Lab#6

– Digital Electronics –

D/A Conversion

Students:

Masson Nicolas s293826

Muhammad Arshad s288059

Martin Callegarin Demangeat s294064

Wojciech Paluszkiewicz s293958

In this lab we will create a simple Digital to Analogic converter with the PWM and PPM technologies. PWM stands for Pulse Width Modulation and is a signal with a constant frequency but with a varying duty cycle. PPM stands for Pulse Position Modulation and here, the length and the amplitude of the pulses stay constant but the position of each pulse depend on the amplitude of the modulation signal.

This Laboratory being based on the dialog between the nios system and the VHDL decoders, we will first initialize the hardware to make possible the dataflow of digital bit words from the nios system to the decoders. Then we will program both the PWM and PPM converters with VHDL. Since they can only output 1 and 0, the signals they output will have to be filtered by a low pass filter. Given the 2nd order low pass filter given in the lab documentation, we can assume it's selectivity as high enough to eliminate the harmonics present in a square signal, leaving only the fundamental frequency to give in fine an average analogic value.

1. Nios system.

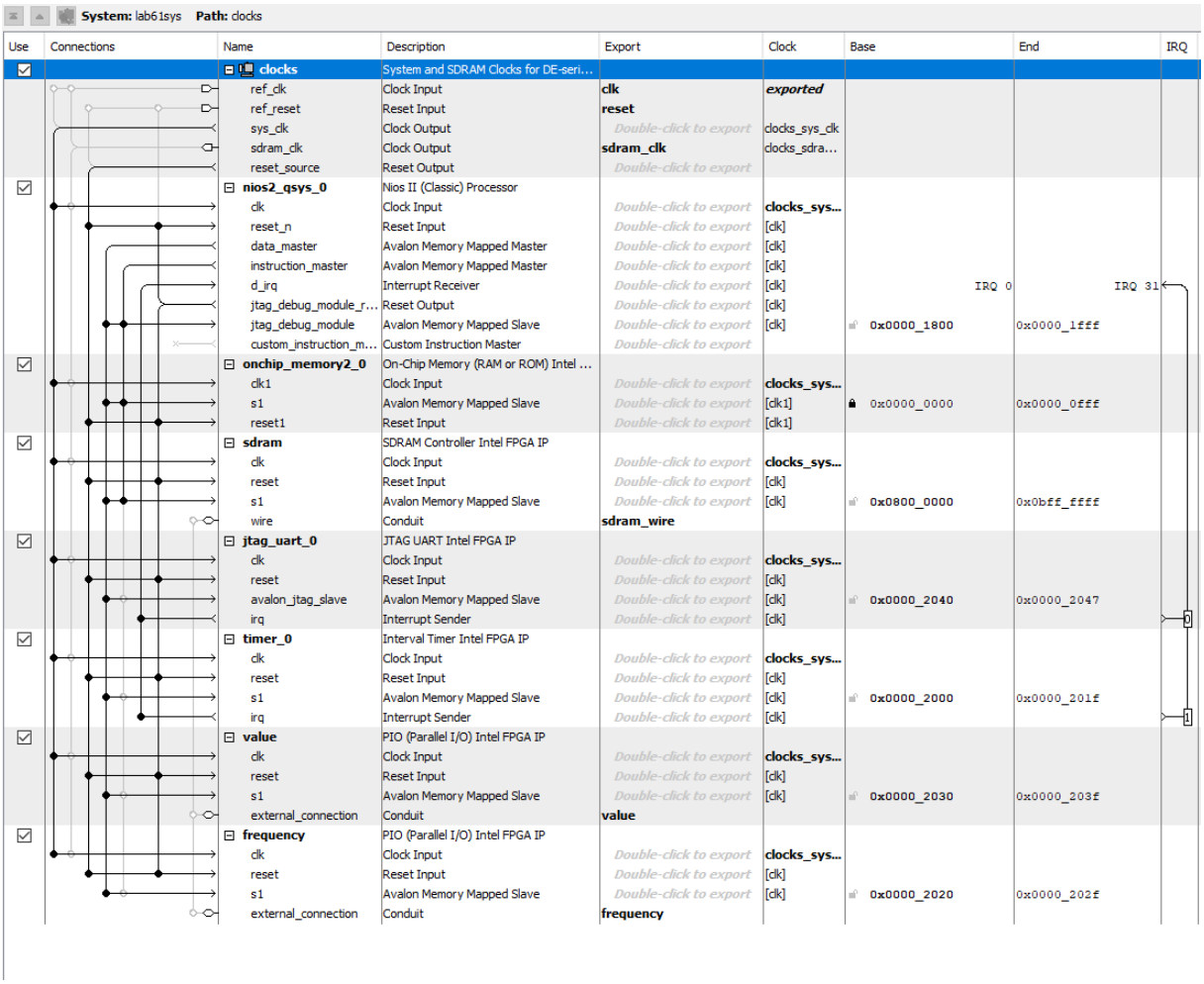


Figure 1: the final hardwiring of the used nios system.

As shown on figure 1 is the nios system that we chose to initialize with an additional SDRAM and a new clock based on PLL as it has been said to ensure reliability. (It has proven true as we will see)

The different peripheral port named value and frequency will be the link between the user on his laptop and the PWM, PPM decoders on the FPGA (for the mainly understandable data at least). There are still wirings to do in the top entity VHDL code. This is shown Figure 2. First there is the classic I/O from the FPGA that we are used to such as the system clock, a key for asynchronous reset, the 7 seg and led for display. Then there is also the GPIO_0. Those are physicals pins on the FPGA that we will drive to output the PWM and PPM signals coming out the converters. The value_export logic_vector from nios component is quite important as it is the digital value to be converted. We'll deepen this subject later on.

