UNIVERSITÀ DELLA CALABRIA
DIPARTIMENTO DI
INGEGNERIA INFORMATICA,
MODELLISTICA, ELETTRONICA
E SISTEMISTICA
DIMES

Report

Progettazione, implementazione e simulazione su Vivado di un Convolutore SIMD 3x3

Corso di Progettazione Sistemi Digitali

Corso di Laurea Magistrale in Ingegneria Elettronica

Unical, aa 2020/2021

Andrea Alecce Matricola 214611

Matteo Cannistrà 216735

Prof.sa S. Perri

Sommario

1. Intro

1.1. Analisi del problema

Obiettivo della relazione è quello di illustrare la progettazione di un circuito digitale per l'elaborazione delle immagini implementando il paradigma SIMD.

È richiesto, come specifica progettuale, l'utilizzo di un kernel 3×3 con coefficienti a 4-bit rappresentati in complemento a due (valori compresi tra -8 e 7), sia capace di operare secondo il paradigma SIMD per filtrare porzioni di immagini composte da almeno 3×64 pixels rappresentati in scala di grigi a 8-bit oppure a 16-bit. L'uscita è a 32 bit. Nel caso di parallelismo 2, i pixels in ingresso sono a 8-bit ottenendo due pixels filtrati, a 16-bit ciascuno. Nel caso di parallelismo 1, i pixels in ingresso sono a 16-bit ed il circuito produce un pixel filtrato a 24-bit.

Nel progetto, è stato utilizzando lo stesso kernel in entrambe le modalità su una immagine campione di dimensione 64x64 pixel a 16 bit in scala di grigi. I risultati del filtraggio sono stati successivamente confrontati con quelli ottenuti tramite tool software (Matlab), per validarne il funzionamento. Infine, sono stati riportati i report relativi al timing, potenza e le risorse utilizzate dal modulo hardware realizzato.

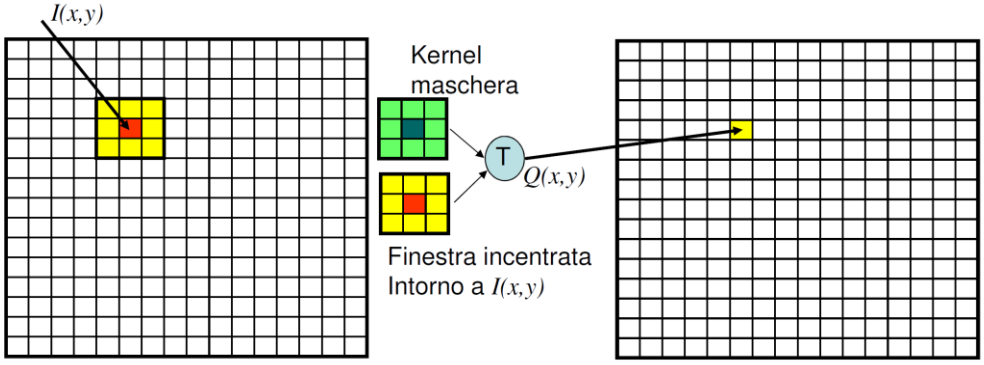
Il progetto è stato realizzato selezionando la scheda ***Nexys4*** DDR (*xc7a100tcsg324-1*) della famiglia ***Artix-7*** e simulato su ***Vivado*** nella versione *2017.4*.

1.2. Filtraggio nel dominio dello spazio

Data una immagine di dimensione *nxm* pixel "P" ad *i* bit e un filtro di dimensione *kxk* di elementi "W" (weight, pesi) ad *i* bit con ***anchor point*** centrale, il generico pixel filtrato è dato dalla convoluzione discreta dei corrispettivi coefficienti, che si traduce in operazioni di somma di prodotti. Nel caso di kernel 3x3 si ottiene il generico pixel filtrato Q tramite:

$$Q_{(i,j)} = \sum_{\substack{k=-1,0,1 \\ h=-1,0,1}} P_{(k+i,h+j)} \cdot W_{(k+1,h+1)}$$

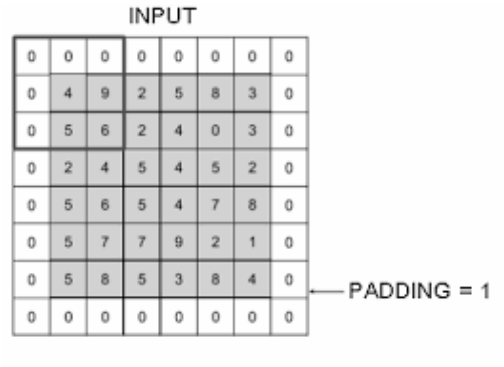
Il filtraggio consiste nell'andare a sovrapporre per ogni elemento della matrice di pixel dell'immagine la finestra del kernel, rispettando l'anchor point centrale. Ogni elemento della finestra verrà moltiplicato con l'elemento corrispettivo della matrice di pixel. I prodotti ottenuti verranno successivamente sommati, per ottenere il generico pixel filtrato, nella stessa posizione del pixel di partenza.



Per sua natura, l'immagine discretizzata in pixel presenta, nelle operazioni di filtraggio, il ***problema ai bordi***. Sovrapponendo la finestra di filtraggio lungo i bordi, alcune posizioni non saranno contenute all'interno della matrice. Per ovviare a questo problema, ci sono alcune tecniche:

- ***Zero-Padding***;
- ***Estensione a specchio***;
- ***Estensione toroidale***.

La scelta tra queste tecniche è arbitraria e dipende dal tipo di applicazione. In generale, si preferisce lo zero-padding in quanto gli altri due richiedono accessi a valori molto lontani in memoria, rallentando il sistema. Nella figura, un esempio di implementazione dello zero padding su una immagine 6x6.

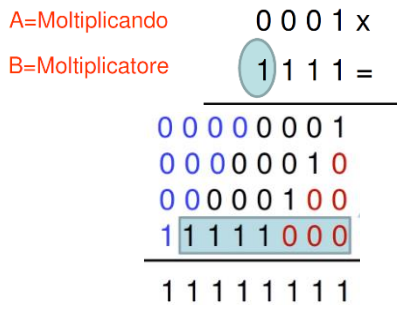


Il risultato della convoluzione di una immagine ad n bit ed un kernel kxk con coefficienti a *w* bit sarà rappresentato in uscita con dimensione

$$n + w + \log_2 k^2$$

Dove *n+w* rappresenta il prodotto tra pixel e coefficienti del kernel, mentre il termine k^2 il numero di prodotti da sommare.

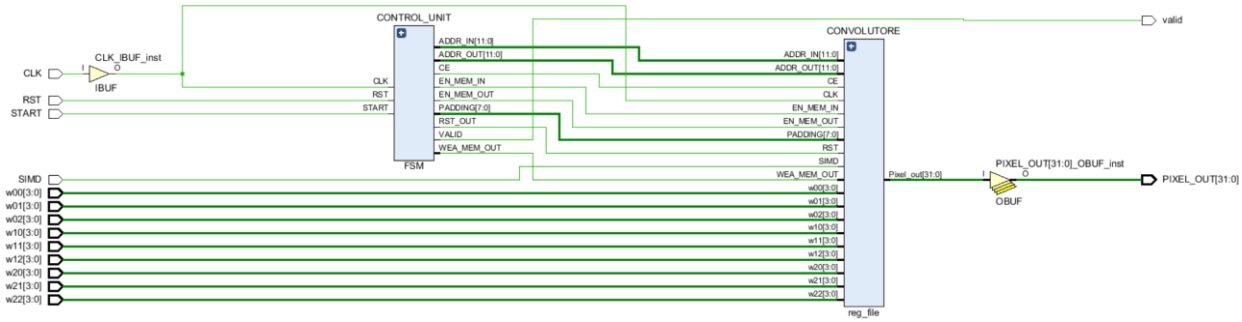
I pixel dell'immagine utilizzano la rappresentazione *unsigned,* mentre i coefficienti del filtro sono rappresentati in complemento a 2. Di conseguenza, per eseguirne i prodotti bisogna tenere in considerazione il segno, dettato dai bit più significativi dei corrispettivi coefficienti del kernel. Il generico prodotto parziale, dato dal prodotto bit a bit, sarà posizionato in funzione del peso degli operandi, come illustrato di seguito. Inoltre, è necessario estendere opportunamente i prodotti parziali in base alla rappresentazione usata.



