UNIVERSITÀ
DEGLI STUDI
DI BRESCIA

Progetto di Sistemi Elettronici Digitali per l'Internet of Things

Relazione di progetto

Implementazione di un filtro CIC su FPGA Cyclone V

Belotti Nicolò, Shqepa Frenki

1 Filtro Cascaded-Integrator-Comb (CIC)

I filtri CIC (Cascaded Integrator Comb) sono stati proposti per la prima volta da Eugene B. Hogenauer nel 1979 come un'alternativa "economica" agli altri filtri digitali utilizzati. I filtri CIC, al contrario dei classici filtri FIR e IIR, non hanno bisogno di moltiplicatori; infatti, includono nella loro struttura solo sommatori, sottrattori (che si possono implementare come sommatori in complemento a 2), registri e blocchi per il cambio di frequenza. Questa struttura, in vista di un'implementazione su hardware, permette di risparmiare risorse fisiche.

Questo progetto si concentra sulla realizzazione del filtro CIC sull'hardware della FPGA Altera Cyclone V. Il dato viene letto dall'ADC on-board della DE10-nano, filtrato e, tramite l'HPS ARM integrato nel SoC, viene inviato alla piattaforma cloud di ThingSpeak. È quindi necessario implementare anche un protocollo che faccia comunicare la FPGA con l'HPS, ad esempio SPI.

1.1 Struttura del filtro: blocchi fondamentali

Un filtro CIC di ordine N è composto tre blocchi fondamentali:

- N stadi COMB;
- stadio di inserimento di zeri;
- N stadi integratori (INT).

1.1.1 Stadio comb

Lo stadio comb esegue la differenza tra il campione attuale e quello ad un valore precedente, dove quest'ultimo è definito dal fattore di ritardo M

$$y[n] = x[n] - x[n - M] \tag{1}$$

Questo stadio sta di fatto decimando il segnale, portando quindi ad un'attenuazione delle alte frequenze. Per implementare in digitale questo stadio, bisogna pensare ai passi per arrivare al risultato y[n].

Il campione x[n] va tenuto memorizzato per M periodi e per farlo si sfrutta un numero M di flip flop di tipo D. Dopo M campioni, all'attuale x[n] si può sottrarre quello che ora è diventato x[n-M] e per fare la differenza, si sfrutta la somma in complemento a due. È necessaria quindi l'implementazione di un sommatore e, se si vogliono mettere in cascata più stadi comb, è necessario includere i bit di carry.

Se allora si prende in ingresso allo stadio un dato a N bit e il bit di carry in, lo stadio porta in uscita un bit la somma e il bit di carry out, come visibile in Figura 1.

Ricordando che i filtri CIC sono stati introdotti per il risparmio di risorse hardware, questo stadio fa capire perché l'ordine sia comb - zero insertion – integrator. Il blocco comb è quello più dispendioso in termini di memoria perché utilizza $M \cdot N_{bits}$ flip flops per la linea di ritardo. Far lavorare allora questo stadio dopo l'inserimento degli zeri, a frequenza R volte più alta, significa:

- sprecare memoria perché servirebbe una linea di ritardo R volte più grande;
- spreco di potenza perché la maggior parte dei campioni sono zeri.

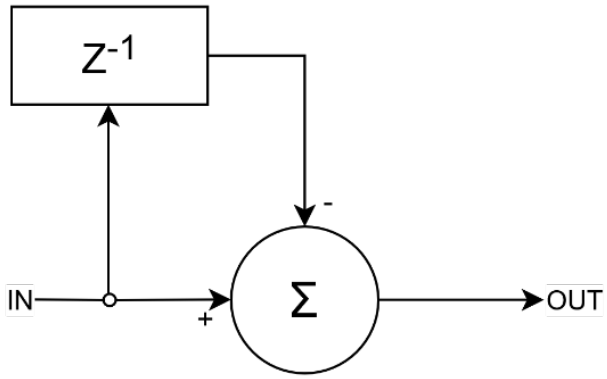

Figura 1. Schema a blocchi dello stadio comb.

1.1.2 Stadio di inserimento di zeri

Questo stadio si pone tra gli stadi COMB e gli stadi INT, questi ultimi lavorano alla frequenza f_{fast} mentre i primi a $\frac{f_{fast}}{R}$. Il suo segnale d'uscita si aggiorna alla stessa frequenza degli integratori ed è costruito inserendo R-1 zeri dopo ogni campione del segnale in ingresso. Con questa modalità di inserimento il filtro complessivo ha funzionalità di interpolatore, aumentando la frequenza dei campioni in uscita di un fattore R.

1.1.3 Stadio integratore

Lo stadio integratore esegue la somma tra campione attuale del segnale e tutti i precedenti come riportato dall'equazione di seguito; quindi, il valore dell'uscita y[n] dipende quindi dalla somma di tutti i campioni precedenti y[n-1]

$$y[n] = y[n-1] + x[n] \tag{3}$$

Come nel blocco comb, questo stadio accetta in ingresso segnali ad N bit e il carry in, restituendo in uscita il risultato a N bit e il bit di carry out.

Per implementare il blocco integratore in digitale è necessario ricordare la somma di tutti i campioni precedenti, grazie ad un flip flop di tipo D, e sommarla al campione attuale con un sommatore.

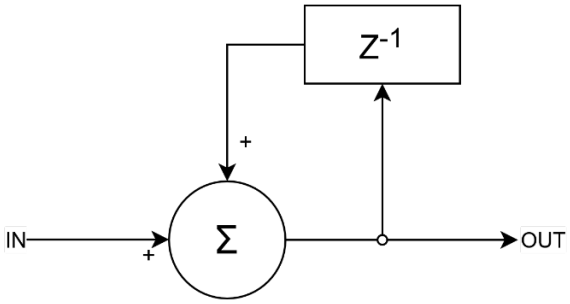

Figura 2. Schema a blocchi dello stadio integrator.

1.1.4 Problema della crescita dei blocchi

La presenza dei sommatori all'interno degli stadi del filtro comporta la crescita della dimensione del dato man mano che avanza nella catena che, se non viene opportunamente gestita può causare overflow. Per questo motivo solitamente si effettua un dimensionamento progressivo degli stadi in base al fattore di crescita che ogni stadio introduce rispetto al precedente.

La crescita massima del dato viene valutata sulla base del peggior segnale che può essere portato in ingresso al filtro ed è descritta dall'equazione 4, con R il fattore di interpolazione, M la profondità della memoria di ogni stadio e N l'ordine del filtro.

Con j compreso tra 1 e 2N, nel nostro caso in cui N=4 gli indici tra 1 e 4 corrispondono agli stadi COMB mentre quelli tra 5 e 8 corrispondono agli stati INT.

$$G_j = \begin{cases} 2^j & j = 1,2,\dots,N \\ \dfrac{2^{(2N-j)}(RM)^{j-N}}{R} & j = N+1,\dots,2N \end{cases} \tag{4}$$

Data B_{in} la dimensione in bit del dato in ingresso al filtro, Il numero di bit massimo per i registri del j-esimo stadio sarà quindi

$$W_j = \lceil B_{in} + \log_2(G_j) \rceil \tag{5}$$

1.2 Struttura del filtro: specifiche

Dalla letteratura è nota la funzione di risposta in frequenza di un filtro CIC interpolatore di ordine N, ritardo dei blocchi comb M e fattore di interpolazione R:

$$H(z) = \left[\frac{1 - z^{-RM}}{1 - z^{-1}}\right]^N \tag{7}$$

Per definire le specifiche del progetto, bisogna partire con la dimensione dei dati in ingresso al filtro CIC, perché sappiamo che questi sono caratterizzati da una crescita, come riportato nell'eq.5.

I dati che entrano nel filtro provengono dal convertitore analogico digitale incluso nella scheda di sviluppo (LTC2308). Questo dispositivo è costituito da 8 canali a 12 bit, di cui utilizzeremo solo il canale 0 in modalità single ended; quindi, riferito a massa e con fondo scala a 5 V. La dimensione del dato in ingresso al filtro è dunque $\boldsymbol{B_{in} = 12.}$

Scegliendo un filtro del quarto ordine (quattro integratori e quattro comb) quindi **N=4**, con profondità della memoria **M=1,** si utilizza l'eq.5 per calcolare la dimensione massima dei registri dei vari stadi.

La dimensione massima per l'ultimo stadio comb è quindi

$$W_4 = \lceil B_{in} + \log_2(2^4) \rceil = 16$$

Mentre l'ultimo stadio integratore avrà dimensione

$$W_8 = \left\lceil B_{in} + \log_2\left(\frac{(RM)^4}{R}\right)\right\rceil$$

Per i casi analizzati ovvero R=4 e R=10, si avrà rispettivamente $W_8 = 18$ e $W_8 = 22$.

Per praticità è stato scelto di istanziare tutti gli stadi alla loro dimensione massima, W_4 per i COMB e W_8 per gli INT.

L'uscita dell'ADC è un segnale unsigned e viene esteso a 16 bit semplicemente ponendo a 0 gli MSB rimanenti. Tra l'ultimo stadio COMB (16 bit) ed il primo stadio INT (18 o 22 bit) il segnale è signed in complemento a due e viene perciò inserito uno shifter in modo tale da estendere la dimensione mantenendo la notazione. In uscita dall'ultimo integratore il segnale viene troncato a 16 bit sacrificando i bit meno significativi per poi essere passato all'hardware che gestisce l'SPI.

Per definire le frequenze di lavoro del filtro utilizziamo la velocità massima dell'SPI come fattore limitante: la trasmissione avviene alla frequenza massima di 1 MHz quindi per trasmettere 16 bit sono necessari 16 microsecondi. Scegliendo di mantenere il canale inattivo per il 50% del tempo (MISO=0) limitiamo il throughput ad un campione ogni 32 microsecondi, quindi 31,25 ksps.

Questo vincolo definisce la frequenza di CLK_FAST che va agli integratori: 31,25 KHz, ottenuta dividendo il clock principale della FPGA a 50 MHz per un fattore 1600.

CLK_SLOW per gli stadi COMB viene ottenuto da CLK_FAST dividendo la sua frequenza per il fattore R, ottenendo 7,812 KHz per R=4 o 3,125 KHz per R=10. Indipendentemente dal caso, L'ADC viene configurato per lavorare alla frequenza di campionamento più alta, quindi 7,812 ksps.

In ingresso al filtro, prima di COMB1, viene messo un DFF aggiornato con CLK_SLOW che svolge la funzione di zero-order-hold che mantiene stabile il campione anche se l'ADC lo aggiorna.

La frequenza di campionamento effettiva del sistema è quindi pari alla frequenza di CLK_SLOW.