```
 6  entity dac is
 7  port(
 8      CLOCK_50     : in std_logic;
 9      KEY          : in std_logic_vector(0 downto 0);--resetn
10      HEX0,HEX1    : out std_logic_vector(6 downto 0);
11      LEDR         : out std_logic_vector(1 downto 0);
12      GPIO_0       : out std_logic_vector(1 downto 0);--the output to the filter
13      DRAM_DQ                          : inout std_logic_vector (15 downto 0);
14      DRAM_ADDR                        : out   std_logic_vector (12 downto 0);
15      DRAM_BA                          : out   std_logic_vector (1 downto 0);
16      DRAM_CAS_N, DRAM_RAS_N, DRAM_CLK : out   std_logic;
17      DRAM_CKE, DRAM_CS_N, DRAM_WE_N   : out   std_logic;
18      DRAM_UDQM, DRAM_LDQM             : out   std_logic
19      );
20  end dac;
21
22  architecture beh of dac is
23  --component declaration
24  component lab61sys is
25  port (
26          clk_clk          : in    std_logic                       := '0';
27          frequency_export : out   std_logic_vector(1 downto 0);
28          reset_reset      : in    std_logic                       := '0';
29          sdram_clk_clk    : out   std_logic;
30          sdram_wire_addr  : out   std_logic_vector(12 downto 0);
31          sdram_wire_ba    : out   std_logic_vector(1 downto 0);
32          sdram_wire_cas_n : out   std_logic;
33          sdram_wire_cke   : out   std_logic;
34          sdram_wire_cs_n  : out   std_logic;
35          sdram_wire_dq    : inout std_logic_vector(15 downto 0) := (others => '0');
36          sdram_wire_dqm   : out   std_logic_vector(1 downto 0);
37          sdram_wire_ras_n : out   std_logic;
38          sdram_wire_we_n  : out   std_logic;
39          value_export     : out   std_logic_vector(7 downto 0)
40      );
41  end component;
42
```

