Filtro CIC Interpolatore
Progetto di Apparati Elettronici

Andrea Mondelli

2010/2011

Indice

Capitolo 1

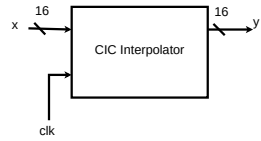
Specifiche di Progetto

Progettare un circuito digitale che realizzi un filtro CIC (Cascated Integrator-Comb) passa basso interpolatore. Le sue caratteristiche dovranno essere le seguenti.

- Fattore di interpolazione pari a R = 4
- Ritardo degli stadi Comb pari a M = 1
- Numero di stadi del filtro pari a N = 4
- Inserzione di R-1 zeri nel passaggio fra stadi Comb e stadi Integrator (Zero-Insertion)

$$y[n] = \left[\sum_{k=0}^{RM-1} x[n] - k \right]^N$$

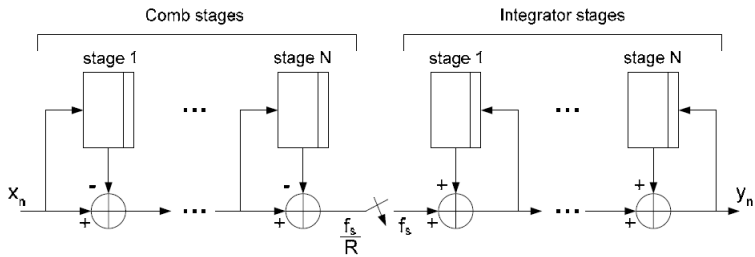
Per ingressi e uscite utilizzare una rappresentazione su 16 bit.



La relazione finale del progetto deve contenere:

- Introduzione (descrizione algoritmo, possibili applicazioni, possibili architetture, etc.)
- Descrizione dell'architettura (diagramma a blocchi, ingressi\uscite, etc.)
- Codice VHDL (con commenti dettagliati)
- Testbench per la verifica
- Conclusioni

Un'architettura di questo tipo è la seguente:



Capitolo 2

Filtro CIC

2.1 Descrizione del Filtro CIC

Nell'ambito del digital signal processing, un Cascaded Integrator-Comb (CIC) è una tipologia di filtri a risposta di impulso finita, detti anche FIR, con caratteristiche di interpolazione o decimazione. Un filtro CIC consiste in una o più coppie di blocchi integratori e comb. Nel caso di filtro CIC di tipo decimatore, consiste in uno o più integratori posizionati a cascata, seguiti da un down-sample (riduttore di frequenza), seguito a sua volta da uno o più sezioni comb in cascata. Il numero di blocchi comb e di blocchi integrator devono necessariamente essere uguali. Un filtro CIC di tipo interpolatore invece è il suo reciproco, con i blocchi comb a monte separati dai blocchi integrator di valle da un blocco che effettua una zero-insertion ed una moltiplicazione di frequenza.

Questa classe di filtri fu inventata da Eugene B. Hogenauer[1] come classe di filtri FIR. Lo scopo principale di questa classe di filtri è di effettuare decimazione ed interpolazione. A differenza di molti filtri FIR, i filtri CIC inglobano già nell'architettura tali funzioni.

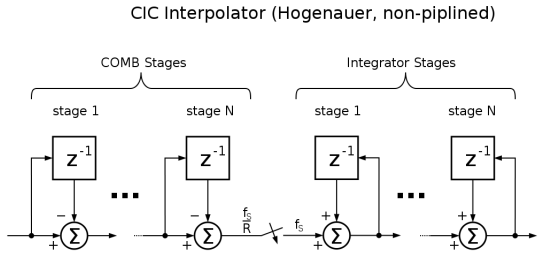


Figura 2.1: Interpolatore CIC con fattore di interpolazione R

Un esempio di filtro interpolatore è rappresentato nella figura 2.1. In riferimento ad un filtro CIC a frequenza f_s, la funzione che la descrive è la 2.1. I parametri del filtro sono:

R = fattore di interpolazione o di decimazione

M = fattore di delay per ogni stadio[1]

N = numero di stadi del filtro

$$H(z) = \left[\sum_{k=0}^{RM-1} z^{-k}\right]^N = \left(\frac{1-z^{RM}}{1-z^{-1}}\right)^N \tag{2.1}$$

Le peculiarità dei filtri CIC sono:

1. Una risposta in fase lineare
2. L'assenza di componenti complesse come i moltiplicatori
3. Una struttura regolare

[1] Generalmente 1 ma a volte anche 2

2.2 Progettazione di un filtro CIC

Un filtro CIC è formato da due componenti essenziali:

- Stadio Comb
- Stadio Integrator

2.2.1 Stadio Comb

Come riportato in [1], uno stadio Comb è realizzato come un sistema che prende in ingresso un dato di N bit ed un bit di carry in, e restituisce un dato di N bit ed il bit di carry out. All'interno dello stadio Comb il dato in ingresso viene sommato al suo negato ritardato di un fattore M. Essendo presente un sommatore, i bit di carry sono necessari per poter realizzare uno stadio Comb in cascata con altri stadi Comb con input inferiori agli N bit richiesti in progettazione.Ad esempio se in ingresso ad uno stadio Comb sono necessari M bit, e le singole celle Comb utilizzabili accettano N bit, tutto lo stadio sarà composto da una cascata di N/M celle Comb.

