

# **PROGETTO**

Progettazione di Sistemi Digitali

A.A. 2022-2023

GIORGIO UBBRIACO 247287

AMEDEO ROSARIO GRANDINETTI 242507

HARPREET SINGH 247048

SALVATORE COZZA 242509

ROCCO DEPIETRA 247139

FRANCESCO GOLLUSCIO 247713

## Sommario

0. Progetto .....	6
1. Introduzione e stato dell'arte.....	7
1.1. Image processing.....	7
1.2. Storia delle tecniche di image processing .....	7
1.3. L'image processing nel campo dell'ingegneria elettronica.....	7
1.4. Filtraggio spaziale .....	8
1.5. Filtro isotropico .....	9
1.6. Scansione raster.....	11
1.7. Problema ai bordi e tecnica di padding .....	11
1.8. Xilinx PYNQ-Z2 .....	12
2. Descrizione schema a blocchi .....	13
2.1. FSM .....	15
2.2. Contatore .....	17
2.3. BufferLine .....	18
2.4. Circuiti sommatori.....	21
2.3.1. Pre-Adder .....	21
2.3.2. Final-Adder .....	23
2.5. Booth Multiplier.....	24
2.5.1. Caratteristiche generali e scelte progettuali .....	24
2.5.2. Progettazione.....	25
3. Implementazione .....	28
4. Simulazione.....	30
4.1. Behavioral Simulation.....	30
4.2. Post-Implementation Simulation.....	32
5. Report.....	35
5.1. Report di timing.....	35
5.2. Report di utilizzazione delle risorse .....	35
5.3. Report di dissipazione di potenza senza file .saif.....	36
5.4. Report di dissipazione di potenza con file .saif.....	38
6. Testing e Validazione.....	40
6.1. Lettura dei pixel dell'immagine da filtrare.....	40
6.2. Inizializzazione del filtro isotropico e scelta dei coefficienti.....	40
6.3. Generazione della finestra di processing.....	41
6.4. Waiting dell'ouput del circuito progettato .....	42
6.5. Lettura dell'output generato dal circuito progettato.....	43

6.6.	Comparazione dei risultati ottenuti .....	43
6.7.	Analisi dell'errore.....	43
6.7.1.	Analisi tramite errore assoluto .....	43
6.7.2.	Analisi tramite PSNR.....	44
6.7.3.	Analisi tramite SSIM .....	44
6.7.4.	Analisi dei valori di PSNR, SSIM e DSSIM .....	45
7.	Conclusioni .....	47

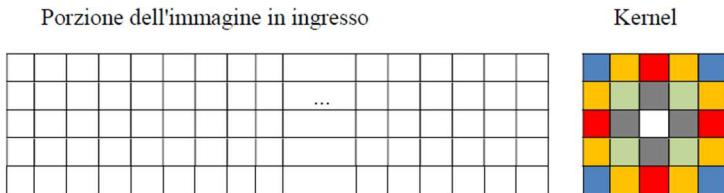
## Indice delle figure

Figura 1 Sensore di luce e disposizione dei sensori.....	7
Figura 2 immagine in scala di grigi e immagine con valore dei pixel .....	8
Figura 3 Filtraggio nel dominio dello spazio .....	8
Figura 4 Confronto tra il numero di operazioni di un filtro non isotropico e isotropico .....	9
Figura 5 Filtro isotropico .....	9
Figura 6 Coefficienti di un filtro isotropico .....	9
Figura 7 Filtro isotropico Laplaciano 2.....	10
Figura 8 Filtro isotropico Laplaciano 1.....	10
Figura 9 Filtro isotropico Mediano .....	10
Figura 10 Filtro isotropico Gaussiano.....	10
Figura 11 Filtro isotropico Laplaciano 3.....	10
Figura 12 Raster scan.....	11
Figura 13 PYNQ-Z2.....	12
Figura 14 Schema a blocchi del circuito utilizzato per filtrare un’immagine 64x64.....	13
Figura 15 Schematic RTL del circuito ottenuto in Vivado per filtrare un’immagine 64x64 pixels ..	14
Figura 16 Diagramma a stati della macchina di controllo del circuito .....	15
Figura 17 Contatore sincrono.....	18
Figura 18 Schematic RTL relativo al Flip Flop T.....	18
Figura 19 Finestra di convoluzione.....	18
Figura 20 Filtro isotropico .....	19
Figura 21 Struttura di bufferizzazione .....	20
Figura 22 Schematic RTL relativo al Pre-Adder Module.....	22
Figura 23 Schematic RTL relativo al Pre-Adder a 8 operandi a 9 bit .....	23
Figura 24 Schematic RTL relativo al Pre-Adder a 4 operandi a 9 bit .....	23
Figura 25 Schematic RTL relativo al Final-Adder .....	24
Figura 26 Tabella delle codifiche del moltiplicatore con l’algoritmo di Booth.....	24
Figura 27 Schematic del moltiplicatore di Booth .....	25
Figura 28 Tabella delle codifiche per il moltiplicatore di Booth .....	26
Figura 29 Posizione delle risorse utilizzate dal circuito progettato nell’FPGA.....	28
Figura 30 Path critico del circuito implementato.....	28
Figura 31 Primo estratto della simulazione behavioral del circuito progettato .....	30
Figura 32 Secondo estratto della simulazione behavioral del circuito progettato .....	31
Figura 33 Terzo estratto della simulazione behavioral del circuito progettato.....	31
Figura 34 Quarto estratto della simulazione behavioral del circuito progettato .....	32
Figura 35 Primo estratto della simulazione post-implementation del circuito progettato .....	33
Figura 36 Secondo estratto della simulazione post-implementation del circuito progettato .....	33
Figura 37 Terzo estratto della simulazione post-implementation del circuito progettato.....	34
Figura 38 Report sul timing ottenuto con un constraint sul clock di 7.9 ns.....	35
Figura 39 Report di utilizzo delle risorse.....	36
Figura 40 Distribuzione dell’utilizzo delle risorse nel Buffer Line .....	36
Figura 41 Report di dissipazione di potenza .....	37
Figura 42 Report di dissipazione di potenza per ogni risorsa .....	37
Figura 43 Report di dissipazione di potenza .....	37
Figura 44 Generazione del file .saif .....	38
Figura 45 Aggiunta del file .saif nel Report Power .....	38
Figura 46 Report di dissipazione di potenza considerando il file .saif .....	39

Figura 47 Scelta della tipologia di filtro isotropico da utilizzare per il testing.....	41
Figura 48 Ultime 2 finestre di processing generate .....	42
Figura 49 Waiting nella Console Window di MATLAB dell'output del circuito progettato .....	42
Figura 50 Confronto tra l'immagine filtrata tramite procedura ad alto livello e immagine filtrata tramite circuito progettato (filtro isotropico Laplaciano di tipologia 1) .....	43
Figura 51 Analisi del PSNR, SSIM e DSSIM .....	45
Figura 52 Analisi dell'errore assoluto e del DSSIM tra l'immagine originale e l'immagine filtrata..	46
Figura 54 Immagine filtrata tramite filtro isotropico Laplaciano di tipologia 2 .....	47
Figura 55 Immagine filtrata tramite filtro isotropico Laplaciano di tipologia 3 .....	47
Figura 53 Errore assoluto e dissomiglianza strutturale tra l'immagine originale e l'immagine filtrata tramite filtro isotropico Laplaciano di tipologia 2 .....	47
Figura 56 Errore assoluto e dissomiglianza strutturale tra l'immagine originale e l'immagine filtrata tramite filtro isotropico Laplaciano di tipologia 3 .....	47
Figura 59 Immagine filtrata tramite filtro isotropico mediano .....	48
Figura 60 Errore assoluto e dissomiglianza strutturale tra l'immagine originale e l'immagine filtrata tramite filtro isotropico mediano.....	48
Figura 58 Immagine filtrata tramite filtro isotropico Gaussiano .....	48
Figura 57 Errore assoluto e dissomiglianza strutturale tra l'immagine originale e l'immagine filtrata tramite filtro isotropico Gaussiano.....	48

## 0. Progetto

Progettare un circuito che sia capace di filtrare immagini greyscale, di dimensione arbitraria, a 8-bit. Il kernel da impiegare dev'essere composto da  $5 \times 5$  coefficienti a 8-bit rappresentati in complemento a due. I coefficienti dei filtri non devono essere fissati a priori.



I colori impiegati nella rappresentazione del kernel evidenziano i gruppi di coefficienti che hanno lo stesso valore.

Il circuito va progettato in modo da sfruttare il massimo livello di pipeline e l'isotropicità del kernel.

Non è richiesto che le uscite siano saturate.

Si richiede di caratterizzare il circuito progettato in termini: latenza, throughput, massima frequenza di funzionamento, occupazione di risorse e dissipazione di potenza.

Nella relazione e nella presentazione da preparare per la prova orale, oltre a fornire una descrizione dettagliata del circuito progettato e della macchina a stati impiegata per gestire il processo di calcolo, bisogna presentare e discutere i risultati ottenuti dalla caratterizzazione post-layout.

# 1. Introduzione e stato dell'arte

## 1.1. Image processing

L'elaborazione delle immagini, conosciuta anche come *image processing*, è una disciplina che, attraverso specifici programmi e dispositivi elettronici, permette la manipolazione, l'elaborazione e l'analisi di immagini digitali per ottenere informazioni e per apportare opportune modifiche. Nello specifico, il filtraggio delle immagini si pone come obiettivo l'ottimizzazione di quest'ultime sia dal punto di vista della qualità sia per quello che riguarda la riduzione di rumore o disturbi.

## 1.2. Storia delle tecniche di image processing

La maggior parte delle tecniche relative all'elaborazione digitale di immagini sono state concretizzate nel 1960 presso il *Jet Propulsion Laboratory*, il *MIT*, i *Bell Laboratories* e altre strutture di ricerca proponendosi come obiettivo lo sviluppo di applicazioni a immagini satellitari, immagini medicali e miglioramenti fotografici.

Oggi, l'*image processing* è ancora oggetto di ricerca soprattutto nel campo dell'elettronica per lo studio dell'hardware dedicato e dell'informatica per l'ottimizzazione di algoritmi sull'elaborazione di immagini.

## 1.3. L'Image processing nel campo dell'ingegneria elettronica

Un sistema elettronico per l'elaborazione delle immagini è un sistema che comprende un microprocessore, memorie di massa per conservare l'immagine e risultati parziali intermedi e sensori per l'acquisizione dell'immagine.

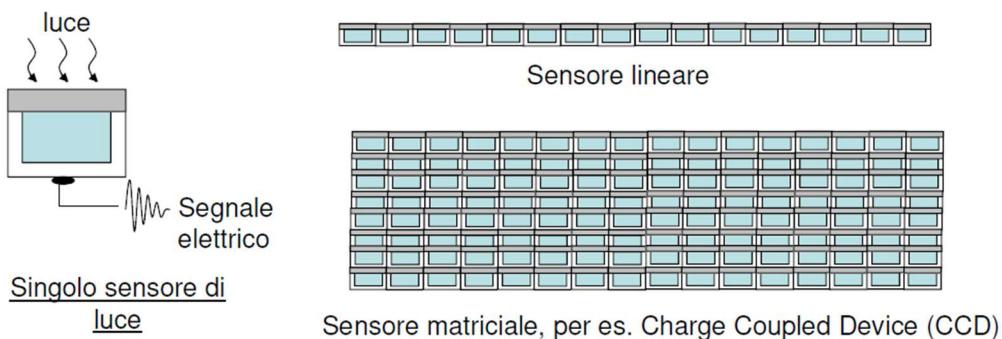


Figura 1 Sensore di luce e disposizione dei sensori

Riguardo l'acquisizione, è necessario utilizzare dei sensori di luce (trasduttori), i quali possono essere disposti linearmente o in forma matriciale. Il valore di tensione in uscita dal sensore (analogico) dovrà essere convertito tramite un *ADC* in un corrispondente valore (digitale). Ad ogni sensore corrisponderà un pixel a cui verrà associato un certo numero di bit in base alla rappresentazione utilizzata (*B/N*, scala di grigio o *RGB*). Ovviamente, aumentando il numero di sensori, aumenta la qualità dell'immagine ripresa, poiché la porzione di immagine associata ad un singolo sensore diminuisce così esaltando anche i più piccoli dettagli. Pertanto, facendo riferimento a questa metodologia, è possibile considerare un vettore o una matrice (in base alla disposizione lineare o matriciale dei sensori) di valori binari per la rappresentazione dell'immagine acquisita.

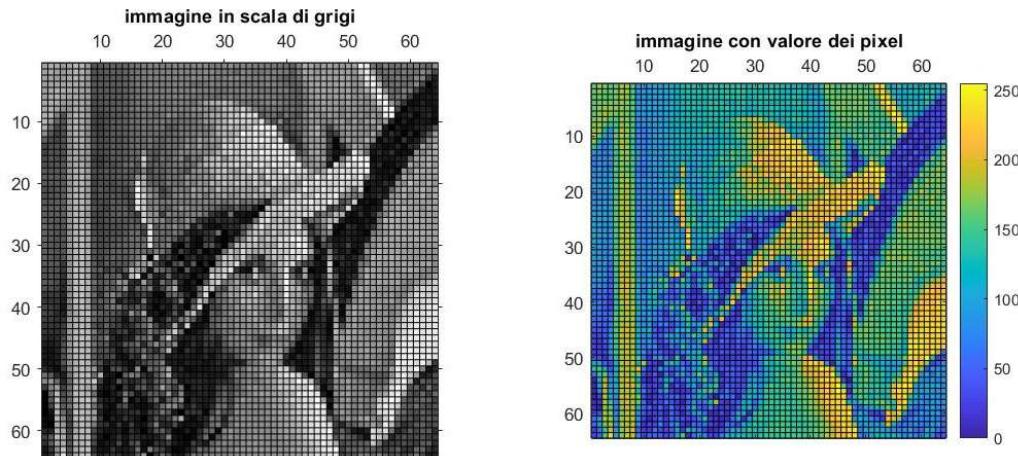


Figura 2 immagine in scala di grigi e immagine con valore dei pixel

In questo caso specifico, l’ipotesi iniziale è che l’immagine sia stata ottenuta tramite sensori disposti in forma matriciale e che essa presenti, pertanto, un numero di righe e di colonne entrambi pari a 64. Nello specifico, ogni elemento della matrice corrisponderà ad un valore binario ad 8 bit *unsigned* e ognuno di essi verrà filtrato in una maniera opportuna per ottenere in output il corrispondente pixel filtrato.

#### 1.4. Filtraggio spaziale

Trattandosi di un filtraggio spaziale e, pertanto, operando direttamente sui pixel dell’immagine, bisognerà considerare un filtro che possa permettere l’elaborazione di ogni elemento della matrice. Nello specifico, data l’immagine iniziale  $I$ , l’immagine  $Q$  filtrata sarà ottenuta applicando un filtro all’immagine  $I$  tramite un operatore  $T$ .

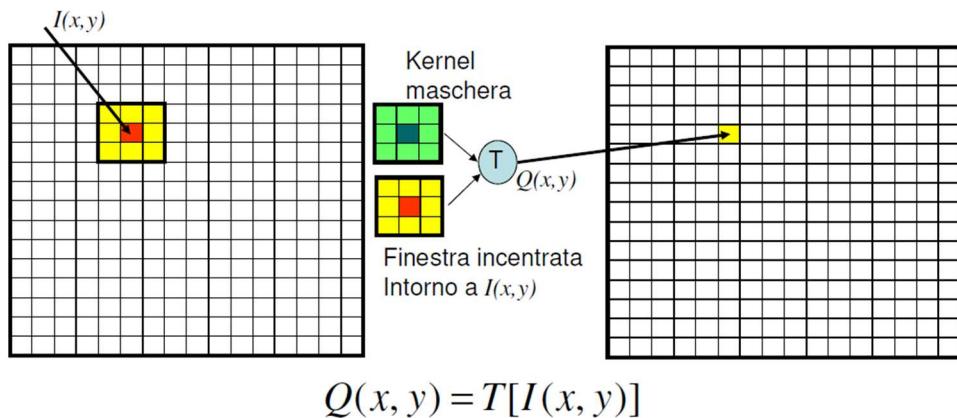


Figura 3 Filtraggio nel dominio dello spazio

Bisogna precisare che l’operatore  $T$  effettua una serie di operazioni ben specifiche: inizialmente vengono effettuate le moltiplicazioni tra la finestra di pixel di dimensione  $K \times K$  creata nell’immagine iniziale e il filtro  $K \times K$  e, infine, si sommano i risultati parziali ottenendo, di conseguenza, il pixel filtrato. In particolare, considerando la dimensione del filtro in questione, verranno effettuati 25 prodotti e 24 somme per ogni pixel dell’immagine  $I$  e, pertanto, un totale di 102400 prodotti e 98304 somme.

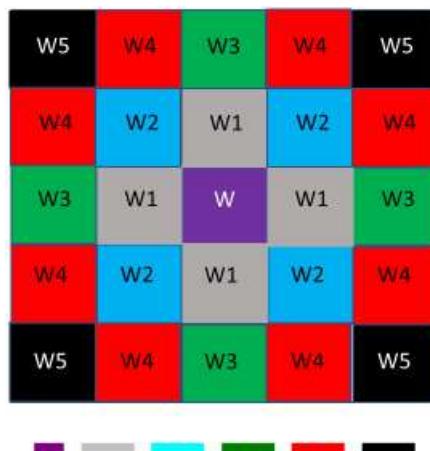
### 1.5. Filtro isotropico

Considerato il numero di operazioni sopra-citato, nelle specifiche di progetto è stato considerato un filtro isotropico che introduce alcune semplificazioni a livello hardware.

	FILTRO NON ISOTROPICO	FILTRO ISOTROPICO
MOLTIPLICAZIONI	102400	24576
SOMME	98304	98304

Figura 4 Confronto tra il numero di operazioni di un filtro non isotropico e isotropico

Specificatamente, nel caso di un filtro isotropico  $5 \times 5$ , risultano esserci 5 coefficienti all'interno del filtro che risultano occupare almeno 4 posizioni all'interno della matrice dei coefficienti. L'unico coefficiente che risulta essere presente soltanto una volta all'interno del filtro è il coefficiente centrale. In questo caso, non servirà più considerare 25 prodotti e 24 somme come precedentemente citato, ma saranno necessari solo 6 prodotti per i corrispettivi coefficienti, al più 7 operazioni di somma per il coefficiente che presenta più valori all'interno del filtro e, infine, 5 somme di convoluzione.



