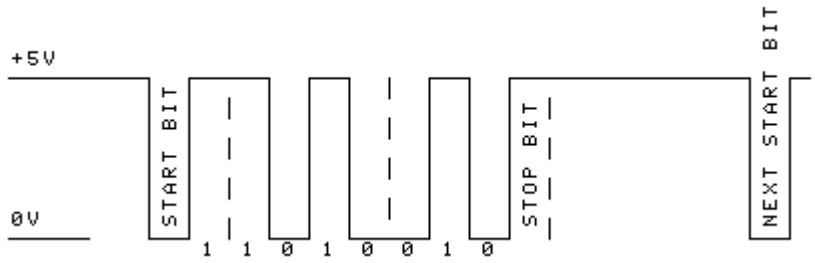
Projeto completo de uma UART em MyHDL - Gerando VHDL / Verilog e testando no FPGA

Por **André Castelan Prado** - 25/08/2014



ÍNDICE DE CONTEÚDO [MOSTRAR]

A UART é um velho conhecido de todo engenheiro de sistemas embarcados, é provavelmente o primeiro protocolo de comunicação que aprendemos na universidade. Neste artigo vamos implementar o nosso próprio circuito serial em MyHDL. Assim podemos fazer nosso próprio hardware que se comunica com nosso código em Java, C#, ou com outro microcontrolador como um Arduino.

Conforme vimos em um artigo anterior, MyHDL é uma linguagem de descrição de hardware em Python que possibilita gerar arquivos em VHDL e Verilog. Além é claro de prover todo o potencial do Python para o usuário, como por exemplo, bibliotecas para testes unitários.

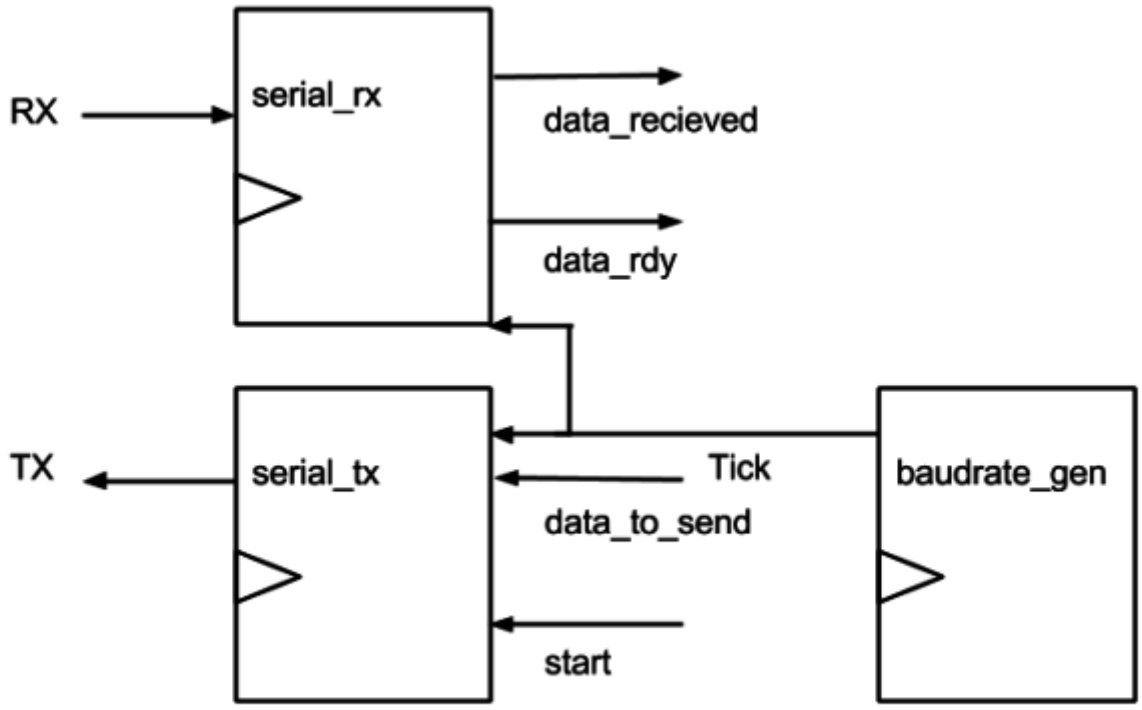
UART

Como sabemos, tanto o transmissor quanto o receptor entram em um acordo quanto à taxa de transferência (Baud Rate), o número de bits de dados, o número de stop bits e

outros detalhes mais. Neste exemplo vamos implementar uma UART com 8 bits de dados, 2 stop bits e 115200 bps de Baud Rate.

