VHDL - Practical Example - Designing an UART

Bertrand CUZEAU
Technical Manager - ALSE
ASIC / FPGA Design Expert
Doulos HDL Instructor (Verilog-VHDL)
info@ALSE-FR.COM
http://www.alse-fr.com
☎ : 33.(0)1 45 82 64 01
Introduction

We will demonstrate, on a “real-life” example, how a sound HDL methodology can be used in conjunction with modern synthesis and simulation tools.

Note : the source code we provide here if for teaching purpose only.  
This code belongs to ALSE.  
If you want to use it in your projects please contact us.
UART Specification

We want to address the following needs:

- Transmit / Receive with h/w handshake
- “N81” Format, but plan for parity
- Speed: 1200..115200 baud (Clock = 14.7456 MHz)
- No internal Fifo (usually not needed in an FPGA !)
- Limited frame timing checks
Methodology

We adopt the following constraints:

- Standard & 100% portable VHDL Description:
  - Synthesis
  - Simulation
  - Target FPGA (or CPLD)

- Complete functional Simulation with file I/O.

- Should work “in vivo” on an existing ALSE demo board.
Application Architecture

**RS232 Inputs**

- **RXFLEX**: RawRx
- **CLK**: C
- **RST**: 1306

**External Baud Rate Selection**

- DIPSW[2]: Baud[2]
- DIPSW[1]: Baud[1]
- DIPSW[0]: Baud[0]

**RS 232 Output**

- **RTSFLEX**: RawRTS
- **RXFLEX**: Noted on PCB = CTSFlex

**Inversion needed**

© Bertrand CUZEAU - info@alse-fr.com
Baud Rate Generator

• Embedded in UARTS.

• Divides by 8, 16, 28, 48, 96, 192, 384 or 768 and builds Top16.

• Generates two “ticks” by further dividing Top16:
  - Transmit : TopTx, fixed rate
  - Receive : TopRx, mid-bit, resynchronized
-- Baud rate selection

process (RST, CLK)
begin
  if RST='1' then
    Divisor <= 0;
  elsif rising_edge(CLK) then
    case Baud is
      when "000" => Divisor <= 7; -- 115.200
      when "001" => Divisor <= 15; -- 57.600
      when "010" => Divisor <= 23; -- 38.400
      when "011" => Divisor <= 47; -- 19.200
      when "100" => Divisor <= 95; --  9.600
      when "101" => Divisor <=191; --  4.800
      when "110" => Divisor <=383; --  2.400
      when "111" => Divisor <=767; --  1.200
      when others => Divisor <= 7; -- n.u.
    end case;
  end if;
end process;

-- Clk16 Clock Generation

process (RST, CLK)
begin
  if RST='1' then
    Top16 <= '0';
    ClK16 <= 0;
  elsif rising_edge(CLK) then
    Top16 <= '0';
    if ClK16 = Divisor then
      ClK16 <= 0;
      Top16 <= '1';
    else
      ClK16 <= ClK16 + 1;
      end if;
    end if;
  end process;

-- Rx Sampling Clock Generation

process (RST, CLK)
begin
  if RST='1' then
    TopRx <= '0';
    RxDi <= 0;
  elsif rising_edge(CLK) then
    TopRx <= '0';
    if ClRDi <= 1 then
      RxDi <= 0;
      TopRx <= '1';
    else
      RxDi <= RxDi + 1;
      end if;
    end if;
end process;

© Bertrand CUZEAU - info@alse-fr.com
Transmitter

We use a very simple State Machine to control the transmit shift register. The FSM inputs are:

- **LD**: Loads the character to transmit ($D_i$ $n$)
- **TopTx**: Bit shifting command

For simplicity we code the FSM as a “re-synchronized Mealy”.
--- Transmit State Machine

TX <= Tx_Reg(0);

Tx_FSM process (RST, CLK)
begin
if RST='1' then
    Tx    <= (others => '1');
    TxBitCnt <= 0;
    TxFSM <= idle;
    TxBusy <= '0';
    RegDin <= (others=>'0');
elsif rising_edge(CLK) then
    TxBusy <= '1';  -- except when explicitly '0'
    case TxFSM is
    when idle =>
        if LD='1' then
            -- latch the input data immediately.
            RegDin <= Din;
            TxBusy <= '1';
            TxFSM <= Load_Tx;
        else
            TxBusy <= '0';
        end if;
    when Load_Tx =>
        if TopTx='1' then
            TxFSM <= Shift_Tx;
        end if;
    when Shift_Tx =>
        if TopTx='1' then
            TxBitCnt <= TxBitCnt - 1;
            if TxBitCnt=0 then
                TxFSM <= Stop_Tx;
            end if;
    when Stop_Tx =>
        if TopTx='1' then
            TxFSM <= idle;
        end if;
    when others =>
        TxFSM <= idle;
    end case;
end if;
end process;
We also use a State Machine:

- Wait RX (Start bit) falling edge,
- Synchronize the Half-bit counter
- Sample RX at mid-bit and verify the Start bit
- Loop on the data bits (+ parity):
  * Skip transition
  * Sample at mid-bit
- Sample and Test Stop bit
- Return to Idle state (waiting for a new Start condition)
-- RECEIVE State Machine