$W, W_1, W_2, W_3, W_4, W_5$

Figura 5 Filtro isotropico

Pertanto, è possibile definire il filtro isotropico come un filtro che a parità di spostamento intorno al coefficiente centrale, è possibile rinvenire stessi valori. Nel caso di filtro  $5 \times 5$  è possibile evincere le seguenti relazioni:

COEFFICIENTE ISTROPICO	COEFFICIENTI UGUALI
$W$	$W(2,2)$
$W_1$	$W(1,2) = W(2,1) = W(2,3) = W(3,2)$
$W_2$	$W(1,1) = W(1,3) = W(3,1) = W(3,3)$
$W_3$	$W(0,2) = W(2,0) = W(2,4) = W(4,2)$
$W_4$	$W(0,1) = W(0,3) = W(1,0) = W(1,4) = W(3,0) = W(3,4) = W(4,1) = W(4,3)$
$W_5$	$W(0,0) = W(0,4) = W(4,0) = W(4,4)$

Figura 6 Coefficienti di un filtro isotropico

Quindi, è possibile notare come alcuni dei termini dei prodotti siano in comune essendo uguali tra di loro i coefficienti all'interno del filtro considerato.

In questo progetto, sono stati considerati 5 filtri isotropici differenti per effettuare simulazioni e testing del circuito progettato.

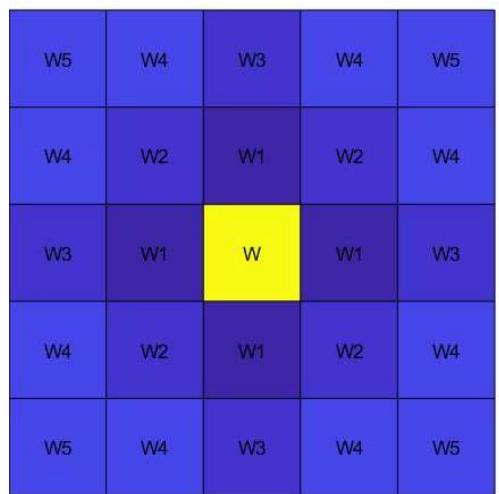


Figura 10 Filtro isotropico Gaussiano

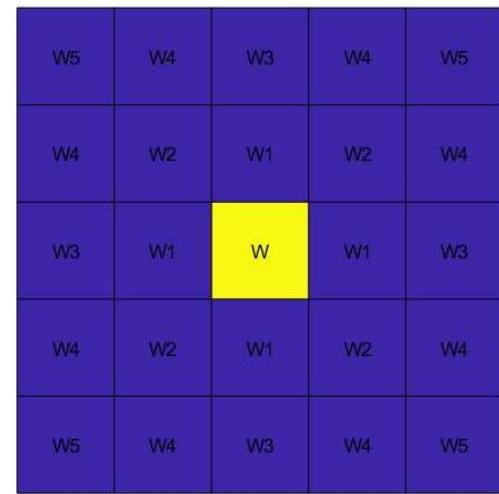


Figura 9 Filtro isotropico Mediano

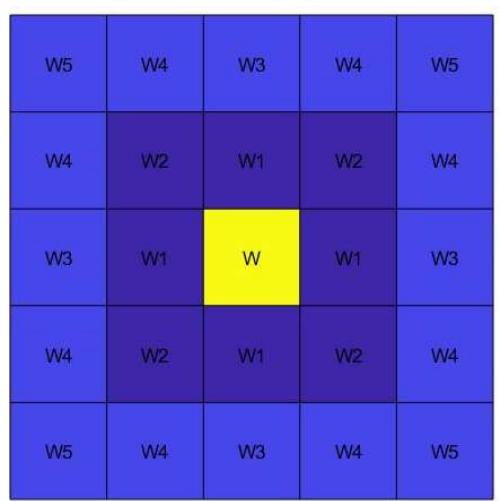


Figura 8 Filtro isotropico Laplaciano 1

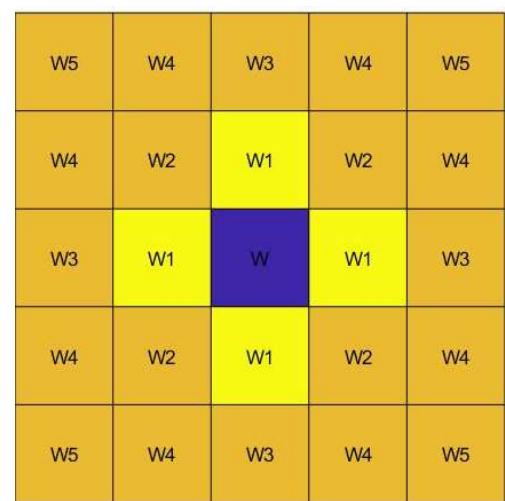


Figura 7 Filtro isotropico Laplaciano 2

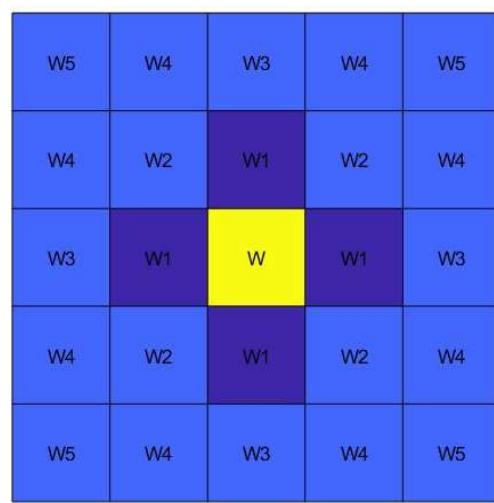


Figura 11 Filtro isotropico Laplaciano 3

### 1.6. Scansione raster

Effettuando il filtraggio spaziale, per filtrare ogni pixel verrà considerata una finestra dell'immagine differente, in base a quella che è la posizione della finestra del filtro nell'immagine stessa. Pertanto, considerando la sequenza  $X$  di pixel iniziale e la finestra del filtro  $f$ , l'obiettivo del filtraggio spaziale sarà ottenere una sequenza  $Y$  di pixel filtrati.

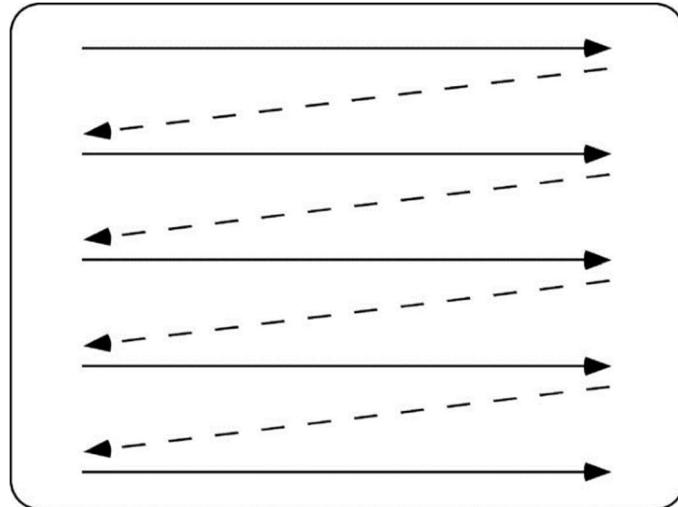


Figura 12 Raster scan

La sequenza  $X$  iniziale dovrà corrispondere all'insieme dei pixel della matrice di pixel iniziali. Questa sequenza di elementi potrà essere ottenuta considerando la scansione dell'immagine di tipologia raster. Nello specifico, la sequenza  $X$  sarà composta dall'insieme di righe considerate ognuna dall'indice 0 di colonna fino all'indice 63. Questo ordine corrisponde a quello con cui vengono realmente ricevuti i valori digitali dal sensore *i-esimo* al circuito progettato. Per la precisione, ad ogni iterazione  $i$ , considerando un anchor-point centrale, bisognerà fare in modo che il pixel *i-esimo* corrisponda al pixel centrale della finestra del filtro *i-esima* considerata. Pertanto, considerando un'operazione di filtraggio di questo tipo, il circuito riuscirà a produrre in uscita il pixel filtrato *i-esimo* per ogni finestra del filtro *i-esima* considerata.

### 1.7. Problema ai bordi e tecnica di padding

Considerando, pertanto, una finestra di filtro  $5 \times 5$  e i pixel ai bordi della matrice, è evidente che parte della finestra risulta essere incompleta poiché alcuni dei pixel considerati non esistono. Ad esempio, considerando il primo pixel della matrice, la finestra di filtro corrispondente sarà incompleta poiché risulteranno mancanti  $F(0,0)$ ,  $F(0,1)$ ,  $F(0,2)$ ,  $F(0,3)$ ,  $F(0,4)$ ,  $F(1,0)$ ,  $F(1,1)$ ,  $F(1,2)$ ,  $F(1,3)$ ,  $F(1,4)$ ,  $F(2,0)$ ,  $F(2,1)$ ,  $F(3,0)$ ,  $F(3,1)$ ,  $F(4,0)$ ,  $F(4,1)$  dal momento che il pixel centrale deve corrispondere al pixel corrente che si sta filtrando, cioè il pixel  $I(0,0)$ . Dato questo problema ai bordi, viene adottata una tecnica di padding, lo *zero-padding*, che consiste nel disporre il valore 0 in corrispondenza dei valori inesistenti.

### 1.8. Xilinx PYNQ-Z2

La PYNQ-Z2 è una scheda di sviluppo basata sull'FPGA ZYNQ XC7Z020, progettata per supportare PYNQ, un nuovo framework open-source che consente ai programmatori embedded di esplorare le possibilità dei SoC ZYNQ di Xilinx senza dover riprogettare circuiti logici di programmazione. Traendo vantaggio dalla logica programmabile e dal processore ARM, i progettisti hanno la possibilità di creare potenti sistemi embedded.

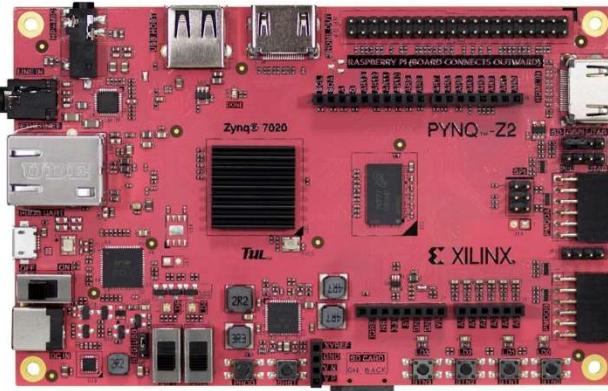


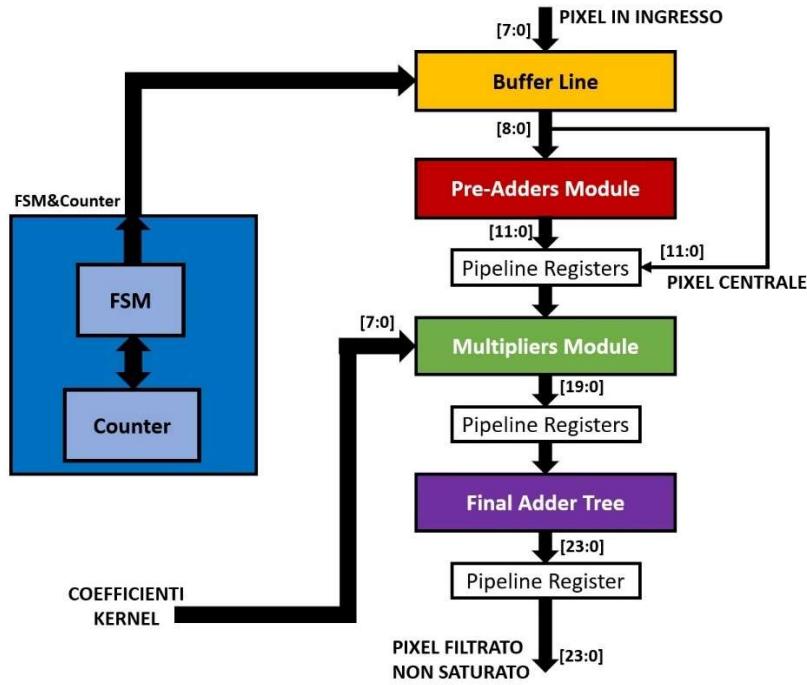
Figura 13 PYNQ-Z2

Di seguito sono riportate alcune specifiche della scheda PYNQ-Z2:

- 650MHz Dual-Core ARM® Cortex™-A9
- 512 MB DDR3
- 85,000 logic cells

## 2. Descrizione schema a blocchi

Da un punto di vista strutturale, lo schema a blocchi del circuito a più alto livello che nel nostro progetto realizza l'operazione di filtraggio è rappresentato in **Figura 14**.



*Figura 14 Schema a blocchi del circuito utilizzato per filtrare un'immagine 64x64*

Il primo blocco che incontriamo è il Buffer Line, la cui utilità sta nel garantire che i pixel possano trovarsi nelle posizioni corrette all'interno della finestra da processare, cosa che non potrebbe accadere in sua assenza dal momento che essi vengono letti dall'esterno in Raster Order, ovvero riga per riga. Vi è poi il blocco FSM&Counter, contenente un contatore e una macchina di controllo, che ha il compito di gestire l'intero flusso di elaborazione, ovvero determinare quando iniziare e terminare il processamento dei pixel. Successivamente, sapendo che la procedura di filtraggio è riconducibile ad un'operazione di Multiply and Accumulate (MAC), ovvero delle somme di prodotti, incontriamo un modulo di Pre-Adders, che somma, in via preliminare, i pixel accomunati allo stesso coefficiente del kernel isotropico, le cui uscite vanno, insieme al pixel centrale, in ingresso ad un modulo di moltiplicatori che calcolano il prodotto con i relativi coefficienti, le cui uscite vanno, a loro volta, in un ultimo albero di somma che le va a sommare per ottenere il generico pixel filtrato. Tra un modulo e l'altro, infine, sono stati inseriti dei registri di pipeline che incrementano il throughput complessivo del circuito. È da sottolineare, però, che tali registri non sono stati inseriti anche internamente ai moduli stessi, in quanto ciò avrebbe comportato non solo un aumento delle risorse utilizzate, ma anche un aumento ulteriore della latenza del circuito. Questo perché, nel momento in cui viene imposto un constraint sul clock, viene garantito che un qualsiasi circuito combinatorio, quale un sommatore, produca un risultato valido in un tempo minore del periodo del clock stesso. Se inseriamo dei registri di pipeline anche internamente ai moduli, però, lo stesso risultato verrebbe prodotto dopo diversi cicli di clock, rallentando di conseguenza tutto il circuito.

Prima di effettuare l'implementazione in VHDL, è stata condotta un'analisi su carta per comprendere con quanti bit dovessero lavorare i singoli componenti. Nel buffer line vi entrano i valori dei pixel su

8 bit, che andranno in ingresso al modulo dei Pre-Adders estesi su 9 bit. Questo si è ritenuto necessario per poterli effettivamente trattare come numeri positivi, poiché altrimenti i valori compresi tra 128 e 255, dal momento il circuito lavora in complemento a due, sarebbero stati considerati negativi. In uscita dal modulo dei Pre-Adders vi sono poi dati a 12 bit; in realtà, il numero di bit più elevato è 14, poiché vi è un albero di somma a otto operandi che, essendo strutturato su cinque livelli, produrrà un risultato su 14 bit; il motivo per cui, però, è stato possibile considerarli a 12 bit, è dato dal fatto che, facendo un'analisi nel worst case, ovvero quando i pixel che compongono la generica finestra sono tutti pari a 255, il massimo numero che si ottiene è 2040, che può tranquillamente essere rappresentato su 12 bit. Questo ci ha consentito di ottimizzare il carico di lavoro per i circuiti a valle. Successivamente, questi dati vanno, insieme ai coefficienti del kernel a 8 bit, in ingresso ai moltiplicatori, i quali producono, quindi, dati rappresentati su  $8+12=20$  bit. In realtà, anche in questo caso, questi ultimi si sarebbero potuti troncare a 19 bit, in quanto, mettendoci sempre nel worst case, gli estremi dell'intervallo di numeri ottenibili in uscita dai moltiplicatori sono  $-261120$  e  $259080$ , che possono essere rappresentati senza problemi su 19 bit, compreso il bit di segno. Tuttavia, è stato comunque ritenuto opportuno prelevare l'uscita con 20 bit, per rimanere in accordo con la definizione secondo cui il prodotto tra due numeri è espresso con un numero di bit pari alla somma di quelli che caratterizzano gli operandi in ingresso. Infine, questi valori vanno in un ultimo albero di somma che, essendo scomposto su quattro livelli, produce un'uscita su 24 bit. Logicamente anche in questo caso avremmo potuto troncare il risultato a 21 bit, dal momento che i valori massimo e minimo sono 809625 e -816000, ma, siccome essi rappresentano l'uscita complessiva del circuito, è stato evitato di farlo. Invece, se ci fossero stati altri circuiti a valle, sarebbe stato opportuno effettuare il troncamento poiché avrebbe ulteriormente ridotto il carico di lavoro complessivo.

In **Figura 15** viene mostrato lo schematic RTL del Filter Circuit, a livello di modulo, in cui sono stati evidenziati i moduli descritti nello schema a blocchi.

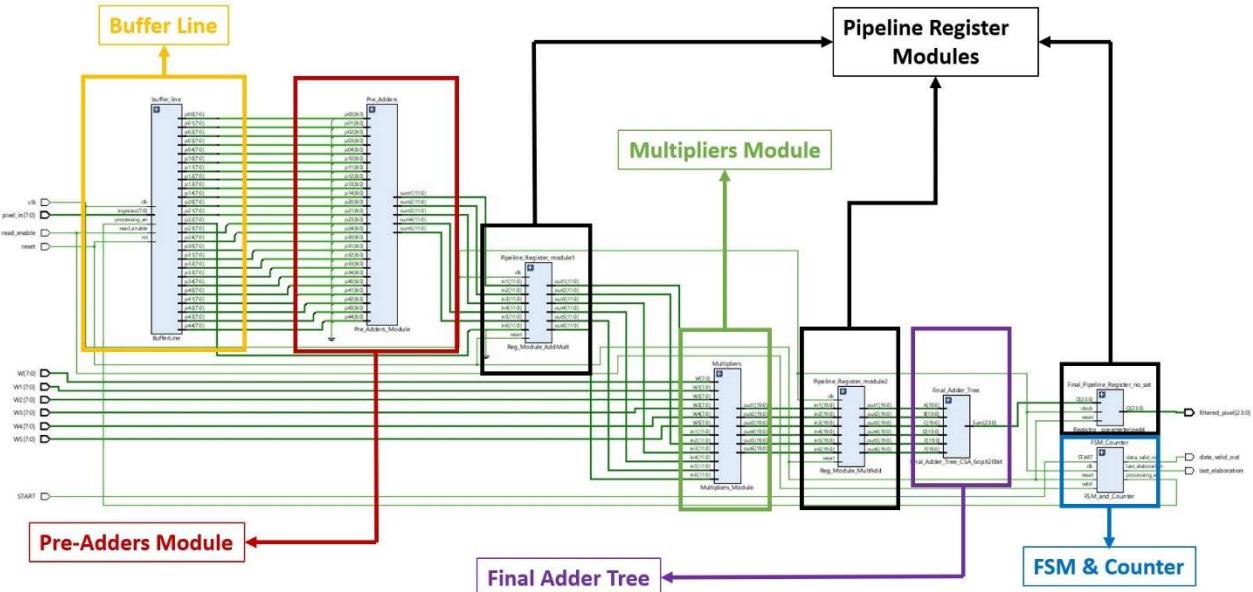


Figura 15 Schematic RTL del circuito ottenuto in Vivado per filtrare un'immagine 64x64 pixels

## 2.1. FSM

In **Figura 16** è illustrato il diagramma a stati della FSM impiegata per controllare il processo di elaborazione dei pixel dell'immagine.