Nessa UART é composta por três módulos:

- **baudrate_gen:** Circuito responsável por gerar nossos pulsos de baudrate;
- **serial_tx:** Circuito responsável por enviar dados. Possui como entrada uma porta de 8 bits com o dado a ser enviado e uma porta de start. Quando start vai para '1' o dado é enviado pela porta de saída TX;
- **serial_rx**: Circuito responsável por receber dados. Possui como entrada a porta RX e como saída uma porta de 8 bits de dados e uma porta indicando quando os dados estão prontos.



Arquitetura da nossa UART em MyHDL

baudrate_gen

Este é o nosso módulo mais simples, com base no *clock* de entrada é gerada uma saída na taxa especificada. A fórmula é bem simples: Frequência do clock / Frequência do baudrate. Tenho um cristal de 50MHz no meu FPGA e desejo uma taxa 115200 Hz. 50.000.000 / 115.200 da aproximadamente 434. Então, a cada 434 pulsos do meu clock de 50 MHz vou gerar um pulso de baud, descrito como "tick" na nossa arquitetura.

Como entrada temos o clock (sysclk), um reset (reset_n), a taxa de baud desejada (baud_rate_i) . Como saída temos os ticks: half_baud_rate_o e baud_rate_o. A primeira vai para '1' na metade do tempo, neste caso, a cada 217 pulsos de clock de 50 MHz.

O código fica bem similar a uma descrição em VHDL ou Verilog. O módulo é tratado como uma função em Python, os argumentos das funções são as nossas portas. O MyHDL identifica automaticamente o que é entrada e o que é saída no módulo. Um registrador é inferido para guardar o número de pulsos, baud_gen_count_reg.

A keyword *Signal* indica que é um Sinal (equivalente ao VHDL). O tipo intbv se refere a INT BIT VECTOR, é algo semelhante a um STD_LOGIC_VECTOR . Com a diferença de que o MyHDL sabe na hora de gerar VHDL/Verilog, se vai ser um vetor lógico, um vetor com sinal ou vetor sem sinal.

```
1  from myhdl import *
2
3  def baudrate_gen(sysclk, reset_n, baud_rate_i, half_baud_rate_tick_o, baud_rate_tick_o):
4
```

```
5       """ Serial
6       This module implements a baudrate generator
7
8       Ports:
9       -----
10      sysclk: sysclk input
11      reset_n: reset input
12      baud_rate_i: the baut rate to generate
13      baud_rate_tick_o: the baud rate enable
14
```



```
18      half_baud_const = baud_rate_i//2
19
20      @always_seq(sysclk.posedge, reset = reset_n)
21      def sequential_process():
22          baud_gen_count_reg.next = baud_gen_count_reg + 1
23          baud_rate_tick_o.next = 0
24          half_baud_rate_tick_o.next = 0
25          if baud_gen_count_reg == baud_rate_i:
26              baud_gen_count_reg.next = 0
27              baud_rate_tick_o.next = 1
28          if baud_gen_count_reg == half_baud_const:
29              half_baud_rate_tick_o.next = 1
30
31
32      return sequential_process
```

O decorator @always_seq emite automaticamente o nosso PROCESS (VHDL) ou ALWAYS (VERILOG) de forma síncrona e reseta todos os registradores para zero. É sempre necessário ter ao final um return com todos os nossos process/always/decorators.

serial_tx

Este módulo é responsável por enviar os dados via serial. Tem-se uma máquina de estados que aguarda um sinal de (start_i) para iniciar o processo de enviar o byte (data_i) pela nossa saída de TX (transmit_o).