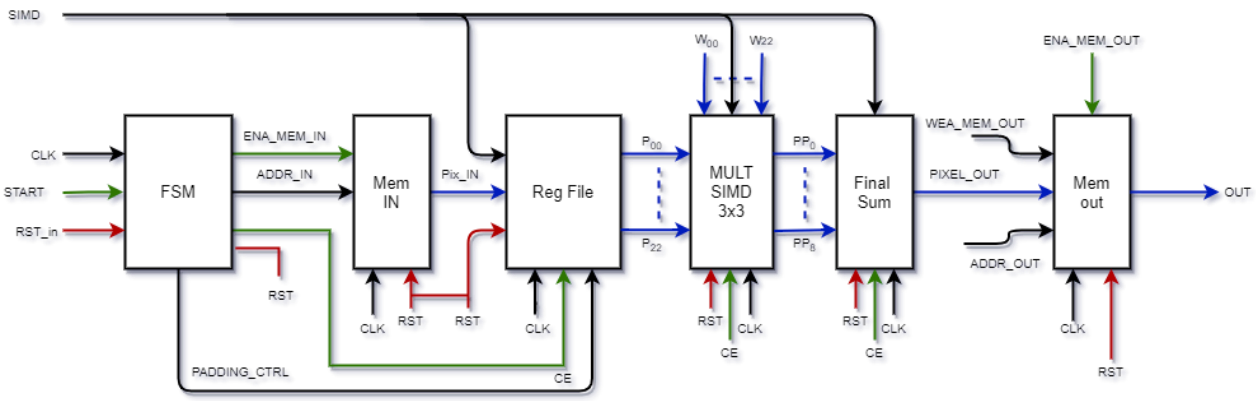
Per implementare l'operazione di somma rispettando la rappresentazione, bisogna estendere l'ultimo prodotto parziale in funzione del segno del moltiplicatore. Considerando il numero di addendi, è necessario utilizzare una struttura ***Adder Tree*** per sommare tutti i prodotti parziali. Anche nel caso di somma si estende con segno l'operando.

2. Implementazione hardware

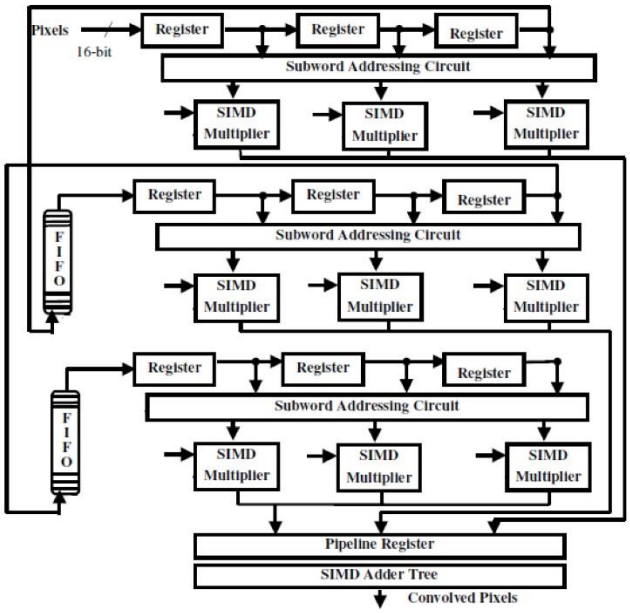
Per eseguire l'operazione di convoluzione dell'immagine 64x64 a 16 bit con un filtro 3x3 a 4 bit in complemento a due usando il paradigma SIMD, l'architettura realizzata è la seguente.



L'architettura Top-Level si compone di un modulo ***Control Unit*** (FSM) e un ***Convolutore***. FSM (Finite State Machine) regola le operazioni e produce i segnali controllo e di validità.



La struttura del *Convolutore* è realizzata come illustrato in seguito:

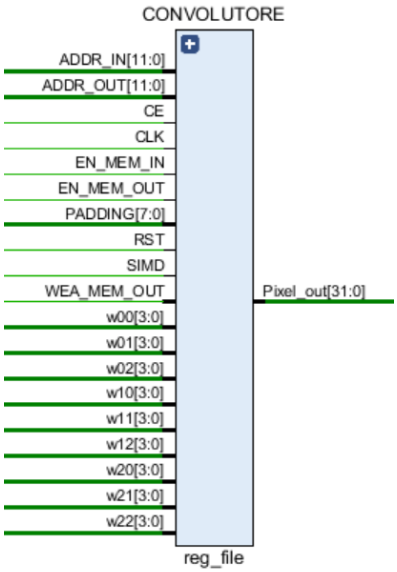


I moduli utilizzati sono i seguenti:

- ***Memoria SRAM*** Single Port 16x4096 bits di ingresso, in cui è memorizzata l'immagine in formato .COE da filtrare;
- Un ***Register File*** composto da 9 registri a 16 bit e due FIFO di profondità 61;
- ***Subword Addressing Circuit*** formato da una serie di multiplexer 2x1 ad 8 bit per gestire il padding e il SIMD;
- ***Moltiplicatore SIMD***;
- ***Adder Tree SIMD***;
- ***Memoria SRAM*** Single Port 32x4096 bits di uscita, in cui si memorizzato i pixel convoluti;

2.1. Convolutore:

Il modulo Convolutore è interessato dei seguenti segnali di ingresso e uscita:



- ADDR_IN[11:0]: indirizzi dato della memoria di ingresso;
- ADDR_OUT[11:0]: indirizzi dato della memoria di uscita;
- CE: clock enable;
- EN_MEM_IN: enable memoria di ingresso;
- EN_MEM_OUT: enable memoria di uscita;
- PADDING[7:0]: segnali di configurazione per lo zero-padding;
- WEA_MEM_OUT: abilitazione della scrittura sulla memoria di uscita;
- RST: segnale di reset;

prodotti dalla FSM, e:

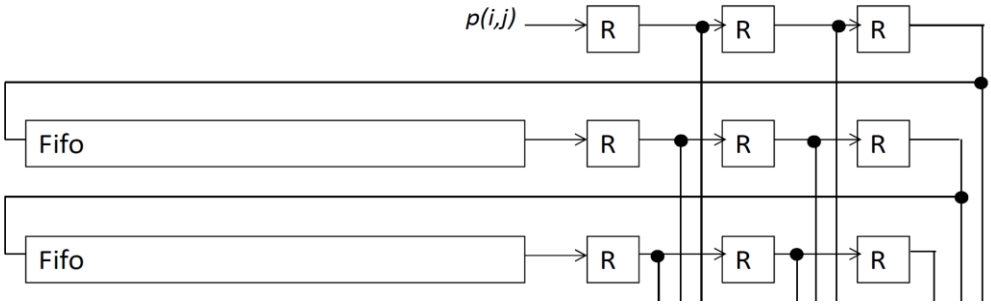
- CLK: clock;
- W[3:0]: valore dei 9 coefficienti del filtro;
- SIMD: segnale per il parallelismo;

2.1.1. Register File

L'immagine è memorizzata all'interno della memoria SRAM, la quale riceve l'indirizzo a 12 bit (per rappresentare i 4096 elementi della matrice) relativo al dato 16bit da leggere, fornendolo in uscita dopo una latenza pari a 2.

Il Write Enable della SRAM e il dato in ingresso (Dina) sono settati a zero: questo perché l'immagine è precaricata in memoria, non è dunque richiesta l'operazione di scrittura. Dopo i due colpi di clock iniziali, verrà letto un dato alla volta in *raster order* ad ogni colpo di clock, rendendolo disponibile al circuito di elaborazione.

Al fine di realizzare la convoluzione, è stata implementata una struttura di bufferizzazione: sono stati istanziati 9 registri per formare la finestra e due FIFO per far scorrere i pixel dell'immagine utili per le operazioni successive. Le FIFO devono essere dimensionate considerando la dimensione delle colonne dell'immagine. In generale, se l'immagine rappresentata in n righe e m colonne ed il filtro è di dimensione k, il numero di FIFO sarà *k-1* di profondità *m-k*. Un esempio di questa struttura è riportata in figura.



La lettura dell'immagine dalla memoria avviene riga per riga, in raster-order: ad ogni colpo di clock un nuovo pixel viene inserito nella struttura buffer, facendo scorrere gli altri. Per questo motivo, la finestra di convoluzione per come ideata inizialmente risulterà ribaltata.

W00	W01	W02
W10	W11	W12
W20	W21	W22

W22	W21	W20
W12	W11	W10
W02	W01	W00

Tuttavia, affinché si venga a creare la prima finestra utile per il filtraggio per ottenere il primo dato valido, è necessario attendere che il primo pixel si porti in posizione centrale nella finestra dei registri. Ciò avviene dopo una latenza pari a 67, ricavata da:

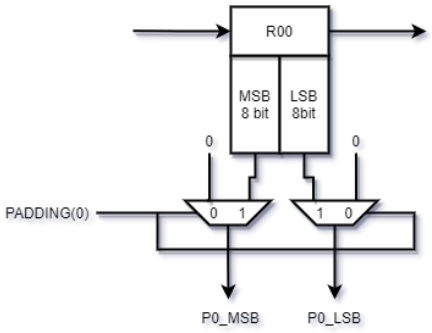
$$latency = [(ncol + r) * r] + ReadLatency$$

Dove *ncol*=64 è il numero di colonne dell'immagine, *r*=1 è il raggio del filtro e ReadLatency=2 è il ritardo dovuta alla memoria in lettura.

