

# $\alpha$ (ALPHA)

## AMPIEZZA TOTALE COSTANTE

Considerato che si sta lavorando in “hardware” è necessario tener conto del problema dell’*overflow*. Ciò significa che, desiderando variare l’intensità dell’eco, si deve mantenere all’incirca costante l’intensità sonora totale, o al più, non incrementarla. Supponendo che il segnale audio abbia un valore efficace costante, si desidera che:

$$(1) \quad V_u \leq V_i \quad [V]$$

Ricordando la seguente convenzione:

- $a_b$  : valore istantaneo della componente variabile;
- $a_B$  : valore istantaneo totale;
- $A_b$  : valore efficace della componente variabile;
- $A_B$  : valore di riposo.

In tal modo, dato che l’uscita del A/D ha un valore interno al range di overflow, si ha che l’uscita non dovrebbe saturare.

La (1) può essere riscritta nel seguente modo:

$$(2) \quad V_u \leq V_i \cdot (1 - \alpha + \alpha) = \alpha V_i + (1 - \alpha) V_i \quad [V]$$

Quindi, per un qualsiasi valore del parametro  $\alpha$  si ha che il D/A (convertitore d’uscita) non andrà in overflow. Ora, scegliendo un valore  $0 < \alpha < 1$  si ha che la (2) è sempre verificata se:

$$(3) \quad V_u \leq \alpha V_i + (1 - \alpha) \alpha V_i = \alpha [1 + (1 - \alpha)] V_i \quad [V]$$

## AMPLIFICAZIONE SEGNALE DIRETTO

Il blocco  $\alpha$  si occupa dell’amplificazione (attenuazione, volendo esser precisi) del “segnale diretto” quantizzato e discretizzato. Supponendo che un segnale (elettrico) analogico  $x_o(t)$  sia applicato al jack (J1) d’ingresso (si veda lo *schema a blocchi* e la sezione A/D e D/A presente sul sito), esso diverrà un  $x_o[n]$  (discretizzato) e successivamente un  $x[n]$  (quantizzato). Ora  $x[n]$  si ritroverà in ingresso al blocco  $\alpha$ , dove il “segnale diretto” di cui si parlava prima non è altro che, appunto,  $x[n]$ , e immaginando di aver memorizzato esclusivamente degli zeri, con il blocco  $z^{-N}$ , si ha che il segnale amplificato (attenuato, dato che  $0 < \alpha < 1$ )  $s[n] = y[n] = \alpha \cdot x[n]$  prosegue verso due direzioni:

- al convertitore D/A;
- al filtro blocco di ritardo  $z^{-N}$  e al moltiplicatore  $1 - \alpha$ .

La prima strada provoca la conversione  $y[n] \mapsto y_o[n] \mapsto y_o(t)$  e il segnale si trova al jack (J2) d’uscita. La seconda via, invece, attua la seguente funzione  $y[n] \mapsto y[n - N] \mapsto (1 - \alpha)y[n - N] = w[n]$  o, analogamente,  $Y(z) \mapsto Y(z) \cdot z^{-N} \mapsto (1 - \alpha)Y(z) \cdot z^{-N} = W(z)$ .

Ora il blocco sommatore ha come uscita il segnale  $s[n] = w[n] + y[n]$  cioè  $S(z) = W(z) + Y(z)$  e ha spettro d'ampiezza pari a:

$$(4) \quad |S(e^{j\omega})| = |W(e^{j\omega}) + Y(e^{j\omega})| = |X(e^{j\omega})[\alpha(1-\alpha)e^{-jN} + \alpha]| = |X(e^{j\omega})| |\alpha(1-\alpha)e^{-jN} + \alpha| < |X(e^{j\omega})| \alpha(1+(1-\alpha)) \quad [V]$$

Ne segue che:

$$(5) \quad |S(e^{j\omega})| < |X(e^{j\omega})| \alpha(1+(1-\alpha)\alpha) \quad [V]$$

Quindi, come dimostrato nella (3),  $|S(e^{j\omega})| < |X(e^{j\omega})|$  riuscendo quindi nell'intento di non far saturare il convertitore, incappando nell'overflow.