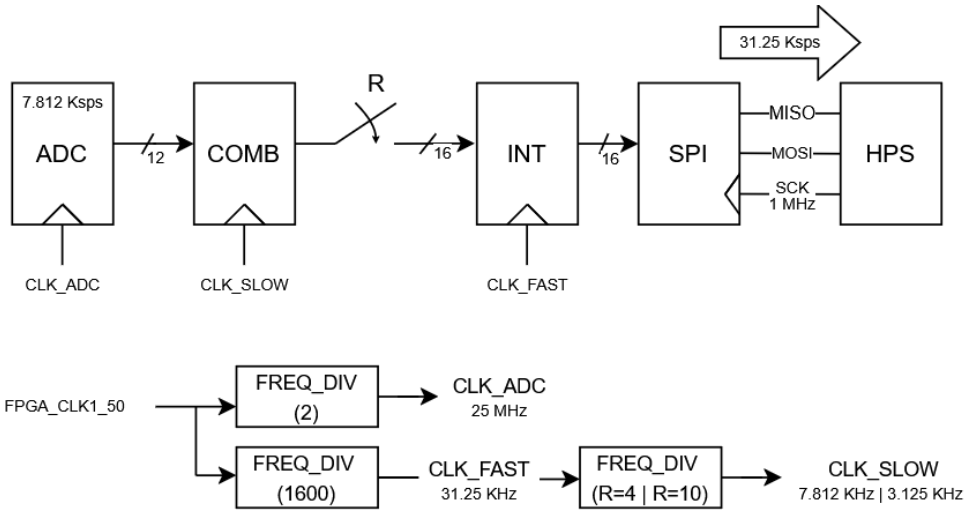

Figura 3. Diagramma sintetico del sistema con i segnali utilizzati.

1.3 Implementazione VHDL degli stadi

1.3.1 D Flip Flop

Il flip flop D è stato realizzato per accettare in ingresso N bit, permette di aggiungere il ritardo tra gli addendi dei sommatori. La dimensione del dato è stata definita con **generic (N : integer),** in modo che poi sia modificabile in caso si volesse cambiare. Lavora in modo sincrono: sul fronte di salita di **clk** il valore **d** viene memorizzato, rendendolo disponibile in uscita **q** al ciclo successivo. In questo modo si implementa il buffer di ritardo necessario per i blocchi del filtro.

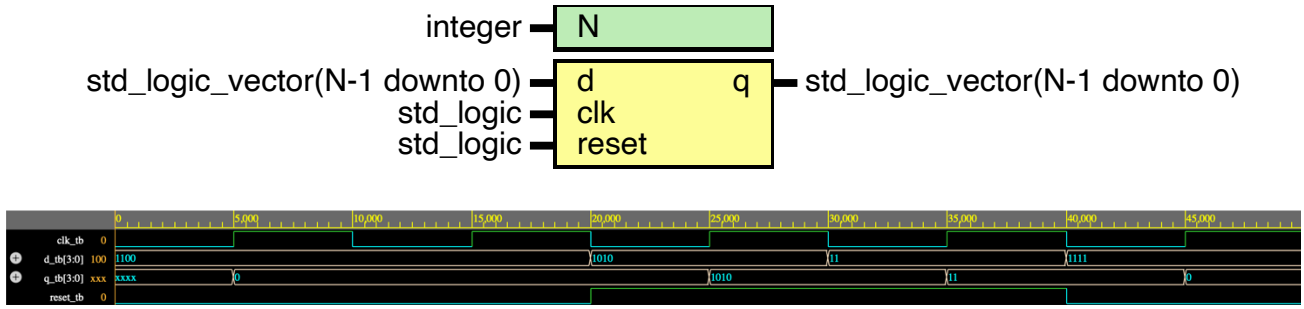

Figura 4. In alto il diagramma dell'interfaccia del flip flop realizzato e in basso il testbench per verificarne il funzionamento.

Nella Figura 4 è visibile il risultato del testbench del Flip Flop. Il segnale **reset**, attivo alto, nei primi due colpi di clock non permette all'ingresso di propagarsi sull'uscita. Quando poi **reset** passa a '1', **d_tb** viene messo in uscita al fronte di salita di clock successivo.

1.3.2 Sommatore generico

Questo blocco permette di eseguire sia la somma sia la differenza come somma in complemento a 2, utilizzate sia nei blocchi integrator sia nei blocchi comb.

In ingresso abbiamo le porte degli addendi e il carry-in, mentre in uscita ci sono il risultato e il carry-out. Opera in modo combinatorio: esegue l'addizione attraverso l'operazione XOR, mentre per il calcolo riporto utilizza le porte AND/OR.

Il risultato del testbench è visibile nella Figura 5. I segnali **a_tb** e **b_tb** sono gli addendi in ingresso al sommatore, mentre **s_tb** la loro somma. Come vediamo nella prima somma tra 0ns e 10ns, la somma tra '10' e '11' con carry_in=0 risulta '101' come ci si aspetta. Un caso particolar è la somma tra '111' e '110', tra 20n e 30ns, perché qui c'è anche da considerare il carry_in. Anche in questo caso la somma risulta corretta (7+5=12) e vediamo che carry_out passa a '1'.

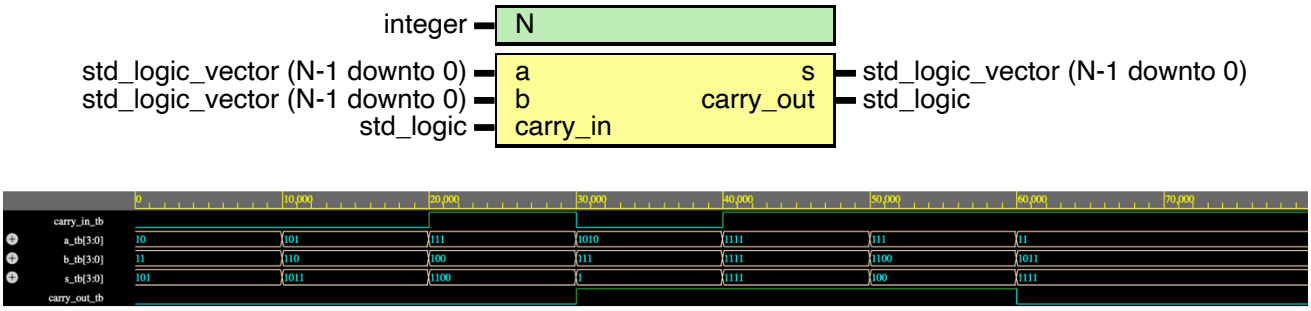

Figura 5. In alto il diagramma dell'interfaccia dello stadio sommatore realizzato e in basso il testbench per verificarne il funzionamento.

1.3.3 Shifter

Questo blocco ha l'obiettivo di aumentare la lunghezza del segnale in ingresso, con lunghezza M, ad una lunghezza N, con N>M.

Visto che i dati che entrano nella catena di filtraggio vanno incontro al fenomeno di bit growth, questo stadio permette di allargare il bus alla dimensione desiderata, senza utilizzare il bit di carry-out ed evitando l'overflow.

Inoltre, questa operazione è necessaria nel blocco comb, che deve eseguire una differenza come somma in complemento a 2. Utilizzando quindi lo shifter, si può preservare il segno copiando il MSB.

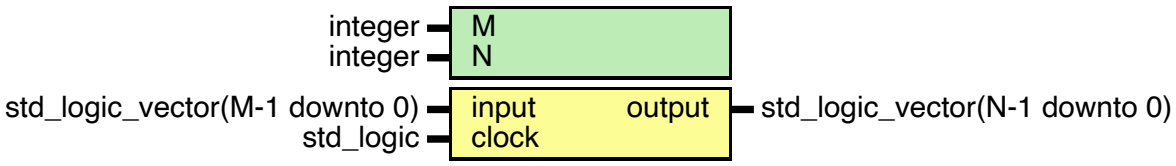

Figura 6. In alto il diagramma dell'interfaccia dello shifter realizzato e in basso il testbench per verificarne il funzionamento.

Visto che stadio è fondamentale per la somma in complemento a 2, lo shift viene fatto mantenendo il segno (definito dal MSB) del numero in codifica signed in ingresso.
Il software utilizzato per i testbench, EDAPlayground, spesso non mostra gli '0' più significativi, come nel caso del primo ingresso, ma si può comunque notare come '0101' sia diventato '**0000**0101', passando da 4 a 8 bit. Il secondo ingresso '1101' conferma il corretto funzionamento, in quanto **output_tb** mostra correttamente '"**1111**1101"'.

1.3.4 Complemento a 2

Il complemento a 2 è un'operazione fondamentale per eseguire la differenza nei blocchi comb.
Realizza il complemento a 2 secondo la formula in cui si invertono tutti i bit dell'ingresso e si somma 1 al risultato (**C2(A) = NOT(A) + 1**).

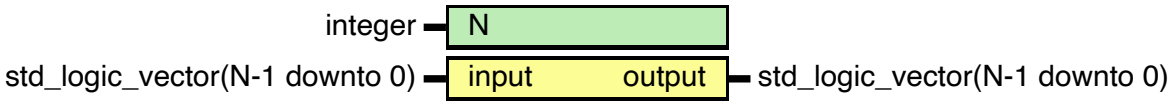

Figura 7. In alto il diagramma dell'interfaccia dello stadio che esegue il complemento a due e in basso il testbench per verificarne il funzionamento.

Anche in questo caso il testbench conferma il funzionamento dello stadio, in particolare:

- '0101_2' (5_{10}) → '1011_2' (-5_{10}) ;
- '1101_2' (-3_{10}) → '1100_2' (3_{10}) ;
- '0000_2' (0_{10}) → '0000_2' (0_{10} con carry);
- '1000_2' (-8_{10}) → '1000_2' (-8_{10}) .

1.3.5 Integratore

Lo stadio è stato realizzato utilizzando due componenti: un flip flop per implementare il ritardo necessario e un sommatore per sommare l'ingresso e il valore di feedback. In particolare, il risultato della somma viene memorizzato nel flip flop sul fronte di salita del clock. Nel ciclo successivo, questo valore viene sommato al nuovo ingresso, come previsto dalla formula X. Una volta compilato il progetto su Quartus, è possibile visualizzare la rappresentazione grafica del codice scritto.

