Universidad de Guadalajara
Departamento de electrónica

Actividad 5

Receptor UART

Verificacion de circuitos digitales

Eduardo Vazquez Diaz
lalohao@gmail.com

Contenido

Resumen

1. Objetivo

Implementar un receptor UART estándar RS-232 en lenguaje de descripción de hardware (*verilog*) capaz de trabajar a 9600bps y verificarlo.

2. Introducción

La comunicacion serial requiere dos señales **TxD** y **RxD**. Usualmente se utiliza un conector de 9 pines (Figura 1).

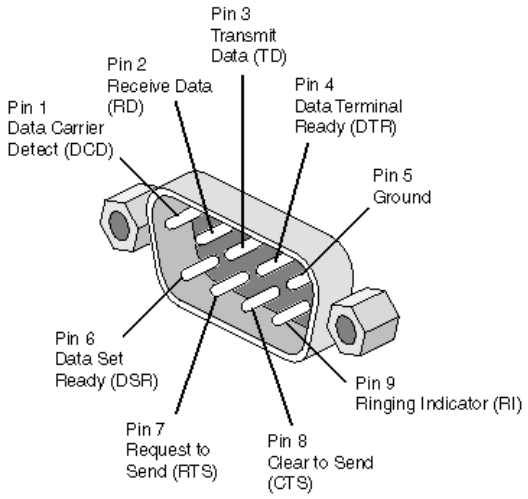


Figura 1: Conector macho utilizado en el estandar RS-232

Los datos se transmiten a cierta velocidad conocida como **taza de baudios** o **baud rate**. La velocidad esta dada en bits por segundo, las mas comunes son 4800, 9600, 56000, 115200.

Existe ciertos bits *especiales* utilizados para designar el inicio y fin de la comunicación, como también lo hay para chequeo de errores.

El bit de chequeo de error también se conoce como bit de paridad, esta puede ser **par** o **impar** y se obtiene contando el numero de **1** que contiene el dato, suponiendo que la paridad **par** esta seleccionada y el numero de bits es impar entonces el bit de paridad será 1. (Figura 2)

7 bits of data	(count of 1 bits)	8 bits including parity	
		even	odd
0000000	0	00000000	00000001
1010001	3	10100011	10100010
1101001	4	11010010	11010011
1111111	7	11111111	11111110

Figura 2: Cálculo de paridad.

Una forma mas facil de calcular la paridad es aplicando el operador modulo (Ecuación 1)

$$n\,\%2 = p \tag{1}$$

Sustituyendo n (el conteo de bits) se obtiene una paridad p que es **par**, para la **impar** solo se invierte el bit.

La transmisión serial se sincroniza con los bits de inicio y los de fin, normalmente se utiliza el código ASCII para comunicarse. Un ejemplo de transmision de la letra **T** en codigo ASCII se da en la figura 3.

Figura 3: ASCII 0x54 = 01010100 **T** enviado sin paridad.

Figura 4: ASCII 0x54 = 01010100 **T** con paridad **par**.

Figura 5: ASCII 0x54 = 01010100 **T** con paridad **impar**.

3. Desarrollo

Los datos entran de forma serial en **RxD** en cada ciclo de lectura y se pasan al registro **rx_data**, al finalizar la recepción tambien se activa el bit **rdrf** (*received data ready flag*).

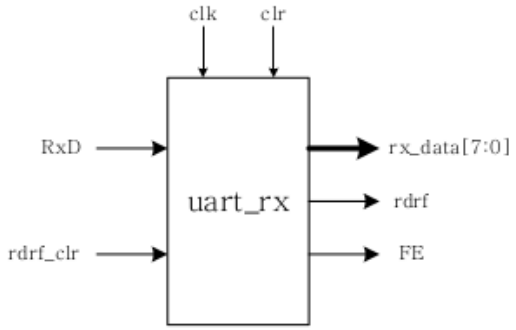


Figura 6: Diagrama de bloques del receptor UART.

El modulo se implementa como una maquina de estados, como se muestra en la figura 7.

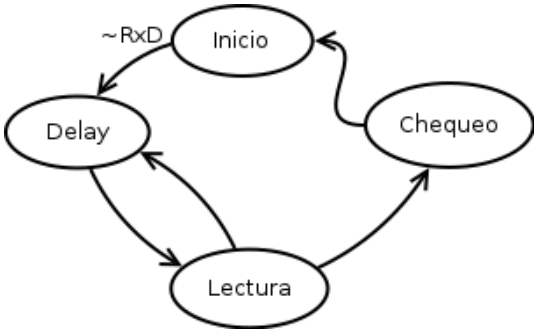


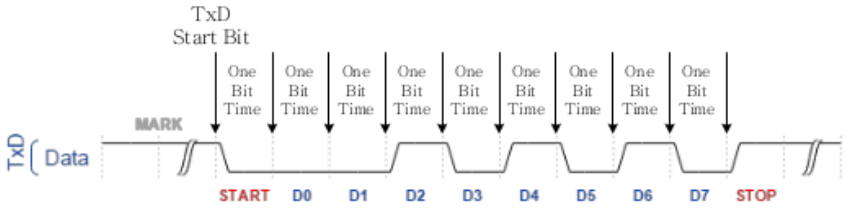
Figura 7: Diagrama de estados del receptor UART.

