Sviluppo di una piattaforma general-purpose basata su Zynq-7000 per il controllo e la caratterizzazione di convertitori elettronici di potenza

Platania Gabriele, Ziviani Pietro, Voltarel Nicola, Pennacchio Francesco

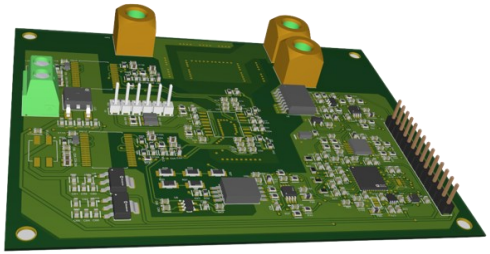
[1]Department of Management and Engineering (DTG) – Vicenza

Relatore: Biadene Davide

Co-relatore: Caldognetto Tommaso

Descrizione del progetto

L'obiettivo di questo progetto è la realizzazione di una piattaforma di controllo avanzata per convertitori di potenza, basata sul dispositivo Zynq-7000 di Xilinx (AMD). Tale sistema sfrutta l'integrazione di una sezione di elaborazione (Processing System – PS) e di logica programmabile (Programmable Logic – PL) all'interno di un unico System on Chip (SoC), permettendo di unire la flessibilità dell'FPGA con le capacità di calcolo del microprocessore.



Obiettivi del progetto:

- Generazione di segnali digitali per il controllo dei convertitori
- Dimensionamento e simulazione del circuito di condizionamento basato su amplificatore operazionale per strumentazione
- Implementazione interfaccia di comunicazione SPI
- Acquisizione di un segnale analogico tramite l'XADC
- Implementazione Filtri Digitali IIR e FIR

1222·2022 800 ANNI

UNIVERSITÀ DEGLI STUDI DI PADOVA

DPWM

Platania Gabriele

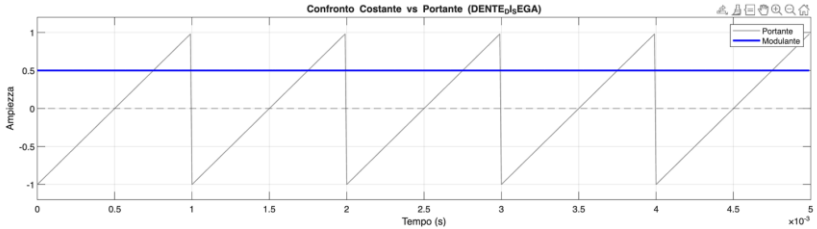
DPWM

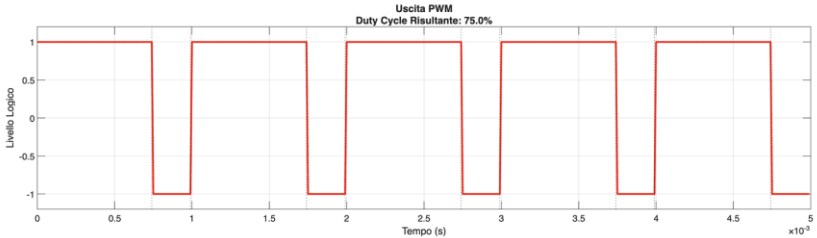
Introduzione al PWM

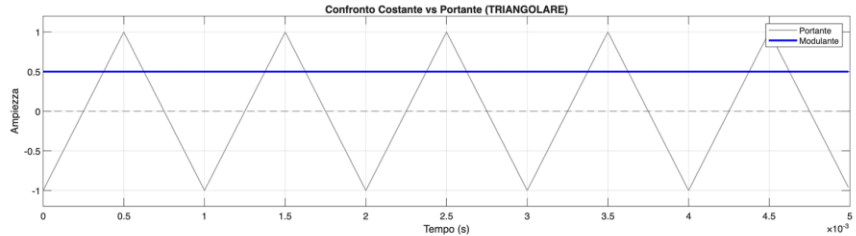
La PWM (Pulse Width Modulation) è una tecnica di controllo digitale che consiste nel generare un'onda quadra la cui durata dell'impulso in un periodo T è modificabile. In generale con questa tecnica si vuole regolare la potenza media da fornire verso un carico in un periodo. Le applicazioni sono molteplici, ma le più ricorrenti sono la tecnica di controllo dei motori e di convertitori elettronici di potenza, che utilizzano questo tipo di segnale per pilotare gli interruttori elettronici (MOSFET, IGBT…).

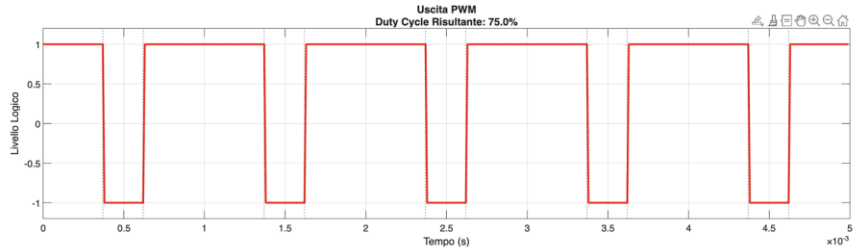
In genere l'onda è frutto di una comparazione tra due segnali principali detti **onda portante e modulante** dove la seconda stabilisce fondamentalmente la larghezza dell'impulso.

$$\delta = \frac{ton}{T} * 100$$









Generazione di segnale PWM digitale (DPWM) in VHDL

In questo caso specifico la comparazione è eseguita tra una portante realizzata con un contatore binario a 8 bit e un valore fisso rientrante nel range del contatore, anch'esso binario. Si può notare anche la differenza in termini di frequenza del segnale modulato nel caso di portante a dente di sega oppure onda triangolare.

$Qmax = valore\ massimo\ contatore\ impostato$

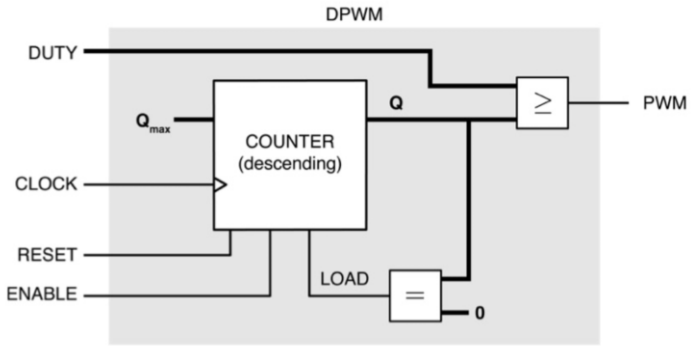
$Duty = valore\ fisso\ modulante$

$f_{clk} = freq\ clk\ principale$

$$\delta = \frac{Ton}{T_{pwm}} = \frac{Duty}{Qmax+1} \Rightarrow \Delta_{\delta\_\min} = \frac{1}{\mathrm{Qmax}+1} = 0{,}39\%$$

$$f_{pwm\_saw} = \frac{f_{clk}}{Qmax}$$

$$f_{pwm\_triang} = \frac{fclk}{2*\mathrm{Qmax}}$$



Esempi con onda triangolare e fclk 100Mhz

$$\mathrm{Qmax} = 2^8 = 256 \Rightarrow \frac{100 * 10^6}{2 * 256} = 193{,}5\ \mathrm{kHz}$$

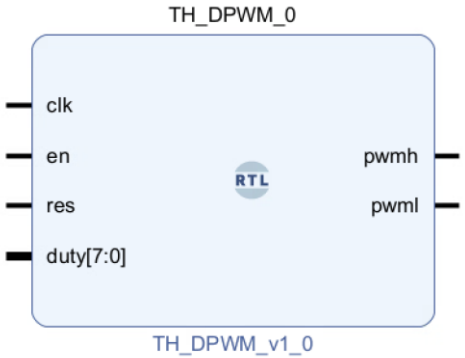
$$\mathrm{Qmax} = 100 \Rightarrow \frac{100 * 10^6}{2 * 100} = 500\ \mathrm{kHz}$$

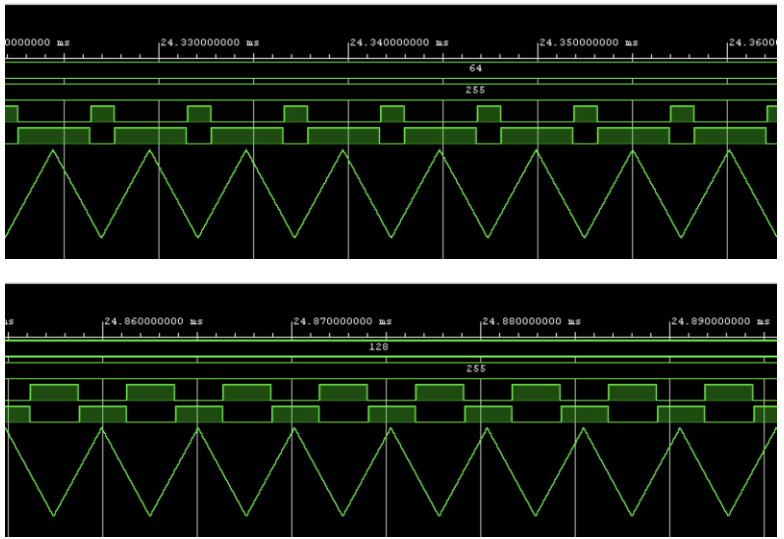
$$\mathrm{Qmax} = 50 \Rightarrow \frac{100 * 10^6}{2 * 50} = 1\ \mathrm{MHz}$$

Implementazione e simulazione

Si è dunque realizzata un'entity in VHDL per sintetizzare il modulatore PWM, avente in ingresso i segnali mostrati in figura, è dunque possibile effettuare un preset del contatore nel momento dell'utilizzo del segnale load, e soprattutto impostare il valore di duty cycle, esso non è altro che il segnale modulante precedentemente citato.

In particolare è stata aggiunta un'uscita PWM negata rispetto a quella principale al fine di testare la funzionalità del dead time implementata tra le due forme d'onda.