2.2.2 Stadio Integrator

Sempre in [1] Hogenauer descrive lo stadio Integrator come un insieme a cascata di singole celle Integrator. Al pari delle celle Comb, le celle Integrator accettano N bit in ingresso ed il bit di carry in, e restituiscono in output N bit ed il bit di carry out. Internamente alla cella Integrator è presente un sommatore che accetta come addendi l'input di N bit della cella e l'output ad N bit della stessa cella dall'elaborazione precedente. Il risultato sarà dunque un valore dato dalla somma dell'input al tempo t e dell'output al tempo $t-1$.

2.2.3 Zero-Insertion

Per la realizzazione di un filtro CIC interpolante è necessario inserire tra gli stadi Comb e gli stadi Integrator un moltiplicatore di frequenza che effettui una zero-insertion. Questo blocco fornisce un output ad una frequenza maggiore di quella di ingresso di un fattore R, ed inserisce $R-1$ zeri nei cicli di clock aggiunti. Questo permette di agli stadi Integrator a valle del filtro di effettuare interpolazione riempiendo questi cicli di clock con i valori interpolati.

2.2.4 Crescita dei blocchi

L'implementazione proposta da Hogenauer comporta crescita della dimensione dei blocchi. Infatti essendo il filtro CIC formato da una catena di blocchi contenenti sommatori, la dimensione della parola a N bit cresce fino ad arrivare ad un numero maggiore secondo un determinato fattore di crescita. Per evitare che nel passaggio tra stadi avvenga un overflow, le dimensioni dei blocchi dovranno crescere con un fattore di crescita G dato dall'equazione 2.2, fattore utilizzato all'interno poi dell'equazione di calcolo della dimensione minima del registro per ogni stadio Comb ed Integrator descritta in 2.3 dove B_{in} è la larghezza dell'input del primo blocco e G_j il fattore di crescita calcolato in 2.2.

$$G_i = \begin{cases} 2^j, & j = 1, 2, ..., N \\ \frac{2^{2N-j} R M^{j-N}}{R}, & j = N-1, ..., 2N \end{cases} \tag{2.2}$$

$$W_j = \lceil B_{in} + \log_2 G_j \rceil \tag{2.3}$$

Esiste però una condizione speciale per l'ultimo stadio Comb: solo nel caso in cui $M = 1$, ovvero gli stadi sono a delay unitario, la larghezza dell'ultimo stadio Comb rispetta la 2.4.

$$W_N = B_{in} + N - 1 \tag{2.4}$$

L'unica sorgente di eventuali errori aritmetici è l'ultimo stadio Integrator. Infatti questo stadio fornisce un output con un numero maggiore di bit rispetto a B_{in}, in questo caso è necessario effettuare un troncamento di B_T bit meno significativi in accordo alla 2.5.

$$B_T = W_{2N} - B_{out} \tag{2.5}$$

Capitolo 3

Blocchi base in VHDL

L'implementazione del filtro VHDL è stata effettuata realizzando blocchi elementari e unendoli per ottenere funzionalità più avanzate. I blocchi elementari sono i seguenti:

comb_block_c Blocco Comb di N bit

integrator_block Blocco Integrator di N bit

zero_insertion Blocco che aggiunge $R-1$ zeri tra gli stadi Comb e Integrator

I blocchi su cui vengono costruiti sono:

dff_n Registro Flip-Flop D per introdurre il fattore di delay M visto in 2.1

generic_adder Sommatore interno ad N bit

complement Blocco per il complemento a 2 per la sottrazione

Ulteriori blocchi sono stati utilizzati per effettuare test alternativi e implementare varianti del filtro CIC originale. Tali varianti sono descritte nella sezione 3.3.

3.1 Blocchi Elementari

Sono i blocchi minimi che compongono il filtro CIC. Sono stati utilizzati blocchi generici il cui funzionamento è pensato per un facile riuso.

3.1.1 Flip Flop D

Questo registro è stato realizzato per permettere il delay degli addendi ai sommatori. La sua entity è definita come segue:

```
ENTITY dff_n is
  generic (N:integer );
    port (
      d : in std_logic_vector(N-1 downto 0);
      clk : in std_logic;
      reset: in std_logic;
      q : out std_logic_vector(N-1 downto 0)
    );
END dff_n;
```

3.1.2 Sommatore

L'adder è il medesimo sia per i blocchi Comb che per i blocchi Integrator. Nel primo caso fa la somma dell'ingresso $x[n]$ con $x[n-1]$, dove $x[n-1]$ è l'ingresso nel periodo precedente. La entity vhdl è definita come segue:

```
ENTITY generic_adder is
        generic (N : INTEGER);
        port( a          : in  std_logic_VECTOR (N-1 downto 0);
            b          : in  std_logic_VECTOR (N-1 downto 0);
                carry_in   : in  std_logic;
                s          : out std_logic_VECTOR (N-1 downto 0);
                carry_out  : out std_logic);
END generic_adder;
```

3.1.3 Shifter

Questo blocco è stato concepito per estendere la grandezza in bit di un dato ed adeguarlo a quella di uscita del blocco che lo rice-

verà. Questa operazione serve ad evitare l'uso dei carry out in ogni blocco. Utilizzando l'equazione vista in 2.4, evitiamo il rischio di overflow.

```
entity shifter_n is
        generic (M:integer ; N:integer );
                port (
                input : in std_logic_vector(M-1 downto 0);
                output : out std_logic_vector(N-1 downto 0)
                );
end shifter_n;
```

Lo shifter estende la parola a M bit in ingresso su N bit di uscita.

3.2 Blocchi Principali

Questi sono i blocchi di cui è composto il filtro CIC, così come realizzato in questa implementazione.