O código é bem semelhante a uma implementação em VHDL ou Verilog, descrevemos tudo a nível RTL. Temos agora também o decorator @always_comb que é equivalente a um PROCESS/ALWAYS puramente combinacional. Enviamos um bit a cada "tick" do nosso baudrate. Como ficou um pouco mais complexo, optei pelo estilo de RTL dividido em duas máquinas de estados, aonde os registradores são inferidos no decorator sequencial e a lógica combinacional no decorator combinacional.


```
from myhdl import *

t_State = enum('ST_WAIT_START', 'ST_SEND_START_BIT', 'ST_SEND_DATA' , 'ST_SEND_STOP_BIT' )


def serial_tx(sysclk, reset_n, start_i, data_i, n_stop_bits_i, baud_rate_tick_i, transmit_o):

    """ Serial
    This module implements a transmiter serial interface

    Ports:
    -----
    sysclk: sysclk input
    reset_n: reset input
    baud_rate_tick_i: the baud rate
    start_i: start sending data
    data_i: the data to send
    transmit_o: data output
    -----

    """
    END_OF_BYTE = 7

    state_reg = Signal(t_State.ST_WAIT_START)
    state = Signal(t_State.ST_WAIT_START)

    transmit_reg = Signal(bool(0))
    transmit = Signal(bool(0))

    count_8_bits_reg = Signal(intbv(0, min = 0, max = 8))
    count_8_bits = Signal(intbv(0, min = 0, max = 8))

    count_stop_bits_reg = Signal(intbv(0, min = 0, max = 8))
    count_stop_bits = Signal(intbv(0, min = 0, max = 8))

    @always_comb
    def outputs():
        transmit_o.next = transmit_reg

    @always_seq(sysclk.posedge, reset = reset_n)
    def sequential_process():
        state_reg.next    = state
        transmit_reg.next  = transmit
        count_8_bits_reg.next = count_8_bits
        count_stop_bits_reg.next = count_stop_bits

    @always_comb
    def combinational_process():
        state.next  = state_reg
        transmit.next = transmit_reg
        count_8_bits.next = count_8_bits_reg
        count_stop_bits.next = count_stop_bits_reg

        if state_reg == t_State.ST_WAIT_START:
```

```
55              transmit.next = True
56              if start_i == True:
57                  state.next = t_State.ST_SEND_START_BIT
58
59          elif state_reg == t_State.ST_SEND_START_BIT:
60              transmit.next = False
61              if baud_rate_tick_i == True:
62                  state.next = t_State.ST_SEND_DATA
63
64          elif state_reg == t_State.ST_SEND_DATA:
```



```
68                  count_8_bits.next = 0
69                  state.next = t_State.ST_SEND_STOP_BIT
70              else:
71                  count_8_bits.next = count_8_bits_reg + 1
72                  state.next = t_State.ST_SEND_DATA
73
74
75          elif state_reg == t_State.ST_SEND_STOP_BIT:
76              transmit.next = True
77              if baud_rate_tick_i == True:
78                  if count_stop_bits_reg == (n_stop_bits_i - 1):
79                      count_stop_bits.next = 0
80                      state.next = t_State.ST_WAIT_START
81                  else:
82                      count_stop_bits.next = count_stop_bits_reg + 1
83          else:
84              raise ValueError("Undefined State")
85
86
87
88      return outputs, sequential_process, combinational_process
```

serial_rx

Este módulo é responsável por transformar os dados que estão vindo via serial em um byte de informação. Aguarda-se o start bit no RX e armazena-se os próximos 8 bits de dados em um registrador de 1 byte. Assim que o dado (data_o) está pronto, o ready_o vai para nível lógico alto.

```
1  from myhdl import *
2
3  t_State = enum('ST_WAIT_START_BIT', 'ST_GET_DATA_BITS', 'ST_GET_STOP_BITS' )
4
5
6  def serial_rx(sysclk, reset_n, n_stop_bits_i, half_baud_rate_tick_i, baud_rate_tick_i, reciev
7
```

```
8       """ Serial
9       This module implements a reciever serial interface
10
11      Ports:
12      -----
13      sysclk: sysclk input
14      reset_n: reset input
15      half_baud_rate_tick_i: half baud rate tick
16      baud_rate_tick_i: the baud rate
17      n_stop_bits_i: number of stop bits
```



```
21      -----
22
23      """
24      END_OF_BYTE = 7
25
26      state_reg = Signal(t_State.ST_WAIT_START_BIT)
27      state = Signal(t_State.ST_WAIT_START_BIT)
28
29      data_reg = Signal(intbv(0, min = 0, max = 256))
30      data = Signal(intbv(0, min = 0, max = 256))
31      ready_reg = Signal(bool(0))
32      ready = Signal(bool(0))
33
34      count_8_bits_reg = Signal(intbv(0, min = 0, max = 8))
35      count_8_bits = Signal(intbv(0, min = 0, max = 8))
36
37      count_stop_bits_reg = Signal(intbv(0, min = 0, max = 8))
38      count_stop_bits = Signal(intbv(0, min = 0, max = 8))
39
40      @always_comb
41      def outputs():
42          data_o.next = data_reg
43          ready_o.next = ready_reg
44
45
46      @always_seq(sysclk.posedge, reset = reset_n)
47      def sequential_process():
48          state_reg.next   = state
49          data_reg.next  = data
50          ready_reg.next = ready
51          count_8_bits_reg.next = count_8_bits
52          count_stop_bits_reg.next = count_stop_bits
53
54      @always_comb
55      def combinational_process():
56          state.next  = state_reg
57          data.next = data_reg
58          ready.next = ready_reg
59          count_8_bits.next = count_8_bits_reg
60          count_stop_bits.next = count_stop_bits_reg
61
62          if state_reg == t_State.ST_WAIT_START_BIT:
63              ready.next = False
64              if baud_rate_tick_i == True:
65                  if recieve_i == False:
66                      state.next = t_State.ST_GET_DATA_BITS
67
68          elif state_reg == t_State.ST_GET_DATA_BITS:
69              if baud_rate_tick_i == True:
70                  data.next[count_8_bits_reg] = recieve_i
71                  if count_8_bits_reg == END_OF_BYTE:
72                      count_8_bits.next = 0
73                      state.next = t_State.ST_GET_STOP_BITS
74                  else:
75                      count_8_bits.next = count_8_bits_reg + 1
76                      state.next = t_State.ST_GET_DATA_BITS
```

```
77
78
79          elif state_reg == t_State.ST_GET_STOP_BITS:
80              if baud_rate_tick_i == True:
81                  if count_stop_bits_reg == (n_stop_bits_i - 1):
82                      count_stop_bits.next = 0
83                      ready.next = True
84                      state.next = t_State.ST_WAIT_START_BIT
85                  else:
86                      count_stop_bits.next = count_stop_bits_reg + 1
```



```
90
91
92      return outputs, sequential_process, combinational_process
```

