

Corso di Laurea Magistrale in Ingegneria Elettronica



Relazione di Progettazione dei Sistemi Digitali
A.A. 2023/2024

*Filtro di immagini RGB
tramite kernel 3x3 isotropico*

RAFFAELE PETROLO

SOMMARIO

<i>INTRODUZIONE</i>	4
<i>DESCRIZIONE DEL CIRCUITO</i>	5
BUFFER LINE.....	6
BLOCCO DI CALCOLO	7
BLOCCO DI CONTROLLO.....	8
1. TOP	9
VERSIONE 1	9
VERSIONE 2	10
VERSIONE 3: 32x32 e 64x64	11
VERSIONE 4	12
2. CARRY-SAVE-8BIT A 4 OPERANDI.....	13
2.1.1 FIRST (1 LIVELLO CSA_8BIT)	14
2.1.2 BLOCCO FA	15
2.2 PIPE_TO_ADDER	15
2.3 ADDER_12	16
3. BOOTH MULTIPLIER	17
3.1 BOOTH ENCODER	18
3.2 PARTIAL PRODUCTS	19
3.3 PLUS	20
3.4 MULTIPLEXER PER I PRODOTTI PARZIALI	20
3.5 SOMMA DEI PRODOTTI PARZIALI	21
4. SOMMA FINALE.....	22
5. BLOCCO DI SATURAZIONE	23
6. MACCHINA A STATI FINITI E DECODERS	24
6.1 FSM MINIMALE	24
6.2 FSM CON GESTIONE DEI BORDI.....	26
6.3 FSM PARAMETRICA CON GESTIONE DEI BORDI	31
7. CONVERTITORE RGB2GRAY	41
7.1 RGB2GRAY: GESTIONE DEL RESTO.....	43
<i>REPORT DI TIMING</i>	48
<i>REPORT DI POTENZA E RISORSE</i>	51
<i>SIMULAZIONE BEHAVIORAL DEI SEGNALI</i>	56
SEGNALE DI INIZIO LETTURA	56
SEGNALE DI ATTIVAZIONE DELLA FSM	60
SEGNALE DI VALIDAZIONE OUTPUT.....	64

SEGNALE DI FINISH INTERNO.....	68
SEGNALE DI FINE FILTRAGGIO	71
TEMPO DI FILTRAGGIO	75
<i>SIMULAZIONI POST IMPLEMENTATION TIMING.....</i>	<i>79</i>
LATENZA TRA INPUT E OUTPUT	79
SEGNALI DI FINISH.....	83
TEMPI DI FILTRAGGIO.....	87
SEGNALI DI INIZIO LETTURA	91
<i>RISULTATI IN MATLAB</i>	<i>95</i>
VERSIONE 1	95
VERSIONE 2	96
VERSIONE 3	99
VERSIONE 4	102
<i>TEST BENCH.....</i>	<i>107</i>
TESTBENCH V1	107
TESTBENCH V2.....	111
TESTBENCH V3	115
TESTBENCH V4.....	119

INTRODUZIONE

Il seguente progetto propone la realizzazione di un circuito finalizzato al **filtraggio bidimensionale** di immagini digitali.

Il sistema è implementato su schede **prototipali FPGA** della famiglia *Virtex 7* e rispetta i seguenti requisiti tecnici:

- Implementazione di un filtro di dimensione 3x3, con coefficienti rappresentati su 8 bit in complemento a due.
- Dimensioni minime dell'immagine in ingresso pari a 32x32 pixel in **grayscale**, ognuno rappresentato ad 8 bit in formato unsigned.

Il circuito è stato progettato per gestire efficientemente l'elaborazione delle immagini secondo le specifiche fornite, utilizzando il maggior numero di stadi di pipeline al fine di ottimizzare le prestazioni. Il circuito garantisce la **corretta saturazione dei pixels** in uscita ad una dimensione di 8 bit, corrispondente al range di valori interi [0 - 255].

La progettazione ha richiesto l'implementazione di un buffer per la lettura dell'immagine e l'utilizzo di una macchina a stati finiti (FSM) per garantire la corretta **sincronizzazione** delle operazioni di filtraggio.

