# Tutorial technical document

## Introduction

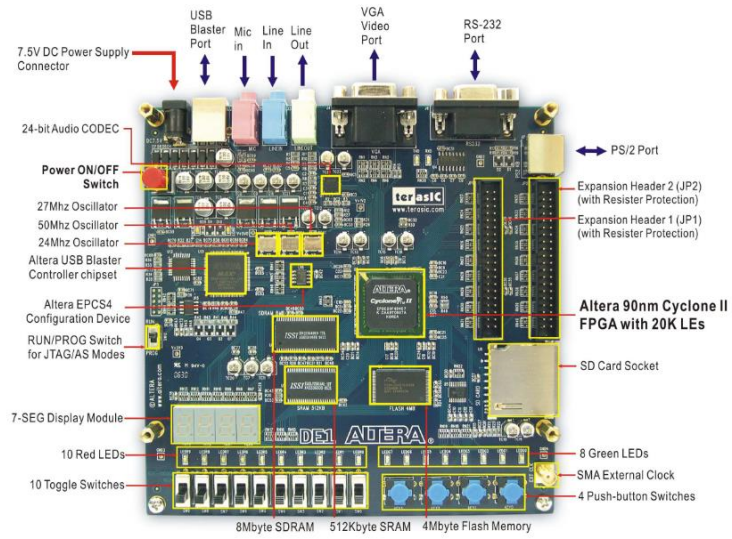
The goal of this project is to control a DC motor by pwm. To achieve this, a code to generate a pwm signal will be written in VHDL language and then implemented on the DE1 ALTERA board.

- PWM

A PWM signal is a periodic signal. In binary, it varies between '1' and '0'. The value of a pwm is defined by an important parameter, the duty cycle. This is the ratio of the time the signal is at '1' to the time of a period. To control the DC motor, the average value of the pwm signal is used, which depends directly on the duty cycle. The higher the value, the higher the average value.

- DE1 ALTERA

The DE1 Altera card is produced by INTEL. It is composed of one RS-232 port, one PS/2 port, one USB port, one VGA output, 72 cyclone II I/O pins, 4 push buttons and 10 switches. 10 red LEDs and a seven-segment display are present on the board. It has 512 Kbytes of SRAM memory and 8 Mbytes of dynamic SDRAM memory. It has 3 internal clocks (24 MHz - 27 MHz - 50 MHz) and allows the use of an external clock thanks to the SMA input.



## PWM Generator 50M.vhd

- **Definition of the entity's entry and exit ports**

The first thing to do when writing a vhdl file is to define its entity with its input and output ports. In our case, the entity is called PWM_Generator_50M and has 4 input and output ports. The first input port corresponds to the 50MHz clock, this clock will be the reference clock in the system and will be brought as input by our DE1 Altera card. Then, the 3 other input ports correspond to the activation of 3 different buttons on the board. The first button DUTY_INCREASE will increase the Duty Cycle by 10%, the second DUTY_DECREASE will decrease the Duty Cycle by 10% and finally the third button direction_H will allow to choose the direction of rotation of the DC motor. Then, for the output ports we have the PWM_OUT_H allowing to turn the motor clockwise, the PWM_OUT_A allowing to turn the motor anticlockwise. But also the PWM_OUT_O which is our pwm output signal regardless of clockwise or counter-clockwise direction. And finally, a 4-bit std_logic_vector signal called LED_OUT to control the LED display.

```
entity PWM_Generator_50M is
port (
    clk_50M: in std_logic; -- 50MHz clock input
    DUTY_INCREASE: in std_logic; -- button to increase duty cycle by 10%
    DUTY_DECREASE: in std_logic; -- button to decrease duty cycle by 10%
     sens_H: in std_logic; -- button to drive motor of clockwise
     PWM_OUT_H:out std_logic;-- rotation motor of clockwise
     PWM_OUT_A:out std_logic;-- rotation motor of anticlockwise
     PWM_OUT_O:out std_logic;-- rotation motor
     LED_OUT: out std_logic_vector(3 downto 0) -- gestion des leds

  );
end PWM_Generator_50M;
```

- **The component DFF_debounce as well as the process debouncing**

The DFF_Debounce.vhd file is an implementation of the D flip-flop to handle bounce when pressing buttons (entries).

```
architecture Behavioral of PWM_Generator_50M is
  -- D-Flip-Flop for debouncing module

component DFF_Debounce
Port(
       CLK : in std_logic;
       en : in std_logic;
       D : in std_logic;
       Q : out std_logic
    );
end component;
```