Testbenches

Vamos testar da forma tradicional o nosso MyHDL, ou seja, criando um testbench e verificando a waveform.

Importe os módulos que acabamos de criar (**serial_tx.py**, **serial_rx,py** e **baudrate_gen.py**) no arquivo **tb_serial.py.**

```
1  from math import *
2  from myhdl import *
3
4  from serial_tx import serial_tx
5  from serial_rx import serial_rx
6  from baudrate_gen import baudrate_gen
7
8  def bench():
9
10     CLK_PERIOD = 20
11     clk_freq = 50000000
12     baud_const = int(floor(clk_freq / 115200))
13     clock = Signal(bool(0))
14     reset = ResetSignal(0, active=0, async=True)
15     start = Signal(False)
16     rx_rdy = Signal(False)
17     tx_data = Signal(intbv(0, min = 0, max = 256))
18     rx_data = Signal(intbv(0, min = 0, max = 256))
19     n_stop = 2
20     baudrate_tick = Signal(bool(0))
```

```
    half_baudrate_tick = Signal(bool(0))
    tx = Signal(bool(0))
    rx = Signal(bool(0))


    # design under test
    baud_gen_inst = baudrate_gen(clock, reset, baud_const, half_baudrate_tick, baudrate_tick)
    serial_tx_inst   = serial_tx(clock, reset, start, tx_data, n_stop, baudrate_tick, tx)
    serial_rx_inst   = serial_rx(clock, reset, n_stop, half_baudrate_tick, baudrate_tick, tx
```



```
        clock.next = not clock

    @instance
    def stimulus():
        tx_data.next = 196
        reset.next = 0
        for i in range(1):
            yield clock.negedge
        reset.next = 1
        for i in range(10):
            yield clock.negedge
        start.next = 1
        yield clock.negedge
        start.next = 0

    return baud_gen_inst, serial_tx_inst, serial_rx_inst, clockgen, stimulus


def test_bench():
    tb = traceSignals(bench)
    sim =  Simulation(tb)
    sim.run(1000000)

test_bench()
```

Definimos a nossa função bench responsável por parametrizar nosso projeto e também gerar os estímulos. Colocamos como constantes o periodo do clock (1 / 50000000), a frequência do clock (50000000), e calculamos a constante da nossa taxa de baudrate (número de pulsos para tick).

Então declaramos os tipos de dados dos nossos módulos, ao utilizar um ResetSignal para o reset você pode especificar se ele é síncrono ou assincrono e se é ativo em nível lógico alto ou baixo. Isto é necessário para o decorator @always_seq gerar o reset da forma correta. Também instanciamos os nossos módulos e fazemos as conexões, bem semelhante a forma feita em VHDL/Verilog.

Então temos dois decorators, um para gerar o clock e outro para gerar os estímulos do circuito. Como podemos observar, estamos colocando o dado 196 no serial_tx e enviando. Esperamos que o serial_rx nos dê este dado de volta ao emitir o sinal rx_rdy. A keyword yield é equivalente ao wait for em VHDL, ele espera que a condição seja verdadeira e então continua a execução.