3.2.1 Stadio Comb

Lo stadio Comb rappresentato nello schema a blocchi 3.1 è descritto dalla seguente entity vhdl:

```
ENTITY comb_block_c IS
  generic (N: integer );
  port(
    input      : in std_logic_vector(N-1 downto 0);
    carry_in   : in std_logic;
    clock      : in std_logic;
    reset      : in std_logic;
    carry_out : out std_logic;
    output     : out std_logic_vector(N-1 downto 0)
  );
END comb_block_c;
```

Il blocco è rappresentato in figura 3.2 mentre l'entity del blocco complement è il seguente:

```
entity complement is
  generic (N:integer  );
  port (
    input : in std_logic_vector(N-1 downto 0);
    output : out std_logic_vector(N-1 downto 0)
```

```
  );
end complement;
```

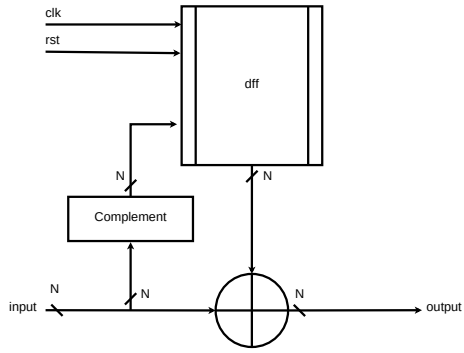


Figura 3.1: Schema a blocchi Stadio Comb

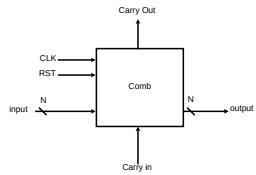


Figura 3.2: Blocco Comb

La presenza del blocco complement serve a lasciare invariato il blocco sommatore. Infatti il dato memorizzato nel flip flop è il complemento a due del dato di ingresso, in modo che il sommatore operi una somma normale tra dati.

3.2.2 Stadio Integrator

La struttura interna dello stadio Integrator è molto simile a quanto visto per il Comb: essendo però una funzione di somma, non è necessario il blocco che faccia il complemento a 2 del dato. La entity è dichiarata nel seguente modo:

```
ENTITY integrator_block IS
  generic (N: integer );
  port(
    input      : in std_logic_vector(N-1 downto 0);
    carry_in   : in std_logic;
    clock     : in std_logic;
    reset     : in std_logic;
    carry_out : out std_logic;
    output     : out std_logic_vector(N-1 downto 0)
  );
END integrator_block;
```

Mentre i diagrammi a blocchi sono quelli di figura 3.3 e 3.4.

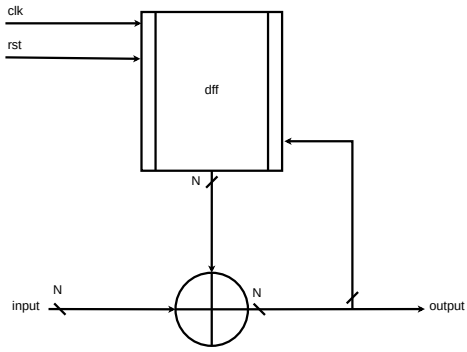


Figura 3.3: Schema a blocchi Stadio Integrator

3.2.3 Zero-Insertion

Questo blocco inserisce, ad una frequenza R volte maggiore di quella di ingresso del filtro, degli zeri negli R-1 cicli aggiunti. La sua

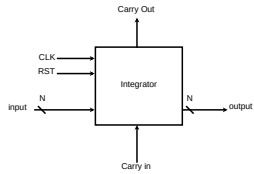


Figura 3.4: Blocco Integrator

entity è la seguente:

```
ENTITY zero_insertion IS
  generic (N: integer);
  port(
    ingresso        : in std_logic_vector(N-1 downto 0);
    clock     : in std_logic;
    reset     : in std_logic;
    uscita     : out std_logic_vector(N-1 downto 0)
  );
END zero_insertion;
```

Il processo di inserimento degli zeri viene eseguito dal seguente process:

```
process(clock)
  variable counter : INTEGER := 1;
begin

if rising_edge(clock) then
  if (counter=0) and reset='1' then
  uscita <= ingresso;
  counter:=counter+1;
  else
   for i in N-1 downto 0 LOOP
   uscita(i) <=  '0';
  end LOOP;
  counter := counter+1 ;
   if ((counter mod 4) = 0 )  then
   counter:=0;
  end if;
end if;
end if;
end process;
```

3.3 Blocchi per le varianti

3.3.1 Hold-Insertion

È possibile ottimizzare il funzionamento del filtro CIC risparmiando un blocco Comb ed un blocco Integrator. Tale ottimizzazione è possibile eliminando tali blocchi e sostituendo lo zero-insertion con un blocco di hold-insertion. Il funzionamento di tale blocco è il medesimo di zero-insertion, con la sola eccezione che nei cicli di clock aggiunti l'inserimento degli zeri viene sostituito dalla ripetizione del segnale di input. Questo è possibile grazie alle proprietà degli stadi Comb ed Integrator.

3.3.2 Stadio Comb con inverter

Nella ideazione originale di Hogenauer [1] del filtro è previsto uno stadio Comb con un un sommatore ed un inverter. La tecnica originale infatti inseriva dei carry in unitari a tutti gli stadi Comb, e recuperava i carry out nell'ultimo stadio. Questo è possibile poichè è richiesto che i bit seguano una implementazione di tipo ad anello. Tale implementazione è realizzabile sia con un inverter e l'uso accurato dei carry, che con l'aritmetica del complemento a 2. In questo progetto è stata utilizzata la seconda soluzione.