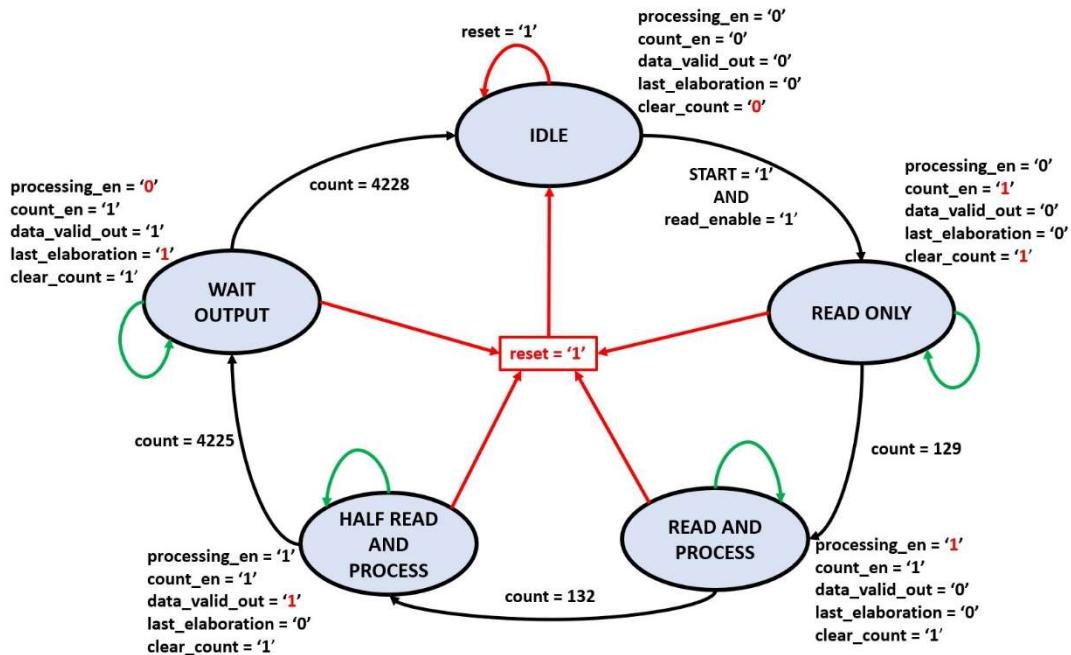


Figura 16 Diagramma a stati della macchina di controllo del circuito

Essa è stata implementata utilizzando il modello della macchina di Moore e perseguiendo due obiettivi principali:

- 1) Fare in modo che il processamento dei pixel non inizi fintanto che non trascorre la latenza di ingresso del circuito, passaggio necessario per attendere che venga costruita la prima finestra utile da poter processare;
- 2) Fare in modo che, al termine della lettura dei dati, il processamento non si fermi di conseguenza, bensì continui fino a quando non viene processato un numero di pixel pari a quello che compone l'immagine di partenza, ovvero fino a quando non si effettua il filtraggio sull'ultimo pixel.

Entrambi gli obiettivi sono stati raggiunti attraverso l'ausilio del segnale **processing\_en**, il cui effetto è tale per cui, se si trova al valore logico alto, i pixel della finestra creata vengono trasmessi alle circuiterie a valle del Buffer Line, sommatori e moltiplicatori, altrimenti scorrono in esso, ma non vengono portati in uscita.

Ovviamente, le transizioni dei segnali di controllo sono state gestite avvalendoci di un contatore modulo 13 che conta i cicli di clock che trascorrono. Prima di offrire, però, una descrizione dei singoli stati della FSM, è opportuno dare un'interpretazione dei segnali in ingresso e in uscita dalla stessa, poiché ci aiuta a comprendere cosa succede nel circuito.

**START**: è un segnale comandato dall'utente per decidere quando far lavorare il circuito;

**read\_enable**: si alza nel momento in cui iniziano ad entrare i pixel nel Buffer Line e rimane tale fino a quando non viene letto l'ultimo pixel.

**processing\_en**: indica se il processamento dei pixel è abilitato o meno;

**count\_en**: rappresenta il segnale di abilitazione del contatore utilizzato;

**data\_valid\_out**: indica se in uscita dal circuito complessivo abbiamo dati validi o meno;

**last\_elaboration**: determina quando si sta processando l'ultimo pixel dell'immagine;

**clear\_count**: lo utilizziamo per azzerare il valore del conteggio ad elaborazione terminata, operazione necessaria qualora si volesse processare subito un'altra immagine. Come si può notare dal diagramma a stati, questo segnale è considerato attivo basso, ovvero che il contatore viene azzerato quando è al valore logico basso.

Compreso il significato dei segnali connessi alla FSM, si può passare alla descrizione dei vari stati:

**IDLE**: in questo stato il circuito è completamente inattivo, motivo per cui le uscite sono tutte al valore 0;

**READ\_ONLY**: si transita in questo stato non appena i segnali START e read\_enable sono contemporaneamente alti e vi si resta fintanto che non trascorre la latenza di ingresso, durante la quale bisogna leggere senza fare operazioni, nell'attesa che il Buffer Line si riempia. Per tener conto dello scorrimento del tempo e capire quando si arriva all'istante in cui la latenza si può considerare trascorsa, ci si avvale del contatore, poiché, essendo che esso viene abilitato quando si passa in questo stato, ovvero appena inizia la lettura, e dal momento che la latenza è pari a 131 (è il numero di registri attraverso cui deve passare il primo pixel per arrivare nel centro della finestra di convoluzione), si può abilitare il processamento dell'immagine non appena il conteggio arriva a 129. Il motivo per cui viene considerato 129 e non 131 è dato dalle seguenti spiegazioni:

- 1) Per come è stato progettato il circuito, il primo pixel viene letto quando il valore del conteggio è ancora a zero, per cui esso raggiunge la posizione centrale della finestra a 130 e non 131;
- 2) Essendo che lo stato della FSM cambia in corrispondenza del fronte di salita del clock, anche le sue uscite subiscono la stessa sorte, per cui, è necessario cambiare stato quando il conteggio arriva a 129 in modo da avere queste ultime già stabili quando passa a 130. Se così non fosse, il primo pixel ad essere processato sarebbe il secondo, mentre il primo verrebbe scartato del tutto.

**READ\_AND\_PROCESS**: Trascorsa la latenza in ingresso, si passa finalmente in questo stato, in cui, abilitando il processamento, i pixel della finestra vengono portati ai circuiti che stanno a valle del Buffer Line, partendo dal modulo dei Pre-Adders. Dal momento che però, nel circuito complessivo vi sono tre registri di pipeline, il primo risultato valido viene prodotto in uscita solo dopo tre cicli di clock rispetto a quando inizia il processamento stesso, motivo per cui il segnale **data\_valid\_out**, in questo stato, si mantiene al valore logico basso. Non appena il conteggio arriva a  $129+3=132$ , invece, viene prodotto il primo pixel filtrato, per cui si passa allo stato **HALF\_READ\_AND\_PROCESS**.

**HALF\_READ\_AND\_PROCESS**: a partire da Count=132, ad ogni colpo di clock viene prodotto un pixel filtrato, per cui, in questo stato, oltre a continuare la fase di processing, il segnale **data\_valid\_out** transita al valore logico alto e vi si resta fino a quando non vengono processati tutti i pixel, situazione che viene a crearsi quando il conteggio assume un valore pari a  $4095+130=4225$  (non si considera 131 per lo stesso motivo espresso quando abbiamo descritto lo stato **READ\_ONLY**), ovvero la somma tra il tempo necessario a leggere tutti i pixel dell'immagine e la latenza in uscita.

**WAIT\_OUTPUT:** Terminato il processamento, si passa, infine, nello stato WAIT\_OUTPUT, dove il valid in uscita viene mantenuto ancora alto dal momento che, quando il conteggio arriva a 4225, è vero che terminano i pixel da processare, ma ancora devono essere prodotti in uscita gli ultimi tre pixel filtrati. Infatti, quando il conteggio arriva poi a  $4225+3=4228$  si passa automaticamente allo stato IDLE, poiché l'elaborazione dell'immagine può considerarsi terminata.

Come si sarà potuto notare durante la descrizione della FSM, non è presente nessuno stato che indichi quando la lettura termina. Questo perché in questa situazione le uscite della FSM stessa non subiscono variazioni, per cui l'inserimento di uno stato in più in questo punto è del tutto inutile, poiché complicherebbe soltanto la sua struttura.

## 2.2. Contatore

Come detto precedentemente, è stato necessario implementare un circuito contatore, che si occupasse di gestire gli stati della FSM tramite il conteggio dei dati in ingresso. È un elemento sequenziale, sincrono in quanto il conteggio viene regolato dal clock. Oltre al clock possiede altri 2 segnali di ingresso:

-**Clear:** ha la funzione di azzerare il conteggio del contatore quando viene alzato il segnale di reset, questo segnale sarà la OR tra il segnale di reset generale e il segnale di azzeramento del conteggio generato dalla FSM (**clear\_count**);

-**Enable:** rappresenta la condizione per cui il contatore è abilitato al conteggio;

In uscita invece fornisce il segnale Q a 13-bit che rappresenta il valore del conteggio istantaneo a cui è arrivato il contatore. Non è stato necessario utilizzare i segnali di threshold in uscita.

Nel descrivere tramite codice VHDL un circuito contatore ci si trova a dover utilizzare l'operatore ad alto livello dell'addizione per incrementare internamente il conteggio. Come scelta progettuale si è voluto escludere l'utilizzo di operatori ad alto livello e descrivere singolarmente ogni circuito utilizzato.

Si è pensato inizialmente di utilizzare un tradizionale circuito di somma e accumulo dove uno dei due operandi del sommatore si pone al valore “1” e si utilizza un registro in uscita come tampone per mantenere il dato. Questo circuito però avrebbe potuto introdurre un notevole ritardo quindi si è deciso di scartare questa opzione.

Il circuito che invece si è implementato è un contatore basato su Flip Flop Toggle, quest'ultimo deriva dal Flip Flop JK con i pin J e K collegati assieme.

Lo schema più semplice di Flip Flop Toggle ha un ingresso, un clock e due uscite complementari. A questo schema base si è prevista l'aggiunta del segnale di reset e del segnale di enable.

Le funzioni che svolge sono di memoria e di toggle:

- lo stato di toggle consiste nella negazione del valore precedentemente memorizzato;
- lo stato di memoria mantiene invariata l'uscita.

Il concetto alla base di questo contatore è che si inverte l'uscita del FF solo se tutte le uscite precedenti sono pari a “1”; questo si effettua collegando sul toggle la “and” dell'uscita e dell'ingresso del FF precedente.

Lo schema del contatore sincrono vede quindi questi FF collegati in cascata secondo lo schema riportato nella **Figura 17**.

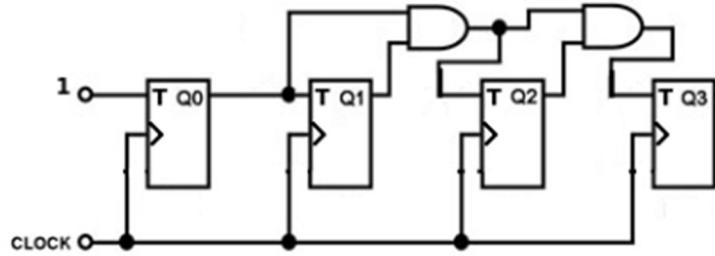


Figura 17 Contatore sincrono

Ponendo il primo ingresso pari a “1” e prendendo parallelamente le uscite dei FF si ottiene il numero di colpi di clock trascorsi dall’ultimo reset. Nel nostro caso questo contatore dovrà essere a 13 bit e, quindi, si estenderà lo schema con 13 Flip Flop.

Per facilitare l’operazione di descrizione dell’architettura tramite un solo costrutto di *generate* si implementa direttamente nel modulo del Flip Flop la logica legata alla porta AND.

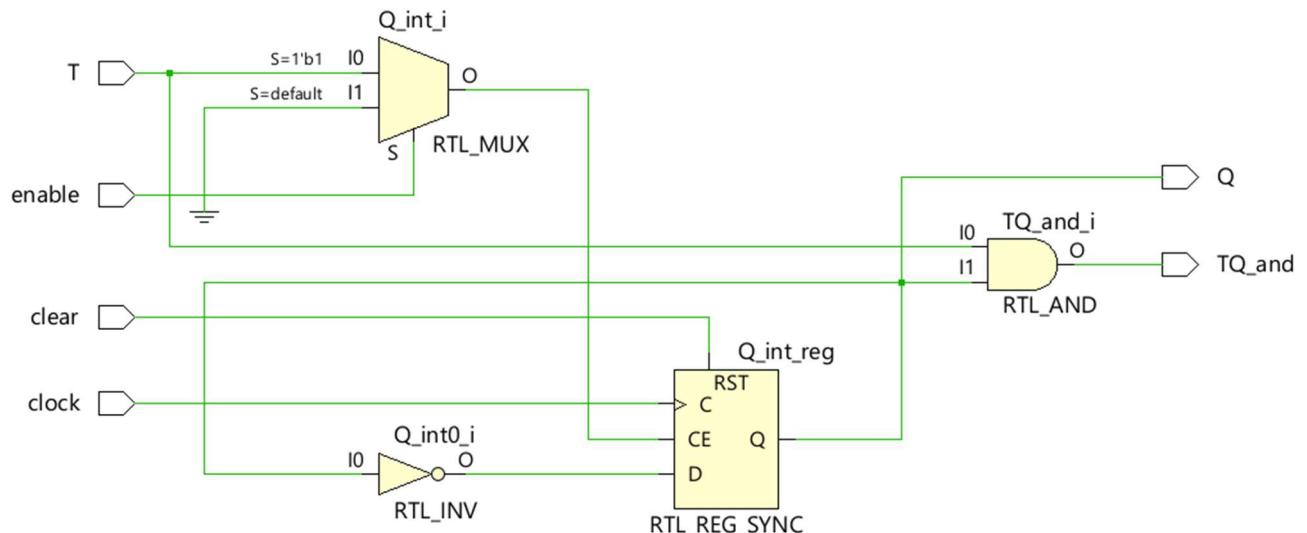


Figura 18 Schematic RTL relativo al Flip Flop T

### 2.3. BufferLine

Il modulo Buffer Line è il primo componente dello stack di moduli del componente Filter Circuit che ingloba tutta l’unità di convoluzione bidimensionale. In ingresso a tale componente abbiamo: il clock, il reset, il pixel di ingresso, un read enable e un processing enable. In uscita da tale modulo abbiamo la finestra di convoluzione costituita dai 5x5 pixel, da moltiplicare successivamente per i rispettivi 5x5 coefficienti del kernel. Tali pixel della finestra in uscita sono nominati secondo la riga e la colonna di appartenenza, partendo dall’angolo in alto a sinistra, per cui avremo:

P00	P01	P02	P03	P04
P10	P11	P12	P13	P14
P20	P21	P22	P23	P24
P30	P31	P32	P33	P34
P40	P41	P42	P43	P44

Figura 19 Finestra di convoluzione

In base all'isotropicità del kernel i pixel della finestra aventi lo stesso colore saranno moltiplicati per lo stesso coefficiente del kernel; pertanto, nella entity del componente sono riportati affiancati, ad esempio:

$P_{00}, P_{04}, P_{40}, P_{44}$  sono i pixels ai corner più esterni che andranno moltiplicati per il coefficiente  $W_5$  in base allo schema:



Figura 20 Filtro isotropico

Per quanto riguarda la costruzione della finestra, la buffer line internamente è costituita da un array di registri connessi in cascata tra di loro, ogni registro contiene un pixel dell'immagine gray scale ad 8 bit. Ad ogni evento di clock ogni registro alla  $i$ -esima posizione campiona l'uscita del registro precedente, causando lo shift dei pixel all'interno della struttura di bufferizzazione.

Per poter risolvere il problema dell'accesso a posizioni non consecutive alla memoria, viene effettuata un'estensione circolare ai bordi, per cui la struttura risultante è dipendente dalla dimensione dell'immagine, dalla dimensione del kernel e dal throughput che si vuole ottenere (nel caso in esame 1, ovvero verrà prodotto un pixel dell'immagine processata per ciclo di clock).

Data un'immagine di  $N \times M$  pixel, ed un kernel di dimensione  $K \times K$ ,  $K = 2R + 1$ , dove  $R$  è il raggio del filtro, occorrono per la creazione della finestra:

- $K \times K$  registri per i pixel della finestra
- $K - 1$  FIFO da  $m-k$  registri

Ogni FIFO dovrà avere proprio  $M - K$  registri (anche chiamata profondità della FIFO) perché il numero di elementi presenti in ogni istante nella FIFO e nel filtro è pari ad una riga completa dell'immagine.