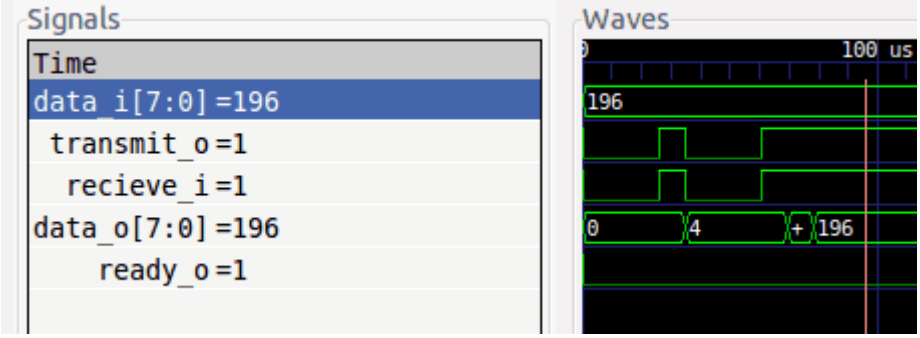
A função test_bench executa o bench em MyHDL. O traceSignals é responsável por gerar os nossos waveforms no formato .vcd, iremos rodar a simulação por 1 ms. Para isto basta executar na linha de comando:

>python tb_serial.py

O arquivo bench.vcd foi gerado, podemos abri-lo em qualquer visualizador de waveform. Para abrir no gtkwave (grátis e opensource) basta executar

>gtkwave bench.vcd

Vamos visualizar o nosso waveform:



E funcionou! O valor 196 foi transmitido via TX (transmit_o), recebido via RX (recieve_i) e quando o ready foi para nível lógico alto lá estava nosso dado de novo!

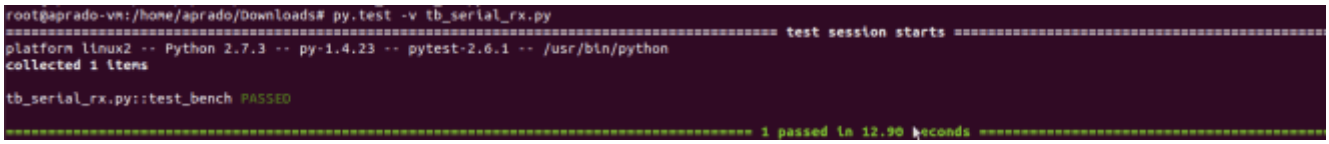
Teste automatizado

Uma vantagem do MyHDL é a possibilidade de automatizar o nosso teste e não verificar mais waveforms. Para isto vamos alterar o nosso tb_serial.py. Basta adicionar as duas linhas abaixo do start.next = 0:

```
1 yield rx_rdy.posedge
2 assert tx_data == rx_data
```

e rodar o py.test no terminal:

>py.test tb_serial.py



Pronto! Automaticamente o MyHDL verifica se o dado de envio e recebimento são os mesmos após a borda de subida do rx_rdy acontecer.

Gerando o VHDL / Verilog

MyHDL. Isto é muito simples, basta alterar estas linhas no testbench:

tb_serial.py:

```
# design under test
    #baud_gen_inst = baudrate_gen(clock, reset, baud_const, half_baudrate_tick, baudrate_tick)
    #serial_tx_inst   = serial_tx(clock, reset, start, tx_data, n_stop, baudrate_tick, tx)
    #serial_rx_inst   = serial_rx(clock, reset, n_stop, half_baudrate_tick, baudrate_tick, tx

    # generate VHDL
    baud_gen_inst = toVHDL(baudrate_gen, clock, reset, baud_const, half_baudrate_tick, baudrat
    serial_tx_inst   = toVHDL(serial_tx, clock, reset, start, tx_data, n_stop, baudrate_tick,
    serial_rx_inst   = toVHDL(serial_rx, clock, reset, n_stop, half_baudrate_tick, baudrate_ti
```

E mais estas:

```
    tb = bench()
    #tb = traceSignals(bench)
    sim =  Simulation(tb)
    sim.run(1000000)
```

Agora, além de testar o seu design, os arquivos VHDL também já são gerados automaticamente. A função toVerilog geraria Verilog ao invés de VHDL.