L'operazione alla base del filtraggio è la **convoluzione**, che viene impiegata per analizzare un segnale e per evidenziare o sopprimere una sua particolare caratteristica.

Dati dei generici pixel appartenenti all' intorno di un **pixel specifico $P(i,j)$** e i coefficienti del **kernel $W(k,h)$** , dove h e k rappresentano le due dimensioni del filtro, il **pixel filtrato $Q(i,j)$** è ottenuto mediante la **convoluzione discreta bidimensionale** tra i pixel dell'intorno e i coefficienti del filtro stesso. Nell'operazione di **finestratura** si deve stabilire un *anchor point* (punto di ancoraggio), che rappresenta la posizione nella finestra 3x3 del generico pixel da filtrare. Questo valore nel kernel può essere fissato al centro, a sinistra o a destra; nel nostro caso è fissato al centro.

Immagine in ingresso 8X8

$P_{(0,0)}$	$P_{(0,1)}$	$P_{(0,2)}$	$P_{(0,3)}$	$P_{(0,4)}$	$P_{(0,5)}$	$P_{(0,6)}$	$P_{(0,7)}$
$P_{(1,0)}$	$P_{(1,1)}$	$P_{(1,2)}$	$P_{(1,3)}$	$P_{(1,4)}$	$P_{(1,5)}$	$P_{(1,6)}$	$P_{(1,7)}$
$P_{(2,0)}$	$P_{(2,1)}$	$P_{(2,2)}$	$P_{(2,3)}$	$P_{(2,4)}$	$P_{(2,5)}$	$P_{(2,6)}$	$P_{(2,7)}$
$P_{(3,0)}$	$P_{(3,1)}$	$P_{(3,2)}$	$P_{(3,3)}$	$P_{(3,4)}$	$P_{(3,5)}$	$P_{(3,6)}$	$P_{(3,7)}$
$P_{(4,0)}$	$P_{(4,1)}$	$P_{(4,2)}$	$P_{(4,3)}$	$P_{(4,4)}$	$P_{(4,5)}$	$P_{(4,6)}$	$P_{(4,7)}$
$P_{(5,0)}$	$P_{(5,1)}$	$P_{(5,2)}$	$P_{(5,3)}$	$P_{(5,4)}$	$P_{(5,5)}$	$P_{(5,6)}$	$P_{(5,7)}$
$P_{(6,0)}$	$P_{(6,1)}$	$P_{(6,2)}$	$P_{(6,3)}$	$P_{(6,4)}$	$P_{(6,5)}$	$P_{(6,6)}$	$P_{(6,7)}$
$P_{(7,0)}$	$P_{(7,1)}$	$P_{(7,2)}$	$P_{(7,3)}$	$P_{(7,4)}$	$P_{(7,5)}$	$P_{(7,6)}$	$P_{(7,7)}$

Kernel di convoluzione 3x3

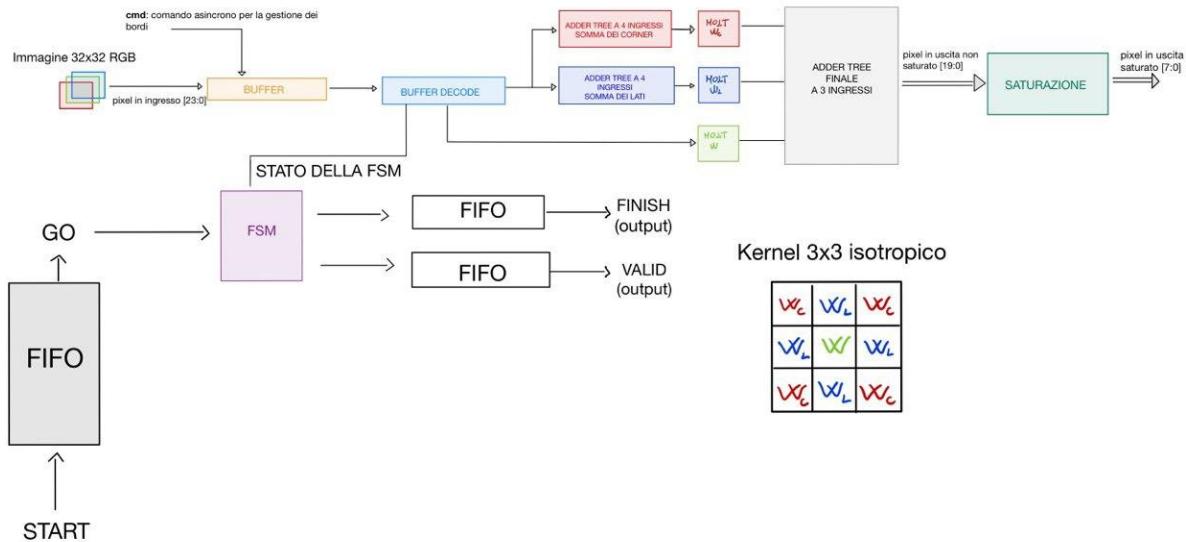
$W_{(0,0)}$	$W_{(0,1)}$	$W_{(0,2)}$
$W_{(1,0)}$	$W_{(1,1)}$	$W_{(1,2)}$
$W_{(2,0)}$	$W_{(2,1)}$	$W_{(2,2)}$