Indeed, when the FPGA buttons are pressed, there are unexpected bouncebacks that are generally undesirable in the process operation, the Debouncing process allows to manage its anti-bounce

hazards on the FPGA buttons by generating only one pulse with an input clock period when the FPGA button is still pressed, held long enough and released. It creates a delay between two readings.

```
process(clk_50M)
  begin
    if(rising_edge(clk_50M)) then
    counter_slow <= counter_slow + x"0000001";
    --if(counter_slow>=x"BEBC20") then -- for running on FPGA -- comment when running simulation
       if(counter_slow>=x"0000001") then -- for running simulation -- comment when running on FPGA
       counter_slow <= x"0000000";
       end if;
    end if;
  end process;
```

Stage0 and stage1 implement two D flip-flops whose "Duty_increase" button is connected to the input of the first one and its output is connected to the input of the second one. The output of 2nd will then be used as the processed information of the button. This is for bounce management and input button signal shaping. The same applies to stage3 and stage4.

```
stage0: DFF_Debounce port map(clk_50M,slow_clk_en , DUTY_INCREASE, tmp1);
stage1: DFF_Debounce port map(clk_50M,slow_clk_en , tmp1, tmp2);
duty_inc <=  tmp1 and (not tmp2) and slow_clk_en;

-- debounce part for duty decreasing button ( géré les antis rebond décrémentation)

stage2: DFF_Debounce port map(clk_50M,slow_clk_en , DUTY_DECREASE, tmp3);
stage3: DFF_Debounce port map(clk_50M,slow_clk_en , tmp3, tmp4);
duty_dec <=  tmp3 and (not tmp4) and slow_clk_en;
```

- **Counter_pwm, Duty_cycle, pwm_out**

We want to create a pwm signal from our 50 MHz clock with a period equivalent to 10 clock periods. Since the period of the 50MHz clock signal lasts 20ns, the period of our pwm signal will last 200 ns.

For this, we need a counter that will count the number of clock signals that will constitute the pwm signal. So we created a 4-bit std_logic_vector signal, all initiated at 0, called counter_PWM. Thanks to a process that is triggered at each movement of the 50M clock, the value of the counter_pwm will be increased by 1 at each rising edge of the clock. Since we want a pwm of 10 clock periods, we need to limit the counter_pwm to 9 (0 to 9 => 10 periods). For this, if the current value of the counter_pwm is equal to or greater than 9, the counter_pwm takes the value 0.

```
process(clk_50M)
begin
    if(rising_edge(clk_50M)) then
        counter_PWM <= counter_PWM + x"1";
        if(counter_PWM>=x"9") then
            counter_PWM <= x"0";
        end if;
    end if;
end process;
```

Now that the pwm period is set, we need to manage the pwm duty_cycle (set its width) and thus the duration in high position (=1) in the pwm period. To do this, we create the DUTY_CYCLE signal which is std_logic_vector of 4 bits but initiated at hexadecimal value 3. In order that the value of the DUTY_Cycle does not vary within a pwm signal period, we have created the PWM_Cycle signal which takes the value 1 when the counter_PWM is equal to 9. This signal therefore marks the end of a pwm period. We are therefore going to create a process that is triggered each time the PWM signal managing the DUTY_CYCLE changes. In this process, at each rising edge of the PWM_Cycle signal announcing the end of a pwm period, DUTY_CYCLE will increase by 1 when DUTY_INCREASE is equal to 1 or will decrease by 1 when DUTY_DECREASE is equal to 1. However, as the pwm signal has 10 clock periods, we need to limit the DUTY_CYCLE signal to 10 (=100% of the period), for this the DUTY_CYCLE must only increase by 1 by the DUTY_INCREASE when the DUTY_CYCLE is less than or equal to 10. Likewise, since a DUTY_CYCLE being less than zero is not possible, we will only allow the DUTY_CYCLE to be reduced when it is greater than or equal to 1.

```
 PWM_Cycle <= '1' when counter_PWM = x"9"  else '0';
process(PWM_Cycle)
begin

    if(falling_edge(PWM_Cycle)) then
        if(DUTY_INCREASE='1' and DUTY_CYCLE <= x"9") then
            DUTY_CYCLE <= DUTY_CYCLE + x"1";--increase duty cycle by 10%
        elsif(DUTY_DECREASE='1' and DUTY_CYCLE>=x"1") then
            DUTY_CYCLE <= DUTY_CYCLE - x"1";--decrease duty cycle by 10%
        end if;
  end if;
end process;
```