Figure 2.1 : the main hardware and nios instantiation.

```
75  --signals here
76  signal valuetoread : std_logic_vector(7 downto 0);--to break
77  signal freqbits : std_logic_vector(1 downto 0); --from nios to send to clock divider
78  signal newclock : std_logic; --from clock divider to send to pwm and ppm
79
80  begin--architecture
81
82  --component port map
83  niosII : lab61sys
84      port map(
85      clk_clk => CLOCK_50,
86      sdram_wire_addr   => DRAM_ADDR,
87      sdram_wire_ba     => DRAM_BA,
88      sdram_wire_cas_n  => DRAM_CAS_N,
89      sdram_wire_cke    => DRAM_CKE,
90      sdram_wire_cs_n   => DRAM_CS_N,
91      sdram_wire_dq     => DRAM_DQ,
92      sdram_wire_dqm(1) => DRAM_UDQM,
93      sdram_wire_dqm(0) => DRAM_LDQM,
94      sdram_wire_ras_n  => DRAM_RAS_N,
95      sdram_wire_we_n   => DRAM_WE_N,
96      sdram_clk_clk     => DRAM_CLK,
97      reset_reset       => not(key(0)),
98      frequency_export  => freqbits,--on 2bits
99      value_export      => valuetoread
00  );
01
```

Figure 2.2: the port map of the nios system and the signals used in the circuit

As we can see on figure 2.2, we need to write the frequency and value export to be written on additional wires. This is only due to the client demand to display the current digital value and the work frequency of the converters.

The new clock signal is, as his name mentions it, is the new (slower) clock based the system's 50MHz clock. As we must use a clock value based on the user's table depicted in the lab subject. We had to divide the 50MHz with a VHDL component presented figure 3.

```
 6  entity divider is
 7  port(
 8      clkin : in std_logic;
 9      div   : in std_logic_vector(1 downto 0);
10      clkout: out std_logic
11      );
12  end divider;
13
14  architecture beh of divider is
15  shared variable upto : integer range 0 to 999;
16  shared variable count : integer range 0 to 999 :=0;
17
18  begin
19
20  process(clkin)
21  begin--process
22      if(clkin'event and clkin ='1') then--at each rising edge...
23          case div(1 downto 0) is--...we increase counters
24              when "00" =>--400kHz => 125 clock cycle
25                  upto := 125-1;
26              when "01" =>--200kHz => 250 cc
27                  upto := 250-1;
28              when "10" =>--100kHz => 500 cc
29                  upto := 500-1;
30              when others =>--50 kHz => 1000 cc
31                  upto := 1000-1;
32          end case;
33          if(count >= upto) then --we reached max value
34              clkout <= '1';
35              count := 0;
36          else --we must keep counting
37              count := count + 1;
38              clkout <= '0';
39          end if;--count=upto
40      end if;--clkin
41  end process;
42
43  end beh;
```

Figure 3: clock divider.

We will then plug the clkin to CLOCK_50, the div to the 2-bit word given by user and clkout will be the newclock signal used by decoders.

This clock divider is based on strobe sending, the duty cycle of the new clock will be very low compared to the 50% of traditional digitals clocks. Every n clock cycle of the 50MHz we will send a strobe. Since the converters using this new clock are configured to work on rising edges of clocks, they don't really care about the rest. Let's talk about them. Figure 4 and 5 are respectively the VHDL code of PWM and PPM digital to analog converters.

```
 6  entity pwm is
 7  generic(
 8      N           : integer := 255);--limit value of the duty cycle
 9  port(
10      clk,rstn    : in std_logic;
11      D           : in integer;--value coming from nios and sinus.h file
12      pwmout      : out std_logic);--the logic signal of the pwn to send to filter
13  end pwm;
14
15  architecture beh of pwm is
16
17  shared variable Dcount : integer range 0 to 255 := D;
18  shared variable Ncount : integer range 0 to 255 := N;
19
20  begin--architecture
21
22  process(clk,rstn)
23  begin--process
24      if(rstn = '0') then
25          pwmout <= '0';
26          Dcount := D;
27          ncount := N;
28      else
29          if(clk'event and clk = '1') then--rising edge
30              if (ncount = 0) then--reset of counters
31                  ncount := N;
32                  dcount := D;
33                  else if(dcount > 0) then --must send 1 for the duty cycle duration
34                      pwmout <= '1';
35                      ncount := ncount - 1;
36                      dcount := dcount - 1;
37                      else if(dcount = 0) then
38                          pwmout <='0';
39                          ncount := ncount - 1;
40                      end if;
41                  end if;
42              end if;--ncount check
43          end if;--clock
44      end if;--reset
45  end process;
46  end beh;
```