Il primo dato valido in uscita si avrà dopo la latenza dovuta al circuito convolutore, che esegue la somma dei prodotti (pari a 11). I risultati sono salvati in una memoria SRAM di uscita, la cui latenza in lettura è anch'essa pari a 2. La latenza complessiva affinché si ottenga il primo dato valido in uscita dal sistema, a partire da quando il primo pixel dell'immagine si trova in posizione centrale nella pila di registri, è pari a 13.

2.1.2. Subword Addressing Circuit: zero-padding

Il *Subword Addressing Circuit* viene implementato per la gestione del **padding** e del **SIMD**. Questo modulo si compone di alcuni multiplexer 2x1 per pilotare opportunamente i segnali.



Il dato all'interno di ogni registro è a 16 bit. La parola intera si suddivide in MSB e LSB da 8 bit ciascuno. Lo schema in figura viene utilizzato 8 volte, tranne per il registro centrale, il quale non è mai soggetto al padding. La scelta di suddividere la parola è stata effettuata anche per gestire la finestra nel caso di parallelismo pari a 2. Per gestire il padding, si utilizza un segnale formato da 8 bit: ciascun bit ha il controllo sui relativi 2 MUX della sotto parola, per ogni registro. Se il relativo bit di padding è pari a 1, il segnale in uscita ai mux sarà la sotto parola, altrimenti si effettuerà lo zero-padding. Per elaborare la finestra, si considerano tutti i possibili casi di padding, e quindi i possibili valori del segnale di controllo:

```
constant PAD_1: std_logic_vector(7 downto 0):="11010000"; -- pixel in alto a sx (0,0)
constant PAD_2: std_logic_vector(7 downto 0):="11111000"; --  prima riga (0,i)
constant PAD_3: std_logic_vector(7 downto 0):="01101000"; -- pixel in alto a dx (0,63)
constant PAD_4: std_logic_vector(7 downto 0):="11010110"; -- prima colonna (j,0)
constant PAD_5: std_logic_vector(7 downto 0):="00010110"; -- pixel in basso a sx (63,0)
constant PAD_6: std_logic_vector(7 downto 0):="00011111"; -- ultima riga  (63,i)
constant PAD_7: std_logic_vector(7 downto 0):="00001011"; --  pixel in baso a dx (63,63)
constant PAD_8: std_logic_vector(7 downto 0):="01101011"; -- pixel ultima colonna (j,63)
constant PAD_9: std_logic_vector(7 downto 0):="11111111"; -- passa l'intera finestra ( no padding)
```

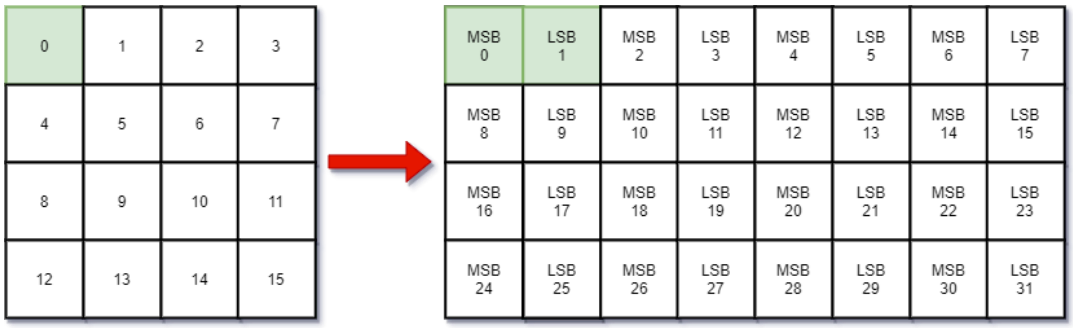
- **PAD_1** effettua il padding nel caso in cui la finestra di convoluzione si trova sul primo pixel, in alto a sinistra dell'immagine;
- **PAD_2** effettua il padding lungo tutta la prima riga;
- **PAD_3** effettua il padding nel caso in cui la finestra di convoluzione si trova sull'ultimo pixel della prima riga;
- **PAD_4** effettua il padding lungo la prima colonna;
- **PAD_5** effettua il padding per il primo pixel dell'ultima riga;
- **PAD_6** effettua il padding lungo tutta la riga;
- **PAD_7** effettua il padding per l'ultimo pixel dell'ultima riga**;**
- **PAD_8** effettua il padding lungo l'ultima colonna;
- **PAD_9** fa passare l'intera finestra (no padding).

La gestione di quale segnale di padding fornire al circuito è determinata dalla FSM, in funzione del pixel attuale da processare.

2.1.3. Subword Addressing Circuit: SIMD

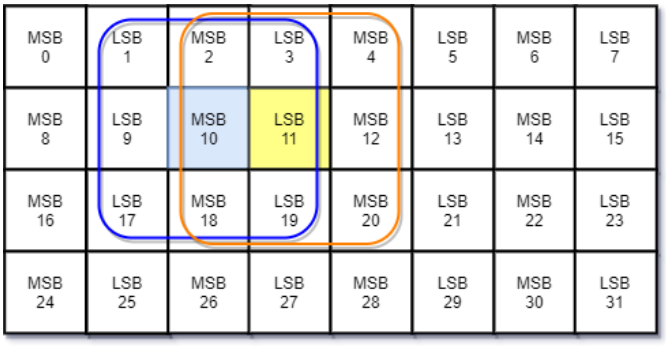
Per gestire il parallelismo 2 nel caso di segnali in ingresso ad 8 bit, è stato adottato il seguente approccio: il generico pixel dell'immagine iniziale 64x64 a 16 bit è stato suddiviso in due sotto-parole ad 8 bit contenti rispettivamente la parte MSB e la parte LSB del pixel iniziale. Trattando l'immagine in questo modo, è come se si fosse raddoppiato il numero di colonne andando ad elaborare un'immagine 64x128 ad 8 bit, posizionando sulle colonne con indice dispari gli MSB e sulle colonne con indice pari gli LSB.

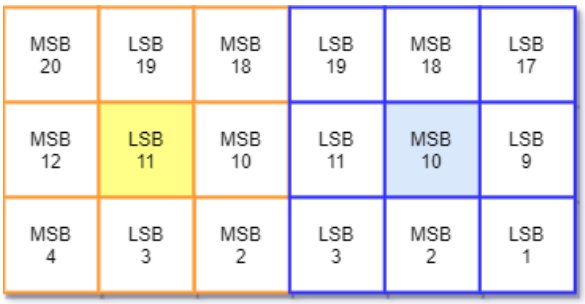
La convoluzione in questo caso verrà eseguita sul pixel ad 8 bit e poiché nei registri si memorizzano parole a 16 bit indipendentemente dalla scelta del tipo di parallelismo, si è in grado di effettuare 2 operazione di convoluzione 3x3 contemporaneamente. Ad esempio, in un caso di matrice 4x4:



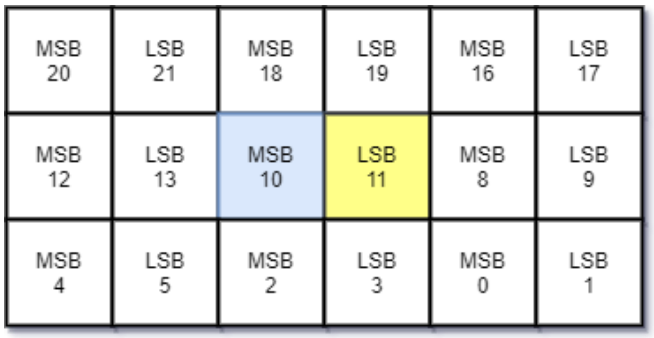
Tuttavia, dividere la parola iniziale in MSB ed LSB non è sufficiente per eseguire l'operazione di convoluzione. Per costruire la finestra intorno al generico pixel è stata utilizzata una rete di MUX 2:1 andando a creare nei 9 registri le 2 sottomatrici di convoluzione 3x3 ad 8 bit.

