //
              FIFO_W = 2 // bits de direcciones en buff fifo
            //DVSR = 163,
            //DVSR_BIT = 8, //

)
(
    input wire clk, reset,
    input wire rd_uart , wr_uart , rx,
    input wire [7:0] w_data,
    output wire tx_full, rx_empty, tx,
    output wire [7:0] r_data
);

// signal declaration
    wire tick, rx_done_tick , tx_done_tick;
    wire tx_empty, tx_fifo_not_empty;
    wire [7:0] tx_fifo_out , rx_data_out;

//body
// Instacicacion de modulo Baud Rate Generator
    Baut_Rate_Generator ibrg (
        .clock(clk),
        .tick(tick)
    );


// Instancia de receptor rx
    Rx #(.N(DBIT), .M(SB_TICK))
    irx (
        .clk(clk),
        .reset(reset),
        .rx(rx),
        .s_tick_clk(tick),
        .rx_done_tick(rx_done_tick),
        .dout(rx_data_out)
    );

// Instancia de interfaz receptor (buff fifo)
    fifo #(.N(DBIT), .W(FIFO_W))
    iic_rx (
        .clk(clk),
        .reset(reset),
        .rd(rd_uart),
        .wr(rx_done_tick),
        .w_data(rx_data_out),
        .empty(rx_empty),
        .full(),
        .r_data(r_data)
    );

// Instancia de interfaz transmisor (buff fifo)
    fifo #(.N(DBIT), .W(FIFO_W))
    iic_tx (
        .clk(clk),
        .reset(reset),
        .rd(tx_done_tick),
        .wr(wr_uart),
```

```
63.         .w_data(w_data),
64.         .empty(tx_empty),
65.         .full(tx_full),
66.         .r_data(tx_fifo_out)
67.     );
68.
69. // Instancia de transmisor tx
70.     Tx #(.N(DBIT), .M(SB_TICK))
71.     itx (
72.         .clk(clk),
73.         .reset(reset),
74.         .tx_start(tx_fifo_not_empty),
75.         .s_tick(tick),
76.         .din(tx_fifo_out),
77.         .tx_done_tick(tx_done_tick),
78.         .tx(tx)
79.     );
80.
81.     assign tx_fifo_not_empty = ~tx_empty;
82.
83. endmodule
```

Módulo INTF - Interfaz uart-alu



Como vemos en el diagrama del bloque INTF, este es un bloque que contiene dos entradas de 8 bits y dos salidas de 8 bits. Un par de entrada y salida es por parte del módulo UART, que se corresponde a los datos recibidos por el receptor rx (Serial In) que son convertidos de modo serial a modo paralelo en un buf de 8 bits y dicha salida se la da como entrada a la interfaz para que se encargue de adaptar el dato en código ASCII a dato binario que luego de dicha adaptación sale en forma binaria para ser almacenada en la ALU. El otro par de entrada salida a la interfaz es por parte del módulo ALU que es el resultado de la misma para ser readaptado como dato en código ASCII y de esa manera poder ser transmitido por el módulo UART y así poder ser observado de manera correcta en la computadora PC.
A continuación podemos ver el código del módulo.