Immagine di uscita 8X8

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

DESCRIZIONE DEL CIRCUITO

Il progetto realizzato effettua il filtraggio di **immagini digitali RGB** di dimensione 32x32. Di seguito è riportato lo schema a blocchi generale del circuito progettato.



Da un punto di vista strutturale questo circuito si compone di due macro-blocchi:

- 1) **BUFFER LINE**, comprende la struttura di buffer, per gestire le righe delle immagini, collegata ad un eventuale blocco di *buffer decode* per la gestione opportuna dei bordi dell'immagine (varia ad ogni versione del circuito).
- 2) **BLOCCO DI CALCOLO**, dedicato all'elaborazione del pixel da filtrare, comprensiva di moltiplicatori, sommatori e blocco di saturazione.
- 3) **BLOCCO DI CONTROLLO**, consente la sincronizzazione delle operazioni per ogni blocco; comprende l'FSM e diverse FIFO ausiliarie.

BUFFER LINE

Per la corretta gestione dell'intorno dei pixel, è stato utilizzato un **buffer lineare**. La funzione principale di questo buffer è la **memorizzazione** adeguata delle righe interessate al filtraggio per garantire che, dopo una latenza iniziale, sia disponibile la finestra di pixel corretta per l'elaborazione **a ogni colpo di clock**.

La struttura scelta per il Buffer è di tipo seriale, dove i pixel entrano di volta in volta al suo interno e ad ogni colpo di clock il pixel avanza nella posizione successiva.

