Desenvolvimento de circuitos reconfiguráveis para interface série usando o protocolo RS-232

Daniel Baptista

***Resumo* – Este artigo descreve como implementar, em circuitos reconfiguráveis, uma interface série utilizando o protocolo RS-232. Este protocolo, ao longo dos tempos, tem sido muito usado para comunicação série entre sistemas electrónicos. Por ser standard, permite interligar dispositivos facilmente. Actualmente, o protocolo RS-232 tem sido, gradualmente, substituída pelos controladores USB (*universal serial bus*) para a comunicação local, visto estes serem mais rápidos nas transferências de dados. Mesmo assim, o protocolo RS-232 continua a ser utilizado em periféricos de baixa capacidade de processamento, onde não se pode desenvolver software para processamento. Por esta razão, computadores e outros dispositivos electrónicos continuam a ser produzidos com portas RS-232, tanto *on board* como em placas para barramentos PCI ou em barramento ISA.**

***Abstract* – This article describes how to implement a serial interface using RS-232 protocol. Along the time this protocol has been used for serial communication between electronic systems. Because it is a standard protocol, it allows us to easily connect devices. Nowadays, the RS-232 protocol is being gradually replaced by USB (universal serial bus) controllers, for the local communications, because these controllers allow faster data transfers. Even though, the RS-232 protocol is still used in peripherals with low processing capability that do not support processing software. For this reason, computers and other electronic devices are still produced with RS-232 ports on board, on PCI bus boards, or on ISA bus.**

I. INTRODUÇÃO

Este artigo tem como base um projecto final para a disciplina de Sistemas Digitais Reconfiguráveis (SDR). Para este projecto foi necessário estudar o protocolo RS-232 com grande detalhe, tendo em consideração as taxas de transmissão, a construção da trama de envio com os seus bits de sinalização e ainda os vários estados que a linha pode tomar. Com este estudo concluído, passamos para a implementação em VHDL sintetizável, com ferramentas de projecto assistido por computador e em seguida, configurámos os vários controladores em FPGA, para esta poder comunicar utilizando o protocolo RS-232. Para testar o projecto foi utilizada a placa TE-XC2Se [1], pois esta já tem todo o hardware necessário à realização deste projecto, e ainda, o *ModelSim XE III 6.1e* [2], que permite simular circuitos com base na descrição em VHDL. O resto deste artigo está organizado em três secções. Na secção **II. Protocolo RS-232**, este protocolo é descrito com algum pormenor. Na secção **III. Implementação em VHDL**, é demonstrado como o protocolo RS-232 pode ser especificado em VHDL e implementado na placa TE-XC2Se. Por fim, na secção **IV. Conclusão**, são descritas as conclusões deste projecto.

II. PROTOCOLO RS-232

No protocolo RS-232 [3] foi convencionado que a linha está em repouso no estado lógico um, isto é, no estado inactivo ou *idle* indicando que a linha não está a ser utilizada. No início de uma transmissão, o emissor comuta a linha para o zero lógico durante um bit, o que é designado por *start bit*. O *start bit* indica ao receptor que vai iniciar-se uma sequência de dados, e serve ainda para sincronizar o relógio do receptor.

No protocolo RS-232 os bits de dados, ou *data bits*, são os que transportam realmente a informação sendo os outros bits de controlo. O número de bits de dados pode ser configurável, tendo as seguintes possibilidades: 5, 6, 7 e 8 bits. Normalmente são utilizados 7 e 8 bits de dados para a manipulação do código ASCII.

A estes bits de dados pode ser acrescentado um bit de paridade que, por ser opcional, normalmente, não é utilizado. O bit de paridade serve para a verificação da trama de dados, ou seja, permite detectar a ocorrência de alguns tipos de erros durante a transmissão.

A paridade pode ser par ou ímpar. Na paridade par, se existir um número par de uns lógicos o bit de paridade virá a zero lógico, caso exista um número ímpar de uns lógicos o bit de paridade virá a um lógico. Na paridade

ímpar é o inverso, se o número de uns lógicos for ímpar então o bit de paridade virá a zero lógico, caso o número de uns lógicos seja par o bit de paridade será um lógico.

