Università della Calabria

Dipartimento di ingegneria informatica, modellistica, elettronica e sistemistica

Corso di studio in

Ingegneria Elettronica

“FPGA implementation of spatial filtering circuit for 2D grayscale images”

Studenti

Giuseppe Pirilli 237909
Giuseppe Vizza 235491

Anno Accademico 2023/2024

Indice

Introduzione

Questo elaborato si concentra sulla progettazione, implementazione e analisi prestazionale di un filtro spaziale in tecnologia FPGA, da impiegare per la realizzazione di un sistema di elaborazione di immagini bidimensionali con dimensioni 64×64 in scala di grigi a 8 bit (valori compresi tra 0 e 255). L'FPGA scelto per l'implementazione è lo $Zynq\ XC7Z020 - 1CLG400C$ di cui è dotata la scheda di sviluppo $Pynq - Z2$.

La ragione principale nella scelta della tecnologia FPGA è l'impiego congiunto tra hardware e software, ovvero, consente di abbinare le operazioni a basso livello, eseguite da un circuito custom implementato in logica programmabile (parte hardware) mediante la programmazione VHDL (tool Vivado), con le operazioni ad alto livello scritte in C/C++ ed eseguite da un processore (parte software) , in modo da massimizzare l'efficienza e le prestazioni e, garantendo, allo stesso tempo, una grande flessibilità, dando al progettista la possibilità di variare l'implementazione del circuito custom per soddisfare le esatte richieste dell'applicazione. Ciò rappresenta una piattaforma ideale per l'implementazione integrata dell'intero sistema, offrendo un rapporto costo/prestazioni tale da renderla un'ottima alternativa alla tecnologia ASIC.

Il filtraggio spaziale, riconducibile ad un'operazione di convoluzione, più in particolare ad un'operazione MAC (Multiply Accumulate), rappresenta una delle operazioni di basso livello più frequentemente utilizzate nell'ambito dell'elaborazione delle immagini, in cui il generico pixel di output viene determinato applicando una disposizione spaziale di elementi (coefficienti del filtro/kernel) su un intorno del pixel di input i cui elementi occupano la medesima posizione. La selezione di coefficienti adeguati consente di utilizzare lo stesso circuito logico per molte delle tipiche applicazioni di filtraggio spaziale su un'immagine, come la rimozione del rumore, la nitidezza dell'immagine, la sfocatura/levigatura e l'estrazione di dettagli. Per questo elaborato vengono impiegati filtri isotropici 5×5 di tipo Laplaciano e Laplaciano-Gaussiano, la cui funzione è quella di evidenziare i bordi in un'immagine. Ciò è importante, qualora si abbia la necessità di identificare degli oggetti o caratterizzare dei dettagli in una scena osservata.

Le prestazioni vengono analizzate in termini di latenza, frequenza di clock, utilizzo delle risorse e dissipazione di potenza. Inoltre, per testarne la precisione/robustezza, i risultati ottenuti dal circuito di filtraggio progettato a basso livello in VHDL vengono confrontati, attraverso delle opportune metriche, con un'implementazione dell'algoritmo di filtraggio ad alto livello tramite l'utilizzo del tool MATLAB.

1 Sistema di acquisizione e image processing

Un'immagine non è altro che una rappresentazione spaziale di una scena, ricavata in seguito ad un processo di acquisizione mediante l'impiego di opportuni sensori di luce disposti linearmente o in forma matriciale (vedi **fig.1.1**). Il singolo sensore è un trasduttore che assorbe l'intensità della luce riflessa da una porzione della scena osservata e la converte in un valore di tensione analogico proporzionale il quale dovrà essere convertito, a sua volta, tramite un ADC, in un corrispondente valore digitale.

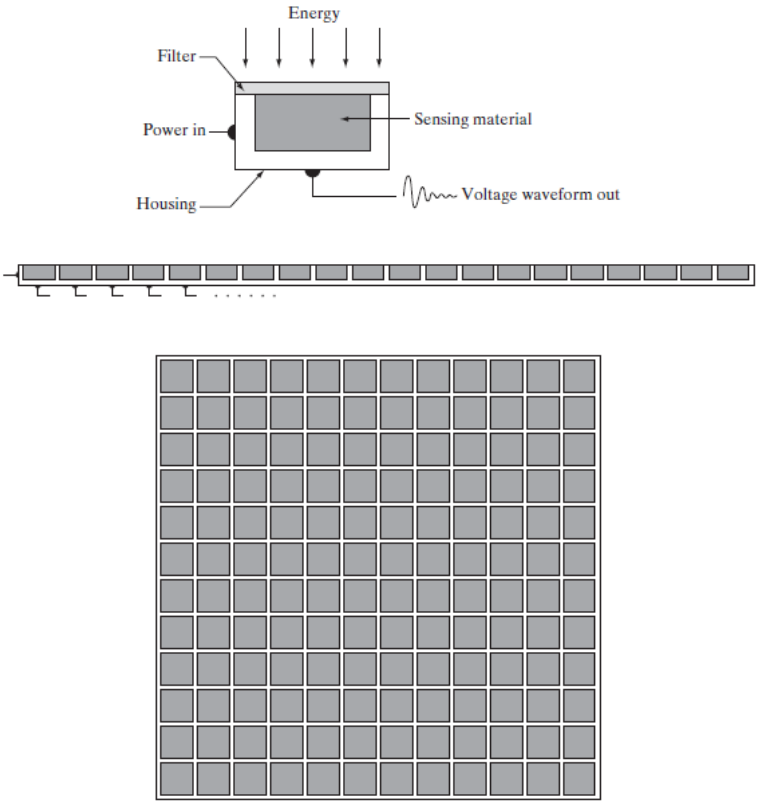


Figura 1.1 - Single imaging sensor, line sensor, array sensor

Il processo di digitalizzazione (campionamento, quantizzazione e codifica) restituisce gli elementi dell'immagine, anche noti come $pixels$ a cui verranno associati k bit in base alla rappresentazione utilizzata (B/N ⇒ 1 bit, scala di grigio ⇒ 8 bit o RGB ⇒ 24 bit, ovvero, 8 bit per componente), il cui corrispondente valore decimale, compreso tra $[0, 2^k - 1]$, rappresenta una discretizzazione del valore reale, inoltre, a causa della limitata portata del sensore, i valori d'intensità luminosa che si trovano al di fuori del suo range di funzionamento vengono mappati sui valori minimo e massimo, di conseguenza, l'immagine ottenuta sarà un'approssimazione della scena reale osservata. Occasionalmente, una scala "floating-point" viene utilizzata in applicazioni come l'imaging medico o spaziale, in cui è richiesta una maggiore precisione. Anche l'impiego di un maggior numero di sensori garantisce una maggiore qualità dell'immagine, poiché si riduce la porzione di scena associata ad ogni sensore e, quindi la luce media assorbita tende ad essere la luce in un punto la quale è più probabile che questa rientri nel range desiderato.

Un esempio di processo di acquisizione di un'immagine digitale in 2D è illustrato nella **fig.1.2**.

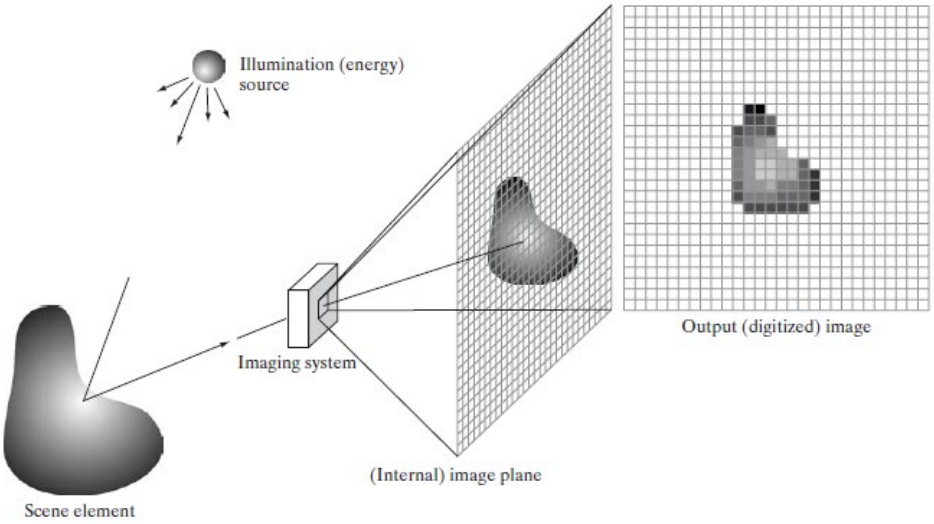


Figura 1.2 - An example of the digital image acquisition process

In termini più formali, un'immagine digitale I è una funzione bidimensionale di coordinate intere $\mathbb{N} \times \mathbb{N}$ che mappa un intervallo di valori dell'immagine $\mathbb{P}$ tale che:

$$I(u, v) \in \mathbb{P} \qquad con \qquad u, v \in \mathbb{N}$$

Un esempio di un'immagine digitale è illustrato nella **fig.1.3**.

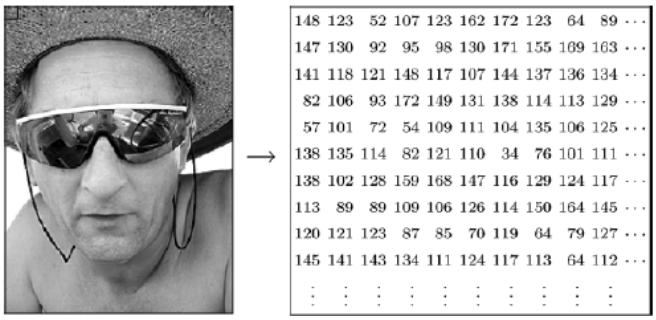


Figura 1.3 - Trasformation of a continuous image F(x,y) to a discrete digital image I(u,v)

La dimensione di un'immagine è determinata dal numero di righe N e dal numero di colonne M della matrice dell'immagine I, dove $I(u, v)$ rappresenta il generico valore nel dominio spaziale associato al generico elemento (o pixel) dell'immagine (vedi **fig.1.4**).

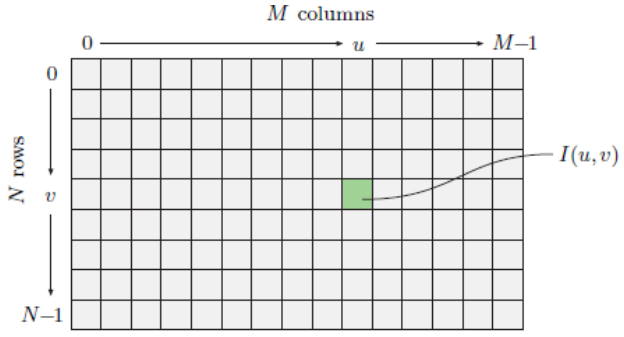


Figura 1.4 - Image spatial coordinates

L'impiego di tecniche di $digital\ image\ processing$ può essere definito come il sottoporre le immagini digitali a una serie di operazioni matematiche al fine di apportare opportune modifiche e ottenere il risultato desiderato. Nello specifico, il filtraggio delle immagini si pone come obiettivo l'ottimizzazione di quest'ultime dal punto di vista della qualità (minimizzando il fenomeno della pixelatura), rilevabilità o rimozione di alcuni dettagli, riduzione del rumore, miglioramento del contrasto, nitidezza e correzione del colore.

2 Filtraggio nel dominio dello spazio

Il filtraggio di un'immagine nel dominio spaziale opera sul piano della stessa, ovvero, si basa sulla manipolazione diretta dei pixel dell'immagine in contrapposizione, ad esempio, al dominio della frequenza in cui le operazioni vengono eseguite sulla trasformata di Fourier dell'immagine. In generale, le tecniche di elaborazione che operano nel dominio spaziale sono più efficienti dal punto di vista computazionale e richiedono meno risorse di elaborazione per essere implementate. Dal punto di vista analitico, il filtraggio spaziale è denotato dall'espressione:

$$I'(u,v) = T[I(u,v)]$$

dove I rappresenta l'immagine da filtrare, mentre I' è l'immagine filtrata ottenuta applicando un filtro H di dimensione $h \times h$ all'immagine I tramite un operatore T a cui sono associate una serie di operazioni necessarie ad ottenere l'immagine filtrata. Applicare il filtro a un'immagine è un processo semplice, come illustrato nella **fig.2.1**.

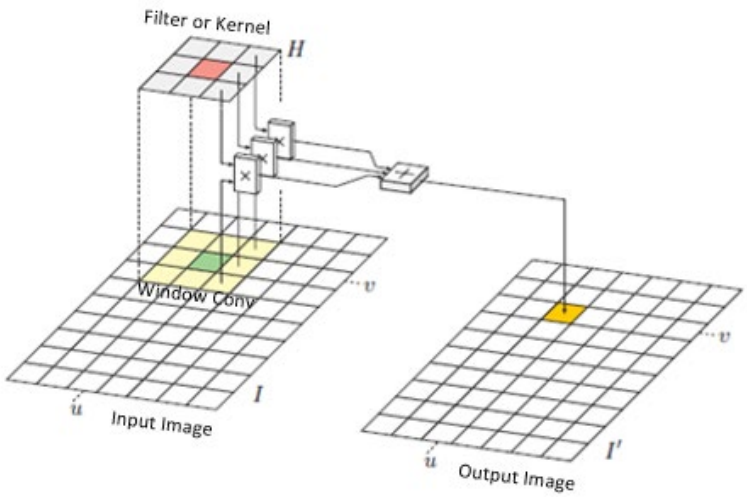


Figura 2.1 - Spatial filtering

I seguenti passaggi vengono effettuati per ciascun pixel dell'immagine I, i quali vengono letti in sequenza utilizzando una scansione $raster$, ovvero riga per riga (vedi **fig.2.2**):

1. Una volta definito il tipo di anchor point, il filtro o kernel H viene spostato su ciascun pixel dell'immagine originale I. Generalmente, si utilizza un anchor point centrale in modo tale che l'elemento centrale del filtro sia contrapposto al generico pixel da filtrare $I(u,v)$. Ciò restituisce l'intorno del generico pixel da filtrare, ovvero, quella che è nota come $finestra\ di\ convoluzione$ di dimensione $h \times h$.

2. Gli elementi del filtro $H(i,j)$, anche noti come $coefficienti\ del\ filtro$, vengono moltiplicati per gli elementi della finestra di convoluzione in posizione omologa $I(u+i, v+j)$ e i prodotti ottenuti dall'operazione di moltiplicazione vengono sommati. La somma risultate restituisce il generico pixel filtrato $I'(u,v)$.

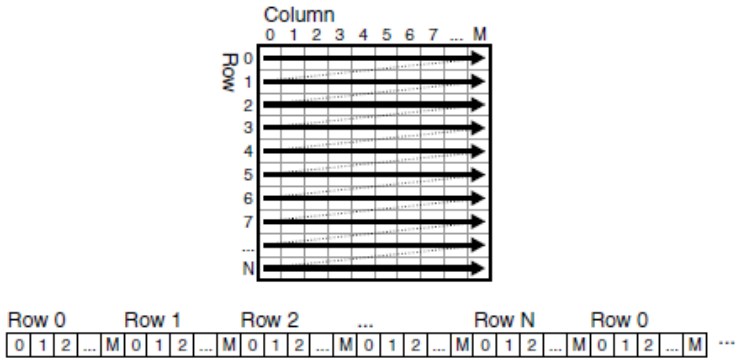


Figura 2.2 - Raster scan

In definitiva, il filtraggio spaziale è riconducibile ad un'operazione di convoluzione, più in particolare ad un'operazione MAC $(Multiply\ Accumulate)$. Analiticamente:

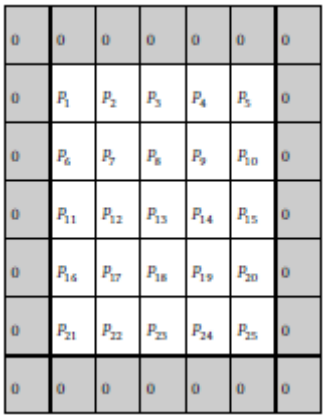
$$I'(u,v) = \sum_{(i,j)\in R_H} I(u+i, v+j) \cdot H(i,j)$$

dove R_H indica l'insieme dei pixel coperti dal filtro H.
Per un filtro 5×5, come quello impiegato per questo elaborato, la relazione precedente può essere riscritta come:

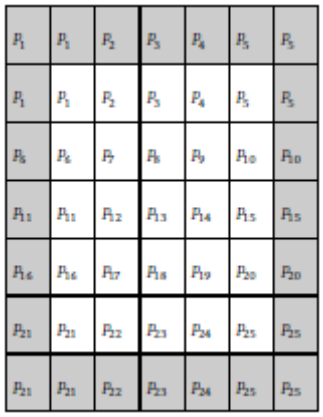
$$I'(u,v) = \sum_{i=0}^{4} \sum_{j=0}^{4} I(u+i, v+j) \cdot H(i,j)$$

Tuttavia, esiste un problema evidente ai bordi dove il filtro si estende all'esterno dell'immagine e non trova valori di pixel corrispondenti da utilizzare nel processo di filtraggio, poiché parte della finestra di convoluzione risulta incompleta. Per ovviare a questo problema, si ricorre a differenti tecniche di padding per aggiungere gli elementi mancanti alla finestra di convoluzione, quali:

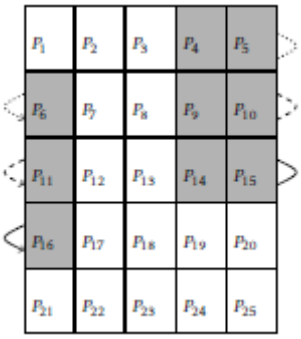
- $Zero\ padding$, gli elementi inesistenti vengono sostituiti con un valore costante pari a '0'.