Considerando come pixel centrali i pixel 10 ed 11 corrispondenti al P(1,1) della matrice iniziale, si vogliono ottenere le seguenti matrici di convoluzione, centrate rispettivamente su 10 ed 11:





invece dal Register File si ottiene:



Per ottenere il risultato voluto si prendono dai registri adiacenti la porzione MSB/LSB necessaria per costruire la nuova parola; quindi, alla fine di questa operazione di SUBWORD si ottiene nel caso con parallelismo 1 la seguente finestra di convoluzione

*P(i+1,j+1)*W(2,2)*	*P(i+1,j)*W(2,1)*	*P(i+1,j-1)*W(2,0)*
*P(i,j-1)*W(1,2)*	*P(i,j)*W(1,1)*	*P(i,j-1)*W(1,0)*
*P(i-1,j-1)*W(0,2)*	*P(i-1,j)*W(0,0)*	*P(i-1,j-1)*W(0,0)*

nel caso di parallismo 2 (SIMD=1) otteniamo la seguente matrice

P(i+1,j+1)_MSB	*P(i+1,j)_LSB*	*P(i+1,j)_MSB*	*P(i+1,j)_LSB*	*P(i+1,j)_MSB*	*P(i+1,j-1)_LSB*
P(i,j-1)_MSB	*P(i,j)_LSB*	*P(i,j)_MSB*	*P(i,j)_LSB*	*P(i,j)_MSB*	*P(i,j-1)_LSB*
P(i+1,j+1)_MSB	*P(i-1,j)_LSB*	*P(i-1,j)_LSB*	*P(i-1,j)_LSB*	*P(i-1,j)_LSB*	*P(i-1,j-1)_LSB*

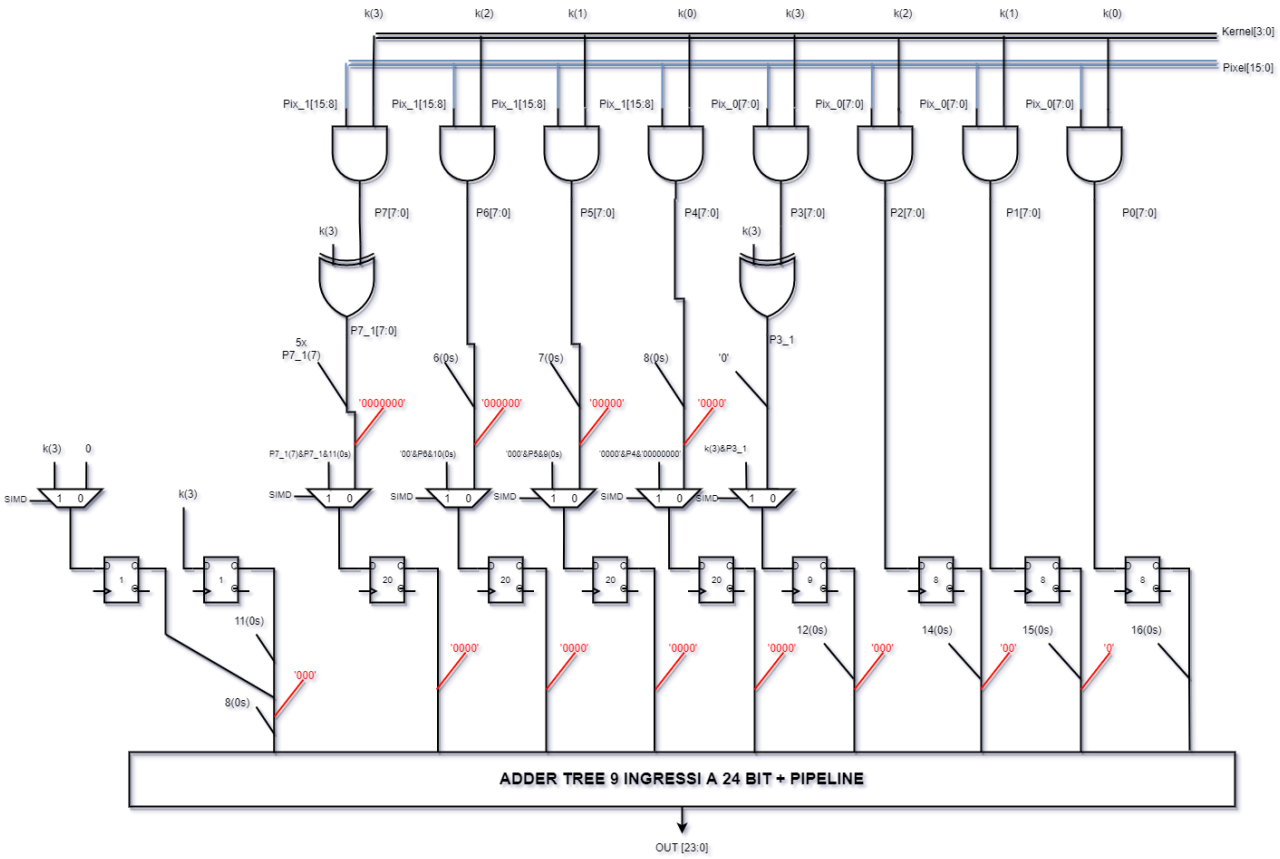
Per ottenere il risultato corretto dalla convoluzione, è necessario moltiplicare correttamente ogni coefficiente del filtro con il corrispettivo elemento della finestra selezionata della matrice.

2.2. Moltiplicatore carta e penna

Il moltiplicatore realizzato è basato sul *metodo carta e penna* e rispetta la seguente equazione:

$$P(A,B) = \sum_{i=0}^{n-1} \sum_{j=0}^{n-1} (b_i \cdot a_j) \times 2^{i+j}$$

L'architettura realizzata è la seguente:



Dato un moltiplicando **A** ad *n* bit ed un moltiplicatore **B** ad *m* bit, il prodotto deve essere rappresentato su *n+m* bit. Dunque, nel caso a 16 bit (parallelismo 1) il prodotto è a 20 bit mentre nel caso di parallelismo 2 otteniamo 2 risulti a 12 bit concatenati con una singola parola ad 12 bit.

Quindi per utilizzare il paradigma SIMD adottando un approccio modulare, istanziando per ogni operazione di moltiplicazione la stessa architettura in grado di elaborare in entrambi i casi, viene usato un modulo: ***MULT_SIMD_3X3*** che riceve un pixel a 16 bit e produce un'uscita a 24 bit. Nel caso 8 bit, basterà concatenare opportunamente i due risultati, mentre nel caso a 16 bit bisognerà estendere il prodotto per portarlo ai 24 bit di uscita. Il pixel in ingresso (moltiplicando) viene diviso in MSB ed LSB e si esegue l'*AND BITWISE* tra l'iesimo elemento dell'moltiplicatore e rispettivamente le parti MSB ed LSB, calcolando in questo modo i prodotti parziali. I prodotti parziali devono essere shiftati a sinistra in funzione del peso del singolo prodotto parziale, l'estensione invece deve essere scelta ad hoc a seconda della rappresentazione. Nel caso 16 bit si hanno 4 prodotti parziali, ma siccome si è splittata inizialmente la parola in 2 si colloca in maniera corretta i singoli prodotti parziali, come segue:

LSB*W(0)	2^0
LSB*W(1)	2^1
LSB*W(2)	2^2
LSB*W(3)	2^3

MSB*W(0)	2^8
MSB*W(1)	2^9
MSB*W(2)	2^10
MSB*W(3)	2^11

Nel caso a precisione doppia i pesi associati ai prodotti parziali sono differenti: la parte LSB rimane la stessa al caso di parallelismo 1 mentre i pesi associati alla parte MSB sono:

MSB*W(0)	2^12
MSB*W(1)	2^13
MSB*W(2)	2^14
MSB*W(3)	2^15

Nel caso in cui k(3) (MSB del coefficiente del kernel) è uguale a 1, il prodotto per k(3) equivale a calcolare il 2's(A), che a livello logico può essere implementato come *not(A)+1*.

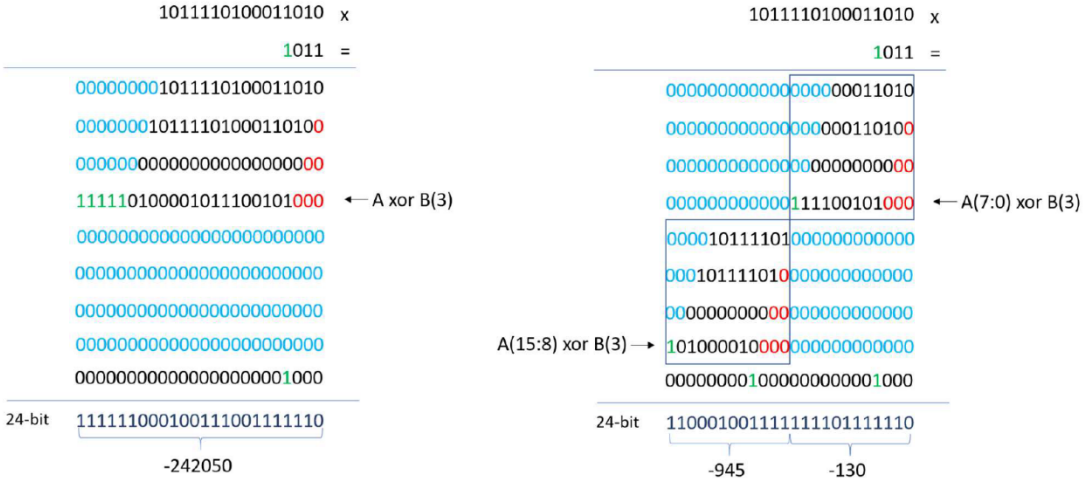
Per calcolare il 2's di A a livello circuitale, viene messo in AND BITWISE A(lsb) e A(msb) con k(3): se quest'ultimo è uguale ad 1, la and restituisce l'intera sequenza, mentre la xor per 1 restituisce il negato della sequenza a cui va successivamente aggiunto 1 per calcolare il 2's.

