B. Cuzeau – July 2004 – v1.3

http://www.alse-fr.com

Application Note 204189

Motor Control Basics

and Programmable Logic

Introduction

Motor control is a very old field which has seen very interesting developments lately, due to the emergence of digital techniques and the availability of high-voltage, high current, low RdsOn and fast switching components. We will see in this Application Note that using Programmable Logic can also bring tremendous advantages in terms of performance, while minimizing the cost of the control system.

At ALSE, we have developed many applications in this area, and we have helped our customers build systems with performance which would not have been achievable by other means !
This simple ApNote highlights some basic rules and shows how simple building a motor controller with Programmable Logic can be.

Power Control

The electronics engineer usually views motors as power-hungry devices that can draw large amounts of currents, especially in short bursts during acceleration and braking. Controlling this current is a necessity, either as a safety issue or as a part of speed and/or torque control.

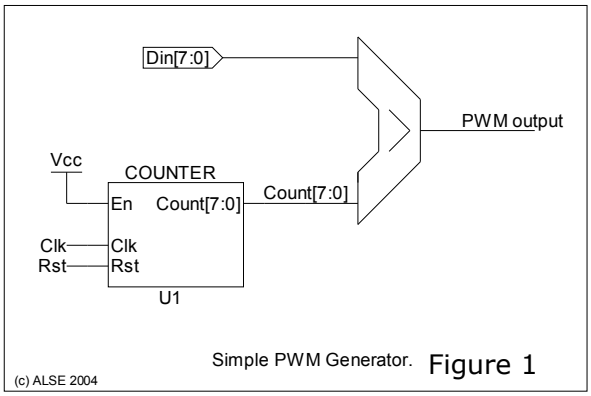
In the very early days (and quite a while after) modulating large currents was done mainly by inserting variable resistors in the path. Obviously, this led to power losses, heavy and expensive "controllers", and poor performance. Today, Pulse Width Modulation (PWM) and its derivatives is a technique which enables the control of high energies with maximal efficiency and power savings. (power conversion shares many issues with motor control).

The Simplest PWM

A single digital output pulsed at a sufficiently high rate with adequate filtering, can be equivalent to an analog proportional voltage or current source.

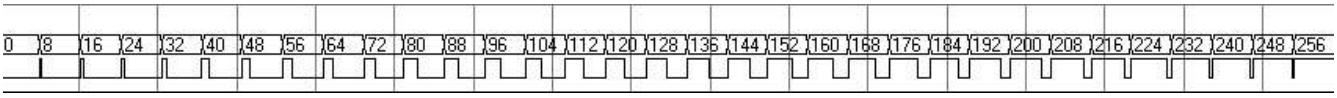
We can demonstrate this with any CPLD demo board with an LED (see "Heartbeat" module description in Appendix).

A simple PWM controller can be built with a counter and a comparator as in Figure 1.



Simple PWM Generator. Figure 1

The Comparator's input simply drives the cycle ratio, the frequency remains constant, and there is only one transition (both ways) per period. Some care must be exercised to handle the fact that allowing values in $0..2^N$ range *inclusive* requires $2^N + 1$ values... See the simulation waveform below. The VHDL code is in the Appendix.



Another possible implementation can use a single accumulator as in Figure 2.

In this case, the carry output of the accumulator is the PWM information. It is pulsed between 0 and 2^N -1 times during one counter's clock period. This system (aka "one bit DAC") is usually better suited to **Digital -> Analog conversion** since it is easier to filter : the PWM period is further divided since high values are evenly spread during the period and interspersed with 0 values. A 50% cycle ratio creates a frequency of 128 times the PWM frequency (for an 8 bits resolution PWM), as illustrated with the simulation waveform :

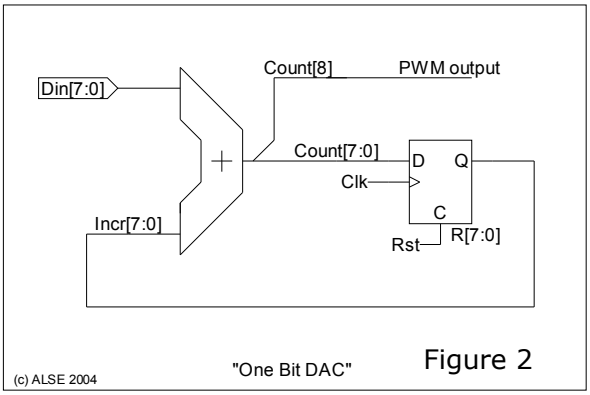
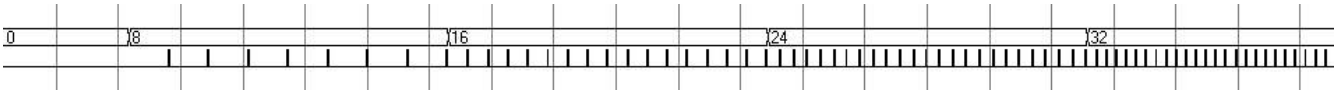


Figure 2



But this solution is sometimes less appropriate in power control where switchings come at a cost and must be minimized (but its advantage is the reduction of low frequencies). As a first approach, we will not discuss this technique any further for our motor control applications.