Para terminar a sequência, temos um ou mais bits de paragem ou *stop bits*. Estes podem ser configurados com 1, 1.5 ou 2 bits de duração e o seu estado lógico é um. Os *stop bits* são necessários para proporcionar um intervalo de tempo mínimo entre duas transmissões consecutivas enviadas sobre a linha e dando indicação imediata de que a linha está no estado inactivo ou *idle*.

Assim, temos uma representação gráfica do protocolo RS-232:

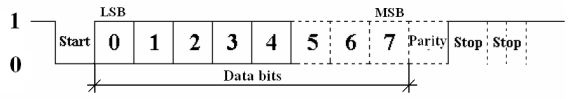

Fig. 1 – Representação da trama do protocolo RS-232.

Convencionou-se uma notação para as opções de configuração de uma ligação. A notação tem o formato *Nº Data bits / Parity (Null, Odd or Even) / Nº Stop bits*. A paridade pode ser nula (N), ímpar (O) ou par (E). Então, se tivermos a notacão 7/E/2 significa que temos sete bits de dados, um bit de paridade par e dois bits de paragem. A configuração mais comum é a 8/N/1, que especifica que são transmitidos 8 bits de dados, sem paridade e um bit de paragem.

Por fim, o protocolo RS-232 tem uma taxa de transmissão ou *baud rate* configurável, isto é, é possível definir o número de bits por segundo passíveis de serem transmitidos numa dada ligação. As taxas mais comuns de transmissão são, entre outras, 300, 1200, 2400, 9600, 19200, 38400, 57600, 115200, 230400 bits/s. Numa comunicação utilizando o protocolo RS-232, é recomendável que ambos os dispositivos estejam configurados com a mesma taxa de transmissão, podendo alguns dispositivos ser configurados para auto--detectar essa mesma taxa.

III. IMPLEMENTAÇÃO EM VHDL

A. Sincronização de transmissão e de recepção

Para a transmissão e recepção os dois dispositivos em comunicação devem estar configurados com as mesmas características, isto é, o mesmo *baud rate*, o mesmo número de bits de dados, se existe paridade, se esta é ímpar ou par, e o número de bits de paragem. O receptor vai sincronizar-se pelo flanco negativo do *start bit* e vai fazer as leituras a meio do intervalo reservado ao bit. Contudo, a frequência do receptor não pode ser igual à de transmissão, pois basta existir um erro de fase entre os dois relógios e as diferenças de frequência darão origem a erros cumulativos, o que pode originar erros na leitura, como podemos ver na Fig. 2:

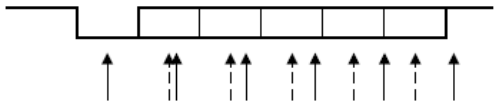

Fig. 2 – Erros de leitura devido às diferenças de frequência.

Para se evitar tal situação utilizam-se relógios cuja frequência é:

$$f_{CLK} = N \times Baud\ rate = N \times \frac{1}{T_{Bit}} \quad (1)$$

Em que N, tem valores típicos de 16 ou 64. Deste modo, podemos evitar as diferenças de fase e com N=16 é possível obter um erro de fase cujo valor máximo é $T_{bit}/16$. Assim temos o relógio do receptor representado pela Fig.3.

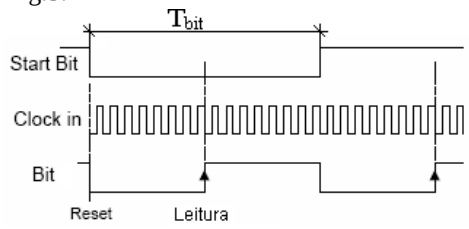

Fig. 3 – Relógio do receptor.

Em diversos dispositivos electrónicos, a sua frequência não é compatível com a frequência do *baud rate* de transmissão, pois, normalmente, é necessário dividir a frequência do dispositivo por uma constante. E assim, chegamos à fórmula.

$$Const_{DIV} = \frac{f_{CLK}}{N \times Baud\ rate} \quad (2)$$

Mesmo que a constante não seja inteira é sempre possível arredondar para o inteiro mais próximo porque o erro de fase de $T_{bit}/16$ é tolerável.