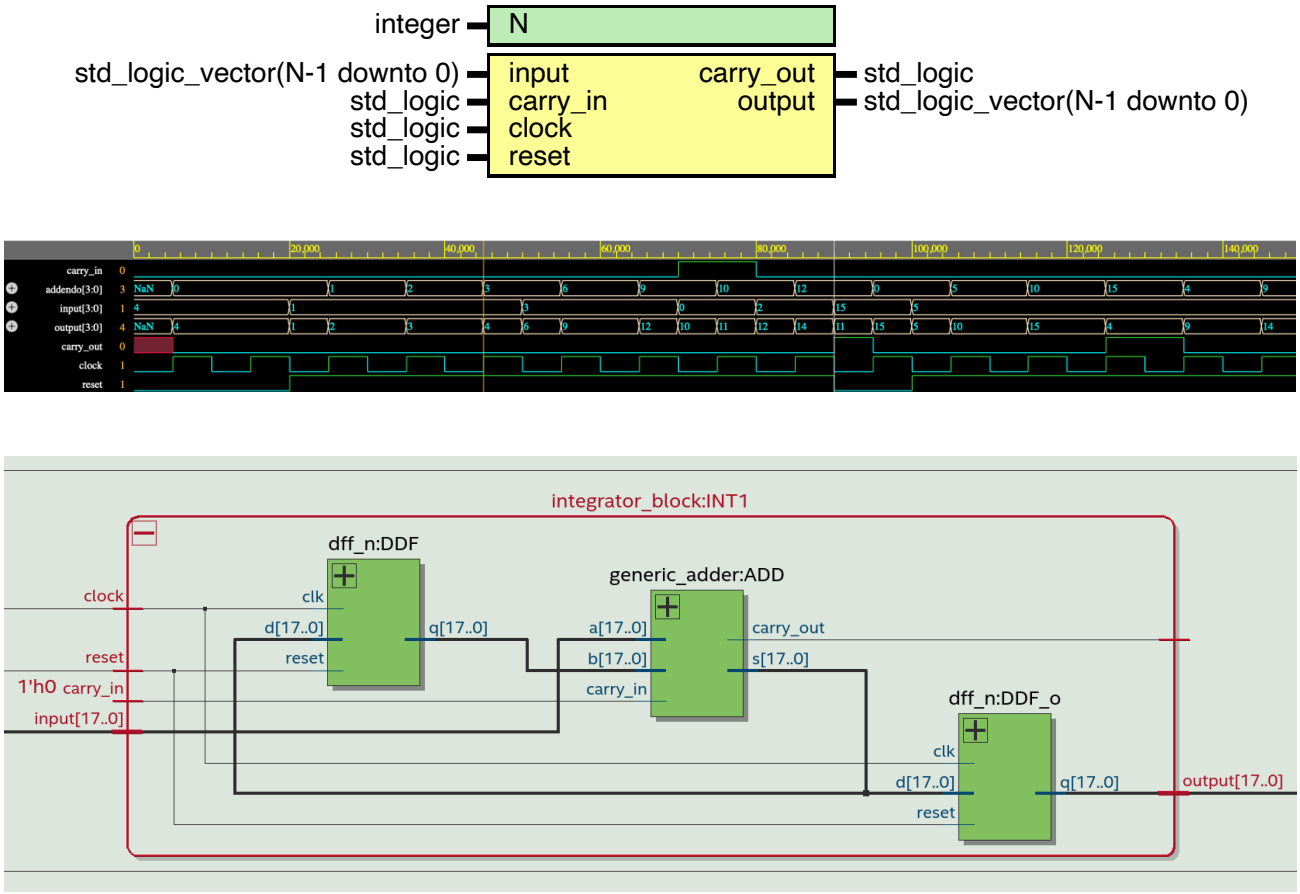

Figura 8. A partire dall'alto il diagramma dell'interfaccia dello stadio integratore realizzato, il testbench per verificarne il funzionamento e lo schema grafico della architettura utilizzata.

1.3.6 Comb

La struttura di questo stadio è simile a quella dell'integratore, con la differenza che la somma viene eseguita in complemento a 2, quindi è necessario istanziare anche questo componente. Quindi i blocchi fondamentali sono il complemento a due, il flip flop e il sommatore. L'ingresso viene convertito in

complemento a due e poi entra nel flip flop. A questo punto, nel ciclo successivo, questo valore negato viene sommato al nuovo ingresso, implementando la differenza.

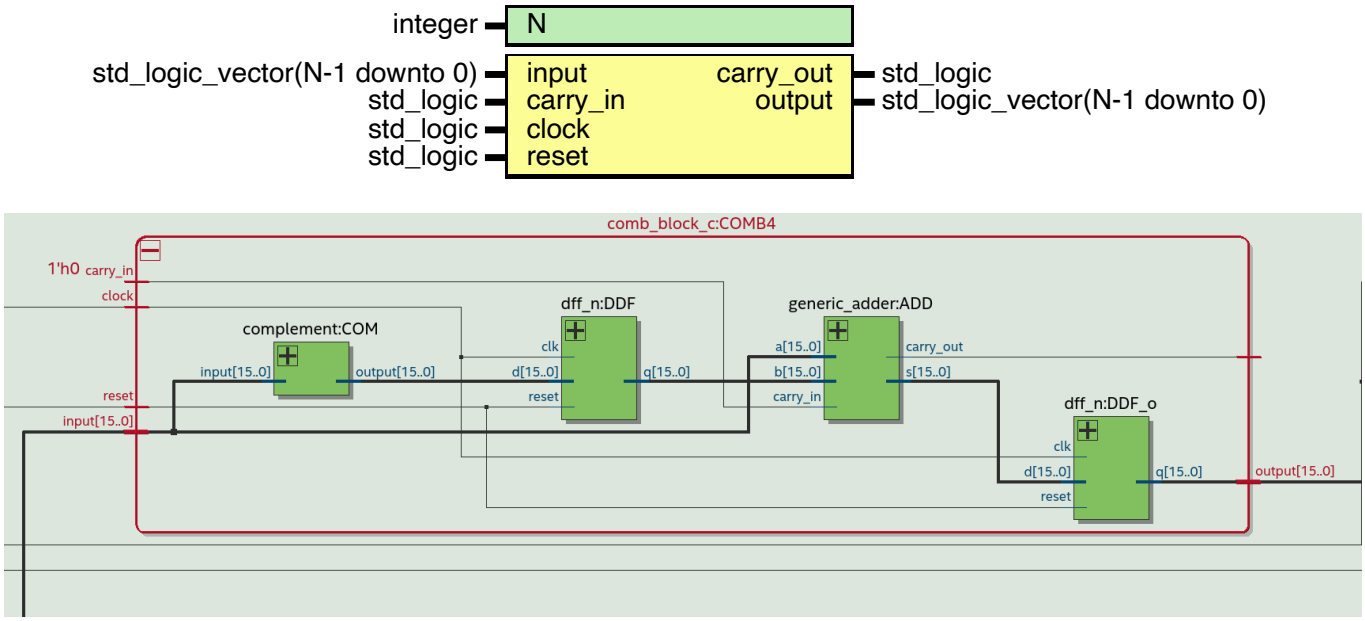

Figura 9. In alto il diagramma dell'interfaccia dello stadio comb realizzato e in basso lo schema grafico della architettura utilizzata.

1.3.7 Inseritore di zeri

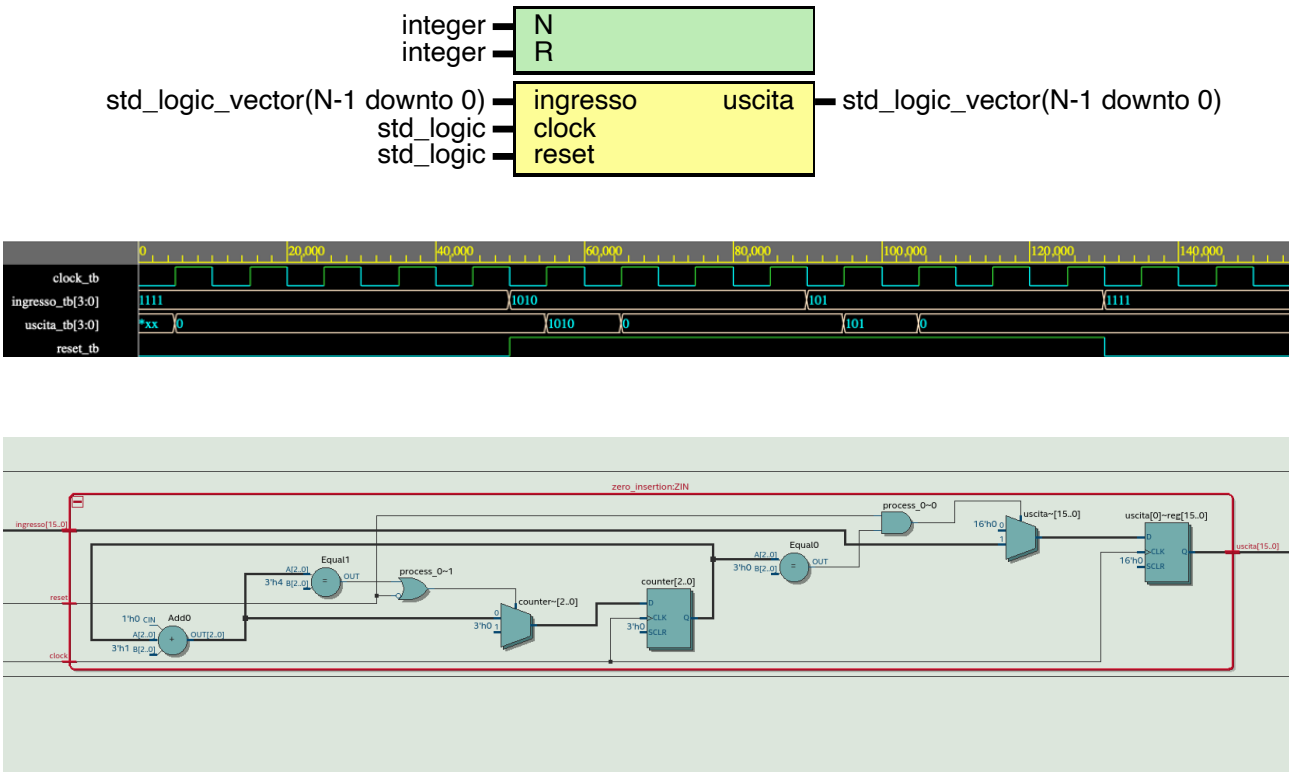

Figura 10. A partire dall'alto il diagramma dell'interfaccia dello stadio inseritore di zeri realizzato, il testbench per verificarne il funzionamento e lo schema grafico della architettura utilizzata.

L'implementazione hardware è gestita da un semplice **contatore** sincronizzato con il clock. Scegliendo di aggiungere 3 zeri, il contatore cicla da 0 a 3. Quando il contatore è a 0, il blocco emette il campione in ingresso, mentre per gli altri valori del contatore (1, 2, e 3), il blocco emette un vettore di zeri, come visibile nel testbench.