Thanks to the counter_PWM and DUTY_CYCLE signals, we can now get our pwm signal by defining PWM_OUT as 1 only when the counter_PWM is less than the DUTY_CYCLE, otherwise it is 0. Since the pwm signal period corresponds to 10 clock signal periods and DUTY_CYCLE is limited to 10 and varies in steps of 1, each variation of DUTY_CYCLE therefore corresponds to the variation of the pwm (duty cycle) width by 10%.

```
PWM_OUT <= '1' when counter_PWM < DUTY_CYCLE else '0';
```

- **PWM_OUT_H and PWM_OUT_A**

In order to be able to run our motor in one direction or the other, our server must have two outputs. So, to make the motor turn in one direction, you have to send the pwm to the first output and ground the second output. And conversely, to make it rotate in the other direction, the pwm signal must be placed in the second output and the first output must be grounded.

For this in the vhdl code it's simple, we need two output signals PWM_OUT_A (counterclockwise) and PWM_OUT_H (clockwise) as well as the direction_H signal (button). When the direction_H is equal to the output the pwm signal is sent in PWM_OUT_A while when it is equal to 0 the signal is sent in PWM_OUT_H.

```
PWM_OUT_H <= PWM_OUT when sens_H ='1'  else '0';
PWM_OUT_A <= PWM_OUT when sens_H ='0'  else '0';
PWM_OUT_0 <= PWM_OUT;
```

- **LED application**

The application will simply indicate the variation of the duty cycle to the user by displaying on the leds of the DE1 board Altera the number of clock periods that the width of the pwm constitutes.

For this, a 4-bit output port called LED_OUT has been defined as well as an inter LED_Counter signal also defined as a 4-bit std_logic_vector and all set to zero.

In a process, the LED_Counter will be increased by 1 in hexadecimal at each rising edge of the clock when PWM_OUT is equal to 1, and when counter_PWM is greater than or equal to 9 (end of the pwm period), the LED_Counter returns to zero. LED_OUT continuously takes the value of this counter and will be displayed on the leds of the board.

## TEST BENCH

To be able to test the VHDL code generating a pwm signal, it is necessary to create a file to build the test signals called the test bench. The input and output signals must be defined in this one. The same signals as the pwm signal generating entity have been chosen to match the pwm signal generating entity. The names must not be identical but the number of inputs and outputs must match.

```
COMPONENT PWM_Generator_50M
PORT(
                clk_50M: in std_logic; -- 50MHz clock input
                DUTY_INCREASE: in std_logic; -- button to increase duty cycle by 10%
                DUTY_DECREASE: in std_logic; -- button to decrease duty cycle by 10%
                sens_H: in std_logic; -- button to drive motor of clockwise
                PWM_OUT_H:out std_logic;-- rotation motor of clockwise
                PWM_OUT_A:out std_logic;-- rotation motor of anticlockwise
                PWM_OUT_O:out std_logic;-- rotation motor
                LED_OUT: out std_logic_vector (3 downto 0) -- gestion des leds

    );
END COMPONENT;
```

The signals must then be initialized. The input signals (clk_50M, DUTY_INCREASE, DUTY_DECREASE, H-direction) are set to '0'. The output signals are not assigned to a start value.

The next step is the initialization of the unit under test called UUT (Unit Under Test). The vhdl code you want to test must be called and the test bench signals must be matched to those of the UUT.

```
-- Instantiate the Unit Under Test (UUT)
  uut: PWM_Generator_50M PORT MAP (
         clk_50M => clk_50M,
         DUTY_INCREASE => DUTY_INCREASE,
         DUTY_DECREASE => DUTY_DECREASE,
                          sens_H => sens_H ,
         PWM_OUT_H => PWM_OUT_H ,
                          PWM_OUT_A => PWM_OUT_A ,
                          PWM_OUT_O => PWM_OUT_O,
                          LED_OUT   => LED_OUT

        );
```

Next comes the definition of the process for creating the clock signal. A constant called clk_period was initialized at 20ns (corresponding to 50 MHz). In this process, the clock (clk_50M) is set to '0' during half of clk_period (10ns) and to '1' during the other half. This process therefore creates a square wave signal with a period of 20ns.