```
1.  module intf_v2
2.  #(parameter N = 8) // N: numero de bits de datos
3.  (clk, reset, dato_A, dato_B, dato_Op, empty_uart, uart_in, alu_in,
4.   uart_out, rd_uart, wr_uart);
5.     // Entradas intf
6.     //input wire [3:0] buff_in; // buffer de entrada vinculado a la FPGA y asociado a cada
    estado de carga de valores en manejador_ALU
7.     input wire empty_uart; // bandera empty de uart como entrada
8.     input wire [N-1:0] uart_in;
9.     input wire [N-1:0] alu_in;
10.    input wire clk, reset; // clk: clock de FPGA - reset: señal para resetear FSM
11.    //input wire s_tick_clk; // s_tick_clk: clock de baud rate gen.
12.
13.    // Salidas intf
14.    output reg [N-1:0] dato_A, dato_B;
15.    output reg [5:0] dato_Op;
16.    output wire [N-1:0] uart_out;
17.    //output reg [N-1:0] alu_out;
18.    //output reg [2:0] pulsador; // pulsador que controlara p_abc de manejador_ALU segun
    estado de intf
19.    output reg rd_uart=0, wr_uart=0;
20.
21.    // Declaracion de estados
22.   localparam [4:0]
23.      idle = 5'b00001, // estado de espera para cambio de estado
24.        operando_a = 5'b00010,// estado para almacenar recepcion de UART en dato_A de
    manejador_ALU
25.      operando_b = 5'b00100,// estado para almacenar recepcion de UART en dato_B de
    manejador_ALU
26.      operacion  = 5'b01000,// estado para almacenar recepcion de UART en operacion de
```

```
    manejador_ALU
27.     resultado  = 5'b10000;// estado para avisar resultado a transmitir por UART
28.
29.   // Declaracion de señales con inicializacion en cada una
30.   reg [4:0] state, state_next=idle;// estados de 4 bits
31.    reg [3:0] s_reg, s_next=0; //s_reg: Tick counter -> cuenta los ticks desde el inicio,
    maxima cuenta 15(4 bits)
32.    //reg [2:0] p_next = 0; // siguiente pulsador
33.    reg [N-1:0] dato_A_next=0, dato_B_next=0;
34.    reg [5:0] dato_Op_next = 6'b10000;
35.    reg enable, enable_next = 1'b1; // bit de habilitacion para cargar dato
36.    reg [1:0] turn, turn_next = 0;  // turno de carga de estados
37.                                    // si turn = 00 -> se carga A
38.                                    // si turn = 01 -> se carga B
39.                                    // si turn = 10 -> se carga Op
40.    reg [N-1:0] reg_uart_out = 52;
41.    reg [N-1:0] entrada_op = 1;
42.
43.    reg detector, detector_next = 0;
44.    reg [N-1:0] aux;
45.    reg [7:0] i, j;
46.    reg flag = 0;
47.
48.   // Estados de la FSM
49.   always @(posedge clk, posedge reset)
50.     if (reset)
51.       begin
52.         state <= idle;
53.            s_reg <= 1'b0;
54.            turn  <= 2'b00;
55.            dato_A <= 0;
56.            dato_B <= 0;
57.            dato_Op <= 6'b10000;
58.            enable <= 1'b1;
59.            detector <= 1'b0;
60.            //wr_uart <= 0;
61.            //rd_uart <= 0;
62.       end
63.     else
64.       begin
65.         state <= state_next;
66.            s_reg <= s_next;
67.            turn <= turn_next;
68.            dato_A <= dato_A_next;
69.            dato_B <= dato_B_next;
70.            dato_Op <= dato_Op_next;
71.            enable <= enable_next;
72.            detector <= detector_next;
73.       end
74.
75.   // Logica de siguiente estado
76.   always @*
77.   begin
78.      // default: por defecto se mantienen los valores anteriores
79.     state_next = state;
80.     s_next = s_reg;
81.       dato_A_next = dato_A;
82.       dato_B_next = dato_B;
83.       dato_Op_next = dato_Op;
84.       enable_next = enable;
85.       turn_next = turn;
86.       rd_uart = 1'b0; // recepcion no leida
87.       wr_uart = 1'b0; // no realizar transmision
88.       detector_next= detector;
89.     case (state)
```