Anyway, keep in mind that :

Fast enough clock + Programmable Logic + Fast power switches = Accurate Power Control !

Bi-directional Current Flow

The simple PWM we've seen first, associated with an appropriate power switch (transistor, FET, igBT, etc...) and current protection can be suitable to many DC motor control applications.

When we have to drive an AC motor (or to reverse direction in a DC motor, or to apply brake), the **H-bridge** represented Figure 3 is a good solution.

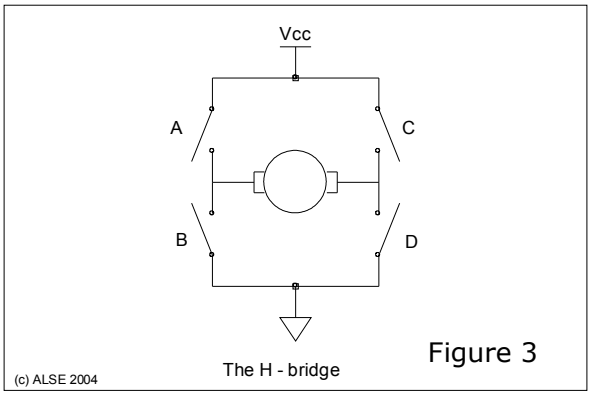


Figure 3

However, this bridge imposes new constraints : if the two diagonally opposed branches (AD and BC) are simultaneously conducting, or even just A and B (or C and D) together, a current peak happens between the power supply rails and can quickly lead to the destruction of the current switches ("smoke condition").
Since switching devices are not ideal, they exhibit turn-on and turn-off delay. As a consequence, the simultaneous conduction can happen *even when there is no overlap in the digital commands !*

For this reason, it is wise and common practice to include "dead times" in the switching cycle, waiting for "AD" to shut down before switching on the "BC" branch (and vice-versa). A small current overlap may be acceptable, and this adjustment may become quite critical, often implying a balance between safety and performance. Sophisticated systems may have these delays adjustable, sometimes on-the-fly (according to operating parameters and actual current flows).

Another nice feature of the H-bridge is that by driving B and D simultaneously (or A and C), we can "brake" by short-circuiting the motor and thus applying its back-EMF to itself.

By contrast, we can also leave the motor floating when at least 3 switches are open (free-wheeling). Opening all four switches may not be a good idea, however.

When high currents and safety become serious issues, the system is complemented with current and voltage sensors. The open-loop system can then become a closed loop one (see next section).

The H bridge structure is so popular that semiconductor vendors have created hundreds of ready-to-use drivers for generic purpose, low power to high power, low voltage to high voltage, for specific applications (stepper motors), with embedded thermal control and protection, etc...
Before building your own bridge driver, make sure you've first checked out all the commercially-available driver devices. If you build your own, you will probably have to add many extra components to drive, protect and compensate your switching elements, prevent spikes, etc...

Advanced Switching Techniques

This Application Note is only a generic introduction, but we must however mention that many advanced switching techniques have been developed. The motivation behind these techniques is usually about handling high currents with maximum efficiency. This applies to many domains, from arc-welding machines to automotive traction. Among these techniques we find "***Soft switching***", "***Quasi-resonant***", "***Zero-current***", "***Zero-voltage***" switching, etc.
A simple Internet search with these keywords will point to a lot of interesting literature.

In many of these systems, there is ***more than one switch*** *to control for each leg* of the bridge : the main switch is often "assisted" by a smaller switch that injects or derives current or voltage.

Typically, these systems work at higher switching Frequencies, more in the100 kHz range than in the 10 kHz range.

Achieving the proper and accurate timing of all the switches of a complex bridge is then only possible with **Programmable Logic**.

SineWave Digital Synthesis

The block diagram Figure 4 illustrates how a purely digital system can accurately generate sinewaves at any frequency (provided the logic runs at a much higher clock rate than the frequency to generate, which is the case up to a few kHz).
Pulsing `SpdCntrl` at the correct rate does the trick.

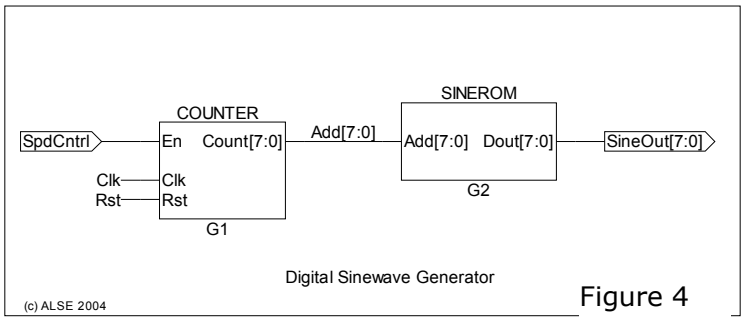
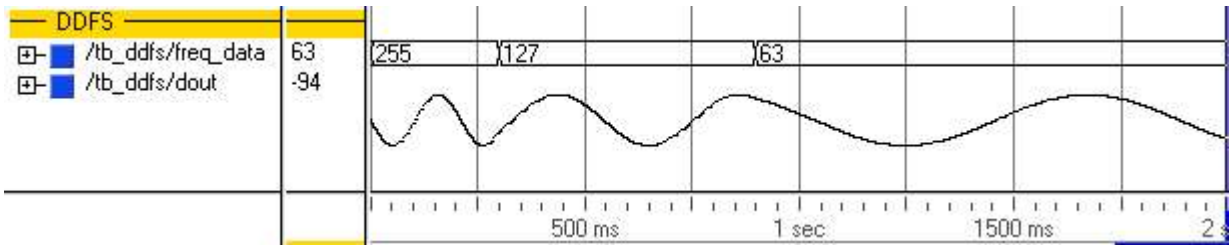