Sviluppo di un digital Pattern generator basato su ROM

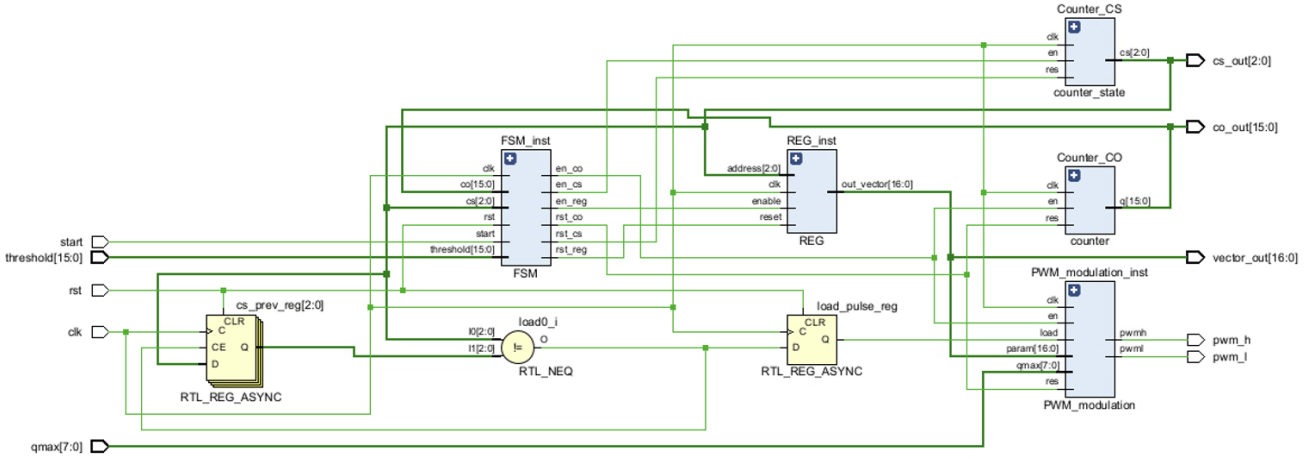
Platania Gabriele

Sviluppo di un digital Pattern generator basato su ROM

Introduzione e struttura

L'obiettivo era quello di generare dei segnali pwm temporizzati secondo un periodo noto dove per ognuno dei quali vengono modificati alcuni parametri caratteristici dell'onda, quali duty cycle e fase. La struttura principale è formata da più entity collegate, al fine di avere una struttura modulare e più comodamente editabile.

Il meccanismo alla base è rappresentato da una macchina a stati finiti la quale, con l'integrazione di 2 contatori punta a specifici indirizzi di memoria sulla ROM.



Elenco delle parti:

- TOP
- FSM
- ROM
- Counter Principale 8 bit.
- Counter State 3 bit.
- DPWM

Sviluppo di un digital Pattern generator basato su ROM

ROM

L'unità di memorizzazione dei parametri al momento è una Read Only Memory avente 8 (2^3) indirizzi di memoria (uno per ogni stato), ai quali la FSM punta tramite il counter state. Il contenuto è essenzialmente formato da una parola a 16 bit divisa in 2. Dunque 8 bit per indicare il valore del duty cycle e nei secondi 8 la fase rispetto all'inizio del periodo di stato.

ROM CONTENT MAP (q_{max} = 100)

ADDRESS (3 bit)	DATA WORD (17 Bit)			FUNCTION DESCRIPTION
	DUTY CYCLE [16:9] (8 bit)	DIR [8]	PHASE PRESET [7:0] (8 bit)	
000	00000000	0	00000000	IDLE (Off)
001	00011001	0	00000000	DUTY: 25%
010	00110010	0	00000000	DUTY: 50%
011	01001011	0	00000000	DUTY: 75%
100	01100100	0	00000000	DUTY: 100%
101	00110010	0	00110010	Phase Shift 90° (Start @ Half UP)
110	00110010	1	01100100	Phase Shift 180° (Start @ PEAK)
111	00110010	1	00110010	Phase Shift 270° (Start @ Half DW)

```
architecture Behavioral of REG is

type ROM is array (0 to 2**N - 1) of std_logic_vector(M-1 downto 0);
    -- Structure: [16:9] Duty (8b) | [8] Dir (1b) | [7:0] Phase Preset (8b)

    constant Patt_1 : ROM := (
        -- 1. IDLE: 0% Duty, 0° Phase
        -- Duty = 0, Phase = 0, Dir = UP
        "00000000" & "0" & "00000000",
        -- 2. LOW POWER: 25% Duty, 0° Phase
        -- Duty = 25 (dec) -> "00011001" (bin)
        "00011001" & "0" & "00000000",
        -- 3. HALF POWER: 50% Duty, 0° Phase
        -- Duty = 50 (dec) -> "00110010" (bin)
        "00110010" & "0" & "00000000",
        -- 4. HIGH POWER: 75% Duty, 0° Phase
        -- Duty = 75 (dec) -> "01001011" (bin)
        "01001011" & "0" & "00000000",
        -- 5. FULL POWER: 100% Duty, 0° Phase
        -- Duty = 100 (dec) -> "01100100" (bin)
        "01100100" & "0" & "00000000",
        -- 6. PHASE SHIFT 90° (Lagging): 50% Duty
        -- Duty = 50.
        -- Phase: Triangle is at 50 going UP (halfway up) -> 90 degrees shift.
        -- Preset = 50 ("00110010"), Dir = UP ('0')
        "00110010" & "0" & "00110010",
        -- 7. PHASE SHIFT 180° (Opposite): 50% Duty
        -- Duty = 50.
        -- Phase: Triangle starts at PEAK (100) going DOWN -> 180 degrees shift.
        -- Preset = 100 ("01100100"), Dir = DOWN ('1')
        "00110010" & "1" & "01100100",
        -- 8. PHASE SHIFT 270° (-90°): 50% Duty
        -- Duty = 50.
        -- Phase: Triangle is at 50 going DOWN (halfway down) -> 270 degrees shift.
        -- Preset = 50 ("00110010"), Dir = DOWN ('1')
        "00110010" & "1" & "00110010"
    );
```

Interazione DPWM e ROM

Per effettuare la modulazione, i parametri vengono messi a disposizione al blocco DPWM già introdotto all'inizio. Esso in ogni stato preleva le due parti della parola binaria. I primi 8 bit sono impostati come DUTY, ovvero segnale modulante, mentre i secondi vengono essenzialmente impostati come load del contatore, in modo da caricare un preset sulla portante e modificare la fase dell'onda in uscita dopo la comparazione.

$$Ticks = 2*qmax \qquad \text{Ticks} = \frac{\varphi}{2\pi}*2*qmax = \frac{\varphi*\text{qmax}}{\pi}$$

$$\text{Se } 0 < \varphi < \pi \Longrightarrow \text{Contatore in salita e Preset} = \frac{\varphi}{\pi}*qmax$$

$$\text{Se } \pi < \varphi < 2\pi \Longrightarrow \text{Contatore in discesa e Preset} = \frac{2\pi\text{-}\varphi}{\pi} * qmax$$

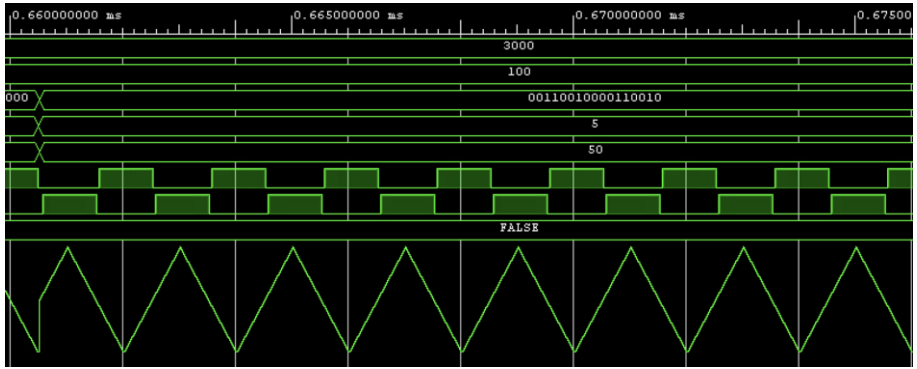
Il valore del preset del contatore non basta come unica informazione per impostare la quantità di sfasamento, bensì è necessario specificare qual è il verso di conteggio, essendo l'onda portante triangolare.
Di conseguenza, viene aggiunto un bit per definire questo aspetto e la ROM fornisce quindi 17 bit complessivi.

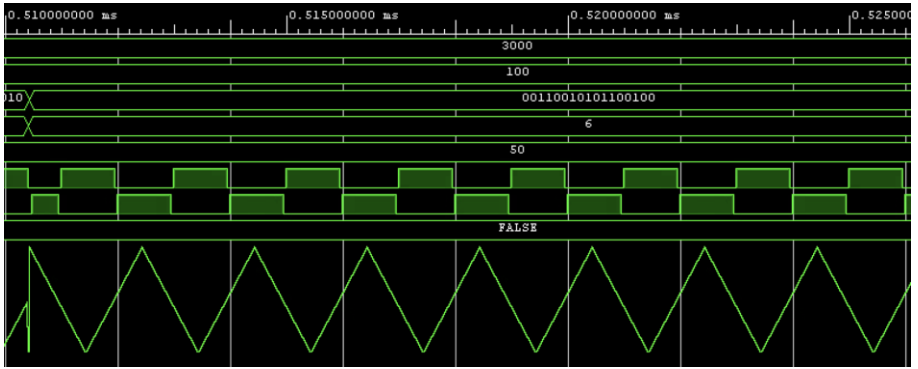
Sviluppo di un digital Pattern generator basato su ROM

SIMULAZIONI e RISULTATI

Viene eseguito dunque un testbench di quanto introdotto e viene verificato il corretto precaricamento del contatore e la variazione del duty cycle in ogni rispettivo stato. Si specifica che durante le prove è stato impostato anche un valore di dead time impostato ad 1 ciclo di clock, quindi 10ns, tuttavia il valore è impostabile intervenendo nel blocco PWM_modulation, sempre in valori scalati rispetto a Tclk.

Esempio di transizione tra stato 5 e 6 con variazione di fase $\frac{\pi}{2}$ e $\frac{3}{2}\pi$, dove si può notare la variazione nel preset del contatore.