B. Descrição do controlador em VHDL

Neste ponto podemos definir a interface externa para o controlador RS-232, de acordo com a Fig. 4.

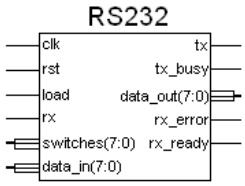

Fig. 4 – Interface externa para o controlador RS-232.

Os sinais de entrada deste controlador são os seguintes: *clk* é o sinal de relógio do circuito, *rst* é o sinal de *reset* da FPGA, *load* é o bit de controlo de leituras de dados de entrada, *rx* é o bit de entrada recebido pela porta série, *switches* é o registo de entrada para selecção das configurações da ligação e *data_in* é o registo de dados a transmitir. Para os sinais de saída do controlador são os segiuntes: *tx* é o bit de transmissão, *tx_busy* é o bit de sinalização de transmissão a decorrer, *data_out* é o registo dos bits recebidos, *rx_error* é o bit de sinalização de erro e o *rx_ready* é o bit que sinaliza o fim de transmissão.

A interface externa do controlador pode ser descrita em VHDL da maneira seguinte:

```
entity RS232 is
Port (
clk: in std_logic;
      rst : in std_logic;
      switches: in std_logic_vector (7 downto 0);
      data_in : in std_logic_vector (7 downto 0);
      load : in std_logic;
      rx : in std_logic;
      tx : out std_logic;
      tx_busy: out std_logic;
      data_out : out std_logic_vector(7 downto 0);
      rx_ready : inout std_logic;
      rx_error : out std_logic
);
end RS232;
```

Necessariamente, é preciso definir mais alguns bits e registos para o controlador RS-232. A explicação destes bits e registos, está descrita logo a seguir à sua declaração em VHDL, devido ao elevado número de bits e registos.

```
architecture Behavioral of RS232 is

-- maquina de estados de transmissao
type STATE_TYPE1 is (idle_tx, load_data_tx, shift_tx, stop_tx);
signal fsm_tx : STATE_TYPE1;
-- maquina de estados de recepcao
type STATE_TYPE2 is (idle_rx, start_bit_rx, edge_bit_rx,
middle_bit_rx, stop_rx, error_rx );
signal fsm_rx: STATE_TYPE2;

signal divisor: natural range 0 to 1400; -- constante de divisao
signal cnt_16: natural range 0 to 1400; -- constante de divisao
signal cnt_tx: natural; -- contador para transmissao

signal cnt_rx: natural; -- contador para recepcao
signal tx_bit_num: natural; -- n. de bits a transmitir
signal rx_bit_num: natural; -- n.de bits a receber
signal rst_cnt_rx: std_logic; -- reset contador rx
signal clk_16: std_logic; -- relogio N x baud rate
signal clk_tx: std_logic; -- relogio do transmissor
signal clk_rx: std_logic; -- relogio do receptor
-- registo de dados a transmitir
signal data_in_reg : std_logic_vector (7 downto 0);
-- registo de trama a transmitir
signal reg_tx : std_logic_vector (9 downto 0);
--registo de trama a receber
signal reg_rx : std_logic_vector (7 downto 0);
signal parity : std_logic; -- bit de paridade
signal num_data_bits : integer; -- n. data bits
signal even : std_logic; -- paridade par
signal par : std_logic; -- bit com o conteudo da paridade
signal stop_bits : std_logic; -- stop bit
signal stop2 : std_logic; -- 2 stop bits
signal index :integer; -- index do for

    begin
```