La struttura è stata dimensionata per poter filtrare un'immagine  $64 \times 64$  pixel, di conseguenza, avremo:

- 25 registri per la finestra
- 4 FIFO da 59 registri, costituite cioè da 236 registri

In totale, dunque, il numero di registri della struttura di bufferizzazione sarà di 261 registri, ed in base alle scelte fatte precedentemente la struttura risultante è:

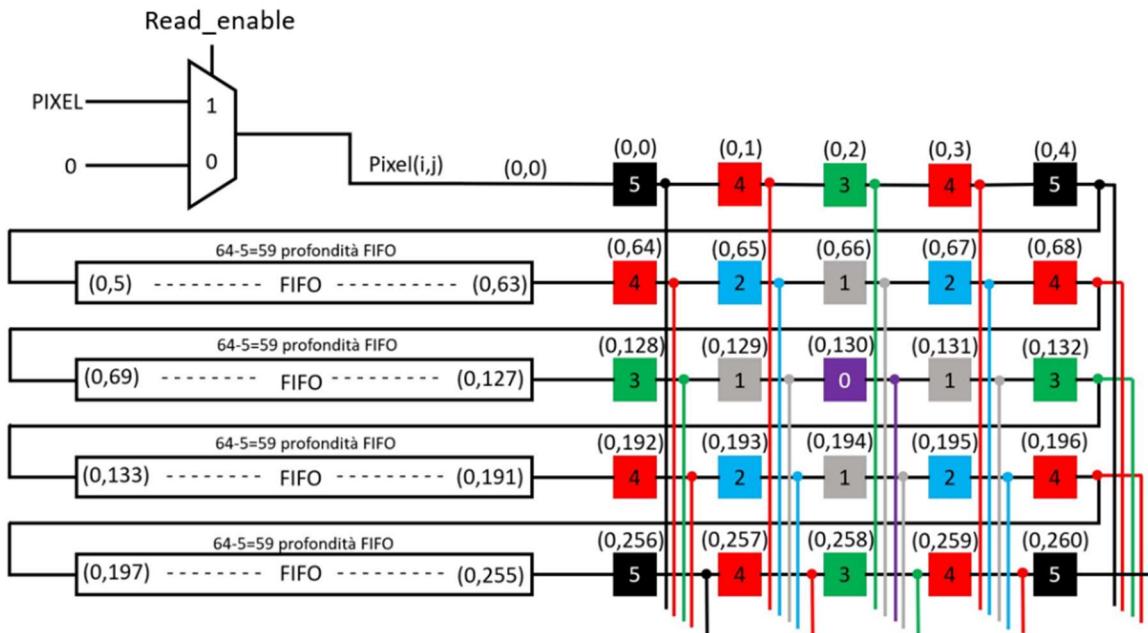


Figura 21 Struttura di bufferizzazione

Noto questo schema, la prima distinzione fatta tra registri afferenti alla finestra e registri afferenti alle FIFO può essere tralasciata, l'intera struttura può essere vista come un unico array monodimensionale di registri connessi in cascata da 261 posizioni (da 0 a 260).

Nel codice sorgente vhdl relativo al componente Buffer Line, dunque, viene creato un tipo detto reg\_array di 261 std\_logic\_vector, e il tipo matrix che definisce una matrice di  $5 \times 5$  di std\_logic\_vector per la finestra.

Nell'unico process dell'architecture, viene descritto il seguente comportamento, sul fronte di salita del clock:

Se reset è alto (attivo alto) tutti i registri vengono azzerati, altrimenti si shiftano i pixel ( $buffer1(j) <= buffer1(j - 1)$ ) inoltre se read\_enable è alto si fa entrare il pixel dell'immagine dall'esterno, se è basso un pixel nullo (ovvero 00000000).

È necessario fare entrare un pixel nullo ad ogni ciclo di clock una volta che viene letto l'ultimo pixel dell'immagine poiché l'operazione di filtraggio non è ancora completa.

Attraverso 2 for-generate innestati la finestra viene costruita prelevando i pixel nei registri nelle posizioni: 0 – 4 per la prima riga della finestra, 64 – 68 per la seconda riga, 128 – 132 per la terza riga e così via, come riportato in figura.

Per valorizzare ogni riga della finestra si effettua un incremento di un multiplo di 64 ad ogni iterazione.

Infine, verranno valorizzati i segnali da P00 a P44 della finestra se processing\_en è alto, altrimenti le uscite rimarranno al valore logico basso, questo consente di ridurre la dissipazione di potenza dell'intera unità di convoluzione bidimensionale quando ancora non è trascorsa la latenza del circuito, poiché le uscite ovvero gli elementi della finestra di convoluzione rimangono statici.

## 2.4. Circuiti sommatori

Come circuito sommatore si è scelto di adottare l'approccio ad albero di somma Carry Save. Tale approccio necessita di due tipi di sommatori: il primo ha lo scopo di sommare i valori dei pixel dell'immagine da filtrare e quindi svolgere la funzione di pre-adder; il secondo circuito dovrà invece sommare i valori in uscita dal moltiplicatore.

Poiché lavoriamo con un filtro isotropico 5x5, sono richiesti 4 pre-adder che eseguono la somma a 4 operandi e un quinto pre-adder che esegue la somma a 8 operandi. Dato che i valori in ingresso provengono dalla matrice dell'immagine, gli operandi dei pre-adder saranno a 9 bit, in quanto il dato in uscita dal buffer line è una word a 9 bit (signed).

I pre-adder eseguiranno la somma tra i valori dei pixel utilizzati nelle somme comuni.

È stato preferito l'approccio Carry Save in quanto è un'architettura molto più performante visto che limita la propagazione del carry all'ultimo stadio del processo e fornisce nel complesso performance migliori. L'utilizzo di un approccio ad albero di RCA avrebbe causato un ritardo notevolmente maggiore per effetto della propagazione interna dei riporti.

Il concetto base dell'approccio Carry Save è quello di parallelizzare il calcolo della somma a 3 operandi tramite un banco di Full Adder che genera due uscite:

1. Somma parziale (SP);
2. Vettore dei riporti (VR).

Quindi per ottenere il risultato di questa somma a 3 operandi è possibile dividere l'operazione in 3 step:

1. STEP1: Calcolare la somma parziale (SP) e il vettore dei riporti (VR) dalla somma delle terne dei bit singoli;
2. STEP2: Estendere con segno la SP e shiftare il VR di una posizione a sinistra;
3. STEP3: Sommare SP esteso e VR shiftato per avere la somma dei 3 operandi.

Dal punto di vista dei ritardi, lo STEP 1 introduce un ritardo di un Full Adder a prescindere dal numero di bit degli operandi. Lo STEP 2 invece viene gestito a livello di routing dei bus, quindi è praticamente un'operazione a ritardo nullo. Infine lo STEP 3, che è il processo più oneroso, introduce un ritardo che dipende dal tipo di sommatore utilizzato, nel nostro caso (RCA) introduce un ritardo proporzionale al numero di bit degli operandi in ingresso.

### 2.3.1. Pre-Adder

Effettuando un confronto tra il ritardo dei due approcci, si ha:

Sommatore a 4 operandi	
Albero di RCA	Albero di CSA
$\tau = \tau_{RCA}(n + 1) + \tau_{RCA}(n + 2)$	$\tau = 2\tau_{FA} + \tau_{RCA}(n + 2)$

Sommatore a 8 operandi	
Albero di RCA	Albero di CSA
$\tau = \tau_{RCA}(n + 1) + \tau_{RCA}(n + 2) + \tau_{RCA}(n + 3)$	$\tau = 4\tau_{FA} + \tau_{RCA}(n + 4)$

( $n = \text{numero di bit}$ )

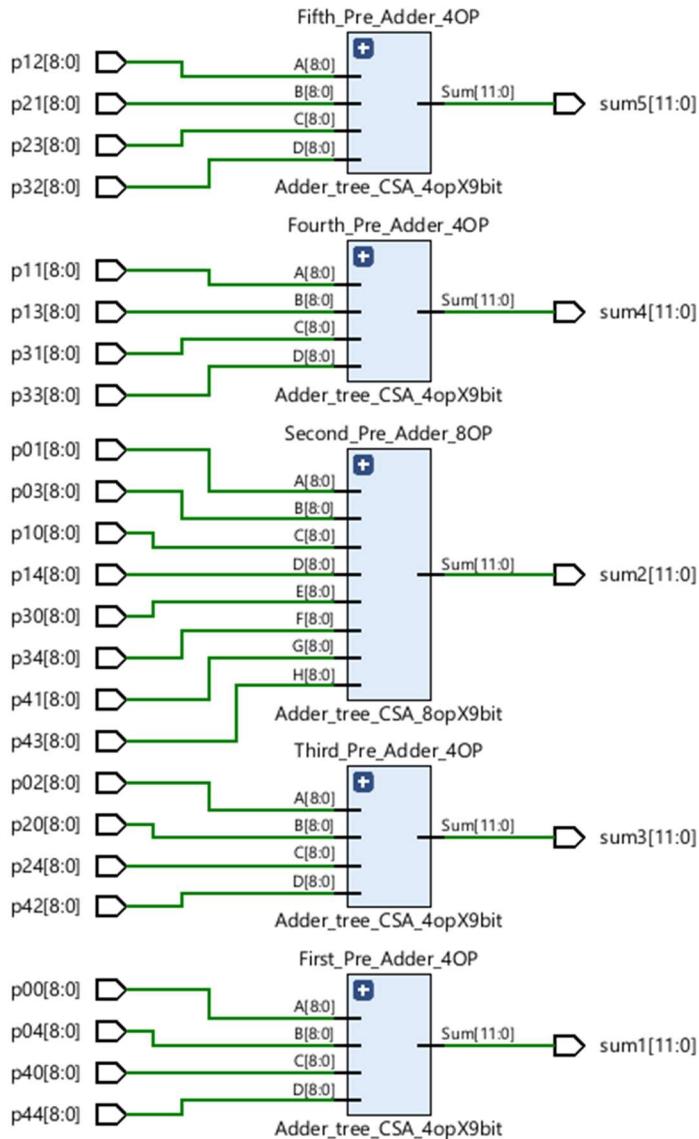


Figura 22 Schematic RTL relativo al Pre-Adder Module

### 2.3.1.1. Pre-Adder a 8 operandi a 9 bit

Nel circuito sommatore a 8 operandi, si sommano sei operandi in due banchi di FA, unendo gli step 1 e 2 in modo da incorporare in un unico modulo estensione e shift dei vettori in uscita. Dopo un ritardo pari al ritardo introdotto dal singolo FA, si ottengono due coppie di vettori:  $SP_1, SP_2$ , e  $VR_1, VR_2$ .

Si utilizza un secondo di livello di FA che in due banchi somma i 4 vettori ottenuti precedentemente e i due ingressi (opportunamente estesi). Dopo il ritardo di un singolo FA si ottengono due vettori di somme parziali ( $SP_3, SP_4$ ) e due vettori relativi ai riporti ( $VR_3, VR_4$ ).

Con un terzo livello di FA si sommano tra loro 3 dei 4 vettori ottenuti allo step precedente e il quarto viene esteso per renderlo uniforme ai risultati di questo stadio. Dopo il ritardo di un singolo FA si ottengono 3 vettori, uno relativo alla somma parziale ( $SP_5$ ), uno relativo al vettore dei riporti già shiftato ( $VR_5$ ) e la somma parziale ( $SP_4$ ) estesa.

Considerando questi tre vettori si procede ad una nuova somma con uno stadio di FA. Dopo il ritardo di un singolo FA si ottiene la somma parziale finale ( $SP_6$ ) e il vettore dei riporti finale ( $VR_6$ ) che andranno in ingresso ad un RCA ad  $n+4$  bit per completare l'operazione di somma.

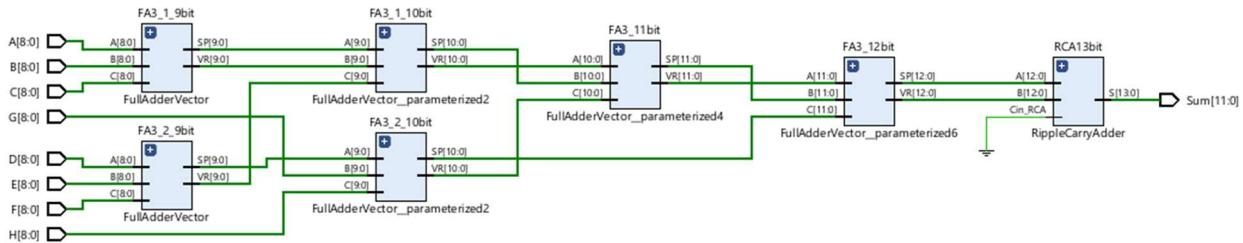


Figura 23 Schematic RTL relativo al Pre-Adder a 8 operandi a 9 bit

### 2.3.1.2. Pre-Adder a 4 operandi a 9 bit

Il circuito sommatore a 4 operandi, seguendo l'architettura di quello appena descritto, utilizza due livelli di FA e un livello di RCA finale. Il primo livello vede 3 dei 4 operandi come ingressi del primo banco di FA, dopo il ritardo di un FA si ottengono in uscita il vettore di somma parziale  $SP_1$  e il vettore dei riporti  $VR_1$  e verrà esteso il quarto operando.

Il secondo livello vede questi vettori appena processati in ingresso ad un banco di FA che, trascorso il ritardo di un FA, genererà in uscita  $SP_2$  e  $VR_2$ . Questi vettori verranno sommati tramite il sommatore finale RCA per generare la somma totale.

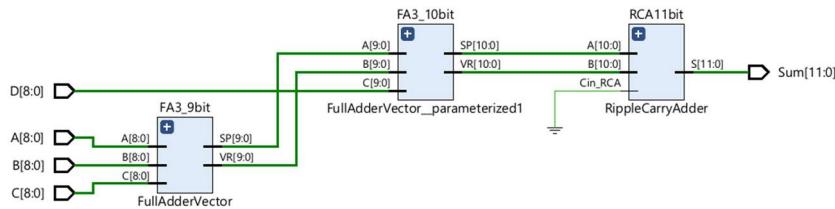


Figura 24 Schematic RTL relativo al Pre-Adder a 4 operandi a 9 bit

Data l'estensione e lo shift dei vettori in uscita al banco di filtri, si aumenta di un bit la dimensione della word in ingresso per ogni livello di somma dell'albero. Per esempio, nel caso del sommatore ad 8 operandi a 9 bit, si ha un'uscita a 14 bit ma il valore massimo che si ha comandato 8 operandi a 9 bit è rappresentabile con 12 bit. Per questo motivo vengono troncati i primi due bit in uscita.

### 2.3.2. Final-Adder

Il sommatore in uscita al moltiplicatore è a 6 operandi a 20 bit, e segue lo stesso schema visto per i pre-adder: si inviano gli operandi direttamente ai due banchi di FA che generano in uscita le due coppie di vettori  $SP_1$ ,  $SP_2$ ,  $VR_1$  e  $VR_2$ . I primi tre andranno ad un successivo banco di FA che genera le uscite  $SP_3$  e  $VR_3$ .

$VR_2$  verrà esteso e sommato tramite un ultimo banco di FA a  $SP_3$  e  $VR_3$ . La somma totale sarà l'uscita di un RCA a 24 bit che ha in ingresso i vettori d'uscita di quest'ultimo banco.

Da notare come i banchi di FA a 20 bit introducano lo stesso ritardo dei banchi di FA da 9 bit.

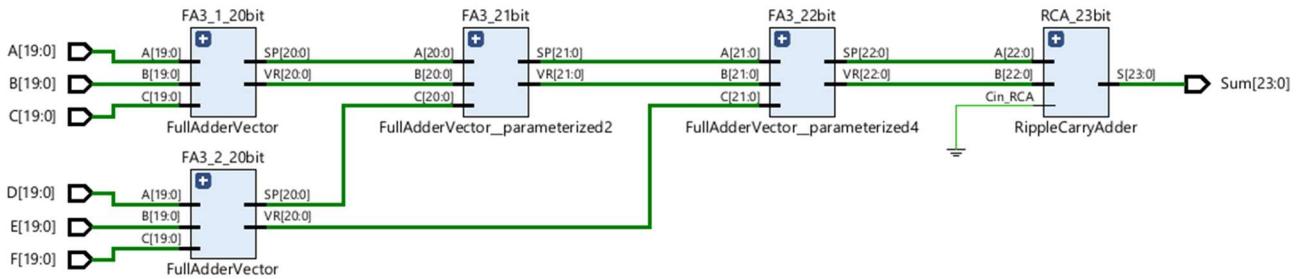


Figura 25 Schematic RTL relativo al Final-Adder

## 2.5. Booth Multiplier

### 2.5.1. Caratteristiche generali e scelte progettuali

Il moltiplicatore è senza dubbio un altro componente fondamentale per le operazioni di filtraggio, motivo per cui è stato necessario progettarne uno per il nostro progetto. Durante il corso è stato analizzato il principio di funzionamento di due tipi di moltiplicatori, quello che esegue l'algoritmo “carta e penna” e quello di Booth, il cui vantaggio, rispetto al primo, è dato dal fatto che consente di ridurre il numero complessivo di prodotti parziali alla metà di quelli che avremmo con l'approccio “carta e penna”. Infatti, proprio per questo è stato deciso di implementare un moltiplicatore di Booth, poiché, diminuendo le operazioni di somma da effettuare, si riduce, di conseguenza, il tempo di calcolo e la dissipazione di potenza. Partendo, innanzitutto, da una sua descrizione generale, esso si basa sull'esecuzione di operazioni di codifica effettuate (tra i due operandi da moltiplicare) sul moltiplicatore. Il vantaggio di avere un numero di prodotti parziali minore viene dal fatto che tale codifica è calcolata non sul singolo bit, bensì su terne, ognuna delle quali condivide un bit con la precedente e uno con la successiva. In base ad una tabella predefinita, che, per comodità, viene riportata qui di seguito in **Figura 26**, ad ogni terna viene associata una certa codifica, rappresentata in digit, che consente di determinare il valore che assume il prodotto parziale relativo a quella stessa terna semplicemente basandosi sul valore del moltiplicando. Questa è una cosa vantaggiosa, in quanto, sapendo quanto vale quest'ultimo, si possono precalcolare tutti i possibili valori dei prodotti parziali, in modo tale che quello da associare ad una particolare terna possa essere determinato semplicemente selezionando quello corretto, ovvero, utilizzando un banale multiplexer.

Terna	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 26 Tabella delle codifiche del moltiplicatore con l'algoritmo di Booth

Relativamente alle scelte progettuali, è stata condotta, per prima cosa, un'analisi su carta, per capire con quanti bit dovesse lavorare il moltiplicatore e decidere come classificare i due operandi. Relativamente al primo punto, è stato notato che, mentre il coefficiente del filtro è un dato sempre a 8 bit in complemento a due, l'altro operando non ha sempre lo stesso numero di bit poiché ciò dipende dal numero di pixel sommati nei vari blocchi di Pre-Adders, utilizzati per sfruttare l'isotropicità del kernel. Infatti, vi è un albero di somma a quattro operandi, che, prendendo ingressi a 9 bit, produce un risultato a 12 bit, un altro albero di somma a otto operandi, che invece, produce in uscita dati a 14 bit, e poi il pixel centrale a 8 bit che, non venendo sommato con nessun altro pixel, va direttamente in ingresso al moltiplicatore. Da quanto appena detto, dunque, sembrerebbe che si dovrebbero utilizzare tre moltiplicatori diversi per poter lavorare opportunamente nei tre casi. In realtà, però, per rendere la struttura quanto più modulare possibile, è stata uniformata la lunghezza dei dati in ingresso ai moltiplicatori a 12 bit, in modo da poter implementare un unico moltiplicatore di Booth che, lavorando su dati a 8 e 12 bit, produce risultati a 20 bit. Riguardo alla classificazione dei due operandi, invece, si è preferito considerare come moltiplicatore il coefficiente del kernel piuttosto che l'uscita dal Pre-Adder in quanto abbiamo un numero più piccolo di prodotti parziali da sommare (quattro nel primo caso e sei nel secondo).