Figure 4

The **Sinewave table** is a simple Rom, or can even be implemented in combinational logic (gates), depending on the width and number of steps.
Indeed, it is not necessary to code the full 360° sinewave : coding between 0 and 90° is enough, since the rest of the sinewave can be deduced by simple symmetries (which can be implemented by simple transformations on the sine table's address bus). Our DDFS example, reproduced in Appendix, is such a generator (we take advantage of only one symmetry). Below the output of this Very Low Frequency Generator :



This technique can also be employed to **enhance the resolution of some stepper motors**, by replacing the 4 usual on-off phases with 90° out-of-phase sine waves.

Closing the Loop

When current and/or voltage sensors are added, the command system can implement simple protection, but the complexity can go up to the most sophisticated servo regulation algorithms. Lots of feedback control algorithms exist, from the venerable P.I.D. (Proportional, Integral, Differential) still used in some applications, to trickier but potentially more efficient predictive algorithms like Kalman filter (used with some success in positioning systems).
The PID algorithm is relatively simple to implement in hardware, but digital sampling poses serious questions (as compared to traditional analog implementations). Care must be exercised while dimensioning and implementing digital **P.I.D.** loops (especially regarding the "D" term).

Digital techniques can also be employed to filter the input measurements, remove noise, and make complex calculations (effective, apparent and complex powers, cos-phi evaluation and correction, etc...).
Again, the advances in Digital Signal Processing and the power of modern hardware components have combined to allow new levels of performance. Entire books have been written on this subject.

But simple ideas are still welcome, like applying ramps instead of brutal variations to a command. Such a "**ramp**" module is very easy to implement in an FPGA. This "ramp" function is also very welcome in the case of open-loop systems.

Input Data processing

The current and voltage information collected by the ADC(s) usually needs some processing to provide useful information. Simple to implement algorithms will allow you to measure active and reactive powers, cos Phi, etc... and let you act accordingly.

Some of the popular popular "filters" are :

- Zero crossing : useful for phase measurements and for synchronized switching.
- Maximum values (positive and negative) = peak-to-peak amplitude.
- Summing of positive (negative) samples = half-period integral, RMS evaluator.
- Moving average. Sometimes used as a low pass filter.
  * Over exactly one period, no matter the relative phase, represents the **DC offset** of the AC signal. Very important when driving magnetic materials which can easily saturate (this will reduce the coil's inductance, making it behave as a short-circuit).
  * Over exactly one half period, the moving average of the *absolute value* is an image of the **RMS** voltage or current.
- Summing of V * I products (power measurements).
- Fast Fourier Transform : useful to determine the amount of spurious harmonics.

When **Three-Phase Power** is used, digital processing is even more valuable !
Specific ADCs exists that can measure precisely **I and V** for the three phases (like CS5451A).
It is then quite easy to recombine these measurements according to the known math formulas and obtain all the necessary real-time information.

Last, let us just mention that the **temperature** (of the motor, or of the switching elements, or both) can also be used in the complete system, either for information, for compensation, or for safety purposes. Overheating (and in some cases thermal runaway) is a serious design concern which should be considered early in the design.

Sophisticated processing can also open a new realm of applications, like **failure prediction**, **preventive maintenance**, accurate **failure diagnostics**, etc...

Position Encoders

This section deviates slightly from the original subject of Motor control, but it is so often used in conjunction with motors that we think it can be useful here.

Most of the time, we are interested in controlling the mechanical output driven by the electrical motor. An analog measurement of the velocity may suffice (like when driving a drill's motor) but in many cases, a precise measurement of the instantaneous position is needed, and we use position encoders for this purpose.

Many position encoders are based on the principle illustrated on Figure 5.