Occorre fare una precisazione nel caso di parallelismo 1, quindi quando si opera sull'intera parola a 16 bit: il bit "1" va inserito in posizione 3, quindi solo sulla parte LSB.

Nel caso di parallelismo 2 ad 8 bit invece, se k(3)=1, occorre calcolare il 2's sia della parte LSB che quella MSB, quindi and + xor+1. In questo caso si dovrà aggiungere 1 nella posizione 3 (peso 10^3) della parte LSB ed in posizione 3 della parte MSB (peso 10^).

Per tener conto di ciò si usa un mux che gestisce questo bit nel caso di parallelismo 2. In particolare, se SIMD=1, quindi se si lavora con parallelismo 1, si andrà a scrivere questo 1 in posizione 16 altrimenti se SIMD=0, per cui basta aggiungere 1 solo solla porzione meno significativa.

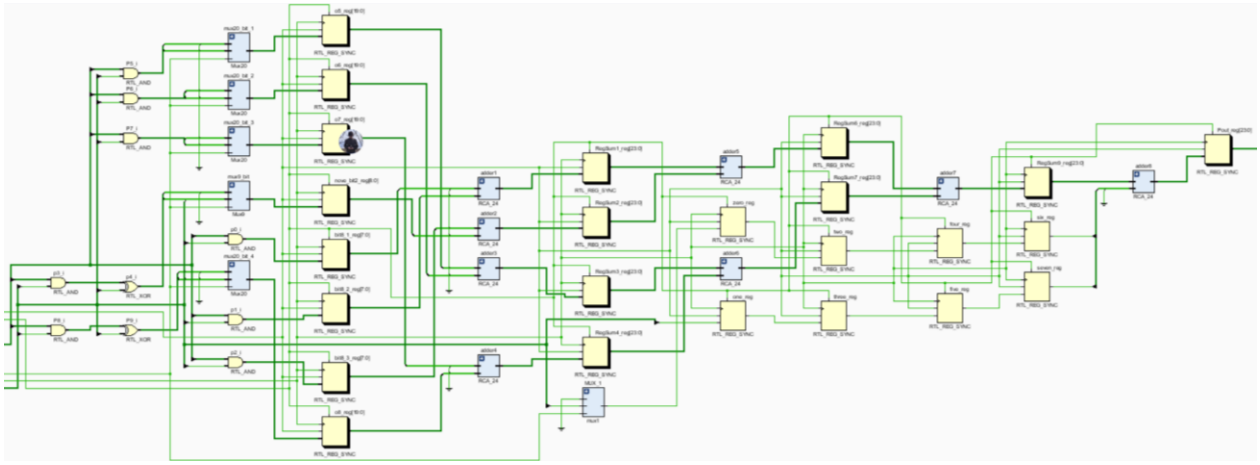
Tutto ciò che è stato appena detto può essere visto in figura:

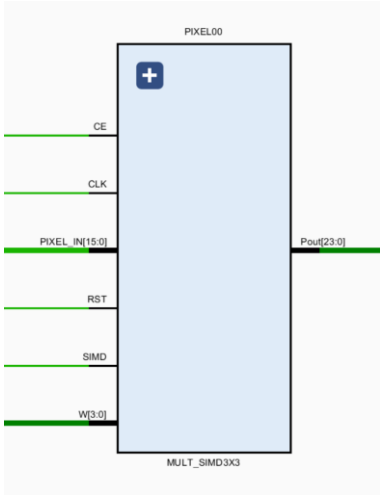


Una volta che i prodotti parziali sono stati calcolati ed estesi, gestendo i 2 diversi parallelismi, portando il singolo prodotto parziale su 24 bit, decidendo dunque di sprecare risorse nel caso del parallelismo 1 dove avremmo ottenuto come risultato del singolo prodotto un out a 20 bit.

I 9 prodotti parziali vengono campionati prima di entrare in ingresso ad un *adder tree* caratterizzato da 8 sommatori su 4 livelli (livelli=log2(numero_operandi)). Grazie all'estensioni preventive non occorre effettuare ulteriori operazioni per gestire il segno.

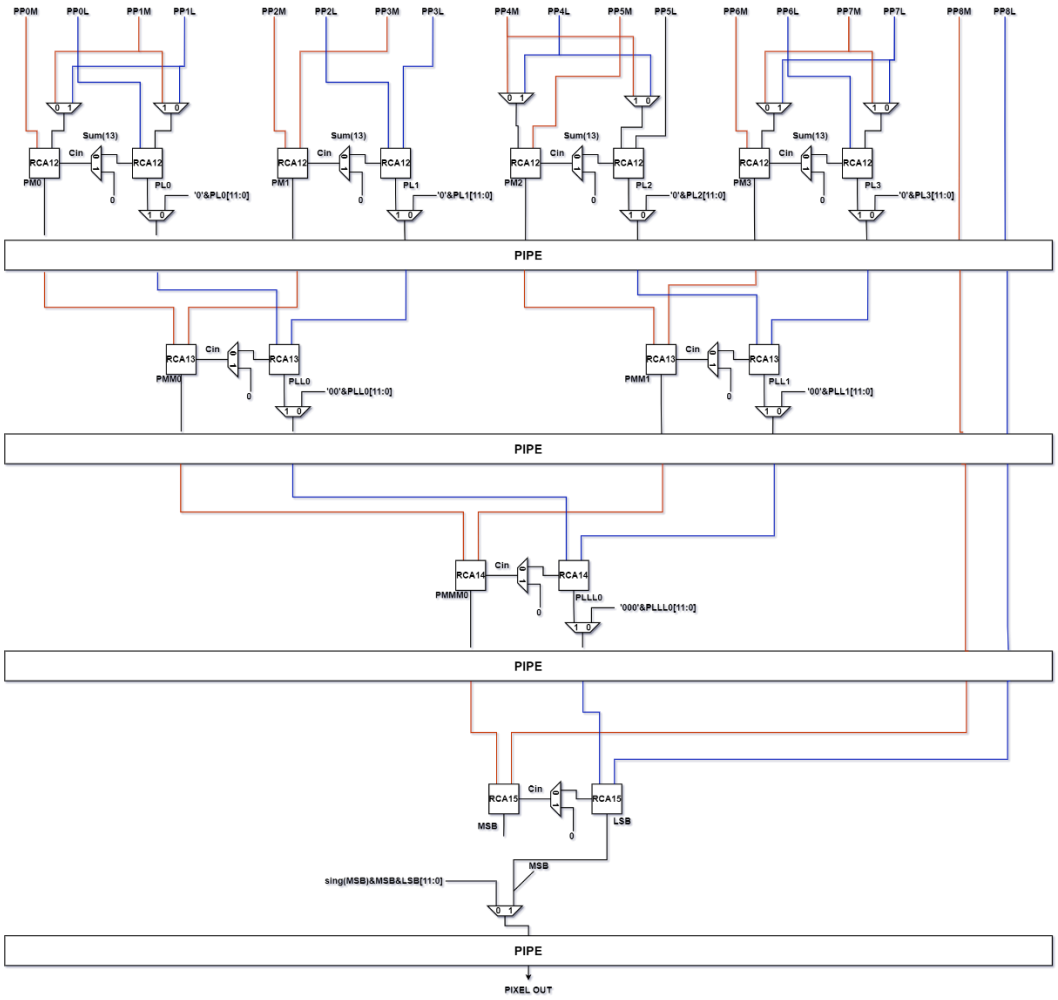
Per migliorare la frequenza di funzionamento è stato introdotto uno stadio di *pipe*, per campionare gli ingressi dell'adder tree, ed ulteriori stadi di pipe per ciascun livello dell'albero di somma.




PIXEL00
CE
CLK
PIXEL_IN[15:0]
RST
SIMD
W[3:0]
Pout[23:0]
MULT_SIMD3X3

2.3. Sommatore

Lo step successivo alle 9 moltiplicazioni è quello della somma finale, per generare il pixel filtrato. L'architettura generale è riportata in figura:



Anche in questo caso i dati in ingresso sono stati splittati in MSB ed LSB (a 12 bit ciascuno) per ciascun prodotto calcolato, e i vari livelli di somma operano in maniera distinta su queste parti per ottenere il risultato.