A placa TE-XC2Se tem dois geradores de relógio, um de 48MHz e outro de 25MHz. Neste projecto foi utilizada a frequência de 25MHz, devido à interacção com o controlador VGA [4,5] para visualizar os dados transmitidos. Esta placa tem adicionalmente um conjunto de oito *dipswitches*, que são controlados pelo CPLD [1,4,5] da placa TE-XC2Se. Assim, utilizamos os *dipswitches* para configurar os parâmetros da ligação que descrevemos anteriormente.

```
-- Seleccao do Baud rate, bit de paridade, stop bit, nº de bites de dados

process (rst, clk)
begin
   if rst='0' then
      divisor <= 0;
      parity <= '0';
      num_data_bits <= 8;
      even <= '1';
      stop_bits <='0';
   elsif rising_edge(clk) then
      case switches(3 downto 0) is
         when "0000" =>divisor <= 7; -- baud rate 230.400
         when "0001" =>divisor <= 14; -- baud rate 115.200
         when "0010" =>divisor <= 27; -- baud rate 57.600
         when "0011" =>divisor <= 41; -- baud rate 38.400
         when "0100" =>divisor <= 81; -- baud rate 19.200
         when "0101" =>divisor <= 163; -- baud rate 9.600
         when "0110" =>divisor <= 326; -- baud rate 4.800
         when "0111" =>divisor <= 651; -- baud rate 2.400
         when "1000" =>divisor <= 1302; --  baud rate1.200
         when others =>divisor<=14; -- default baud 115.200
      end case;
```

Os primeiros quatro dipswitches de um conjunto de oito, configuram o *baud rate*. Conforme os dipswitches estão ligados ou desligados, assim é escolhida a constante de divisão para gerar o relógio correspondente ao *baud rate*.

```
        if (switches(4)='1')then
            parity <= '1'; -- parity
        else
            parity <= '0'; -- no parity
        end if;

        if (switches(5)='1')then
            even <= '0'; -- parity odd
        else
            even <= '1'; -- parity even
        end if;

        if (switches(6)='1')then
            num_data_bits <= 7; -- 7 data bits
        else
            num_data_bits <= 8; -- 8 data bits
        end if;

         if (switches(7)='1')then
            stop_bits <='1'; -- 2 stop bits
         else
            stop_bits <='0'; -- 1 stop bits
         end if;

    end if;
end process;
```

Os outros quatro *dipswitches* configuram a paridade (se é nula, par ou ímpar), o número de *data bits* e o número de *stop bits*. Deste modo, obtemos a configuração desejada seleccionando os *dipswitches* pretendidos. Como é também visível, não foram implementadas todas as opções do protocolo RS-232, mas facilmente se podem implementar por analogia do código descrito. A constante de divisão para seleccionar o *baud rate* está calculada para a frequência de 25MHz. Este parâmetro pode ser alterado conforme a frequência de relógio dos diversos circuitos, mas tendo em consideração a fórmula (2).

Neste ponto, é necessário gerar o relógio com dezasseis vezes a frequência de *baud rate* (N x *baud rate*, com N igual a 16), como descrito anteriormente, para sincronizar o sinal no receptor. A implementação feita em VHDL foi um simples contador de impulsos [2,4,5], pois quando o número de impulsos contados for igual à constante de divisão é gerado um impulso que é um quociente do relógio original.

```
process (rst, clk)
begin
    if rst='0' then
            clk_16 <= '0';
            cnt_16 <= 0;
    elsif rising_edge(clk) then
            clk_16 <= '0';
            if cnt_16 = (divisor-1) then
                    cnt_16 <= 0;
                    clk_16 <= '1';
            else
                    cnt_16 <= cnt_16 + 1;
            end if;
    end if;
end process;
```

No caso do transmissor, o relógio pode ser da mesma frequência que o *baud rate*. Assim, basta dividir o relógio anterior por dezasseis, utilizando um código semelhante ao apresentado acima:

```
process (rst, clk, clk_16)
begin
    if rst='0' then
            clk_tx <= '0';
            cnt_tx <= 0;
    elsif rising_edge(clk) then
            clk_tx <= '0';
            if clk_16='1' then
                    cnt_tx <= cnt_tx + 1;
                    if cnt_tx = 15 then
                            clk_tx <= '1';
                            cnt_tx <= 0;
                    end if;
            end if;
    end if;
end process;
```