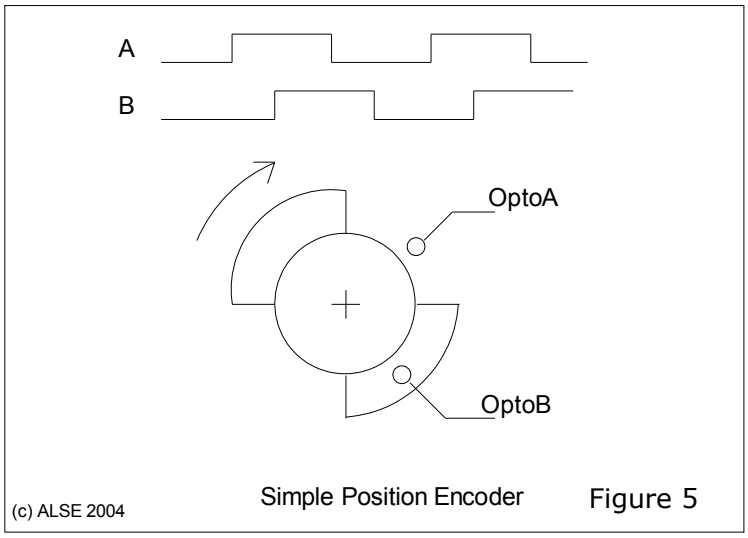


Figure 5

The sensors are placed out of phase by 90°. Therefore, it is possible to identify four positions of the shaft by the (A,B) combination. By definition, the (A,B) binary code is a Gray code (one single bit change at a time).

Real-life encoders are indeed more accurate than 90° ! In this case, more sectors can simply be added. If you open an old PC mouse (not an optical one), you will likely uncover such a coding wheel and two optocouplers. Industrial encoders sometimes use Hall-effect sensors with metal gears.We will see now that decoding the (A,B) values into position + direction of turn information is in fact quite easy by hardware.

The (A,B) cycle can be represented (and decoded) by a simple Finite State Machine (FSM).
For example, the change from state 00 to state 01 should increment the position counter, and the change from 11 to 01 should decrement the counter.

A possible implementation could be to memorize the previous A and B values in (oldA, oldB) with a pair of FlipFlops, and decode UP and DN from the (A, B, oldA, old B) truth table. This is the most compact solution possible.

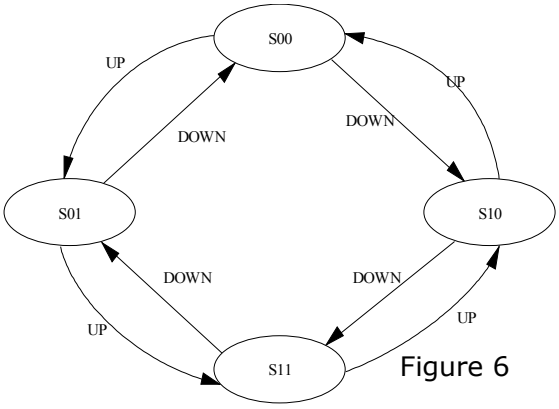
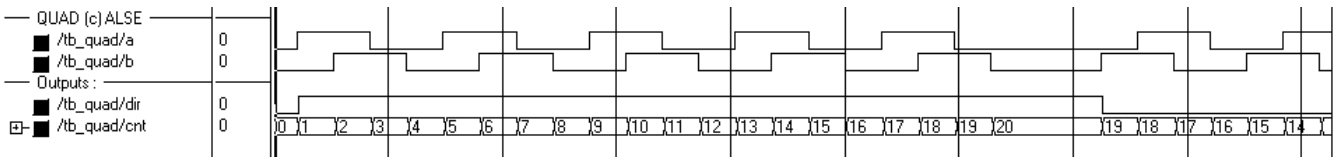


Figure 6

A more versatile and useful solution consists of implementing the decoder precisely as a FSM, matching the one represented Figure 6. See the VHDL code in Appendix. It is then possible to use the UP and DN pulses directly, or the counter which represents an absolute position.
The advantages of the state machine coding are that it becomes easy to match the application's needs exactly (counting +/-1 per turn, or per phase, setting a **Dir**ection bit, etc...) and also to detect "impossible" transitions (not gray compliant) like (1,1) -> (0,0).

Turning this quadrature decoder into an industrial and robust application requires just a few extra precautions and coding, like resynchronizing carefully the A and B inputs, handling correctly the power up state (at power up, the encoder's position may not be 0,0 !), etc.

Here is the simulation of the code provided in Appendix :



Safety issues

It is everyone's experience that protection design is an art.

Way too often, Murphy's law sees to it that the most costly part blows **first** and very "efficiently" protects the $0.10 fuse :-)

When high currents, costly peripherals or indeed human safety are involved, several different levels of protection mechanisms must usually be implemented.

We recommend keeping simple external devices as "last chance" or ultimate protection, like fuses, circuit breakers, analog comparators, etc... Being independent and simple, these devices must be trimmed to act only when all other protections have failed, but still leave a chance to save the system from destruction. They must not act within the range of "finer" protections which should be designed to prevent the system from *entering* dangerous conditions.

A good system should therefore have (at least) three levels of protection :

(1) Finest : which prevents entering the danger zone with some safety margin. Sophisticated systems try acting as early as possible (like by smoothing the input command discontinuities, or by keeping an eye on the derivatives),

(2) Safety mechanisms when the system enters the danger zone.
This often means handling the "zero margin area", like when I = 100% x Imax,

(3) Fast and independent "last chance" shutdown devices, usually beyond 100%.