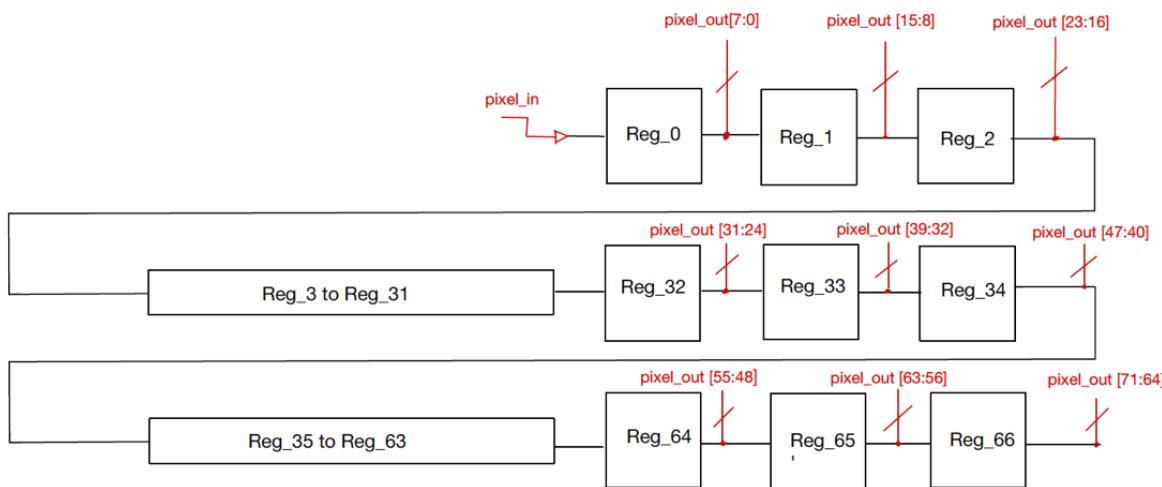


Figura 1.1 Schema a blocchi del Buffer

La lettura dell'immagine avviene riga per riga secondo la tecnica del *Raster Scan Order*, per cui l'ultimo pixel di ogni riga precede il primo pixel della successiva.

La scelta di un kernel $k \times k$ e di un'immagine di dimensione $n \times n$ richiede l'utilizzo di:

- **$k \times k$ registri** di uscita per comporre la finestra di filtraggio;
- **$k - 1$ strutture FIFO (First In First Out)**, di dimensione pari a $n-k$ per "bufferizzare" in maniera corretta le righe dell'immagine.

Al fine di soddisfare le specifiche progettuali, sono stati definiti i parametri $n = 32$, $k = 3$.

L'intero buffer può essere visto come la **serie di 67 registri**, ognuno dei quali implementato utilizzando un registro a 8 bit.

Per descrivere le 9 uscite dei singoli registri si è utilizzato un unico segnale “*pixel_out*”, ovvero un vettore di 72 bit (8 bit x 9 reg). Per collegare tutti i registri si è sfruttato un segnale interno “*Qinternal*”, ovvero un vettore di 536 bit scaglionato a 8 bit per ogni registro (8bit x 67reg), al fine di collegare in maniera efficiente, tramite un *for generate*, il tutto.

```

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use ieee.std_logic_unsigned.all;

entity TOP_BUF is
Port (pixel_in: in std_logic_vector(7 downto 0);
clk,rst: in std_logic;
pixel_out: out std_logic_vector(71 downto 0)
);
end TOP_BUF;

architecture Behavioral of TOP_BUF is

signal Qinternal: std_logic_vector(535 downto 0);
signal pix_p: std_logic_vector(7 downto 0);

component REG8 is
Port (Din: in std_logic_vector(7 downto 0);
clk, rst: in std_logic;
Q: out std_logic_vector (7 downto 0)
);
end component;
begin
pix_p<=pixel_in;
pixel_out(7 downto 0)<=Qinternal(7 downto 0);
pixel_out(15 downto 8)<=Qinternal(15 downto 8);
pixel_out(23 downto 16)<=Qinternal(23 downto 16);
pixel_out(31 downto 24)<=Qinternal(263 downto 256);
pixel_out(39 downto 32)<=Qinternal(271 downto 264);
pixel_out(47 downto 40)<=Qinternal(279 downto 272);
pixel_out(55 downto 48)<=Qinternal(519 downto 512);
pixel_out(63 downto 56)<=Qinternal(527 downto 520);
pixel_out(71 downto 64)<=Qinternal(535 downto 528);

REG8_0: REG8 port map(pix_p,clk,rst,Qinternal(7 downto 0));
FIFO: for i in 1 to 66 generate
REG8_i: REG8 port map (Qinternal(8*i-1 downto 8*(i-1)),clk,rst,Qinternal(8*(i+1)-1 downto 8*i));
end generate;

```

Figura 1.2 Descrizione VHDL del Buffer lineare

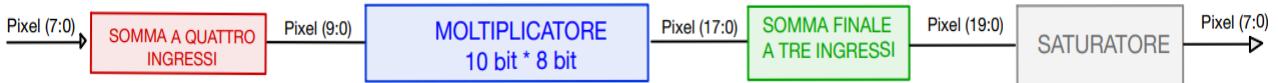
In merito alla **figura 1.1** e **1.2** i vari *pixel_out* sono in realtà i segnali $Qinternal(8*(i+1)-1 \text{ downto } 8*i)$ per $i = 0, 1, 2, 32, 33, 34, 64, 65, 66$, praticamente le uscite dei registri a 8 bit con quegli indici visti in **figura 1.1**. Da notare il salto di +30 ogni tre “*i*” dovuto sostanzialmente alle FIFO di dimensione 29 più il registro dal quale viene prelevato il dato.