```
90.         idle:
91.             begin
92.             s_next = 0; // tick counter inicia en cero
93.           if(enable && ~empty_uart)
94.                 begin
95.                   if(turn[0]==0 && turn[1]==0)
96.                     state_next = operando_a; // estado sig -> carga de operando_a
97.
98.                   else if(turn[0] && turn[1]==0)
99.                     state_next = operando_b; // estado sig -> carga de operando_b
100.
101.                     else if(turn[1] && ~turn[0])
102.                       state_next = operacion; // estado sig -> carga de operacion
103.
104.
105.                  end
106.
107.                  else begin
108.                    if(~empty_uart)
109.                      enable_next = 1'b1; // se habilita bit para cargar nuevo dato
110.                      end
111.               end
112.
113.            operando_a:
114.            begin
115.              if(enable)
116.              begin
117.                if(uart_in == 13) // si entrada es un enter
118.                begin
119.                  detector_next = 1'b0;
120.                  turn_next = 2'b01; // se cambia turno para carga de B en proxima
    subida de empty_uart
121.                end
122.                else begin
123.                  if(detector) begin
124.                    dato_A_next = (10* dato_A) + (uart_in -48);
125.                    detector_next = 1'b0;
126.                  end
127.
128.                  else begin
129.                    dato_A_next = uart_in - 48;// -48
130.                    detector_next = 1'b1;
131.                  end
132.                end
133.                rd_uart = 1'b1; // aviso a uart de recepcion leida
134.                enable_next = 1'b0; // se deshabilita carga hasta que se vacie
    nuevamente fifo
135.                state_next = idle;// estado sig -> se vuelve a idle
136.            end
137.            end
138.
139.
140.            operando_b:
141.            begin
142.              if(enable)
143.              begin
144.                if(uart_in == 13) // si entrada es un enter
145.                begin
146.                  detector_next = 1'b0;
147.                  turn_next = 2'b10; // se cambia turno para carga de Op en proxima
    subida de empty_uart
148.                end
149.                else begin
150.                  if(detector) begin
151.                    dato_B_next = (10* dato_B) + (uart_in -48);
```

```
152.                        detector_next = 1'b0;
153.                    end
154.
155.                    else begin
156.                        dato_B_next = uart_in - 48;// -48
157.                        detector_next = 1'b1;
158.                    end
159.                end
160.                rd_uart = 1'b1; // aviso a uart de recepcion leida
161.                enable_next = 1'b0; // se deshabilita carga hasta que se vacie
    nuevamente fifo
162.                state_next = idle;// estado sig -> se vuelve a idle
163.            end
164.            end
165.
166.            operacion:
167.            begin
168.                if(enable)
169.                begin
170.                    //dato_Op_next = uart_in[5:0] - 48; // -cte para obtener Op
171.                    entrada_op = uart_in - 48; // -48
172.                    case (entrada_op)
173.                        1: dato_Op_next = 6'b100000; // op suma
174.                        2: dato_Op_next = 6'b100010; // op resta
175.                        3: dato_Op_next = 6'b100100; // op AND
176.                        4: dato_Op_next = 6'b100101; // op OR
177.                        5: dato_Op_next = 6'b100110; // op XOR
178.                        6: dato_Op_next = 6'b000011; // op SRA
179.                        7: dato_Op_next = 6'b000010; // op SRL
180.                        8: dato_Op_next = 6'b100111; // op NOR
181.                        default: dato_Op_next = 6'b100000; // op suma
182.                    endcase
183.
184.                detector_next = 1'b0; // se reinicia detector de decenas a unidades (0)
185.                turn_next = 2'b11; // se cambia turno para carga de A en proxima subida de
    empty_uart
186.                rd_uart = 1'b1; // aviso a uart de recepcion leida
187.                enable_next = 1'b0; // se deshabilita carga hasta que se vacie nuevamente
    fifo
188.                i = 0;
189.                j = 0;
190.                aux = alu_in;
191.                state_next = resultado;// estado sig -> resultado para enviar resultado por
    uart
192.                end
193.            end
194.
195.            resultado:
196.            begin
197.                reg_uart_out = alu_in + 48;
198.                wr_uart = 1'b1; // realizar transmision tx uart
199.                turn_next = 2'b00; // se cambia turno para carga de A en proxima subida
    de empty_uart
200.                state_next = idle;// estado sig -> se vuelve a idle
201.            end
202.
203.        endcase
204.    end
205.
206.     // Logica de Salida
207.    //assign uart_out = alu_in + 48; // Se suma 48 ya uart transmite en ASCII
208.     assign uart_out = reg_uart_out; // Se suma 48 ya uart transmite en ASCII
209.    //assign alu_out = uart_in - 48; // Se resta 48 ya uart recibe todo es en ASCII
210.
211.
```