The next step is the creation of the simulation process itself. Indeed, the input variables (other than the clock) are defined in this process. For the tests presented in the "Simulation" part, the DUTY_INCREASE changes value each time the value of PWM_OUT_O changes. This means that the DUTY_INCREASE is activated every pwm period, which increases the DUTY_CYCLE every pwm period. This operation is repeated 4 times (4 duty cycle increases of 10%). Then the DUTY_INCREASE is left at '0' and the same operation is performed with DUTY_DECREASE. The behavior is similar except that DUTY_CYCLE decreases.

```
stim_proc: process
begin

            -- duty increase

DUTY_INCREASE <= '1';
    wait until PWM_Out_O = '1';
DUTY_INCREASE <= '0';
            wait until PWM_Out_O = '0';
DUTY_INCREASE <= '1';
    wait until PWM_Out_O = '1';
DUTY_INCREASE <= '0';
            wait until PWM_Out_O = '0';
DUTY_INCREASE <= '1';
    wait until PWM_Out_O = '1';
DUTY_INCREASE <= '0';
            wait until PWM_Out_O = '0';
DUTY_INCREASE <= '1';
    wait until PWM_Out_O = '1';
DUTY_INCREASE <= '0';
            wait until PWM_Out_O = '0';
```

```
            -- duty decrease

DUTY_DECREASE <= '1';
    wait until PWM_Out_O = '1';
DUTY_DECREASE <= '0';
            wait until PWM_Out_O = '0';
DUTY_DECREASE <= '1';
    wait until PWM_Out_O = '1';
DUTY_DECREASE <= '0';
            wait until PWM_Out_O = '0';
DUTY_DECREASE <= '1';
    wait until PWM_Out_O = '1';
DUTY_DECREASE <= '0';
            wait until PWM_Out_O = '0';
DUTY_DECREASE <= '1';
    wait until PWM_Out_O = '1';
DUTY_DECREASE <= '0';
            wait until PWM_Out_O = '0';
```

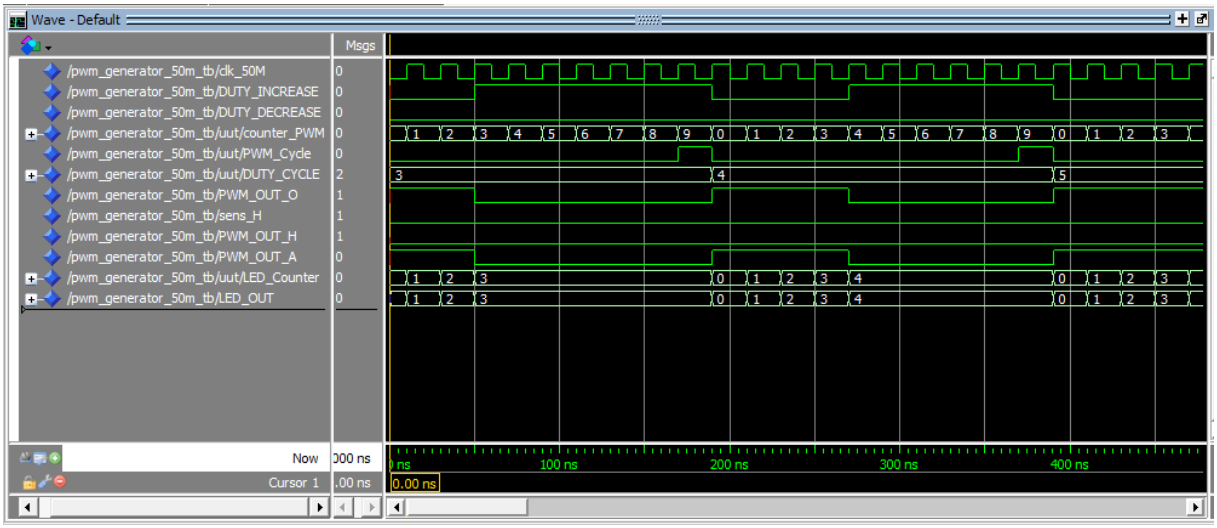
The last command of this process is the reversal of the direction of rotation direction_H.

```
sens_H <= not sens_H ;

 end process;
```

This process "simulates" the use of the Altera card. Indeed, the control of the DUTY_INCREASE, DUTY_DECREASE and sens_H signal values simulates the pressure of the board's buttons.

## Simulation

- **Basic Operation**




Here's the first 400 nanoseconds of our pwm generator test bench simulation.

We can already see on these first 400 nanoseconds that, as previously explained, the pwm signal has a period equivalent to 10 clock periods and that the counter_pwm is counting these 10 periods from 0 to 9.