Capitolo 4

Implementazione del Filtro CIC

4.1 Filtro CIC senza variante

Il filtro CIC è realizzato tramite l'inserimento di N stadi Comb ed N stadi integrator. Seguendo la formulazione esposta in [1], il dimensionamento dei blocchi segue l'equazione 2.3, ovvero:

$$B_{in} = 16 \begin{cases} W_1 = 17 \\ W_2 = 18 \\ W_3 = 19 \\ W_4 = 20 \\ W_5 = 20 \\ W_6 = 20 \\ W_7 = 21 \\ W_8 = 22 \end{cases} \tag{4.1}$$

Realizzando i blocchi con le dimensioni di 4.1, abbiamo la garanzia di evitare overflow preoccupandoci solo di eventuali arrotondamenti

sull'ultimo stadio Integrator.

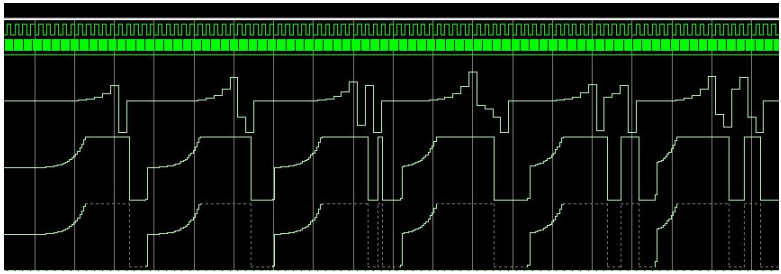


Figura 4.1: Approssimazione al MSB

Lo schema a blocchi del filtro implementato è rappresentato in figura 4.2.

Per realizzare l'interpolazione è necessario fornire un clock esterno ad una frequenza R volte superiore al clock iniziale. Una alternativa sarebbe stata inserire un oscillatore interno ad una frequenza di f_s dove $\frac{f_s}{R}$ è la frequenza di ingresso del filtro CIC.

L'approssimazione dell'uscita è stata fatta sia scartando i bit più significativi in eccesso, che i bit meno significativi. La prima soluzione taglia i valori più alti del segnale di uscita ma a valori bassi l'uscita interpola perfettamente l'ingresso, mentre la seconda soluzione diventa meno sensibile a valori piccoli ma riesce a riprodurre fedelmente il segnale di ingresso interpolando anche i picchi di input.

L'immagine 4.1 mostra l'output del filtro (segnale centrale) ad un dato segnale confrontato con l'output del filtro senza approssimazione (terzo segnale). Il segnale di ingresso è il primo. Si può vedere come in caso di approssimazione al MSB il taglio dei valori alti possa diventare inaccettabile. L'immagine 4.5 invece mostra che in presenza di input in un range di valori attorno allo zero, il segnale interpolato segue l'ingresso in modo esatto.