O VHDL/Verilog gerado é completamente legível e bem estruturado, na verdade é basicamente uma tradução do seu RTL em Python. Um RTL ruim em Python gera um

arquivo VHDL/Verilog ruim. Como exemplo o nosso serial_tx.vhd gerado:

```
1  -- File: serial_tx.vhd
2  -- Generated by MyHDL 0.8
3  -- Date: Tue Aug 19 14:17:48 2014
```



```
7  use ieee.std_logic_1164.all;
8  use IEEE.numeric_std.all;
9  use std.textio.all;
10
11 use work.pck_myhdl_08.all;
12
13 entity serial_tx is
14     port (
15         sysclk: in std_logic;
16         reset_n: in std_logic;
17         start_i: in std_logic;
18         data_i: in unsigned(7 downto 0);
19         baud_rate_tick_i: in std_logic;
20         transmit_o: out std_logic
21     );
22 end entity serial_tx;
23 -- Serial
24 -- This module implements a transmiter serial interface
25 --
26 -- Ports:
27 -- -----
28 -- sysclk: sysclk input
29 -- reset_n: reset input
30 -- baud_rate_tick_i: the baud rate
31 -- start_i: start sending data
32 -- data_i: the data to send
33 -- transmit_o: data output
34 -- -----
35
36 architecture MyHDL of serial_tx is
37
38
39 constant n_stop_bits_i: integer := 2;
40 constant END_OF_BYTE: integer := 7;
41
42
43 type t_enum_t_State_1 is (
44     ST_WAIT_START,
45     ST_SEND_START_BIT,
46     ST_SEND_DATA,
47     ST_SEND_STOP_BIT
48 );
49
50 signal transmit_reg: std_logic;
51 signal count_8_bits: unsigned(2 downto 0);
52 signal count_8_bits_reg: unsigned(2 downto 0);
53 signal state: t_enum_t_State_1;
54 signal transmit: std_logic;
55 signal count_stop_bits_reg: unsigned(2 downto 0);
56 signal count_stop_bits: unsigned(2 downto 0);
57 signal state_reg: t_enum_t_State_1;
58
59 begin
60
61
62
```