- $Mirror\ padding$, l'immagine viene specchiata su tutti e quattro i bordi.



- $Estensione\ toroidale$, si considerano adiacenti l'ultimo pixel della riga i-esima ed il primo pixel della riga successiva.



2.1 Kernel isotropico

L'operazione di convoluzione è molto costosa dal punto di vista computazionale, poiché richiede una notevole quantità di calcoli: per un generico pixel di input, appartenente ad un'immagine di dimensioni $n \times m$, dato un kernel di dimensioni $k \times k$, sono richieste $(m \times n) \times k^2$ moltiplicazioni e $(m \times n) \times (k^2 - 1)$ addizioni. Una singola moltiplicazione richiede una notevole quantità di risorse hardware e produce tempi lunghi. Al fine di migliorare l'operazione di convoluzione, è possibile far uso di un kernel isotropico.

Un kernel si dice isotropico quando i coefficienti equidistanti dal centro hanno lo stesso valore, dove la distanza è intesa come la differenza tra gli indici (riga, colonna) e, le possibili direzioni lungo la quale la si può valutare sono orizzontale, verticale e diagonale. Sfruttare l'isotropicitá del kernel in un'operazione di filtraggio garantisce una riduzione dei prodotti e quindi del carico computazionale, questo implica delle semplificazioni a livello circuitale, poiché si riduce il numero di moltiplicatori e, conseguentemente la complessità dell'albero di somma.
La **fig.2.3** illustra un generico kernel isotropico 5×5, come quello utilizzato per testare il circuito di filtraggio.

KC1(0,0)	KC5(0,1)	KC2(0,2)	KC5(0,3)	KC1(0,4)
KC5(1,0)	KC3(1,1)	KC4(1,2)	KC3(1,3)	KC5(1,4)
KC2(2,0)	KC4(2,1)	KC6(2,2)	KC4(2,3)	KC2(2,4)
KC5(3,0)	KC3(3,1)	KC4(3,2)	KC3(3,3)	KC1(3,4)
KC1(4,0)	KC5(4,1)	KC2(4,2)	KC5(4,3)	KC1(4,4)

Figura 2.3 - Kernel isotropico 5x5

Si dispone di 6 differenti coefficienti, pertanto, è sufficiente eseguire solo 6 operazioni di moltiplicazione, anziché 25, per i pixel della generica finestra di convoluzione accomunati dallo stesso coefficiente, preliminarmente sommati.

	Filtro non isotropico	**Filtro isotropico**
# Moltiplicazioni	102400	24576
# Addizioni	98304	98304

Analiticamente, per la generica finestra di convoluzione illustrata in **fig.2.4**, il generico pixel filtrato è dato da:

$$\begin{aligned} P_{filtered}(i,j) = {} & [P(0,0) + P(0,4) + P(4,0) + P(4,4)] \cdot KC_1 \\ & + [P(0,2) + P(2,0) + P(2,4) + P(4,2)] \cdot KC_2 \\ & + [P(1,1) + P(1,3) + P(3,1) + P(3,3)] \cdot KC_3 \\ & + [P(1,2) + P(2,1) + P(2,3) + P(3,2)] \cdot KC_4 \\ & + [P(0,1) + P(0,3) + P(1,0) + P(1,4) + P(3,0) + P(3,4) + P(4,1) + P(4,3)] \\ & \cdot KC_5 + P(2,2) \cdot KC_6 \end{aligned}$$

P(0,0)	P(0,1)	P(0,2)	P(0,3)	P(0,4)
P(1,0)	P(1,1)	P(1,2)	P(1,3)	P(1,4)
P(2,0)	P(2,1)	P(2,2)	P(2,3)	P(2,4)
P(3,0)	P(3,1)	P(3,2)	P(3,3)	P(3,4)
P(4,0)	P(4,1)	P(4,2)	P(4,3)	P(4,4)

Figura 2.4 - Finestra di convoluzione del generico pixel da filtrare

3 Architettura del circuito di filtraggio spaziale 2D

La **fig.3.1** illustra lo schema a blocchi del circuito di filtraggio spaziale implementato su FPGA ed applicato ad un'immagine 64×64 in scala di grigi a 8 bit unsigned con un Kernel isotropico 5×5 signed.

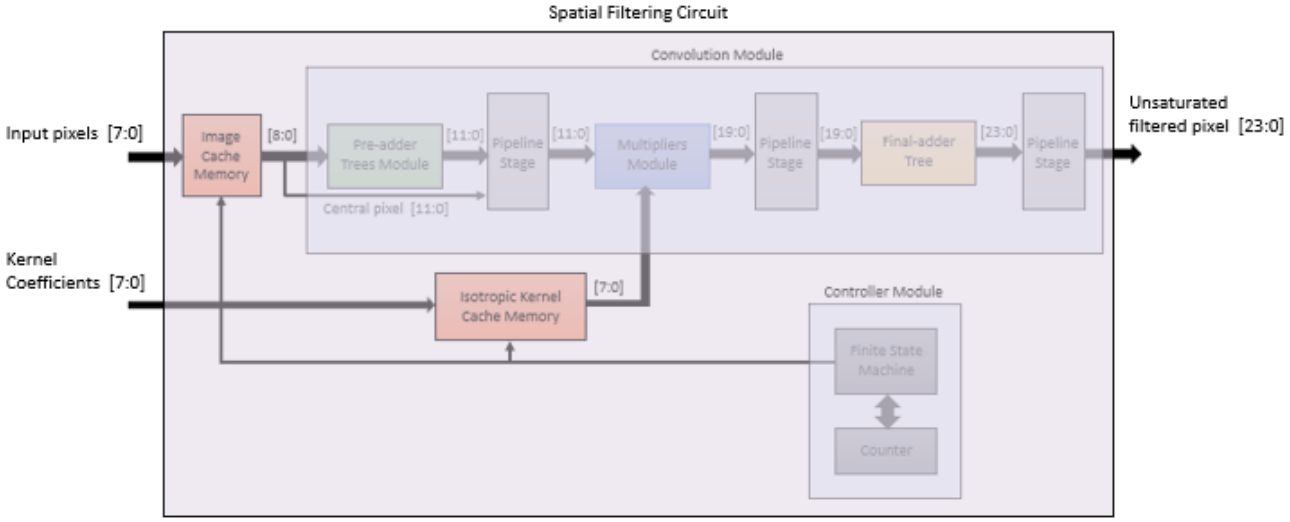


Figura 3.1 - Diagramma a blocchi del circuito di filtraggio spaziale per immagini 2D

1. I pixel dell'immagine sorgente vengono memorizzati all'interno di una $cache$, costituita da una cascata di registri sincroni (registro a scorrimento), necessaria per la corretta costruzione della finestra di convoluzione dell'i-esimo pixel di input. Si presuppone che i pixel vengano letti in sequenza da una memoria esterna in cui sono stati precedentemente memorizzati in modalità $Raster\ Order$, di conseguenza, poiché i pixel di cui è composta la generica finestra di convoluzione sono tra loro non adiacenti, il non impiego di un circuito di bufferizzazione porterebbe alla generazione di una finestra di convoluzione errata.
2. I coefficienti del kernel isotropico sono anch'essi memorizzati in un $buffer$ (o registro a scorrimento), presupponendo che vengano letti in sequenza da una BRAM in cui sono stati precedentemente memorizzati. In questo contesto, i coefficienti del kernel possono essere aggiornati per alterare l'effetto dell'operazione di filtraggio.
3. Sapendo che il filtraggio è riconducibile ad un'operazione di convoluzione, più in particolare ad un'operazione MAC (Multiply-Accumulate), ovvero, una somma di prodotti, viene progettato un $modulo\ di\ convoluzione$ composto da diversi sotto-moduli: un $modulo\ Pre-adders$ che consente di mettere in evidenza i coefficienti del kernel che si ripetono, andando a sommare in parallelo, in via preliminare, i pixels della finestra accomunati allo stesso coefficiente del filtro isotropico, le cui uscite vanno in ingresso, insieme al pixel centrale, al $modulo\ Multipliers$ che esegue, in parallelo, l'operazione di moltiplicazione tra le somme parziali ed i relativi coefficienti del kernel. I prodotti così generati vengono, infine, sommati attraverso un $Adder-tree$, il cui risultato rappresenta il generico pixel filtrato non saturato.
4. Tra un sotto-modulo e l'altro del modulo di convoluzione vengono inseriti degli $stadi\ di\ pipeline$ per spezzare la logica combinatoria distribuendola su più cicli di clock. Ciò consente di ridurre il ritardo di propagazione su ciascun ciclo di clock, soddisfacendo i vincoli

temporali, e aumentare il throughput complessivo del circuito, ovvero, il numero di risultati validi generati ad ogni ciclo di clock, poiché mentre il sotto-modulo successivo elabora l'output del sotto-modulo precedente, quest'ultimo può iniziare ad elaborare un nuovo dato. Il pipelining garantisce anche una riduzione dei glitches (causa di una maggiore dissipazione di potenza dinamica) sull'uscita di ciascun sotto-modulo combinatorio, poiché rende disponibile il dato utile sul fronte sensibile del clock. Ovviamente, bisogna verificare che il dato campionato corrisponda effettivamente a quello voluto. Tuttavia, non sono stati inseriti stadi di pipeline internamente ai sotto-moduli stessi, poiché, nonostante il pipelining fornisce miglioramenti significativi delle prestazioni, comporta anche degli svantaggi, quali, maggior impiego di risorse logiche essendo composti da registri, e un aumento della latenza complessiva del circuito, ovvero, il numero di cicli di clock necessari prima di ottenere un risultato valido in seguito all'elaborazione.

5. L'$unità\ di\ controllo$, implementata come macchina a stati e gestista mediante un contatore, controlla il funzionamento del filtro, ovvero, quando iniziare e terminare l'elaborazione dei pixel.
6. Superata una certa latenza iniziale, ad ogni ciclo di clock, il circuito restituisce un risultato valido, ovvero, un pixel filtrato (throughput = 1).

Prima di effettuare l'implementazione in VHDL, viene condotta un'analisi carta e penna per comprendere con quanti bit dovessero lavorare i singoli moduli. Nella cache associata all'immagine entrano i valori dei pixel rappresentati su 8 bit unsigned. Questi vengono successivamente estesi su 9 bit e inviati in ingresso al modulo Pre-adders. Ciò si è ritenuto necessario per poterli effettivamente trattare come numeri positivi, poiché altrimenti i valori compresi tra 128 e 255 sarebbero stati considerati negativi, dal momento che tutti i moduli combinatori del circuito (sommatori e moltiplicatori) utilizzano la notazione signed in complemento a due. I dati in uscita dal modulo Pre-adders dovrebbero essere estesi su 14 bit per la presenza di un albero di somma a 8 ingressi strutturato su 5 livelli, infatti, per definizione il risultato di un albero di somma con k ingressi a n-bit è rappresentato con $(n + num.livelli)$ bit, tuttavia, è possibile rappresentali su 12 bit, poiché nel caso peggiore per cui tutti i pixel della finestra da processare siano pari a 255, il massimo numero che si ottiene è 2040, che può essere tranquillamente rappresentato su 12 bit. Successivamente, queste somme parziali vanno in ingresso al modulo Multipliers per essere moltiplicate con i corrispettivi coefficienti del kernel a 8 bit signed. I prodotti ottenuti vengono rappresentati con 20 bit: per definizione il risultato di un'operazione di moltiplicazione tra due operandi ad n-bit e m-bit è rappresentato con $(n + m)$ bit. Infine, questi prodotti vanno in ingresso ad un albero di somma che produce un risultato a 24 bit. Anche in questo caso sarebbe stato possibile troncare il risultato a 21 bit, ma siccome rappresenta l'uscita complessiva del circuito, non è stato ritenuto necessario. Invece, se ci fossero stati altri circuiti a valle, sarebbe stato opportuno attuare il troncamento, poiché avrebbe ulteriormente ridotto il carico computazionale complessivo.
Nelle **fig.3.2-3.3** viene riportato, rispettivamente, lo schematic RTL del circuito di filtraggio a livello di modulo e l'esploso dello stesso.