Para o caso da recepção, como visto na Fig. 3, o sincronismo é feito pelo relógio de N x *baud rate*, mas as leituras são feitas no centro do bit, para que estas sejam as mais correctas possíveis. Assim, para gerar este relógio de leitura, é necessário dividir o relógio N x *baud rate* por oito, tendo em atenção que o contador deste relógio tem necessariamente de ser posto a zero, quando se detecta um *start bit*.

```
process (rst, clk, rst_cnt_rx, clk_16)
begin
    if rst='0' then
        clk_rx <= '0';
```

```
        cnt_rx <= 0;
    elsif rising_edge(clk) then
        clk_rx <= '0';
        if rst_cnt_rx='1' then  --sincronizacao com start bit
            cnt_rx <= 0;
        elsif clk_16='1' then
            if cnt_rx = 7 then
                cnt_rx <= 0;
                clk_rx <= '1';
            else
                cnt_rx <= cnt_rx + 1;
            end if;
        end if;
    end if;
end process;
```

C. Implementação do transmissor

Já com os sinais de relógio todos gerados, podemos passar à implementação do transmissor. Este foi elaborado através da seguinte máquina de estados finitos [6,7,8]:

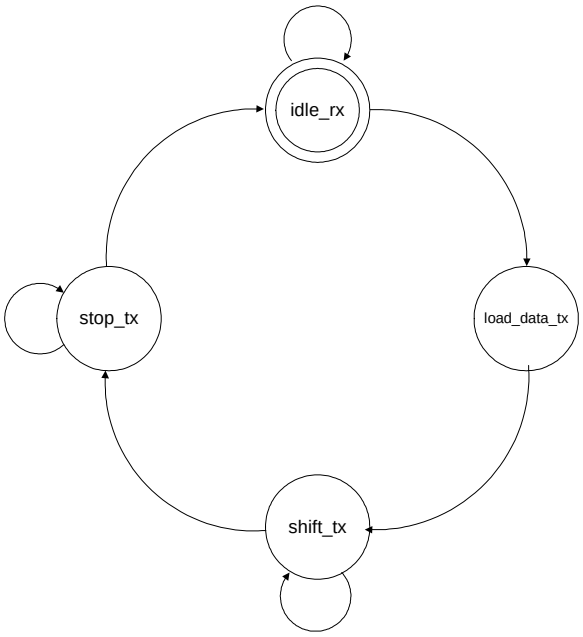

Fig. 5 – Máquina de estados finitos do transmissor.

O primeiro estado da máquina de estados de transmissão é o de *idle_tx*. Neste estado fica-se continuamente em ciclo de espera até que haja uma trama para envio; logo que exista uma trama passa-se para o estado seguinte. O segundo estado é o de *load_data_tx*. Neste estado é formulada a trama completa de envio de acordo com as configurações tomadas. Em seguida, passa-se para o estado de *shift_tx*. Neste terceiro estado é enviada a trama completa, isto é, a trama é transmitida bit a bit até ao *stop bit*. O último estado é o de *stop_tx*. Neste estado fica-se em ciclo de espera até enviar o número de *stop bits* configurado. Após serem enviados os *stop bits* volta-se para o estado de *idle_tx*.

Esta máquina de estados foi implementada em VHDL da seguinte maneira:

```
process (rst, clk, par, stop_bits, load, data_in, clk_tx, num_data_bits)
begin
    if rst='0' then
        par<='0';
        stop2 <='0';
        index <=0;
        reg_tx <= (others => '1');
        tx_bit_num <= 0;
        fsm_tx <= idle_tx;
        tx_busy <= '0';
        data_in_reg <= (others=>'0');
    elsif rising_edge(clk) then
        tx_busy <= '1';--so vai a '0' quando não tem bits a enviar
        case fsm_tx is
            when idle_tx =>
                if load='1' then
                    -- leitura dos dados de entrada
                    data_in_reg <= data_in;
                    tx_busy <= '1';
                    fsm_tx <= load_data_tx;
                else
                    tx_busy <= '0';
                end if;
```