BLOCCO DI CALCOLO

Nell'architettura descritta sono presenti due **Adder Tree** che gestiscono la somma dei pixel, la cui posizione nella **finestra di filtraggio** (che scorre pixel per pixel attraverso l'immagine) è associata al relativo coefficiente del **kernel isotropico**: in particolare, un sommatore è dedicato ai **pixel laterali**, mentre l'altro si occupa dei **pixel ai corner**.

Le uscite di questi **Adder Tree**, insieme al **pixel centrale**, vengono inviati rispettivamente a **tre moltiplicatori di Booth**. Questi moltiplicatori calcolano il prodotto tra i rispettivi pixel e i coefficienti del kernel: **W** per il pixel centrale, **W_L** per i pixel laterali e **W_C** per i pixel agli angoli.

Le uscite dei moltiplicatori vengono poi combinate in un **ADDER TREE** finale, che somma i risultati dei prodotti precedenti finalizzando l'operazione di convoluzione. L'uscita di questo sommatore è un pixel filtrato a **20 bit**.



Per garantire che il pixel in uscita rispetti le specifiche richieste, è presente un **blocco di saturazione**. Questo blocco si occupa di ridurre il valore del pixel filtrato da 20 bit a **8 bit**, assicurando che il risultato finale non superi i limiti consentiti.

BLOCCO DI CONTROLLO

Il circuito permette di scegliere il tipo di gestione dei bordi da utilizzare, come ad esempio estensione con **padding di zeri (neri)**, **padding di 255 (bianchi)**, **toroidale mista** o **mirroring**.

Per implementare questa flessibilità, è presente un modulo di **Buffer Decode**, controllato da una macchina a stati finiti **FSM** che modifica l'intorno dei pixel sui bordi in base all'estensione selezionata e allo stato corrente.

Per alleggerire la **FSM**, al fine di ottenere maggiori performance in termini di velocità, si è scelto di utilizzare delle **FIFO** per rispettare la latenza di elaborazione e gestire i segnali di start e di finish.

Nel seguente progetto sono state presentate **quattro differenti versioni**:

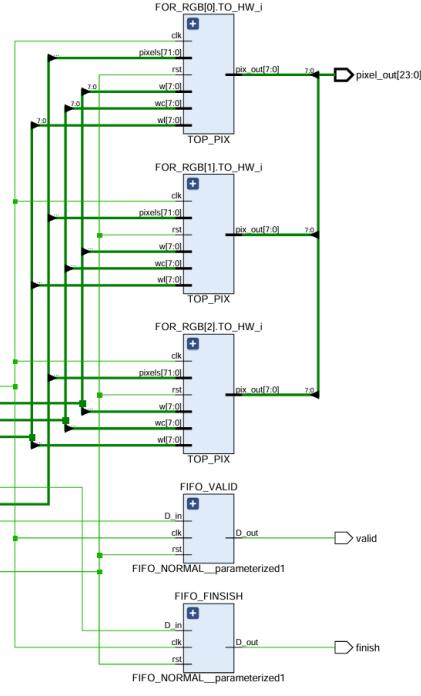
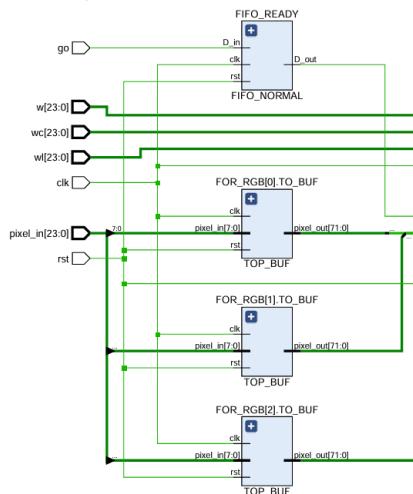
1. Nella prima versione progettata, la **FSM** non presenta la struttura di **Buffer Decode**, poiché viene implementata un'unica gestione dei bordi e pertanto non è possibile selezionarne una differente.
2. Nella seconda versione realizzata è possibile scegliere quattro differenti gestioni dei bordi.
3. La terza versione vanta di una **FSM** più performante in termini di frequenza di clock. Inoltre, permette di selezionare la dimensione dell'immagine da filtrare tramite variabili "generic", configurando l'hardware della scheda in base alla dimensione specificata prima dell'implementazione.
4. L'ultima versione, oltre a permettere all'utente di scegliere tra 4 differenti gestioni dei bordi, presenta un blocco di conversione che permette di passare dal formato **RGB** al formato **GRAYSCALE**, la procedura verrà spiegata nel **capitolo 7**.