1.3.8 Divisore di frequenza

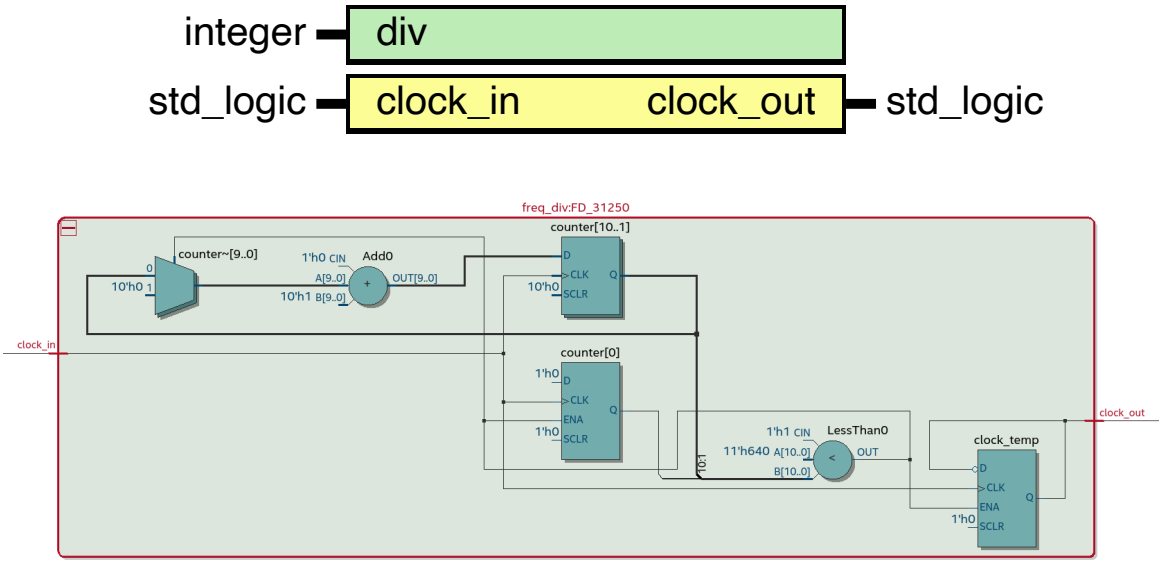

Figura 11. In alto il diagramma dell'interfaccia dello divisore di frequenza realizzato e lo schema grafico della architettura utilizzata.

Il divisore di frequenza è stato realizzato con un **process** sincrono con il fronte di salita del segnale **clock_in**. Quando viene rilevato un fronte di salita, un contatore incrementa fino a raggiungere il valore **div**, che definisce il fattore di divisione della frequenza del segnale in ingresso. Nel momento in cui viene raggiunto questo valore, il segnale **clock_out** in uscita viene commutato e il contatore resettato, per ricominciare il nuovo ciclo.

2 Convertitore Analogico-Digitale

L'ADC incluso nella scheda DE10-NANO è l'integrato LTC2308. La comunicazione con l'FPGA avviene tramite una linea SPI dedicata in cui l'ADC agisce da slave, con i seguenti pin per le interconnessioni sul lato FPGA:

- ADC_CONVST, Output, PIN_U9;
- ADC_SCK, Output, PIN_V10;
- ADC_SDI, Output, PIN_AC4;
- ADC_SDO, Input, PIN_AD4.

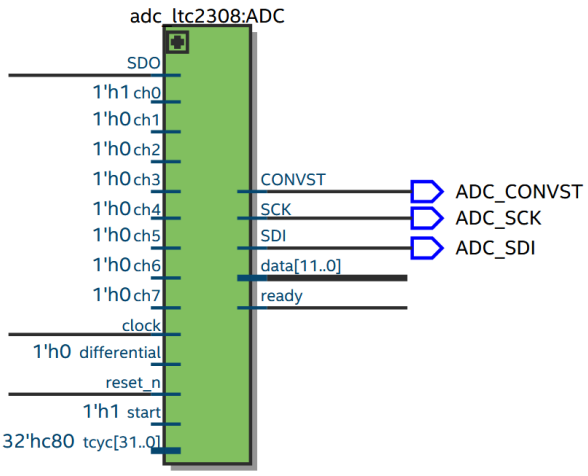

Figura 12. Interfaccia del driver per la gestione dell'ADC.

L'LTC2380 supporta una frequenza di clock di al massimo 40 MHz, decidiamo quindi di generare un clock a 25 MHz dividendo di un fattore 2 il clock di sistema della FPGA.

Il driver per la gestione dell'ADC è un blocco di codice Verilog di terze parti che abbiamo istanziato come component nel top level del progetto. Il component è stato configurato in modo da eseguire la conversione continua del canale 0 in modalità single ended, fissando gli ingressi ***ch0*** e ***start*** a 1 mentre ***differential*** a 0. Il segnale ***Tcy[31..0]*** definisce il tempo di conversione totale dell'ADC come multiplo del tempo di clock, assegnandogli il valore integer unsigned 3200 otteniamo una frequenza di campionamento di 7812.5 sps.

Quando una nuova conversione è disponibile, il driver solleva la linea ***ready*** e mette a disposizione in formato unsigned integer il nuovo dato sul registro d'uscita ***data[11..0]***, che verrà aggiornato al termine della conversione successiva.

3 Implementazione del filtro CIC

Per realizzare il filtro CIC non bisogna fare altro che mettere insieme i blocchi fino ad ora realizzati.

Per quanto riguarda la entity, è stata definita come segue:

```
1
2  ENTITY cic_normal IS
3    generic (N: integer; M: integer; R: integer);
4    port(
5      ingresso  : in std_logic_vector(N-1 downto 0);
6      clock_fast     : in std_logic;
7      reset       : in std_logic;
8      uscita     : out std_logic_vector(N-1 downto 0)
9    );
10 END cic_normal;
```

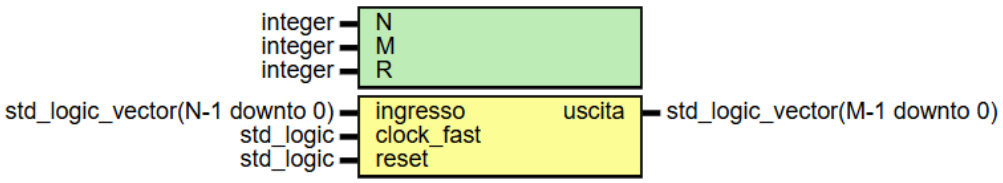

Figura 13. Diagramma dell'interfaccia del filtro CIC realizzato.

Il component istanziato è generic di **N**: lunghezza del dato in ingresso pari a 12; **R**: fattore di interpolazione, pari a 4 o 10 a seconda del caso analizzato; **M**: la dimensione del dato in uscita, pari a 16.

Alle sue porte troviamo:

- **ingresso**: il dato lungo N bit, proveniente dall'ADC;
- **clock_fast**: alla porta di ingresso, è il clock interno della FPGA;
- **reset**: alla porta di ingresso, segnale asincrono che riporta il sistema allo stato iniziale;
- **uscita**: alla porta di uscita, è il dato di dimensione M filtrato e interpolato.

Il segnale ***clock_slow*** viene generato internamente dividendo ***clock_fast*** per il fattore di interpolazione.

Per ottenere la struttura di un filtro CIC interpolatore del quarto ordine è dunque necessario istanziare ed interconnettere in ordine:

- un divisore di frequenza per generare ***clock_slow***;
- Un registro d'ingresso per campionare il segnale alla frequenza di ***clock_slow***;
- Quattro stadi comb di dimensione **N**;
- Uno stadio di zero insertion;

- Uno shifter per estendere la dimensione del dato alla dimensione degli integratori
- Quattro stadi integratori tutti alla dimensione massima calcolata al capitolo 1.2.

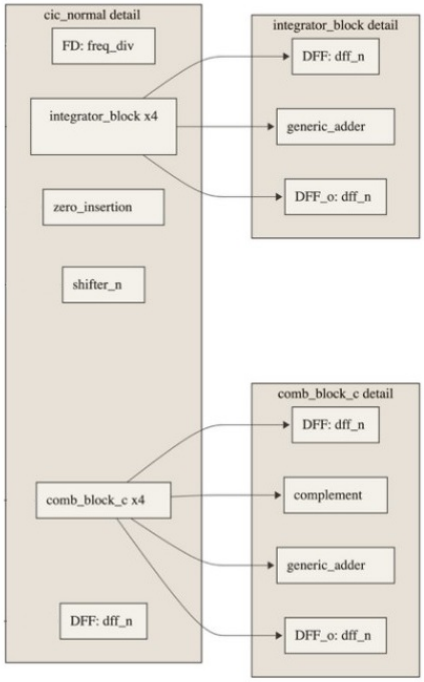

Figura 14. Gerarchia del codice VHDL sel filtro realizzato.

Come già accennato, a causa della crescita di dimensione del dato, dopo l'ultimo blocco integratore è necessario effettuare un troncamento del segnale per poterlo trasmettere via SPI con una sola transazione. Durante l'assegnamento del segnale ***uscita[M-1..0]*** decidiamo di sacrificare i bit meno significativi in eccesso, eseguendo quindi un arrotondamento dell'uscita ai 16 bit MSB.

4 Comunicazione con HPS via SPI

Il protocollo SPI lavora con la logica Master-Slave e in questo caso, l'HPS fa la richiesta (Master) e l'FPGA risponde con i dati (Slave). In genere è possibile che ci siano più slave, quindi è necessario avere un segnale SS (Slave Select), ma nel nostro caso specifico questo non serve, in quanto i dati vengono inviati solo dalla FPGA. Le linee di comunicazione sono:

- **MISO**: Master Input Slave Output;
- **MOSI**: Master Output Slave Input;
- **SCK**: Serial Clock, generato dal master per sincronizzare la trasmissione dei dati.

4.1 Implementazione VHDL dell'interfaccia SPI

Nella Figura 15 è rappresentata la entity dell'interfaccia SPI dove alle porte ci sono:

- **Clk** (input): il segnale di clock del sistema a 50 MHz che sincronizza la logica interna dell'FPGA;
- **Reset** (input): segnale attivo alto.
- **MOSI** (input): la linea attraverso cui l'FPGA riceve i dati inviati dall'HPS, va mappata al pin AG16;
- **SCK** (input): è il segnale di clock generato dal master (HPS). Viene sincronizzato internamente attraverso un registro a scorrimento a tre stadi, per filtrare eventuali metastabilità e rilevare i fronti di salita e discesa. Si assume Va mappato al pin AH12.
- **MISO** (output): La linea attraverso cui l'HPS riceve i dati inviati dall'FPGA, va mappata al pin AH11.

Sono stati inseriti i seguenti parametri generic per rendere il modulo più flessibile:

- **DATA_W**: la dimensione del dato trasmesso.

4.1.1 Descrizione della architecture e process

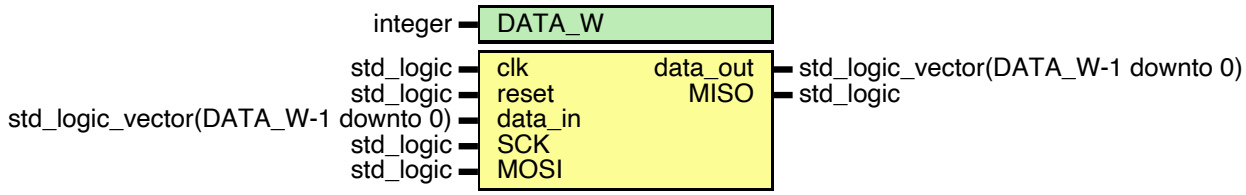

Figura 15. Diagramma dell'interfaccia SPI realizzato.