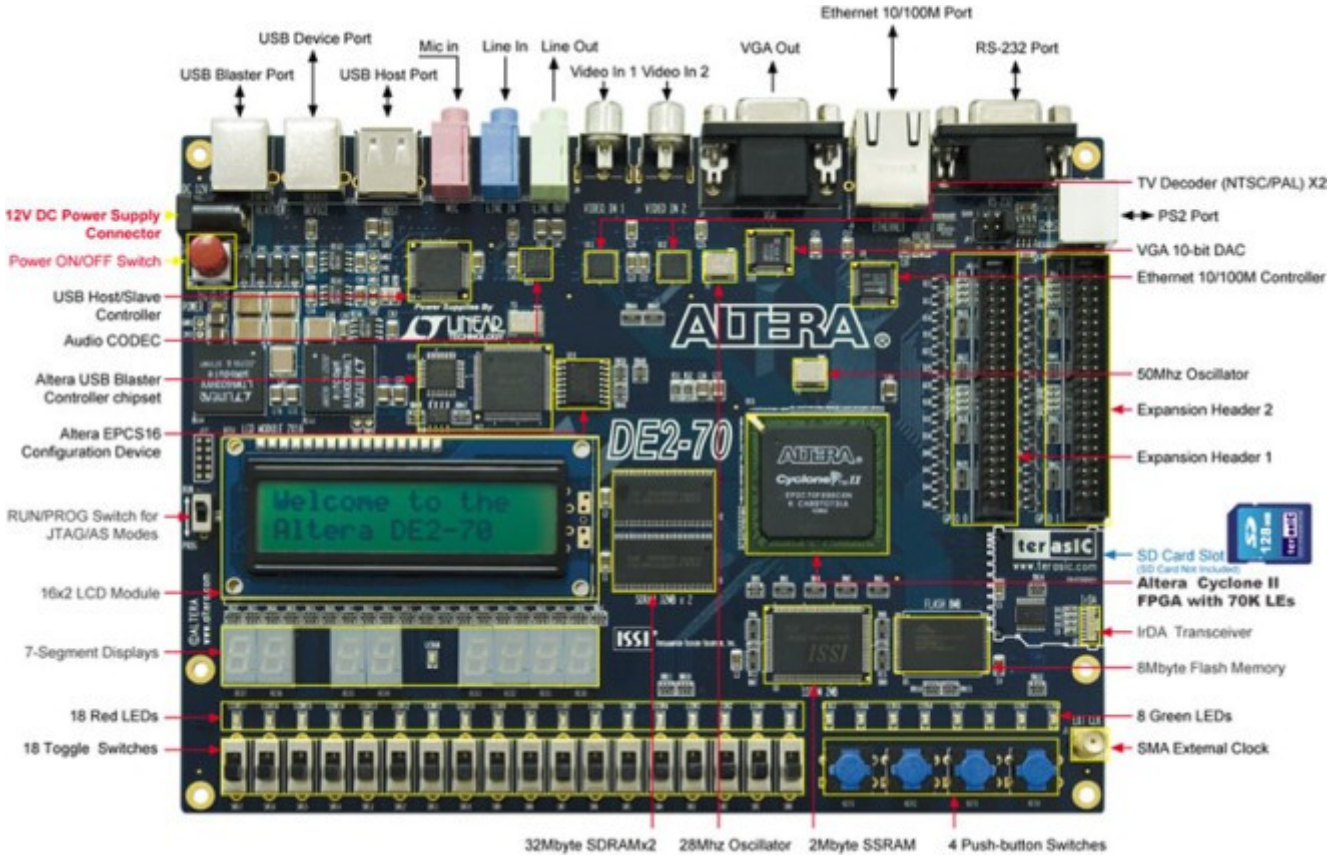
```
63
64
65  transmit_o <= transmit_reg;
66
67
68  SERIAL_TX_SEQUENTIAL_PROCESS: process (sysclk, reset_n) is
69  begin
70      if (reset_n = '0') then
71          count_8_bits_reg <= to_unsigned(0, 3);
72          count_stop_bits_reg <= to_unsigned(0, 3);
```



```
76          state_reg <= state;
77          transmit_reg <= transmit;
78          count_8_bits_reg <= count_8_bits;
79          count_stop_bits_reg <= count_stop_bits;
80      end if;
81  end process SERIAL_TX_SEQUENTIAL_PROCESS;
82
83
84  SERIAL_TX_COMBINATIONAL_PROCESS: process (transmit_reg, start_i, count_8_bits_reg, data_i, b
85  begin
86      state <= state_reg;
87      transmit <= transmit_reg;
88      count_8_bits <= count_8_bits_reg;
89      count_stop_bits <= count_stop_bits_reg;
90      case state_reg is
91          when ST_WAIT_START =>
92              transmit <= '1';
93              if (start_i = '1') then
94                  state <= ST_SEND_START_BIT;
95              end if;
96          when ST_SEND_START_BIT =>
97              transmit <= '0';
98              if (baud_rate_tick_i = '1') then
99                  state <= ST_SEND_DATA;
100             end if;
101         when ST_SEND_DATA =>
102             transmit <= data_i(to_integer(count_8_bits_reg));
103             if (baud_rate_tick_i = '1') then
104                 if (count_8_bits_reg = END_OF_BYTE) then
105                     count_8_bits <= to_unsigned(0, 3);
106                     state <= ST_SEND_STOP_BIT;
107                 else
108                     count_8_bits <= (count_8_bits_reg + 1);
109                     state <= ST_SEND_DATA;
110                 end if;
111             end if;
112         when ST_SEND_STOP_BIT =>
113             transmit <= '1';
114             if (baud_rate_tick_i = '1') then
115                 if (signed(resize(count_stop_bits_reg, 4)) = (n_stop_bits_i - 1)) then
116                     count_stop_bits <= to_unsigned(0, 3);
117                     state <= ST_WAIT_START;
118                 else
119                     count_stop_bits <= (count_stop_bits_reg + 1);
120                 end if;
121             end if;
122         when others =>
123             assert False report "End of Simulation" severity Failure;
124     end case;
125 end process SERIAL_TX_COMBINATIONAL_PROCESS;
126
127 end architecture MyHDL;
```

Testando na placa

na porta RS-232.



Kit DE2

Usei como base o GOLD REFERENCE DESIGN, que já possui todas as portas declaradas e os pinos do FPGA identificados.

Instanciei meus três componentes, conectei a condição de START do Serial_TX ao botão KEY[0], conectei o RX vindo direto do RS232 no meu recieve_i e o TX direto do cabo no meu transmit_o.