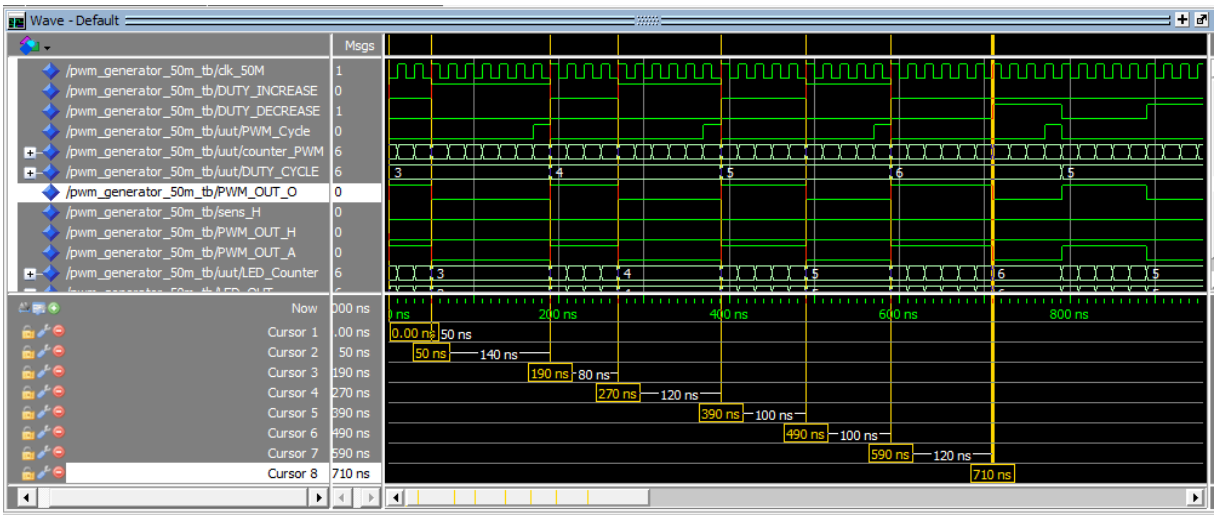
The PWM_Cycle signal is at 1 when counter_pwm is at 9, thus announcing the end of the pwm period. Thanks to this PWM_Cycle, DUTY_CYCLE change takes its new value at the very beginning of the pwm period so obviously the DUTY_INCREASE or the DUTY_DECREASE are changed to 1. In this case, the initial DUTY_CYCLE is 3 and after a DUTY_INCREASE which was changed to one during the first pwm period, the DUTY_CYCLE is now changed to 4. Similarly, after a second DUTY_INCREASE the DUTY_CYCLE now goes to 5.

We also know that pwm is defined so that PWM_OUT =1 when counter_pwm is less than DUTY_CYCLE. And so by comparing these three signals in the simulation we see that it works properly. Indeed, PWM_OUT_O is at 1 during the first 3 clock periods (30% duty cycle) and switches from the moment when counter_pwm is equal to the DUTY_CYCLE. In the same way for the second period of the pwm, the DUTY_CYCLE has changed to 4 and we have PWM_OUT_O which is at 1 during the first 4 periods of the clock (counter_pwm from 0 to 3).

We can also observe that over the first 400 nanoseconds, direction_H is at 0 which means that the pwm signal is sent in PWM_OUT_A (anti clockwise) and therefore PWM_OUT_H is at 0 all along.

Finally, the last two are intended for the application and again everything is working properly. Indeed, LED_counter counts all clock periods where the pwm signal is equal to 1 and LED_OUT displays this counter.