Figure 4.1: PWM converter

As D is an 8bit word, we need a generic constant N equal to 255 acting as the boundary of value to reach. (by writing this report we realize the generic calling is useless as we could have just wrote Ncount … := 255; . We will put this on the account of flexibility)

Then as proposed in lab's subject, we create 2 counters that will decrement and the PWM output 1 as long as D isn't depleted. Once it is, the reset of both counter is done when Ncounter reaches 0. That will be the end of a period. The analog signal can be expressed as the mean value of the output voltage. Figure 4.2 is the output of the PWM before and after filtering. The sent value is 20 at working frequency of 100kHz

Figure 4.2: PWM signal in blue. Filtered one in purple.

```vhdl
entity ppm is
port(
      clk, rstn   : in std_logic;
      D           : in std_logic_vector(7 downto 0);--it will be signed then
      ppmout      : out std_logic
);
end ppm;

architecture beh of ppm is
constant b : integer := 2;--the number of added bit after the adding operation
constant n : integer := 8;--the normal size of incoming data. here 8
constant conc : std_logic_vector(0 downto 0) := "0"; --maybe useless
--signal declaration
signal acc_in : std_logic_vector(8 downto 0); --an n+1 size word
signal acc_out : std_logic_vector(n+b-1 downto 0);
signal feedback : std_logic_vector(7 downto 0);
signal msb : std_logic; --will always take the msb of acc_out


begin--arch

acc_in <= (conc & D) + (conc & feedback);--at every moment we have the addition made

process(clk,rstn)
begin--process
   if(rstn = '0') then
      ppmout <= '0';
      else
      if(clk'event and clk = '1') then
         acc_out <= acc_out + (conc & acc_in);
         msb <= acc_out(n+b-1);--taking the msb
         ppmout <= msb;
      end if;--clock
   end if;--reset check
end process;


with msb select feedback <= --multiplexing
   "10000000" when '1',-- equal to -128 when signed
   "01111111" when others;-- equal to +127

end beh;
```

Figure 5: PPM converter

The PPM functioning is rather more complex than the PWM. Based on the digital schema of the sigma depicted figure 6 we must have a constant addition from the feedback of the MSB and the data input. This is done for every moment at line 27 of the code. Note that the concatenation while being inelegant seems to be necessary for the VHDL syntax as there would be a format conflict with the addition. We aren't experts, just students trying to get a code working.