Nella architecture sono presenti quattro registri

```
signal tx_reg   : std_logic_vector(DATA_W-1 downto 0) := (others => '0');

signal rx_reg   : std_logic_vector(DATA_W-1 downto 0) := (others => '0');

signal sck_sync_reg : std_logic_vector(2 downto 0);

signal sck_rising_edge : std_logic;

signal sck_falling_edge : std_logic;

signal sck_sync : std_logic;
```

di cui

- **tx_reg**: è il registro a scorrimento per la trasmissione;
- **rx_reg**: è il registro a scorrimento per la ricezione;
- **sck_sync_reg**: è il registro per la sincronizzazione e per il rilevamento dei fronti di SCK;
- **sck_rising_edge**: diventa '1' quando viene rilevato un fronte di salita;
- **sck_falling_edge**: diventa '0' quando viene rilevato un fronte di discesa;
- **sck_sync**: è il serial clock sincronizzato, prelevato da sck_sync_reg(1).

La prima cosa da fare è portare il segnale asincrono **SCK** nel dominio del clock di sistema e questo si fa con il registro **sck_sync_reg**, che realizza un sincronizzatore a tre stadi per filtrare eventuali metastabilità e rilevare i fronti di salita-discesa di SCK.
Il sincronizzatore viene realizzato con una descrizione strutturale come segue:

```
sck_sync_reg(2 downto 1) <= sck_sync_reg(1 downto 0);
sck_sync_reg(0) <= SCK;
```

- **sck_sync_reg(0)** è lo stato attuale di SCK. Se avviene una violazione in questo stadio, l'uscita sarà influenzata dalla metastabilità, ma non importa, perché il valore non viene usato nella logica;
- **sck_sync_reg(1)** corrisponde al secondo flip flop della cascata. Si suppone che nel frattempo il primo flip flop abbia avuto tempo di stabilizzare l'uscita metastabile; quindi, questo stadio offre un segnale sicuro, sincronizzato con il clock interno;
- **sck_sync_reg(2)** è il terzo stadio della catena, introdotto per il riconoscimento del fronte di clock.

I segnali **sck_rising_edge** e **sck_falling_edge** vengono assegnati tramite logica combinatoria secondo la seguente tabella della verità:

	IN		OUT	
SCK	**sck_sync_reg(2)**	**sck_sync_reg(1)**	**sck_rising_edge**	**sck_falling_edge**
0	**0**	**0**	**0**	**0**
1	**1**	**1**	**0**	**0**
0→1	**0**	**1**	**1**	**0**
1→0	**1**	**0**	**0**	**1**

Il processo principale è **main_proc: process(clk),** quindi tutto avviene in modo sincrono con il clock di sistema. Nel process sono state definite le due seguenti variabili

```
variable count : integer range 0 to DATA_W := 0;
variable last_sck : integer range 0 to sck_timeout+1 := 0;
```

dove **count** serve per contare i fronti di **SCK** e far scorrere i registri, mentre **last_sck** gestisce il timeout del trasferimento, ossia tiene conto di quanti colpi di clock di sistema (50 MHz) sono passati dall'ultimo fronte di SCK. La costante **sck_timeout=40** è il limite al valore di **last_sck** e corrisponde ad un timeout pari a 0.8 periodi di SCK a 1 MHz, trascorso questo lasso di tempo senza un nuovo fronte di SCK la transazione corrente viene abortita.

La transazione SPI viene implementata in **MODE 1**, MSB first, SCK a 1MHz in idle al livello basso.

In MODE 1 i dati vengono caricati sulla linea MISO al fronte di salita e campionati dal MOSI al fronte di discesa.

Quando viene rilevato un fronte di salita e il contatore è 0, quindi è il primo SCK della trasmissione, il registro di trasmissione **tx_reg** viene aggiornato con il dato in ingresso alla entity. Altrimenti **tx_reg**

viene shiftato a sinistra di un bit e viene inserito 0 nel LSB. Il MSB di **tx_reg** è assegnato strutturalmente alla linea MISO. Altrimenti viene inviato sulla linea MISO.

```
if sck_rising_edge then
   last_sck := 0;
   if (count=0) then
      tx_reg <= data_in;
   else
      tx_reg(DATA_W-1 downto 1) <= tx_reg(DATA_W-2 downto 0);
      tx_reg(0)<='0';
   end if;
end if;
```

Se viene rilevato un fronte di discesa, avviene lo scorrimento a sinistra del registro **rx_reg** e il valore sulla linea MOSI viene inserito nel LSB, permettendo di assemblare ad ogni colpo di SCK la parola da ricevere. A questo punto viene incrementato il contatore per tenere traccia dell'avanzamento della trasmissione.

```
if sck_falling_edge then
        last_sck := 0; -- resetta il timeout
        rx_reg(DATA_W-1 downto 1) <= rx_reg(DATA_W-2 downto 0);
        rx_reg(0)<=MOSI;
        count := count+1;
end if;
```

Alla fine della trasmissione, ovvero quando il contatore raggiunge DATA_W, viene aggiornato il registro **data_out** con la word ricevuta e si puliscono **tx_reg** e **rx_reg**.

```
if count >= DATA_W then
  last_sck := 0;
  data_out <= rx_reg;
  rx_reg <= (others =>'0');
  tx_reg <= (others =>'0'
  count := 0;
end if;
```

Di seguito è riportata anche la gestione del timeout, in cui si puliscono i registri e ci si prepara ad una nuova transazione.

```
if last_sck >= sck_timeout then
  last_sck := 0;
  count := 0;
  rx_reg <= (others =>'0');
  tx_reg <= (others =>'0');
end if;
```

4.1.2 Verifica della trasmissione da parte del Slave

Una volta implementato il driver SPI nella FPGA procediamo a verificarne il corretto funzionamento.
A tale scopo, per testare singolarmente questo blocco con dei dati di prova, è stato realizzato in VHDL un generatore d'onda triangolare utilizzando un semplice contatore UP/DOWN a 16 bit, il clock è stato calcolato in modo da avere un periodo dell'onda di 90 secondi.
Per semplificare il debug è stato deciso di utilizzare come master un ESP32 nel framework di Arduino, in qualità di piattaforma con cui abbiamo più familiarità e abbondanza di librerie collaudate.
Il codice C scritto per il master utilizza la libreria ***SPI.h*** con la seguente configurazione:

```
SPISettings settingsFPGA(1000000, MSBFIRST, SPI_MODE1);
```

Il dato viene quindi letto attraverso la funzione ***SPI.transfer16(uint16_t)*** e trasferito al PC tramite la porta seriale, di cui grafichiamo i dati attraverso un plotter seriale.

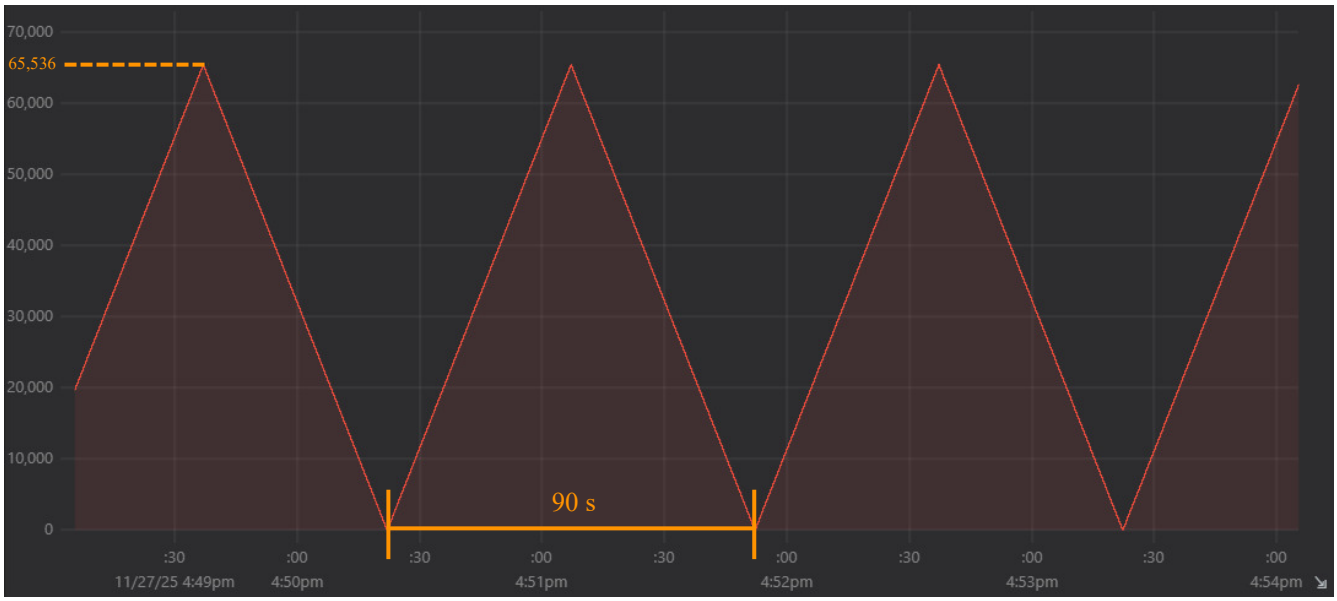

Figura 16. Onda triangolare di test inviata via SPI.

5 Software HPS

Nello stesso IC della Cyclon V utilizzata è integrato anche un HPS (Hard Processor System) basato su ARM. Sulla scheda SD già presente nella board è installata un'apposita distribuzione di Linux che si avvia in automatico all'alimentazione del sistema. Il processore ARM usa il protocollo SPI descritto in precedenza per comunicare con la FPGA a mezzo di dei ponticelli tra il connettore LTC, dove sono esposti i GPIO dell'HPS, e i pin della FPGA. Il protocollo SPI sul lato dell'HPS viene gestito dalla libreria ***SPI_Hw.h*** che, dopo una fase di inizializzazione, permette il trasferimento dei dati tramite la chiamata della funzione ***SPIM_WriteReadTxRxData16(uint16_t).***

5.1.1 Edge computing sull'HPS

Nell'ottica di realizzare un sistema che avesse anche funzionalità di edge computing, il codice sull'HPS elabora il segnale ricevuto dall'interfaccia SPI in modo tale da presentare al cloud un'informazione sintetica di quello che sta accadendo sulla FPGA. Nel nostro caso abbiamo scelto di calcolare il valore RMS del dato ricevuto su finestre di 1 secondo. Questo ci permette nella fase di validazione di misurare in modo semplice la risposta in frequenza del filtro.

L'algoritmo implementato nel ***main()*** è dunque il seguente:

```
float RMS = 0;
const float fattore_scala=65535/2; // fondo scala CIC 16 bit signed per ingresso adc
5000 mV
const float K=5/fattore_scala; // conversione in Volt
const uint32_t num_campioni = 31250; // circa 1 sec di acquisizione
float campione=0;
for (uint32_t i = 0; i < num_campioni; i++) {
      campione = K*(float)(int16_t)SPIM_WriteReadTxRxData16(0x0001);
      float square = campione*campione;
      RMS=RMS + square;
}
RMS = sqrt(RMS/num_campioni);

printf("\n>RMS:");
printf("%f\n",RMS);
```

5.1.2 Comunicazione con ThingSpeak

Per la comunicazione tramite internet con la API di ThingSpeak è stata utilizzata la libreria **libcurl**.