1222·2022 800 ANNI

UNIVERSITÀ DEGLI STUDI DI PADOVA

Dimensionamento e simulazione del circuito di condizionamento basato su amplificatore operazionale per strumentazione

Ziviani Pietro

Dimensionamento e sviluppo INA

SPECIFICHE DI PROGETTO:

Funzioni richieste:

1. **Attenuazione:** 10V → 1V (guadagno G = 0.1)
2. **Shift di livello:** ±5V → 0-1V (offset +0.5V)

Specifiche tecniche dello XADC:

Risoluzione: 12 bit

Frequenza di campionamento: 1 MSPS

Range di ingresso:

- Modalità unipolare: 0.0V a +1.0V
- Modalità bipolare: -0.5V a +0.5V

Requisito di banda del condizionamento:

- Teorema di Nyquist-Shannon: f_signal_max = f_sampling / 2
- Banda massima richiesta: 500 kHz

Dimensionamento e sviluppo INA

I requisiti che deve soddisfare l'INA sono:

PARAMETRO	SPECIFICA	MOTIVAZIONE
Alimentazione	Deve gestire ±5V in ingresso	Alimentazione di ingresso fornita
Drift termico	<2 μV/°C	Stabilità su ΔT = 25°C
Banda passante	≤ 500 kHz	Rispettare Nyquist
Configurazione	Guadagno < 1	Attenuazione richiesta

I modelli di INA possibili sono: INA128, INA333 e INA826.

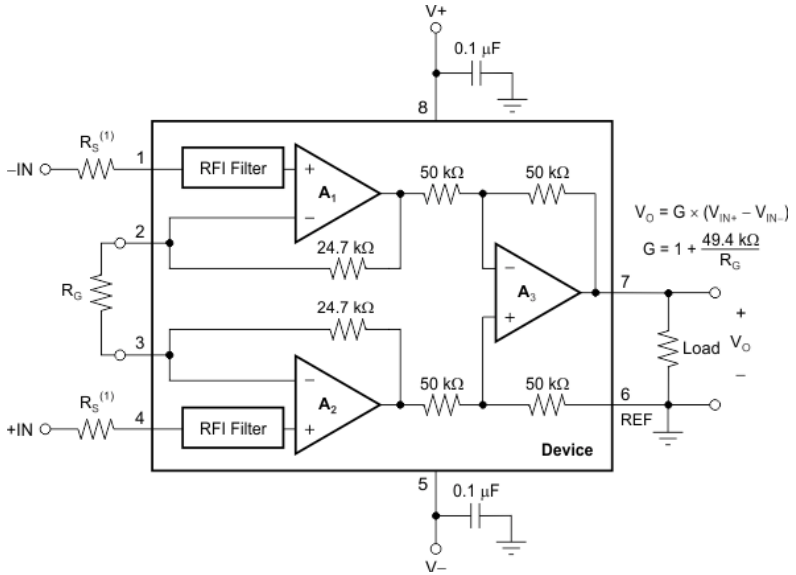
Dimensionamento e sviluppo INA

Mettendo a paragone le diverse INA, scopro che:

MODELLO	ALIMENTAZIONE	OFFSET MAX	BANDA A G=1	ADATTO?
INA128	±4.5V-±18V	150 µV	200 kHz	No, banda limitata
INA333	1.8V-5.5V	25 µV	350 kHz	No, input range inadeguato e banda limitata
INA826	±1.5V-±18V	150 µV	1 MHz	Scelto

INA826

In particolare verrà usata una INA826AID, questa presenta:



VANTAGGI:

- Banda: 1MHz con G = 1
- Offset: 150µV
- Drift: 0,25µV/°C
- Input range: ±5V viene soddisfatto

SVANTAGGI:

- Richiede alimentazione duale
- Più costoso rispetto ad altre INA come per esempio la INA128

Dimensionamento e sviluppo INA

Calcolo dei Componenti

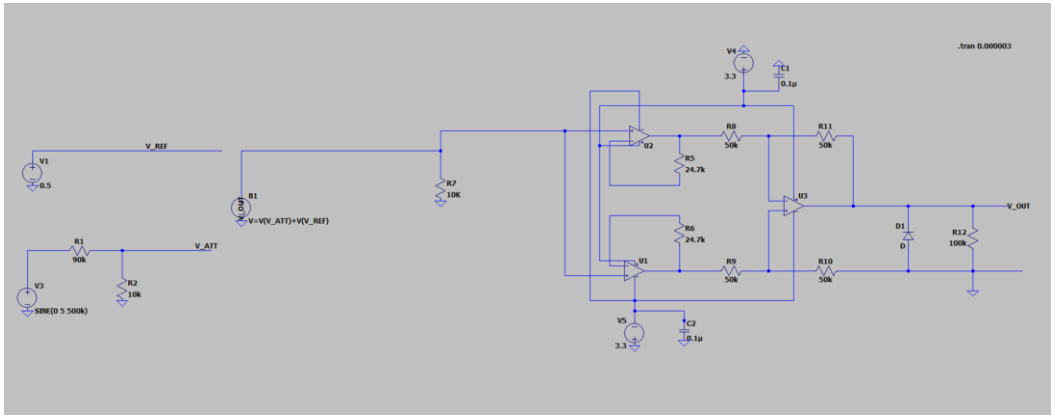
STADIO 1: Partitore Resistivo

Fattore attenuazione richiesto: k = 0.1

Formula: V_ATT = V_IN × (R2 / (R1 + R2))

Scelto R2 = 10 kΩ, calcolo:

R1 = R2 × (1/k - 1) = 10k × 9 = 90 kΩ



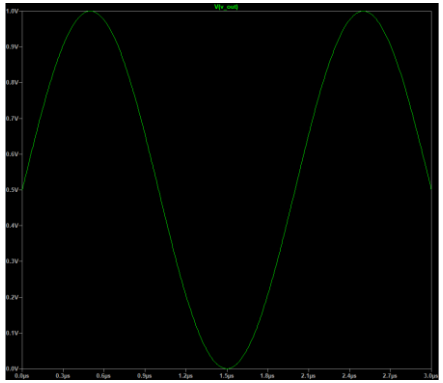
STADIO 2: Configurazione INA826

- **Guadagno:** G = 1 (pin RG lasciati aperti.)
- **Alimentazione:** V+ = +3,3V, V- = -3,3V
- **Pin REF:** collegato a generatore V_REF = 0.5V

GENERATORE V_REF (0.5V):

Partitore da +15V con buffer a bassa impedenza:

- R3 = 30 kΩ (0.1% tolleranza)
- R4 = 1 kΩ (0.1% tolleranza)
- Condensatore bypass: C1 = 100 nF



Implementazione interfaccia di comunicazione SPI

Ziviani Pietro

Macchina a Stati Finiti - 2 Stati

Stato IDLE

Attende che SS venga attivato (ss_n = '0')

Carica tx_data nel registro shift_out

Mette subito il primo bit (MSB) su MISO

Transizione → SHIFT quando ss_n = '0'

Stato SHIFT

Campiona MOSI su fronte di salita di SCLK

Aggiorna MISO su fronte di discesa di SCLK

Conta i bit trasmessi/ricevuti (8 bit totali)

Transizione → IDLE quando ss_n = '1', segnala rx_valid

Implementazione interfaccia di comunicazione SPI

Definizione delle Porte

```
entity spi_slave is
    Port (
        clk       : in  std_logic;        -- clock interno del PL
        reset     : in  std_logic;
        -- segnali SPI dal PS
        ss_n      : in  std_logic;        -- Chip Select attivo basso
        sclk      : in  std_logic;        -- clock generato dal PS
        mosi      : in  std_logic;        -- dato PS → PL
        miso      : out std_logic;        -- dato PL → PS
        -- dati ricevuti
        rx_data   : out std_logic_vector(7 downto 0);
        rx_valid  : out std_logic;
        -- dati da trasmettere
        tx_data   : in  std_logic_vector(7 downto 0)
    );
end spi_slave;
```

clk: Clock interno veloce (100 MHz) per sincronizzazione robusta

sclk: Clock SPI più lento dal master

Dichiarazione Segnali

```
architecture Behavioral of spi_slave is
    type spi_state_type is (IDLE, SHIFT);
    signal state : spi_state_type := IDLE;

    signal bit_cnt  : integer range 0 to 7 := 7;
    signal shift_in  : std_logic_vector(7 downto 0) := (others => '0');
    signal shift_out : std_logic_vector(7 downto 0) := (others => '0');
    signal sclk_prev : std_logic := '0';
```

```
-- rilevamento fronti di SCLK
sclk_prev <= sclk;

case state is
    when IDLE =>
        miso <= '0';
        bit_cnt <= 7;

        if ss_n = '0' then
            -- Inizia transazione SPI
            shift_out <= tx_data;
            miso      <= tx_data(7);  -- mette subito il primo bit
            state     <= SHIFT;
        end if;

    when SHIFT =>
        if ss_n = '1' then
            -- Fine transazione, segnala dato ricevuto
            rx_valid <= '1';
            rx_data  <= shift_in;
            state    <= IDLE;
        else
            -- Campiona MOSI sul fronte di salita
            if sclk_prev = '0' and sclk = '1' then
                shift_in(bit_cnt) <= mosi;
            end if;

            -- Aggiorna MISO sul fronte di discesa
            if sclk_prev = '1' and sclk = '0' then
                if bit_cnt > 0 then
                    bit_cnt <= bit_cnt - 1;
                    miso <= shift_out(bit_cnt - 1);
                else
                    -- Ultimo bit trasmesso, resta a bit_cnt=0
                    bit_cnt <= 0;
                end if;
            end if;
```

Implementazione interfaccia di comunicazione SPI

Gestione Reset e Stato IDLE

```
process(clk, reset)
begin
    if reset = '1' then
        -- Reset asincrono
        state     <= IDLE;
        bit_cnt   <= 7;
        shift_in  <= (others => '0');
        shift_out <= (others => '0');
        sclk_prev <= '0';
        rx_valid  <= '0';
        miso      <= '0';
```

```
elsif rising_edge(clk) then
    -- default
    rx_valid <= '0';

    -- rilevamento fronti di SCLK
    sclk_prev <= sclk;

    case state is
        when IDLE =>
            miso <= '0';
            bit_cnt <= 7;

            if ss_n = '0' then
                -- Inizia transazione SPI
                shift_out <= tx_data;
                miso      <= tx_data(7);  -- mette subito il primo bit
                state     <= SHIFT;
            end if;
```