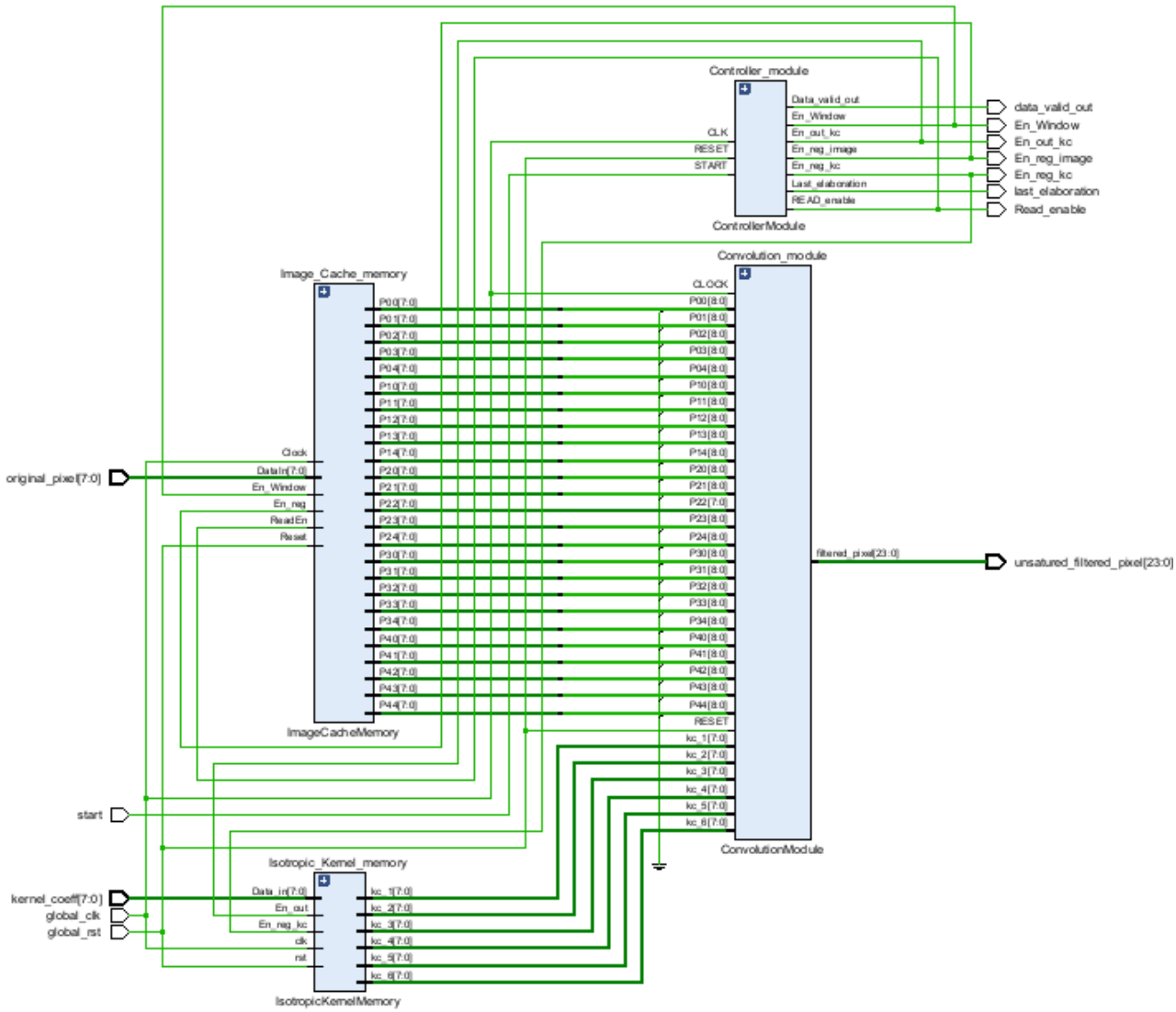


Figura 3.2 - Schematic RTL del circuito di filtraggio spaziale

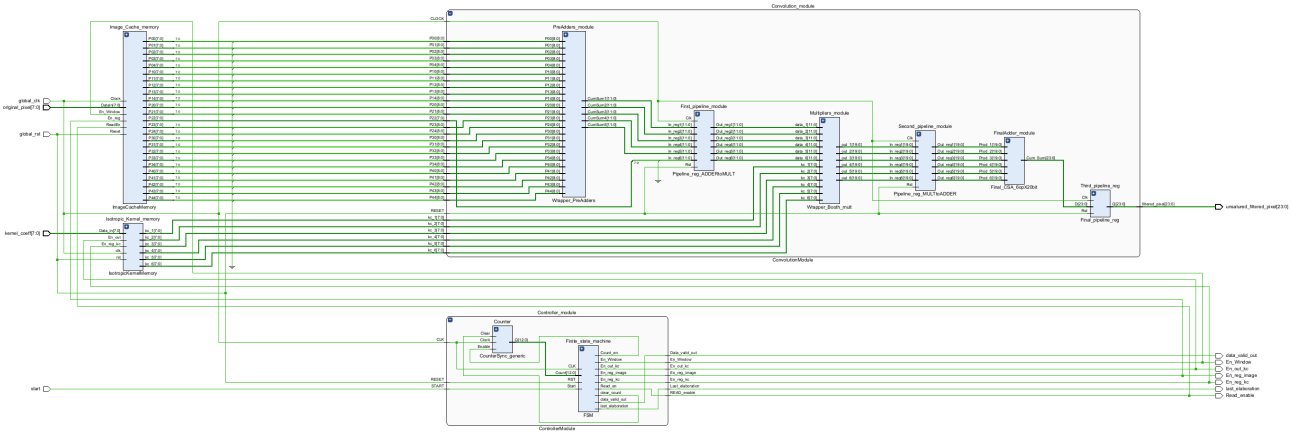


Figura 3.3 - Esploso shematic RTL del circuito di filtraggio spaziale

3.1 Memoria cache o circuito di bufferizzazione dell'immagine

La memoria cache è strutturata in modo da bufferizzare una porzione di pixel dell'immagine da filtrare, che dipende dalle dimensioni dell'immagine e del Kernel, garantendo la corretta costruzione della finestra di convoluzione, ovvero, dell'intorno del generico pixel da filtrare che, come è noto, è costituito da pixel tra loro non adiacenti. Questa soluzione evita la necessità di bufferizzare l'intera immagine sfruttando i blocchi di memoria presenti sul chip FPGA. Infatti, quest'ultimo metodo costituisce un utilizzo inefficiente dei blocchi di memoria e può limitare la capacità del chip di elaborare immagini di grandi dimensioni, ovvero, può diventare un fattore limitante per immagini di grandi dimensioni.
Il tipo di memorizzazione utilizzato nella cache, per garantire la corretta costruzione della finestra di convoluzione del generico pixel da filtrare e ridurre il numero di letture per ogni ciclo di clock, è il buffering delle righe, per cui, per una finestra di dimensioni $w \times w$, i pixel delle $w - 1$ righe precedenti (poiché i pixel della prima riga possono essere scartati) vengono memorizzati in un rispettivo buffer di riga per essere riutilizzati durante la lettura dei pixel dell'i-esima riga (vedi **fig.3.4**). Questa struttura consente di preservare il movimento dei dati in streaming, eliminando la necessità di archiviare un'immagine per intero.
La quantità di pixel da memorizzare all'interno della cache, al fine di ottenere la prima finestra di convoluzione utile per poter essere elaborata ed ottenere il primo pixel filtrato, viene determinata come segue:

$$Pixel\_richiesti = \big(larghezza\_immagine \times (altezza\_kernel - 1)\big) + larghezza\_kernel$$

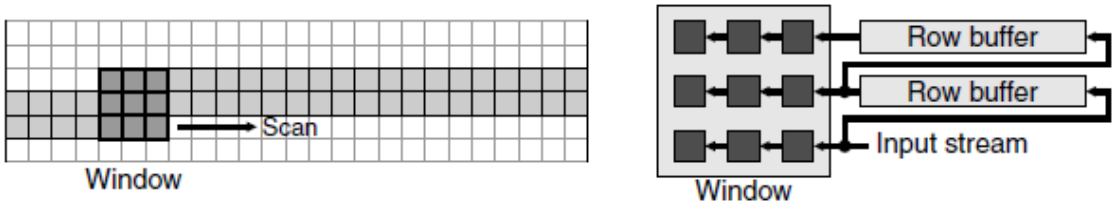


Figura 3.4 - Row buffering per memorizzare le righe precedenti in modo che non sia necessario leggerle nuovamente

Sia l'immagine di dimensioni $N \times M$ ed il Kernel di dimensioni $k \times k$, con $k = 2r + 1$, dove r è il raggio del filtro, la memoria cache sarà costituita da:

- $k \times k$ registri per i pixel della finestra da elaborare
- $k - 1$ buffer di riga composti da $m - k$ registri per i pixel già letti

Superata la latenza iniziale, ovvero, il tempo necessario per generare la prima finestra utile, ad ogni ciclo di clock il circuito di bufferizzazione restituirà una finestra valida da elaborare. La latenza è definita come:

$$Latenza = larghezza\_immagine \cdot raggio\_filtro + (raggio\_filtro + 1)$$

La struttura di bufferizzazione viene dimensionata per poter filtrare un'immagine 64×64 pixel in scala di grigi a 8 bit con un Kernel 5×5, di conseguenza, la cache sarà composta da:

- 25 registri per gli elementi della finestra da elaborare
- 4 buffer di riga composti da 59 registri

L'implementazione su FPGA di un buffer di riga come registro a scorrimento comporta l'utilizzo di una percentuale significativa di risorse logiche essendo costituito da una cascata di registri, in cui ogni bit utilizza un flip-flop. Questo potrebbe risultare problematico nel caso di immagini di grandi dimensioni.
La **fig.3.5** illustra il concetto di come dovrebbe funzionare la cache.

1. Viene abilitato il segnale di reset per inizializzare tutti i registri (memorizzano uno '0').
2. Il segnale di controllo del multiplexer a monte della struttura viene impostato in modo tale che la memoria cache possa leggere in sequenza i pixel dell'immagine sorgente, presupponendo che siano stati precedentemente memorizzati all'interno di una memoria esterna in modalità $Raster\ Order$.
3. Ad ogni colpo di clock, una volta abilitati i registri e presupponendo che questi siano sincroni condividendo lo stesso segnale di clock, entra all'interno della cache un nuovo pixel ed ogni registro alla i-esima posizione campiona l'uscita del registro precedente, causando lo shift dei pixel.
4. Nel momento in cui il primo pixel immesso nella struttura raggiunge la posizione della finestra corrispondente all'anchor point, si ottiene la prima finestra utile.
5. Il segnale di controllo in comune tra i multiplexer a valle della struttura viene impostato in modo da abilitare le uscite e, quindi trasferire i pixel della finestra di convoluzione al modulo successivo, ovvero al componente convolutore per essere elaborati, ovvero per eseguire le operazioni matematiche inerenti al processo di filtraggio. L'impiego di questi multiplexer consente di ridurre la dissipazione di potenza dinamica del modulo di convoluzione quando ancora non è trascorsa la latenza del circuito, poiché le uscite della cache, ovvero gli elementi della finestra di convoluzione rimangono statici, in particolare, mantengo il valore logico '0'.
6. Da questo momento in poi, ovvero, superata la latenza iniziale, ad ogni colpo di clock viene fornita una nuova finestra utile da elaborare.
7. Una volta letto l'ultimo pixel, il multiplexer a monte della struttura viene configurato per far entrare un valore nullo ad ogni ciclo di clock fino a quando non viene generata l'ultima finestra utile. A questo punto, i registri ed i multiplexer a valle vengono disabilitati.

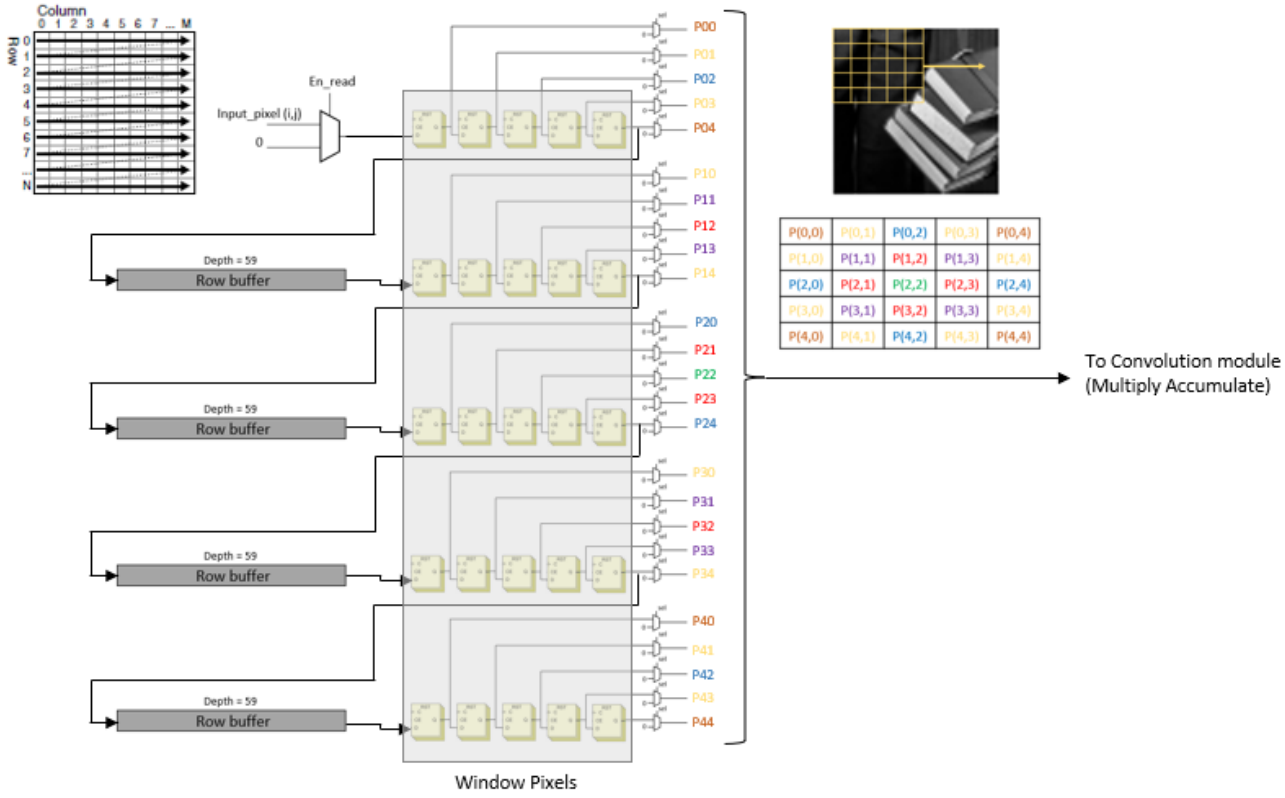


Figura 3.5 - Memoria cache o circuito di bufferizzazione per immagini 64x64

Per l'aggiunta degli elementi inesistenti nell'intorno di un generico pixel da filtrare, situato ai bordi dell'immagine sorgente, viene sfruttata una tecnica di padding ibrida (zero padding + toroidale). Non sono richiesti circuiti aggiuntivi per implementare la tecnica di padding, poiché, per come è stato progettato il circuito di bufferizzazione, riesce di per sé ad attuarla.
Da notare che, l'intero circuito di bufferizzazione può essere visto come un unico array monodimensionale composto da 261 registri connessi in cascata (da 0 a 260).

3.2 Memoria cache o circuito di bufferizzazione del kernel isotropico

Come per la cache dell'immagine sorgente, il buffer del kernel isotropico consente di memorizzare i coefficienti da utilizzare per l'operazione di convoluzione. La struttura di bufferizzazione viene dimensionata per un kernel 5×5, di conseguenza, sarà composta da 25 registri (da 0 a 24) in cascata. La **fig.3.6** illustra il concetto di come dovrebbe funzionare il buffer.

1. Viene abilitato il segnale di reset per inizializzare tutti i registri (memorizzano uno '0').
2. Ad ogni colpo di clock, una volta abilitati i registri e presupponendo che questi siano sincroni condividendo lo stesso segnale di clock, entra all'interno del buffer un nuovo coefficiente ed ogni registro alla i-esima posizione campiona l'uscita del registro precedente, causando lo shift dei coefficienti.
3. Nel momento in cui il primo coefficiente immesso raggiunge l'ultima posizione, i registri vengono disabilitati ed il segnale di controllo in comune tra i multiplexer a valle della struttura viene impostato in modo da abilitare le uscite e, quindi trasferire gli opportuni coefficienti al

modulo successivo, ovvero al componente convolutore per eseguire le operazioni matematiche inerenti al processo di filtraggio.

Il buffer del kernel lavora in parallelo al buffer dell'immagine, pertanto non introduce ulteriore latenza, inoltre, i coefficienti in uscita da questo modulo saranno disponibili molto prima che venga generata la prima finestra di convoluzione utile.

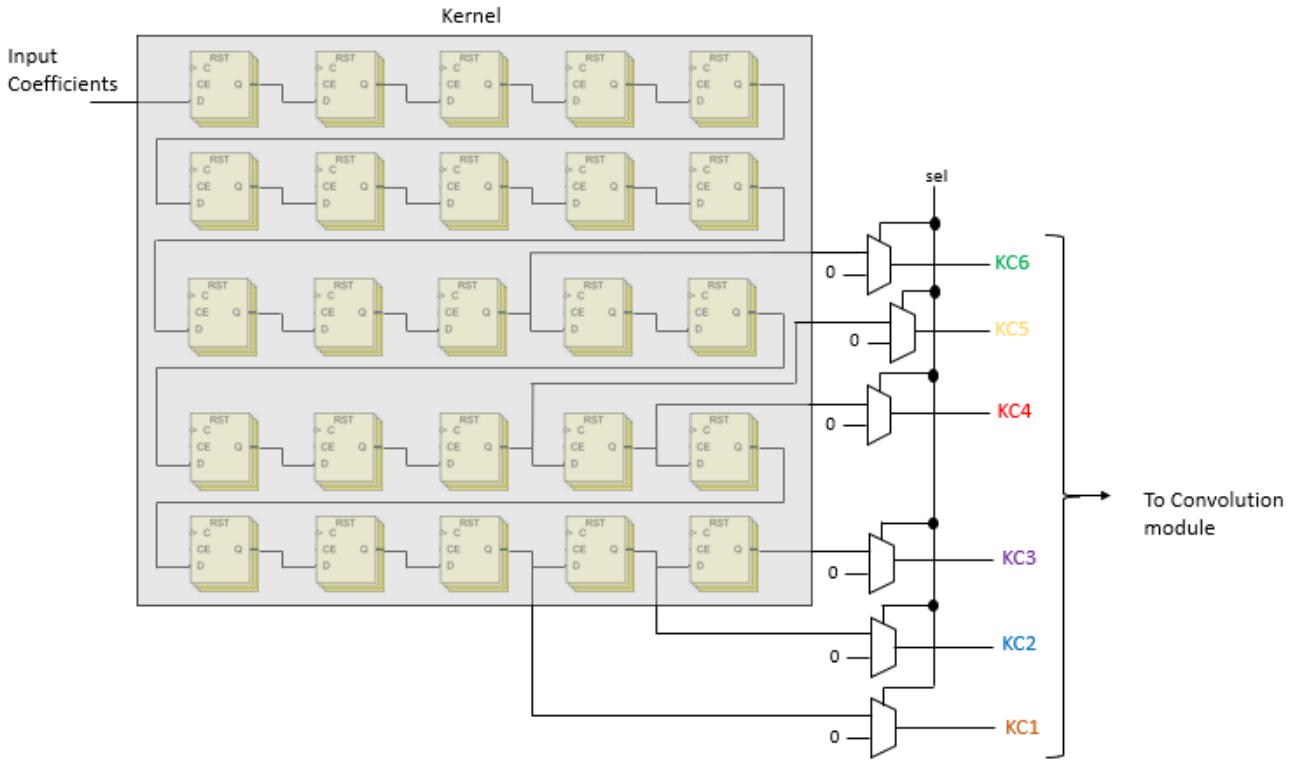


Figura 3.6 - Buffer filtro isotropico 5x5

3.3 Circuiti sommatori

Per i circuiti sommatori del modulo Pre-adders, del modulo Multipliers e per il sommatore finale a valle del circuito di filtraggio, in cui è richiesta la somma di più operandi, viene adottato l'approccio ad albero di somma Carry Save.