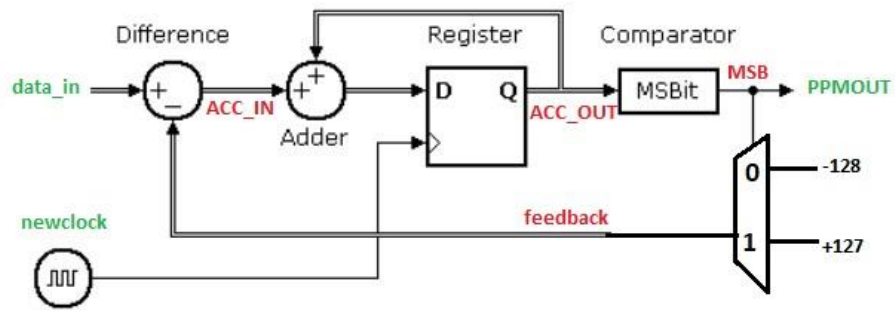


Figure 6: sigma delta functioning adapted with our signals denominations

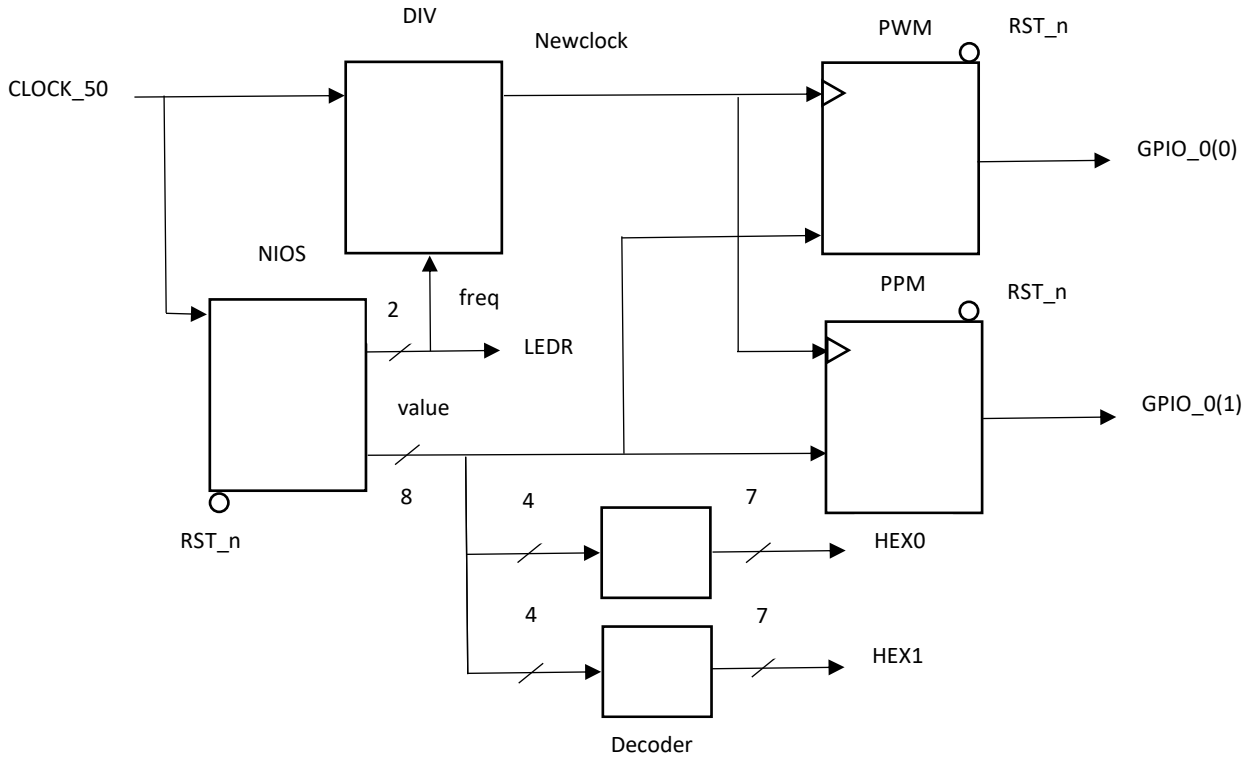
Since the feedback and data_in are 8bit words we must consider the ACC_IN signal on 8+1 bits. The same reasoning is to be done when adding the register ACC_OUT signal with ACC_IN, that is why we put ACC_OUT on 8+2 bits. Rest of the processing is extracting the MSB out of ACC_OUT and sending it out for the PPM signal output. We also need to implement the multiplexer choosing the adequate feedback to the first adder.

```
113    deevee : divider
114    port map(
115        clkin => CLOCK_50,
116        div => freqbits(1 downto 0),
117        clkout => newclock
118    );
119
120    pewoum : pwm
121        port map(
122        clk => newclock,
123        rstn => key(0),
124        D => to_integer(unsigned(valuetoread)),
125        pwmout => GPIO_0(0)
126    );
127
128    peepeem : ppm
129        port map(
130        clk => newclock,
131        rstn => key(0),
132        D => valuetoread, --no need to cast as we want to have signed std_logic_vectors
133        ppmout => GPIO_0(1)
134    );
135
136    LEDR(1 downto 0) <= freqbits(1 downto 0);
137
138    end beh;
139
```

Figure 7: some of the port map used.

The hexadecimal displaying being part of previous lab, we won't come back on the 4 to 7 bits decoder.

Here is the block scheme of our program :



2. C programming

The first program is supposed to ask the user any value between 0 and 255 as well as the frequency he wants from the menu given. Therefore, he'll have to write 00 to have f0 at 400 kHz, 01 to have it at 200kHz and so on so forth…

We created the code without any verification made on the input from the user; we assume he knows what he is doing.