- **DUTY_INCREASE**

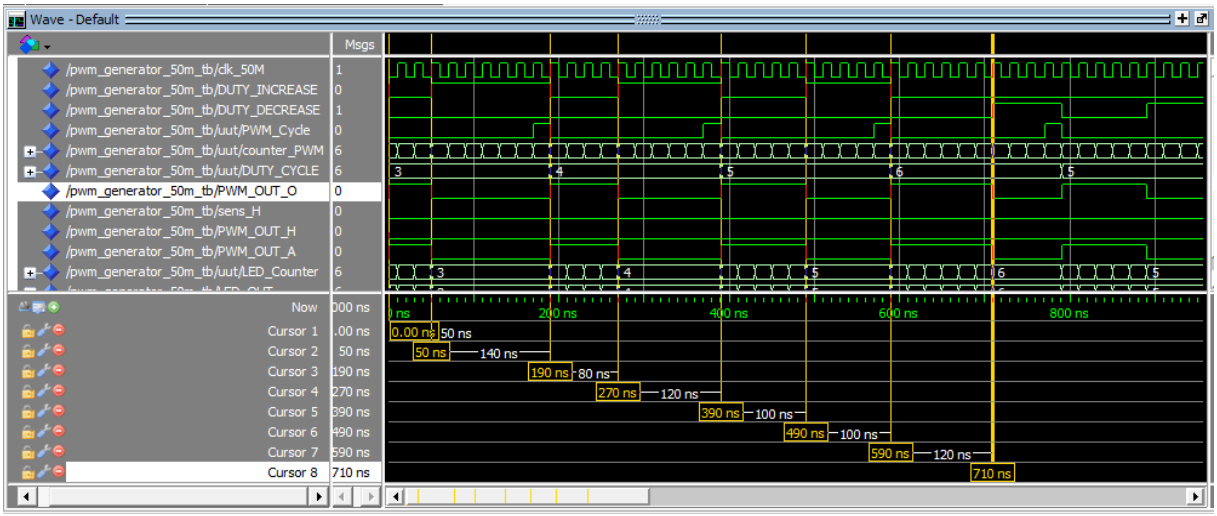


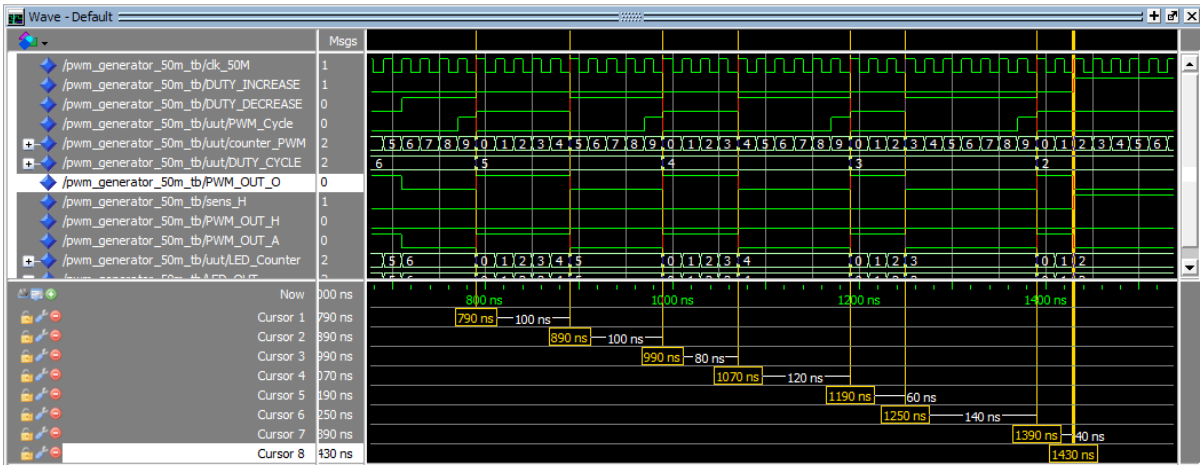

On this second simulation image, we see this time entirely the first 4 periods of the pwm signal (up to 800ns) and the beginning of the 5th period. We observe that during this time labs, there were 3 DUTY_INCREASE consecutively and that each time the DUTY_CYCLE increased by 1 at the beginning of the pwm period.

We placed markers at each rising and falling edge of the PWM_OUT_O signal in order to be able to measure the different widths of the pwm signal.

During the first period of the pwm, the width is 50ns. In reality, it should have been 60ns since the Duty cycle is 30% (DUTY_CYCLE=3). This difference can be explained by the fact that the clock starts with a half period and thus 10ns instead of its 20 nanosecond period. Then on the second period the pwm has a width of 80ns, the fourth a width of 100ns and the fifth a width of 120ns. This corresponds to a duty cycle of 40%, then 50% and finally 60%.

