Author : Bert Cuzeau – ALSE
http://www.alse-fr.com

Simple UART Intellectual Property - VHDL

Introduction

A few years ago, we have developed and used a simple yet efficient UART module (both in VHDL and Verilog, though this document only describes the VHDL version). Designing this core of the UART was a rather simple task of no more than a few hours, and it performed very well where we needed it. For simulation purpose, we did also develop a behavioral (i.e. not synthesizable) model.
In some conferences (which you may find at http://www.alse-fr.com), we used this project as an example to demonstrate how HDL Design techniques (and especially Finite State Machines) can solve apparently complex problems with a minimal effort.

Many years after, we are still asked to provide the complete source code. Apparently, despite the simplicity of the project and the availability of several free models, there is still a demand for it.

Therefore, we took the decision to clean it a bit, add a few features, comment the code, enhance the test bench, create a working example, and make it available. This is what you have in your hands now. We designed it also as a design and methodology example. Beginners will probably learn a lot by understanding any detail of the RTL, the simulation, and the behavioral pieces of code.
Experts have undoubtedly stopped reading this document and have jumped to the code ☺.

Main Features

- Asynchronous RS232 character-based transmit / receive function
- Very compact (around 50 CLB slices in the Spartan II family, or around 100 LCs / atoms in Altera Acex1k / Apex20k technologies). This includes the baud rates generator…
- Good timing performance. Except in very trivial designs, no part of the UART should appear in the critical paths.
- No internal Fifo. The user application is supposed to handle the communication latencies and to consume received characters faster than they are received (very general case). If needed, the characters could be pipelined in an external Fifo.
- Internal Baud rate generator with two baud rates selectable by user input signal.
- Synthesizable to any technology (FPGA or Asic).
- Code tested on most Synthesis tools available today. Script files provided for Leonardo Spectrum, Synplify, Quartus and ISE.
- Demo application to test the UART on most of the existing FPGA demo boards.
- Simulation test bench including reusable behavioral UART and RS232 Terminal (console) simulation with File I/O and results logged in the transcript. A complete simulation script is also provided.

Note that this core is not designed as a computer peripheral (or SoC equivalent). It performs best in a hardware design context, where no software (i.e. no CPU) is involved.

Archive Contents

The archive is organized as follows:

Directory	Contents
Doc	PDF documentation (you're reading it now)
Src	RTL (synthesizable) VHDL Source files
Simu	Test bench, behavioral model, Simulation scripts & I/O file
Fit	Place and Route (FPGA project) base directory.
Fit / Ise	Ready-to-use Xilinx-ISE project for Insight Spartan II board.
Fit / Tornado	Ready-to-use scripts for Tornado board.
Fit / Nios1C20	Use this if your target is an Altera NIOS 1C20 board

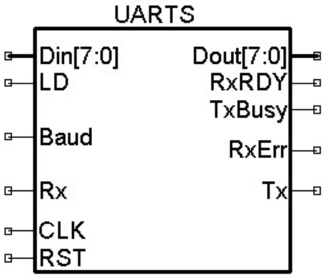
Note that each subdirectory has a small readme.txt file.

Where to go from here ?

Indeed, different users will handle this project in many different ways, so we consider that if you're reading this chapter, you are probably expecting a little help. Once the archive has been unpacked in an empty directory (with no space in the path name), you might consider the following steps:

1. Inspect **UARTS.vhd**. This is the RTL code that likely brought you here. Make sure you understand completely how it works. The way this UART is implemented (Finite State Machines) is a very powerful methodology, which is applicable to most designs of all kinds.
   You'll notice that a reasonable use of generic parameters, functions, constants, etc… can help you create a very reusable yet efficient RTL code.
2. Examine **APPLIC.vhd**. This one is really simple. Its task is to establish a dialog with UARTS to (modify and) echo the characters received. No surprise: we used an FSM here too.
3. Look at **TOP_UART.vhd**. Do not modify it.
4. Open **CONSOLE.vhd**. This is a simulation-only file that reads in a text file and creates an RS232 stream from it. The text file contains simulation commands (delay = '#') and comments.
   "Console" includes a behavioral RS232 transmitter, which can be reused in other situations where it is necessary to send RS232 characters, strings, frames, commands, text files…
5. Open **TB_TOP.vhd**. It includes a behavioral RS232 receiver with transcript logging, which can be reused in other projects.
6. **Simulate the whole design**. You may examine the proposed signals in the waveform to better understand how UARTS works. Don't forget to look at the transcript…
7. **Adapt TOP_UART.vhd** to your hardware (FPGA board) :
   * modify the Generics default value (**entity** section) for the System clock frequency.
   * modify the **UARTS Generic map** section (Baud1 & Baud2).
   * modify the **Reset polarity** if necessary.
   * you may force DIPSW to a fixed level if you don't want to keep two different baud rates,
   * etc…
8. **Synthesize**, do the **pin assignment**, **place & route** your project.
9. **Download** the project, connect a PC and test the communication.
   Don't hesitate to use an oscilloscope to verify the main signals and their polarities…