First step was to implement the necessary libraries such as <stdio.h> in order to use scanf command. The code doesn't need any interruption for the moment so it is rather simple, basic.

```
13  int main()
14  {
15      //init_timer();
16      //i = 0;//index of the array for reading sinus.h
17      int freq ;//on 2 bits
18      int val;// on 8 bits, we'll mask both freq and val when sending
19      printf("what value?");
20      scanf("%d",&val);
21      printf("what freq?");
22      scanf("%d",&freq);
23      IOWR_ALTERA_AVALON_PIO_DATA(FREQUENCY_BASE,freq & 0x03);
24      IOWR_ALTERA_AVALON_PIO_DATA(VALUE_BASE,val & 0xff);
25
26      while (1);
27      return 0;
28  }
```

Figure 8: The C code for sending once the frequency and value wished.

As said on figure 8 line 18, we must mask the sent data on peripheral as a measure of reliability. Even If the user sends something inappropriate, the Hardware must receive something in the correct format.

After what the user's job is done since the nios will send these data to the peripherals and the FPGA's hardware will treat those in continuous, displaying the value in hexadecimal on the 7 segments and the frequency 2 bits code on the 2 leds. But most importantly, the PWM and PPM being also wired to these data, they will output the analog signal in consequence, and we can retrieve the two signals from the GPIO_0 pins.

The second program must implement the timer initialization and the timer interruption subroutine. This is done in 2 separated voids. The timer initialization is called in the main function. This was in comment at line 15. The function being called is shown figure 9. The 2nd void (a static void) will then be called at every interruption created by the timer.

```
43  void init_timer()
44  {
45      /* we want an interrupt every 10 millisec for 100Hz
46       * with clock at 50MHz to have 100Hz it's every 500k cycles 500.000 = 0x 0007 A120
47       */
48    IOWR_ALTERA_AVALON_TIMER_PERIODL(TIMER_0_BASE, 0xA120);
49    IOWR_ALTERA_AVALON_TIMER_PERIODH(TIMER_0_BASE, 0x0007);
50    int test=alt_ic_isr_register(TIMER_0_IRQ_INTERRUPT_CONTROLLER_ID, TIMER_0_IRQ,timer_isr, NULL,NULL);
51    if (test == 0)
52      printf("Timer Interrupt Routine Registered\n");
53    IOWR_ALTERA_AVALON_TIMER_CONTROL(TIMER_0_BASE, ALTERA_AVALON_TIMER_CONTROL_ITO_MSK |
54      ALTERA_AVALON_TIMER_CONTROL_CONT_MSK | ALTERA_AVALON_TIMER_CONTROL_START_MSK);
55  }
```

Figure 9: Timer initialization.

If we want an interruption to occur at 100Hz with an intern clock frequency of 50MHz, then we have to code the period of interruption to be of 500k clock cycles. 500k translated in hexadecimal is 0x7A120. Line 48 and 49 of figure 9 is coding the period of timer.

Then we instruct the timer to start, work in continuous and generate interrupts. This is done line 53 and 54 by using the macro mask, the compiler knows which bit of which register of the timer to turn to 1.

Once the timer is set, we must program the interruption subroutine. Have a look at figure 10. The goal is to send data from the sinus.h file in a cycle. The variable i is declared as volatile integer.

```
29  static void timer_isr(void *context)
30  {
31      //clearing interrupt
32      IOWR_ALTERA_AVALON_TIMER_STATUS(TIMER_0_BASE,0x00);
33      //what do we want at each interrupt from timer?
34      if(i == 20)
35      {
36          i = 0;
37      }
38      IOWR_ALTERA_AVALON_PIO_DATA(VALUE_BASE,samples[i] & 0xff);
39  //  IOWR_ALTERA_AVALON_PIO_DATA(VALUE_BASE,val & 0xff);
40      i ++;
41  }
```

Figure 10: Timer interrupt subroutine to send data from sinus.h file to the converters.

After implementing the sinus.h file in the right folder and declaring it in the first lines of code, we can extract the values like in an array. (Here the array is called “samples”) rest of code is sending data to converters and surveying rank of the array value which is i.