### 2.5.2. Progettazione

Dal punto di vista implementativo, il moltiplicatore di Booth progettato, il cui schema è riportato in **Figura 27**, è costituito da un insieme di componenti interni fondamentali:

- Precomputation module;
- Booth encoders;
- Partial Product selectors;
- Adder tree a quattro operandi con approccio Carry Save Adder.

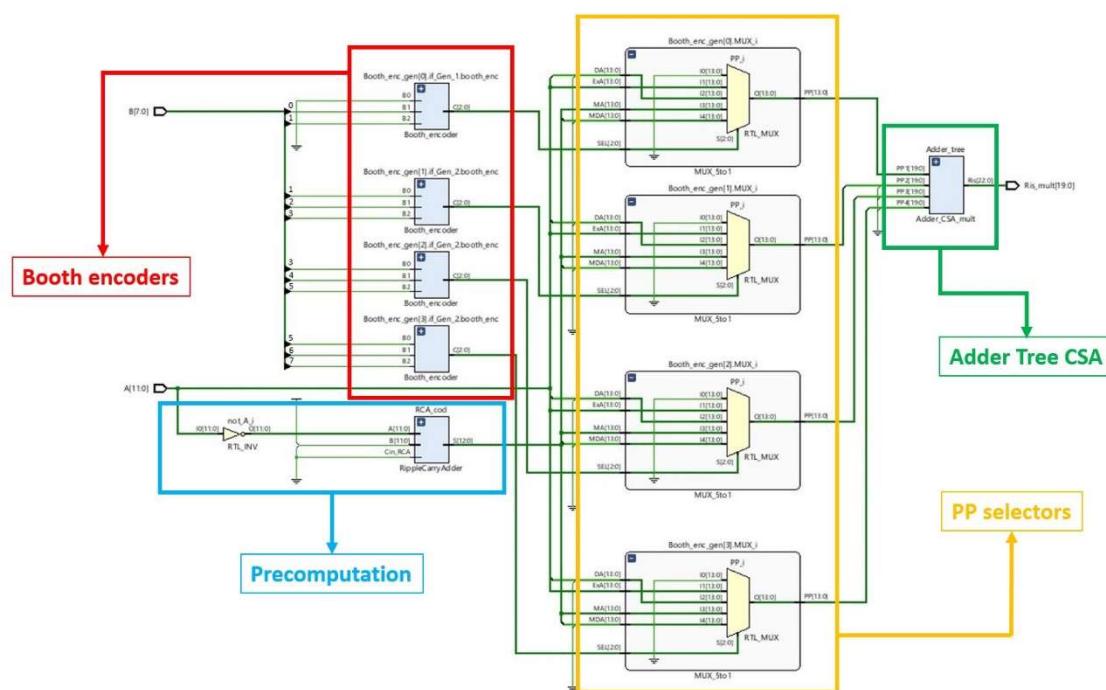


Figura 27 Schematic del moltiplicatore di Booth

### 2.5.2.1. Precomputation module

Questo blocco serve per calcolare, in via preliminare, tutti i prodotti parziali che possono essere associati alla generica terna del moltiplicatore, in modo da averli già pronti ed evitare di doverli determinare in fase di processamento, il che costituirebbe motivo di ulteriore rallentamento nei tempi di calcolo oltre a quello imposto di default dalle piste di interconnessione. In particolare, considerando la tabella illustrata precedentemente in **Figura 26**, vi sono cinque possibili valori per i prodotti parziali: 0, A, 2A, -2A e -A, dove A indica il moltiplicando, per cui una sequenza di 12 bit. Ovviamente però, nella progettazione di tale modulo, abbiamo dovuto tenere in considerazione due punti fondamentali:

- 1) Per calcolare quantità come -A e -2A, è necessario usare un sommatore, dal momento che entrambe vengono determinate calcolandone il complemento a due, il quale sappiamo essere dato dalla somma tra l'unità e la negazione del moltiplicando stesso (per tale motivo vi sono un Ripple Carry Adder e una porta “NOT”);
- 2) 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 il valore -2A in maniera corretta, è necessario aggiungere due bit. Di conseguenza, tutti i prodotti parziali calcolati dovranno essere maggiorati di due bit, ovvero rappresentati su 14 bit.

### 2.5.2.2. Booth encoders

Questi componenti, complessivamente in quattro, uno per ogni terna del moltiplicatore, hanno il compito di determinare la codifica da associare alla generica terna, la quale però, piuttosto che essere rappresentata come digit (lavoriamo in binario e non in decimale), viene indicata come una sequenza di 3 bit, con il MSB indicante il segno e i due meno significativi che quantificano il modulo della codifica, determinata come mostra la tabella di **Figura 28**.

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

Figura 28 Tabella delle codifiche per il moltiplicatore di Booth

### 2.5.2.3. Partial product selectors

Questi selettori sono dei semplici multiplexer 5 a 1 che, sulla base della codifica prodotta dal Booth encoder, selezionano e portano in uscita il prodotto parziale relativo a quella codifica, che verrà inviato in ingresso all'albero di somma finale.

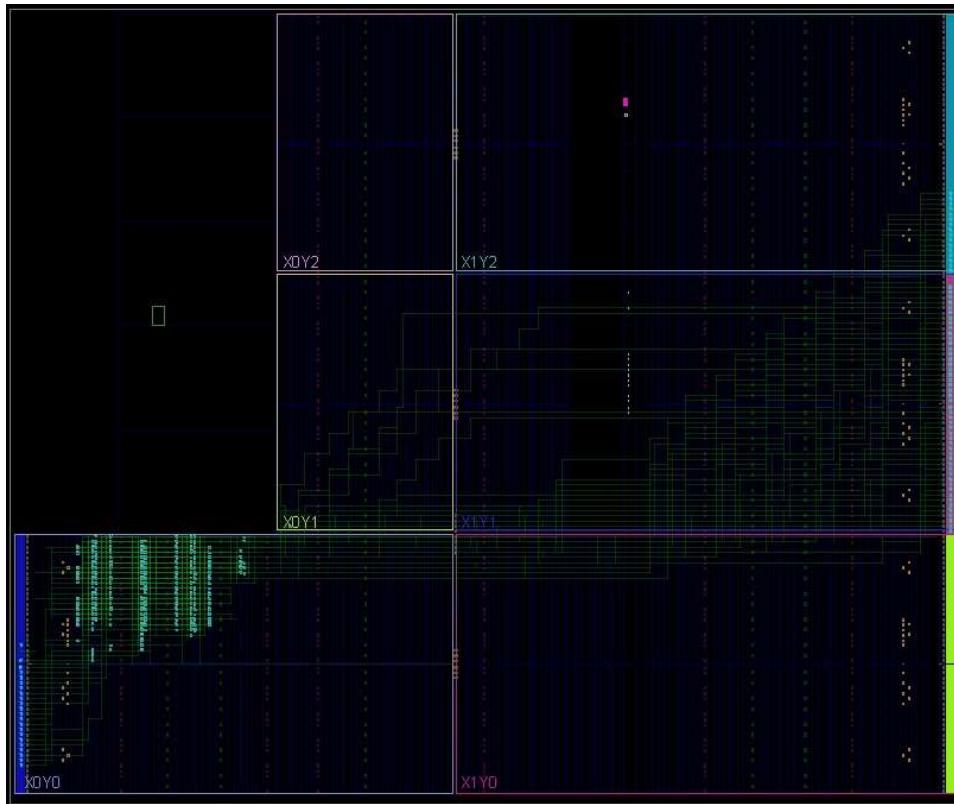
### 2.5.2.4. Adder tree a quattro operandi con approccio Carry Save Adder

A proposito dell'albero di somma finale, bisogna spendere qualche parola in più, per via di alcuni accorgimenti su cui è stato necessario porre attenzione. Innanzitutto, tenendo in considerazione che,

in un ipotetico calcolo su carta, bisogna shiftare il generico prodotto parziale verso sinistra di un numero di posizioni pari al peso della terna alla quale esso è associato, i vari prodotti parziali non possiamo lasciarli a 14 bit, poiché, altrimenti non sarebbero allineati. Per questo motivo è stato necessario, prima di tutto, uniformarli alla stessa lunghezza (estendendoli con segno e aggiungendo degli zeri nelle posizioni meno significative), in modo da poterli sommare correttamente. In particolare, essendo che essi si shiftano verso sinistra, gli uni con gli altri, di due posizioni man mano che ci spostiamo verso pesi più elevati, abbiamo una lunghezza complessiva delle sequenze che li caratterizzano di 20 bit, per cui, è necessario l'impiego di un albero di somma a quattro operandi con 20 bit ciascuno. Dal momento che, però, le sequenze sono abbastanza lunghe, si è preferito realizzare l'Adder Tree finale utilizzando l'approccio del Carry Save Adder, dal momento che offre la possibilità di calcolare la stessa somma ma con un ritardo minore rispetto a quello introdotto da un albero di RCA. In particolare, essendo che l'albero di somma aggiunge un bit ad ogni livello e che gli operandi in ingresso sono a 20 bit, il risultato prodotto sarebbe rappresentato su 23 bit. In realtà però, essendo che, per definizione, il prodotto tra due numeri restituisce un valore rappresentato su un numero di bit dato dalla somma di quello degli operandi in ingresso, il risultato ottenuto si può tranquillamente troncare a 20 bit, risparmiando, di conseguenza ulteriore tempo di calcolo per i circuiti a valle.

### 3. Implementazione

La porzione di chip occupata dalle risorse utilizzate nell'operazione di filtraggio è rappresentata in **Figura 29**.



*Figura 29 Posizione delle risorse utilizzate dal circuito progettato nell'FPGA*

Attraverso il processo di *placing & routing*, il sintetizzatore posiziona tutte le risorse nella parte in basso a sinistra del chip (nella sezione X0Y0). Dall'immagine si può notare che le piste di interconnessione che collegano le uscite del circuito con i pad I/O in output sono molto lunghe, motivo che fa pensare che la maggior parte del ritardo nella produzione dei risultati sia da attribuire ad esse. In particolare, il path critico risulta essere quello che va dal pin C del terzo registro presente nel modulo di registri di pipeline che porta le uscite dei Pre-Adders ai moltiplicatori, fino al pin D di quello presente nel modulo di registri di pipeline che porta le uscite dei moltiplicatori in ingresso all'ultimo albero di somma, come mostra la seguente immagine. A tale path è associato un ritardo totale di 7.693 ns, corrispondente ad uno slack di 0.176 ns (WNS), dei quali il 25% dovuto alla logica combinatoria e il restante 75% alle interconnessioni.

Name	Slack ^ 1	Levels	High Fanout	From	To	Total Delay	Logic Delay	Net Delay
Path 1	0.176	10	24	Pipeline_Register_module1/Reg3/Q_reg[0]/C	Pipeline_Register_module2/Reg3/Q_reg[17]/D	7.693	1.924	5.769

*Figura 30 Path critico del circuito implementato*

Relativamente alla latenza del circuito, essa è intesa come il tempo necessario a fare in modo che, dal momento in cui esso inizia a lavorare, venga prodotto il primo risultato valido in uscita. Considerando che nel nostro caso l'istante in cui inizia il flusso di elaborazione corrisponde al momento in cui i pixel iniziano ad essere letti, la latenza complessiva del circuito è pari a 134 cicli di clock, corrispondente alla somma tra il tempo necessario a far sì che il primo pixel, una volta letto, arrivi in posizione centrale nella finestra da processare e quello che deve trascorrere prima di ottenere il relativo pixel filtrato, che corrisponde a 3 cicli di clock, dal momento che vi sono tre registri di

pipeline tra i vari moduli del circuito. Tenendo conto, però, che dopo l'implementazione vi sono i ritardi associati alle piste di interconnessione, è stata calcolata una latenza complessiva pari a 135 cicli di clock.

Per quanto riguarda il throughput, invece, indica il numero di risultati che il circuito riesce a produrre in un periodo di clock. Da quanto appena detto quindi, possiamo affermare che il throughput è pari a 1 poiché, a partire da quando termina la latenza del circuito, viene prodotto un pixel filtrato ad ogni colpo di clock.

## 4. Simulazione

Terminata la fase di progettazione e implementazione, si è passati alla simulazione del circuito ottenuto, al fine di analizzarne il comportamento in un intervallo di tempo che permettesse di testarlo in maniera adeguata. Per fare ciò, è stata analizzata, in prima battuta, la simulazione a livello behavioral, per poi passare, soltanto in un secondo momento, a quella post-implementation. Il filtro di riferimento per l'analisi è un laplaciano di tipo 1, i cui coefficienti sono, a partire da quello centrale,  $8, -1, -1, 0, 0, 0$ . Premesso che i componenti più importanti, quali buffer line, sommatori e moltiplicatori, sono stati testati in maniera esaustiva, ci concentriamo, in questa sezione, ad analizzare solo il comportamento del circuito complessivo. Inoltre, tenendo conto del fatto che, per ovvie ragioni, non è possibile riportare l'intero testbench in questa relazione, ci limitiamo a descriverne soltanto i punti salienti, basandoci su un approccio secondo il quale verranno illustrate delle waveforms, estrapolate da Vivado, e, per ognuna di esse, verrà fornita una descrizione di ciò che accade nel circuito.

### 4.1. Behavioral Simulation

Questa simulazione è stata utilizzata come strumento per comprendere se il circuito, ancor prima di implementarlo su chip, funzionasse a dovere oppure no, al fine di rilevare più agevolmente eventuali errori di progettazione e poterli opportunamente correggere.

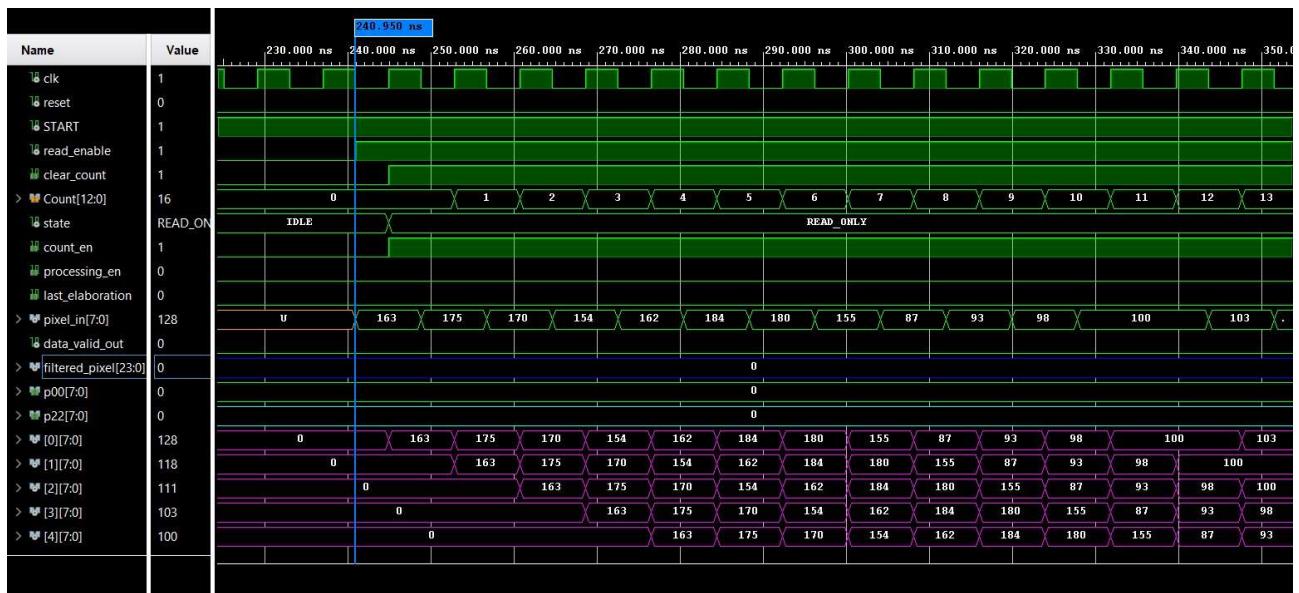


Figura 31 Primo estratto della simulazione behavioral del circuito progettato

In **Figura 31** viene illustrato ciò che accade quando l'immagine inizia ad essere letta dall'esterno pixel per pixel, riga per riga (l'istante preciso è indicato dal marker blu). Ovviamente, dovendo attendere un periodo di tempo pari alla latenza in ingresso (bisogna aspettare che si crei la prima finestra utile), i pixel non possono essere filtrati sin da subito; di conseguenza, mentre scorrono nel buffer line, una cui parte è rappresentata dai segnali di colore fucsia, essi non vengono portati in uscita, poiché il segnale **processing\_en** è basso. Ciò viene dimostrato dal fatto che i dati **p00** e **p22**, corrispondenti rispettivamente all'uscita del registro in alto a sinistra e in posizione centrale della finestra, restano ad un valore nullo (questo contribuisce alla riduzione della potenza dissipata). Inoltre, possiamo notare che il primo pixel viene letto quando il conteggio è ancora nullo, come già anticipato nella descrizione degli stati della macchina di controllo.