1. TOP

In questo capitolo andremo a visualizzare e discutere i TOP di ogni progetto al fine di completare il quadro di ogni componenti descritta.

VERSIONE 1

```
entity TOP is
    generic ( DIM_FIFO_GO : integer := 34;
              DIM_FIFO_VALID: integer:= 11;
              DIM_FIFO_FINISH: integer:= 11);
    port(clk,rst,go: in std_logic;
          pixel_in: in std_logic_vector(23 downto 0);
          wc,wl,w: in std_logic_vector(23 downto 0);
          pixel_out: out std_logic_vector(23 downto 0);
          valid,finish: out std_logic);
end TOP;
```



Questa versione è la meno elaborata di tutte e si sviluppa nel seguente modo:

- sono presenti tre pixel ad 8 bit in un vettore unico *pixel_in* (*23 downto 0*), uno per colore; tre coefficienti a 24 bit poiché ad ogni colore può essere applicato un coefficiente di filtraggio differente. È presente anche un *go* che fungerà da trigger per la FSM.
- La **FIFO_NORMAL** è una FIFO a un singolo bit di ingresso e uno di uscita. Viene utilizzata per ritardare opportunamente il **valid**, il **finish** e il **go**, in modo tale che tutto si sincronizzino a dovere.
- Sono presenti tre buffer in parallelo (**TOP_BUF**) e tre Hardware in parallelo (**TOP_PIX**).
TOP_PIX è il blocco che prende in ingresso tutta la finestra disponibile al buffer e ne effettua l'operazione di convoluzione. Ogni coppia buffer-Hardware effettua il filtraggio del singolo colore.
- La **FSM** gestisce solo **valid** e **finish**.

```
--R
w(7 downto 0)<=conv_std_logic_vector(2,8);
wc(7 downto 0)<=conv_std_logic_vector(-1,8);
wl(7 downto 0)<=conv_std_logic_vector(1,8);
--
--G
w(15 downto 8)<=conv_std_logic_vector(10,8);
wc(15 downto 8)<=conv_std_logic_vector(-1,8);
wl(15 downto 8)<=conv_std_logic_vector(-1,8);
--
--B
w(23 downto 16)<=conv_std_logic_vector(10,8);
wc(23 downto 16)<=conv_std_logic_vector(-1,8);
wl(23 downto 16)<=conv_std_logic_vector(-1,8);
```