Poiché viene utilizzato un filtro isotropico 5×5, il modulo Pre-adders è costituito da 5 alberi di somma Carry Save in parallelo (4 sommatori a 4 operandi e 1 sommatore a 8 operandi), ognuno dei quali effettua la somma tra i valori dei pixel accomunati dallo stesso coefficiente del kernel, il modulo Multipliers è costituito da 6 alberi di somma Carry Save in parallelo (6 sommatori a 4 operandi), ognuno dei quali esegue la somma dei prodotti parziali derivanti dall'operazione di moltiplicazione e, infine, l'albero di somma Carry Save a valle del circuito di filtraggio (sommatore a 6 operandi) esegue la somma dei prodotti in uscita dal modulo Multipliers.

Il concetto base dell'approccio Carry Save è quello di parallelizzare il calcolo di somma a 3 operandi limitando, eventualmente, la propagazione del riporto all'ultimo stadio del processo: una cascata di Full-adder calcolano i bit di somma senza effettuare la propagazione del carry, ciò significa che il bit di somma i-esimo viene determinato senza attendere il carry dalla posizione precedente. I bit di

somma e i bit di carry vengono raggruppati in vettori noti come VSP (vettore somma parziale) e VR (vettore dei riporti). Non essendoci propagazione del carry, il tempo di calcolo di VSP e VR è pari al ritardo di un Full-adder, τ_{FA}. Una volta effettua l'estensione con segno di VSP e lo shift verso sinistra di VR in modo da allinearli, questi vettori vengono sommati con un sommatore tradizionale ed estesi con segno nel caso di operandi di tipo signed in complemento a 2, il cui ritardo dipende dal tipo di sommatore utilizzato. Per questo elaborato si è fatto uso di un sommatore Ripple Carry di tipo signed che introduce un ritardo proporzionale al numero di bit degli operandi in ingresso.
In definitiva, l'approccio Carry Save offre prestazioni migliori in termini di velocità di elaborazione rispetto al caso in cui viene impiegato un classico sommatore a due ingressi per la progettazione di un albero di somma, infatti, sapendo che in un albero di somma si ha un'accumulazione del ritardo ad ogni livello di somma, il ritardo complessivo per i due approcci, in questo caso specifico, è dato da:

Sommatore a 4 operandi	
Albero Ripple-Carry-Adder	**Albero Carry-Save-Adder**
$\tau = \tau_{RCA(n+1)} + \tau_{RCA(n+2)}$	$\tau = 2\tau_{FA} + \tau_{RCA(n+2)}$

Sommatore a 8 operandi	
Albero Ripple-Carry-Adder	**Albero Carry-Save-Adder**
$\tau = \tau_{RCA(n+1)} + \tau_{RCA(n+2)} + \tau_{RCA(n+3)}$	$\tau = 4\tau_{FA} + \tau_{RCA(n+4)}$

In generale, un albero di somma Carry Save a k operandi è composto da $\log_2 k$ livelli di FA più un livello associato al sommatore tradizionale.
Nelle **fig.3.7-3.10** viene riportato l'esploso dello schematic RTL degli alberi di somma Carry Save impiegati nei vari sotto-moduli del modulo Convolutore.

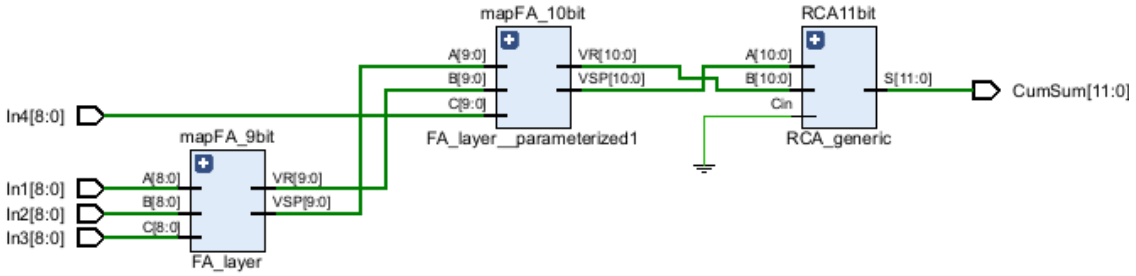


Figura 3.7 - Schematic RTL sommatore 4 operandi del modulo Pre-adders

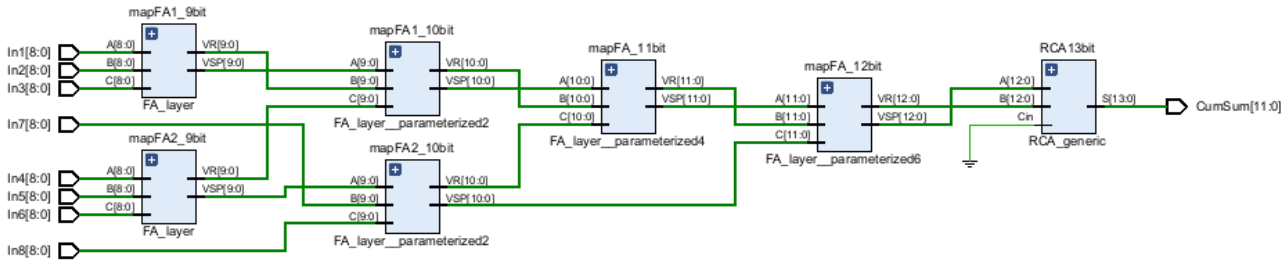


Figura 3.8 - Schematic RTL sommatore 8 operandi del modulo Pre-adders

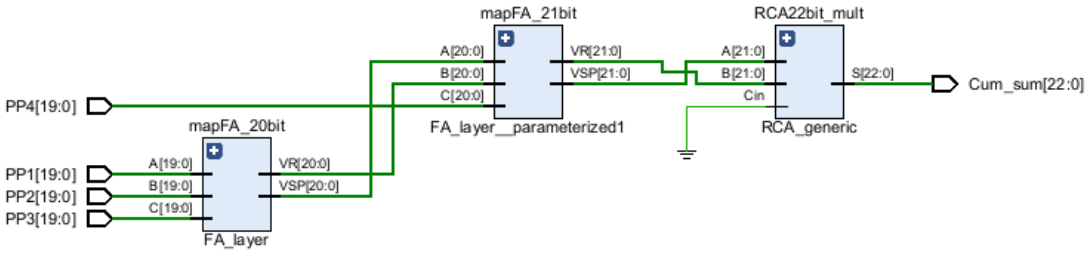


Figura 3.9 - Schematic RTL sommatore 4 operandi del modulo Multipliers

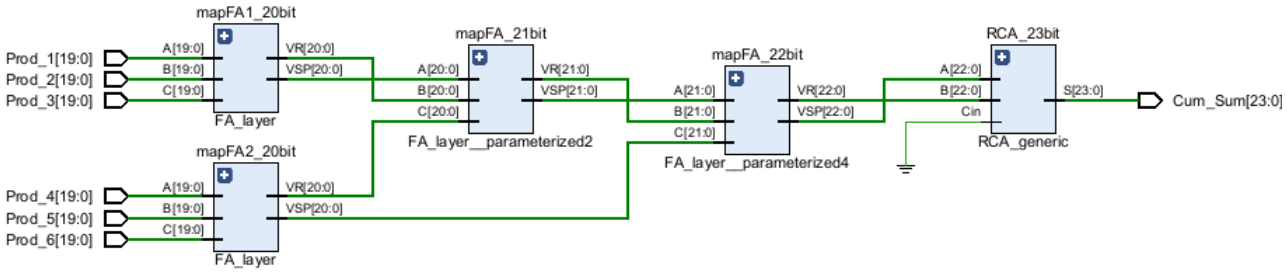


Figura 3.10 - Schematic RTL sommatore finale 6 operandi a valle del modulo Convolutore

Nelle **fig.3.11** viene illustrato lo schematic RTL del modulo Pre-adders.

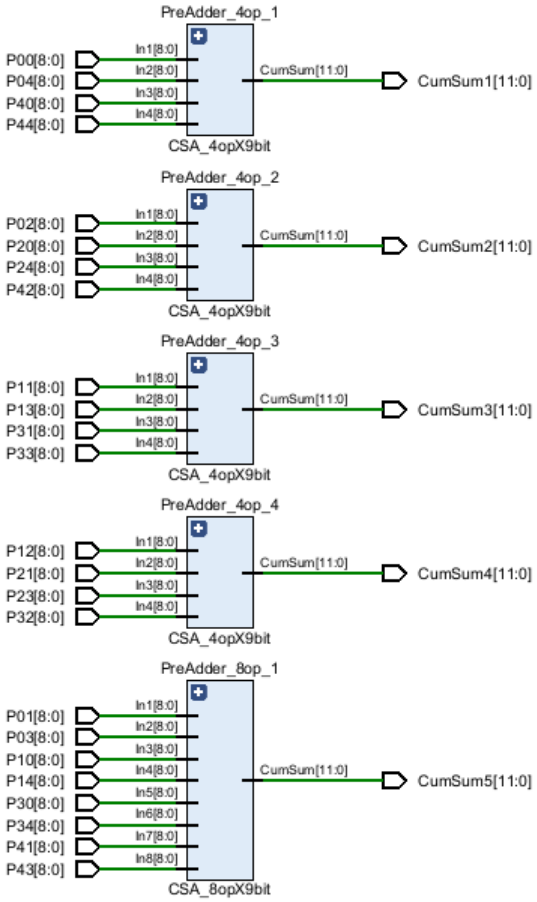


Figura 3.11 - Schematic RTL modulo Pre-adders

3.4 Modulo Multipliers

Il modulo Multipliers è costituito da 6 moltiplicatori di Booth in parallelo, il cui compito è quello di eseguire la moltiplicazione tra le somme parziali in uscita dal modulo Pre-adders ed i rispettivi coefficienti del kernel in uscita dal buffer impiegato per memorizzarli.
Il moltiplicatore di Booth utilizza una codifica del moltiplicatore, rappresentato dai coefficienti del kernel, per ridurre il numero di prodotti parziali, in particolare, si dimezza il numero di prodotti parziali, di conseguenza, si riduce la complessità, il tempo di elaborazione e la dissipazione di potenza dell'albero di somma Carry Save utilizzato per sommare i prodotti parziali.
La codifica di Booth prevede che i bit nel moltiplicatore, a partire dall'LSB, vengano suddivisi in terne di bit e che ciascuna terna abbia in comune un bit con la terna successiva, inoltre, se la terna LSB risulta incompleta si aggiungono degli zeri, mentre, se è la terna MSB a risultare incompleta si aggiungono degli uni. Una volta ottenute le terne, ad ognuna di esse viene associata una certa codifica (vedi **fig.3.12**), rappresentata in digit, che permette di determinare il prodotto parziale associato alla rispettiva terna semplicemente basandosi sul valore del moltiplicando, rappresentato dalla somma parziale corrispondente.

Terna b_{i+1} b_i b_{i-1}	Digit codificato	Prodotto parziale
0 0 0	0	0
0 0 1	1	A
0 1 0	1	A
0 1 1	2	2A
1 0 0	-2	-2A
1 0 1	-1	-A
1 1 0	-1	-A
1 1 1	0	0

Figura 3.12 - Tabella codifica di Booth

Questa soluzione garantisce dei vantaggi, in quanto, conoscendo il valore del moltiplicando, è possibile determinare preventivamente tutti i possibili valori dei prodotti parziali e, utilizzando un banale multiplexer, selezionare il prodotto parziale da associare ad una particolare terna. Per un moltiplicando ad n bit, sono necessari $n + 2$ bit per rappresentare il generico prodotto parziale. Ogni prodotto parziale viene shiftato verso sinistra di una quantità pari al peso del bit centrale della rispettiva terna, perciò, prima di sommarli ed ottenere il risultato della moltiplicazione, è necessario allinearli effettuando un'estensione del segno. Riguardo alla classificazione dei due operandi, si è deciso di considerare come moltiplicatore i coefficienti del kernel piuttosto che le uscite del modulo Pre-adders, poiché garantiscono un minor numero di prodotti parziali, in quanto risultano rappresentati da un minor numero di bit.
La **fig.3.13** illustra lo schematic RTL di uno dei moltiplicatori di Booth che costituiscono il modulo Multipliers, composto da:

- $Logica\ di\ pre-calcolo$, per calcolare in parallelo, in via preliminare, tutti i prodotti parziali che possono essere associati alla generica terna del moltiplicatore, evitando di doverli determinare in fase di processamento, il che costituirebbe motivo di ulteriore rallentamento

nei tempi di calcolo. Per calcolare quantità come -A e -2A, dove A rappresenta il moltiplicando, è necessario usare un sommatore, dal momento che entrambe vengono determinate calcolandone il complemento a 2, il quale è dato dalla somma tra l'unità e il negato del moltiplicando stesso (perciò vi sono un RCA e una porta NOT). Inoltre, nel calcolo di -2A vengono aggiunti due bit, uno al MSB, per effetto della somma, e uno al LSB, a causa della moltiplicazione per due (si shifta a sinistra di un bit). Questo significa che per poter rappresentare -2A correttamente, è necessario aggiungere due bit. Di conseguenza, tutti i prodotti parziali dovranno essere maggiorati di due bit.

- $Booth\ encoders$, determinano la codifica da associare alla generica terna, la quale, piuttosto che essere rappresentata come digit, viene indicata come una sequenza di 3 bit, utilizzando la rappresentazione modulo e segno (l'MSB indica il segno e due LSB quantificano il modulo), che verrà utilizzata come selettore dei multiplexers.

B(i+1) B(i) B(i-1)	Cj(2:0)	Cj(2) Cj(1) Cj(0)
0 0 0	0	0 0 0
0 0 1	1	0 0 1
0 1 0	1	0 0 1
0 1 1	2	0 1 0
1 0 0	-2	1 1 0
1 0 1	-1	1 0 1
1 1 0	-1	1 0 1
1 1 1	0	0 0 0

La scelta della codifica permette di ottenere delle ottimizzazioni a livello circuitale (utilizzo mux 5:1, anziché mux 8:1) e, conseguentemente, si avrà una minore dissipazione di potenza.

- $Multiplexers$, si tratta di mux 5:1 che, sulla base della codifica prodotta dal Booth encoder della generica terna, selezionano e portano in uscita il prodotto parziale associato a quella particolare codifica.
- $Albero\ di\ somma\ con\ approccio\ Carry\ Save$ (vedi **fig.3.9**), riceve in ingresso i prodotti parziali in uscita dai multiplexers che, prima di essere sommati per ottener il risultato della moltiplicazione, vengono shiftati ed estesi con segno opportunamente.

I prodotti in uscita dai moltiplicatori di Booth vengono, infine, sommati attraverso l'albero di somma Carry Save, il cui schematic è illustrato in **fig.3.10**, per ottenere il risultato complessivo dell'elaborazione, ovvero, il pixel filtrato.