- **DUTY_DECREASE**

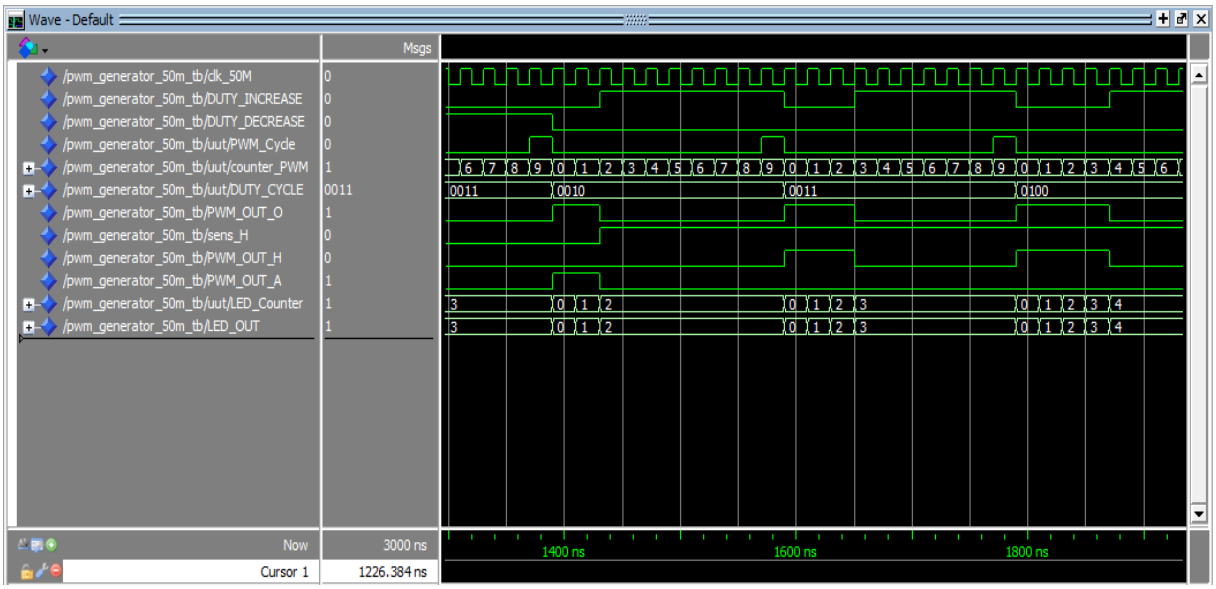



We can see from these two images that after the 3 DUTY_INCREASE, the test bench is followed by 4 DUTY_DECREASE. Once again the simulation shows us that everything works, after each DUTY_DECREASE the DUTY_CYCLE decreases by 1 and therefore the width decreases each time by 10%.

The first pwm period is 100ns wide (50%), the second is 80ns wide (40%), the third is 60ns wide (30%) and the fourth is 40ns wide (20%). This means that each time we have one period less and therefore a 10% reduction in the duty cycle.

- **Sens_H change**




In this image, we observe the first variation of direction_H in the test bench. We see that from this passage to 1 of direction_H the pwm signal is well sent in PWM_OUT_H while PWM_OUT_A is now all the time at 0.

- **Tmp1, tmp2, tmp3, tmp4 + slow_clk_en + counter_slow**

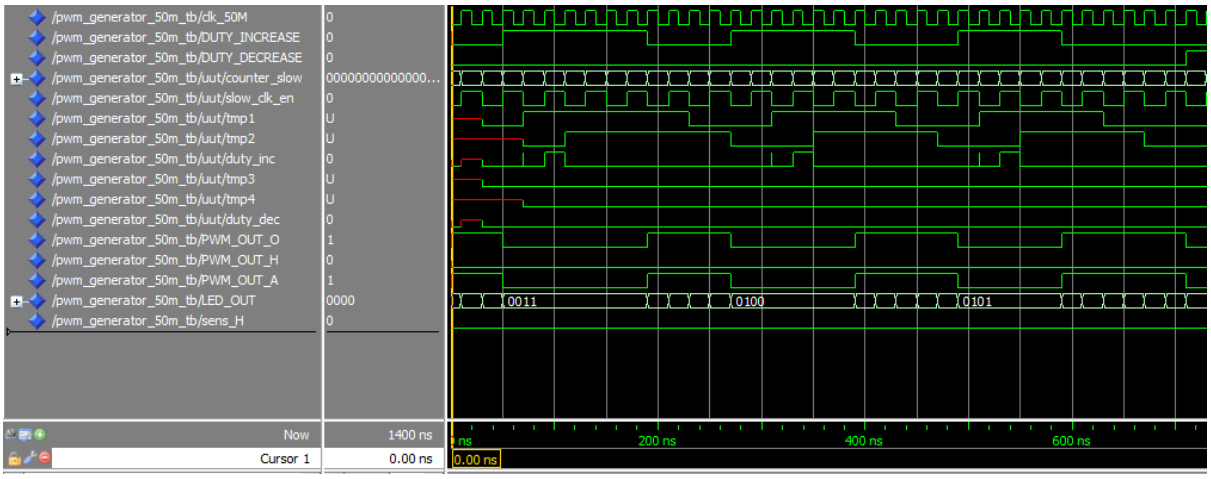
As explained above, the debounce function is designed to handle bounces when the DUTY_INCREASE and DUTY_DECREASE buttons are pressed.