Il primo bit viene messo su MISO immediatamente per rispettare il timing di CPHA=0

Stato SHIFT - Trasferimento Bit

```
when SHIFT =>
    if ss_n = '1' then
        -- Fine transazione, segnala dato ricevuto
        rx_valid <= '1';
        rx_data  <= shift_in;
        state    <= IDLE;
    else
        -- Campiona MOSI sul fronte di salita
        if sclk_prev = '0' and sclk = '1' then
            shift_in(bit_cnt) <= mosi;
        end if;

        -- Aggiorna MISO sul fronte di discesa
        if sclk_prev = '1' and sclk = '0' then
            if bit_cnt > 0 then
                bit_cnt <= bit_cnt - 1;
                miso <= shift_out(bit_cnt - 1);
            else
                -- Ultimo bit trasmesso, resta a bit_cnt=0
                bit_cnt <= 0;
            end if;
        end if;
    end if;
```

Timing Critico

8 cicli di SCLK per trasferire 8 bit

Ogni bit campionato/aggiornato sui fronti corretti

(il campionamento viene eseguito sul fronte di salita)

rx_valid si attiva per 1 ciclo di clk al termine

Implementazione interfaccia di comunicazione SPI

Procedura spi_transfer

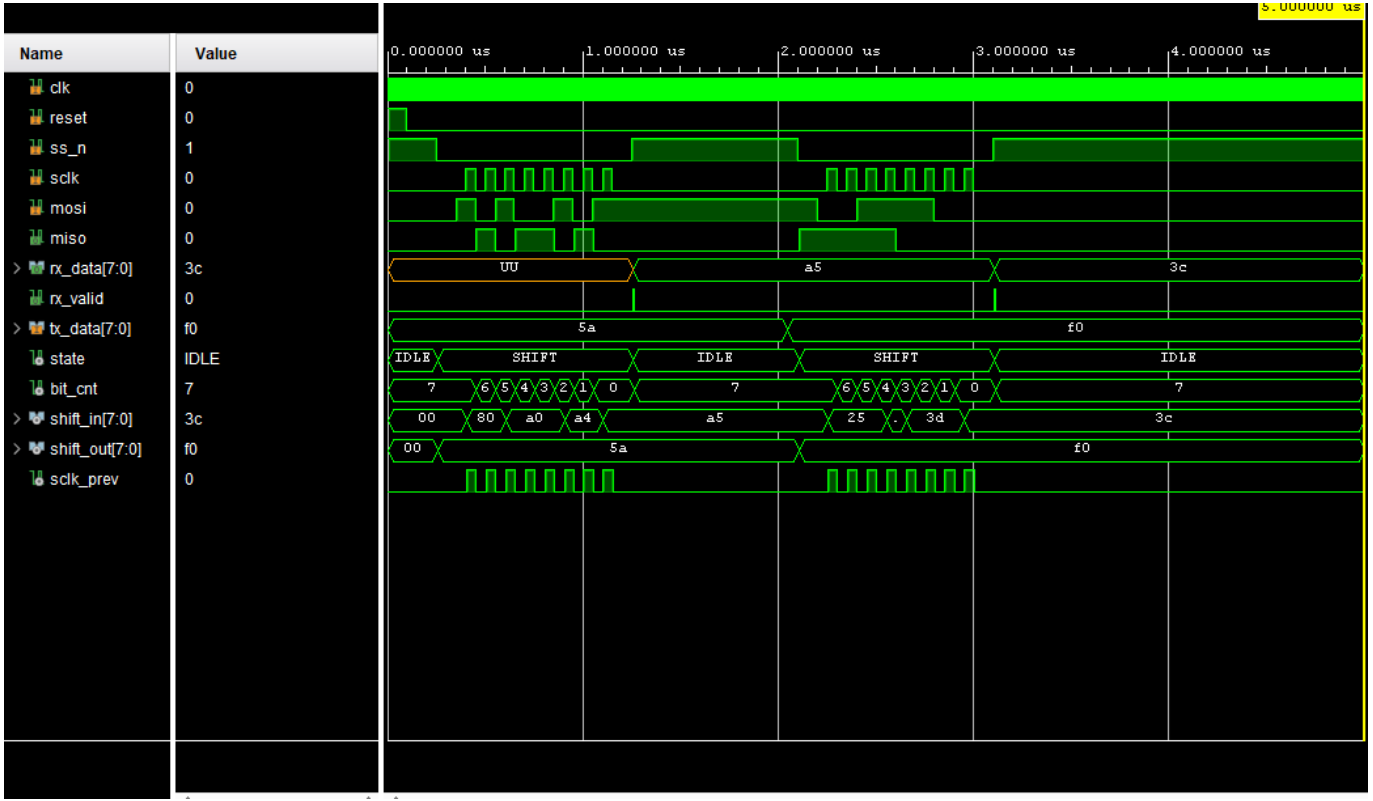
```
procedure spi_transfer (
    signal ss       : out std_logic;
    signal sclk_out : out std_logic;
    signal mosi_out : out std_logic;
    signal miso_in  : in  std_logic;
    constant tx_byte : in std_logic_vector(7 downto 0);
    variable rx_byte : out std_logic_vector(7 downto 0)
) is
begin
    -- Attivo chip select
    ss <= '0';
    wait for SCLK_PERIOD;

    -- Trasferimento 8 bit
    for i in 7 downto 0 loop

        -- Imposto MOSI
        mosi_out <= tx_byte(i);
        wait for SCLK_PERIOD/2;

        -- Fronte di salita SCLK
        sclk_out <= '1';
        wait for SCLK_PERIOD/4;

        -- Campiona MISO a metà del periodo alto
        rx_byte(i) := miso_in;
        wait for SCLK_PERIOD/4;

        -- Fronte di discesa SCLK, slave aggiorna MISO
        sclk_out <= '0';
    end loop;

    -- Disattivo chip select
    wait for SCLK_PERIOD;
    ss <= '1';
    wait for SCLK_PERIOD;
end procedure;
```

Test Eseguiti

Test 1: Master invia 0xA5, Slave risponde 0x5A

Test 2: Master invia 0x3C, Slave risponde 0xF0

Implementazione interfaccia di comunicazione SPI



1222·2022 800 ANNI

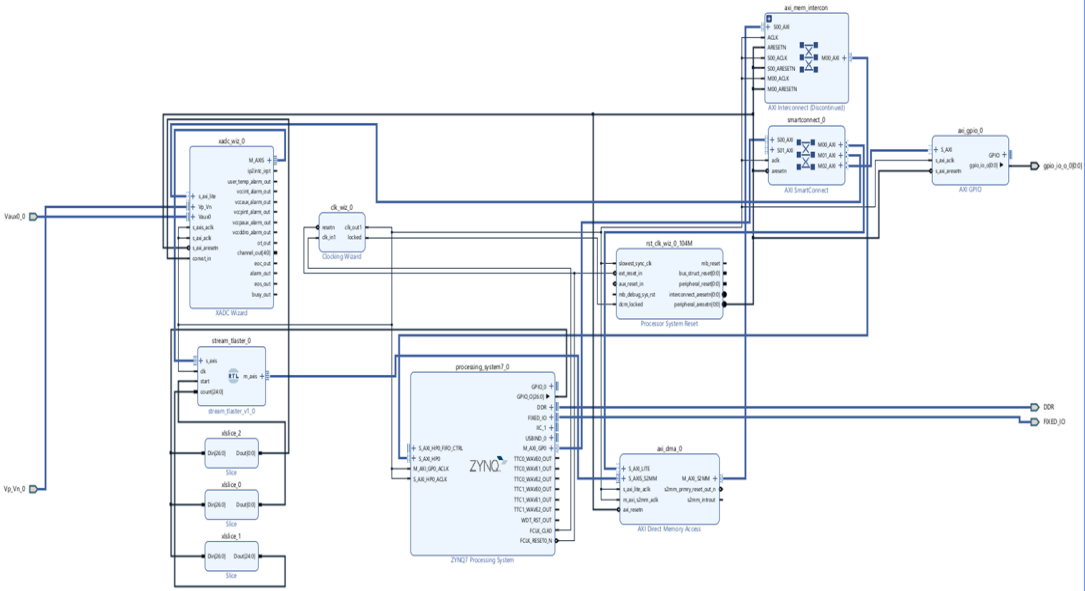
UNIVERSITÀ DEGLI STUDI DI PADOVA

Acquisizione di un segnale analogico tramite l'XADC

Voltarel Nicola

Panoramica

- Predisposizione dell'XADC per il funzionamento in Event Mode
- Trasmissione dati tramite AXI DMA
- Configurazione di un interrupt timer per il controllo dell'istante di acquisizione
- Utilizzo di un GPIO pin per visualizzare la chiamata della sub-routine



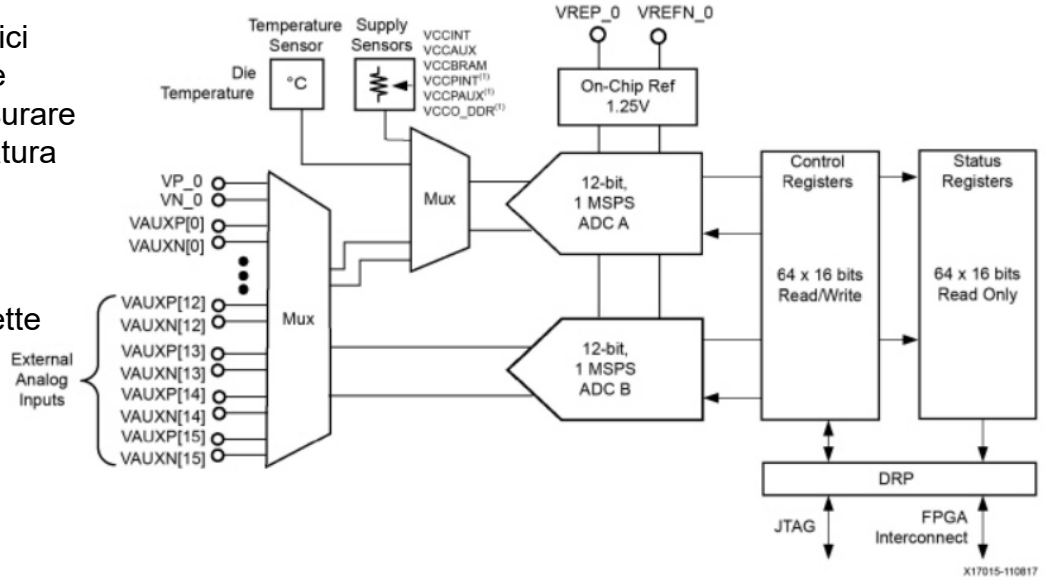
Cos'è l'XADC

XADC è un blocco contenuto all'interno dell'FPGA il quale include due adc a 12 bit, capaci di frequenza di campionamento fino a 1 MSps.