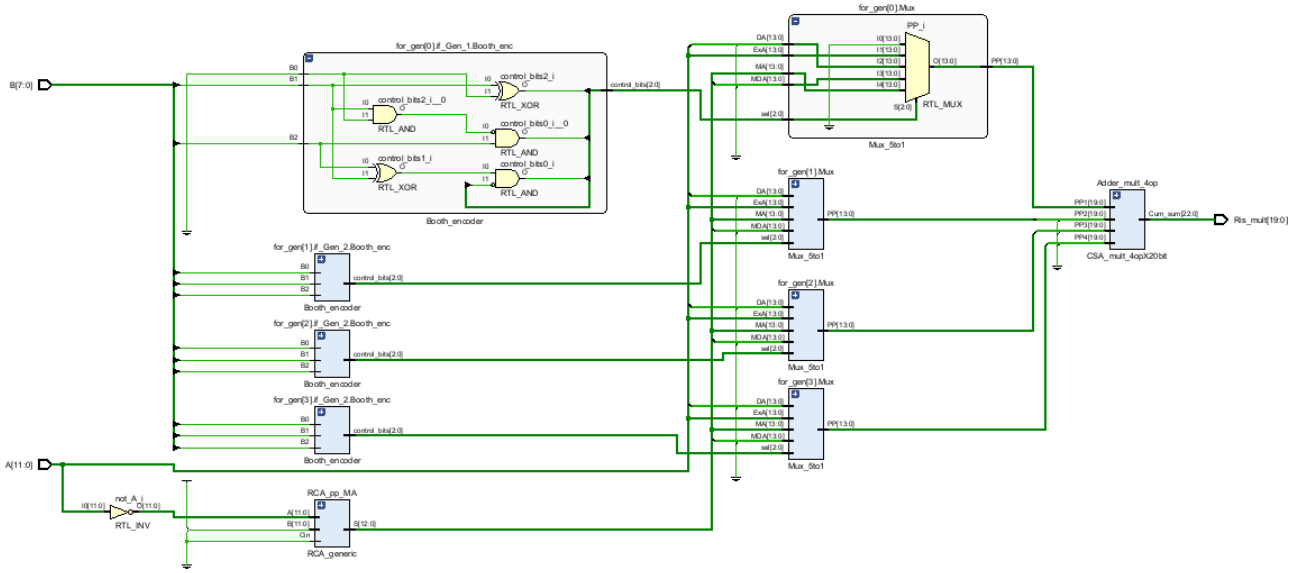


Figura 3.13 – Schematic RTL moltiplicatore di Booth

3.5 Modulo di controllo

Il modulo di controllo, che si occupa della gestione del circuito di filtraggio durante la fase di elaborazione dei dati, è composto da due sotto blocchi, quali:

- $Macchina\ a\ stati\ finiti\ (FSM)\ sincrona$, è l'entità che gestisce i vari stati del sistema. Essa è composta da segnali di input, stati finiti e segnali di output. Lo stato in cui si trova in un dato istante è chiamato stato corrente. La FSM viene implementata utilizzando il modello di Moore: la transizione da uno stato a quello successivo dipende dallo stato corrente e dagli eventuali ingressi applicati alla FSM, mentre le uscite dipendono dallo stato corrente. Nella **fig.3.14** è illustrato un modello generale di una macchina sequenziale sincrona di Moore.

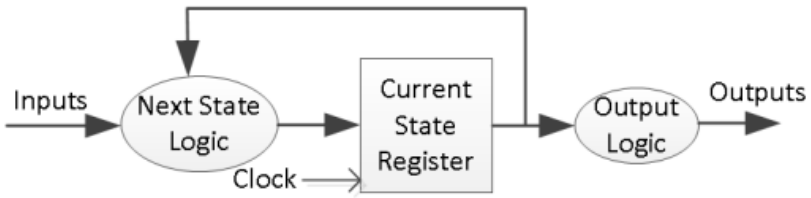


Figura 3.14 - Moore Finite State Machine

In particolare, la FSM progettata per il circuito di filtraggio si occupa della gestione di 7 stati. Le transizioni da uno stato a quello successivo vengono gestite attraverso un contatore tramite il conteggio dei dati in ingresso (uno per ogni ciclo di clock). Questo rappresenta il secondo sotto-blocco fondamentale del modulo di controllo e la sua architettura verrà descritta in un secondo momento. La classificazione dei vari stati della FSM è preceduta dalla descrizione dei segnali di ingresso e uscita della stessa, ai fini di una migliore comprensione di ciò che accade all'interno del circuito.

  - $global\_rst$: serve per inizializzare il circuito.

- $start$: gestisce il funzionamento o meno del circuito.
- $Read\_en$: gestisce la lettura o meno dei pixels da parte del circuito di bufferizzazione.
- En_reg_image: gestisce l'abilitazione o meno dei registri del circuito di bufferizzazione relativo ai pixels.
- En_reg_kc: gestisce l'abilitazione o meno dei registri del circuito di bufferizzazione relativo ai coefficienti del kernel.
- En_window, En_out_kc: gestiscono l'abilitazione o meno delle uscite dei circuiti di bufferizzazione.
- $Count\_en$: gestisce l'abilitazione o meno del contatore.
- $clear\_count$: si occupa di inizializzare il contatore e mantenerlo in stand-by fin tanto che questo segnale è attivo.
- $data\_valid\_out$: serve per comunicare la validità o meno dei risultati in uscita dal circuito.
- $last\_elaboration$: serve per comunicare l'elaborazione dell'ultimo pixel.

Gli stati vengono classificati come segue:

1. $IDLE$, il circuito viene inizializzato ponendo $global\_rst = '1'$ e rimane in fase di stand-by, in attesa di poter iniziare l'elaborazione. Questo rappresenta anche lo stato in cui si ritorna una volta elaborati tutti i dati in ingresso. Di seguito la configurazione dei segnali di uscita:

   $Read\_en \rightarrow '0'$

   $En\_reg\_image \rightarrow '0'$

   $En\_reg\_kc \rightarrow '0'$

   $En\_window \rightarrow '0'$

   $En\_out\_kc \rightarrow '0'$

   $Count\_en \rightarrow '0'$

   $data\_valid\_out \rightarrow '0'$

   $last\_elaboration \rightarrow '0'$

   $clear\_count \rightarrow '1'$

   Si rimane nello stato di $IDLE$ fin tanto che $start = '0'$.
   Se $start = '1'$ si passa allo stato $S1$.

2. $S1$, i circuiti di bufferizzazione ed il contatore vengono abilitati, dando inizio alla lettura dei pixels e dei coefficienti del kernel, durante il quale il circuito non dovrà eseguire alcuna elaborazione fin tanto che non sia trascorsa la latenza iniziale, nell'attesa che venga generata la prima finestra di convoluzione utile da processare. Ci si avvale del contatore per individuare l'istante in cui potrà avere inizio l'elaborazione, in particolare questa potrà essere avviata una volta che il conteggio sarà pari a 130, per cui il primo pixel avrà raggiunto la posizione centrale della finestra di convoluzione. In realtà la latenza è pari a 131 cicli di clock, ma dal momento che il

primo pixel viene letto quando il conteggio è pari a 0, questo raggiungerà la posizione centrale a 130. Di seguito la configurazione dei segnali di uscita:

$Read\_en \rightarrow '1'$

$En\_reg\_image \rightarrow '1'$

$En\_reg\_kc \rightarrow '1'$

$En\_window \rightarrow '0'$

$En\_out\_kc \rightarrow '0'$

$Count\_en \rightarrow '1'$

$data\_valid\_out \rightarrow '0'$

$last\_elaboration \rightarrow '0'$

$clear\_count \rightarrow '0'$

Si rimane nello stato $S1$ fin tanto che il conteggio non raggiunge il valore 24.
Se il conteggio è pari a 24 si passa allo stato $S2$.

3. $S2$, il circuito di bufferizzazione che memorizza i coefficienti del kernel viene disabilitato, terminando la lettura di quest'ultimi, e i coefficienti d'interesse al suo interno vengono portate in uscita. In parallelo, continuano ad essere letti i pixels da processare. Di seguito la configurazione dei segnali di uscita:

   $Read\_en \rightarrow '1'$

   $En\_reg\_image \rightarrow '1'$

   $En\_reg\_kc \rightarrow '0'$

   $En\_window \rightarrow '0'$

   $En\_out\_kc \rightarrow '1'$

   $Count\_en \rightarrow '1'$

   $data\_valid\_out \rightarrow '0'$

   $last\_elaboration \rightarrow '0'$

   $clear\_count \rightarrow '0'$

   Si rimane nello stato $S2$ fin tanto che il conteggio non raggiunge il valore 130.
   Se il conteggio è pari a 130 si passa allo stato $S3$.

4. $S3$, viene generata la prima finestra utile da processare, dunque gli elementi che costituiscono tale finestra vengono trasmessi al modulo convolutore per iniziare la fase di elaborazione. In contemporanea, si continua a leggere nuovi pixels da processare. Di seguito la configurazione dei segnali di uscita:

   $Read\_en \rightarrow '1'$

$En\_reg\_image \rightarrow '1'$

$En\_reg\_kc \rightarrow '0'$

$En\_window \rightarrow '1'$

$En\_out\_kc \rightarrow '1'$

$Count\_en \rightarrow '1'$

$data\_valid\_out \rightarrow '0'$

$last\_elaboration \rightarrow '0'$

$clear\_count \rightarrow '0'$

Si rimane nello stato $S3$ fin tanto che il conteggio non raggiunge il valore 133.
Se il conteggio è pari a 133 si passa allo stato $S4$.

5. $S4$, trascorsi 3 cicli di clock dall'istante in cui a inizio l'elaborazione, per la presenza di tre stadi di pipeline all'interno del modulo convolutore, viene generato il primo risultato valido. Da questo momento in poi, verrà generato un risultato valido ad ogni ciclo di clock. Di seguito la configurazione dei segnali di uscita:

   $Read\_en \rightarrow '1'$

   $En\_reg\_image \rightarrow '1'$

   $En\_reg\_kc \rightarrow '0'$

   $En\_window \rightarrow '1'$

   $En\_out\_kc \rightarrow '1'$

   $Count\_en \rightarrow '1'$

   $data\_valid\_out \rightarrow '1'$

   $last\_elaboration \rightarrow '0'$

   $clear\_count \rightarrow '0'$

   Si rimane nello stato $S4$ fin tanto che il conteggio non raggiunge il valore 4095.
   Se il conteggio è pari a 4095 si passa allo stato $S5$.

6. $S5$, termina la lettura dei pixels e, da questo istante in poi, verranno memorizzati solo valori nulli, necessari alla corretta elaborazione dei pixels restanti. Di seguito la configurazione dei segnali di uscita:

   $Read\_en \rightarrow '0'$

   $En\_reg\_image \rightarrow '1'$

   $En\_reg\_kc \rightarrow '0'$

   $En\_window \rightarrow '1'$

$En\_out\_kc \rightarrow '1'$

$Count\_en \rightarrow '1'$

$data\_valid\_out \rightarrow '1'$

$last\_elaboration \rightarrow '0'$

$clear\_count \rightarrow '0'$

Si rimane nello stato $S5$ fin tanto che il conteggio non raggiunge il valore 4226.
Se il conteggio è pari a 4226 si passa allo stato $S6$.

7. $S6$, il circuito di bufferizzazione viene disabilitato e ha inizio l'elaborazione dell'ultimo pixel. Di seguito la configurazione dei segnali di uscita:

   $Read\_en \rightarrow '0'$

   $En\_reg\_image \rightarrow '0'$

   $En\_reg\_kc \rightarrow '0'$

   $En\_window \rightarrow '0'$

   $En\_out\_kc \rightarrow '1'$

   $Count\_en \rightarrow '1'$

   $data\_valid\_out \rightarrow '1'$

   $last\_elaboration \rightarrow '1'$

   $clear\_count \rightarrow '0'$

   Si rimane nello stato $S6$ fin tanto che il conteggio non raggiunge il valore 4229, condizione necessaria per ottenere il risultato dell'ultima elaborazione.
   Se il conteggio è pari a 4229 o se $global\_rst = '1'$ si passa nuovamente allo stato $IDLE$.

In tutti gli stati, se $global\_rst = '1'$, si ritorna allo stato $IDLE$. Viene, inoltre, definito uno stato di default, durante il quale il circuito viene inizializzato, per garantire che non ci siano stati indefiniti. Le transizioni da uno stato all'altro sono riassunte in **fig.3.15**.

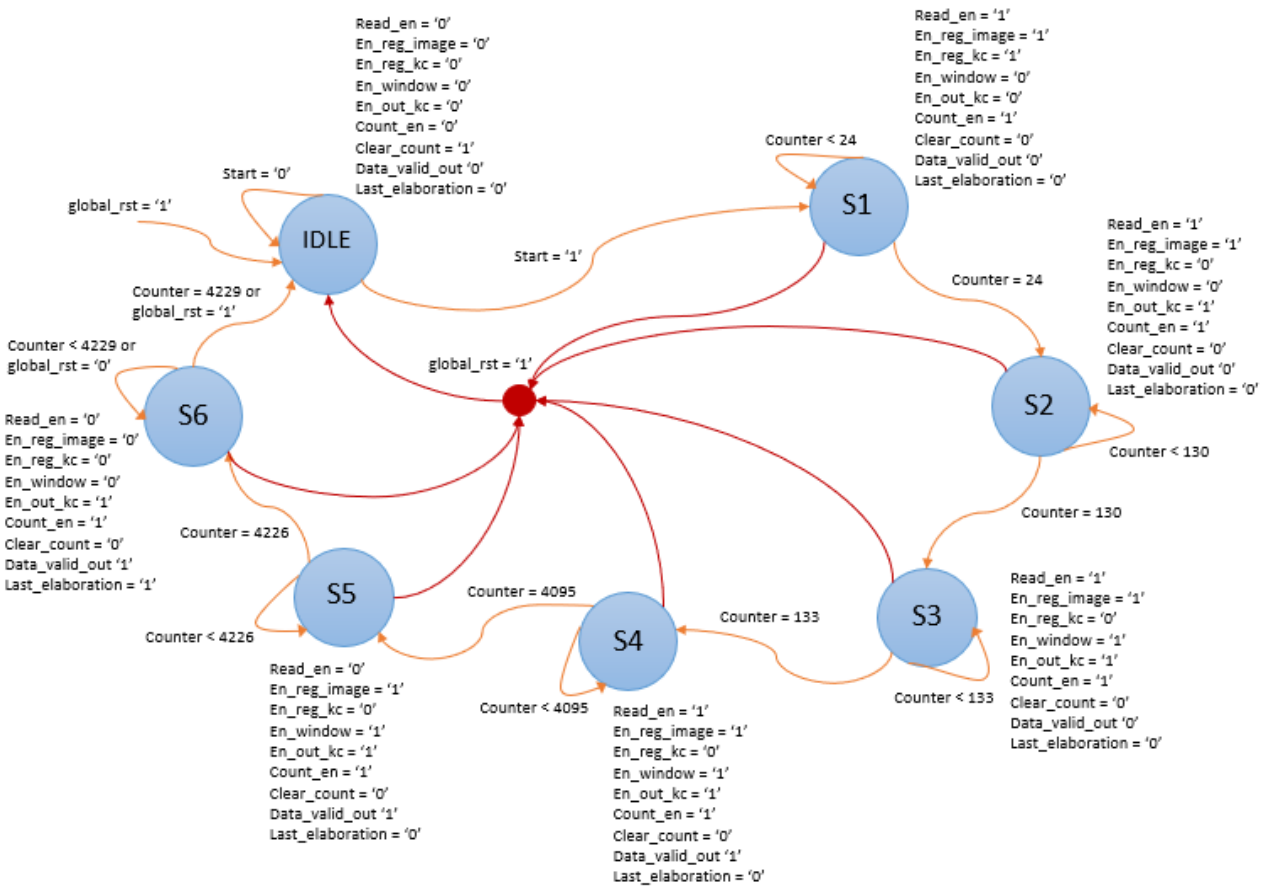


Figura 3.15 - Diagramma degli stati

- $Contatore\ sincrono$, si occupa di gestire le transizioni da uno stato all'altro della FSM. Si tratta di un contatore 13 bit, necessari affinché il conteggio avvenga nel range di valori desiderati, costituito da una cascata di flip-flop di tipo T (toggle) dotati dei segnali di $clear$ ed $enable$.
  Il funzionamento del flip-flop T è molto semplice: sul fronte sensibile del clock la sua uscita Q commuta se l'ingresso T è alto e rimane invariata se T è basso.
  In un contatore sincrono il clock alimenta simultaneamente tutti i flip-flop e il segnale di toggle, posto pari a 1, raggiunge solo il primo flip-flop (ciò significa che la sua uscita commuta sempre), mentre l'uscita del flip-flop i-esimo, successivo al primo, commuta solo se tutte le uscite dei flip-flop precedenti sono pari a 1. Questo si effettua collegando sul toggle dell'i-esimo flip-flop la AND delle uscite dei flip-flop precedenti (ogni toggle viene determinato in maniera indipendente). Tuttavia, questo va bene finché il contatore è piccolo, ma per contatori sufficientemente grandi, come in questo caso, man mano che ci si sposta verso la posizione più significativa aumenta il fan-in delle porte AND, con conseguente aumento delle dimensioni e della potenza dissipata. La soluzione adottata è stata quella di concatenare le porte AND creando un percorso a propagazione (vedi **fig.3.16**), anche se ciò comporta degli svantaggi in termini di velocità (la catena di propagazione rappresenta il path critico del contatore che imposta la frequenza massima di clock dello stesso).