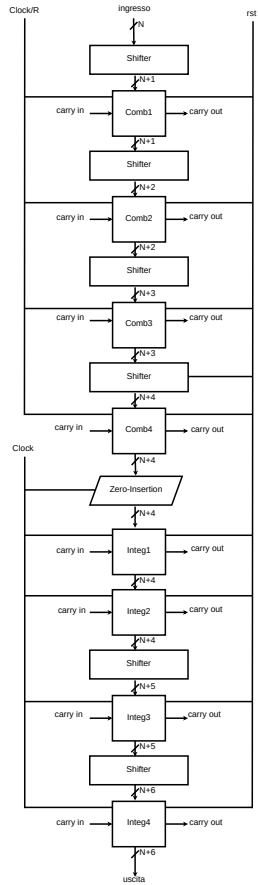


Figura 4.2: Filtro CIC

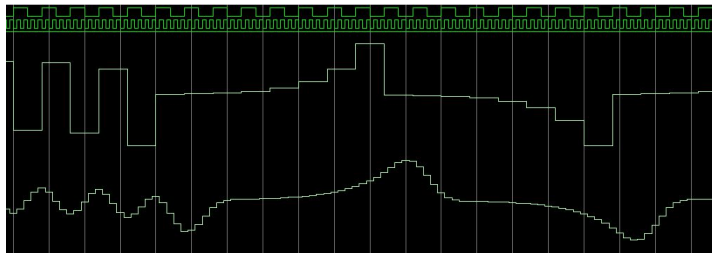


Figura 4.3: Approssimazione al LSB

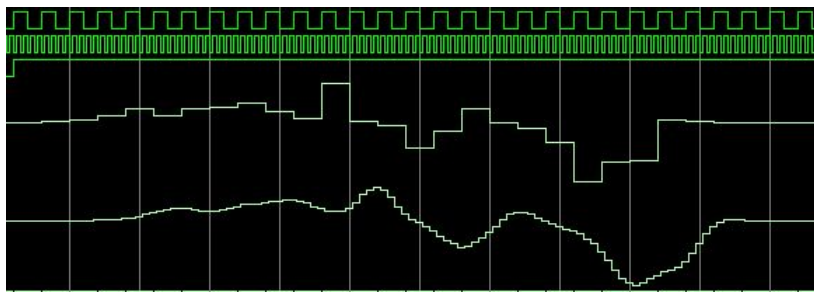


Figura 4.4: Approssimazione al LSB

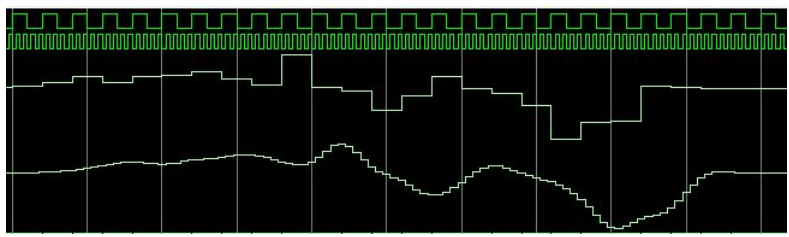


Figura 4.5: Approssimazione al MSB

Le figure 4.3 e 4.4 mostrano la risposta ai medesimi segnali di input, nel caso in cui l'approssimazione venga effettuata ai bit meno significativi. Il codice relativo all'approssimazione di tipo LSB è il seguente:

```
  process(segnale9)
    begin
      TRUNK: for i in 0 to N-1 loop
        uscita(i) <= segnale9(i);
      end loop;
    if segnale9(M-1) = '0' then
      DIO1: for i in M-2 downto N-1 loop
        if segnale9(i) /= '0' then
          uscita <= "0111111111111111";
          exit;
        end if;
      end loop;
    end if;
    if segnale9(M-1) = '1' then
      DIO2: for i in M-2 downto N-1 loop
        if segnale9(i) /= '1' then
          uscita <= "1000000000000000";
          exit;
        end if;
      end loop;
    end if;
 end process;
```

Invece l'approssimazione al MSB ha richiesto semplicemente un processo che escludesse gli $M - N$ bit meno significativi, dove:

M Numero di bit in uscita all'ultimo stadio Integrator

N Numero di bit in ingresso al filtro

4.2 Filtro CIC con variante

Nel corso degli anni la letteratura scientifica ha prodotto numerosi articoli su modifiche strutturali [4] [2] e varianti del filtro CIC [3]. La variante utilizzata per questa versione del filtro CIC è quella suggerita da Losada e Lyons [3] che prevede l'eliminazione di uno stadio Comb ed uno Stadio Integrator e la sostituzione del blocco di zero-insertion con un blocco hold-insertion.

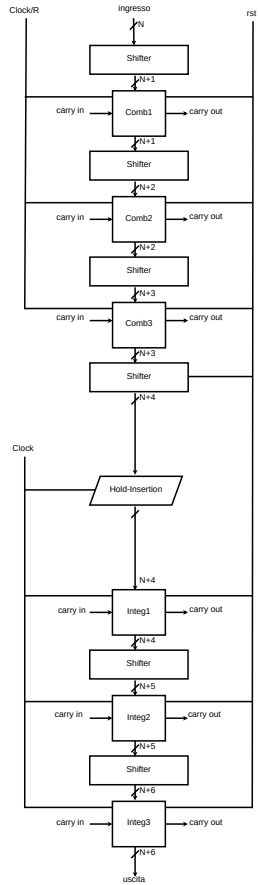


Figura 4.6: Filtro CIC

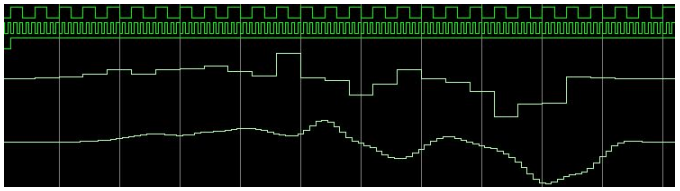


Figura 4.7: Output del filtro CIC con variante

Il diagramma a blocchi è quello della figura 4.6, mentre la figura 4.7 rappresenta la risposta del filtro ad un segnale. Come si evince dalla figura, il funzionamento rimane invariato pur avendo utilizzato due stadi in meno.

Capitolo 5

Conclusioni

I filtri Cascated Integrated Comb rappresentano una ottima soluzione per le operazioni di decimazione ed interpolazione. Si è visto infatti come la realizzazione non implica l'utilizzo di complessi elementi come moltiplicatori, bensì di semplici sommatori e registri. Questo comporta notevoli vantaggi sia a livello di progettazione che di complessità realizzativa. È emersa anche la notevole capacità di questi filtri di essere realizzati con varianti che ne migliorano le performance e che ne semplificano ulteriormente l'implementazione. Nella fattispecie il filtro CIC interpolatore si è rivelato un filtro con grandi prestazioni nei casi di approssimazione al MSB. Questa approssimazione comporta una perdita di sensibilità negli intorni del segnale prossimi allo zero, ma permette di mantenere una interpolazione molto fedele al segnale di input pur avendo in uscita lo stesso numero di bit dell'ingresso. È risultato però necessario fornire al filtro un ulteriore segnale di clock esterno, per permettere l'inserimento di ulteriori campioni al segnale tramite il blocco zero-insertion o hold-insertion (a seconda della variante utilizzata). In conclusione la struttura regolare di questo filtro, unito ai vantaggi sopra descritti, rende il filtro CIC un ottimo filtro di

interpolazione tra quelli appartenenti alla classe di filtri digitali a fase lineare e risposta impulsiva.