Gli adc hanno accesso a 17 input analogici esterni (di cui 16 sono gli input ausiliari) e posseggono diversi sensori capaci di misurare la tensione di alimentazione e la temperatura del die di silicio

Event Mode

Questa modalità di funzionamento permette all'XADC di avviare il campionamento soltanto dopo la ricezione di un segnale di trigger.



Interrupt timer

i segnali di trigger responsabili delle acquisizioni dell'ADC vengono generati tramite una ISR a sua volta avviata dall'interrupt di un timer opportunamente configurato.

l'immagine seguente mostra la parte saliente della configurazione del timer

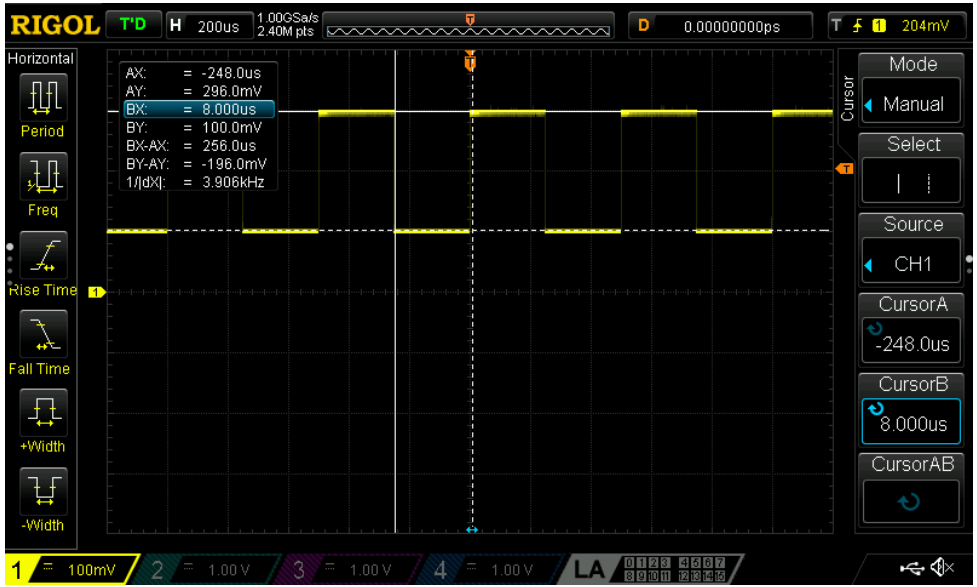
```
193     // Setup interrupt system
194     Status = SetupInterruptSystem(&GicInstance, &TimerInstance);
195     if (Status != XST_SUCCESS) {
196         cout << "setup interrupt fallito" << endl;
197         return XST_FAILURE;
198     }
199
200     // Calcola valore di load per il periodo desiderato
201     LoadValue = (TIMER_CLOCK_HZ / 1000000) * PERIODO_MS;
202     XScuTimer_LoadTimer(&TimerInstance, LoadValue);
203
204     // Abilita modalita auto-reload e interrupt
205     XScuTimer_EnableAutoReload(&TimerInstance);
206     XScuTimer_EnableInterrupt(&TimerInstance);
```

```
void TimerIntrHandler(void* CallBackRef)
{
    u32 status;

    XScuTimer* TimerInstancePtr = (XScuTimer*)CallBackRef;

    XScuTimer_ClearInterruptStatus(TimerInstancePtr);



    // invia l'impulso alla porta CONVST per avviare la conversione
    XGpioPs_WritePin(&GpioInstance, CONVST_PIN, 1 /*high*/);
    XGpioPs_WritePin(&GpioInstance, CONVST_PIN, 0 /*low*/);
```

Sezione dell'ISR che invia il segnale di trigger

Test di laboratorio

solo XADC

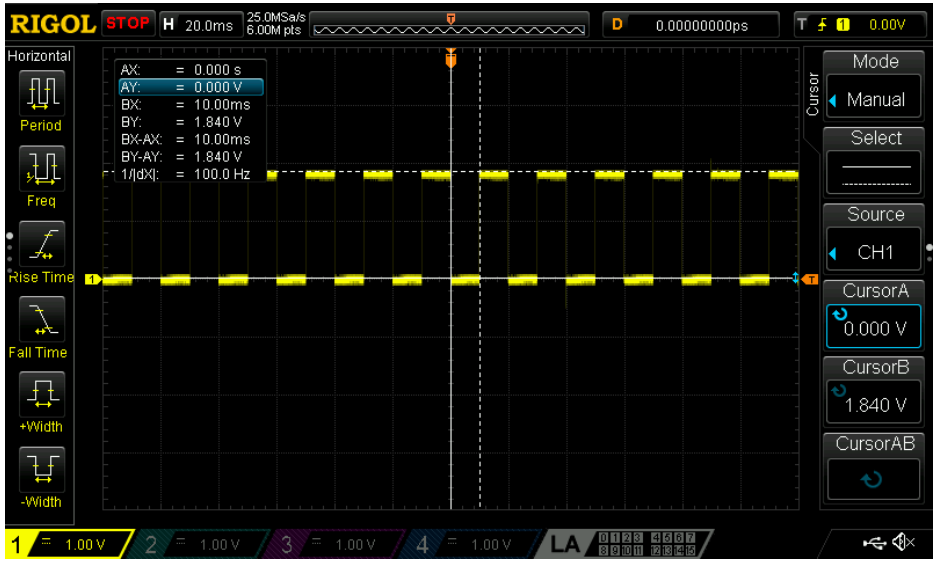


Segnale in uscita dal generatore di funzione

Segnale campionato del Zynq-7000

Test di laboratorio

Solo timer interrupt

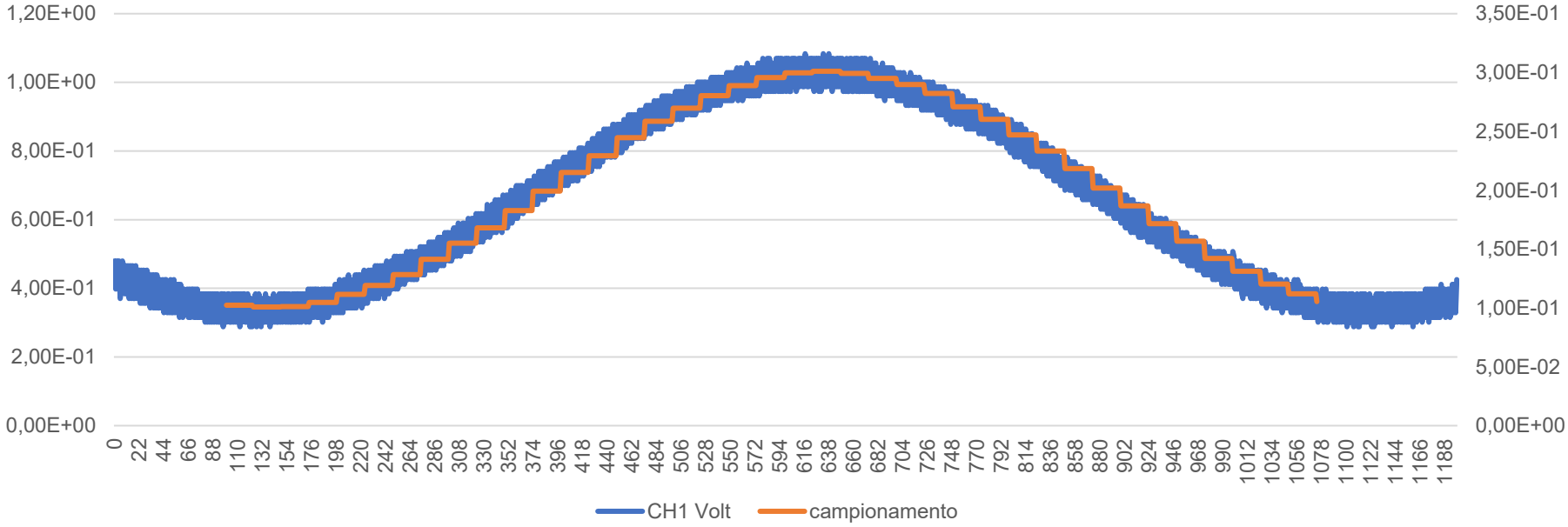


Per questo test è stato impostato, da codice, un periodo del timer pari a 10 ms e da come si può vedere la frequenza mostrata dall'oscilloscopio corrisponde perfettamente al risultato aspettato