## CODICE ARCHITETTURALE

```
-- alpha.vhd
-- Canziani Alfredo & Viviani Emanuele production
-- DEEI, Università degli studi di Trieste

-- %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
-- %
-- %                               LIBRERIE INCLUDE
-- %
-- %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

-- %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
-- %
-- %                               PORTE DI ALPHA
-- %                               (terminali di ingresso ed uscita)
-- %
-- %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

entity alpha is
    Port (    data_recorder    : in  std_logic_vector (15 downto 0);
            switch            : in  std_logic_vector (1 downto 0);
            data_out          : out std_logic_vector (15 downto 0)
            );
end alpha;

-- %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
-- %
-- %                               FUNZIONAMENTO
-- %
-- %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

architecture comportamento of alpha is

    signal dato2: std_logic_vector (15 downto 0); -- segnali utilizzati e associati
    signal dato4: std_logic_vector (15 downto 0); -- ad uno scalaggio per un fattore
    signal dato8: std_logic_vector (15 downto 0); -- 1/2, 3/4 e 7/8 per il segnale "diretto"

begin
    process (switch, data_recorder) -- variamo il fattore o arriva un nuovo segnale...
    begin

        case (switch) is -- ricordando che gli switch si chiudono a massa e si aprono a Vcc
            when "11" => -- alpha = 1
                data_out <= data_recorder; -- in uscita ciò che ascolto: eco = 0

            when "01" => -- alpha = 7/8 = 1/2 + 1/4 + 1/8
```

```

dato2 (15 downto 0) <= data_recorder (15) & data_recorder (15 downto 1);...
    ...-- shift x numeri in complemento a 2
dato4 (15 downto 0) <= data_recorder (15) & data_recorder (15) &...
    ...data_recorder (15 downto 2);
dato8 (15 downto 0) <= data_recorder (15) & data_recorder (15) &...
    ...data_recorder (15) & data_recorder (15 downto 3);
data_out <= dato2 + dato4 + dato8;

when "10" =>    -- alpha = 3/4 = 1/2 + 1/4
    dato2 (15 downto 0) <= data_recorder (15) & data_recorder (15 downto 1);
    dato4 (15 downto 0) <= data_recorder (15) & data_recorder (15) &...
        ...data_recorder (15 downto 2);
    data_out <= dato2 + dato4;

when others => -- alpha = 1/2, switch = 00
    dato2 (15 downto 0) <= data_recorder (15) & data_recorder (15 downto 1);
    data_out <= dato2;

end case;

end process;
end comportamento;

```

## SHIFT PER NUMERI IN COMPLEMENTO A 2

Uno dei tanti errori in cui siamo incappati è stato quello inerente lo *shift*. Sapendo che la divisione per due, per i numeri binari, si effettua tramite lo shift a destra (come con i numeri decimali, la divisione per 10 si effettua semplicemente “spostando” i numeri verso destra), abbiamo cercato di creare questo operatore.

Quindi, partendo da un numero, espresso in base due, come ad esempio:

$$1\ 0010\ 1101_2$$

il risultato dell’operazione  $301_{10} : 2_{10}$  è:

$$1001\ 0110_2$$

e fin qua non vi è alcun problema.

Consideriamo ora un dispositivo hardware, con quindi una dimensione *finita* di *bit* per *word*. Supponiamo essere il numero di bit a disposizione maggiore della *lunghezza* di  $301_{10}$ . Riproponiamo lo stesso esempio:

$$\begin{aligned}
 0000\ 0001\ 0010\ 1101_2 : 10_2 &= \\
 &= 0000\ 0000\ 1001\ 0110_2
 \end{aligned}$$

A rigor di logica sembra che per effettuare uno shift sia necessario “spostare” i bit di una (o più) posizioni verso destra (sinistra, se moltiplichiamo) e aggiungere degli zeri “davanti” (“dietro”).

Dopo aver implementato tale soluzione, ciò che si poteva udire dalle casse connesse al J2 era parte del segnale d’ingresso più una buona quantità di *rumore*. Inizialmente si era pensato ad un overflow del *blocco*  $\alpha$ , ma non era così.

Consideriamo sempre il numero binario:

$$1\ 0010\ 1101_2$$

che può essere interpretato come  $-211_{10}$ ; il risultato dell’operazione  $-211_{10} : 2_{10} = -106_{10}$ , pari a:

$$1\ 1001\ 0110_2 \neq 0\ 1001\ 0110_2 = 150_{10}$$

Quindi, a lato pratico, considerando una dimensione delle parole a 16 bit, si avrà che:

$$\begin{aligned} 1111\ 1111\ 0010\ 1101_2 &: 10_2 = \\ &= 1111\ 1111\ 1001\ 0110_2 \end{aligned}$$

Ciò si traduce nel seguente *algoritmo di shift* (divisione per 2) per una *parola a N bit*:

- si copiano gli  $N - 1$  bit della parola di partenza: da  $bit1_{old}$  a  $bit15_{old}$  in  $bit0_{new}$  fino a  $bit14_{new}$ ;
- si effettui la seguente associazione:  $bit15_{new} = bit15_{old}$ .

#### UN ESEMPIO DI DIVISIONE PER 8

```
new_word (15 downto 0) <= old_w (15) & old_w (15) & old_w (15) & old_w (15 downto 3);
```