The result on the output of the converters pins isn’t visibly interesting as it is a square signal changing at 100Hz. However, after implementing an appropriate filter, we have interesting results for the eyes. (Well, it’s just a sinus wave but it’s the result of a long journey that’s why it’s rewarding)

The period is 200ms which is expected since we have 20 samples in the array and one sample is sent every 10ms. The amplitude from the GPIO_0 output is roughly 4.25 V without the probe amplification and the sinus amplitude is roughly 3V.

Figure 11: sinus wave collected from the PWM converter and after filter.

###TODO###

upload the photos of the different value @ different frequencies and from different converters.

Talk about the measuring and conclude

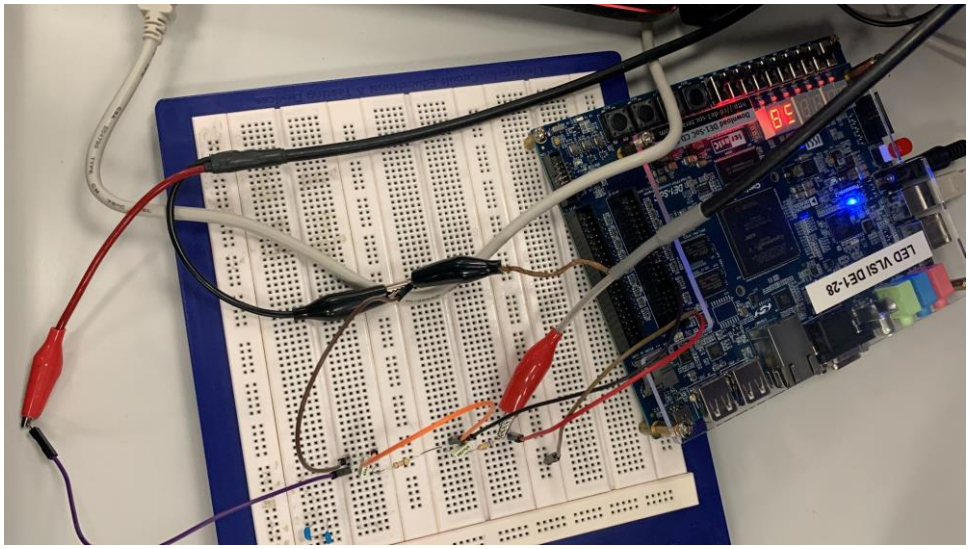
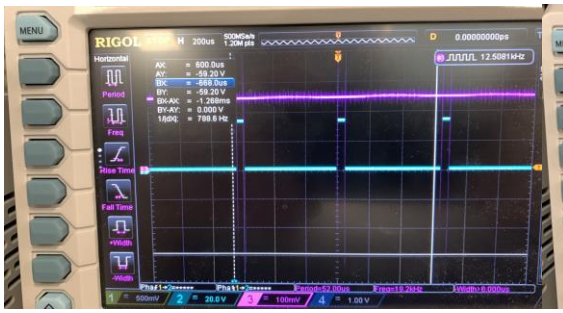


Figure 12 : 2nd order low-pass filter circuit

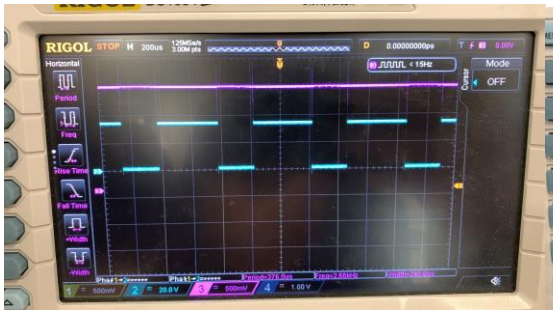
We built the 2nd order low pass filter shown in the instructions and and we connected the output of each converter to the filter. We display on the oscilloscope the output of each converter by changing the value of f0 and the data value. So, we did 24 measurements but we just put the most relevant ones.