O primeiro estado, da máquina de estados finitos da transmissão é o de *idle*, pois só é possível começar uma transmissão se a linha estiver em repouso. Neste caso, a máquina de estados espera pela leitura de um barramento em paralelo, o que só acontece quando o bit de *load* está a um lógico. Feita a leitura, passa-se para o estado de *load_data_tx* e indica-se em *tx_busy* que está uma transmissão a ser efectuada.

```
when load_data_tx =>
    if clk_tx='1' then
            fsm_tx <= shift_tx;
            if parity='1' then
                -- start + data + parity
                tx_bit_num <= ( num_data_bits + 2);
                par <=data_in_reg(0);                                    -
- criar a paridade
                if even = '1' then
                    for index in 1 to 8-1 loop
                        par <= data_in_reg(index)xor par;
                    end loop;
```

```
                else
                    for index in 1 to 8-1 loop
                        par <= data_in_reg(index)xnor par;
                    end loop;
                end if;
                if num_data_bits=8 then
                    reg_tx <= par & data_in_reg & '0';
                 else
          reg_tx <= '1' & par & data_in_reg(6 downto 0) & '0';
                 end if;
            else
                -- start + data
                tx_bit_num <= ( num_data_bits + 1);
                if num_data_bits=8 then
                    reg_tx <= '1' & data_in_reg & '0';
                 else
                    -- só quando são 7 bits
               reg_tx <= "11" & data_in_reg(6 downto 0) & '0';
                 end if;
            end if;
end if;
```

No estado *load_data_tx* (descrito no código anterior), na máquina de transmissão, é formada uma sequência de envio de acordo com as opções tomadas inicialmente. Neste estado, é calculada a paridade das sequências de dados e junta-se-lhes o *start bit* e o bit de paridade, se existir; se não existir, será posto um bit a um lógico no seu lugar, que por sua vez, já serve de *stop bit*.

```
when shift_tx => -- shift dos bits
    if clk_tx='1' then
            tx_bit_num <= tx_bit_num - 1;
            reg_tx <= '1' & reg_tx(9 downto 1);
    if tx_bit_num=1 then
            fsm_tx <= stop_tx;
    end if;
end if;
```

O estado *shift_tx*, apresentado no código acima, é um simples registo de deslocamento *shift register* [2, 4, 5], em que se vai descartando o bit menos significativo da sequência e vão sendo inseridos uns lógicos nos bits mais significativos. Este procedimento é efectuado até que todos os bits sejam enviados.

```
when stop_tx => -- stop
    if clk_tx='1' then
            if stop_bits = '1' and stop2='0' then
                -- 2 stop bits
                fsm_tx <=stop_tx;
                stop2 <= '1';
            else
                fsm_tx <= idle_tx;
                stop2 <='0';
            end if;
end if;
```

Este último estado (apresentado no código acima) é o de *stop_tx*. Neste estado não existe nenhum processamento lógico. A função deste estado é apenas esperar o tempo dos bits de paragem já que foram acrescentados uns lógicos nos bits mais significativos da sequência.

```
            when others =>
                fsm_tx <= idle_tx;
            end case;
        end if;
end process;

tx <= reg_tx(0);
```

Por fim, é necessário definir que o bit a transmitir é igual ao bit de índice zero da sequência formulada. Como já foi visto, a deslocação do registo é feita no estado de *shift_tx*.

D. Implementação do receptor

A máquina de estados finitos de recepção [6,7,8] foi elaborada da seguinte maneira:

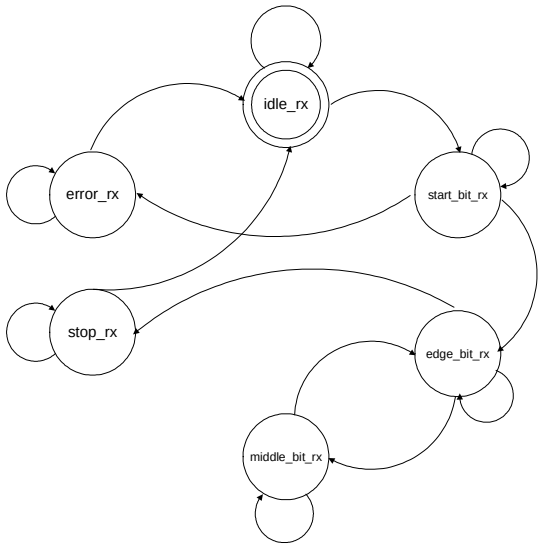

Fig. 6 – Máquina de estados finitos do receptor.

