Advanced **L**ogic **S**ynthesis for **E**lectronics
http://www.alse-fr.com

RC Servo Controller
Free IP

© ALSE - July 2009, ver 2.1c

Introduction

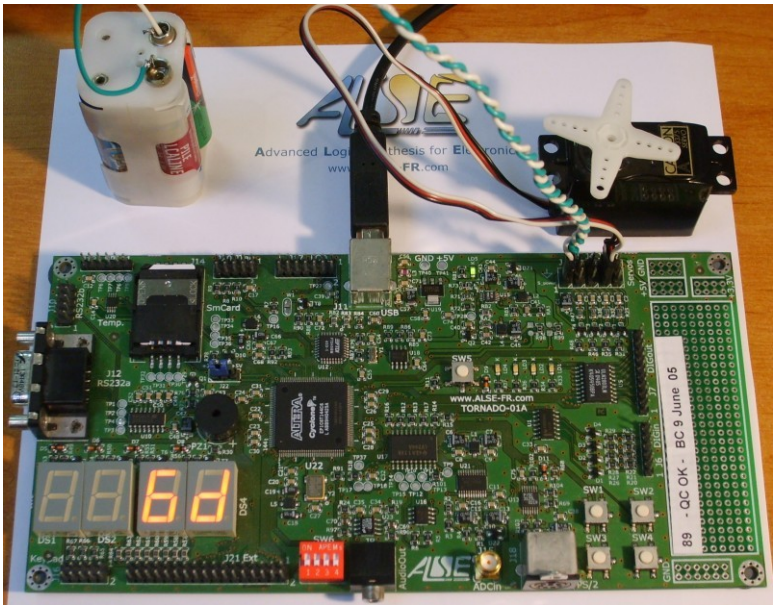
Preamble

We have designed this simple IP to demonstrate how Hardware functions can be simple to design and verify while being very efficiently implemented, provided the right Methodology is used.

In spite of its simplicity, this project does include most of the techniques that are applied to much more complex projects :

- Functional blocks with adhoc complexity, reuse by entities and cut & paste
- Structural design with unitary verification (glass box) + system level simulation (black box)
- Test benches and simulation scripts
- Fully synchronous RTL descriptions with efficient implementation (FSM)
- Automated P& R with Tcl script.

The Servo controller block has been instantiated in a complete system running on an ALSE Tornado FPGA board. Tornado has four RC Servo interfaces, ready to accept widely available Servos. The complete system is provided with the multiplexed 2-digits Hexadecimal display and Push-Button management for Position control.



On the picture above, we can see the servo and its pack of batteries, both connected to the appropriate Tornado connectors (J1 & J3 eg). The value displayed is x6D (109 decimal) and the servo is slightly turned on the right. The value after reset is x80 (128) and with the default settings in the provided code, this servo is centered and the swing is full between 0 and 255 (xFF).

Pushing SW1 does decrement the Position, SW2 does increment the Position, and SW3 does clear (x00) the position; SW4 is the reset and it centers the value (x80).

Pushing SW3 and SW4 allows to measure the servo maximum speed (for a half turn).

Principle

Principle

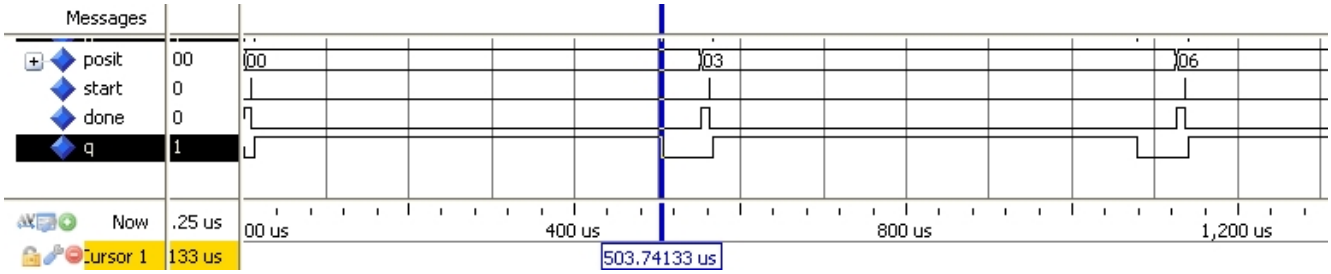
The servo used for the tests is driven by a positive Pulse which duration is :

- 0.5 ms for full right = + 90° (top view)
- 1.4 ms centered (0°)
- 2.3 ms for full left = -90° (top view).

Exact values may depend on particular servos and how they are calibrated, but they are quite typical.

The repetition rate is not particularly important, it's usual to send the pulses every 10 milliseconds, and that's what is done in the top level.

Note that on Tornado we use transistors for the voltage translation (3.3 to up to 6V), and this produces a logic inversion: a '1' at the FPGA output does drive the Open Collector transistor which create a logic 0V at the servo. This logic inversion is implemented in the top level (the Servo Control IP doe issue positive pulses).



The simulation waveform above shows the minimum pulse width duration (Position = 0).

We also see the Done bit coming true some time after the end of the pulse. This amount of time is called “Deadtime” and is a modifiable constant in the IP.

