Trabajo Practico 2: Universal Asynchronous Receiver Transmitter

Perez, Federico
perezfederico@unc.edu.ar

Sardoy, Juan Manuel
jmsardoy@gmail.com

10 de octubre de 2018

Universidad Nacional
de Córdoba

1. Descripción del trabajo

El siguiente trabajo consiste en la implementación práctica de un módulo completamente funcional del protocolo UART o Üniversal Asynchronous Receiver-Transmitter, en un dispositivo FPGA, en el lenguaje de descripción de hardware Verilog.
Como su nombre lo especifica, se trata de un protocolo asíncrono y *full-duplex* pero de facil uso e implementación dado su simplicidad.
La arquitectura completa del trabajo de aplicación será aproximadamente en siguiente:

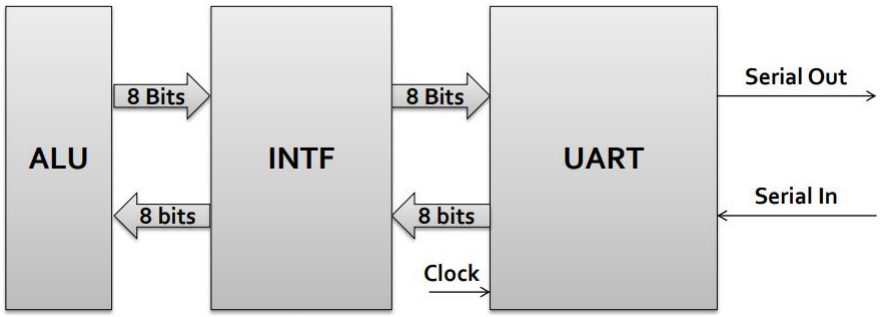

Figura 1: *Diagrama de la arquitectura básica del proyecto*

A fines prácticos y demostrativos, dicho módulo UART se conectará mediante un módulo que actuará de interfaz, a una *Unidad Aritmetico Lógica* o ALU. Dicho módulo interfaz contendrá lógica que permitirá procesar instrucciones y argumentos recibidos por el módulo RX del UART, enviarlos a la ALU para su resolución, y reenviarlos por el módulo TX nuevamente hacia el solicitante del cálculo.

Como usuario de dicho sistema, conectaremos el puerto USB de una computadora, y con un convertidor *USB-UART*, se enviarán las instrucciones y argumentos requeridos. Para dicho fin, además se desarrollará algún tipo de software que haga uso del hardware convertidor, y facilite el envío de instrucciones y la recepción de los resultados calculados en la ALU.

2. Protocolo UART

Este protocolo, como se ha explicado anteriormente, es bastante simple. El siguiente diagrama muestra los datagramas básicos de una comunicación UART.

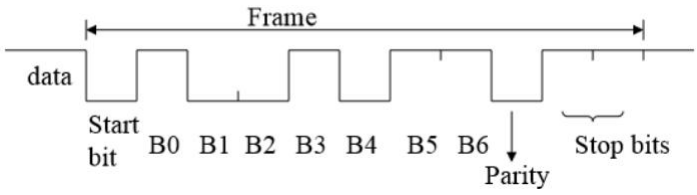

Figura 2: *Diagrama del datagrama o frame del protocolo UART*

El pin RX del receptor, cuando se encuentra en estado de espera o *stand-by*, debe estar en alto o *HIGH*.
Cuando se esté por recibir un dato, el nivel baja a *LOW*. Los siguientes siete u ocho bits son de datos, dependiendo si se utiliza un bit de paridad o no, respectivamente. Luego para finalizar, deben venir dos bits en alto, que indican el final del dato y se pasa a modo *stand-by* nuevamente.

Los dos parámetros mas importantes en una comunicación UART son el *baudrate*, el cual es la frecuencia de símbolos (en este caso *bit*) a la que opera el receptor y transmisor. El otro parametro es si se está usando paridad o no, lo cual afecta la confiabilidad y el tamaño del dato transmitido.

3. Implementación

Un transmisor UART es modelado muy facilmente mediante el uso de maquinas de estados finitas (*FSMs o Finite-state machines)*, por esto mismo, se notará que este comportamiento o lógica es replicada en en hardware desarrollado, para ambos modulos RX y TX.