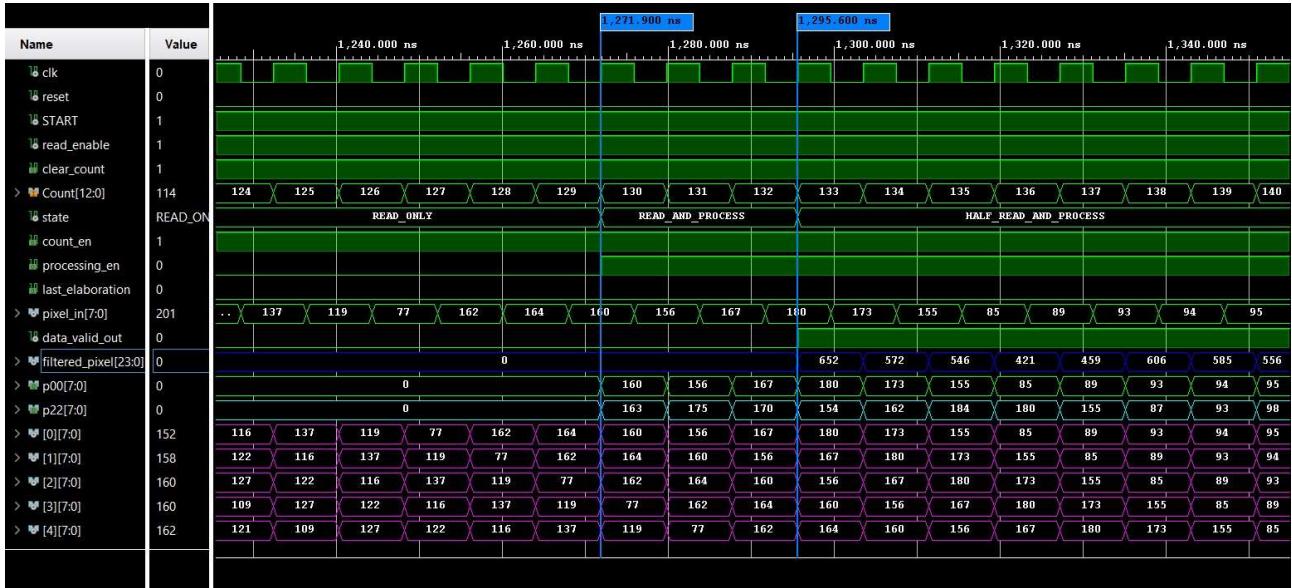


Figura 32 Secondo estratto della simulazione behavioral del circuito progettato

In **Figura 32** viene illustrata la situazione in cui i pixel iniziano ad essere processati uno per volta. Infatti, quando il primo arriva in posizione centrale, ovvero il segnale **p22** assume il valore del primo pixel letto, il **processing\_en** transita al valore logico alto. Da notare che la prima uscita valida si ha dopo tre cicli di clock dal momento in cui inizia il processing, dopo di che viene prodotto un pixel filtrato ad ogni colpo di clock e questa condizione si mantiene tale fino a quando non viene processato l'ultimo.



Figura 33 Terzo estratto della simulazione behavioral del circuito progettato

In **Figura 33** vengono rappresentati gli ultimi istanti di lettura; infatti, quando il valore del conteggio è pari a 4095, l'ultimo pixel entra nel circuito. Notiamo, inoltre, che, non appena esso viene letto, dal colpo di clock immediatamente successivo iniziano ad entrare degli zeri nel Buffer Line, segno del fatto che non solo esso inizia a svuotarsi, ma che anche viene applicata l'estensione ai bordi con zero padding per il processamento degli ultimi pixel dell'immagine. Da questo momento in poi quindi, smettiamo di leggere, ma continuiamo semplicemente a processare i pixel restanti.



Figura 34 Quarto estratto della simulazione behavioral del circuito progettato

In **Figura 34** abbiamo una rappresentazione dell'ultima parte importante della simulazione. Innanzitutto, notiamo che quando il conteggio arriva a 4225 il processamento si ferma; ciò accade nonostante non siano ancora stati prodotti tutti i pixel filtrati in uscita poiché, come si può notare dall'immagine, mancano ancora gli ultimi tre. In realtà però, è stato comunque possibile fare ciò in quanto, essendo che il segnale di abilitazione del processamento serve solo per consentire o meno che i pixel vengano portati in uscita dal Buffer Line, basta che esso sia alto soltanto quando abbiamo l'ultimo pixel nella posizione centrale per garantire che esso venga elaborato lo stesso, anche a processamento disabilitato. D'altronde però, questo passaggio si è ritenuto comunque necessario, poiché se non fosse stato così, sarebbero stati processati tre pixel in più che, in realtà, non esistono. Dunque, dal momento che quando il segnale **processing\_en** si abbassa dobbiamo aspettare altri tre cicli di clock prima di dichiarare terminato il flusso di elaborazione, viene utilizzato il segnale **last\_elaboration** per indicare che la trasmissione dei pixel in uscita ancora non è terminata. Infatti, questo segnale si alza quando viene processato l'ultimo pixel e si abbassa solo quando il corrispondente viene prodotto in uscita dal circuito.

#### 4.2. Post-Implementation Simulation

Una volta aver analizzato la simulazione behavioral, si è passati all'analisi di quella post-implementation, per verificare semplicemente che il circuito funzionasse correttamente una volta implementato sul chip. Sostanzialmente ciò che accade nelle due simulazioni è molto simile, poiché l'operazione svolta è la stessa; l'unica cosa che cambia è il fatto che in quella post-implementation entrano in gioco i ritardi associati alle piste di interconnessione, per cui le transizioni dei segnali avvengono con un certo ritardo rispetto al fronte sensibile del clock. In **Figura 35**, infatti, possiamo notare che, rispetto a quest'ultimo, i vari pixel filtrati vengono prodotti con un ritardo pari a poco più della metà del periodo del clock stesso; ovvero, essendo quest'ultimo uguale a 7.9 ns, che corrisponde al minimo valore tale per cui sono soddisfatti sia i constraint temporali che il funzionamento corretto del circuito, il ritardo di produzione delle uscite è pari a circa 4.4 ns. Ovviamente notiamo la presenza di glitch, ma sono comunque innocui dal momento che avvengono lontanamente dai fronti sensibili del clock.

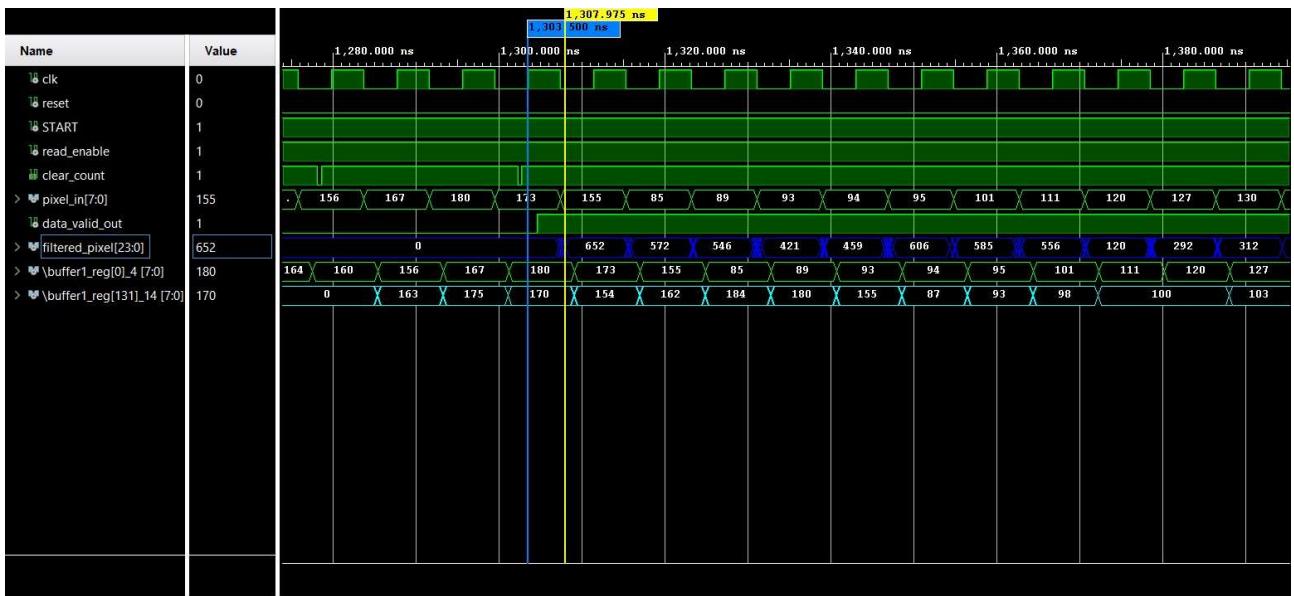


Figura 35 Primo estratto della simulazione post-implementation del circuito progettato

In particolare, guardando l'immagine, ci si può accorgere del fatto che il segnale `data_valid_out` si alza prima che venga prodotto il primo pixel filtrato; il motivo di tale cosa è dovuto al fatto che, mentre questo segnale è un'uscita della macchina di controllo e per determinare il suo valore basta solo verificare una condizione, la produzione dei pixel filtrati viene da una serie di operazioni di somma e moltiplicazioni che richiedono sicuramente più tempo di calcolo. Questo dà anche l'idea dell'effetto introdotto dai ritardi di propagazione nel circuito fisico. Tuttavia, questa situazione non costituisce nessun problema particolare, poiché lo stesso segnale `data_valid_out`, dal momento che si alza dopo circa 1.1 ns dal fronte sensibile del clock, risulta essere alto per la prima volta in corrispondenza del fronte che produce in uscita il primo pixel filtrato, per cui non c'è rischio che i dati vengano considerati validi a partire dallo 0.

Una situazione simile si ha al termine del processamento dei pixel, riportato in **Figura 36**.



Figura 36 Secondo estratto della simulazione post-implementation del circuito progettato

In pratica, il segnale `data_valid_out` si abbassa quando ancora in uscita vi è il valore 137, associato al risultato dell'elaborazione sull'ultimo pixel, per cui sembrerebbe che non viene considerato valido.

In realtà, invece, ciò che accade è proprio il contrario, poiché, come mostra la seguente figura, il segnale di valid sull'uscita si abbassa pochi picosecondi dopo il fronte del clock, per cui è tale che il 137 venga comunque considerato valido, e quindi, un'eventuale circuiteria a valle del circuito progettato lo riuscirà a ricevere senza problemi.

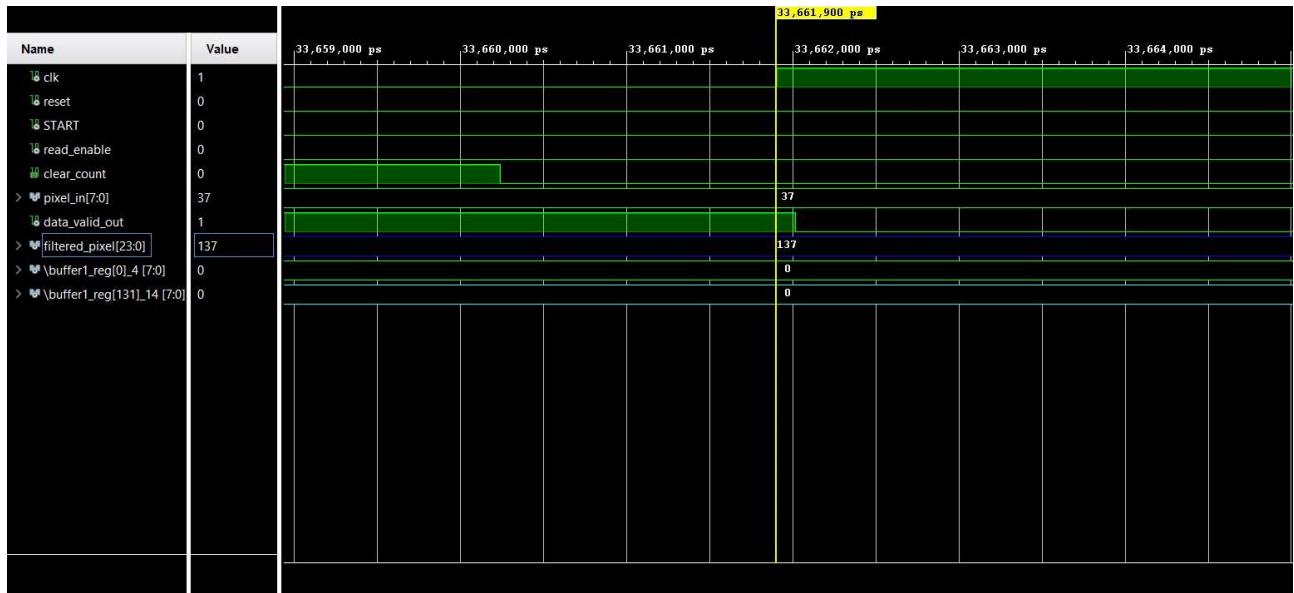


Figura 37 Terzo estratto della simulazione post-implementation del circuito progettato

## 5. Report

### 5.1. Report di timing

Per completare la fase di implementazione, è stato impostato un constraint sul clock, al fine di determinare quale fosse la massima frequenza alla quale il circuito riesce a lavorare bene. Le quantità su cui ci si deve basare per determinarla sono il tempo di setup (indica quanto prima del fronte sensibile del clock un dato deve essere valido e stabile) e il tempo di hold (indica per quanto tempo dopo il fronte sensibile del clock il dato deve rimanere stabile), rappresentati rispettivamente dai parametri WNS (Worst Negative Slack) e WHS (Worst Hold Slack). In particolare, è garantito che i vincoli temporali imposti siano rispettati fintanto che tali parametri si mantengono a valori positivi. Impostando diversi constraint temporali, però, si è arrivati alla conclusione che il periodo di clock che garantisce il corretto funzionamento del circuito è pari a 7.9 ns, poiché per valori più bassi, nonostante si ottengano WNS e WHS positivi, alcune operazioni di filtraggio non vanno a buon fine. Di conseguenza, è stato impostato un constraint sul clock pari a 7.9 ns, col quale sono stati ottenuti i seguenti valori:

Setup	Hold	Pulse Width
Worst Negative Slack (WNS): 0.176 ns	Worst Hold Slack (WHS): 0.107 ns	Worst Pulse Width Slack (WPWS): 2.970 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: 582	Total Number of Endpoints: 582	Total Number of Endpoints: 574

Figura 38 Report sul timing ottenuto con un constraint sul clock di 7.9 ns

Sulle considerazioni fatte, si può affermare che la frequenza di funzionamento garantita, ovvero la minima frequenza alla quale si può desumere il corretto funzionamento del circuito, corrisponde a:

$$f_{garantita} = \frac{1}{t_{constraint}} = 126.58 \text{ MHz}$$

Considerando inoltre che il periodo di clock si definisce come la somma tra il periodo corrispondente alla massima frequenza di funzionamento e quello di setup, abbiamo:

$$t_{max} = T_{clk} - t_{setup} = (7.9 - 0.176)ns = 7.724 ns$$

La frequenza massima di funzionamento, quindi, sulla base del constraint applicato, equivale a:

$$f_{max} = \frac{1}{t_{max}} = 129.46 \text{ MHz}$$

### 5.2. Report di utilizzazione delle risorse

In merito all'utilizzo delle risorse, si può effettuare un'analisi sulla base del numero di LUT e di registri utilizzati. Nel chip su cui è stato implementato il circuito vi sono complessivamente 53200 LUT e 106400 Registri. Riguardo alle prime, ne vengono utilizzate 1146, corrispondenti all'2.15% di quelle totali, mentre relativamente ai secondi, ne vengono impiegati 509, corrispondenti invece allo 0.48% di quelli totali. La seguente immagine, tratta dal file *utilization\_report.txt*, riporta tale informazione.

Site Type	Used	Fixed	Prohibited	Available	Util%
Slice LUTs	1146	0	0	53200	2.15
LUT as Logic	1082	0	0	53200	2.03
LUT as Memory	64	0	0	17400	0.37
LUT as Distributed RAM	0	0	0	0	0
LUT as Shift Register	64	0	0	0	0
Slice Registers	509	0	0	106400	0.48
Register as Flip Flop	509	0	0	106400	0.48
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

Figura 39 Report di utilizzo delle risorse

Entrando più nel dettaglio, di tutte le 1146 LUT, 1082 vengono usate per operazioni di logica combinatoria, mentre le restanti come registri a scorrimento. In particolare, queste ultime, come dimostra la seguente immagine, vengono impiegate tutte nel modulo del Buffer Line. Ciò significa che evidentemente, l'azione di scorrimento dei pixel da un registro all'altro è stata in parte tradotta dal sintetizzatore come uno shift-register.

Name	1	Slice LUTs (53200)	Slice Registers (106400)	Slice (13300)	LUT as Logic (53200)	LUT as Memory (17400)
FilterCircuit		1146	509	358	1082	64
buffer_line (BufferLine)		282	291	128	218	64
Final_Pipeline_Register_no_sat (Registro_parameterized4)		0	24	17	0	0
FSM_Counter (FSM_and_Counter)		27	18	11	27	0
Counter (Counter_13bit)		16	13	6	16	0
Finite_state_machine (FSM)		11	5	9	11	0
Pipeline_Register_module1 (Reg_Module_AddMult)		444	59	188	444	0
Reg1 (Registro)		86	10	38	86	0
Reg2 (Registro_5)		71	11	36	71	0
Reg3 (Registro_6)		81	10	34	81	0
Reg4 (Registro_7)		82	10	37	82	0
Reg5 (Registro_8)		88	10	33	88	0
Reg6 (Registro_9)		36	8	23	36	0
Pipeline_Register_module2 (Reg_Module_MultAdd)		394	117	177	394	0
Reg1 (Registro_parameterized2)		69	20	50	69	0
Reg2 (Registro_parameterized2_0)		54	20	40	54	0
Reg3 (Registro_parameterized2_1)		133	20	66	133	0
Reg4 (Registro_parameterized2_2)		43	20	35	43	0
Reg5 (Registro_parameterized2_3)		22	20	24	22	0
Reg6 (Registro_parameterized2_4)		75	17	49	75	0

Figura 40 Distribuzione dell'utilizzo delle risorse nel Buffer Line

Riguardo, invece, ai registri, si può notare che tutti quelli utilizzati vengono usati come dei Flip Flop e vengono impiegati nel Buffer Line, nei registri di pipeline, nella FSM e nel contatore.

### 5.3. Report di dissipazione di potenza senza file .saif

Ogni circuito elettronico digitale richiede una certa quantità di potenza per funzionare. La potenza dissipata cresce linearmente con la frequenza di funzionamento del sistema. Pertanto, per circuiti che lavorano a maggiori frequenze e, quindi, a maggiori velocità di funzionamento, corrisponde una potenza dissipata superiore. In generale, la potenza dissipata di un circuito elettronico digitale può

essere di due tipologie: potenza statica dissipata e potenza dinamica dissipata. La prima corrisponde alla potenza dissipata in condizioni in cui l'uscita del circuito si trova ad un valore costante, mentre la seconda corrisponde alla potenza dissipata durante le commutazioni dell'uscita.

La Vivado Design Suite, dopo aver effettuato l'implementazione del circuito elettronico digitale progettato sul dispositivo di riferimento, permette di analizzare la Total On-Chip Power. Essa è descritta dalla seguente relazione:

$$\text{Total On - Chip Power} = \text{Static} + \text{Design Dynamic}$$

dove *Static* è ottenuta tramite la seguente somma:

$$\text{Static} = \text{Device Static} + \text{Design Static}$$

Nello specifico la *Device Static* indica la potenza di leakage del transistor quando il dispositivo è alimentato ma non configurato, la *Design Static* indica la potenza quando il dispositivo è stato configurato e non vi è alcuna attività di commutazione e, infine, la *Design Dynamic* indica la potenza media derivante dalla logica e dall'attività di commutazione.