Appendice A

Sorgenti VHDL dei filtri

A.1 Sorgenti Filtro CIC senza variante e con LSB

```
LIBRARY IEEE;
USE IEEE.std_logic_1164.all;
use ieee.numeric_std.all;

ENTITY cic_normal IS
  generic (N: integer :=16);
  port(
    ingresso  : in std_logic_vector(N-1 downto 0);
    clock_slow     : in std_logic;
    clock_fast     : in std_logic;
    reset       : in std_logic;
    uscita     : out std_logic_vector(N-1 downto 0)
  );
END cic_normal;

architecture BEHAVIOURAL of cic_normal is

  component integrator_block
  generic (N:integer );
  port(
    input      : in std_logic_vector(N-1 downto 0);
    carry_in   : in std_logic;
    clock     : in std_logic;
    reset     : in std_logic;
    carry_out : out std_logic;
    output     : out std_logic_vector(N-1 downto 0)
  );
```

```
end component;

component comb_block_c
  generic (N:integer );
  port(
    input      : in std_logic_vector(N-1 downto 0);
    carry_in   : in std_logic;
    clock     : in std_logic;
    reset     : in std_logic;
    carry_out : out std_logic;
    output     : out std_logic_vector(N-1 downto 0)
  );
end component;

component shifter_n
generic (M:integer ; N:integer);
  port (
    input : in std_logic_vector(M-1 downto 0);
    output : out std_logic_vector(N-1 downto 0)
  );
end component;

component zero_insertion
  generic (N: integer);
  port(
    ingresso       : in std_logic_vector(N-1 downto 0);
    clock     : in std_logic;
    reset     : in std_logic;
    uscita     : out std_logic_vector(N-1 downto 0)
  );
END component;

constant M : integer := N+6;

type myarray is array(7 downto 0) of std_logic;
signal carry : myarray ;

signal ingressoe : std_logic_vector(N downto 0) ;
signal segnale1 : std_logic_vector(N downto 0) ;
signal segnale2 : std_logic_vector(N+1 downto 0) ;
signal segnale3 : std_logic_vector(N+2 downto 0) ;
signal segnale4 : std_logic_vector(N+3 downto 0) ;
signal segnale5 : std_logic_vector(N+3 downto 0) ;
signal segnale6 : std_logic_vector(N+3 downto 0) ;
signal segnale7 : std_logic_vector(N+3 downto 0) ;
signal segnale8 : std_logic_vector(N+4 downto 0) ;
signal segnale9 : std_logic_vector(N+5 downto 0) ;

signal segnale1e : std_logic_vector(N+1 downto 0) ;
signal segnale2e : std_logic_vector(N+2 downto 0) ;
signal segnale3e : std_logic_vector(N+3 downto 0) ;
signal segnale4e : std_logic_vector(N+4 downto 0) ;
signal segnale5e : std_logic_vector(N+4 downto 0) ;
signal segnale6e : std_logic_vector(N+4 downto 0) ;
signal segnale7e : std_logic_vector(N+4 downto 0) ;
signal segnale8e : std_logic_vector(N+5 downto 0) ;
```

```
signal check : std_logic := '0';

begin

  SH1: shifter_n
  generic map( N,N+1)
  port map(ingresso,ingressoe);

  COMB0: comb_block_c
  generic map(N+1)
  port map(ingressoe,'0',clock_slow,reset,carry(0),segnale1);

  SH2: shifter_n
  generic map(N+1,N+2)
  port map(segnale1,segnale1e);

  COMB1: comb_block_c
  generic map(N+2)
  port map(segnale1e,'0',clock_slow,reset,carry(1),segnale2);

  SH3: shifter_n
  generic map(N+2,N+3)
  port map(segnale2,segnale2e);

  COMB2: comb_block_c
  generic map(N+3)
  port map(segnale2e,'0',clock_slow,reset,carry(2),segnale3);

  SH4: shifter_n
  generic map(N+3,N+4)
  port map(segnale3,segnale3e);

  COMB3: comb_block_c
  generic map(N+4)
  port map(segnale3e,'0',clock_slow,reset,carry(3),segnale4);

  ZIN: zero_insertion
  generic map(N+4)
  port map(segnale4,clock_fast,reset,segnale5);

  INT1: integrator_block
  generic map(N+4)
  port map(segnale5,'0',clock_fast,reset,carry(4),segnale6);

  INT2: integrator_block
  generic map(N+4)
  port map(segnale6,'0',clock_fast,reset,carry(5),segnale7);

  SH5: shifter_n
  generic map(N+4,N+5)
  port map(segnale7,segnale7e);

  INT3: integrator_block
  generic map(N+5)
  port map(segnale7e,'0',clock_fast,reset,carry(6),segnale8);

  SH6: shifter_n
```

```
  generic map(N+5,N+6)
  port map(segnale8,segnale8e);

  INT4: integrator_block
  generic map(N+6)
  port map(segnale8e,'0',clock_fast,reset,carry(7),segnale9);

  process(segnale9)
    begin
      TRUNK: for i in 0 to N-1 loop
        uscita(i) <= segnale9(i);
      end loop;
    if segnale9(M-1) = '0' then
      DIO1: for i in M-2 downto N-1 loop
        if segnale9(i) /= '0' then
          uscita <= "0111111111111111";
          exit;
        end if;
      end loop;
    end if;
    if segnale9(M-1) = '1' then
      DIO2: for i in M-2 downto N-1 loop
        if segnale9(i) /= '1' then
          uscita <= "1000000000000000";
          exit;
        end if;
      end loop;
    end if;

 end process;

end BEHAVIOURAL;
```