```
212.  endmodule
```

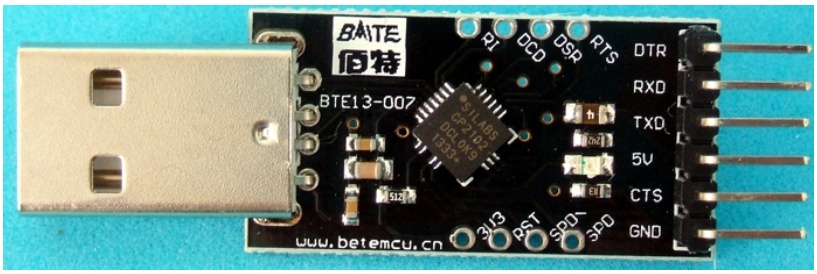
Como podemos ver en el código el módulo las entradas y salidas descritas anteriormente fueron definidas como *uart_in, alu_in, uart_out, alu_out*. Pero además el módulo tiene un par más de salidas y una entrada extra.
Por el lado de la entrada extra se debe a la bandera *empty_uart* que nos avisa si el buffer se encuentra de recepción del módulo UART se encuentra vacío, para detectar cuando el mismo se llene.
Por el lado de las salidas se tiene *rd_uart* y *wr_uart*, que se son las señales de control del módulo UART que debe manejar la interfaz cuando ya leyó un dato o cuando tiene que transmitir un dato nuevo para que sea transmitido por el transmisor tx a la PC.

Por último podemos decir que el módulo interfaz es una máquina de estado que consiste de un estado inicial IDLE de espera de una nueva recepción del módulo UART, es decir que cuando ai una nueva recepción el módulo UART lo indica con su bandera *empty_uart* de manera que la interfaz analizando dicha bandera sabe que ai recepción por lo que cambia de estado para cargar el operando A de la ALU (estado *operando_a*) hasta que se presione *ENTER* (permitiendo el ingreso de 2 cifras, y si se ingresan más se vuelve a reemplazar por una nueva de dos cifras) y descarta la recepción, ya que la utilizo para lo deseado. Luego del *ENTER* desde el estado de carga de operando A, se pasa al estado de carga de operando B (estado *operando_b*), que procede de la misma forma que el anterior y por último presionado *ENTER* se pasa al estado de carga de código de operación (estado *operacion*) que recibiendo un número del 1 al 6 realizara cualquiera de las operaciones de la ALU según se indique y una vez seleccionada la operación instantáneamente se transmite el resultado (estado *resultado*) sin tener que presionar *ENTER*. Volviendo nuevamente al estado IDLE para realizar nuevamente la carga de los operandos y nueva operación en caso de desearlo.

Validación y verificación de funcionalidad

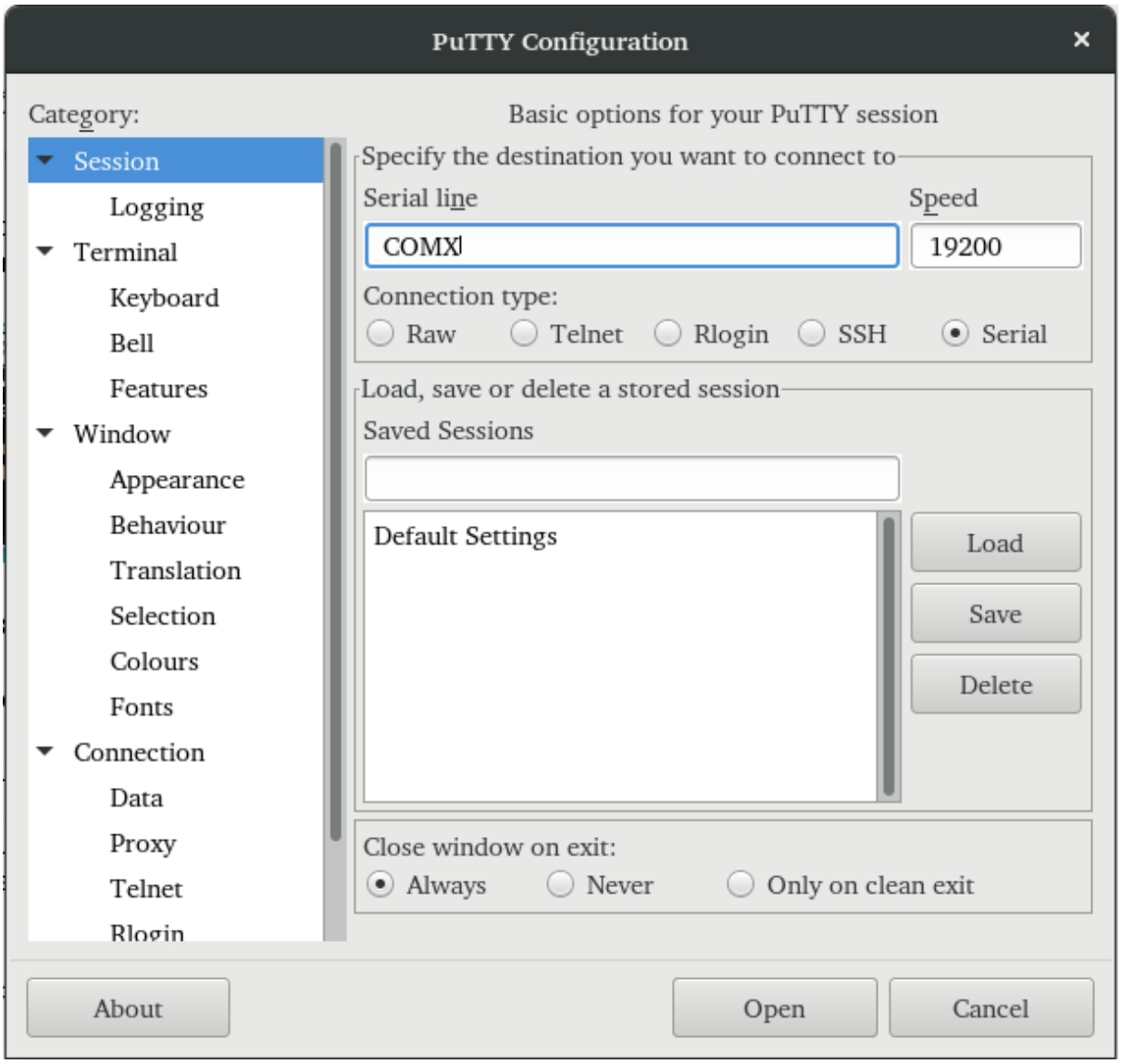