Inoltre, la suite permette di analizzare ulteriori parametri fondamentali quali la *Junction Temperature*, il *Thermal Margin* e l'*Effective θJA*.

Nello specifico è possibile notare come, dopo aver applicato un constraint relativo al clock, la dissipazione di potenza del circuito è relativamente bassa. Inoltre, la temperatura di giunzione risulta essere inferiore del margine di temperatura.

Power	
<b>Total On-Chip Power:</b>	<b>0.161 W</b>
<b>Junction Temperature:</b>	<b>26.9 °C</b>
Thermal Margin:	58.1 °C (4.9 W)
Effective θJA:	11.5 °C/W
Power supplied to off-chip devices:	0 W
Confidence level:	Low

Figura 41 Report di dissipazione di potenza

Nello specifico, si può notare come il maggior contributo di potenza dinamica dissipata è dato dalle risorse di I/O (0.027W).

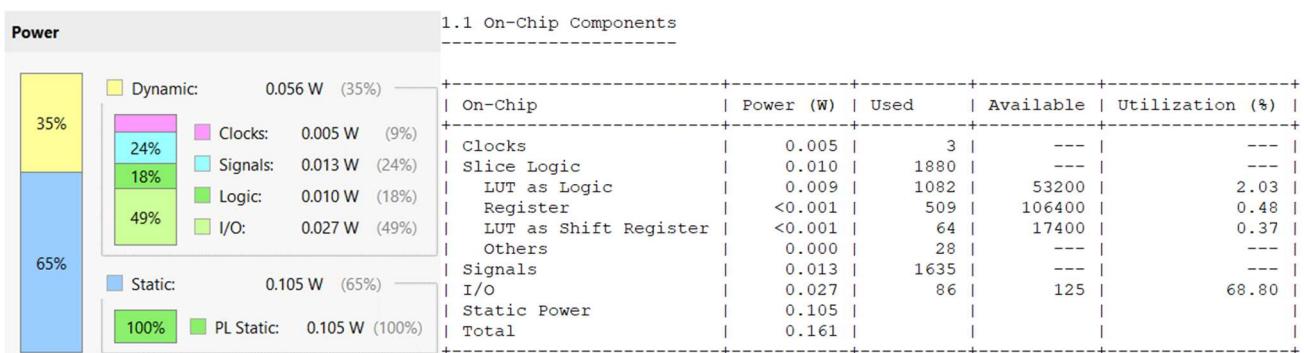


Figura 43 Report di dissipazione di potenza

Figura 42 Report di dissipazione di potenza per ogni risorsa

Dalla Figura 42 si evince come il contributo di dissipazione di potenza dato dalle risorse di I/O sia dovuto al fatto che la maggior parte delle componenti sono occupate (68.80%). Dal momento che le

percentuali di utilizzazione di risorse rispetto a quelle occupate dei *Register* e delle *LUT as Shift Registers* è davvero irrigoria, la dissipazione di potenza risulta essere inferiore a  $0.001W$ .

#### 5.4. Report di dissipazione di potenza con file .saif

Analizzando nello specifico il report di dissipazione di potenza, si può notare come il *confidence level* risulta essere basso. Bisogna ricordare che il *livello di confidenza* rappresenta l'affidabilità del report in questione, cioè una misura dell'accuratezza e della completezza dei dati di input che il Report Power utilizza durante l'esecuzione dell'analisi di potenza. Per far fronte a tale problema, si genera un file *.saif* che viene utilizzato per 'guidare' ulteriormente l'algoritmo di analisi della potenza. Quindi, si specifica la temporizzazione corrispondente al runtime della Post-Implementation Simulation e si spuntano le caselle relative al log di tutti i segnali considerati nella simulazione all'interno della finestra *Settings>Simulation* del tool:

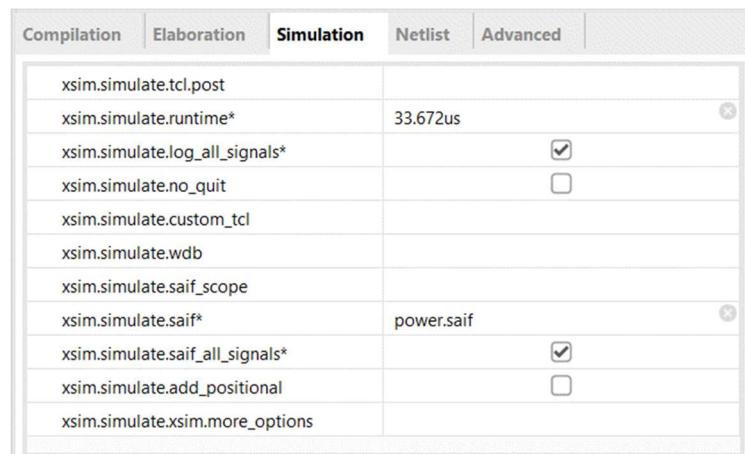


Figura 44 Generazione del file .saif

Pertanto, si effettua nuovamente la simulazione (Post-Implementation Simulation) e, successivamente, si aggiunge, tramite il Flow Navigator nella sezione **Implementation>Report Power**, il file *.saif* generato e salvato nella directory *sim/sim\_1/impl/timing/xsim*.

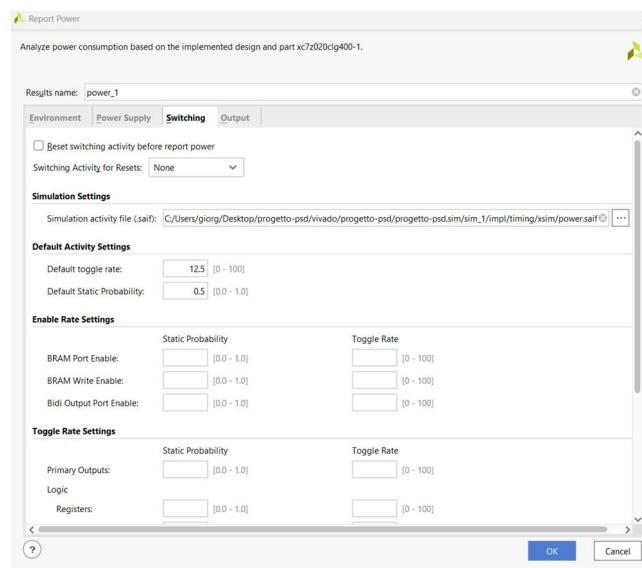


Figura 45 Aggiunta del file .saif nel Report Power

Dopo aver aggiunto il file *.saif*, si può notare come il report di dissipazione di potenza sia cambiato:

**Summary**

Power analysis from Implemented netlist. Activity derived from constraints files, simulation files or vectorless analysis.

<b>Total On-Chip Power:</b>	<b>0.176 W</b>
<b>Design Power Budget:</b>	<b>Not Specified</b>
<b>Power Budget Margin:</b>	<b>N/A</b>
<b>Junction Temperature:</b>	<b>27.0°C</b>
Thermal Margin:	58.0°C (4.9 W)
Effective $\theta_{JA}$ :	11.5°C/W
Power supplied to off-chip devices:	0 W
Confidence level:	<b>High</b>

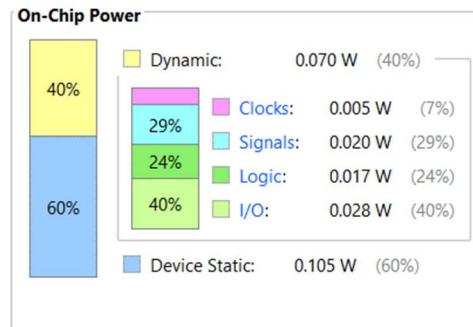


Figura 46 Report di dissipazione di potenza considerando il file .saif

È possibile notare un *High Confidence Level* e, inoltre, una maggiore dissipazione di potenza. Pertanto, considerando il file .saif, si può concludere affermando che la *Total On-Chip Power* effettiva sia pari a  $0.176W$  ed la temperatura di giunzione sia pari a  $27.0^{\circ}C$ .

## 6. Testing e Validazione

Il testing è un processo attraverso il quale un progetto viene sottoposto affinché sia verificato il suo corretto funzionamento. Tale processo è utilizzato per identificare eventuali bug, errori o problemi di funzionamento ed è alla base di qualsiasi progetto hardware-software.

Il processo di testing può essere suddiviso in diverse fasi, tra cui il testing unitario, il testing di sistema ed il testing finale. Nello specifico, il testing unitario si concentra sul funzionamento dei singoli moduli del sistema, il testing di sistema verifica come i componenti interagiscono tra loro ed il testing finale verifica che il sistema, hardware in questo caso, soddisfi le specifiche richieste.

In merito al testing unitario, esso è stato effettuato per ogni modulo progettato (sommatore, albero di somma, moltiplicatore, *buffer line*, *FSM*, contatore). Inoltre, ogni testing è stato effettuato secondo una logica esaustiva, cioè verificando ogni possibile input per il modulo considerato.

Invece, per quanto riguarda il testing finale, esso è stato implementato tramite procedure descritte in *MATLAB*. Nello specifico, è stato concretizzato considerando diversi script e funzioni che hanno permesso di implementare un testing completo del circuito progettato.

### 6.1. Lettura dei pixel dell'immagine da filtrare

La lettura dei pixel dell'immagine in input al circuito progettato è stata gestita tramite la *function* nativa ‘*imread*’ che riceve direttamente in input l’immagine in questione presente nella directory ‘input’. La matrice corrispondente viene acceduta successivamente tramite lo script ‘*read\_gray\_scale\_image.m*’ che permette di salvare il contenuto di tale struttura dati in un array monodimensionale di dimensione  $1 \times 4096$  (essendo il numero di righe e di colonne della matrice entrambi pari a 64) ed in seguito salvato su un file .txt denominato ‘*gray\_scale\_image\_matrix*’. Ovviamente lo stato di accesso a tale file viene monitorato ed eventualmente, in caso di un errore durante la fase di scrittura o di un’eventuale chiusura anomala, viene stampato un messaggio di errore nella Console Window.

### 6.2. Inizializzazione del filtro isotropico e scelta dei coefficienti

Successivamente, viene inizializzata la matrice dei coefficienti del filtro isotropico considerando la dimensione  $k = 5$  ipotizzata da progetto. Pertanto, tramite Console Window, viene chiesto all’utente la scelta di un filtro isotropico, tra quelli proposti, per l’operazione di filtraggio ad alto livello da eseguire. Ovviamente, tale scelta viene ‘guidata’ dalla Console così da effettivamente rendere consapevole l’utente del filtro scelto. Proprio per questo motivo, al momento dell’invocazione della *function* in questione, viene mostrata una matrice rappresentante la struttura del filtro isotropico all’interno della quale è possibile osservare gli *ID* dei coefficienti che caratterizzano il filtro ( $W$ ,  $W_1$ ,  $W_2$ ,  $W_3$ ,  $W_4$  e  $W_5$ ). Invece, in Console Window saranno mostrati diversi filtri isotropici con i corrispondenti valori dei coefficienti associati. Quindi, l’utente sarà in grado di constatare l’effettiva struttura del filtro comprendente i valori scelti. Tanto è vero che, non appena l’utente avrà inserito l’*ID*, corrispondente al filtro scelto, e avrà premuto *INVIO*, lo script genererà automaticamente una nuova *figure* rappresentante il filtro isotropico scelto.

```

File gray_scale_image_matrix.txt written correctly.
SCEGLIERE UNA TIPOLOGIA DI FILTRO ISOTROPICO TRA QUELLE INDICATE:
- filtro gaussiano ~ g
  W5=0; W4=0; W3=-1; W2=-1; W1=-2; W=16;

- filtro mediano ~ m
  W5=0; W4=0; W3=0; W2=0; W1=0; W=1;

- filtro laplaciano 1 ~ 11
  W5=0; W4=0; W3=0; W2=-1; W1=-1; W=8;

- filtro laplaciano 2 ~ 12
  W5=0; W4=0; W3=0; W2=0; W1=1; W=-4;

- filtro laplaciano 3 ~ 13
  W5=0; W4=0; W3=0; W2=0; W1=-1; W=4;
N.B.: per scegliere la tipologia, indicare l'ID presente dopo il '~'

```

**fx** Inserire la tipologia di filtro isotropico scelta:

Figura 47 Scelta della tipologia di filtro isotropico da utilizzare per il testing

### 6.3. Generazione della finestra di processing

Per quanto riguarda la generazione della finestra di processing, rivolta all'uso del testing effettuato tramite l'ambiente di *MATLAB*, è stato descritto un algoritmo nel corrispondente linguaggio al fine di emulare il funzionamento ad alto livello del circuito di filtraggio progettato. Più specificatamente, è stata descritta una procedura che simulasse la struttura di convoluzione secondo l'architettura trattata.

Nella parte iniziale dello script sono state dichiarate alcune variabili tramite il costrutto '*global*' così da essere successivamente utilizzate rapidamente in altri script e *function* presenti in cartelle differenti dalla corrente. Inoltre, sono state inizializzate alcune variabili come, ad esempio, il parametro '*k*' (dimensione del filtro) ed il raggio del filtro isotropico, la finestra tramite il costrutto '*zeros(nr\_righe, nr\_colonne)*', il buffer *FIFO* tramite il costrutto appena citato e, infine, il buffer line descritto in alcuni passi dell'algoritmo che sono stati riportati successivamente a tale introduzione.

Bisogna specificare che il funzionamento principale dell'algoritmo è basato sul costrutto '*for*' che determina la creazione delle finestre di filtraggio fin quando non sono terminate le iterazioni di calcolo consentite. Si può notare come il massimo numero di iterazioni consentite è pari alla somma tra il prodotto tra le due dimensioni (numero di righe e numero di colonne della matrice rappresentante l'immagine originale) e la latenza relativa al filtraggio del primo pixel dell'immagine in questione. Pertanto, il valore corrispondente al numero di iterazioni massime risulta essere pari a 4226.

Qui di seguito sono riportati i passi per la descrizione dell'algoritmo appena proposto:

- creazione della struttura buffer line: implementazione come lista di 261 elementi (inizialmente tutti nulli) dalla quale, ad ogni colpo di clock, viene eliminato l'elemento in coda e viene inserito l'elemento corrente in testa;

- creazione della finestra di convoluzione di dimensione  $5 \times 5$  (matrice che viene stampata nella *Console Window*) ad ogni colpo di clock selezionando le posizioni dalla lista buffer line;
- calcolo del risultato effettuando il prodotto termine per termine tra la finestra corrente e la matrice relativa al kernel;
- salvataggio del risultato corrente in un file .txt e stampa nella *Console Window*;

Infine, tramite la *function* ‘vector2img.m’ è stato possibile ricostruire, considerando i valori ottenuti tramite la procedura precedentemente descritta (pertanto, i pixel filtrati), l’immagine filtrata considerando un filtro isotropico di dimensione  $5 \times 5$ .

```

clock: 4225
    66    96    95    48    23
    108   103    56    44    28
    116    75    31    37     0
        0      0      0      0      0
        0      0      0      0      0

filtered_pixel: -67
*****
*****
```

```

clock: 4226
    96    95    48    23    57
    103   56    44    28    47
    75    31    37     0     0
        0      0      0      0      0
        0      0      0      0      0

filtered_pixel: 137
*****
*****
```

*Figura 48 Ultime 2 finestre di processing generate*

#### 6.4. Waiting dell’output del circuito progettato

Essendo stato generato l’output del filtraggio tramite procedura ad alto livello, bisogna, ora, confrontarlo con l’output del circuito progettato. A questo proposito, nella Console Window verrà visualizzato un messaggio nel quale verrà specificato di effettuare la simulazione (nello specifico la Post-Implementation Simulation) all’interno dell’ambiente di sviluppo di Vivado. Tanto è vero che nel file di *testbench* è stato specificato un processo di scrittura tramite il quale è possibile scrivere all’interno del file ‘*filtered\_pixels\_project.txt*’ (presente nella directory ‘*output*’) i pixel filtrati dal circuito progettato. Dopo aver effettuato la simulazione in questione, sarà possibile cliccare *INVIO* e procedere con la fase di testing del circuito.

Eseguire testbench presente in Vivado affinché i pixel filtrati dell’immagine siano disponibili.  
fx Premere INVIO se la Post-Implementation Simulation è stata effettuata per almeno 33.672 us:

*Figura 49 Waiting nella Console Window di MATLAB dell’output del circuito progettato*

## 6.5. Lettura dell'output generato dal circuito progettato

Dopo aver cliccato *INVIO* all'interno della Console Window, viene effettuato l'accesso in lettura al file '*filtered\_pixels\_project.txt*' nel quale è stato scritto, durante la fase di simulazione in Vivado, l'output del circuito progettato. Pertanto, tali valori verranno salvati in un array monodimensionale  $1 \times 4096$  che sarà in seguito utile per l'analisi dell'errore ed in una matrice  $64 \times 64$  che rappresenterà l'immagine filtrata dal circuito progettato.

## 6.6. Comparazione dei risultati ottenuti

Dopo aver salvato i risultati ottenuti tramite circuito nelle strutture dati appropriate, viene effettuata una comparazione *visiva* tramite *figure* tra l'immagine filtrata tramite la procedura ad alto livello descritta precedentemente e l'immagine filtrata tramite il circuito progettato.

## 6.7. Analisi dell'errore

In merito all'analisi dell'errore tra i pixel filtrati tramite la procedura ad alto livello e pixel filtrati tramite il circuito progettato, sono stati considerati tre tipologie di analisi di errore differenti: l'errore assoluto, il parametro *PSNR* ed il parametro *SSIM*.

### 6.7.1. Analisi tramite errore assoluto

L'errore assoluto è definito come la differenza tra il valore misurato (in questo caso i pixel filtrati tramite il circuito progettato) ed il valore esatto (i pixel filtrati tramite procedura ad alto livello).

In questo caso esso è stato calcolato effettuando una differenza in valore assoluto tra le strutture dati '*filtered\_pixels\_project*' e '*filtered\_pixels\_procedure*' e, successivamente, rappresentato in una *figure* insieme alla comparazione delle immagini citata precedentemente.