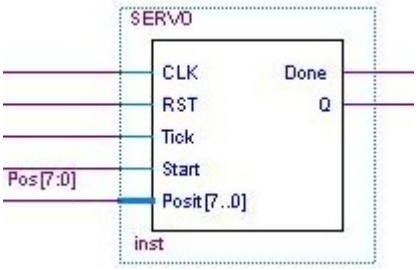
We see the “Start” bit which follows the Done bit, and which starts the generation of a new pulse.

If only one controller is used (ie connected to a servo as for the Tornado setup), we could make the Deadtime much longer and simply connect Done to Start. In practice, we have inserted a ~10 ms delay in the top level between Done and Start.

Another possible use of this IP is to daisy-chain a number of controllers to build a pulse stream. This scheme is used in a multi-channel RC radio transmitter where the pulses are concatenated.

The Servo Controller IP

Servo Controller Interface



This IP is very simple to use.
The signals are described in the entity declaration :

```
-- ---------------------------------------
    Entity SERVO is
-- ---------------------------------------
      Port ( Clk      : In  std_logic;  -- Main Clock
             Rst      : In  std_logic;  -- Asynch Reset
             Tick     : In  std_logic;  -- One clock period high, 7us repetition rate !
             Start    : In  std_logic;  -- Tested only when Done
             Posit    : in std_logic_vector (7 downto 0); -- Pulse length, 0 .. 255
             Done     : out std_logic;  -- indicates end of pulse after deadtime
             Q        : out std_logic   -- Servo PWM output
           );
end entity SERVO;
```

The Tick input should be a one-clock cycle long pulse repeating every 7 us.

All the durations in the controller are timed using this Tick, so it's important to verify that the repetition rate is correct. The accuracy required is not extreme, since servos are typically analog parts and the tolerance is usually wide.

Servo Controller Parameters

Two parameters are available in the source code :

```
constant MinPulse : positive := 500;  -- 0.5 ms min pulse
constant DeadTime : positive := 50;   -- 50 us dead time
```

They are expressed in microseconds for simplicity but keep in mind that they are divided by 7 to end up in number of 7 µs ticks. So the resolution is not 1 µs but **7 µs**.

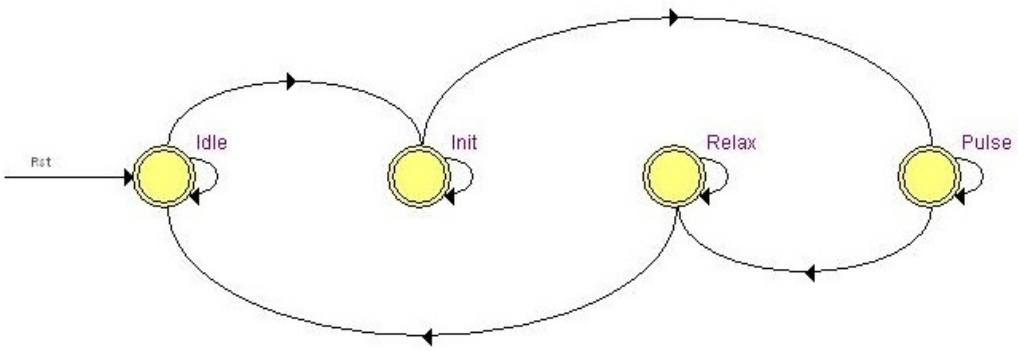
MinPulse is the Pulse length for a Zero input (Posit).

DeadTime is the time between the end of the pulse and the raising of the Done output bit.

If the total swing should be modified, a change in the repetition rate of Tick (7 µs) would be the solution (but DeadTime and MinPulse should also be modified accordingly).

Servo Controller Implementation

The code is easy to understand and basically self-documented. The State machine is very simple :



Complete System

System Blocks

The complete design (top level = TORNADO) includes some logic to drive the peripherals and it does also instantiate the following blocks: RCSERVO (the Controller IP) and FDIV which generates the pulse at 7 µs intervals.

The Tick generation is indeed simple. This module is calculating automatically the correct division factor based on the system clock frequency (which is a generic).

```
-- -------------------------------------
    Entity FDIV is
-- -------------------------------------
   Generic (  Fclock  : positive := 60E6); -- System Clock Freq in Hertz
      Port (  Clk     : In    std_logic;
              Rst     : In    std_logic;
              Tick7us : Out   std_logic  );
end entity FDIV;
```

The generation itself is a simple process :

```
process (Clk,Rst)
begin
  if Rst='1' then
    Count <= 0;
    Tick7us <= '0';
  elsif rising_edge (Clk) then
    Tick7us <= '0';
    if Count = 0  then
      Tick7us <= '1';
      Count <= Divisor-1;
    else
      Count <= Count - 1;
    end if;
  end if;
end process;
```

This code is simple enough to have been coded inside the controller. But if several controllers are used in the same design, it's useless to duplicate the exact same logic as many times.
Factoring out the Tick generation in a common “Frequency Division” module is a good practice.