With a digital system monitoring lots of parameters, other levels of protections can be implemented : for example detecting a body blocking an elevator's door or a car's window.

As a rule of thumb :

> When designing a new **industrial*** system :
> try and **acquire as many parameters** as possible.

With modern multi-channel ADCs (small, cheap , accurate...), the cost is usually minimal.
If you won't use the information later, no big deal. But when the system enters the field tests and if you discover that some unexpected conditions do happen, you will be very happy to be able to add extra processing logic without revising your board.

Temperature monitoring is such an example.

* : Consumer products and very high volume applications are another story, since budget constraints might be so severe that you are trying to remove even a single unnecessary resistor !

Programmable Logic to the rescue

All the techniques described above fit remarkably well in **programmable devices**.
They are simple to implement, and performances are extremely predictable, up-front.

As a comparison, DSP or microcontroller-based solutions are difficult to implement : keeping the processor busy on a task prevents it from handling other tasks. In programmable logic, we can add as many PWMs and regulators as we want, since they all work in parallel and cannot influence each other in any way. Simple tasks, like fast PWMs with dead times, which require just a few percent of the smallest modern CPLD may be very demanding if not impossible to implement by software, and difficult to qualify.
Last but not least, a hardware implementation will be much less prone to fatal problems like simultaneous conduction in a H-bridge. In hardware, it is trivial to prevent A and B from being active together (a simple AND-NOT gate does it !), but this is much more difficult in software.
If you're not convinced, then ask a DSP expert to simply flash four LEDS at different speeds, implementing four "heartbeats" (0.6Hz, 0.8Hz, 1Hz and 1.5 Hz). The four LEDs must vary proportionally with 127 steps between full on of full off. If you end up with a working system, then look at the program the development and verification efforts and the processor load for this trivial task (and we're not even asking for µs resolutions here !). It's a 15 minute effort in HDL, any CPLD, any platform, any clock frequency... Just take a look in the Appendix, and feel free to use this "heartbeat" module in actual designs -as we do- to indicate a working system : it looks better than a lit or blinking LED. A Quartus project for the Nios-Cyclone board is on our website.

Hardware implementations allows very complex filtering algorithms to be used without any potentially negative impact on critical real-time activities (like driving the power switches). Again, everything is done in parallel in hardware; you may run out of hardware resources (gates, flip-flops, memory) when implementing the system but, once you have chosen a sufficiently large device to fit the design, there is no risk of ending up with an overloaded or unreliable system.

Flexibility : be more agile than your competitor(s) !

Let you competitors suffer the dinosaur syndrome, and just be smarter...

A Programmable Logic solution, provided that it is developed by a competent designer, is **easy to develop**, **predictable** from start, and **extremely efficient** : fewer components, better results, shorter development cycle, fewer bugs, and very easy to maintain. For example, you can easily add an extra feature without compromising the whole system (software implementations are not so flexible in this respect).

Get the Digital Controller your system exactly needs, but do not restrict your system's scope for future expansion as a consequence of the limitations of a particular DSP or microcontroller ! Even the cheapest Programmable Logic Devices (like the Altera Cyclone family) offers enough logic to implement very complex algorithms.
A typical sophisticated PWM controller uses a few hundreds Logic Cells at most, while the smallest Cyclone device has more than 3,000 ! You can add several SPIs, ADC interfaces, a UART, a front panel interface, an LCD display and much more, still in the same smallest Cyclone device, with no external chip. And if you cannot live without a processor, you can easily add a NIOS RISC cpu.

Protect your investment !
A (carefully designed) hardware solution is essentially RE-USABLE. Thanks to HDL languages (like VHDL or Verilog), the design has guaranteed portability and tool independence.

One development : many applications !
Once a technology is carefully set up, you can easily apply it to different systems. You can also reuse a common controller board for many different systems.

Better life cycle !
Based on Programmable Logic, your product will have a lot of evolution potential. You can add features, modify its behavior, fix issues due to different external devices or operating conditions, etc... For example, changing a parallel ADC for a serial one (or vice-versa) is a minor effort in an FPGA/CPLD, but can be very troublesome in a software solution.

ALSE : we're here to help...

We have successfully designed many PWM-based systems and each time, the customer was able to greatly improve the system's performance as compared with other solutions (analog, dedicated logic, DSP- or processor-based systems). Such process-control applications are easy to implement in Hardware (Programmable Logic) and can co-exist with other useful functional blocks (data acquisition, filtering, averaging, surge protections, digital I/Os management, servo loop control, communication, i²c, RS232, etc...).

Typically, a complete PWM command system takes us just a few days to design.

Or you can opt to let us build for you a demonstrator based on an existing FPGA/CPLD board and have a ready-to-use mock-up system within a few days, with which you can prove the usefulness and performance of digital control !

Don't hesitate : send us an simple email describing your application. We will then ask you some extra questions if necessary, and we will quote the design of your FPGA/CPLD.