Tramite una rete di Multiplexer si vanno a selezionare gli ingressi dei sommatori per gestire i 2 casi, andando a fare la somma di MSB ed LSB in maniera distinta per il caso simd=0. Per il parallelismo 2 si vanno a posizionare sugli RCA che operano sugli LSB (nel caso del parallelismo 1) i termini relativi alla finestra di convoluzione dell'MSB nel caso ad 8 bit, mentre su quella MSB i termini relativi alla finestra di convoluzione sul pixel LSB della matrice iniziale.

È necessario andare a sommare gli elementi moltiplicati per il corretto coefficiente del kernel nel caso delle due sotto finestre.

L'architettura progettata fa riferimento ad un adder tree realizzato con 4 livelli ***RCA*** di dimensione crescente, in quanto per processare l'intera parola (parallelismo 1) ciascun livello di somma produce un'uscita a n+1 bit, quindi l'uscita sarà ad *n+4* bit.

Nel caso di parallelismo 2, poiché si agisce in maniera indipendente sui dati, ciascun livello di somma restituisce un out ad *n+2* bit, in quanto sia la somma degli MSB che quelli degli LSB devono essere rappresentate rispettivamente ad n+1 bit.

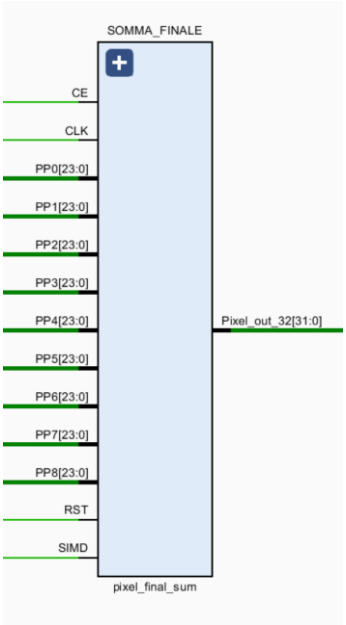
La lunghezza della parola in out dall'adder tree in questo caso è rappresentata ad *n+8* bit: poiché l'ingresso è a 24 (due sequenze da 12) si ottiene un'uscita su 32 bit, dove bisogna posizionare in maniera corretta i risultati delle somme che rappresentano il generico pixel filtrato nelle 2 sotto-finestre come spiegato nell'introduzione, che nel nostro caso sono ottenute al contrario.

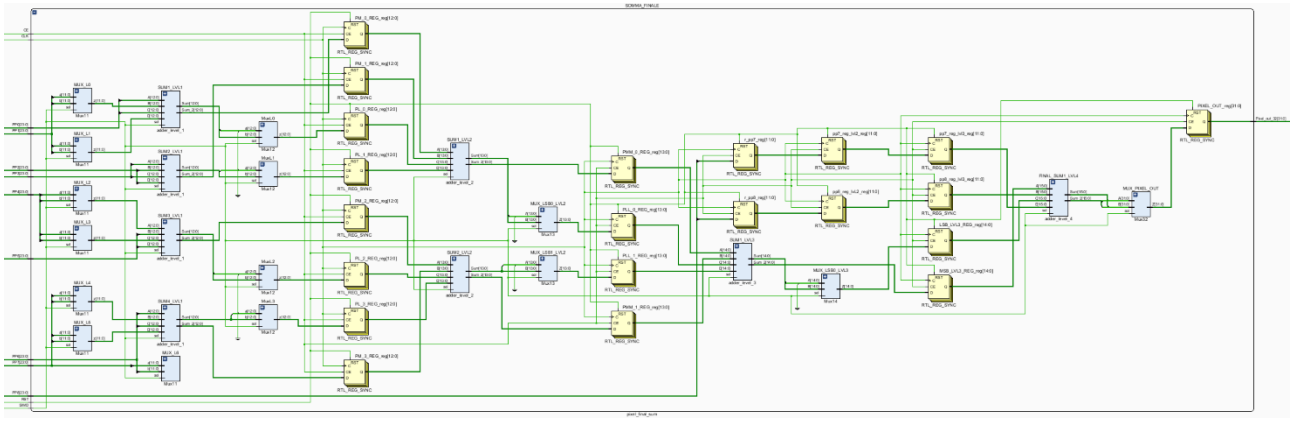
Per realizzare un sommatore SIMD si deve prestare particolare attenzione alla propagazione del riporto. Nel caso di parallelismo 1, quindi quando si vanno a sommare le parole "intere" la parte LSB rimane fissa su 12 bit mentre la parte MSB incrementa di 1 bit per ogni livello.

Quindi tramite un MUX si va a propagare (Simd=0) o meno il riporto in posizione 12 (peso 10^12) che diventa il carry_in del RCA che opera sulla parte MSB.

Per gestire il risultato corretto, viene usato un MUX per far passare così com'è la somma degli RCA che operano sulla Parte LSB restituendo un out ad *n+1* bit per ogni livello (Simd=1), oppure andando a prendere solo somma sui 12 bit nel caso di simd=0, sostituendo con 0 i bit in posizione > di 11(10^11).

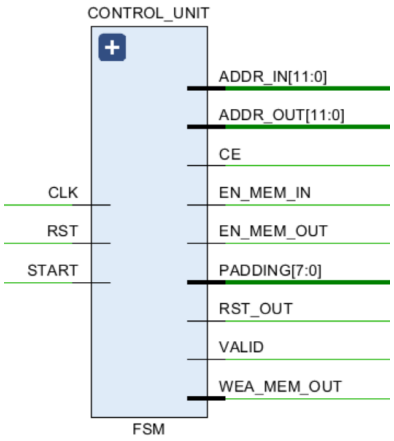
Infine, dopo queste somme, sempre tramite un MUX si andrà a scegliere il risultato opportuno a seconda del parallelismo utilizzato.





2.4. FSM

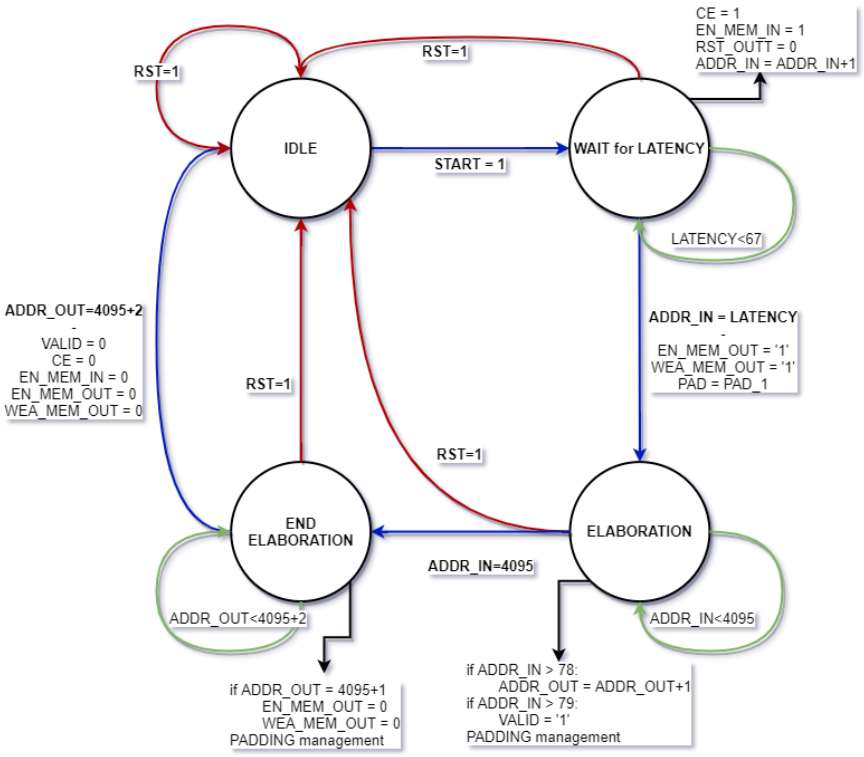
La FSM realizzata per gestire e controllare le operazioni si presenta con la seguente interfaccia:



Quindi l'unità di controllo riceve il clock, il reset ed il segnale di start e produce i seguenti segnali:

- RST out per gestire il reset dell'intero circuito
- addR_in indirizzi memoria di ingresso
- addr_out indirizzi memoria di uscita
- en_mem_in enable memoria di ingresso
- en_mem_out enable memoria di uscita
- wea_mem_out: write enable memoria di uscita
- padding: segnale per gestire il padding
- valid: segnale di validità
- CE: clock enable

È composta da 4 stati:



- **IDLE**: stato in cui il circuito viene resettato, rimanendo in attesa del segnale di **START**.
- **WAIT LATENCY**: viene abilitata la memoria di ingresso, fornendogli inoltre gli indirizzi per leggere l'immagine in uscita. Una volta trascorsa la latenza iniziale, ovvero quando il primo pixel dell'immagine arriverà sul registro centrale, si passa nello stadio **ELABORATION**
- **ELABORATION**: viene gestito il padding a secondo dell'elemento che si sta processando, una volta trascorsa la latenza del circuito convolutore (pari ad 11 colpi di clock) si alza l'***enable*** della memoria di uscita, e viene abilitata in scrittura, e dopo i 2 colpi di clock della memoria di uscita si ottiene il primo risultato valido. La macchina rimane nello stadio di *elaboration* finché l'indirizzo della memoria di ingresso non arriva a 4095, quindi finché l'ultimo dato è stato letto dalla memoria in ingresso.
- **END ELABORATION**: si rimane in attesa finchè l'ultimo pixel filtrato arriva in ingresso alla memoria di uscita, gestendo il padding per le righe rimanenti, continuando ad asserire l'uscita valida finché non viene fornito in uscita l'ultimo pixel filtrato.