The Top level has many very simple features coded directly in it, that could also have been put in separate entities, like the multiplexed display management and the 7-segments decoding. But for simple code as this and used once in the design, “*re-use by cut & paste*” is an acceptable and efficient method. Who would code a unitary test bench for the 7-segments decoder anyway ???

```
process(HexData)
begin
  case HexData is
    when  x"0"  => SevSeg <= not "11111100";
    when  x"1"  => SevSeg <= not "01100000";
    when  x"2"  => SevSeg <= not "11011010";
.../...
    when  x"E"  => SevSeg <= not "10011110";
    when  x"F"  => SevSeg <= not "10001110";
    when others => SevSeg <= (others => '-'); -- Ignored in this (full) case
  end case;
end process;
```

You can also note the use of a one-liner style for the resynchronization Flip-Flops :

```
R1n  <= Reset_n when rising_edge(Clk);
```

Simulation

Simulation

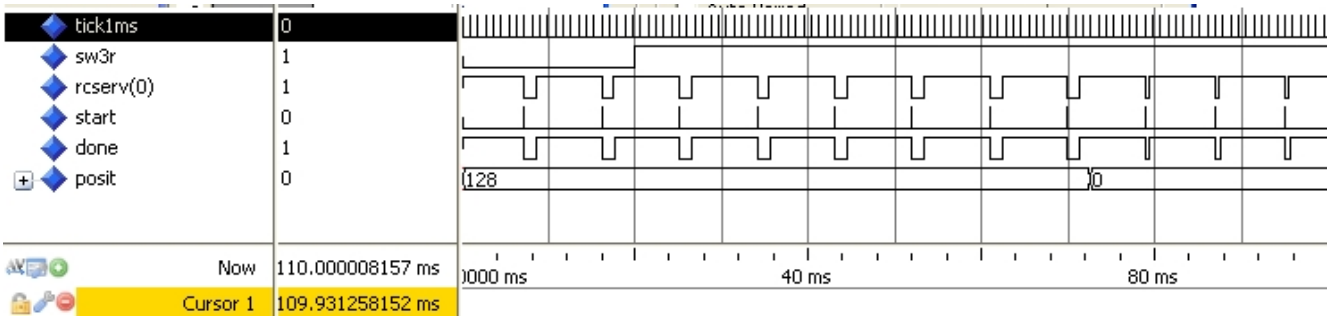
We provide two test benches and two scripts :

- **Servo.do** does compile, load and perform a unitary simulation of the servo controller alone.
- **Tornado.do** does simulate the complete system at the top level, including all the internal modules and functions. Only two values of Posit are simulated : +128 (after reset) and 0 (after te simulation of SW3 being pressed).

Note that he test benches are not self-testing and the user must eyeball the waveforms that are automatically displayed.

A special note to beginners: even if the code is already written and has been tested, **you should always simulate** before trying your code on a board ! Take this habit and stick to it: simulation is centric in the right Methodology for developing complex (or simple) projects.

Top Level Simulation result :



The active pulse is on rcserv(0) and it is active low.

We only simulate two values for *Posit*, and this is sufficient in a system-level verification since the detailed (unitary) verification of the RC controller is performed by a unitary test bench and a separate simulation.

The reason behind the top level simulation is to ensure that the system as a whole will likely operate correctly at first try on the board. We test the interaction of the individual blocks, the correct setting of parameters, the polarity of signals, the interfaces etc...

System-level simulation is different and complementary to unitary simulation.

The top-level *stimuli* are pretty trivial :

```
-- Clock, Reset, Tick7us and simulation end
Clk     <= '0' when Stopped else not Clk after (Period / 2);
Reset_n <= '0', '1' after Period;
SW3     <= '0' after 20 ms;
Stopped <= true after 110 ms;
```

Note that it would be very easy to build self-testing test benches: measure and verify the actual pulse widths in VHDL. We leave this as an exercise.

Project Management – Place & Route

Place and Route

As usual, and thanks to the excellent scripting support inside Quartus II, we designed a Tcl script which does everything for the project :

- Creates the Project
- Add the design files
- Assign the I/O pins
- Define some constraints (like clock frequency)
- Activate some options (like the DRC to verify automatically the absence of coding issues)
- Compile the project
- Builds the Raw Binary file
- Launches our Utility to program the FPGA

This complete script is launched by a single command in the **make.bat** batch file. You can simply double-click on make.bat. We have also included a complete cleanup batch which deletes everything except the rbf file, including the Quartus project etc. This shouldn't be a problem since the Make script can re-create everything in a few seconds.

Needless to say that Quartus II must be properly installed (and licensed) and that you need a Tornado board and a servo to try the system. The free version of Quartus II is perfectly adequate.

Conclusion

This simple IP does demonstrate how to code efficiently and easily typical hardware functions that involve: sequencing, delays and timing controls, data multiplexing, decoding, simulation environment, and automatic project management.

If you are a hobbyist, we welcome small emails to let us know of your projects, often exciting, but we do not provide technical support on our Free IPs.

If you are a student, it's probably a good idea to re-design this from scratch, as a good exercise and re-use as few items from us as possible.

Enjoy !

Bertrand CUZEAU
Technical Manager A.L.S.E.
E-mail: info@alse-fr.com