Contact :

Bertrand CUZEAU
Technical Manager A.L.S.E.
mailto:info@alse-fr.com
http://www.alse-fr.com

Many thanks to :

Jonathan Bromley (Doulos UK) for his kind and valuable review, his useful comments, and for patiently fixing my English :-)
Jean Michel Vuillamy for his encouragements in writing this ApNote.

See also :

http://www.doulos.com/knowhow

Archive's contents - at http://www.alse-fr.com/Motor_ALSE.tar.gz

```
-- PWM project, 2 x versions and simulation

PWM\pwm.vhd     : PWM module, 2 x RTL architectures
PWM\tb_pwm.vhd  : Simulation Test bench for the 2 x architectures (2 x UUTs)
PWM\simpwm.do   : Simulation script for ModelSim

-- Quadrature Decoder project

QUAD\quad.vhd    : Quadrature decoder RTL module
QUAD\tb_quad.vhd : Simulation Test Bench for Quad decoder
QUAD\simquad.do  : Simulation script for ModelSim

-- Direct Digital Frequency Synthesizer

DDFS\ddfs.vhd    : RTL module
DDFS\tb_ddfs.vhd : Simulation Test Bench for DDFS
DDFS\simddfs.do  : Simulation script for ModelSim

-- HeartBeat project - Nios Cyclone Board

\HearBeat\HeartBeat.vhd    : VHDL source : Heartbeat module and multi-module Top-level.
\HearBeat\tb_heartbeat.vhd : VHDL Simulation Test bench
\HearBeat\simheart.do      : ModelSim simulation script
\HearBeat\do_heart.tcl     : "does-it-all" Tcl script for Quartus II.
\HearBeat\TopBeat.sof      : ready to use bitstream for Nios-Cyclone-1C20

To fit & run the "HeartBeat" Nios board project :
- Launch Quartus II
- Make sure the Tcl window is opened (Alt-2)
- in the Tcl window, enter the following commands :
  * cd <Heartbeat_directory>
  * source do_heart.tcl
- Download the cyclone device and look at LEDs 0 .. 3 !
Note : the "clean" utility batch must be used with care.
```

Appendix A
Simple PWMs VHDL Source code (so simple...)

We include here the simple VHDL code for the basic PWM and One-Bit-DAC, as well as for the Direct Digital Frequency Synthesizer. The latter is just an illustration of the principle and should **not** be used "as is".

```
-- ----------------------------------------
--   Simple PWM ! (and One-Bit DAC)
-- ----------------------------------------
-- http://www.alse-fr.com
-- B. Cuzeau

library IEEE;
  use IEEE.std_logic_1164.all;
  use IEEE.numeric_std.all;

-- ----------------------------------------
    Entity PWM is
-- ----------------------------------------
    Port (    Rst : In  std_logic;
              Clk : In  std_logic;
              Din : In  std_logic_vector (8 downto 0);
           PWMout : Out std_logic );
end PWM;


-- ----------------------------------------
    Architecture RTL_Accum of PWM is
-- ----------------------------------------
-- AKA "One-Bit DAC". Not suitable for motor control.
-- can be used for audio signals synthesis for example,
-- by driving a simple RC filter...

signal Accum : unsigned (8 downto 0);

begin

U1: process (Clk,Rst)
begin
  if Rst='1' then
    Accum <= (others=>'0');
  elsif rising_edge(Clk) then
    Accum <= '0' & Accum(7 downto 0) + unsigned('0' & Din(7 downto 0));
  end if;
end process U1;

U2: PWMout <= Accum(8) or Din(8);

end RTL_Accum;


-- ----------------------------------------
    Architecture RTL_Comparator of PWM is
-- ----------------------------------------
-- Simple PWM, counter + comparator solution.
-- do NOT use without registering PWM_out !!!

signal Cnt : unsigned (7 downto 0);

begin

U1: process (Clk,Rst)
begin
  if Rst='1' then
    Cnt <= (others=>'0');
  elsif rising_edge(Clk) then
    Cnt <= Cnt + 1;
  end if;
end process U1;

U2: PWMout <= '1' when unsigned(Din(7 downto 0)) >= Cnt or Din(8)='1'
         else '0';

end RTL_Comparator;
```

Appendix B
Direct Digital (low) Frequency Synthesizer

This small "DDFS" example below demonstrates the generation of a sinewave from a Table of values. Coupling this module with one of the previous PWM generators will make a cheap analog sinewave generator for very low frequency. For higher frequencies, just shorten the 29 bits divisor.

The frequency we get is `Foutput = Freq_Data * Fxtal / 2**29` .

For Freq_Data=255 and a 10 MHz system clock, we get a 210 ms period (~ 4.8Hz).