*Figura 50 Confronto tra l'immagine filtrata tramite procedura ad alto livello e immagine filtrata tramite circuito progettato (filtro isotropico Laplaciano di tipologia 1)*

Si può notare come, considerando un filtro isotropico Laplaciano di tipologia 1, l'immagine filtrata dal circuito progettato corrisponda perfettamente all'immagine filtrata dalla procedura ad alto livello descritta in *MATLAB*. Tanto è vero che l'errore assoluto calcolato risulta essere pari a zero per ogni pixel considerato. Tale errore, inoltre, risulta essere nullo sia nel caso di una simulazione comportamentale del circuito (*Behavioral Simulation*) sia nel caso di una simulazione post-implementazione (*Post-Implementation Simulation*). Bisogna precisare che, l'immagine filtrata presenta una bassa qualità rispetto all'immagine originale ma questo è dovuto anche alla scelta del filtro isotropico. Infatti, la qualità dell'immagine filtrata dipende fortemente dal filtro isotropico scelto e, pertanto, dall'obiettivo che l'utente si predilige come operazione di filtraggio.

### 6.7.2. Analisi tramite PSNR

Il *Peak Signal-to-Noise Ratio* (anche conosciuto con la denominazione *PSNR*) è un parametro utilizzato per valutare la qualità di un’immagine filtrata rispetto a quella originale. Maggiore è il valore del *PSNR* maggiore è la ‘somiglianza’ con l’immagine originale. Nello specifico, tale parametro è espresso dalla seguente relazione:

$$PSNR = 20 \cdot \log_{10} \left( \frac{MAX\{I\}}{\sqrt{MSE}} \right)$$

dove  $MAX\{I\}$  indica il massimo valore possibile dei pixel dell’immagine originale  $I$  mentre  $MSE$  indica l’errore quadratico medio ed è espresso dalla seguente relazione:

$$MSE = \frac{1}{M \cdot N} \sum_{i=0}^{M-1} \sum_{j=0}^{N-1} \|I(i,j) - K(i,j)\|^2$$

dove  $K$  indica l’immagine filtrata,  $M$  ed  $N$  indicano rispettivamente il numero di righe e di colonne delle immagini (dato che le dimensioni devono rispettare le relazioni  $M_I = M_K$  e  $N_I = N_K$ ).

L’analisi dell’errore basata su *PSNR* è spesso adottata per valutare la qualità del filtraggio. In questo caso essa è stata implementata tramite la *function* nativa in *MATLAB* denominata *psnr.m*. Tale procedura prevede in input due immagini: l’immagine candidata, cioè l’immagine filtrata, ed un’immagine di riferimento, cioè l’immagine originale. Nello specifico tale *function* restituisce due valori: *peaksnr*, cioè il *PSNR* in *decibel*, e l’S*NR*, cioè il *Signal-to-Noise Ratio* in *decibel*. Nello specifico quest’ultimo parametro indica il rapporto segnale-rumore, cioè mette in relazione la potenza del segnale effettivo a quella del rumore. L’S*NR* può essere, pertanto, espresso dalla seguente relazione:

$$SNR = \frac{P_{segnale}}{P_{rumore}} \quad , \quad 0 \leq SNR \leq \infty$$

Più specificatamente, un valore di *PSNR* che tende ad infinito esprime un’estrema somiglianza tra l’immagine originale e l’immagine filtrata.

### 6.7.3. Analisi tramite SSIM

La Structural Similarity Index Measure (anche conosciuta con la denominazione *SSIM*) è un ulteriore parametro che descrive la qualità di filtraggio considerando l’immagine originale e l’immagine filtrata. A differenza dei parametri basati sull’errore assoluto come *PSNR* e *SNR*, l’S*SIM* si basa sull’analisi strutturale. Quest’ultima indica lo studio delle informazioni relative alla struttura degli oggetti nella scena visiva dell’immagine.

Tale parametro misura effettivamente la differenza percettiva tra le due immagini. Esso non può giudicare quale tra le due immagini è la migliore ma ovviamente questo deve essere “dedotto” sapendo a priori quale immagine è quella originale e quale è quella filtrata, cioè sottoposta all’elaborazione. Ciò significa che l’S*SIM* misura effettivamente la somiglianza strutturale tra le due immagini e non si limita piuttosto ad una differenza pixel per pixel.

Ogni calcolo relativo alla somiglianza strutturale è basato su ogni pixel dell’immagine ed è possibile descriverlo dalla seguente relazione:

$$SSIM(i,j) = \frac{(2 \cdot \mu_i \cdot \mu_j + c_1)(2 \cdot \sigma_i \cdot \sigma_j + c_2)(cov_{i,j} + c_3)}{(\mu_i^2 + \mu_j^2 + c_1)(\sigma_i^2 + \sigma_j^2 + c_2)(\sigma_i \cdot \sigma_j + c_3)}$$

dove:

- $\mu_i$  è la media campionaria rispetto ai pixel della riga  $i$ ;
- $\mu_j$  è la media campionaria rispetto ai pixel della colonna  $j$ ;
- $\sigma_i^2$  è la varianza rispetto ai pixel della riga  $i$ ;
- $\sigma_j^2$  è la varianza rispetto ai pixel della colonna  $j$ ;
- $cov_{i,j}$  è la covarianza rispetto alla riga  $i$  e alla colonna  $j$ ;
- $c_1, c_2$  e  $c_3$  sono tre variabili utilizzate per stabilizzare la divisione quando il denominatore tende a valori troppo bassi. Nello specifico  $c_1 = (k_1 \cdot L)^2$ ,  $c_2 = (k_2 \cdot L)^2$  e  $c_3 = \frac{c_2}{2}$  dove  $k_1 = 0.01$  e  $k_2 = 0.03$  sono parametri adimensionali ed  $L$  rappresenta la dinamica dei valori dei pixel, ovvero esso risulta essere pari a 255 nel caso di immagini codificate su 8 bit. Tanto è vero che quest'ultimo parametro è calcolato tramite la seguente relazione:

$$L = 2^{\#bits\_per\_pixel} - 1.$$

Bisogna precisare che la somiglianza strutturale varia in un intervallo di valori compreso tra 0 ed 1 tale che l'immagine filtrata perfettamente corrispondente all'immagine originale si ha in corrispondenza di un  $SSIM = 1$ .

Un parametro correlato all' $SSIM$  è la dissomiglianza strutturale  $DSSIM$ . Essa è una metrica che deriva dalla somiglianza strutturale ed è espressa dalla seguente relazione:

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

Ovviamente, in questo caso un filtraggio perfetto, cioè dove l'immagine filtrata corrisponde perfettamente all'immagine originale, si ha in corrispondenza di una dissomiglianza strutturale pari a 0. Pertanto, se un buon filtraggio corrisponde ad un valore di somiglianza strutturale  $SSIM$  che tende ad 1 allora il valore di dissomiglianza strutturale  $DSSIM$  deve tendere a 0.

#### 6.7.4. Analisi dei valori di PSNR, SSIM e DSSIM

Eseguendo le function ‘psnr\_calculation.m’ e ‘ssim\_calculation.m’ è possibile analizzare i risultati ottenuti relativi al  $PSNR$ ,  $SSIM$  e  $DSSIM$ .

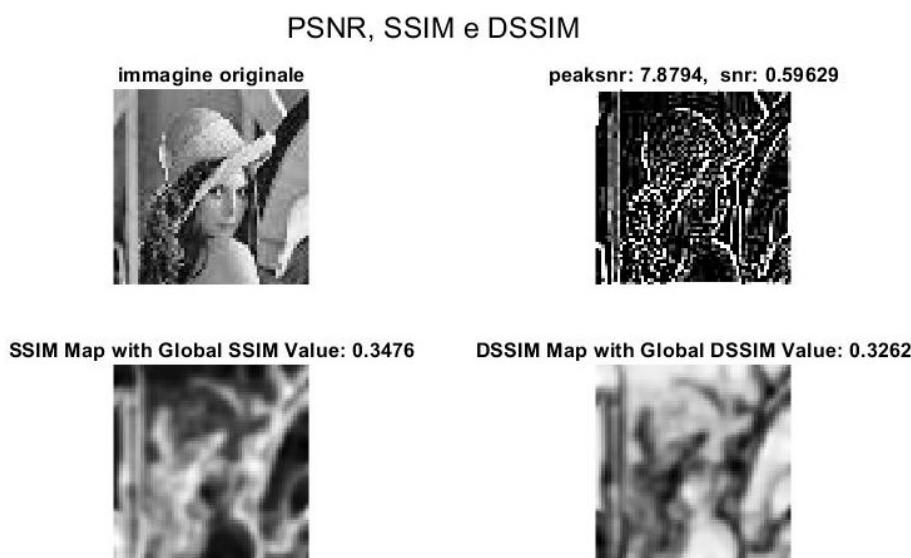


Figura 51 Analisi del PSNR, SSIM e DSSIM

Si può notare come il *Peak Signal-to-Noise Ratio* risulta essere relativamente basso. Tale risultato è la conferma di come l'immagine filtrata sia di qualità drasticamente più bassa rispetto a quella originale. Ovviamente il risultato ottenuto dipende dal filtro isotropico scelto inizialmente. In questo caso, trattandosi di un filtro isotropico Laplaciano, l'obiettivo era quello di enfatizzare i bordi dell'immagine, cioè effettuare un *image sharpening*. Si può evincere come un'enfatizzazione pronunciata dei bordi e, quindi, uno sharpening (nitidezza) pronunciato dell'immagine, può portare ad una degradazione della qualità stessa. Questo risultato si può analizzare anche tramite i parametri di somiglianza e dissomiglianza strutturale. Tanto è vero che il *DSSIM* risulta essere pari a 0.3262, cioè tra l'immagine originale e quella filtrata è presente una modesta dissomiglianza strutturale.

### ERRORE ASSOLUTO e DSSIM tra l'immagine originale e l'immagine filtrata

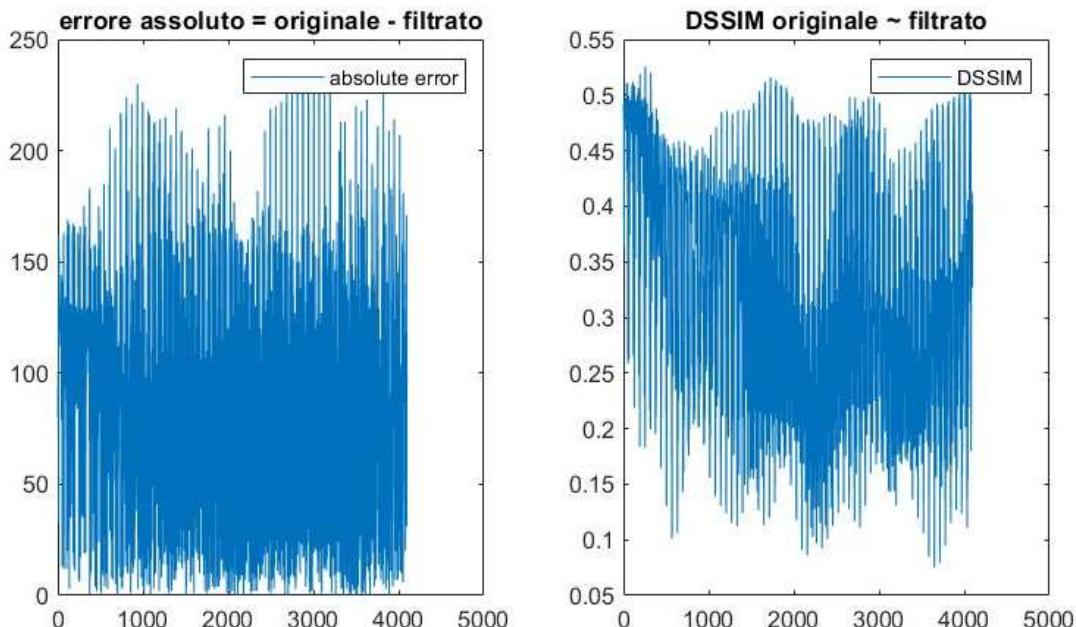


Figura 52 Analisi dell'errore assoluto e del DSSIM tra l'immagine originale e l'immagine filtrata

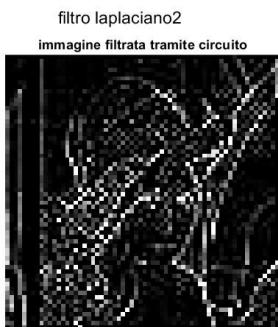
Si può notare come l'errore assoluto e la dissomiglianza strutturale siano entrambi relativamente modesti. Si evince da questa ulteriore considerazione come il filtro isotropico scelto degradi la qualità dell'immagine originale a causa dell'effetto di sharpening troppo pronunciato.

## 7. Conclusioni

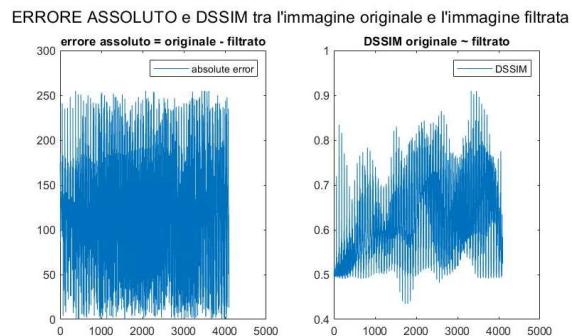
In conclusione, si può evincere come l'Image processing sia una branca in continua evoluzione, soprattutto per i sistemi di visione artificiale, e volta a convertire immagini in un insieme di informazioni digitali che vengono elaborate al fine di migliorare la qualità dell'immagine finale e ottenere quindi informazioni più dettagliate, con livelli di affidabilità anche abbastanza elevati. Il grado di evoluzione di questa disciplina, se così si può chiamare, ha subito e sta subendo tutt'ora un incremento notevole, motivo per cui si è sempre alla ricerca di nuove soluzioni che diano la possibilità di ottimizzare il connubio tra efficienza e velocità di calcolo nell'elaborazione. Basti pensare all'utilità che sistemi di questo tipo hanno in campo medico, militare e automotive, soprattutto se si pensa a quello che sarà il futuro della società mondiale.

Proprio per questo motivo, l'obiettivo del progetto è stato quello di analizzare i risultati ottenuti tramite una struttura progettata nei minimi dettagli. In particolare, sono stati analizzati parametri, fondamentali nell'Image processing, che hanno permesso di comprendere la qualità effettiva del filtraggio eseguito.

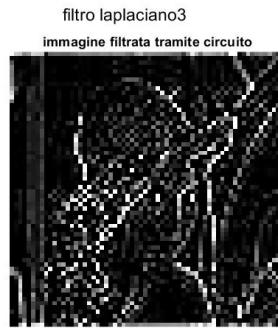
In particolare, per quanto riguarda l'operazione di filtraggio analizzata, si è notato come l'applicazione del filtro Laplaciano di tipologia 1 produca un'immagine filtrata di scarsa qualità rispetto all'immagine originale. Se, invece, si decidesse di considerare le altre due tipologie di filtri isotropici Laplaciani proposti, l'Image processing corrispondente presenterebbe un'elevata dissomiglianza strutturale e, inoltre, un elevato errore assoluto tra l'immagine originale e l'immagine filtrata dal circuito progettato.



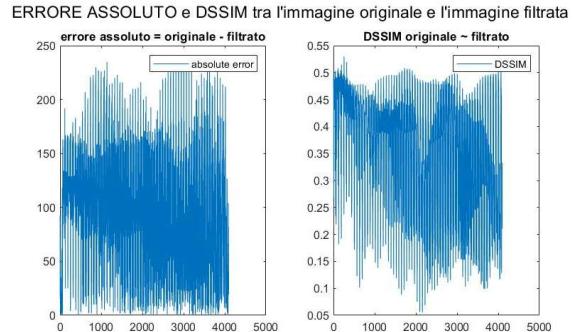
*Figura 56 Immagine filtrata tramite filtro isotropico Laplaciano di tipologia 2*



*Figura 56 Errore assoluto e dissomiglianza strutturale tra l'immagine originale e l'immagine filtrata tramite filtro isotropico Laplaciano di tipologia 2*

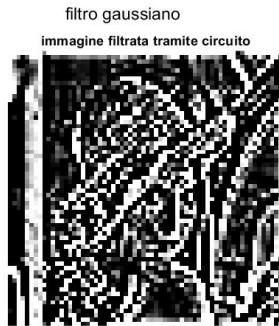


*Figura 56 Immagine filtrata tramite filtro isotropico Laplaciano di tipologia 3*

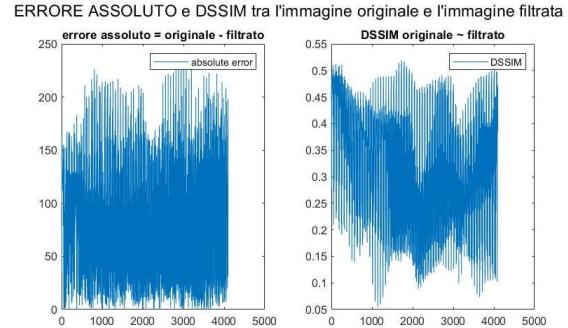


*Figura 56 Errore assoluto e dissomiglianza strutturale tra l'immagine originale e l'immagine filtrata tramite filtro isotropico Laplaciano di tipologia 3*

Per quanto riguarda, invece, l'operazione di filtraggio tramite filtro isotropico Gaussiano, anch'essa presenta un errore assoluto e una dissomiglianza strutturale non indifferenti.



*Figura 60 Immagine filtrata tramite filtro isotropico Gaussiano*

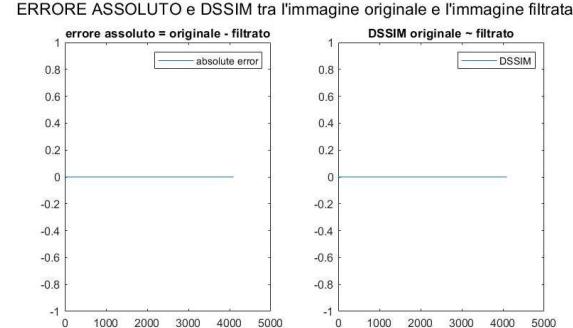


*Figura 60 Errore assoluto e dissomiglianza strutturale tra l'immagine originale e l'immagine filtrata tramite filtro isotropico Gaussiano*

Infine, analizzando l’operazione di filtraggio tramite filtro isotropico mediano, si può notare come essa presenti un errore assoluto e una dissomiglianza strutturale tra l’immagine originale e l’immagine filtrata pari a 0. Pertanto, si evince che tale filtro esegue un image processing riproducendo alla perfezione l’immagine originale considerata.



*Figura 58 Immagine filtrata tramite filtro isotropico mediano*



*Figura 58 Errore assoluto e dissomiglianza strutturale tra l’immagine originale e l’immagine filtrata tramite filtro isotropico mediano*