Al termine del calcolo del loop di acquisizione e del calcolo del valore RMS, si procede all'invio al canale ThingSpeak, selezionandolo tramite una chiave ***KEY_API*** fornita dalla piattaforma in seguito alla creazione del canale.

```
// formattiamo l'URL con la API_KEY ed il dato elaborato
sprintf(buffer, THINGSPEAK_URL, THINGSPEAK_API_KEY, RMS);

// con il comando curl_easy_setopt impostiamo le opzioni della sessione curl
dl_curl_easy_setopt(curl, CURLOPT_URL, buffer);

// Eseguimo la richiesta
res = dl_curl_easy_perform(curl);

// CURLE_OK indica che la richiesta è andata a buon fine
if(res != CURLE_OK) {
fprintf(stderr, "[NET ERROR] Invio fallito: %s\n", dl_curl_easy_strerror(res));
} else {
printf(" -> [NET OK] Inviato a ThingSpeak.\n");
}
```

Una volta compilato il codice sorgente e avviato l'eseguibile, ci colleghiamo all'interfaccia ThingSpeak con un browser e per visualizzare il dato proveniente dal canale desiderato.

L'interfaccia è stata configurata per mostrare a sinistra il numero attuale letto e inviato all'HPS, mentre a destra è visibile la forma d'onda nel tempo.

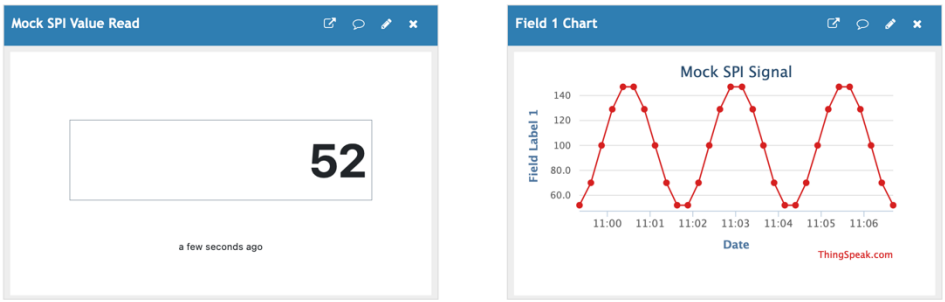

Figura 17. Schermata di ThingSpeak, a sinistra l'ultimo valore inviato, a destra l'andamento nel tempo dei valori.

5.2 Configurazione della rete e cross-compilazione

5.2.1 Configurazione di rete

All'accensione della scheda e in seguito alla connessione del cavo Ethernet è necessario configurare la rete per fare in modo che l'HPS si possa connettere ad internet e comunicare con l'API di ThingSpeak. Per procedere con la configurazione del kernel Linux sull'HPS colleghiamo la scheda tramite USB all'interfaccia USB-UART della scheda e usiamo MobaXterm per aprire una connessione con la porta COM corrispondente e accedere alla shell. Dopo aver effettuato il login come root, è stato eseguito il comando **ifconfig**, che verifica lo stato delle interfacce di rete (Figura 18). Il comando ha mostrato le interfacce disponibili, quella a noi necessaria è eth0, che è la porta Ethernet fisica a cui abbiamo collegato il cavo. L'interfaccia aveva già assegnato staticamente l'indirizzo 192.168.137.252, ma è necessario riconfigurarla.

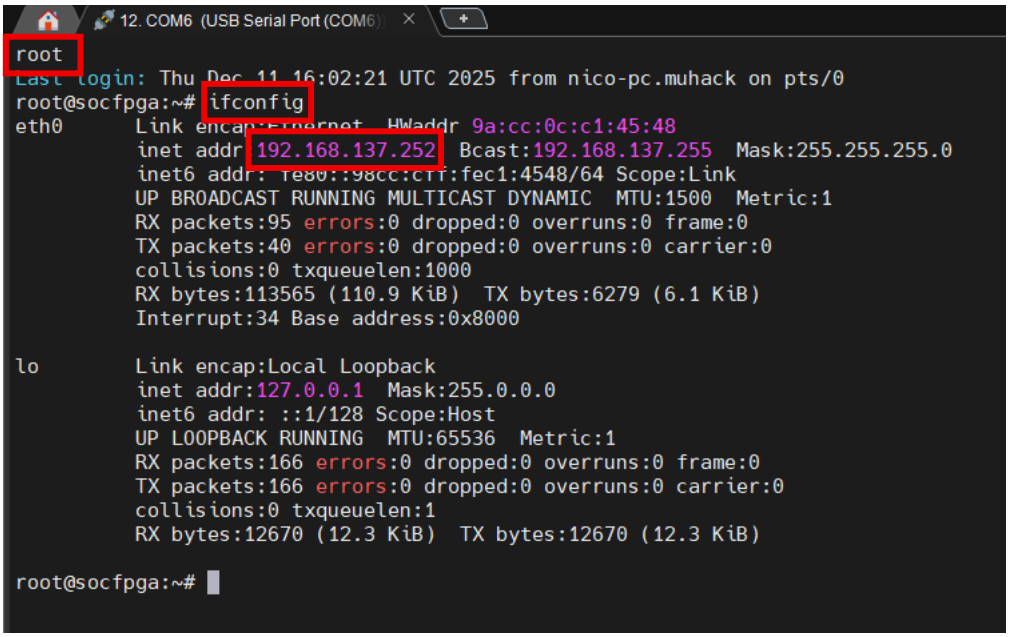

Figura 18. Procedura di connessione alla rete: comando ifconfig.

Per procedere alla configurazione automatica lanciamo il comando **udhcpc eth0** (Figura 20), che assegna all'HPS un nuovo indirizzo attraverso il server DHCP. A questo indirizzo sarà poi possibile aprire una connessione SSH con l'HPS in modo da poter inviare i file del programma.

```
root@socfpga:~# udhcpc eht0
udhcpc (v1.22.1) started
run-parts: /etc/udhcpc.d/00avahi-autoipd exited with code 1
Sending discover...
Sending discover...
Sending select for 192.168.0.177...
Lease of 192.168.0.177 obtained, lease time 43200
run-parts: /etc/udhcpc.d/00avahi-autoipd exited with code 1
/etc/udhcpc.d/50default: Adding DNS 192.168.100.1
root@socfpga:~#
```

Figura 20. Procedura di connessione alla rete: comando udhcpc eth0.

Per verificare il funzionamento della connessione ad internet eseguiamo un ping al DNS di Google tramite il comando **ping 8.8.8.8**. In questo modo verifichiamo che il gateway e il routing verso internet funzionino correttamente (Figura 19).

```
root@socfpga:~# ping 8.8.8.8
PING 8.8.8.8 (8.8.8.8): 56 data bytes
64 bytes from 8.8.8.8: seq=16 ttl=116 time=3.868 ms
64 bytes from 8.8.8.8: seq=17 ttl=116 time=4.024 ms
64 bytes from 8.8.8.8: seq=18 ttl=116 time=3.682 ms
```

Figura 19. Procedura di connessione alla rete: comando ping 8.8.8.8.

Ora che è stata configurata la rete è possibile usare l'indirizzo IP precedentemente ottenuto per instaurare una connessione SSH, così da poter sfruttare la funzionalità di drag-and-drop offerto da MobaXterm grazie al SFTP (SSH File Transfer Protocol). In questo modo si può fare un semplice Drag-and-Drop dell'eseguibile cross-compilato dal PC direttamente nel file system di Linux su FPGA (Figura 21).

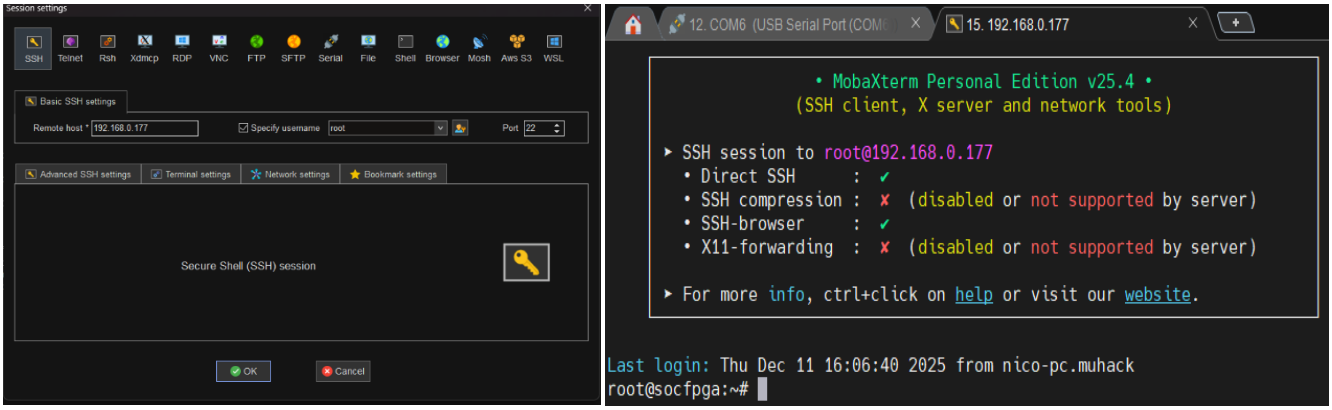

Figura 21. Avvio della sessione SSH con l'indirizzo fornito.

5.2.2 Cross-compilazione e Makefile

Una volta scritto il programma C che legge i dati dall'SPI, li elabora e li invia a ThingSpeak, è necessario compilarlo per l'architettura di destinazione, in questo caso l'HPS Arm Linux. Visto che il PC utilizzato per la compilazione ha un'architettura diversa da quella target, il processo prende il nome di cross-compilazione.

Per scrivere il Makefile è necessario specificare il compilatore da utilizzare e indicare quali librerie includere staticamente nell'eseguibile (nel nostro caso SPI_Hw e dl_curl) e quali linkare dinamicamente: ***librt***: libreria realtime (-lrt), ***libm***: libreria matematica (-lm), ***libpthread***: libreria per la gestione dei threads (-lpthread), ***libdl***: libreria per il caricamento dinamico (-ldl).