UARTS



This is the UART Synthesizable (RTL) module, “***the IP***” and we will describe its parameters, inputs and outputs. In a following section, we will describe a typical application using this module.

Note that we did our best to keep this module as simple as possible.

Generics:

Fxtal	System clock Freq in Hertz.
Parity	Boolean true if you want a parity bit added to the sent and received chars.
Even	Boolean used only when Parity is true, to indicate an Even parity (else Odd).
Baud1	Baud Rate used when Baud input = 1
Baud2	Baud Rate used when Baud input = 0

Inputs / Outputs:

CLK	System Clock, frequency must be declared in the Generic map.
RST	Asynchronous active-high reset.
Baud	Toggles between Baud1 & Baud2 rates.
Din[7:0]	Eight-bits Data : raw character to be sent.
LD	Load pulse (one clock cycle) to load Din in the Transmitter and debut the transmission immediately. The application is supposed to verify beforehand that TxBusy is false and that the external device is ready to receive the character (flow control / RTS).
Rx	RS232 receive signal input. Is ’1’ when the line is idle.
Tx	RS232 transmit signal output. Is ’1’ when the line is idle.
TxBusy	A ‘1’ indicates that the UART is busy sending the previous character and will ignore a LD request (the character would be lost).
Dout[7:0]	Data received. Read this value when the RxRDY pin is ‘1’
RxRDY	A ‘1’ pulse (one system clock cycle long) indicates that a character is received and is available at Dout.
RxErr	A ‘1’ here indicates that a character was received incorrectly. In this case, no RxRDY would occur.

Note that the description is smart enough to size automatically the baud rate generator’s internal counters. However, the user must verify that the combination of system clock frequency and Baud Rate selected are compatible in terms of timing inaccuracies.

Example: Fxtal = 10 MHz, Baud=115200. Bit period = 8.68 µs
The **Half bit period** generation requires a division by : 43.4 which is rounded to 43.
The equivalent Baud rate would then be 1e7 / 86 = ~116.300, i.e. an error of ~ 1%, which is in this case acceptable. **Beware**: ***this error does accumulate* !**
A 1 % error leads to ~20 % error at the last (stop) bit…

Example Design

To demonstrate the use of the UARTS module, we have built a small application that reads in characters from the RS232 line, and echoes them after modification. When a question mark is sent, the system replies by a prompt line saying “*Scrambler Demo.*”.