El reloj funciona a una velocidad de 50Mhz, por lo que se utiliza un

retraso (delay) para sincronizar la señal a 9600bps diviendo el reloj de 50Mhz entre 5208.

$$\frac{50,000,000}{5208} = 9600{,}61443932$$

Durante el estado de inicio o reposo, la entrada **RxD** esta en *alto*, la comunicación empieza cuando **RxD** cambia a *bajo* o 0.



A partir de ese momento se va al estado de espera (*delay*) hasta que transcurre el tiempo suficiente y empiece a recibir el primer dato, se repite el ciclo delay-recepción hasta que recibe todos los bits de datos, paridad y stop.

En verilog se utilizaron 2 bloques `always`, el primero para manejar la transición de estados y el segundo para dirigir las señales y realizar las operaciones pertinentes.

4. Resultados

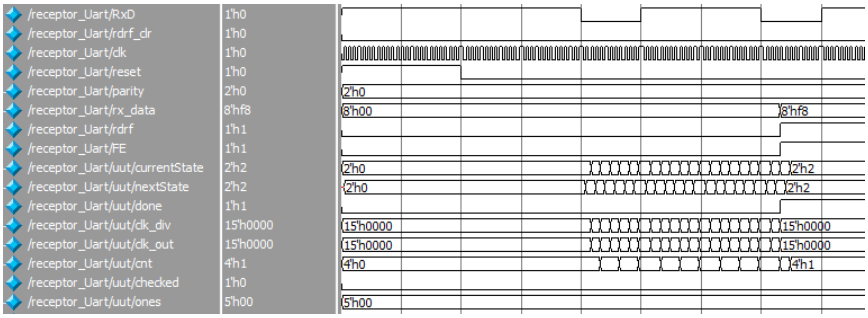


Figura 8: Modulo receptor en funcionamiento.

5. Conclusiones

La verificación es una tarea esencial en el diseño de circuitos, en este caso tuvo mayor complejidad que las tareas anteriores y se notó que el uso de scripts pudieron haber facilitado la tarea.

6. Apéndice

6.1. Código fuente

```
module receptor
  #(parameter DATA_LENGHT = 8,  // Datos de 8 bits
    parameter DELAY_TIME = 1)   // 50Mhz/5208=9600.61443932bps
   (input wire                   RxD,
    input wire                   rdrf_clr, clk, reset,
    input wire [1:0]             parity,
    output reg [DATA_LENGHT-1:0] rx_data,
    output reg                   rdrf, FE);

   reg [1:0]                     currentState, nextState;

   localparam START = 0, READ = 1, CHECK = 2, DELAY = 3; // Estados

   reg [DATA_LENGHT+1:0]         rx_data_tmp;      // Registro temporal de datos
   reg                           done;             // Bandera de finalizacion de lectu
   reg [14:0]                    clk_div, clk_out; // Divisor de frecuencia
   reg [3:0]                     cnt;              // Contador de para recibir datos y
   reg                           checked;          // Bandera de finalizacion de parit
   reg [4:0]                     ones;             // Cuenta cuantos '1' existen
   reg                           parity_error;     // Bandera que designa error en par

   always @(posedge clk, posedge reset)
     if (reset)       currentState <= START;
     else             currentState <= nextState;

   always @(posedge rdrf_clr)
     rdrf <= 0;

   always @(posedge clk)                    // Control de estados
     case (currentState)
```

```
    START:
      if (RxD)      nextState = START;
      else          nextState = DELAY;
    DELAY:
      if (clk_out) nextState = READ;
      else          nextState = DELAY;
    READ:
      if (done)     nextState = CHECK;
      else          nextState = DELAY;
    CHECK:
      if (checked) nextState = START;
      else          nextState = CHECK;
  endcase // case (currentState)

always @(currentState)                      // Flujo de datos
  case (currentState)
    START:
      begin
         {rx_data, rdrf, FE, ones} = 0;     // Poner senhales internas en cero
         {cnt, done, checked} = 0;          // Poner banderas en cero
         {clk_out, clk_div} = 0;            // Poner el divisor de frecuencia en cer
         parity_error = 0;
      end
    DELAY:
      begin
         clk_div = clk_div + 1;             // Divisor de frecuencia
         if (clk_div >= DELAY_TIME - 1)
           clk_out = 1;
      end
    READ:
      begin
         {clk_out, clk_div} = 0;            // Poner el divisor de frecuencia en cer
         rx_data_tmp[cnt] = RxD;            // Guardar dato

         cnt = cnt + 1;
         if (cnt > DATA_LENGHT + 1)
           begin
              done = 1;
              rdrf = 1;
              cnt = 0;
```

```
                    rx_data = rx_data_tmp[7:0]; // Mover datos recibidos al registro Rx
                    if (~rx_data_tmp[9])        // Chequeo de stop bit
                      FE = 1;
                  end
              end
            CHECK:
              begin
                  ones = ones + rx_data[cnt];
                  cnt = cnt + 1;
                  if (cnt > DATA_LENGHT)
                    begin
                      checked = 1;
                      if (parity == 2'b00 || parity == 2'b11) // No parity
                        parity_error = 0;
                      else if (parity == 2'b01)               // Even parity
                        parity_error = rx_data_tmp[DATA_LENGHT] == (ones % 2);
                      else                                    // Odd parity
                        parity_error = rx_data_tmp[DATA_LENGHT] == ~(ones % 2);
                  end
              end
          endcase // case (currentState)

endmodule // receptor
```