O primeiro estado, da máquina de estados de recepção é o de *idle_rx*. Neste estado verifica-se continuamente a linha de recepção até que se detecte um *start bit*. Caso este seja detectado passamos para o estado seguinte. O

segundo estado é o de *start_bit_rx*. Neste estado é verificado se o bit detectado inicialmente corresponde mesmo a um *start bit* de sinalização de início de trama, pois, existem muitos casos onde não corresponde a um *start bit*, como vamos ver neste artigo. O terceiro estado é o de *edge_bit_rx*, que se situa junto à transição dos bits de recepção, daí o seu nome. Neste estado é feita uma espera pelo centro do bit recebido. Quando o número total de bits de dados é atingido passa-se para o estado de *stop_rx*, caso contrário passa-se para o estado de *middle_bit_rx*. Neste quarto estado (*middle_bit_rx*), é feita a leitura da linha de recepção no centro do bit e espera-se pela transição do bit seguinte em que se volta para o estado de *middle_bit_rx*. O quinto estado é o de *stop_rx*. Neste estado é feita a espera pelos *stop bits*, voltando-se para o estado inicial. O sexto e último estado é o de *error_rx*. Caso exista um erro é assinalado neste estado, quebrando-se assim a sequência de estados. Este ainda espera que a linha de recepção passe para o estado de *idle*, para uma nova recepção.

```
process (rst, clk, rx, clk_rx, num_data_bits )
begin
    if rst='0' then
        reg_rx <= (others => '0');
        data_out <= (others => '0');
        rx_bit_num <= 0;
        fsm_rx <= idle_rx;
        rx_ready <= '0';
        rst_cnt_rx <= '0';
        rx_error <= '0';
    elsif rising_edge(clk) then
        rst_cnt_rx <= '0'; --  sempre a zero ate existir o start bit
        -- reset erro quando a palavra foi bem recebida
        if rx_ready='1' then
            rx_error <= '0';
            rx_ready <= '0';
        end if;
          case fsm_rx is
                when idle_rx => -- espera pelo start bit
                  rx_bit_num <= 0;
                  if rx='0' then
                      rst_cnt_rx <='1'; -- sincroniza o divisor
                      fsm_rx <= start_bit_rx;
                  else -- falso inicio. fica em Idle
                      rst_cnt_rx <='0';
                    end if;
```

Após uma transmissão o sinal *rx_ready* é posto a um lógico, indicando que tudo correu bem. No início de uma nova transmissão este sinal tem que ser novamente reposto a zero lógico. O primeiro estado de recepção é o de *idle_rx*, ficando sempre neste estado até que seja detectado um *start bit*. Logo que este apareça, é necessário sincronizar o relógio do receptor.

```
when start_bit_rx => -- espera pelo 1º bit de dados
    if clk_rx='1' then
        if rx='1' then -- framing error
            fsm_rx <= error_rx;
        else
            fsm_rx <= edge_bit_rx;
        end if;
     end if;
```

No estado anterior, ao ser detectado o primeiro bit da trama é necessário verificar se é mesmo um *start bit*, pois em muitos casos pode ser apenas um bit de uma transmissão que já esteja a decorrer, ou um pico de ruído, que podem levar a uma falsa iniciação de trama.

```
when edge_bit_rx => -- Transicao do bit
    if clk_rx='1' then
        fsm_rx <= middle_bit_rx;
        if rx_bit_num = num_data_bits then
            fsm_rx <= stop_rx;
        else
            fsm_rx <= middle_bit_rx;
        end if;
    end if;
```

Para que a leitura possa ser a mais fiel possível, o bit tem que ser lido exactamente no seu centro. Assim, a função deste estado é esperar pelo centro do bit, como visto anteriormente

```
when middle_bit_rx => -- leitura dos dados no centro bit
    if clk_rx='1' then
        rx_bit_num <= rx_bit_num + 1;
        -- shift right
        reg_rx <= rx & reg_rx(7 downto 1);
        fsm_rx <= edge_bit_rx;
    end if;
```