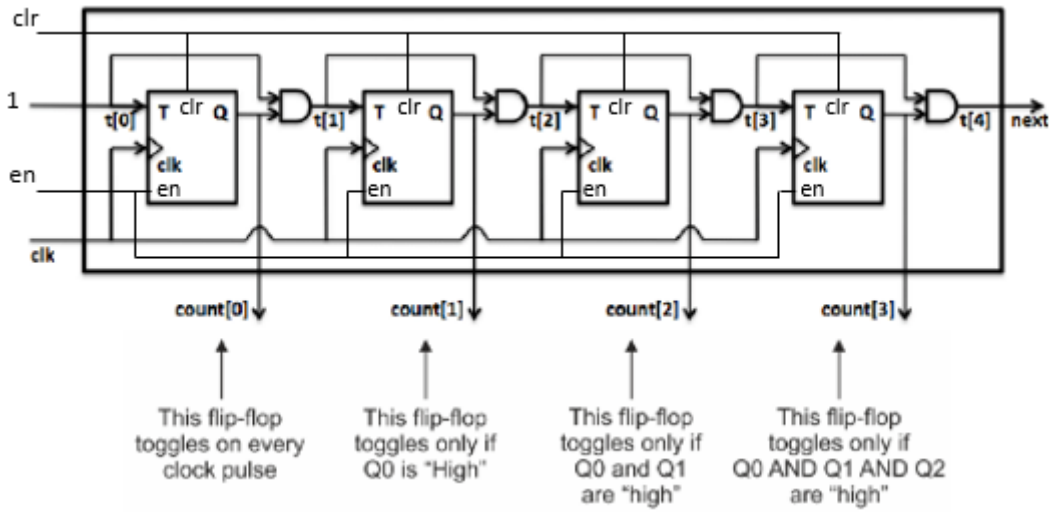


Figura 3.16 - Contatore sincrono

4 Simulazione e analisi pre-implementazione

Prima di procedere alla fase di implementazione su FPGA del circuito di filtraggio, quest'ultimo viene testato attraverso una simulazione a livello behavioral per verificarne il corretto funzionamento in un opportuno intervallo di tempo e rilevare più agevolmente eventuali errori di progettazione.
Il test viene eseguito, attraverso un testbench opportunamente scritto, con ingressi che siano verosimili a quelli reali che il circuito dovrà elaborare nel suo ciclo di vita, ovvero, con immagini in scala di grigi a 8 bit unsigned di dimensioni 64×64 e kernel isotropici 5×5 a 8 bit signed in complemento a due di tipo Laplaciano e Laplaciano-Gaussiano (vedi **fig.4.1**), atti al rilevamento dei bordi, ovvero, rilevano i rapidi cambiamenti nell'intensità della luce che spesso si verificano ai bordi (discontinuità dei livelli di grigio). In particolare, il filtro Laplaciano-Gaussiano rappresenta una variazione del filtro Laplaciano che ha lo scopo di contrastare la sensibilità al rumore di quest'ultimo implementando lo smussamento dell'immagine mediante una sfocatura gaussiana.
Le discontinuità sono indicate da pixels bianchi, mentre le regioni simili sono mostrate in nero.
L'immagine d'interesse ed i coefficienti del kernel, convertiti in file di testo ASCII, vengono applicati come vettore all'interfaccia del circuito di filtraggio. I risultati dell'elaborazione hardware vengono, a loro volta, scritti all'interno di un file di testo e confrontati con i risultati ottenuti mediante algoritmi, implementati nell'ambiente di sviluppo MATLAB, che consentano sia di manipolare i risultati dell'elaborazione hardware, sia emulare l'operazione di convoluzione ad alto livello del circuito di filtraggio.
Per l'analisi dei risultati ottenuti dall'elaborazione sia hardware che software consultare la **sez.7**.
Le **fig.4.2-4.6** illustrano il comportamento del circuito di filtraggio durante l'elaborazione, dalla lettura dei dati fino alla generazione dei risultati.