A questo punto è possibile trasferire l'intera cartella dei file all'HPS, come descritto nel capitolo precedente, e lanciare l'eseguibile ***SPI_Things***.

```
#
TARGET = SPI_Things
ALT_DEVICE_FAMILY ?= soc_cv_av
SOCEDS_ROOT ?= $(SOCEDS_DEST_ROOT)
HWLIBS_ROOT = $(SOCEDS_ROOT)/ip/altera/hps/altera_hps/hwlib
#CROSS_COMPILE = arm-linux-gnueabihf-
CFLAGS = -static -g -Wall   -D$(ALT_DEVICE_FAMILY) -I$(HWLIBS_ROOT)/include/$(ALT_DEVICE_FAMILY)   -
I$(HWLIBS_ROOT)/include/ -Iinc
LDFLAGS =  -g -Wall -lrt -lm -lpthread -ldl
CC = $(CROSS_COMPILE)gcc
LD = $(CROSS_COMPILE)gcc
ARCH= arm

.PHONY: build
build: $(TARGET)

$(TARGET): main.o SPI_Hw.o dl_curl.o
	$(LD) $(LDFLAGS)   $^ -o $@

%.o : %.c
	$(CC) $(CFLAGS) -c $< -o $@

.PHONY: clean
clean:
	rm -f $(TARGET) *.a *.o *~

.PHONY: all
all: build
```

6 Conclusioni

La gerarchia del filtro CIC realizzato è visibile nella Figura 22, dove si possono vedere a partire da ADC_CIC_top, in giallo, i componenti istanziati. È stato scelto di implementare un top level in cui sono contenuti l'interfaccia SPI, il driver per il convertitore analogico digitale, il blocco per la generazione dei segnali di clock necessari e una catena di DSP dove avviene il vero e proprio filtraggio. Nella catena DSP è stato aggiunto anche un registro per la memorizzazione di un valore di offset, aggiornato con l'uscita del filtro alla pressione del pulsante KEY(1), e un sottrattore per rimuovere l'offset dal valore filtrato prima di trasferirlo al blocco SPI. Il filtro CIC implementato è contenuto in cic_normal.vhd, dove sono stati istanziati anche i blocchi elementari già descritti e riportati nel dettaglio della figura, dove sono evidenziate anche le dipendenze dei blocchi comb e integrator.

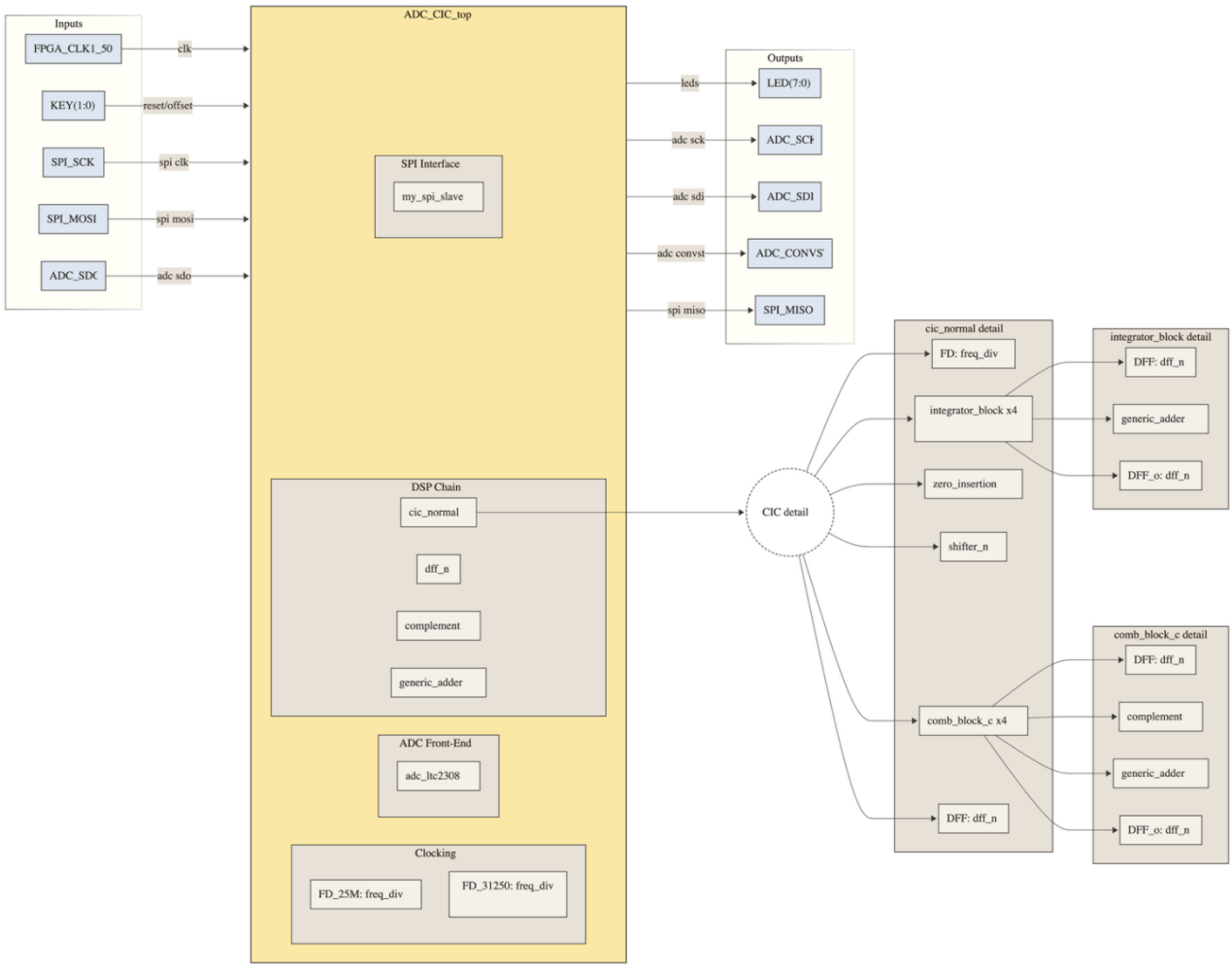

Figura 22. Gerarchia del codice VHDL del sistema complessivo.

Di seguito viene riportata una panoramica delle risorse della FPGA utilizzate dal progetto, ottenuta dai report post fitting di Quartus.

Quartus Prime Version	20.1.0 Build 711 06/05/2020 SJ Lite Edition
Revision Name	ADC_CIC_SPI
Top-level Entity Name	ADC_CIC_top
Family	Cyclone V
Device	5CSEBA6U23I7
Timing Models	Final
Logic utilization (in ALMs)	267 / 41,910 (< 1 %)
Total registers	383
Total pins	18 / 314 (6 %)
Total virtual pins	0
Total block memory bits	0 / 5,662,720 (0 %)
Total RAM Blocks	0 / 553 (0 %)
Total DSP Blocks	0 / 112 (0 %)
Total HSSI RX PCSs	0
Total HSSI PMA RX Deserializers	0
Total HSSI TX PCSs	0
Total HSSI PMA TX Serializers	0
Total PLLs	0 / 6 (0 %)
Total DLLs	0 / 4 (0 %)

	Compilation Hierarchy Node	ALMs needed [=A-B+C]	[A] ALMs used in final placement	Combinational ALUTs	Dedicated Logic Registers	Pins
1	|ADC_CIC_top	267.0 (0.7)	298.0 (1.3)	441 (2)	383 (1)	18
1	|freq_div:FD_31250|	7.5 (7.5)	7.5 (7.5)	13 (13)	14 (14)	0
2	|dff_n:DDF|	3.8 (3.8)	4.2 (4.2)	0 (0)	15 (15)	0
3	|my_spi_slave:SPI_slave|	22.8 (22.8)	23.3 (23.3)	28 (28)	30 (30)	0
4	|complement:COM|	4.2 (4.2)	4.8 (4.8)	16 (16)	0 (0)	0
5	|freq_div:FD_25M|	0.3 (0.3)	1.2 (1.2)	2 (2)	2 (2)	0
6	|generic_adder:ADD|	24.8 (24.8)	26.2 (26.2)	42 (42)	0 (0)	0
7	|adc_ltc2308:ADC|	45.2 (45.2)	46.8 (46.8)	84 (84)	55 (55)	0
8	|cic_normal:CIC|	157.5 (0.0)	182.7 (0.0)	254 (0)	266 (0)	0
1	|comb_block_c:COMB1|	1.5 (0.0)	1.8 (0.0)	3 (0)	6 (0)	0
2	|dff_n:DDF|	0.0 (0.0)	0.3 (0.3)	0 (0)	1 (1)	0
3	|freq_div:FD|	1.3 (1.3)	1.9 (1.9)	3 (3)	3 (3)	0
4	|comb_block_c:COMB4|	25.2 (0.0)	25.8 (0.0)	44 (0)	33 (0)	0
5	|shifter_n:SH2|	4.8 (4.8)	6.3 (6.3)	0 (0)	16 (16)	0
6	|comb_block_c:COMB2|	15.7 (0.0)	17.6 (0.0)	25 (0)	39 (0)	0
7	|comb_block_c:COMB3|	24.5 (0.0)	26.4 (0.0)	44 (0)	41 (0)	0
8	|integrator_block:INT2|	20.5 (0.0)	22.7 (0.0)	33 (0)	26 (0)	0
9	|integrator_block:INT3|	21.1 (0.0)	23.8 (0.0)	33 (0)	27 (0)	0
10	|integrator_block:INT1|	19.4 (0.0)	22.4 (0.0)	32 (0)	23 (0)	0
11	|zero_insertion:ZIN|	4.2 (4.2)	8.3 (8.3)	4 (4)	19 (19)	0
12	|integrator_block:INT4|	19.5 (0.0)	25.2 (0.0)	33 (0)	32 (0)	0

Figura 24. Report post fitting. A sinistra il riassunto delle risorse utilizzate, a destra il dettaglio per ogni elemento del progetto.