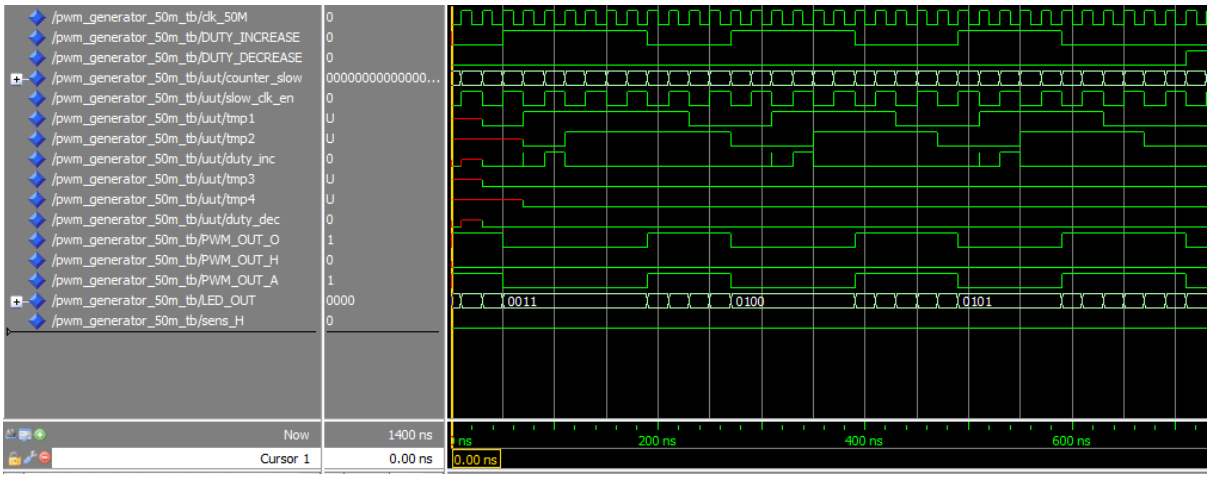
The following codes are intended to achieve this.

```
-- debounce part for duty increasing button( géré les antis rebond incrémentation)

stage0: DFF_Debounce port map(clk_50M,slow_clk_en , DUTY_INCREASE, tmp1);
stage1: DFF_Debounce port map(clk_50M,slow_clk_en , tmp1, tmp2);
duty_inc <=  tmp1 and (not tmp2) and slow_clk_en;

-- debounce part for duty decreasing button ( géré les antis rebond décrémentation)

stage2: DFF_Debounce port map(clk_50M,slow_clk_en , DUTY_DECREASE, tmp3);
stage3: DFF_Debounce port map(clk_50M,slow_clk_en , tmp3, tmp4);
duty_dec <=  tmp3 and (not tmp4) and slow_clk_en;
-- for controlling duty cycle by these buttons
```

It is desired that when the DUTY_INCREASE or DUTY_DECREASE button is pressed, only one activation occurs. To do this, we want that after the DUTY_INCREASE signal goes to 1 (pressing the botton), the duty_inc goes to 1 only during the high level of the slow_clock_en and between tmp_1 and tmp_2 (which differs from a clock duration of the slow_clock_en). The principle is the same for the duty_decrease.

We can see from these simulation captures that the debounce function works correctly. Indeed, duty_inc or duty_dec will each time switch to 1 during a slow_clock_up following the activation of DUTY_INCREASE or DUTY_DECREASE.



## Conclusion

We have adapted the code so that it can work with a 50MHz clock (corresponding to the DE1 Altera board) instead of the 100MHz initially planned. We have also made the necessary modifications to obtain the signals needed to control the DC motor clockwise and anticlockwise. Finally, we have implemented a small application to display in real time on the LEDs of the DE1 Altera board the number of clock cycles that composes the PWM signal width. Simulations of the project could be carried out by the Intel Quartus Prime software via ModelSim.

Unfortunately, as we no longer had access to the university's premises following the Covid-19 epidemic, we were unable to implement our code on the DE1 Altera board.

**Github Repository:** https://github.com/Jolan-Dubrulle/Driver-PWM-bidirectional-DC-Motor