Rx_FSM process (RST, CLK)
begins

if RST='1' then
  Rx_Reg <= (others => '0');
  Dout <= (others => '0');
  RxBitCnt <= 0;
  RxFSM <= Idle;
  RxRdyi <= '0';
  ClrDiv <= '0';
  RxErr <= '0';
else if rising_edge(CLK) then
  ClrDiv <= '0'; -- default value

-- reset error when a word has been received Ok:
if RxRdyi = '1' then
  RxErr <= '0';
  RxRdyi <= '0';
end if;

end process;

case RxFSM is

when idle => -- wait on start bit
  RxBitCnt <= 0;
  if TopRx='1' then
    if Rx='0' then
      RxFSM <= Start_Rx;
      ClrDiv <= 1'; -- Synchronize the divisor
    else false start, stay in idle
    end if;
  end if;

when Start_Rx => -- wait on first data bit
  if TopRx = '1' then
    if Rx='1' then -- framing error
      RxFSM <= RxOVF;
      report "Start bit error." severity note;
    else
      RxFSM <= Edge_Rx;
    end if;
  else
    RxFSM <= Edge_Rx;
  end if;

when Shift_Rx => -- Sample data !
  if TopRx = '1' then
    if RxBitCnt = NDbits then
      RxFSM <= Stop_Rx;
    else
      RxFSM <= Shift_Rx;
    end if;
  end if;

when Stop_Rx => -- during Stop bit
  if TopRx = '1' then
    Dout <= Rx_reg;
    RxRdyi <= '1';
    RxFSM <= Idle;
    assert (debug < 1)
    report "Character received in decimal is : 
      & integer'image(to_integer(unsigned(Rx_Reg)))" severity note;
  end if;

when RxOVF => -- Overflow / Error
  RxErr <= '1';
  if Rx='1' then
    RxFSM <= Idle;
  end if;

end case;

end if;
Receiver State Machine
To test our UART, we use a trivial “application” which increments the characters received and resends them!

(Example: “A” → “B”, “f” → “g”, “HAL” → ”IBM”…)

This way, it is easy to verify the receive and transmit operations, both by simulation and on the demo board.
begin
if RST='1' then
  LD_ <= '0';
  SData <= (others=>'0');
  RData <= (others=>'0');
elsif rising_edge(CLK) then
  LD_ <= '0';
  case State is
  when Idle =>
    if RxRDY='1' then
      RxErr : In  std_logic;
      RData : std_logic_vector (7 downto 0);
      TxBusy : In  std_logic;
    end if;
    if (TxBusy='0') and (RTS='1') then
      case to_integer(unsigned(RData)) is
        when 10|13|32 => null;
      end case;
      SData <= RData;
      when others =>
        SData <= std_logic_vector(unsigned(RData)+1);
      end case;
    end if;
    State <= Send;
  when Send =>
    LD_SDout <= '1';
    SData <= RData;
    State <= Idle;
  end case;
end if;
end process;
end RTL;
The VHDL Test Bench simply sends the ASCII character ‘A’ and displays the character(s) sent back by the system.

It is based on two behavioral UART routines (described in another of our conferences).

A much more sophisticated Test Bench (with file I/O and console emulation with inter-character spacing) is provided by ALSE in the commercial version.
-- Simple VHDL test bench for UART Top_Level
-- (c) ALSE - Bertrand Cuzeau
-- info@alse-fr.com

USE std.textio.all;
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
USE ieee.numeric_std.ALL;
USE ieee.std_logic_textio.ALL;

ARCHITECTURE TEST OF testbench IS
  COMPONENT ALSE_UART
    PORT (     RST : IN  std_logic;
               CLK : IN  std_logic;
               RXFLEX : IN  std_logic;
               RTSFLEX : IN  std_logic;
               DI PSW : IN  std_logic_vector (2 downto 0);
               CTSFLEX : OUT std_logic;
               TXout : OUT std_logic );
  end component;

constant period : TIME := 68 ns;
constant BitPeriod : TIME := 8680 ns; -- 115.200

for i in 0 TO 7 LOOP
  RXFLEX <= RSData(i); wait for BITPeriod;
end LOOP;

signal CTSFLEX : STD_LOGIC;
assert CTSFLEX='1' REPORT "Stop bit Error ???" SEVERITY WARNING;

begin
  UUT : ALSE_UART
    WRITELINE (output,L);       -- char -> transcript
  end process;
end testbench;
Let’s make it work!

After the theory, we are now going to follow the entire design flow, down to the Demo Board (~10 minutes):

1. Build the Project
2. Syntactic Verification
3. Unitary Functional Simulation
4. Unitary Logic Synthesis
5. System-Level Simulation
6. Global Synthesis
7. Place and Route
8. Download & tests on the demo board (using HyperTerminal !)
Conclusion

It takes *less than a working day* to design and test a simple UART like this one. Powerful HDL languages, as well as capable Simulation and Synthesis Tools are now widely available.

With the right methodology and some design practice, projects that used to be considered as “complex” become almost trivial.

Note: an enhanced version of this UART, still simple and efficient, is available at ALSE, at a very affordable cost.