For Freq_Data=1, we get ~ 0.019 Hz (a period of ~ 53 seconds) !

```
-- DDFS.vhd
------------------------------------
-- Direct Digital Freq. Synthesis  --
------------------------------------
-- (c) Bert Cuzeau, ALSE - info@alse-fr.com
-- May be reproduced provided that copyright above remains.
-- We use one of the symetries in the sine function,
-- so the lookup table is re-used twice (128 entries table)
-- The Sine Table is built by a C program...
------------------------------------
-- Design IOs :
--   Clk : Global Clock input
--   Rst : Global Reset input
--   Freq_data : 8-bit frequency control vector
--               from DIP switches on the board.
--   Dout : is signed 8-bit output to the DAC.
-- ----------------------------------------------

library IEEE;
  use IEEE.std_logic_1164.all;
  use IEEE.numeric_std.all;

-- ----------------------------------------------
    Entity DDFS is
-- ----------------------------------------------
Port ( CLK       : in  std_logic;
       RST       : in  std_logic;
       Freq_Data : in  std_logic_vector (7 downto 0);
       Dout      : out std_logic_vector (7 downto 0)
     );
end DDFS;

-- ----------------------------------------------
    Architecture RTL of DDFS is
-- ----------------------------------------------

  signal Address : unsigned (6 downto 0);
  signal Result  : std_logic_vector (7 downto 0);
  signal Accum   : unsigned (28 downto 0); -- we want very low Frequencies !
  signal Sign    : std_logic;

begin

-- Signed Accumulator
-- ------------------
Acc: process (CLK,RST)
begin
  if RST='1' then
    Accum <= (others=>'0');
  elsif rising_edge(CLK) then
    Accum <= Accum + unsigned(Freq_Data);
  end if;
END process acc;

Sign <= Accum(Accum'high); -- MSB

-- Lookup Table Index calculation
-- ------------------------------
Address <= unsigned(Accum(Accum'high-1 downto Accum'high-Address'length));

--  SINE Look-Up TABLE
-- -------------------
-- Inference of an Asynchronous Rom.
-- A synchronous one would be better, but we register the output.
-- This table has been built by GENVEC.exe (C program)
-- We use only positive values ! (sign comes from quadrant info)
-- This could be further optimized by coding only one quadrant...
lookup: process (Address)
  subtype SLV8 is std_logic_vector (7 downto 0);
```

```
  type Rom128x8 is array (0 to 127) of SLV8; -- 0 to 2**Address'length - 1
  constant Sinus_Rom : Rom128x8 := (
    x"00",  x"03",  x"06",  x"09",  x"0c",  x"0f",  x"12",  x"15",
    x"18",  x"1b",  x"1e",  x"21",  x"24",  x"27",  x"2a",  x"2d",
    x"30",  x"33",  x"36",  x"39",  x"3b",  x"3e",  x"41",  x"43",
    x"46",  x"49",  x"4b",  x"4e",  x"50",  x"52",  x"55",  x"57",
    x"59",  x"5b",  x"5e",  x"60",  x"62",  x"64",  x"66",  x"67",
    x"69",  x"6b",  x"6c",  x"6e",  x"70",  x"71",  x"72",  x"74",
    x"75",  x"76",  x"77",  x"78",  x"79",  x"7a",  x"7b",  x"7b",
    x"7c",  x"7d",  x"7d",  x"7e",  x"7e",  x"7e",  x"7e",  x"7e",
    x"7f",  x"7e",  x"7e",  x"7e",  x"7e",  x"7e",  x"7d",  x"7d",
    x"7c",  x"7b",  x"7b",  x"7a",  x"79",  x"78",  x"77",  x"76",
    x"75",  x"74",  x"72",  x"71",  x"70",  x"6e",  x"6c",  x"6b",
    x"69",  x"67",  x"66",  x"64",  x"62",  x"60",  x"5e",  x"5b",
    x"59",  x"57",  x"55",  x"52",  x"50",  x"4e",  x"4b",  x"49",
    x"46",  x"43",  x"41",  x"3e",  x"3b",  x"39",  x"36",  x"33",
    x"30",  x"2d",  x"2a",  x"27",  x"24",  x"21",  x"1e",  x"1b",
    x"18",  x"15",  x"12",  x"0f",  x"0c",  x"09",  x"06",  x"03" );
begin
  Result <= Sinus_Rom (to_integer(Address));
END process lookup;

-- Output registers
-- ----------------
process(CLK,RST)
begin
  if RST='1' then
    Dout <= (others=>'0');
  elsif rising_edge(CLK) then
    if Sign='1' then
      Dout <= Result;
    else
      Dout <= std_logic_vector (- signed(Result));
    end if;
  end if;
end process outreg;

end RTL;
```

Note that we've used attributes so modifying the Accumulator's size automatically adjusts the assignments to **`Address`** and **`Sign`**.