Filtro Laplaciano 1

0	0	0	0	0
0	-1	-1	-1	0
0	-1	8	-1	0
0	-1	-1	-1	0
0	0	0	0	0

Filtro Laplaciano 2

-1	-1	1	-1	-1
-1	-1	1	-1	-1
1	1	8	1	1
-1	-1	1	-1	-1
-1	-1	1	-1	-1

Filtro Laplaciano 3

-1	-1	-1	-1	-1
-1	-1	-1	-1	-1
-1	-1	24	-1	-1
-1	-1	-1	-1	-1
-1	-1	-1	-1	-1

Filtro Laplaciano-Gaussiano 1

0	0	-1	0	0
0	-1	-2	-1	0
-1	-2	16	-2	-1
0	-1	-2	-1	0
0	0	-1	0	0

Filtro Laplaciano-Gaussiano 2

0	0	1	0	0
0	1	2	1	0
1	2	-16	2	1
0	1	2	1	0
0	0	1	0	0

Figura 4.1 - Filtri isotropici Laplaciano e Laplaciano-Gaussiano

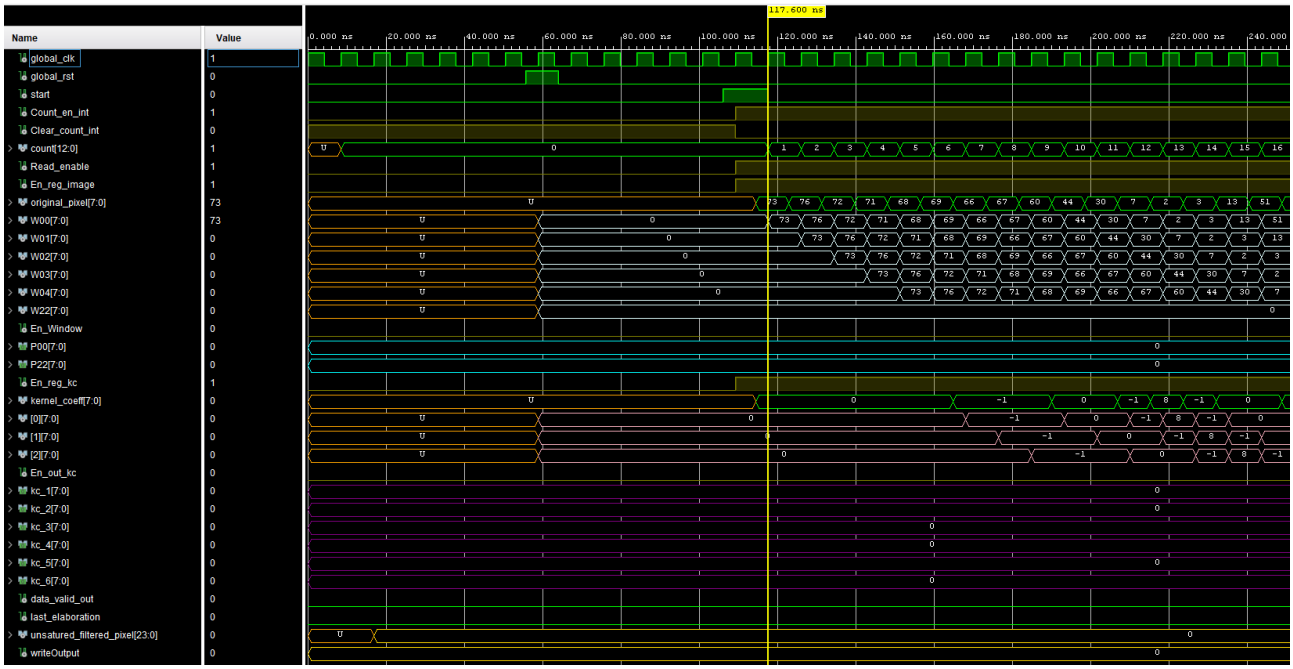


Figura 4.2 - Primo estratto simulazione behavioral

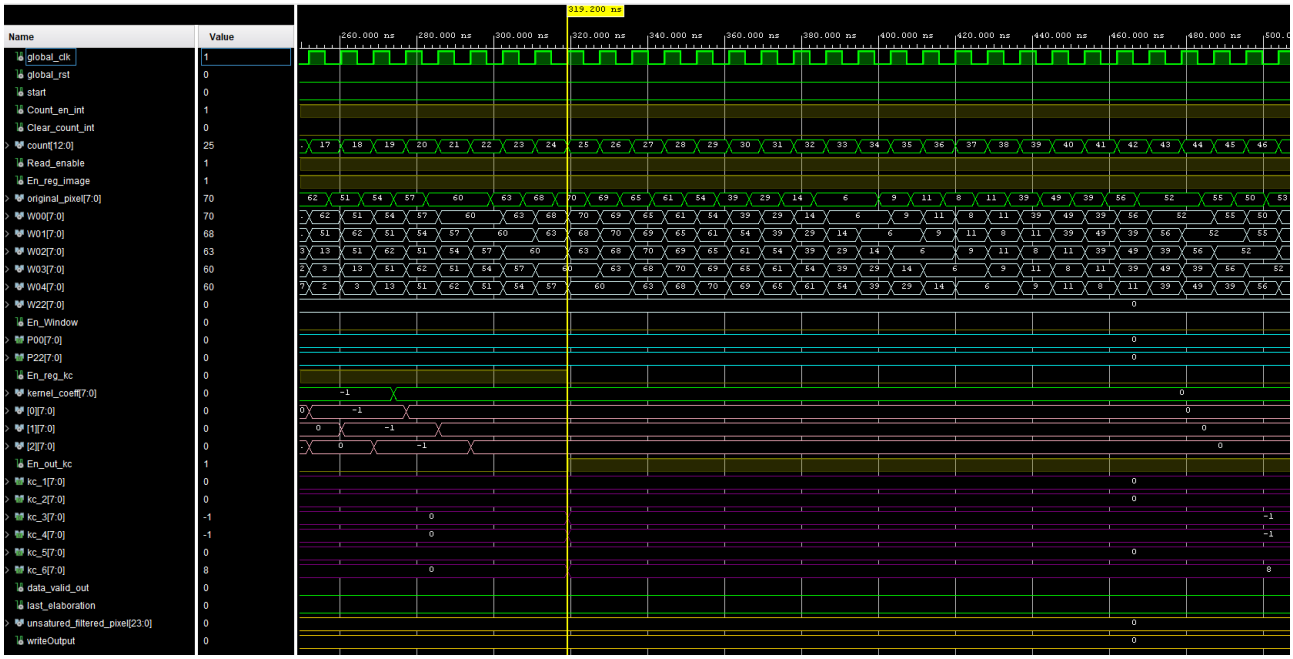


Figura 4.3 - Secondo estratto simulazione behavioral

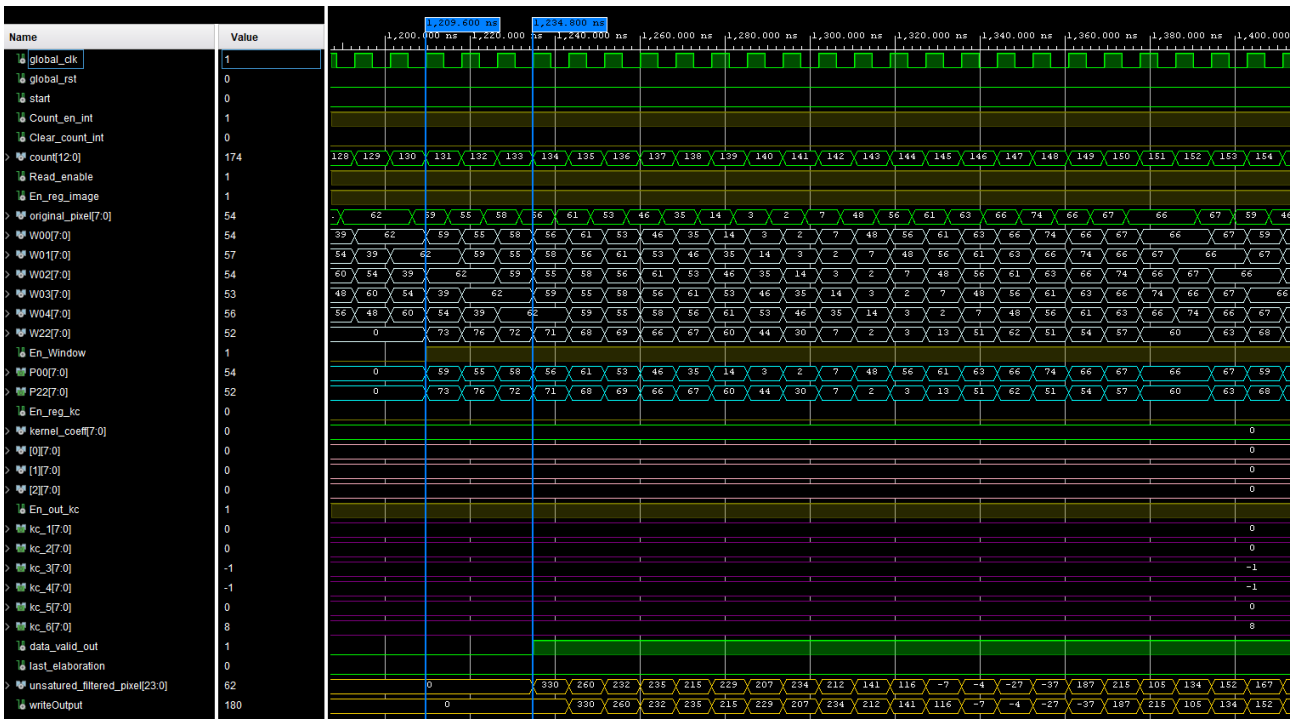


Figura 4.4 - Terzo estratto simulazione behavioral

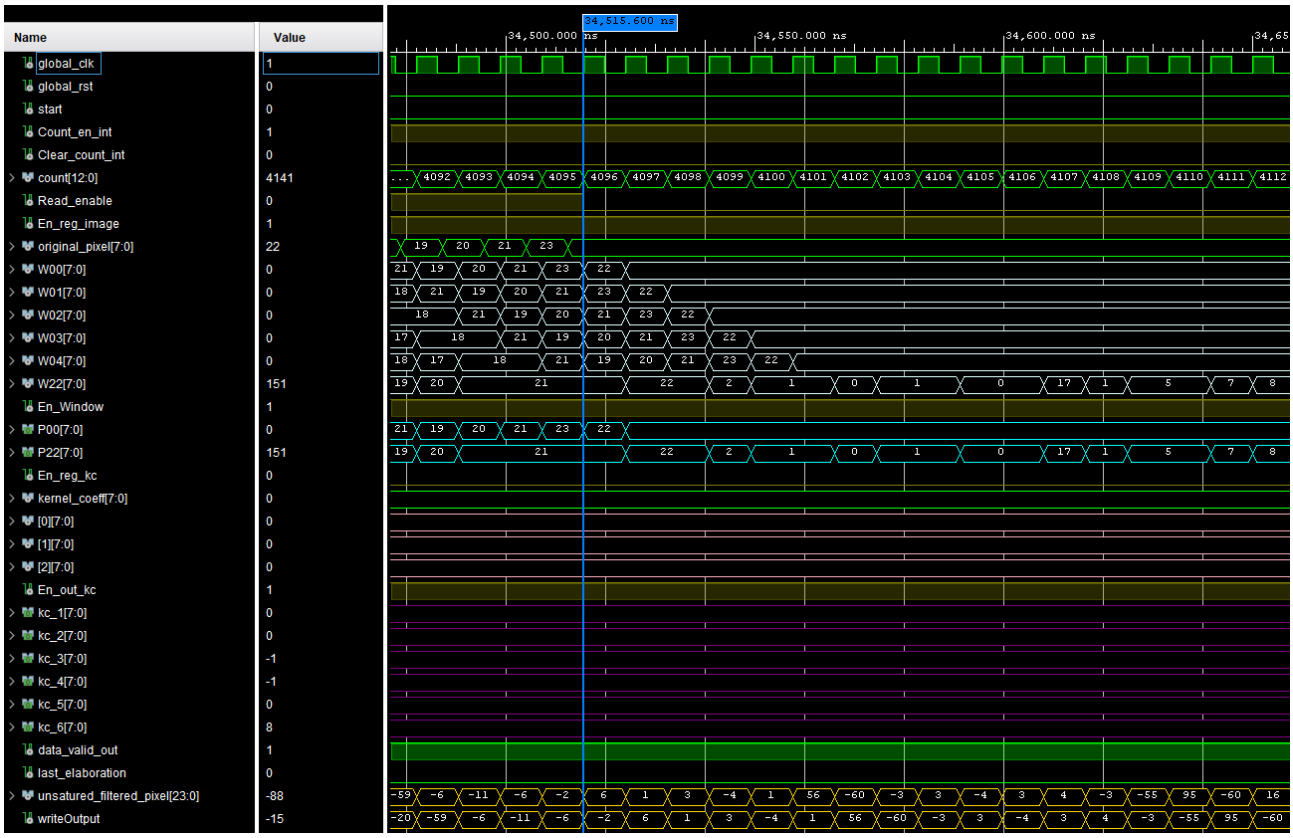


Figura 4.5 - Quarto estratto simulazione behavioral

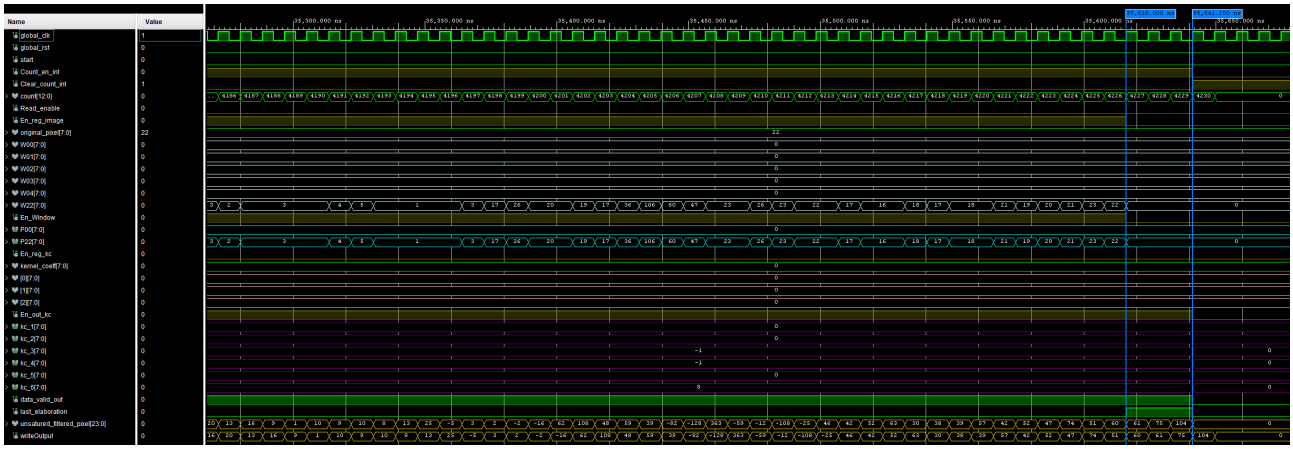


Figura 4.6 - Quinto estratto simulazione behavioral

5 Implementazione

Una volta testato il circuito a livello behavioral, si procede alla fase di sintesi e implementazione, per cui il codice a basso livello viene sintetizzato in una netlist che descrive i componenti di base e le interconnessioni che verranno, successivamente, implementati sul chip FPGA. Il chip scelto sull'ambiente di sviluppo Vivado è lo $Zynq\ XC7Z020 - 1CLG400C$, di cui è dotata la piattaforma prototipale $Pynq - Z2$. È il sintetizzatore che si occupa del piazzamento del circuito logico in una

specifica area del chip in modo da ottimizzare le interconnessioni. La **fig.5.1** illustra il circuito di filtraggio implementato.

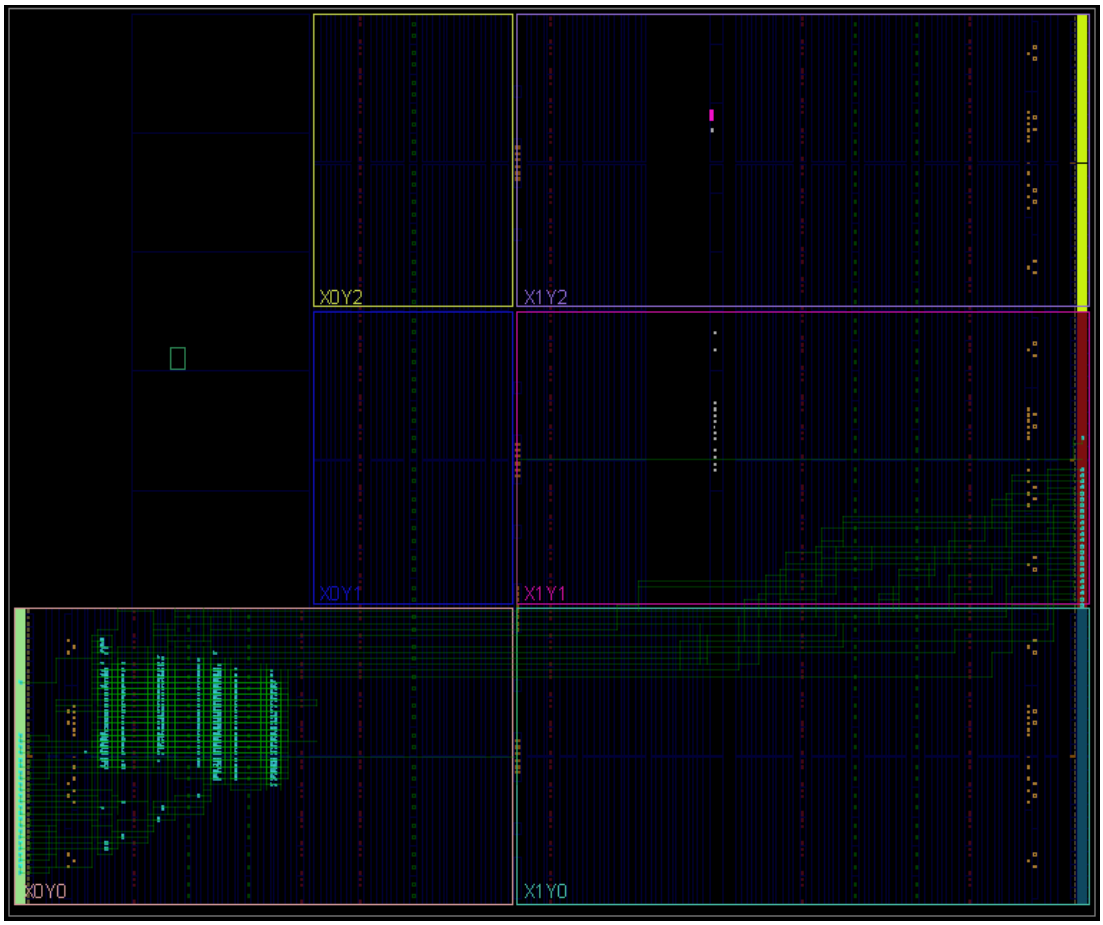


Figura 5.1 - Circuito di filtraggio post-implementazione

In merito all'utilizzo delle risorse, si può effettuare un'analisi sulla base del numero di LUT e di FF utilizzati post-implementazione. Tali informazioni sono riportate nella **fig.5.2**.

1. Slice Logic

Site Type	Used	Fixed	Prohibited	Available	Util%
Slice LUTs	1327	0	0	53200	2.49
LUT as Logic	1239	0	0	53200	2.33
LUT as Memory	88	0	0	17400	0.51
LUT as Distributed RAM	0	0			
LUT as Shift Register	88	0			
Slice Registers	592	0	0	106400	0.56
Register as Flip Flop	592	0	0	106400	0.56
Register as Latch	0	0	0	106400	0.00
F7 Muxes	0	0	0	26600	0.00
F8 Muxes	0	0	0	13300	0.00

Name	Slice LUTs (53200)	Slice Registers (106400)	Slice (13300)	LUT as Logic (53200)	LUT as Memory (17400)
Spatial_Filtering_Circuit	1327	592	387	1239	88
Controller_module (ControllerModule)	71	20	47	71	0
Counter (CounterSync_generic)	21	13	7	21	0
Finite_state_machine (FSM)	50	7	41	50	0
Convolution_module (ConvolutionModule)	569	197	257	569	0
First_pipeline_module (Pipeline_reg_ADDERtoMU	438	60	205	438	0
Second_pipeline_module (Pipeline_reg_MULTtoA	131	113	110	131	0
Third_pipeline_reg (Final_pipeline_reg)	0	24	14	0	0
Image_Cache_memory (ImageCacheMemory)	299	291	128	235	64
Fifth_RowWindow (Shift_reg)	27	40	31	27	0
First_RowWindow (Shift_reg_0)	84	40	49	84	0
Fourth_RowWindow (Shift_reg_1)	11	40	30	11	0
Row_buffer_1 (Row_buffer)	20	67	28	4	16
Row_buffer_2 (Row_buffer_2)	20	8	9	4	16
Row_buffer_3 (Row_buffer_3)	20	8	11	4	16
Row_buffer_4 (Row_buffer_4)	20	8	12	4	16
Second_RowWindow (Shift_reg_5)	31	40	32	31	0
Third_RowWindow (Shift_reg_6)	66	40	43	66	0
Isotropic_Kernel_memory (IsotropicKernelMemory)	391	84	176	367	24

Figura 5.2 - Report utilization

6 Simulazione e analisi post-implementazione

La simulazione post-implementazione, illustrata in **fig.6.1**, consente di verificare il corretto funzionamento del circuito una volta implementato su chip. L'unica differenza rispetto alla simulazione behavioral è che, in questo caso, entrano in gioco i ritardi associati alle interconnessioni, per cui le commutazioni dei segnali avvengono con un certo ritardo rispetto al fronte sensibile del clock.

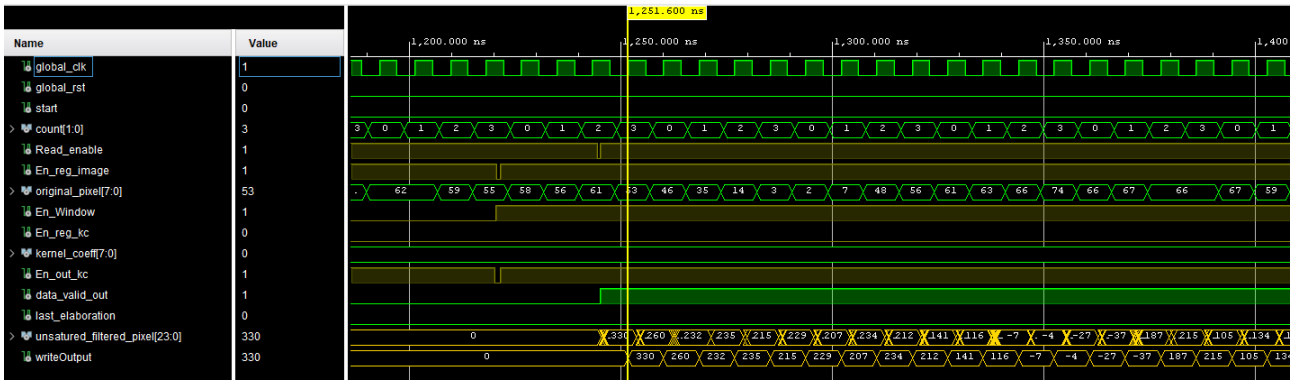


Figura 6.1 – Primo estratto simulazione post-implementazione

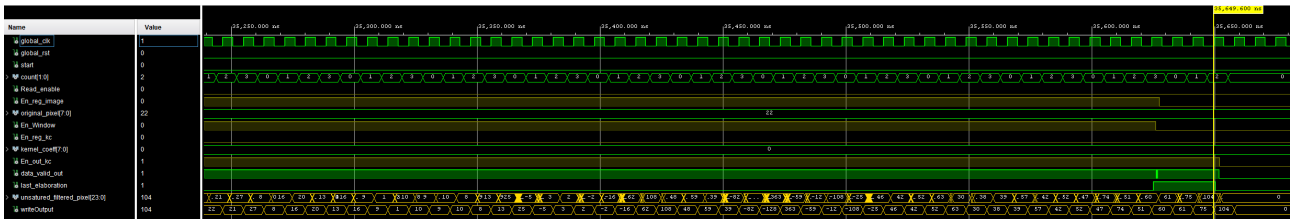


Figura 6.2 - Secondo estratto simulazione post-implementazione

Inoltre, le prestazioni del circuito, in termini di latenza, throughput, frequenza massima e dissipazione di potenza, sono state testate per differenti constraint di clock e si è arrivati alla conclusione che il periodo di clock nel quale rientra il path critico e per il quale è garantito il corretto funzionamento del circuito è pari a $8.4\ ns$, col quale vengono ottenuti i risultati di timing riportati in **fig.6.3**.

Design Timing Summary

Setup		Hold		Pulse Width	
Worst Negative Slack (WNS):	0,169 ns	Worst Hold Slack (WHS):	0,087 ns	Worst Pulse Width Slack (WPWS):	3,220 ns
Total Negative Slack (TNS):	0,000 ns	Total Hold Slack (THS):	0,000 ns	Total Pulse Width Negative Slack (TPWS):	0,000 ns
Number of Failing Endpoints:	0	Number of Failing Endpoints:	0	Number of Failing Endpoints:	0
Total Number of Endpoints:	1153	Total Number of Endpoints:	1153	Total Number of Endpoints:	681

```
Max Delay Paths
--------------------------------------------------------------------------------------
Slack (MET) :             0.169ns  (required time - arrival time)
  Source:                 Controller_module/Finite_state_machine/FSM_onehot_present_state_reg[3]/C
                            (rising edge-triggered cell FDRE clocked by global_clk  {rise@0.000ns fall@4.200ns period=8.400ns})
  Destination:            Convolution_module/Second_pipeline_module/Out_reg1_reg[18]/D
                            (rising edge-triggered cell FDRE clocked by global_clk  {rise@0.000ns fall@4.200ns period=8.400ns})
  Path Group:             global_clk
  Path Type:              Setup (Max at Slow Process Corner)
  Requirement:            8.400ns  (global_clk rise@8.400ns - global_clk rise@0.000ns)
  Data Path Delay:        8.222ns  (logic 1.820ns (22.135%)  route 6.402ns (77.865%))
  Logic Levels:           11  (LUT3=3 LUT6=8)
  Clock Path Skew:        -0.050ns (DCD - SCD + CPR)
    Destination Clock Delay (DCD):    4.615ns = ( 13.015 - 8.400 )
    Source Clock Delay      (SCD):    5.124ns
    Clock Pessimism Removal (CPR):    0.458ns
  Clock Uncertainty:      0.035ns  ((TSJ^2 + TIJ^2)^1/2 + DJ) / 2 + PE
    Total System Jitter     (TSJ):    0.071ns
    Total Input Jitter      (TIJ):    0.000ns
    Discrete Jitter          (DJ):    0.000ns
    Phase Error              (PE):    0.000ns
```

Figura 6.3 - Report timing

Sulle considerazioni fatte, la frequenza massima di funzionamento è pari a:

$$f_{max} = \frac{1}{\tau_{constr}} = \frac{1}{8.4\ ns} = 119.048\ MHz$$

In merito alla latenza, intesa come il tempo necessario affinché vengano prodotto un risultato valido in uscita dal momento in cui il circuito inizia ad elaborare i dati, corrispondente al momento in cui i pixel iniziano ad essere letti, questa risulta essere pari a 134 cicli di clock, di cui 131 cicli di clock rappresentano il tempo necessario ad ottenere la finestra di convoluzione valida da processare e i restanti 3 cicli di clock, dovuti agli stadi di pipeline, rappresentano il tempo necessario che deve trascorrere, dal momento in cui il circuito inizia a processare la finestra, prima di ottenere il pixel

filtrato. Per tener conto anche dei ritardi associati alle piste di interconnessione, viene calcolata una latenza complessiva pari a 135 cicli di clock.

Superata la latenza del circuito, viene generato un risultato valido ad ogni ciclo di clock, pertanto il circuito risulta essere caratterizzato da un throughput unitario.

A questo punto, per determinare la potenza media dissipata dal circuito, in fase di simulazione post-implementazione viene generato il cosiddetto file $.saif$ per guidare l'algoritmo di analisi della potenza tenendo conto dell'attività di switching effettiva dei segnali, ovvero, dei nodi interni del circuito, ottenendo un alto livello di confidenza. In caso contrario, il tool di sviluppo fornirà un valore di potenza media dissipata sulla base di una stima dell'attività di switching dei segnali. La **fig.6.4** riporta le informazioni relative al consumo di potenza.