Test di laboratorio

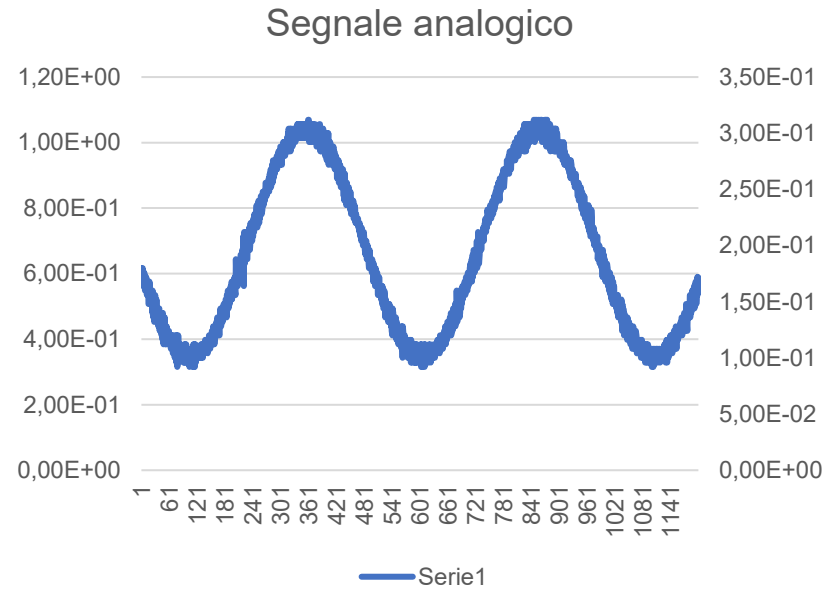
UNIVERSITÀ DEGLI STUDI DI PADOVA

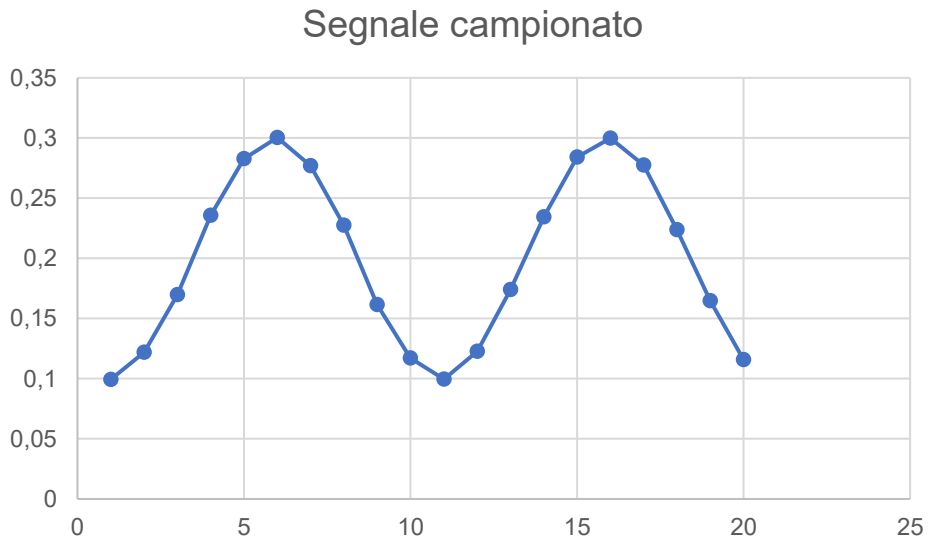
Programma completo 1/2



Test di laboratorio

Programma completo (segnale a 100 KHz e campionamento a 1 MHz)





1222·2022
800 ANNI

UNIVERSITÀ DEGLI STUDI DI PADOVA

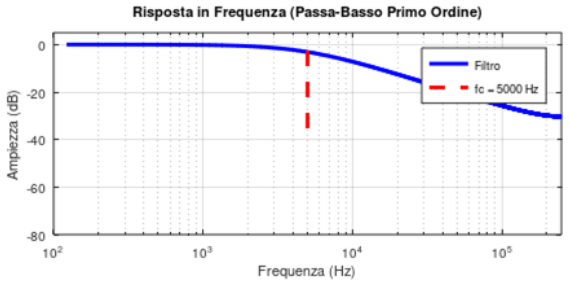
Filtri Digitali

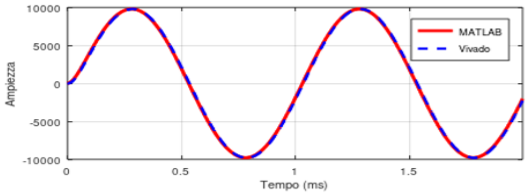
Pennacchio Francesco

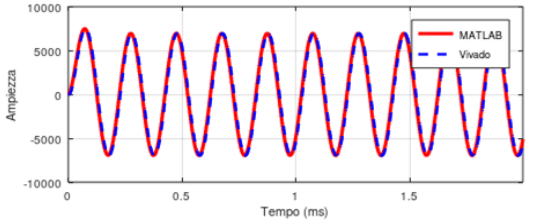
Filtro IIR Passa Basso

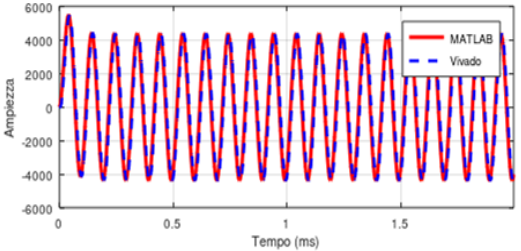
- Filtro del I ordine
- fs = 500Khz
- fc = 5Khz
- $H(s) = \frac{1}{1+s\tau}$

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;

-- fsample = 500KHz
-- fcut = 5 kHz
entity IIR_LPF is
  generic(
    a_val : integer := 30831;  -- 1.15 format of 0.940
    b_val : integer := 1937;   -- 1.15 format of 0.059
    N     : integer := 16;     -- in bit
    M     : integer := 16;     -- out bit
    z     : integer := 16      -- constant format
  );
  Port (
    clk      : in  std_logic;
    res      : in  std_logic;
    in_data  : in  std_logic_vector(N-1 downto 0);
    out_data : out std_logic_vector(M-1 downto 0)
  );
end IIR_LPF;

architecture Behavioral of IIR_LPF is
  constant a : signed(z-1 downto 0) := to_signed(a_val, z);
  constant b : signed(z-1 downto 0) := to_signed(b_val, z);

begin

  lpf : process (res, clk)

    variable v_yk    : signed(M-1 downto 0) := (others => '0');
    variable v_sum_a : signed(N+z downto 0);  -- 33 bit per a * y[k-1]
    variable v_sum_b : signed(N+z downto 0);  -- 33 bit per b * x[k]
    variable v_total : signed(N+z downto 0);  -- 33 bit per la somma totale
  begin
    if res = '1' then
      out_data <= (others => '0');
      v_yk     := (others => '0');

    elsif rising_edge(clk) then
      -- Step 1: calcola a * y[k-1]
      v_sum_a := a * resize(v_yk, M+1);
    elsif rising_edge(clk) then
      -- Step 1: calcola a * y[k-1]
      v_sum_a := a * resize(v_yk, M+1);

      -- Step 2: calcola b * x[k]
      v_sum_b := b * resize(signed(in_data), N+1);

      -- Step 3: somma i due contributi
      v_total := v_sum_a + v_sum_b;

      -- Step 4: estrai il risultato in Q15.0 (scala da Q15.15)
      v_yk := v_total(30 downto 15);

      -- Step 5: aggiorna output
      out_data <= std_logic_vector(v_yk);

    end if;
  end process;

end Behavioral;
```





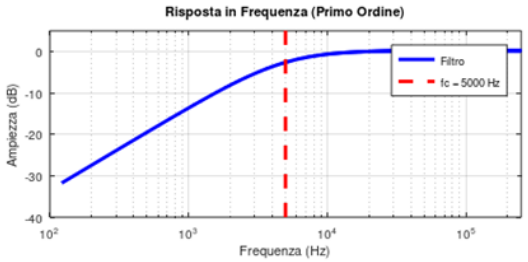


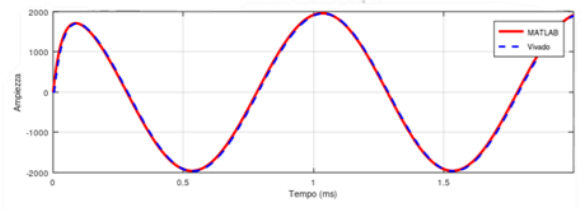


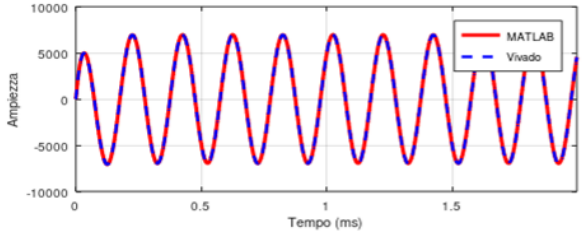
Filtro IIR Passa Alto

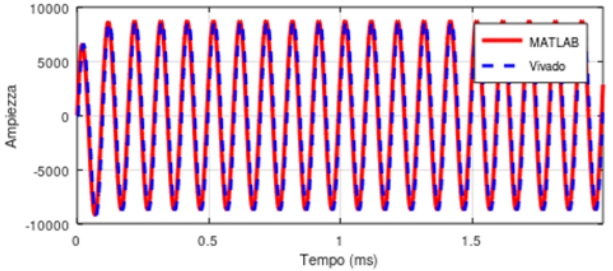
- Filtro I ordine
- fs = 500Khz
- fc = 5Khz
- $H(s) = \frac{s\tau}{1+s\tau}$

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;

entity iir_hpf is      -- fsample = 500 Khz, fcut = 5 KHz
  generic (
    N     : integer := 16;
    M     : integer := 16;
    z     : integer := 16;
    a_val : integer := 30831  -- 1.15 format
  );
  Port (
    in_data  : in  std_logic_vector(N-1 downto 0);
    out_data : out std_logic_vector(M-1 downto 0);
    clk      : in  std_logic;
    res      : in  std_logic
  );
end iir_hpf;

architecture Behavioral of iir_hpf is
  signal past_in : signed(N-1 downto 0) := (others => '0');
  constant a     : signed(z-1 downto 0) := to_signed(a_val, z);

begin

  hpf : process (res, clk)
    variable v_yk   : signed(M-1 downto 0) := (others => '0');
    variable v_temp : signed(M downto 0);       -- 17 bit per evitare overflow nella somma
    variable v_sum  : signed(N+z downto 0);     -- 33 bit per il prodotto
  begin
    if res = '1' then
      out_data <= (others => '0');
      past_in  <= (others => '0');
      v_yk     := (others => '0');

    elsif rising_edge(clk) then
      -- Step 1: calcola (yk + x[k] - x[k-1]) con gestione overflow
      v_temp := resize(v_yk, M+1) + resize(signed(in_data), M+1) - resize(past_in, M+1);

      -- Step 2: moltiplica per a
      v_sum := a * v_temp;

      -- Step 3: estrai il risultato in Q15.0 (scala da 15.15)
      v_yk := v_sum(30 downto 15);

      -- Step 4: aggiorna output e stato
      out_data <= std_logic_vector(v_yk);
      past_in  <= signed(in_data);

    end if;
  end process;

end Behavioral;
```