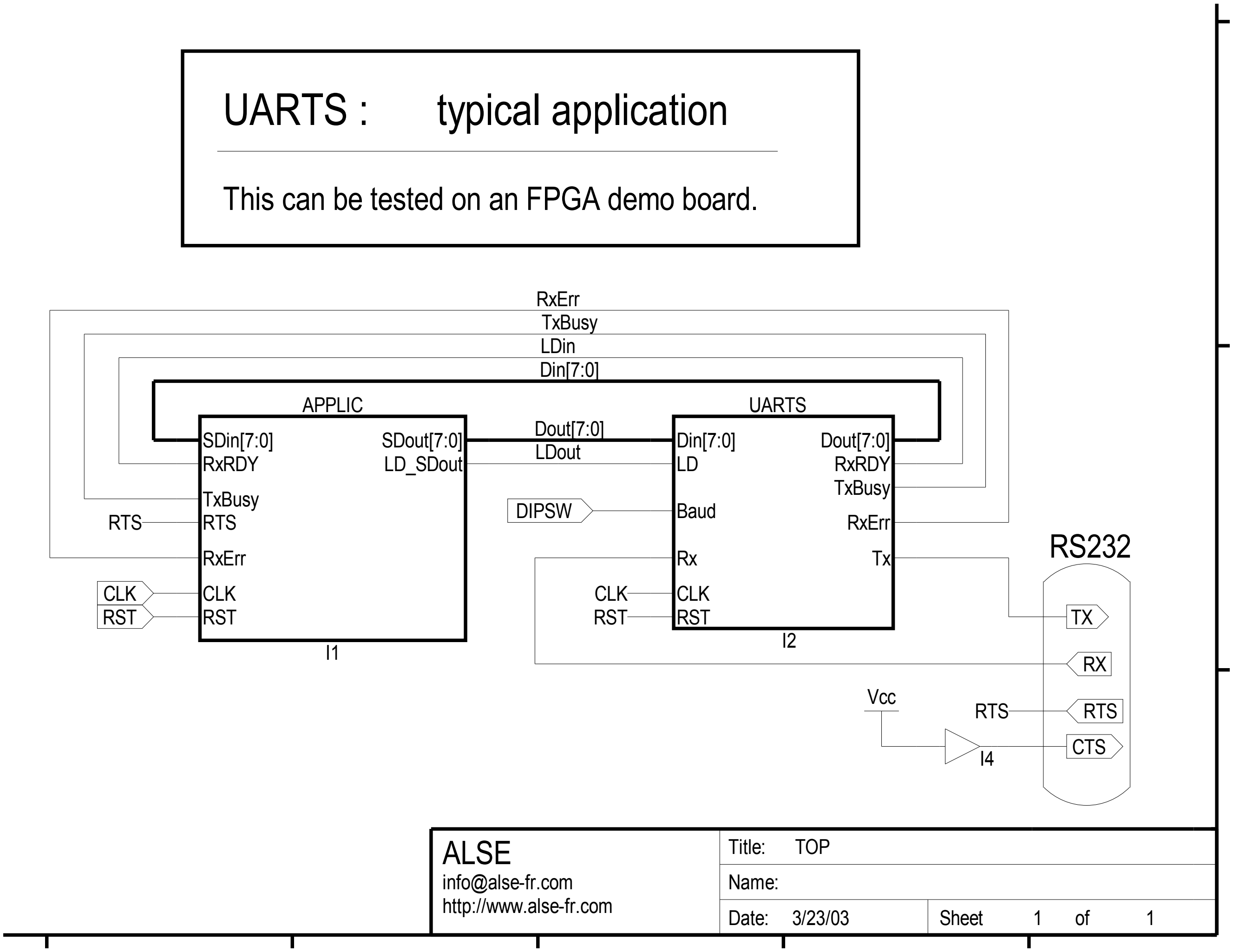
This design can be completely simulated. It can also be fitted on a real FPGA board to test the implementation, the RS232 connection, etc...
A PC running HyperTerminal can be then used to exercise the design manually.

The module name “APPLIC” does handle the UART module completely :

- Waits until a character is received by UARTS.
- If this character is Cr, Lf, or space, it is echoed intact.
- If the character is a digit (‘0’ .. ‘9’), its value is memorized and a dot ‘.’ is echoed
- Otherwise, the Ascii code is incremented by the memorized value and the new character is echoed.

Note: if for some reason the outside system is busy, the character to be echoed may be lost.

The example design’s top level is *roughly* equivalent to the following schematics:



APPLIC Code

Let us review the architecture code for APPLIC:

```
-- ----------------------------------------------------
    Architecture RTL of APPLIC is
-- ----------------------------------------------------
function Char2SLV8 (c : character) return std_logic_vector is
begin
   return std_logic_vector(to_unsigned(character'pos(c),8));
end function;

  constant Prompt : string := "Scrambler Demo." & Cr & Lf;

  type State_Type is  (Boot1, Boot2, Boot3, Idle, Scramble, Send);
  signal State : State_Type;

  signal RData : std_logic_vector (7 downto 0);
  signal SData : std_logic_vector (7 downto 0);

  signal Code  : unsigned(3 downto 0);
  signal RTS_r : std_logic;             -- resync FLipFlop on RTS
  signal CharCount : natural range Prompt'low to Prompt'high+1;
  signal CharAvail : boolean;

begin

SDout <= SData;

process (CLK,RST)
begin
  if RST='1' then
    State <= Boot1;
    LD_SDout <= '0';
    SData <= (others=>'0');
    RData <= (others=>'0');
    Code <= (others=>'0');
    RTS_r <= '0';
    CharCount <= Prompt'low;
    CharAvail <= false;

  elsif rising_edge(CLK) then

    -- memorize any arriving character
    if RxRDY='1' then  -- memorize the incoming char
      RData <= SDin;
      CharAvail <= true;
    end if;

    RTS_r <= RTS;
    LD_SDout <= '0';

    case State is

      when Boot1 =>
        CharCount <= Prompt'low;
        State <= Boot2;

      when Boot2 =>
        if (TxBusy='0') and (RTS_r='1') then
          SData <= Char2SLV8 (Prompt(CharCount));
          LD_SDout <= '1';
          CharCount <= CharCount+1;
          if CharCount=Prompt'high then
            State <= Idle;
          else
            State <= Boot3;
          end if;
        end if;

      when Boot3 => -- we need one clock cycle before testing Busy
        State <= Boot2;

      when Idle =>
        if CharAvail then
          CharAvail <= false;
          State <= Scramble;
        end if;

      when Scramble =>
        if (TxBusy='0') and (RTS_r='1') then
          State <= Send;
          case to_integer(unsigned(RData)) is
            when 10|13|32 =>   -- do not translate Cr Lf Space !!!
              SData <= RData;
            when 48 to 57 =>   -- change the offset & returns a dot
              Code <= unsigned(RData(3 downto 0));  -- x"30"..x"3F"
              SData <= Char2SLV8('.');
            when 63 => -- '?' -> Prompt
              State <= Boot1; -- override Send
            when others =>     -- add the current offset
              SData <= std_logic_vector(unsigned(RData) + Code);
          end case;
        end if;

      when Send =>
        LD_SDout <= '1';
        State <= Idle;

      -- if encoding is safe + binary and the synthesis tool accepts it...
      when others =>
        State <= Idle;

    end case;

  end if;
end process;

end RTL;
```

We designed a **Resynchronized Mealy Finite State Machine**, a style we like since it's easy to write, understand and maintain, and less error-prone than combinational process-based styles.

Note the function Char2SLV8, which converts a character into an Std_logic_vector (7 downto 0).

Important: the duration of the **RxRdy** signal from UARTS is **only one clock cycle !**
It is your (application's) responsibility to capture this event (and the associated Data word).

Typically a polled peripheral (as in a CPU environment) has to provide a handshake mechanism: a FlipFlop is synchronously set by RxRdy and cleared when the Data is used. This is precisely what **CharAvail** does implement ! You *must* provide a similar feature in your own application.

The Scramble + Send states represent the input character processing (translated and echoed back, basically). Note that the "?" character triggers the sending of the prompt line.

You might find handy to reuse APPLIC as a skeleton for your own processing.

A possible application could be an Intel-Hex format reader, which would fill a memory with the binary contents, and issue an Ack / Nack response to the sender if the line checksum is correct or not.
We have already done this for a couple of designs. Contact us if you have a similar need.
Actually, writing the VHDL code for the Intel Hex reader is as simple as a software solution, and it executes a lot faster !

Functional Simulation

The **./Simu** sub-directory includes a script named “**simu.do**” which does create the simulation library, compile all the source files, calls another script to bring interesting signals in the waveform window, and runs the simulation.