For PPM :



Data value=20 and f0=400kHz

Data value=20 and f0=50kHz

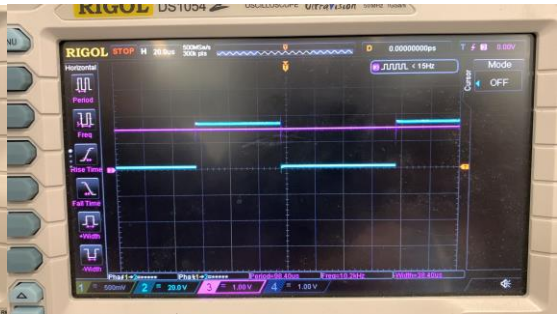


Data value=160 and f0=400kHz

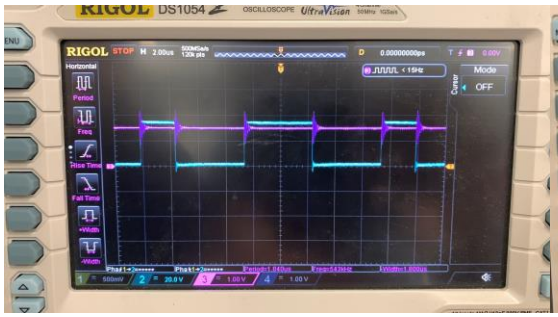
Data value=160 and f0=50kHz

For PWM :

Data value=20 and f0=400kHz



Data value=20 and f0=50kHz



Data value=160 and f0=400kHz

Data value=160 and f0=50kHz

First, we noticed that for any f0, the amplitude of the GPIO_0 is equal to 2.860v.

Oscilloscope observations:

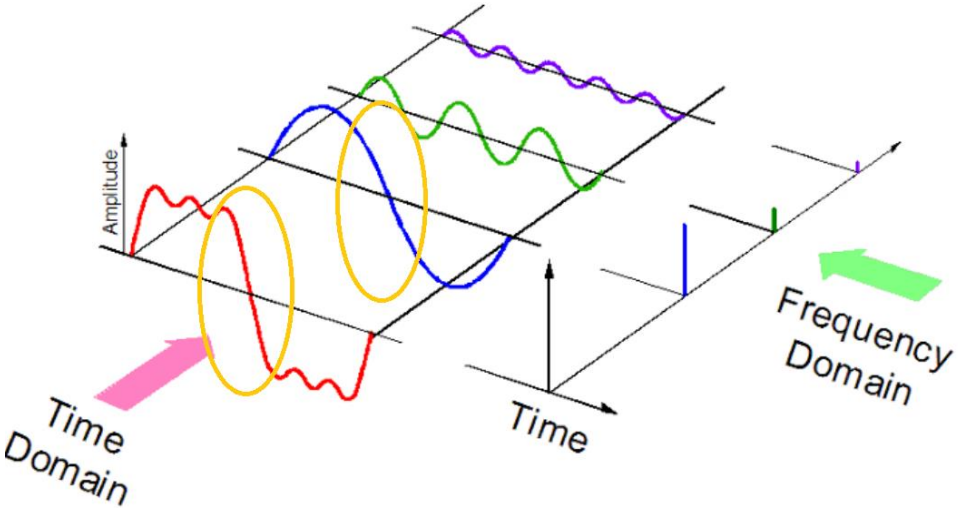
PPM

This approach is more complex, because we are operating on exact numbers with comparator inside. Signals for the high f from filter is stable, because harmonics are active for the greater f than filter passes. Whereas, if f is lower, the sinus shape is clearly seen.

The less samples we take, the more noise is in our filtered signal.

PWM

Moreover, for PWM only lower frequencies are active with high amplitude (purple signal after LOW-pass filter), when converter's output is changing values. This behavior is describedin the graphic below:



While changing the value from logic "1" to "0", mainly we are using only a low frequency band.

PWM converter transmits only average value and discard all harmonic components, that is why the filtered signal is stable in logic "1" or "0".

To conclude, we observed that both PWM and PPM are highly immune toward noise. The most significant difference is that for PWM method a synchronization between transmitter and receiver is needed but not for PPM method. We also concluded that transmission power is variable for PWM because of variation in amplitude and width. On the other hand, the width and the amplitude are constant for PPM so the transmission power is constant.