3.1. Generador de Baud Rate

Dado que un módulo UART puede operar a diferentes *baud rates*, éste debe poseer algun tipo de clock variable que le sirva como entrada. Básicamente, se trata de un divisor de frecuencia o contador del clock principal.

La frecuencia del pulso generado por el generador de baud rate debe ser mayor al baud rate mismo. Esto es para que el usuario de dicho generador tenga una precición más alta a la hora de leer los bits recibidos. Según estudios ese número es 16 veces, lo que da una ecuación para el contador de la siguiente forma:

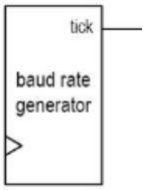

Figura 3: *Diagrama del baud-rate generator*

$$count = \frac{clock}{baudrate*16}$$

Lo que para un clock de 50Mhz da:

$$count = 163$$

Su implementación es la siguiente (ignorando algunas constantes):

```
module BaudRateGenerator
    #(parameter FREQUENCY = `FREQUENCY,
      parameter BAUD_RATE = `BAUD_RATE)
                (input clk,
                 input rst,
                 output reg out);
_
    localparam MAX_COUNT = (FREQUENCY / (BAUD_RATE * 16));
    localparam COUNT_NBITS = $clog2(MAX_COUNT);

    reg [COUNT_NBITS : 0] count;
    always@(posedge clk or posedge rst)
    begin
        if (!rst) begin
            out <= 0;
            count <= 0;
        end
        else begin
            if(count == MAX_COUNT)
            begin
                out <= 1;
                count <= 0;
            end
            else begin
                out <= 0;
                count <= count + 1;
            end
        end
    end
endmodule
```

Es simplemente un módulo con dos entradas, el *clock* o *clk* y un pin de *reset* o *rst*, y un pin de salida o *out*. Hay un registro *count* que actúa de contador. Dicho contador aumenta con cada ciclo de *clk*. Cuando el contador llega al valor de *MAX_COUNT* se invierte el valor de *out*, el cual comienza en 0 luego de un reset.

Vemos como a partir de la frecuencia de operacion, y el *baudrate* elegido, se calculan el parámetro *MAX_COUNT*, en tiempo de síntesis.

3.2. Módulo RX

El módulo RX o receptor es el encargado de detectar el comienzo de la transmisión por el pin de recepción, de interpretar los datos, y exponerlo por el bus de salida. Típicamente el bus de salida, y el largo de los datos recibidos es de un *byte* o 8 *bits*. Como se podrá notar también a continuación en la implementación del módulo TX, ambos son muy facilmente modelados por máquinas de estado finitas o FSMs dado la naturaleza secuencial de su lógica.

El diagrama de la máquina de estados para el módulo RX es el siguiente:

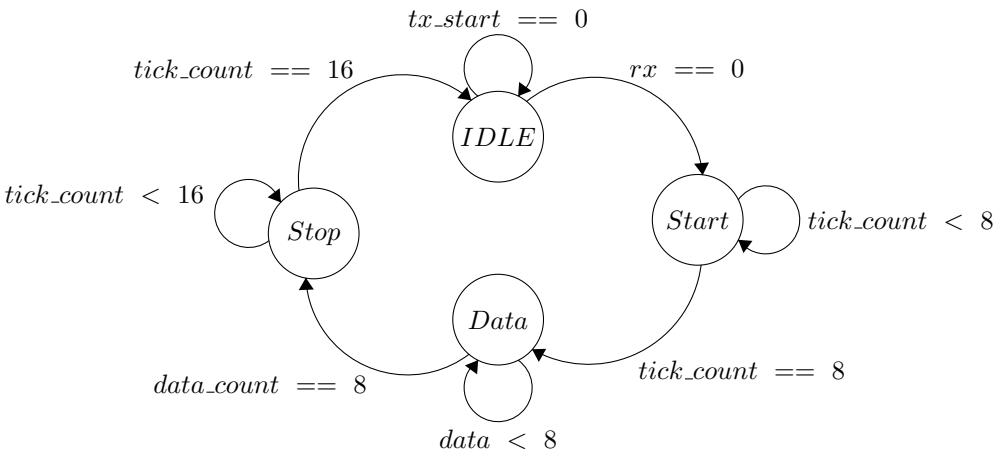

Figura 4: *Diagrama de estados del módulo RX*

La implementación en código es la siguiente:

```
module RX(
    input clk,
    input rst,
    input i_baud_rate,
    input i_rx,
    output reg o_rx_done,
    output reg [`NBITS-1 : 0] o_data
);
```

```
localparam NBITS = `NBITS;
localparam COUNTER_NBITS = $clog2(NBITS)+1;

localparam
    idle  = 'b00,
    start = 'b01,
    data  = 'b11,
    stop  = 'b10;

reg [1:0] state, next_state;
reg [3:0] tick_count, next_tick_count;
reg [COUNTER_NBITS-1:0] data_count, next_data_count;
reg [NBITS-1 : 0] next_data;
reg next_rx_done;

always@(posedge clk or posedge rst) begin
    if (!rst) begin
        state <= idle;
        tick_count <= 0;
        data_count <= 0;
        o_data <= {NBITS{1'b0}};
        o_rx_done <= 0;
    end
    else begin
        state <= next_state;
        tick_count <= next_tick_count;
        data_count <= next_data_count;
        o_data <= next_data;
        o_rx_done <= next_rx_done;
    end
end

always@* begin

    //defaults
    next_state = state;
    next_tick_count = tick_count;
    next_data_count = data_count;
    next_data = o_data;
    next_rx_done = 0;

    case (state)
        idle:
        begin
            if (!i_rx) begin
                next_state = start;
```

```
                    next_tick_count = 0;
                    next_data_count = {COUNTER_NBITS{1'b0}};
                    next_data = {NBITS{1'b0}};
                end
            end
            start:
            begin
                if (i_baud_rate) begin
                    if(tick_count == 7) begin
                        next_state = data;
                        next_tick_count = 0;
                    end
                    else next_tick_count = tick_count + 1;
                end
            end
            data:
            begin
                if (i_baud_rate) begin
                    if (data_count == NBITS) next_state = stop;
                    else begin
                        if(tick_count == 15) begin
                            next_data = {i_rx , o_data[NBITS-1:1]};
                            next_data_count = data_count + 1;
                            next_tick_count = 0;
                        end
                        else next_tick_count = tick_count + 1;
                    end
                end
            end
            stop:
            begin
                if (i_baud_rate) begin
                    if (tick_count == 15) begin
                            next_state = idle;
                            next_rx_done = 1;
                        end
                    else next_tick_count = tick_count + 1;
                end
            end
        endcase
    end
endmodule
```

La entradas del módulo son el *clock*, el pin de *reset*, el input del baud rate generator, y el propio pin de comunicaciones o pin *rx*. A la salida un pin *rx_done*, que se ocupa de avisar al módulo usuario cuando se recibió un dato, y el bus del

dato propiamente recibido. En cuanto a la implementación, vemos que es la típica de una maquina de estados en Verilog, basada en el diagrama de estados presentado anteriormente.

3.3. Módulo TX

El módulo TX del UART es muy similar al RX. Tambien se modela mediante una máquina de estados, la cual es de extrema similitud con la anteriormente explicada.

Su diagrama de estados es el siguiente:

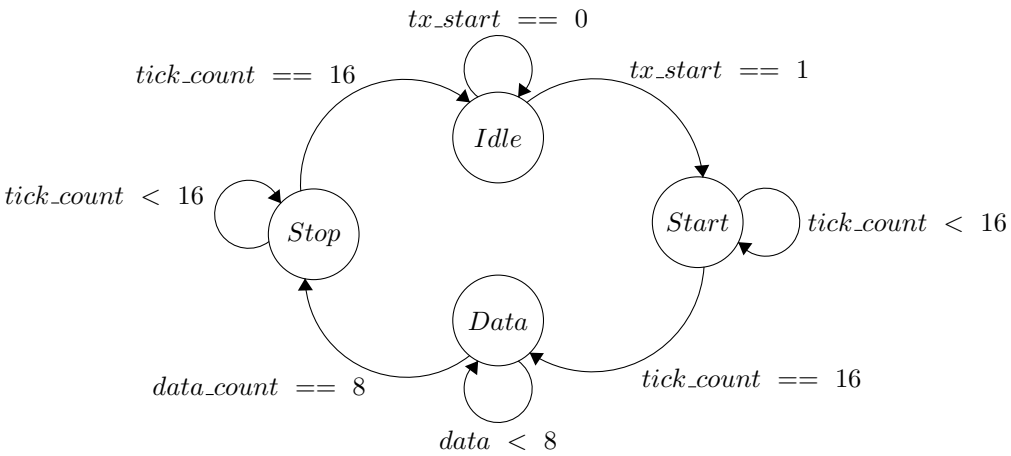

Figura 5: *Diagrama de estados del módulo TX*

Su implementación en Verilog es la siguiente:

```
module TX(
    input clk,
    input rst,
    input i_baud_rate,
    input i_tx_start,
    input [`NBITS-1 : 0] i_data,
    output reg o_tx_done,
    output reg o_tx);

    localparam NBITS = `NBITS;
    localparam COUNTER_NBITS = $clog2(NBITS)+1;

    localparam
        idle  = 'b00,
        start = 'b01,
        data  = 'b11,
```

```
    stop  = 'b10;

reg [1:0] state, next_state;
reg [3:0] tick_count, next_tick_count;
reg [COUNTER_NBITS-1 : 0] data_count, next_data_count;
reg [NBITS - 1 : 0] data_reg, next_data_reg;
reg next_tx, next_tx_done;

always@(posedge clk or posedge rst) begin
    if (!rst) begin
        state <= idle;
        tick_count <= 0;
        data_count <= 0;
        data_reg <= 0;
        o_tx_done <= 1;
        o_tx <= 1;
    end
    else begin
        state <= next_state;
        tick_count <= next_tick_count;
        data_count <= next_data_count;
        data_reg <= next_data_reg;
        o_tx_done <= next_tx_done;
        o_tx <= next_tx;
    end
end

always@* begin

    next_tx = o_tx;
    next_tx_done = o_tx_done;
    next_state = state;
    next_tick_count = tick_count;
    next_data_count = data_count;
    next_data_reg = data_reg;

    case (state)
        idle:
        begin
            next_tx = 1;
            next_tx_done = 1;
            if(i_tx_start) begin
                next_state = start;
                next_tick_count = 0;
                next_data_count = 0;
                next_data_reg = i_data;
```

```
                end
            end
            start:
            begin
                next_tx = 0;
                next_tx_done = 0;
                if (i_baud_rate) begin
                    if(tick_count == 15) begin
                        next_tick_count = 0;
                        next_state = data;
                    end
                    else next_tick_count = tick_count + 1;
                end
            end
            data:
            begin
                if (i_baud_rate) begin
                    if (data_count == NBITS) next_state = stop;
                    else begin
                        if(tick_count == 0) begin
                            next_tx = data_reg[0];
                            next_data_reg = data_reg >> 1;
                        end
                        if (tick_count == 15) begin
                            next_tick_count = 0;
                            next_data_count = data_count + 1;
                        end
                        else next_tick_count = tick_count + 1;
                    end
                end
            end
            stop:
            begin
                next_tx = 1;
                if (i_baud_rate) begin
                    if(tick_count == 15) begin
                        next_state = idle;
                    end
                    else next_tick_count = tick_count + 1;
                end
            end
        endcase
    end
endmodule
```

En este caso, las entradas al módulo son similares al módulo RX pero en

sentido inverso. *clock*, *baudrate* y *reset* se mantienen, pero ahora el bus de datos es la entrada llamado *i_data*, un pin que indica el momento en el que se debe empezar a enviar datos, llamado *i_tx_start*, y de salida, *o_tx_done* que indica la finalización del envio, y *o_tx* que es el pin de la propia comunicación.

3.4. Módulo Interface

El módulo interface tiene la utilidad de conectar y controlar el módulo UART con una ALU diseñada en el trabajo práctico anterior. Lo que hará, sera simplemente esperar dos datos por RX, A y B, y luego un código de operación, realizar la operación solicitada y enviar el resultado por TX. El usuario de dicho sístema será una computadora, corriendo un script de python, que enviaŕa las operaciones solicitadas y esperará el resultado.