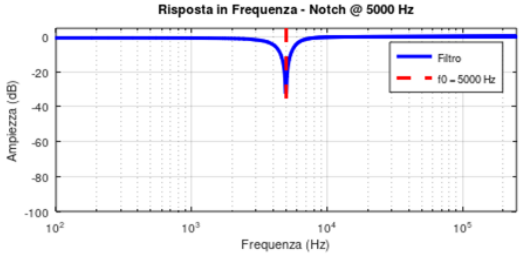




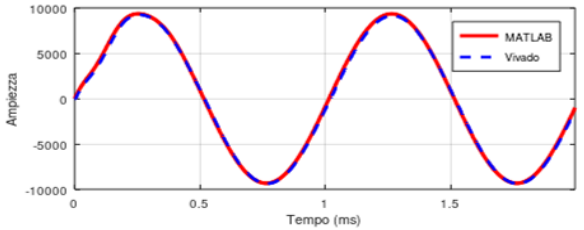


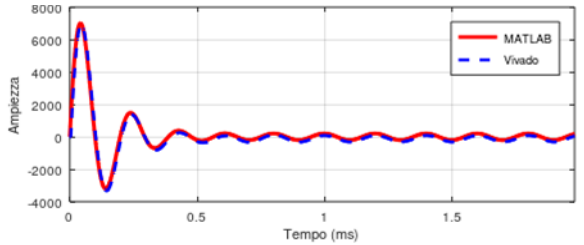
Filtro IIR Notch

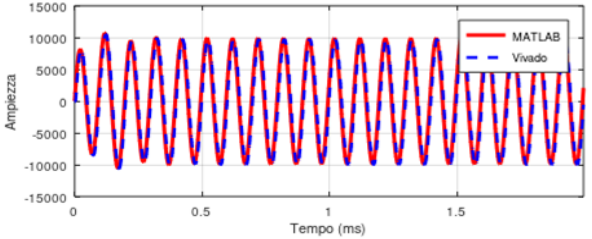
- Filtro del II ordine
- fs = 500Khz
- fnotch = 5Khz
- Smorzamento = 0.5
- $$H(z) = \frac{b0 + b1 \cdot z^{-1} + b2 \cdot z^{-2}}{1 + a1 \cdot z^{-1} + a2 \cdot z^{-2}}$$
- y[n] = b0*x[n] + b1*x[n-1] + b2*x[n-2]
  - a1*y[n-1] - a2*y[n-2]




```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;

entity iir_notch_filter is                -- fs = 500Khz, f0 = 5Khz, Q = 0.5
    generic(
        N : integer := 16;
        M : integer := 16;
        I : integer := 16;
        b0_val : integer := 8192;   -- 3.13 format of 1
        b1_val : integer := -14852; --                 -1.906
        b2_val : integer := 8192;   --                  1
        a1_val : integer := -14595; --                 -1.945
        a2_val : integer := 7937    --                  0.969
        );

    Port (
        clk     : in  std_logic;
        res     : in  std_logic;
        in_data : in  std_logic_vector(N-1 downto 0);
        out_data: out std_logic_vector(N-1 downto 0)
    );
end iir_notch_filter;

architecture Behavioral of iir_notch_filter is

    -- Tipi per coefficienti
    type t_coeff_in is array (0 to 2) of signed(n-1 downto 0);
    type t_coeff_out is array (0 to 1) of signed(n-1 downto 0);

    -- Coefficienti costanti
    constant coeff_in : t_coeff_in := (
      0 => to_signed(b0_val, n),
      1 => to_signed(b1_val, n),
      2 => to_signed(b2_val, n)
    );

    constant coeff_out : t_coeff_out := (
      0 => to_signed(a1_val, n),
      1 => to_signed(a2_val, n)
    );

    -- Shift register per ingressi (x[n-1], x[n-2])
    signal x1 : signed(N-1 downto 0) := (others => '0');  -- x[n-1]
    signal x2 : signed(N-1 downto 0) := (others => '0');  -- x[n-2]

    -- Shift register per uscite (y[n-1], y[n-2])
    signal y1 : signed(N-1 downto 0) := (others => '0');  -- y[n-1]
    signal y2 : signed(N-1 downto 0) := (others => '0');  -- y[n-2]

begin

    -- Processo principale con variabili
    process(clk, res)
    variable x0_var : signed(N-1 downto 0);
    variable mult_b0 : signed(N+I-1 downto 0);
    variable mult_b1 : signed(N+I-1 downto 0);
    variable mult_b2 : signed(N+I-1 downto 0);
    variable mult_a1 : signed(N+I-1 downto 0);
    variable mult_a2 : signed(N+I-1 downto 0);
    variable acc : signed(33 downto 0);  -- 34 bit per sicurezza
    variable y0_var : signed(N-1 downto 0);
begin
    if res = '1' then
        x1 <= (others => '0');
        x2 <= (others => '0');
        y1 <= (others => '0');
        y2 <= (others => '0');
        out_data <= (others => '0');

    elsif rising_edge(clk) then
        x0_var := signed(in_data);

        mult_b0 := coeff_in(0) * x0_var;
        mult_b1 := coeff_in(1) * x1;
        mult_b2 := coeff_in(2) * x2;
        mult_a1 := coeff_out(0) * y1;
        mult_a2 := coeff_out(1) * y2;

        -- Resize a 34 bit
        acc := resize(mult_b0, 34) + resize(mult_b1, 34) + resize(mult_b2, 34)
             - resize(mult_a1, 34) - resize(mult_a2, 34);

        -- SCALING CORRETTO per formato 3.13: dividi per 2^13
        y0_var := acc(28 downto 13);

        x2 <= x1;
        x1 <= x0_var;
        y2 <= y1;
        y1 <= y0_var;

        out_data <= std_logic_vector(y0_var);
    end if;
end process;

end Behavioral;
```

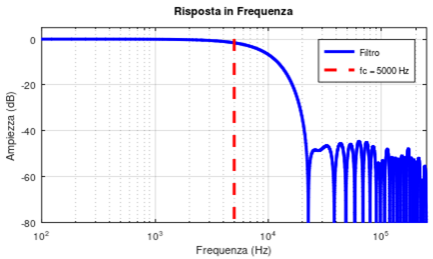






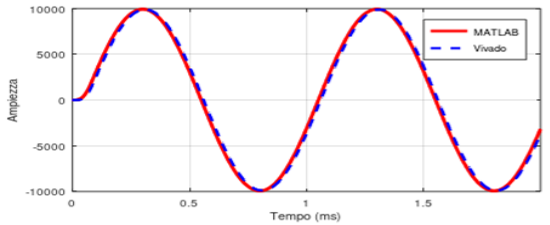
Filtro FIR Passa Basso

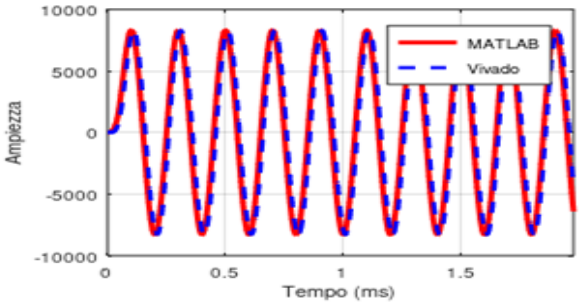
- Ordine 50
- fs = 500KHz
- fc = 5KHz
- B = fir1(N_order, Wn, 'low');

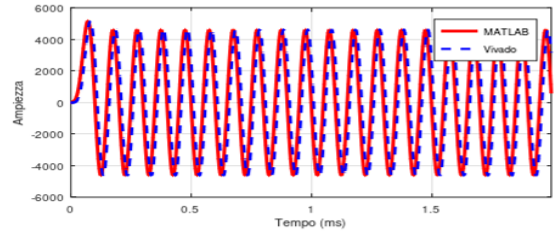



```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

use IEEE.NUMERIC_STD.ALL;


entity fir_lpf is

  generic (
    N : integer := 16;
    M : integer := 16;
    Z : integer := 12;
    ncoeff : integer := 50);
  Port (
    clk : in std_logic;
    res : in std_logic;
    in_data : in std_logic_vector (M-1 downto 0);
    out_data : out std_logic_vector (M-1 downto 0));

end fir_lpf;

architecture Behavioral of fir_lpf is

  type t_coeff is array (0 to ncoeff) of integer;          --coefficents lpf Q12 format
  constant coeff : t_coeff := (
 9, 9, 11, 13, 17, 21, 26, 32, 39,
 47, 56, 65, 74, 84, 93, 103, 112, 121,
 130, 137, 144, 150, 154, 158, 160, 160, 160,
 158, 154, 150, 144, 137, 130, 121, 112, 103,
 93, 84, 74, 65, 56, 47, 39, 32, 26,
 21, 17, 13, 11, 9, 9
  );

  type t_data is array (0 to ncoeff-1) of signed (M-1 downto 0);
  signal s_data : t_data := (others => (others => '0'));

  type t_mult is array (0 to ncoeff-1) of signed (N+z-1 downto 0);
  signal s_mult : t_mult := (others => (others => '0'));

  signal add : signed (M-1 downto 0) := (others => '0');
  signal acc : signed (N+z-1 downto 0):= (others => '0');  --lpf/hpf

--  Multiplications
  mult : process (res, clk)
  begin
    if res = '1' then
      s_mult <= (others => (others => '0'));
    elsif rising_edge(clk) then
      for k in 0 to ncoeff-1 loop
        s_mult(k) <= s_data(k) * to_signed(coeff(k), z);
      end loop;
    end if;
  end process;
begin

--  Shift register
  shift_register : process (clk, res)
  begin
    if res = '1' then
      s_data <= (others => (others => '0'));
    elsif rising_edge(clk) then
      for k in ncoeff-1 downto 1 loop
        s_data(k) <= s_data(k-1);
      end loop;
      s_data(0) <= signed(in_data);
    end if;
  end process;
```

```
-- Sum
  sum : process (res, clk)
    variable acc_var : signed(N+z-1 downto 0);
  begin
    if res = '1' then
      add <= (others => '0');
      acc <= (others => '0');
      out_data <= (others => '0');
    elsif rising_edge(clk) then
      acc_var := (others => '0');
      for k in 0 to ncoeff-1 loop
        acc_var := acc_var + s_mult(k);
      end loop;
      acc <= acc_var;

      add <= acc_var(27 downto 12);   --lpf shift
      out_data <= std_logic_vector(add);
    end if;
  end process;
```