Terminata l'elaborazione l'uscita non sarà più valida e si ritorna in IDLE, stato nel quale la macchina viene resettata.

Il segnale ***valid*** prodotto dalla FSM andrà a gestire inoltre la scrittura dei risultati dell'operazione di filtraggio su un file di testo, attraverso il modulo ***W2F*** (*Write to File*), andando a scrivere il dato in out ad ogni fronte di clock solo se il dato è valido.

```
WHEN IDLE=>
    rst_outt<='1';
    ce<='0';
    valid_s<='0';
    PADDING<=(OTHERS=>'0');
    en_mem_in<='0';
    en_mem_out<='0';
    wea_mem_out<="0";
    riga<=(others=>'0');
    colonna<=(others=>'0');
    address_mem_in<=(others=>'0');
    address_mem_out<=(others=>'0');
    IF (START='1') THEN
        CURRENT_STATE<=WAIT_LATENCY;
         rst_outt<='0';
         EN_MEM_IN<='1';
         valid_S<='0';
         ce<='1';
    ELSE
         RST_OUTt<='1';
    END IF;
```

```
WHEN WAIT_LATENCY=>
      rst_outt<='0';
      en_mem_in<='1';
      en_mem_out<='0';
      address_mem_out<=(others=>'0');
      wea_mem_out<="0";
      address_mem_in<=address_mem_in+1;
      if (address_mem_in = LATENCY) then
         CURRENT_STATE<=ELABORATION;
         en_mem_out<='1';
         wea_mem_out<="1";
         PADDING<=PAD_1;
       else
         current_state<=wait_latency;
       end if;
```

```
WHEN ELABORATION=>
    rst_outt<='0';
    en_mem_in<='1';
    en_mem_out<='1';
    colonna<=colonna+1;
    if (address_mem_in>latenc) then
       address_mem_out<=address_mem_out+1;
    end if;
    if (address_mem_in>(latenc+1)) then
       valid_s<='1';
    end if;
    if (colonna = 63) THEN
       riga<=riga+1;
    end if;
    -- GESTIONE PADDING
    if (riga=0) then
        if (colonna=63) then
            PADDING<=PAD_4;
        elsif (colonna=62) then
            PADDING<=PAD_3;
        else
            PADDING<=PAD_2;
        end if;
    elsif (RIGA=62) THEN
        if (colonna=63) then
            PADDING<=PAD_5;
        elsif (colonna=62) then
            PADDING<=PAD_8;
        else
            PADDING<=PAD_9;
        END IF;
    elsif (RIGA=63) THEN
        IF (COLONNA=63) THEN
            PADDING<=PAD_1;
        ELSIF
           (COLONNA=62) THEN
            PADDING<=PAD_7;
        ELSE
            PADDING<=PAD_8;
        END IF;
    ELSE
        IF (COLONNA=63) THEN
            PADDING<=PAD_4;
        ELSIF
            (COLONNA=62) THEN
            PADDING<=PAD_8;
        ELSE
            PADDING<=PAD_9;
        END IF;
    END IF;

     address_mem_in<=address_mem_in+1;
     wea_mem_out<="1";

    if (address_mem_in = "111111111111") then
        CURRENT_STATE<=END_ELABORATION;
        valid_s<='1';
    else
        CURRENT_STATE<=ELABORATION;
    end if;
```

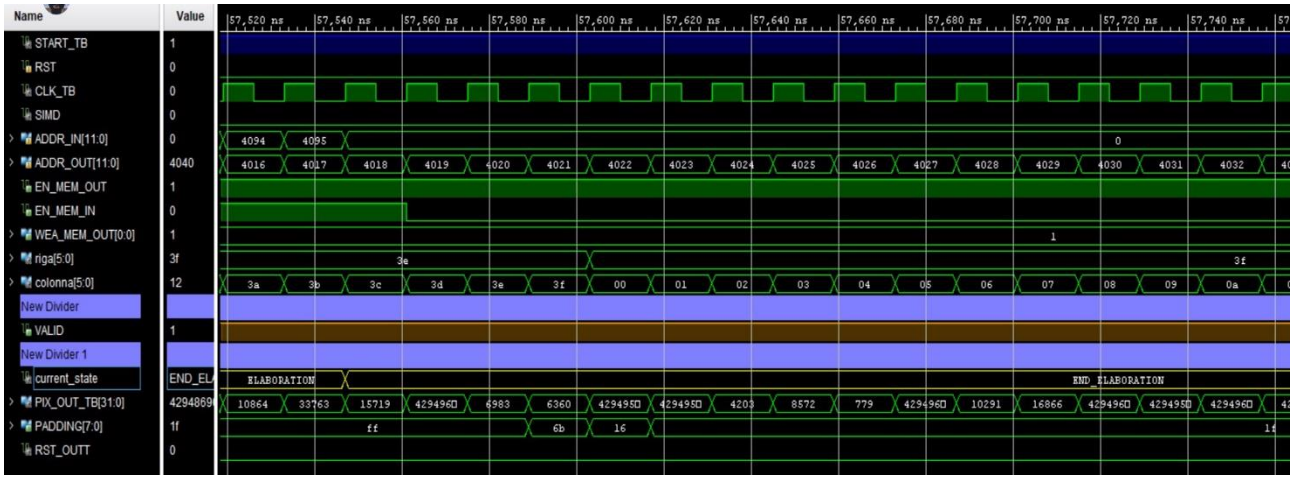
```
when END_ELABORATION =>
    rst_outt<='0';
    en_mem_in<='0';
    en_mem_out<='1';
    wea_mem_out<="1";
    address_mem_in<=(others=>'0');
    address_mem_out<=address_mem_out+1;
    IF (RIGA=62) THEN
        if (colonna=63) then
            PADDING<=PAD_5;
        elsif (colonna=62) then
            PADDING<=PAD_8;
        else
            PADDING<=pad_9;
        END IF;
    ELSIF (RIGA=63) THEN
        IF (COLONNA=63) THEN
            PADDING<=PAD_1;
        ELSIF
            (COLONNA=62) THEN
            PADDING<=PAD_7;
        ELSE
            PADDING<=PAD_6;
        END IF;
    END IF;

    if (address_mem_out = "111111111111"+1) then
        en_mem_out<='0';
        wea_mem_out<="0";
    elsif (address_mem_out = "111111111111"+2) then
        CURRENT_STATE<=IDLE;
        valid_s<='0';
        en_mem_out<='0';
        wea_mem_out<="0";
        colonna<=(others=>'0');
        riga<=(others=>'0');
        ce<='0';
        rst_outt<='1';
        PADDING<=(OTHERS=>'0');
    else
        CURRENT_STATE<=END_ELABORATION;
        if (colonna = 63) then
            colonna<=(others=>'0');
            riga<=riga+1;
        end if;
            colonna<=colonna+1;
            valid_s<='1';
    end if;
```

Transizione da IDLE a Wait Latency:

Transizione da Wait Latency a Elaboration:

Transizione da Elaboration a End Elaboration:



2.5. Confronto e risultati Vivado vs. Matlab

Alla stessa immagine in ingresso sono stati applicati diversi filtri, ed i risultati sono stati confrontati quelli ottenuti sul tool SW di matlab, confermando la correttezza delle operazioni svolte.