A.2 Sorgenti filtro CIC con variante e MSB

```
LIBRARY IEEE;
USE IEEE.std_logic_1164.all;
use ieee.numeric_std.all;



ENTITY cic_easy_lsb IS
  generic (N: integer :=16);
  port(
    ingresso  : in std_logic_vector(N-1 downto 0);
    clock_slow     : in std_logic;
    clock_fast     : in std_logic;
    reset       : in std_logic;
  uscita     : out std_logic_vector(N-1 downto 0)
  );
END cic_easy_lsb;
```

```
architecture BEHAVIOURAL of cic_easy_lsb is

  component integrator_block
  generic (N:integer );
  port(
    input      : in std_logic_vector(N-1 downto 0);
    carry_in   : in std_logic;
    clock     : in std_logic;
    reset     : in std_logic;
    carry_out : out std_logic;
    output     : out std_logic_vector(N-1 downto 0)
  );
end component;

component comb_block_c
  generic (N:integer );
  port(
    input      : in std_logic_vector(N-1 downto 0);
    carry_in   : in std_logic;
    clock     : in std_logic;
    reset     : in std_logic;
    carry_out : out std_logic;
    output     : out std_logic_vector(N-1 downto 0)
  );
end component;

component shifter_n
generic (M:integer ; N:integer);
  port (
    input : in std_logic_vector(M-1 downto 0);
    output : out std_logic_vector(N-1 downto 0)
  );
end component;


component hold_insertion
  generic (N: integer);
  port(
    ingresso       : in std_logic_vector(N-1 downto 0);
    clock     : in std_logic;
    reset     : in std_logic;
    uscita     : out std_logic_vector(N-1 downto 0)
  );
END component;

constant M : integer := N+6;

type myarray is array(7 downto 0) of std_logic;
signal carry : myarray ;

signal ingressoe : std_logic_vector(N downto 0) := (others =>'0') ;
signal segnale1 : std_logic_vector(N downto 0):= (others =>'0') ;
signal segnale2 : std_logic_vector(N+1 downto 0):= (others =>'0') ;
signal segnale3 : std_logic_vector(N+2 downto 0):= (others =>'0') ;
signal segnale6 : std_logic_vector(N+3 downto 0):= (others =>'0') ;
signal segnale7 : std_logic_vector(N+3 downto 0):= (others =>'0') ;
signal segnale8 : std_logic_vector(N+4 downto 0) := (others =>'0');
```

```
signal segnale9 : std_logic_vector(N+5 downto 0) := (others =>'0');

signal segnale1e : std_logic_vector(N+1 downto 0) := (others =>'0');
signal segnale2e : std_logic_vector(N+2 downto 0):= (others =>'0') ;
signal segnale3e : std_logic_vector(N+3 downto 0):= (others =>'0') ;
signal segnale7e : std_logic_vector(N+4 downto 0):= (others =>'0') ;
signal segnale8e : std_logic_vector(N+5 downto 0):= (others =>'0') ;

signal check : std_logic := '0';

begin

  SH1: shifter_n
  generic map( N,N+1)
  port map(ingresso,ingressoe);

  COMB0: comb_block_c
  generic map(N+1)
  port map(ingressoe,'0',clock_slow,reset,carry(0),segnale1);

  SH2: shifter_n
  generic map(N+1,N+2)
  port map(segnale1,segnale1e);

  COMB1: comb_block_c
  generic map(N+2)
  port map(segnale1e,'0',clock_slow,reset,carry(1),segnale2);

  SH3: shifter_n
  generic map(N+2,N+3)
  port map(segnale2,segnale2e);

  COMB2: comb_block_c
  generic map(N+3)
  port map(segnale2e,'0',clock_slow,reset,carry(2),segnale3);

  SH4: shifter_n
  generic map(N+3,N+4)
  port map(segnale3,segnale3e);

    HIN: hold_insertion
    generic map(N+4)
    port map(segnale3e,clock_fast,reset,segnale6);

  INT2: integrator_block
  generic map(N+4)
  port map(segnale6,'0',clock_fast,reset,carry(5),segnale7);

  SH5: shifter_n
  generic map(N+4,N+5)
  port map(segnale7,segnale7e);

  INT3: integrator_block
  generic map(N+5)
  port map(segnale7e,'0',clock_fast,reset,carry(6),segnale8);
```

```
  SH6: shifter_n
  generic map(N+5,N+6)
  port map(segnale8,segnale8e);

  INT4: integrator_block
  generic map(N+6)
  port map(segnale8e,'0',clock_fast,reset,carry(7),segnale9);

  process(segnale9)
    begin
     TRUNK: for i in 0 to N-1 loop
       uscita(i) <= segnale9(M-N+i);
     end loop;
 end process;

end BEHAVIOURAL;
```

Appendice B

Sorgenti VHDL dei testbench

B.1 Testbench con input controllato

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

ENTITY cic_easy_lsb_tb2 IS
END cic_easy_lsb_tb2;

ARCHITECTURE cic_test OF cic_easy_lsb_tb2 IS
  component cic_easy_lsb
    generic (N: integer:=16);
  port(
    ingresso  : in std_logic_vector(N-1 downto 0);
    clock_slow     : in std_logic;
    clock_fast     : in std_logic;
    reset       : in std_logic;
uscita     : out std_logic_vector(N-1 downto 0)
  );
END component;

  CONSTANT N       :  INTEGER  := 16;
   CONSTANT MckPer  :  TIME     := 400 ns;
   CONSTANT SlvPer  : TIME    :=  100 ns;
   CONSTANT TestLen :  INTEGER  := 70;


   signal clock1 : std_logic := '0';
```

```
  signal clock2 : std_logic := '0';
  signal reset : std_logic := '0';

  signal input : std_logic_vector(N-1 downto 0):= "0000000000000000" ;
  signal output : std_logic_vector(N-1 downto 0);

  SIGNAL clk_cycle : INTEGER ;
  SIGNAL Testing: Boolean := True;