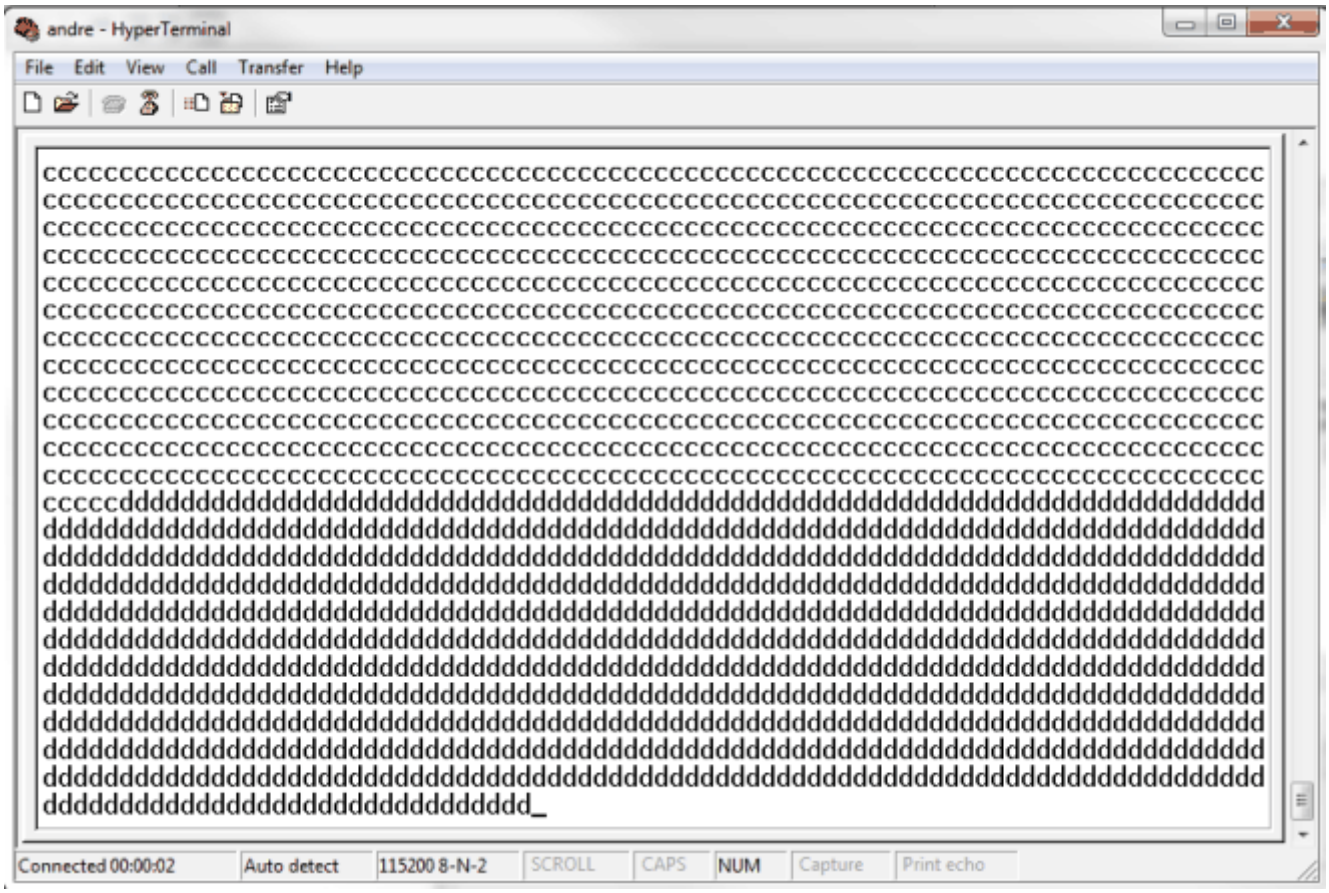
Também conectei como loopback o dado a enviar no TX e o dado recebido no RX. A conexão está exatamente igual à indicada na arquitetura lá no começo do arquivo.


```
wire [7:0] data_from_rx;

serial_tx serial_tx_inst
    (
        .sysclk(CLOCK_50),
        .reset_n(1'b1),
        .start_i(KEY[0]),
        .data_i(data_from_rx),
        .baud_rate_tick_i(baud),
        .transmit_o(UART_TXD)
    );

baudrate_gen baudrate_gen_inst
   (
        .sysclk(CLOCK_50),
        .reset_n(1'b1),
        .baud_rate_tick_o(baud)
    );

serial_rx serial_rx_inst
    (
        .sysclk(CLOCK_50),
        .reset_n(1'b1),
        .half_baud_rate_tick_i(baud),
        .baud_rate_tick_i(baud),
        .recieve_i(UART_RXD),
        .data_o(data_from_rx)
    );
```

E não é que funciona? Basta conectarmos o cabo no FPGA e no computador e configurar o seu software de preferência que se comunique via serial.

Eu utilizei o Hyper Terminal, configurei a taxa de 115200 Hz, 8 bits de dados, sem paridade e 2 bits de stop (115200-8-n-2) e voilá... Tudo que você digita da Eco enquanto o botão estiver pressionado.

(**Salve seu trabalho antes!!! Como esta é uma serial simplificada, não foram observados sinais de controle como CLEAR TO SEND e READY TO SEND. Isto me ocasionou uma tela azul no Windows 7 durante os testes**).



Loopback na serial

Conclusão

Com este projeto podemos observar que o MyHDL funciona e muito bem, gerando um bom VHDL e funcionando na placa. O ciclo de desenvolvimento fica muito mais rápido em Python e você possui ferramentas a mais para testar seu módulo antes de gerar o