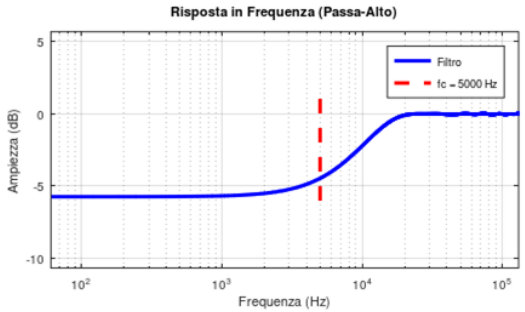






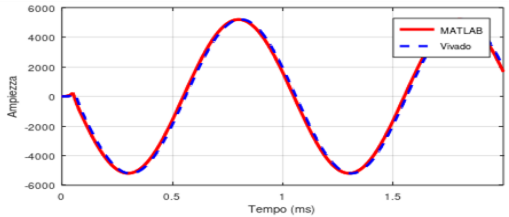
Filtro FIR Passa Alto

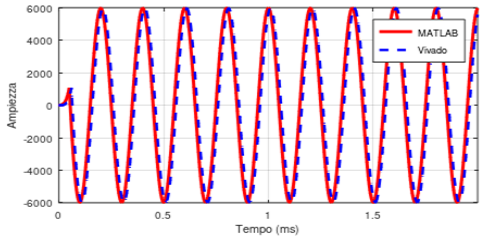
- Ordine 50
- fs = 500KHz
- fc = 5KHz
- B = fir1(N_order, Wn, 'high');

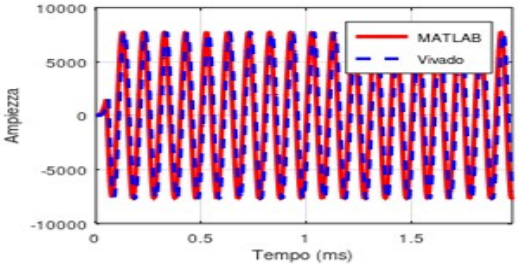



```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

use IEEE.NUMERIC_STD.ALL;


entity fir_lpf is

  generic (
    N : integer := 16;
    M : integer := 16;
    z : integer := 12;
    ncoeff : integer := 50);
  Port (
    clk : in std_logic;
    res : in std_logic;
    in_data : in std_logic_vector (N-1 downto 0);
    out_data : out std_logic_vector (M-1 downto 0));

end fir_lpf;

architecture Behavioral of fir_lpf is

type t_coeff is array (0 to ncoeff) of integer;          --coefficietns hpf 1.11 format
constant coeff : t_coeff := (
    2, 2, 3, 3, 4, 5, 6, 8, 10,
    11, 13, 16, 18, 20, 23, 25, 27, 29,
    31, 33, 35, 36, 37, 38, 39, -2004, 39,
    38, 37, 36, 35, 33, 31, 29, 27, 25,
    23, 20, 18, 16, 13, 11, 10, 8, 6,
    5, 4, 3, 3, 2, 2
);


  type t_data is array (0 to ncoeff-1) of signed (N-1 downto 0);
  signal s_data : t_data := (others => (others => '0'));

  type t_mult is array (0 to ncoeff-1) of signed (N+z-1 downto 0);
  signal s_mult : t_mult := (others => (others => '0'));

  signal add : signed (N-1 downto 0) := (others => '0');
  signal acc : signed (N+z-1 downto 0):= (others => '0');  --lpf/hpf

--   Multiplications
  mult : process (res, clk)
  begin
    if res = '1' then
      s_mult <= (others => (others => '0'));
    elsif rising_edge(clk) then
      for k in 0 to ncoeff-1 loop
        s_mult(k) <= s_data(k) * to_signed(coeff(k), z);
      end loop;
    end if;
  end process;

begin

--   Shift register
 shift_register : process (clk, res)
 begin
    if res = '1' then
      s_data <= (others => (others => '0'));
    elsif rising_edge(clk) then
      for k in ncoeff-1 downto 1 loop
        s_data(k) <= s_data(k-1);
      end loop;
      s_data(0) <= signed(in_data);
    end if;
 end process;
```

```
-- Sum
sum : process (res, clk)
  variable acc_var : signed(N+z-1 downto 0);
begin
  if res = '1' then
    add <= (others => '0');
    acc <= (others => '0');
    out_data <= (others => '0');
  elsif rising_edge(clk) then
    acc_var := (others => '0');
    for k in 0 to ncoeff-1 loop
      acc_var := acc_var + s_mult(k);
    end loop;
    acc <= acc_var;

      add <= acc_var(26 downto 11);     --hpf shift
    out_data <= std_logic_vector(add);
  end if;
end process;
```

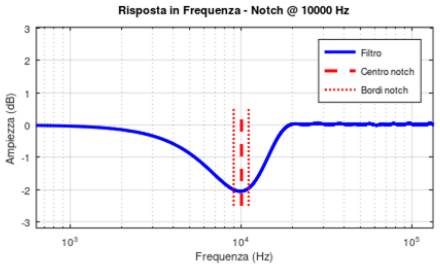






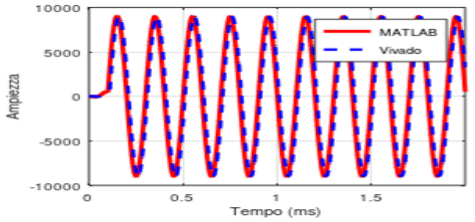
Filtro FIR Notch

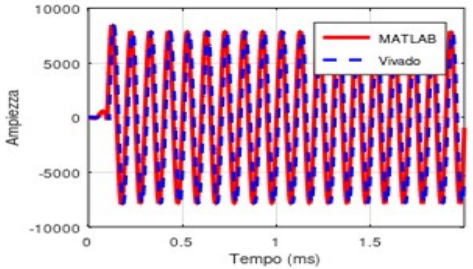
- Ordine 100
- fs = 500Khz
- fnotch = 10Khz
- B = fir1(N_order, [Wn1 Wn2], 'stop');

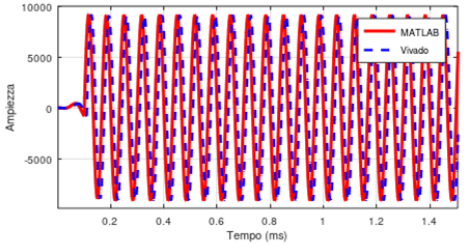



```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

use IEEE.NUMERIC_STD.ALL;

entity fir_lpf is

  generic (
    N : integer := 16;
    M : integer := 16;
    S : integer := 12;
    ncoeff : integer := 50);
  Port (
    clk : in std_logic;
    res : in std_logic;
    in_data : in std_logic_vector (N-1 downto 0);
    out_data : out std_logic_vector (M-1 downto 0));

end fir_lpf;

type t_coeff is array (0 to (ncoeff+ncoeff)) of integer;  -- Coefficienti FIR notch, formato 1.11
constant coeff : t_coeff := (
 -1, -1, -1, -1, -1, -1, -1, -1, -1, -1,
 0, 0, 0, 1, 1, 2, 2, 3, 4, 5,
 5, 6, 7, 8, 8, 9, 9, 9, 9, 9,
 9, 8, 7, 6, 5, 3, 2, 0, -1, -3,
 -5, -7, -9, -10, -12, -13, -14, -15, -16, -16,
 2027, -16, -16, -15, -14, -13, -12, -10, -9, -7,
 -5, -3, -1, 0, 2, 3, 5, 6, 7, 8,
 9, 9, 9, 9, 9, 9, 8, 8, 7, 6,
 5, 5, 4, 3, 2, 2, 1, 1, 0, 0,
 0, -1, -1, -1, -1, -1, -1, -1, -1, -1,
 -1
);


 type t_data is array (0 to 100) of signed (N-1 downto 0);
 signal s_data : t_data := (others => (others => '0'));

 type t_mult is array (0 to 100) of signed (N+z-1 downto 0);
 signal s_mult : t_mult := (others => (others => '0'));

 signal add : signed (M-1 downto 0) := (others => '0');
 signal acc : signed (N+z-1 downto 0):= (others => '0');

begin

 mult : process (res, clk)
 begin
   if res = '1' then
     s_mult <= (others => (others => '0'));
   elsif rising_edge(clk) then
     for k in 0 to 100 loop
       s_mult(k) <= s_data(k) * to_signed(coeff(k), z);
     end loop;
   end if;
 end process;

 shift_register : process (clk, res)
 begin
   if res = '1' then
     s_data <= (others => (others => '0'));
   elsif rising_edge(clk) then
     for k in 100 downto 1 loop
       s_data(k) <= s_data(k-1);
     end loop;
     s_data(0) <= signed(in_data);
   end if;
 end process;
```

```
sum : process (res, clk)
   variable acc_var : signed(N+z-1 downto 0);
 begin
   if res = '1' then
     add <= (others => '0');
     acc <= (others => '0');
     out_data <= (others => '0');
   elsif rising_edge(clk) then
     acc_var := (others => '0');
     for k in 0 to 100 loop
       acc_var := acc_var + s_mult(k);
     end loop;
     acc <= acc_var;

       add <= acc_var(26 downto 11);   --hpf/notch shift
     out_data <= std_logic_vector(add);
   end if;
 end process;

end Behavioral;
```







1222·2022 800 ANNI

UNIVERSITÀ DEGLI STUDI DI PADOVA

GRAZIE PER L'ATTENZIONE

Università degli Studi di Padova