  BEGIN

 I: cic_easy_lsb
 PORT MAP(input,clock1,clock2,reset,output);
    clock1 <= NOT clock1 AFTER MckPer/2 WHEN Testing ELSE '0';
    clock2 <= NOT clock2 AFTER SlvPer/2 WHEN Testing ELSE '0';

   Test_Proc: PROCESS(clock1)
     VARIABLE count: INTEGER:= 0;
  BEGIN
    clk_cycle <= (count+1)/2;

    CASE count IS

      WHEN 1 => input <= "0000000000000000" ; reset <= '1';

        WHEN  3  => input <= "0000000000000001";
        WHEN  5  => input <= "0000000000000010";
        WHEN  7  => input <= "0000000000000101";
        WHEN  9  => input <= "0000000000001010";
        WHEN  11  => input <=  "0000000000000101";
        WHEN  13  => input <= "0000000000001010";
        WHEN  15  => input <= "0000000000001011";
        WHEN  17  => input <= "0000000000001111";
        WHEN  19  => input <= "0000000000001000";
        WHEN  21  => input <=  "0000000000000011";
        WHEN  23  => input <=  "0000000000011110";
        WHEN  25  => input <=  "0000000000000001";

        WHEN  27  => input <= "1111111111111101";
        WHEN  29 => input <= "1111111111101100";
        WHEN  31  => input <= "1111111111111001";
        WHEN  33  => input <= "0000000000001010";
        WHEN  35  => input <=  "1111111111111111";
        WHEN  37  => input <= "1111111111111011";
        WHEN  39  => input <= "1111111111110001";
        WHEN  41  => input <= "1111111111010011";
        WHEN  43  => input <= "1111111111100010";
        WHEN  45  => input <=  "1111111111100011";
        WHEN  47  => input <=  "0000000000000010";
        WHEN  49  => input <=  "0000000000000001";

        WHEN  51  => input <=  "0000000000000000";

         WHEN (TestLen - 1) =>   Testing <= False;
         WHEN OTHERS => NULL;
    END CASE;
```

```
    count:= count + 1;
  END PROCESS Test_Proc;

  END cic_test;
```

B.2 Testbench con input generato randomicamente

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

-- test usando il generatore di segnale

ENTITY cic_easy_lsb_tb IS
END cic_easy_lsb_tb;

ARCHITECTURE cic_test OF cic_easy_lsb_tb IS
  component cic_easy_lsb
    generic (N: integer:=16);
  port(
    ingresso  : in std_logic_vector(N-1 downto 0);
    clock_slow     : in std_logic;
    clock_fast     : in std_logic;
    reset       : in std_logic;
    uscita     : out std_logic_vector(N-1 downto 0)
  );
END component;

component random is
    generic ( width : integer :=  16 );
port (
      clk : in std_logic;
      random_num : out std_logic_vector (width-1 downto 0)
    );
end component;

  CONSTANT N       :  INTEGER  := 16;
   CONSTANT MckPer  :  TIME     := 400 ns;
   CONSTANT SlvPer  : TIME    :=  100 ns;
   CONSTANT TestLen :  INTEGER  := 700;


   signal clock1 : std_logic := '0';
   signal clock2 : std_logic := '0';
   signal reset : std_logic := '0';

   signal rinput : std_logic_vector(N-1 downto 0) ;
   signal output : std_logic_vector(N-1 downto 0);

   SIGNAL clk_cycle : INTEGER ;
```

```
   SIGNAL Testing: Boolean := True;

   BEGIN

  R: random
  port map(clock1,rinput);

  I: cic_easy_lsb
  PORT MAP(rinput,clock1,clock2,reset,output);
     clock1 <= NOT clock1 AFTER MckPer/2 WHEN Testing ELSE '0';
     clock2 <= NOT clock2 AFTER SlvPer/2 WHEN Testing ELSE '0';

    Test_Proc: PROCESS(clock1)
      VARIABLE count: INTEGER:= 0;
   BEGIN
     clk_cycle <= (count+1)/2;

     CASE count IS

       WHEN 1 =>  reset <= '1';

          WHEN (TestLen - 1) =>   Testing <= False;
          WHEN OTHERS => NULL;
     END CASE;

     count:= count + 1;
   END PROCESS Test_Proc;

   END cic_test;
```

B.3 Generatore randomico di segnale

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

entity random is
    generic ( width : integer :=  16 );
port (
      clk : in std_logic;
      random_num : out std_logic_vector (width-1 downto 0)
    );
end random;

architecture Behavioral of random is
begin
process(clk)
        variable rand_temp : std_logic_vector(width-1 downto 0):=
(width-1 =>'1',others => '0');
        variable temp : std_logic := '0';

        begin
                if(rising_edge(clk)) then
```

```
                temp := rand_temp(width-1) xor rand_temp(width-2);
                rand_temp(width-1 downto 1) := rand_temp(width-2 downto 0);
                rand_temp(0) := temp;
        end if;
        random_num <= rand_temp;

end process;

end Behavioral;
```

Bibliografia

[1] Eugene B. Hogenauer. An economical class of digital filters for decimation and interpolation. *IEEE TRANSACTIONS ON ACOUSTICS, SPEECH, AND SIGNAL PROCESSING*, 1981.

[2] Zhang Yuhua Peng Hong. The efficient design and modification of cascaded integrator comb filter. *WASE International Conference on Information Engineering*, 2010.

[3] Richard Lyons Ricardo A. Losada. Reducing cic filter complexity. *IEEE SIGNAL PROCESSING MAGAZINE*, 2006.

[4] Gerhard Fetrwe Tiin Hentsclzel. Reduced complexity comb-filters for decimation and interpolation in mobile communications terminals.