The test bench does analyze the RS232 line, and logs the characters echoed in the simulation transcript. Therefore, examining the waveform window is only useful to understand how the system works internally. The characters echoed are printed in the transcript under the form:

```
# === Simulation starting now ===
#
# Character received (hex) = 53 - 'S'
# Character received (hex) = 63 - 'c'
# Character received (hex) = 72 - 'r'
# Character received (hex) = 61 - 'a'
# Character received (hex) = 6D - 'm'
# Character received (hex) = 62 - 'b'
# Character received (hex) = 6C - 'l'
# Character received (hex) = 65 - 'e'
# Character received (hex) = 72 - 'r'
# Character received (hex) = 20 - ' '
# Character received (hex) = 44 - 'D'
# Character received (hex) = 65 - 'e'
# Character received (hex) = 6D - 'm'
# Character received (hex) = 6F - 'o'
# Character received (hex) = 2E - '.'
# Character received (hex) = 0D - '
# '
# Character received (hex) = 0A - '
# '
# Character received (hex) = 2E - '.'
# Character received (hex) = 48 - 'H'
# Character received (hex) = 2E - '.'
# Character received (hex) = 65 - 'e'
# Character received (hex) = 2E - '.'
# Character received (hex) = 6C - 'l'
# Character received (hex) = 2E - '.'
# Character received (hex) = 6C - 'l'
# Character received (hex) = 2E - '.'
# Character received (hex) = 6F - 'o'
# Character received (hex) = 20 - ' '
# Character received (hex) = 2E - '.'
# Character received (hex) = 57 - 'W'
# Character received (hex) = 2E - '.'
# Character received (hex) = 6F - 'o'
# Character received (hex) = 2E - '.'
# Character received (hex) = 72 - 'r'
# Character received (hex) = 2E - '.'
# Character received (hex) = 6C - 'l'
# Character received (hex) = 2E - '.'
# Character received (hex) = 64 - 'd'
# Character received (hex) = 53 - 'S'
# Character received (hex) = 63 - 'c'
# Character received (hex) = 72 - 'r'
# Character received (hex) = 61 - 'a'
# Character received (hex) = 6D - 'm'
# Character received (hex) = 62 - 'b'
# Character received (hex) = 6C - 'l'
# Character received (hex) = 65 - 'e'
# Character received (hex) = 72 - 'r'
etc..
# ** Failure: End of Simulation (not a failure).
```

For in-depth analysis and debug, you may modify the debug constant located in the test bench file to increase the level of verbosity.

Here is the ASCII script file, named **console.txt,** which generated the transcript above:

```
; -- Console.txt : RSR 232 Input File ---
; Characters are sent as such, except for :
;  - semicolumns ';' which start a comment ending
;    at the end of the line
;  - '#' followed by a delay expressed in micro-seconds.
; Note that Cr & LF are not sent.
;
#2000
0H1d2j3i4k
#80
 5R6i7k8d9[
#200
?
#2000
```

The initial 2 ms delay leaves room for the Prompt line (which is sent after the system reset).

The question mark does also print the prompt, hence the second 2 ms delay.

When the end of the text file is reached, the simulation stops after a small delay.

Synthesis

Before this step, you need first to set up the top level to match your actual hardware & configuration. Open the VHDL file **TOP_UART.vhd** and set the following parameters in the entity generics declarations :

- System Clock Frequency as default value of **Fxtal**,
- Reset active level as default value of **Reset_active**,
- Word format with **Parity** yes/no + **Even**/Odd,

In the generic map declaration of the UARTS instanciation, adjust :
- **Baud1** and **Baud2** parameters to the desired values.

The synthesis project consists of three VHDL files :
UARTS.vhd + **APPLIC.vhd** + **TOP_UART.vhd**, the latter being the top level.

If you use **ISE**, you'll find a ready-to-use project file in ./Fit/Ise which can be modified easily to match any Xilinx board.

If you use **Quartus**, you'll find Tcl scripts for the NIOS 1C20 board and Tornado board.

All the scripts we provide are relocatable and should work wherever you decide to unpack the archive. However, it is strongly recommended to **avoid path names including space character(s)** (PC).

Testing

Prepare the board by attaching a serial cable to the PC's serial line connector (or to an RS232 terminal).

If you use a PC, run Hyperterminal (for example) and adjust the communication parameters to the project's parameters. We suggest 115200 / N / 8 / 1 for Tornado, or 19200 / N / 8 / 1 for an Insight board.

Make sure you set DIPswitch in accordance with the above configuration.

Once the FPGA has been placed and routed as described above, you should be able to download the bitstream into the FPGA. Once this is done, the will LED blink whenever a character is received from the terminal. If this is not the case, verify that the cable is of the correct type and that the communication software is tuned to the proper COM port.

Type “ABC1DEF”. The echo should be “ABC.EFG”.

If this doesn't work immediately, then the DIPswitch is probably in a wrong position, or Hyperterminal isn't set correctly...

Note : for the Nios 1C20 board, use the “debug” port (or modify the Tcl script to use the “Console” port).