Appendix C
Heart-Beat LED Flasher

This very simple example uses PWM to cycle the power of an LED progressively. The effect is a bit like heart beats and is a cool way to blinking an LED.

```
-- HeartBeat.vhd
-- ------------------------------------------------------------------------
--   LED Flasher "Heart Beat" (c) ALSE
-- ------------------------------------------------------------------------
--  Version  : 1.1
--  Date     : Feb 2004
--  Author   : Bert CUZEAU
--  Contact  : info@alse-fr.com
--  Web      : http://www.alse-fr.com
-- ------------------------------------------------------------------

library IEEE;
  use IEEE.std_logic_1164.all;
  use IEEE.numeric_std.all;

entity HeartBeat is
port ( Clk, Rst  : in  std_logic;    -- Clock and async reset
       Tick10us  : in  std_logic;    -- 100 kHZ ticker
       LED       : out std_logic );  -- LED output
end HeartBeat;

architecture RTL of HeartBeat is
begin

process (Clk, Rst)
  variable PWM : unsigned(7 downto 0); -- 8 bits
  variable Lum : unsigned(7 downto 0);
  variable updir : boolean;
```

```
begin
  if Rst='1' then
     PWM := x"00";
     Lum := x"7F";
     LED <= '0';
     updir := true;
  elsif rising_edge (Clk) then
    if Tick10us='1' then
      PWM:=PWM+1;
      if PWM = 0 then
        if Lum = 0 then updir := true; end if;
        if Lum = 255 then updir := false; end if;
        if updir then
          Lum := Lum + 1;
        else
          Lum := Lum - 1;
        end if;
        LED <= '0';
      elsif PWM <= Lum then
        LED <= '1';
      else
        LED <= '0';
      end if;
    end if;
  end if;
end process;
end RTL;
```

Explanations (not really necessary) :

- a PWM cycle take 256 * 10 us (~ 400 Hz).
- After each cycle, the PWM cycle ratio ("Lum") is incremented ("updir" true) or decremented ("updir" false).
- When Lum reaches 255 on the way up, updir goes false and true again when Lum reaches 0 on the way down, and so on.

A complete Full On - Full Off – Full On cycle then takes 512 * 256 *10 us ~= 1.33 second.

In this PWM, the averager (or "integrator") is your eye (retininan persistance), giving you the illusion of proportional lighting level.
The "beat" rate can be adjusted by varying the 10 us Ticker.
Hint : invert the LED output, drive a nearby LED with it, and watch...

Appendix D
Quadrature Decoder

```
-- QUAD.vhd
-- ------------------------------------------
--  Quadrature Decoder State Machine
-- ------------------------------------------
-- (c) 2004 - B. Cuzeau, ALSE
-- http://www.alse-fr.com
-- Contact : info@alse-fr.com
-- Notes :
-- * A and B must absolutely be resynchronized outside !
-- * FSM State encoding should be sequential/binary or custom
-- * Implemented as re-synchronized, one-process, Mealy State machine.
-- * Change the counter size directly in the port declaration.
-- * Cnt (n downto 2) returns the position in full turns.

library IEEE;
  use IEEE.std_logic_1164.all;
  use IEEE.numeric_std.all;

-- ------------------------------------------
    Entity QUAD is
-- ------------------------------------------
-- A & B must absolutely be resynchronized outside
  port( A,B : in  std_logic;
        Rst : in  std_logic;
        Clk : in  std_logic;
        Cnt : out std_logic_vector(7 downto 0); -- unsigned value !
        Dir : out std_logic );
end entity QUAD;
```

```
-- -----------------------------------------
    Architecture MealyR of QUAD is
-- -----------------------------------------

subtype SLV2 is std_logic_vector (1 downto 0);
attribute enum_encoding : string;

type   state_t is (Boot, S00, S01, S10, S11);
attribute enum_encoding of state_t : type is "100 000 001 010 011";
signal state : state_t;

signal Count : unsigned (Cnt'range);

begin

Cnt <= std_logic_vector(Count);

process (Rst,Clk)
begin
  if Rst='1' then
    State <= Boot;
    Count <= (others=>'0');
    Dir   <= '0';
  elsif rising_edge(Clk) then

    case state is

      when Boot =>           -- handle the initial value of the coder
        case SLV2'(A & B) is
            when "00" => State <= S00;
            when "10" => State <= S10;
            when "11" => State <= S11;
            when "01" => State <= S01;
            when others => null;
          end case;

      when S00 =>
        case SLV2'(A & B) is
          when "10" => State <= S10;
                       Count <= Count+1;
                       Dir   <= '1';
          when "11" => null; -- possible : State <= S11;
          when "01" => State <= S01;
                       Count <= Count-1;
                       Dir   <= '0';
          when others => null;
        end case;

      when S10 =>
        case SLV2'(A & B) is
          when "00" => State <= S00;
                       Count <= Count-1;
                       Dir   <= '0';
          when "11" => State <= S11;
                       Count <= Count+1;
                       Dir   <= '1';
          when "01" => null; -- possible : State <= S01;
          when others => null;
        end case;

      when S11 =>
        case SLV2'(A & B) is
          when "10" => State <= S10;
                       Count <= Count-1;
                       Dir   <= '0';
          when "01" => State <= S01;
                       Count <= Count+1;
                       Dir   <='1';
          when "00" => null; -- possible : State <= S00;
          when others => null;
        end case;

      when S01 =>
        case SLV2'(A & B) is
          when "11" => State <= S11;
                       Count <= Count-1;
                       Dir   <= '0';
          when "00" => State <= S00;
                       Count <= Count+1;
                       Dir   <= '1';
          when "10" => null; -- possible : State <= S10;
          when others => null;
        end case;

      when others => State <= Boot;
    end case;
  end if;
end process;
end MealyR;
```