- Explicar que se uso la Basys II y el adaptador uart-usb CP2102 para verificar la funcionalidad
- Explicar como se realizo la experiencia entre la fpga y la PC con el uso de putty (poner la conf del putty como indica en el libro)
- Problemas que quedaron pendientes pero con ideas de como resolverlos.

La implementación de los módulos explicados anteriormente se realizó en una placa de desarrollo Spartan 3 - Basys II, la cual no posee un módulo-interfaz USB para UART incorporado como por ejemplo la Spartan 6 - Nexys III y nos impedía la comunicación de la Basys II con la PC. Por esa razón se hizo uso del adaptador USB-UART CP2102



El cual posee un puerto USB y pines de salida característicos para una transmisión serial (TXD y RXD), proporcionando una interfaz adecuada.
La tensión de alimentación para el adaptador es directamente suministrada por el puerto USB de la PC.
En la Basys II mediante un archivo ucf (user constraints file) se debieron configurar dos pines correspondientes a algún puerto de propósito general de los 4 que posee la placa como TX y RX (que en nuestro caso usamos J3 y B5) y finalmente se realizó una conexión cruzada (TXD-RX y RXD-TX) con los pines del adaptador y los configurados en la placa y una puesta en común de masas (GND).

Se escogió como cliente de comunicación serial al software Putty, el cual debe configurarse para que pueda establecer comunicación con la placa y sea compatible a nuestro módulo UART de la siguiente forma.



Como puede observarse en la imagen se debe elegir como tipo de conexión a la serial y escoger la velocidad de acuerdo al Baud Rate, la cual, para nuestro caso es 19200. En la línea serial debe elegirse el puerto correcto. Se deberá configurar 8 bits de datos, 1 bit de stop, sin bit de paridad, y sin control de flujo. El administrador de dispositivos (Windows) o dmesg (Linux) proporcionan esa información.

Algo a tener en cuenta a la hora de realizar la comunicación o mejor dicho de mostrar y operar los valores obtenidos y enviados entre la PC y la placa es que el software Putty envía y recibe los datos en su equivalente código ASCII, es así que de acuerdo a una tabla ASCII podemos observar que para representar algún valor numérico debe sumarse 48 para obtener su equivalente en ASCII.