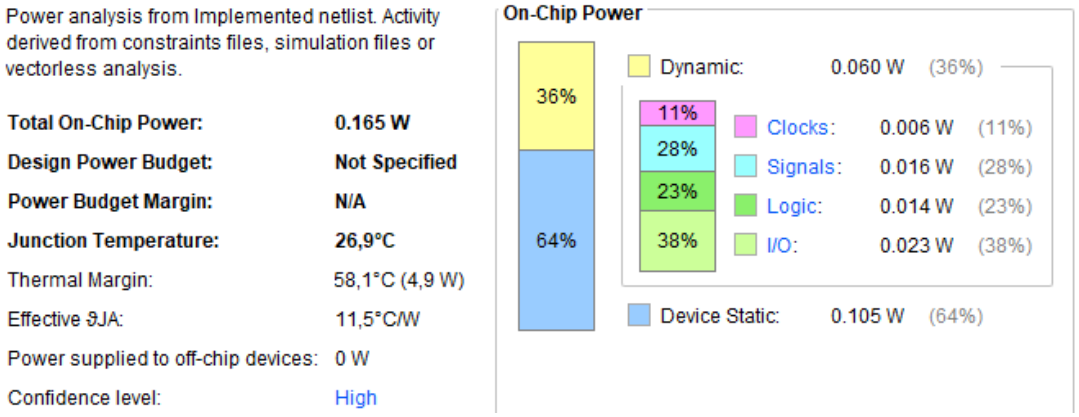


Figura 6.4 - Report power

Viene eseguita un'analisi più dettagliata sul consumo di potenza dinamico, poiché rappresenta il contributo sul quale è possibile intervenire, agendo sull'architettura del circuito, per cercare di ridurre il consumo di potenza complessivo. Pertanto, vengono estrapolati i dati relativi alla dissipazione di potenza dinamica da parte del clock, dei segnali, della logica e degli I/O:

	Dynamic power dissipation [W]	Total dynamic power [mW]
Clock	0.00628282	59.561
Signals	0.01645911	
logic	0.01357104	
I/O	0.02324796	

L'energia mediamente assorbita per singola operazione è ottenuta come:

$$E = P_{dyn} \cdot \tau_{constr} = 59.561\ [mW] \cdot 8.4[ns] = 500.312\ [pJ]$$

7 Risultati

I risultati ottenuti dall'elaborazione hardware, mediante il circuito di filtraggio progettato in VHDL, vengono confrontati con un'implementazione della procedura di filtraggio ad alto livello utilizzando MATLAB, per la quale è richiesto un tempo di calcolo pari a $23.918\ ms$, di gran lunga maggiore da quello richiesto dal circuito logico. Nelle **fig.7.1-7.2** vengono illustrati i risultati ottenuti da entrambe le procedure con l'impiego di differenti tipologie di filtri isotropici.

Figura 7.1 - Immagine filtrata a basso livello tramite hardware specializzato con differenti tipologie di filtri isotropici

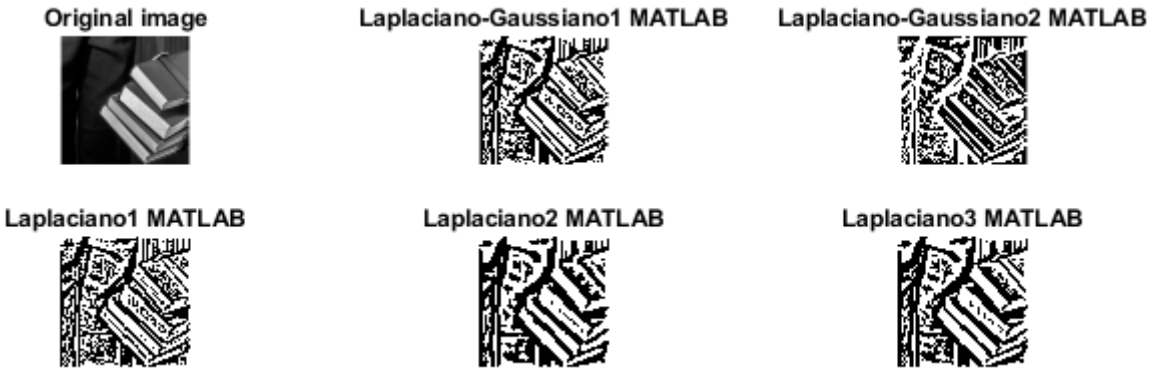


Figura 7.2 - Immagine filtrata ad alto livello tramite software specializzato con differenti tipologie di filtri isotropici

Per rendere quanto più veritiero possibile il processo di filtraggio si è deciso di attuare una saturazione ad alto livello sui risultati di entrambe le procedure, come illustrato nelle **fig.7.3-7.4**.

Figura 7.3 - Saturazione su filtraggio hardware

Figura 7.4 - Saturazione su filtraggio software

A questo punto si procede col confrontare i risultati attraverso un'analisi dell'errore tra i pixel filtrati con procedura a basso livello e i pixel filtrati con procedura ad alto livello. Vengono adottate diverse metriche di confronto, in grado di prevedere la qualità percettiva delle immagini in modo coerente con la valutazione soggettiva umana:

- $PSNR$ ($Peak\ Signal\ to\ Noise\ Ratio$), deriva dall'errore quadratico medio e indica il rapporto tra l'intensità massima dei pixels e la potenza del rumore distorcente. Questo rapporto tra due immagini a confronto viene espresso in decibel e varia tra 30 e $50\ dB$ per la rappresentazione di dati a 8 bit. Maggiore è il valore del $PSNR$, migliore sarà la somiglianza tra le due immagini a confronto. Il $PSNR$ è espresso come:

$$PSNR = 10 \log_{10} \frac{peakval^2}{MSE}$$

  dove $peakval$ (valore di picco) rappresenta il massimo valore d'intensità dei dati dell'immagine e, se questi sono dati unsigned a 8 bit, il valore di picco è pari a 255. MSE rappresenta l'errore quadratico medio. Più basso è questo parametro, minore è l'errore tra le due immagini a confronto. L'MSE viene calcolato utilizzando la seguente relazione:

$$MSE = \frac{1}{M \cdot N} \sum_{i=0}^{N-1} \sum_{j=o}^{M-1} [X(i,j) - Y(i,j)]^2$$

- $SSIM$ ($Structural\ Similarity\ Index\ Measure$), è una metrica basata sull'analisi strutturale, ovvero, non si limita ad una differenza pixel per pixel, ma misura effettivamente la similarità tra due immagini a confronto sulla base di tre aspetti principali: luminanza, contrasto e struttura dell'immagine. $SSIM$ viene applicato localmente pixel per pixel, generando una cosiddetta mappa di qualità locale, ed è espresso dalla seguente relazione:

$$SSIM(i,j) = \frac{(2\mu_i\mu_j + c_1)(2\sigma_i\sigma_j + c_2)}{(\mu_i^2 + \mu_j^2 + c_1)(\sigma_i^2 + \sigma_j^2 + c_2)}$$

  Eseguendo la media di tutti i valori $SSIM$ locali, si determina il valore $SSIM$ globale:

$$MSSIM = \frac{1}{M}\sum_{k=0}^{M} SSIM_k$$

  Bisogna precisare che, la somiglianza strutturale è in scala normalizzata (valori compresi tra -1 e 1).

- $DSSIM$ ($Structural\ Dissimilarity$), deriva dalla somiglianza strutturale ed è espressa come:

$$DSSIM(i,j) = \frac{1 - SSIM(i,j)}{2}$$

  Noto il valore $SSIM$ globale, il valore $DSSIM$ globale è ottenuto come:

$$MDSSIM = \frac{1 - MSSIM}{2}$$

  Per un valore $MSSIM$ che tende ad 1, $MDSSIM$ tende a 0. Ciò si ottiene nel caso in cui le due immagini a confronto sono tra loro perfettamente identiche.

- $GMSD$ ($Gradient\ Magnitude\ Similarity\ Deviation$), misura la similarità tra due immagini confrontando le ampiezze del gradiente pixel per pixel. Il calcolo della deviazione standard dei valori $GMSD$ locali permette di determinare il valore $GMSD$ complessivo. Senza impiegare informazioni aggiuntive, l'utilizzo della sola ampiezza del gradiente può comunque produrre una stima della qualità abbastanza accurata. Si noti che il valore $GMSD$ complessivo è una misura della distorsione presente in un'immagine rispetto a quella di riferimento. Più alto è questo valore, maggiore è la distorsione e quindi minore è la qualità percettiva dell'immagine. Maggiori informazioni sul calcolo del gradiente delle immagini sono consultabili nel seguente articolo 10.1109/TIP.2013.2293423.

L'analisi viene eseguita su una singola tipologia di filtro, in particolare, sulla base del filtro Laplaciano-Gaussiano di tipo 1 (vedi **fig.7.5**), tuttavia, l'analisi può essere estesa, in maniera del tutto analoga, alle altre tipologie di filtro.

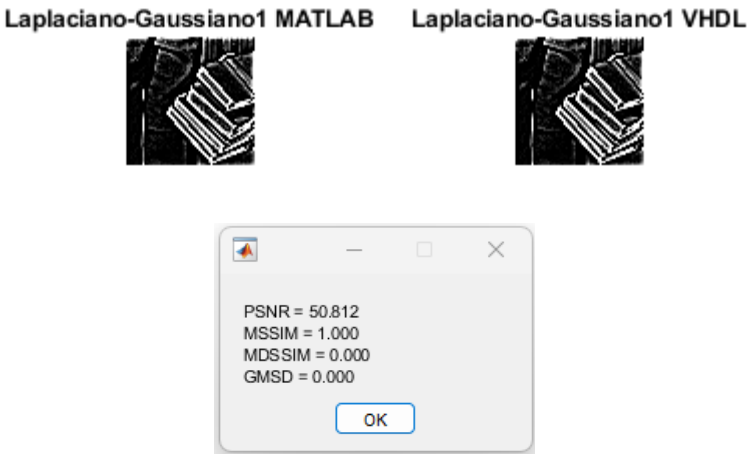


Figura 7.5 – Laplaciano-Gaussiano tipo 1: confronto tra implementazione MATLAB e VHDL

Queste metriche sono utili anche per confrontare l'immagine originale con l'immagine filtrata, in modo da evidenziare gli effetti, in termini di qualità percettiva, del filtro adottato. Dai risultati di **fig.7.6**, associati al filtro Laplaciano-Gaussiano di tipo 1, si osserva come i valori delle metriche di confronto siano relativamente bassi, da ciò si evince come un'enfatizzazione pronunciata dei bordi, attraverso l'impiego di un filtro Laplaciano o Laplaciano-Gaussiano, comporta una degradazione della qualità dell'immagine e quindi un aumento della distorsione. Questa tesi è rafforzata dai risultati illustrati in **fig.7.7** associati al filtro Laplaciano-Gaussiano di tipo 2, il quale garantisce una maggiore enfatizzazione dei bordi e, conseguentemente, una maggiore degradazione dell'immagine stessa a cui corrisponde un'elevata dissomiglianza strutturale.

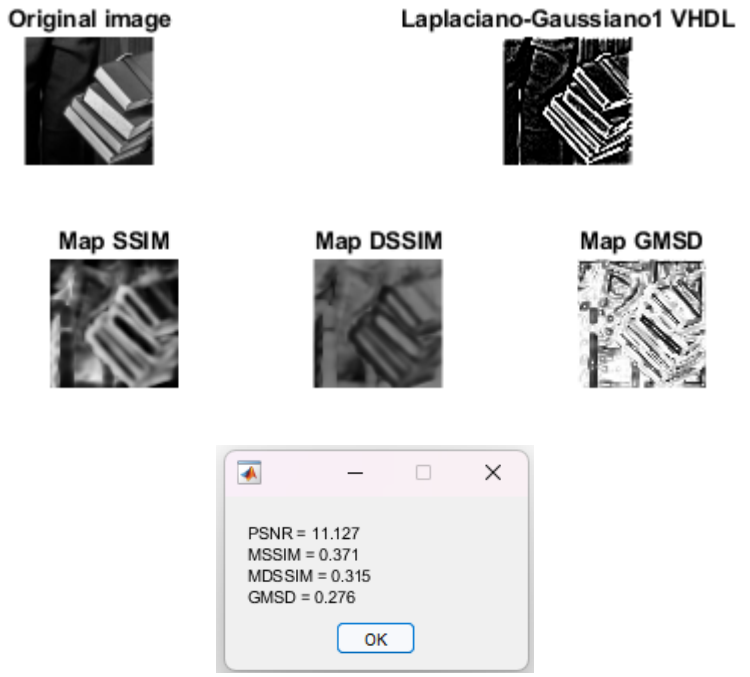


Figura 7.6 – Laplaciano-Gaussiano tipo 1: confronto tra immagine originale e immagine filtrata

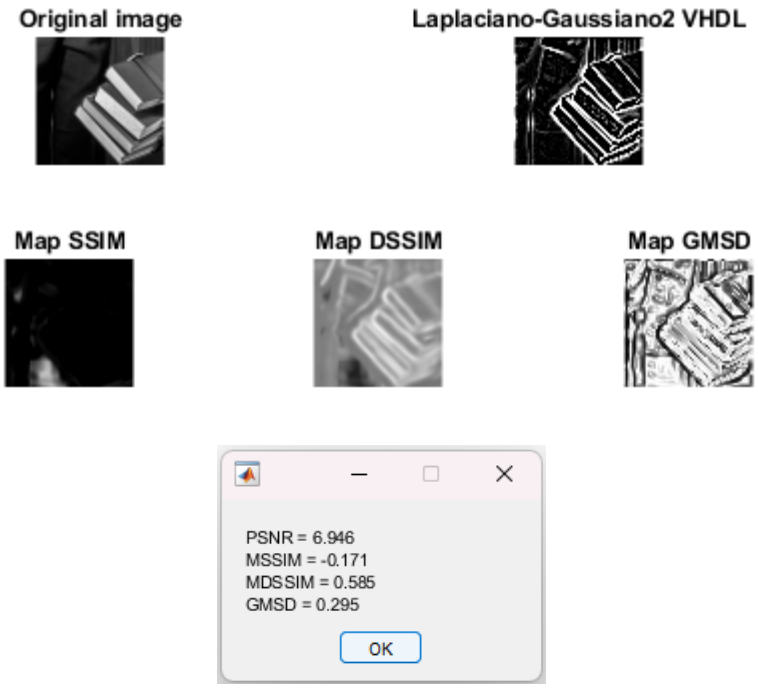


Figura 7.7 - Laplaciano-Gaussiano tipo 2: confronto tra immagine originale e immagine filtrata

8 Conclusioni

Dalle analisi eseguite si evince come l'implementazione di un circuito di filtraggio spaziale su FPGA può rappresentare un'ottima alternativa alle soluzioni tradizionali basate esclusivamente su software. In generale, in un'applicazione reale, l'approccio congiunto tra hardware e software, sfruttando la tecnologia FPGA, consente di massimizzare le prestazioni, garantendo al contempo la possibilità di personalizzare il circuito per soddisfare le esigenze specifiche dell'applicazione. Questo approccio rappresenta, dunque, una soluzione efficace per applicazioni di image processing che richiedono alta velocità e precisione.

9 Miglioramenti futuri

L'implementazione del filtro spaziale su FPGA ha mostrato risultati promettenti, ma esistono diverse aree in cui il progetto potrebbe essere ulteriormente migliorato:

1. Implementare un certo grado di parallelizzazione delle operazioni di convoluzione, modificando opportunamente l'architettura, in modo da elaborare più pixels contemporaneamente. Questo consentirebbe di ridurre le operazioni di somma, individuando quelle somme parziali comuni tra i pixels da elaborare, che verrebbero pre-

calcolate e richiamate ogni qualvolta siano richieste, e di aumentare ulteriormente il throughput del circuito, con conseguente riduzione del ritardo complessivo.

2. Utilizzare kernel di dimensioni maggiori (7×7 o 9×9) potrebbe migliorare la capacità del filtro di rilevare dettagli più fini, sebbene ciò richieda un maggior carico computazionale.
3. Modificare l'architettura per supportare immagini a colori, impiegando filtri separati per ciascuna componente di colore. Questo miglioramento permetterebbe di applicare il filtraggio spaziale a un'ampia gamma di applicazione, come l'elaborazione video e il riconoscimento degli oggetti a colori.
4. Modificare l'architettura per gestire immagini con risoluzioni superiori a 64×64 pixels.
5. Sostituzione del blocco RCA relativo al modulo adder-tree finale, con un sommatore Carry Select o Parallel-prefix per migliorare le prestazioni di calcolo. Il beneficio che si ottiene in termini di velocità deve essere tale da poter compensare le perdite in termini di potenza, risorse impiegate e costo.
6. Sviluppare strategie di gestione della potenza per ridurre il consumo energetico durante il processo di elaborazione. Questo potrebbe includere, ad esempio, la scalabilità della frequenza.