Neste estado *middle_bit_rx*, é feita a leitura da linha de recepção, exactamente no centro do bit, para um registo de deslocamento. Após a leitura volta ao estado anterior de espera pelo meio do bit até atingir o número de *data bits* configurado.

```
when stop_rx => -- Stop bit
    if clk_rx='1' then
            data_out <= reg_rx;
            rx_ready <='1';
            fsm_rx <= idle_rx;
    end if;
```

No estado de *stop_rx*, a leitura é colocada na porta de saída e o *rx_read* é colocado a um lógico indicando que chegou ao fim da trama lida, durante o tempo de *stop bit*, em que este bit pode ser descartado.

```
            when error_rx => -- Overflow / Error
                rx_error <= '1';
                if rx='1' then
                    fsm_rx <= idle_rx;
                end if;
                    end case;
                end if;
    end process;

end Behavioral;
```

O último estado é o de *error_rx*. Sempre que seja detectado um erro, seja em que estado for, é sinalizado neste estado que existiu um erro de transmissão, pondo o sinal *rx_error* a um lógico. Em seguida, espera-se pela linha no estado *idle* e volta-se para o estado *idle_rx*.

E. Implementação da placa TE-XC2Se

A placa TE-XC2Se tem o *hardware* implementado à comunicação via porta série, mas é necessário configurar a FPGA para podermos ligar o *buffer* de entrada aos pinos da FPGA. Para isso, basta ligar de acordo com a figura seguinte, mas tendo em consideração que as linhas estão cruzadas na placa.

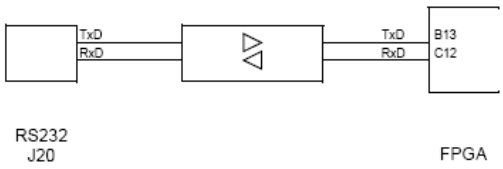

Fig. 7 – Ligação da porta série aos pinos da FPGA XC2S300E da placa TE-XC2Se.

IV. Conclusão

Neste artigo foi descrito como construir um controlador de comunicação série utilizando o protocolo RS-232, com base em circuitos reconfiguráveis para FPGA. Foi feita uma abordagem às características do protocolo em questão e implementado o código em VHDL, devidamente ilustrado, para a placa TE-XC2Se.

A principal dificuldade à implementação do código é a grande gama de configurações possíveis para o protocolo RS-232. Como se pode verificar no código, este apenas foi implementado para algumas configurações, não cobrindo todas as opções possíveis.

Outra dificuldade foi a sincronização do receptor. Para melhor eficácia deste, a verificação de *start bit* está sincronizada com o relógio da FPGA e não com o relógio N x *baud rate* conforme diz o protocolo RS-232.

Referências

[1] Trenz Electronic GmbH, Products, [Online]: http://www.trenz-electronic.de/home/indexen.htm.

[2] Xilinx, Inc., “Xilinx ISE 8 Software Manuals”, “XST User Guide”, “Libraries Guide”. ModelSim XE III/Starter 6.1e, Xilinx ISE 8.2 [Online]: http://www.xilinx.com/

[3] Fonseca, José A., “Barramentos e Interfaces de Comunicação Série”, Abril 2005

[4] Sklyarov, V., Skliarova, I., "Teaching Reconfigurable Systems: Methods, Tools, Tutorials and Projects", IEEE Transactions on Education, vol. 48, no. 2, Maio 2005

[5] Sklyarov, V., Skliarova, I., Tutorials. [Online]: http://www.ieeta.pt/~iouliia/Courses/SDR/tutorials/.

[6] Steiner, Glenn C., “A Software UART for the UltraController GPIO Interface”, XAPP699 (v1.0) March 3, 2004. [Online]: http://www.xilinx.com/

[7] Chapman, Ken, “200 MHz UART with Internal 16- Byte Buffer”, XAPP223 (v1.1) July 10, 2001. [Online]: http://www.xilinx.com/

[8] Chapman, Ken, “UARTs in Xilinx CPLDs”, XAPP341 (v1.3) October 1, 2002. [Online]: http://www.xilinx.com/