In conclusione le performance offerte dal circuito

CICLI DI CLOCK TOTALI = mxn+ LATENZA

Prestazioni		
Freq. max	Throughtput	Latency
71.4286 MHz	1 (SIMD=0) 2(SIMD=1)	67(reg)+10(conv)+2(mem)= **79**

Mentre dai report Timing ed Utilizzation Post implementation otteniamo:

Max Delay Paths

Slack (MET) : 1.193ns (required time - arrival time)
Source: CONVOLUTORE/SOMMA_FINALE/PLL_0_REG_reg[1]/C
(rising edge-triggered cell FDRE clocked by clk {rise@0.000ns fall@7.000ns period=14.000ns})
Destination: CONVOLUTORE/SOMMA_FINALE/MSB_LVL3_REG_reg[13]/D
(rising edge-triggered cell FDRE clocked by clk {rise@0.000ns fall@7.000ns period=14.000ns})
Path Group: clk
Path Type: Setup (Max at Slow Process Corner)
Requirement: 14.000ns (clk rise@14.000ns - clk rise@0.000ns)
Data Path Delay: 12.816ns (logic 4.391ns (34.262%) route 8.425ns (65.738%))
Logic Levels: 13 (LUT3=1 LUT5=10 LUT6=2)
Clock Path Skew: -0.037ns (DCD - SCD + CPR)
Destination Clock Delay (DCD): 4.339ns = (18.339 - 14.000)
Source Clock Delay (SCD): 4.699ns
Clock Pessimism Removal (CPR): 0.323ns
Clock Uncertainty: 0.035ns ((TSJ^2 + TIJ^2)^1/2 + DJ) / 2 + PE
Total System Jitter (TSJ): 0.071ns
Total Input Jitter (TIJ): 0.000ns
Discrete Jitter (DJ): 0.000ns
Phase Error (PE): 0.000ns

Name	Constraints	Status	WNS	TNS	WHS	THS	TPWS	Total Power	Failed Routes	LUT	FF	BRAMs
synth_1 (active)	**constrs_1**	**Synthesis Out-of-date**								**1832**	**2548**	**0.00**
impl_1	constrs_1	route_design Complete!	1.193	0.000	0.028	0.000	0.000	0.134	0	1834	2551	6.00

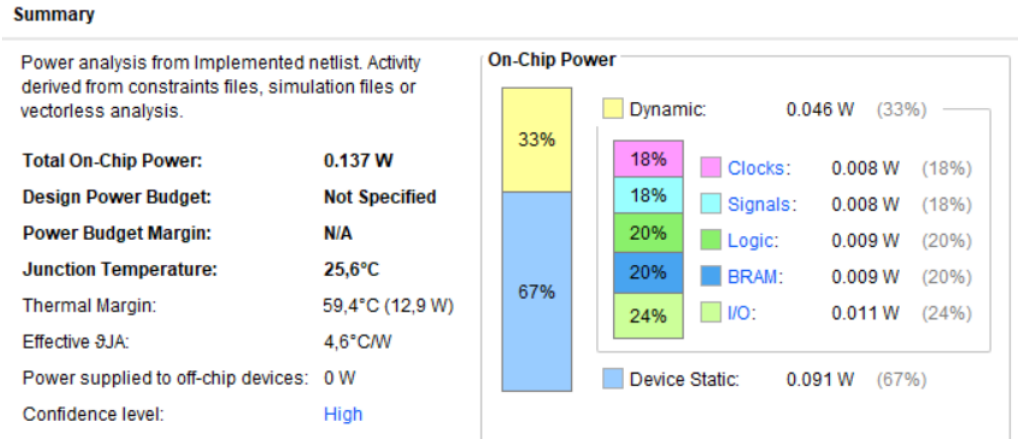
1. Slice Logic

Site Type	Used	Fixed	Available	Util%
Slice LUTs	1834	0	63400	2.89
LUT as Logic	1730	0	63400	2.73
LUT as Memory	104	0	19000	0.55
LUT as Distributed RAM	0	0		
LUT as Shift Register	104	0		
Slice Registers	2551	0	126800	2.01
Register as Flip Flop	2551	0	126800	2.01
Register as Latch	0	0	126800	0.00
F7 Muxes	0	0	31700	0.00
F8 Muxes	0	0	15850	0.00

8. Primitives

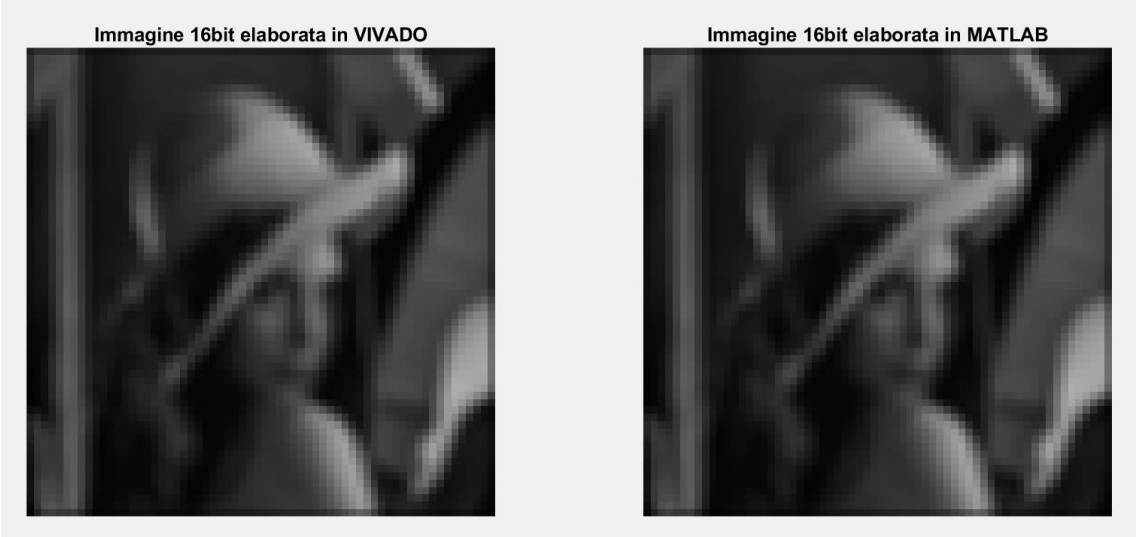
Ref Name	Used	Functional Category
FDRE	2548	Flop & Latch
LUT5	694	LUT
LUT3	639	LUT
LUT2	631	LUT
LUT4	405	LUT
LUT6	354	LUT
SRLC32E	64	Distributed Memory
SRL16E	40	Distributed Memory
IBUF	40	IO
OBUF	33	IO
CARRY4	10	CarryLogic
RAMB36E1	6	Block Memory
LUT1	5	LUT
FDCE	3	Flop & Latch
BUFG	1	Clock

Per il report power per avere un'informazione più precisa, si è andato ad estrapolare il file .saif che contiene tutte le info sull'attività di switching di tutti in nodi del design, dal quale si sono ottenuti i seguenti risultati:



Si riportano infine alcuni confronti grafici con i risultati di Matlab:

Smoothing



Laplacian

Sobel

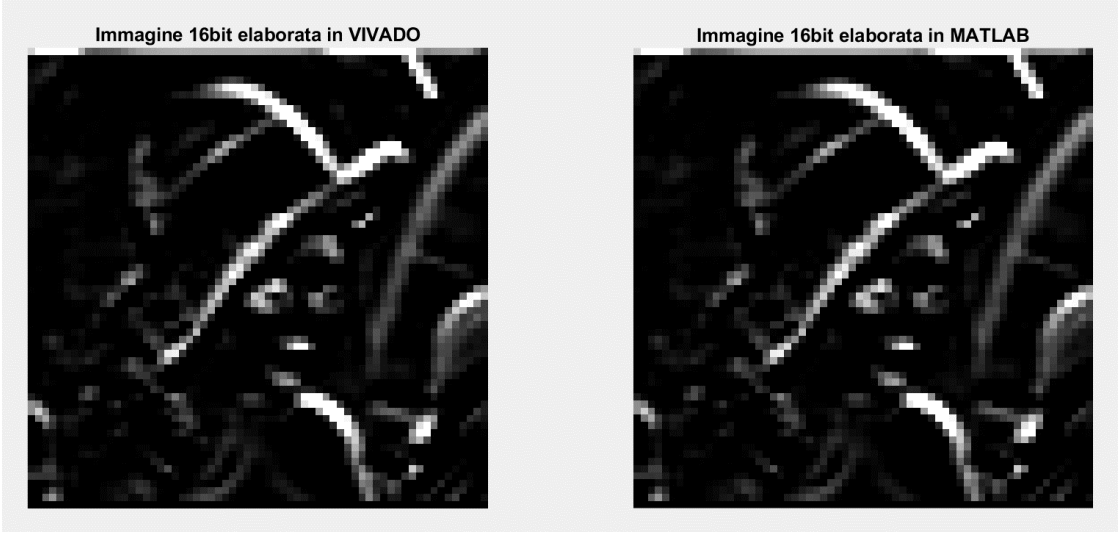


Immagine 16bit elaborata in VIVADO
Immagine 16bit elaborata in MATLAB