El diagrama de estados del modulo interface es el siguiente:

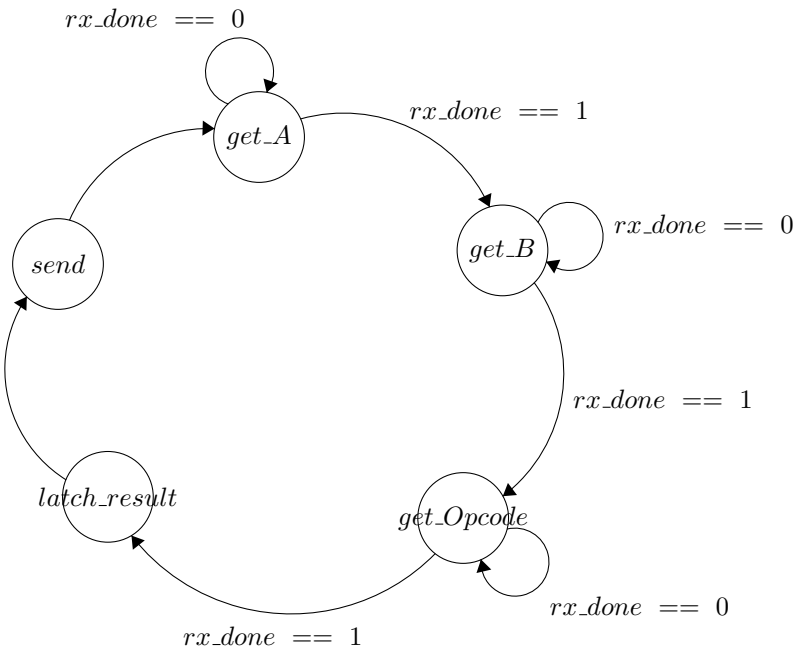

Figura 6: *Diagrama de estados del módulo interface*

El código Verilog de dicho módulo es:

```
module Interface(
    input clk,
    input rst,
    input i_tx_done,
    input i_rx_done,
```

```
    input [7:0] i_rx,
    input [7:0] i_alu_result,
    output reg o_tx_start,
    output reg [7:0] o_tx,
    output reg [7:0] o_alu_a,
    output reg [7:0] o_alu_b,
    output reg [5:0] o_alu_opcode,
    output reg [3:0] o_led
);

    localparam
        get_a        = 'b000,
        get_b        = 'b001,
        get_opcode   = 'b010,
        latch_result = 'b011,
        send_result  = 'b100;

    reg [2:0] state, next_state;
    reg next_tx_start;
    reg [7:0] next_tx;
    reg [7:0] next_alu_a;
    reg [7:0] next_alu_b;
    reg [5:0] next_alu_opcode;

    always@(posedge clk or negedge rst) begin
        if(!rst) begin
            state <= get_a;
            o_tx_start <= 0;
            o_tx <= 0;
            o_alu_a <= 0;
            o_alu_b <= 0;
            o_alu_opcode <= 0;
            o_led <= 0;
        end
        else begin
            o_led <= state;
            state <= next_state;
            o_tx_start <= next_tx_start;
            o_tx <= next_tx;
            o_alu_a <= next_alu_a;
            o_alu_b <= next_alu_b;
            o_alu_opcode <= next_alu_opcode;
        end
    end

    always@* begin
```

```
        //defaults
        next_state = state;
        next_tx_start = 0;
        next_tx = o_tx;
        next_alu_a = o_alu_a;
        next_alu_b = o_alu_b;
        next_alu_opcode = o_alu_opcode;

        case (state)
            get_a:
            begin
                if(i_rx_done) begin
                    next_alu_a = i_rx;
                    next_state = get_b;
                end
            end
            get_b:
            begin
                if(i_rx_done) begin
                    next_alu_b = i_rx;
                    next_state = get_opcode;
                end
            end
            get_opcode:
            begin
                if(i_rx_done) begin
                    next_alu_opcode = i_rx[5:0];
                    next_state = latch_result;
                end
            end
            latch_result:
            begin
                next_tx = i_alu_result;
                next_state = send_result;
            end
            send_result:
            begin
                next_tx_start = 1;
                next_state = get_a;
            end
        endcase
    end
endmodule
```