Binario	Decimal	Hexadecimal	Representación
0011 0000	48	30	0

0011 0001	49	31	1
0011 0010	50	32	2
0011 0011	51	33	3
0011 0100	52	34	4
0011 0101	53	35	5
0011 0110	54	36	6
0011 0111	55	37	7
0011 1000	56	38	8
0011 1001	57	39	9

Teniendo esto presente se hacía necesario sumar 48 en cada valor que se deseaba transmitir desde la placa hacia la PC y restarle 48 a cada valor que la placa recibía desde la PC para poder ser operado.

Un inconveniente relacionado a lo anterior se presentó cuando se obtenía un resultado mayor a 9, es decir que tenía más de una cifra. En la transmisión ese problema fue solucionado al incorporar la tecla “Enter” como un validador de datos, de tal forma que no se aceptaba un valor hasta encontrar esa tecla y nos permitía ingresar valores de más de dos cifras como operandos. Sin embargo cuando se presentaba un resultado de más de una cifra no se pudo lograr que se transmitieran los valores delante de la unidad primero (centena y/o decena) y luego la unidad para formar el valor del resultado.
Se decidió entonces dejar representado el resultado en un solo carácter, que podía luego ser entendido encontrando su equivalente en código ASCII en una tabla.

Bibliografía

1. "Universal Asynchronous Receiver-Transmitter." Wikipedia. Wikimedia Foundation. Web. 10 Oct. 2016.
2. "Máquina De Estados Algorítmica." Wikipedia. Wikimedia Foundation. Web. 08 Oct. 2016.
3. Chu, Pong P. FPGA Prototyping by Verilog Examples: Xilinx Spartan -3 Version. Hoboken, NJ: J. Wiley & Sons, 2008. Print.

Works Cited

Chu, Pong P. FPGA Prototyping by Verilog Examples: Xilinx Spartan -3 Version. Hoboken, NJ: J. Wiley & Sons, 2008. Print.

"Máquina De Estados Algorítmica." *Wikipedia*. Wikimedia Foundation. Web. 08 Oct. 2016.

"Universal Asynchronous Receiver-Transmitter." *Wikipedia*. Wikimedia Foundation. Web. 10 Oct. 2016.

“ASCII.” *Wikipedia*. Wikimedia Foundation. Web. 10 Oct. 2016.