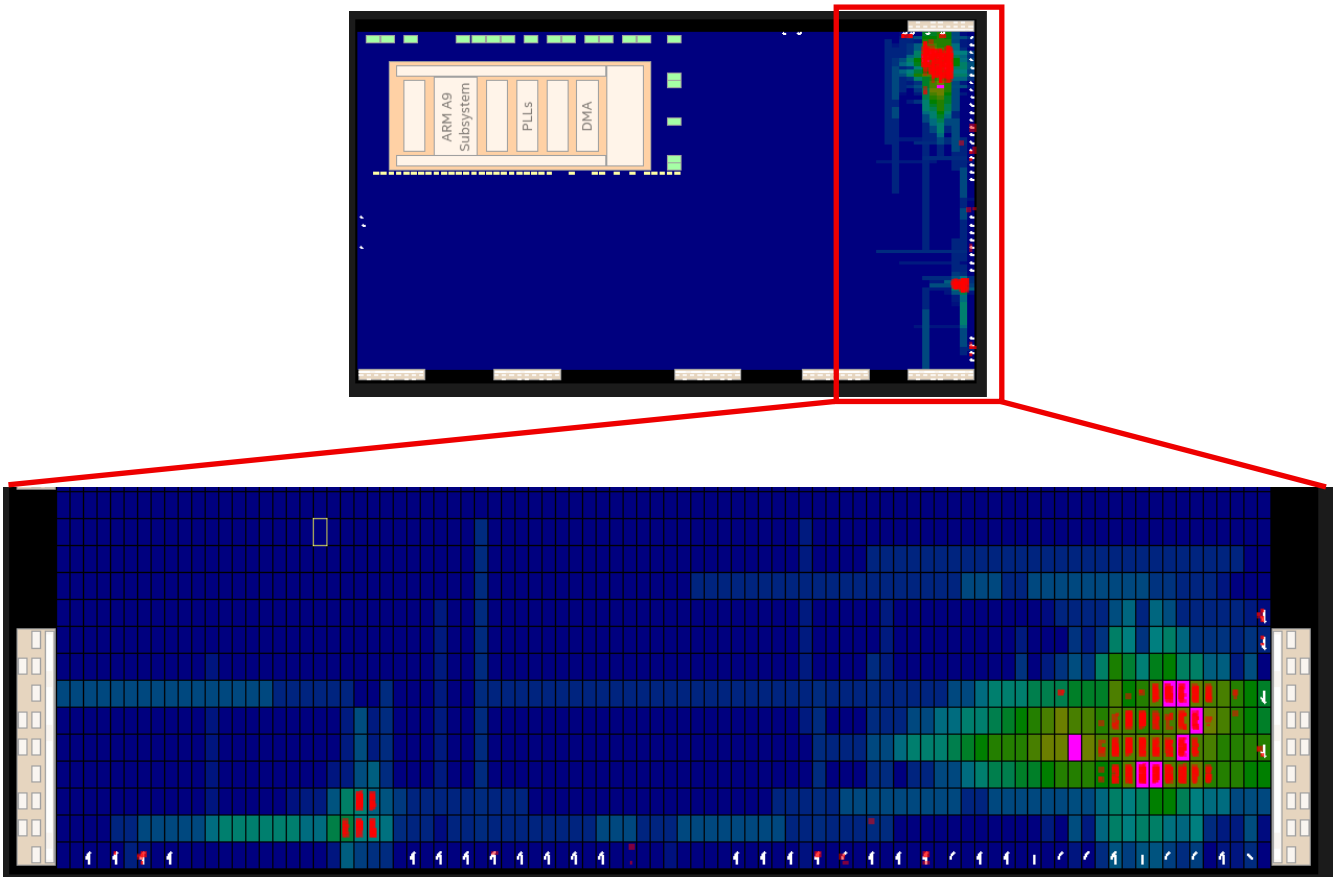

Figura 23. Vista del chip planner di Quartus, il gradiente di colore rappresenta la densità delle interconnessioni, in rosso è evidenziata l'allocazione dei design elements.

Per verificare il corretto funzionamento del filtro, è stato deciso di forzare in ingresso un gradino alla pressione del pulsante e visualizzare l'uscita del filtro. Ci si aspetta che il gradino venga filtrato e che quindi le componenti spettrali oltre la frequenza di taglio del filtro vengano attenuate, ottenendo quindi un gradino smussato. In Figura 25 è visibile il confronto tra le uscite ottenute in simulazione e in SignalTap, mettendo in ingresso al filtro lo stesso gradino. La corrispondenza tra le forme d'onda in uscita conferma il funzionamento atteso del filtro.

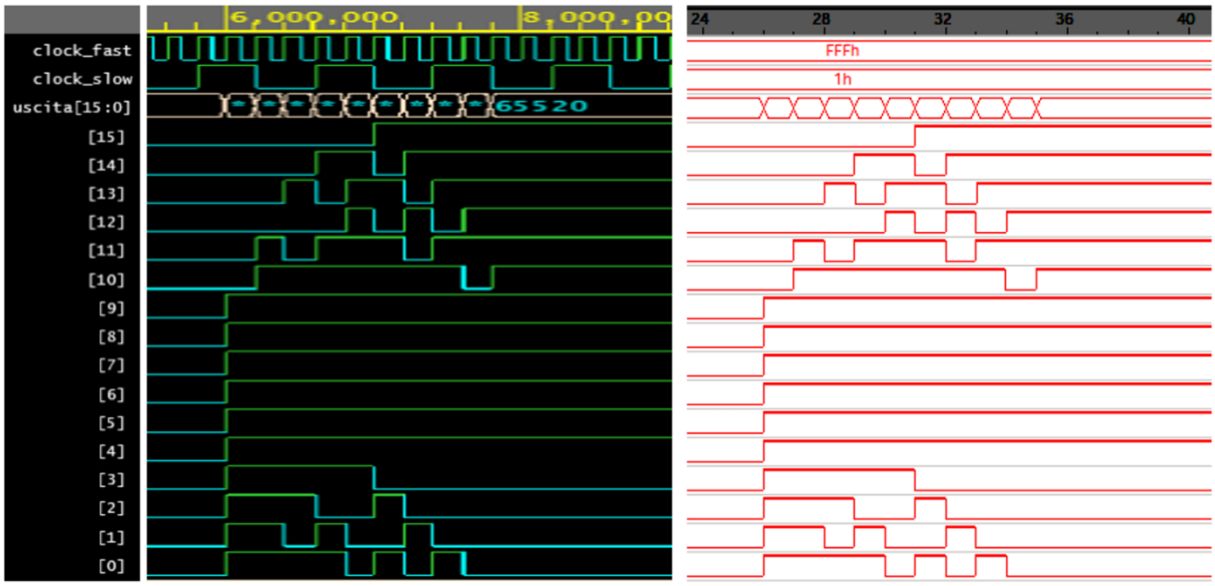

Figura 25.Confronto tra le risposte al gradino ottenute in simulazione (sinistra) e su SignalTap (destra).

Dal momento che la versione gratuita di ThingSpeak permette di inviare un campione ogni 15 secondi, il programma sull'HPS stampa a terminale una volta al secondo il valore RMS e ogni 15 secondi invia a ThingSpeak l'ultimo valore calcolato.

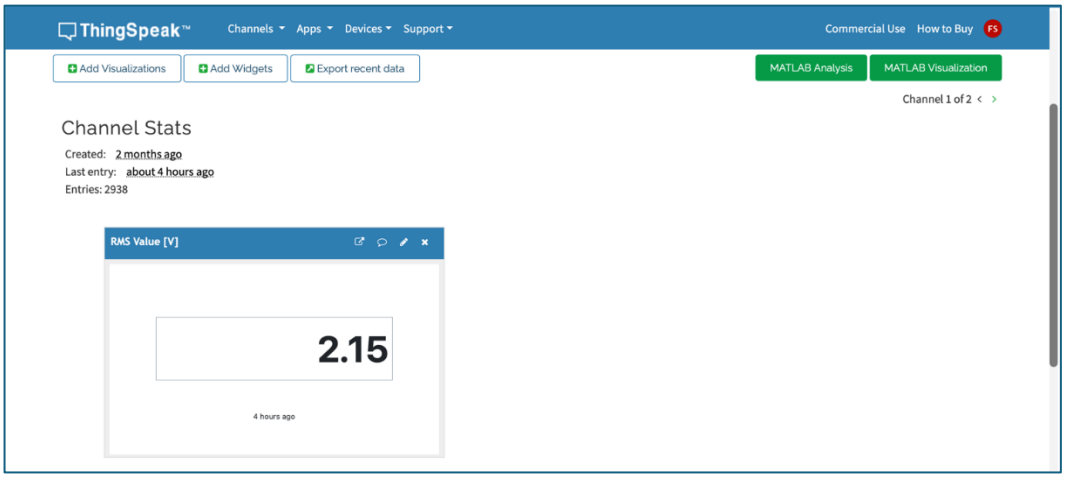

Figura 26. Schermata di visualizzazione di ThingSpeak, dove è visibile il valore RMS del segnale ricevuto in ingresso all'ADC.

Per confrontare il funzionamento teorico del filtro con quello reale, è stata misurata la sua risposta in frequenza:

È stato messo in ingresso all'ADC un segnale sinusoidale con valore RMS-AC uguale a 0.707V e un offset DC di 2.5V per posizionarlo a metà del campo di misura del convertitore. L'ADC è stato impostato con una frequenza di campionamento pari a 7812 campioni al secondo quindi la frequenza del segnale in ingresso è stata variata fino poco oltre la frequenza di Nyquist di questo. Il filtro è stato studiato per due fattori di interpolazione, R=4 e R=10 e le risposte sono visibili nella Figura 27, confermando l'andamento teorico della risposta in frequenza, fino a che non si verifica aliasing approcciando la frequenza di Nyquist.

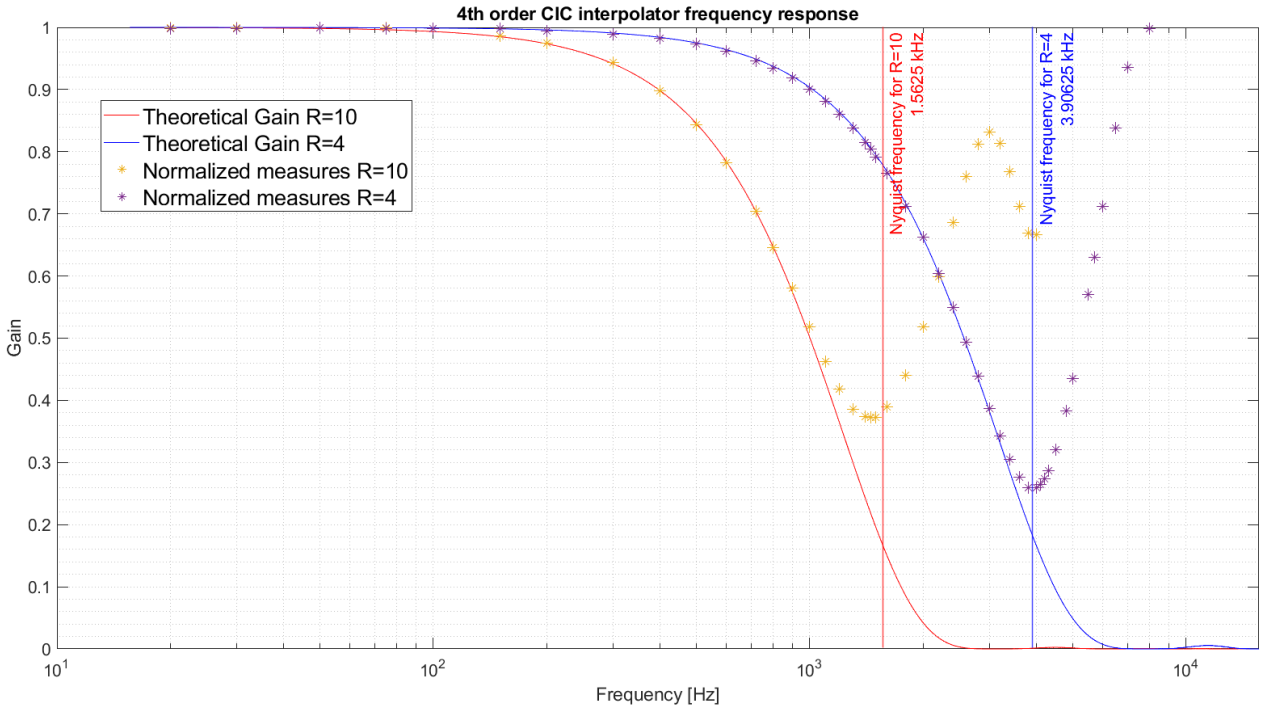

Figura 27. Guadagno del filtro CIC realizzato, per fattore di interpolazione R=4 (blu) e R=10 (rosso).