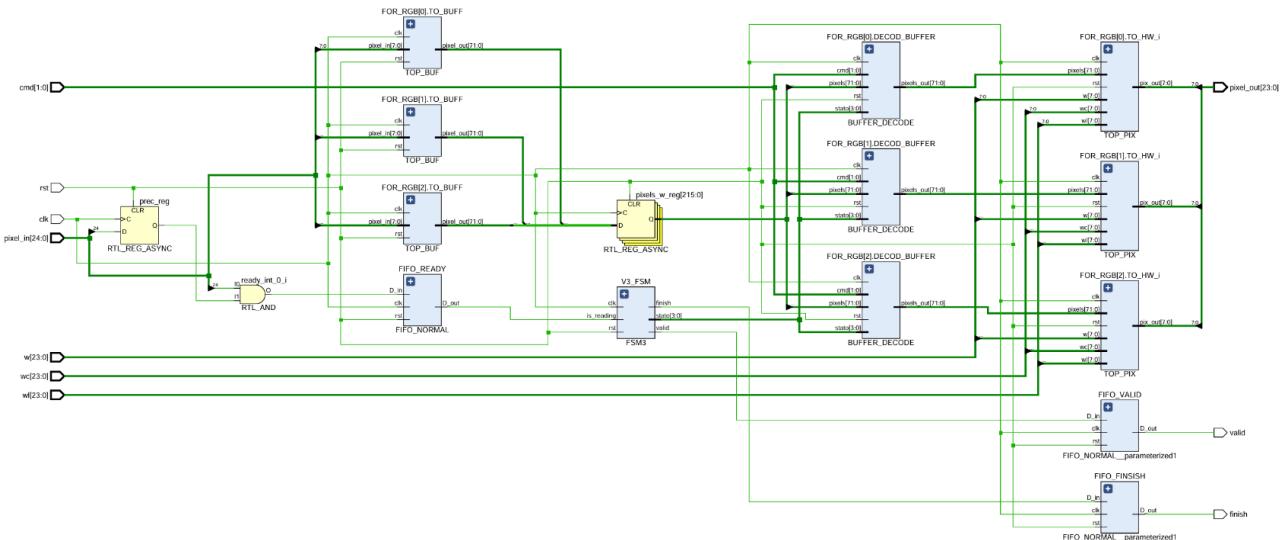
Esempio di coefficienti di filtraggio differenti su ogni colore

VERSIONE 2

```

entity TOP is
generic ( DIM_FIFO_GO : integer := 30;
           DIM_FIFO_VALID: integer:= 12;
           DIM_FIFO_FINISH: integer:= 12);
port(clk,rst: in std_logic;
      cmd: in std_logic_vector(1 downto 0);
      pixel_in: in std_logic_vector(24 downto 0);
      wc,wl,w: in std_logic_vector(23 downto 0);
      pixel_out: out std_logic_vector(23 downto 0);
      valid, finish: out std_logic);
) end TOP;

```



Dopodiché l'uscita di questa **AND** va in ingresso alla **FIFO_READY** la cui uscita, se alta, attiverà la **FSM**. Tutto ciò in linea teorica potrebbe essere evitato poiché il blocco **registro-AND** è del tutto inutile in behavioral e il segnale **pixel (24)** (il trigger della FSM) potrebbe essere inviato direttamente alla **FIFO_READY**.

Tuttavia, questo viene fatto poiché in post implementazione il segnale di **go** non viene captato correttamente: con questi accorgimenti abbiamo risolto il problema. In questa versione rispetto alla precedente sono presenti i tre **DECOD_BUFFER** e la **FSM** che riesce a gestire i bordi in funzione del comando **cmd** inviato dall'utente.

La **FSM**, oltre a gestire **VALID** e **FINISH**, comunica lo stato al **DECOD_BUFFER** che in funzione dello stato e del **cmd**, produce in uscita una nuova finestra di buffer, in modo che l'hardware effettui l'operazione di convoluzione con la finestra adatta alla gestione dei bordi scelta.

Nella seconda versione sono presenti alcune differenze che chiariremo di seguito. I pixel in ingresso sono 25 e non 24 poiché in questa versione il segnale di **go** viene mandato sul venticinquesimo pixel. Quest'ultimo va in ingresso ad un registro e l'uscita di questo registro va in una **AND**.

VERSIONE 3: 32x32 e 64x64

```

entity TOP is
generic(nv: integer := 16;
         nf: integer := 15;
         ddim: integer := 32;
         bdim: integer := 5 );
port(clk,rst, go: in std_logic;
cmd: in std_logic_vector(1 downto 0);
pixel_in: in std_logic_vector(23 downto 0);
wc,wl,w: in std_logic_vector(23 downto 0);
pixel_out: out std_logic_vector(23 downto 0);
valid, finish: out std_logic );
end TOP;

entity TOP is
generic(nv: integer := 16;
         nf: integer := 15;
         ddim: integer := 64;
         bdim: integer := 6 );
port(clk,rst, go: in std_logic;
cmd: in std_logic_vector(1 downto 0);
pixel_in: in std_logic_vector(23 downto 0);
wc,wl,w: in std_logic_vector(23 downto 0);
pixel_out: out std_logic_vector(23 downto 0);
valid, finish: out std_logic );
end TOP;

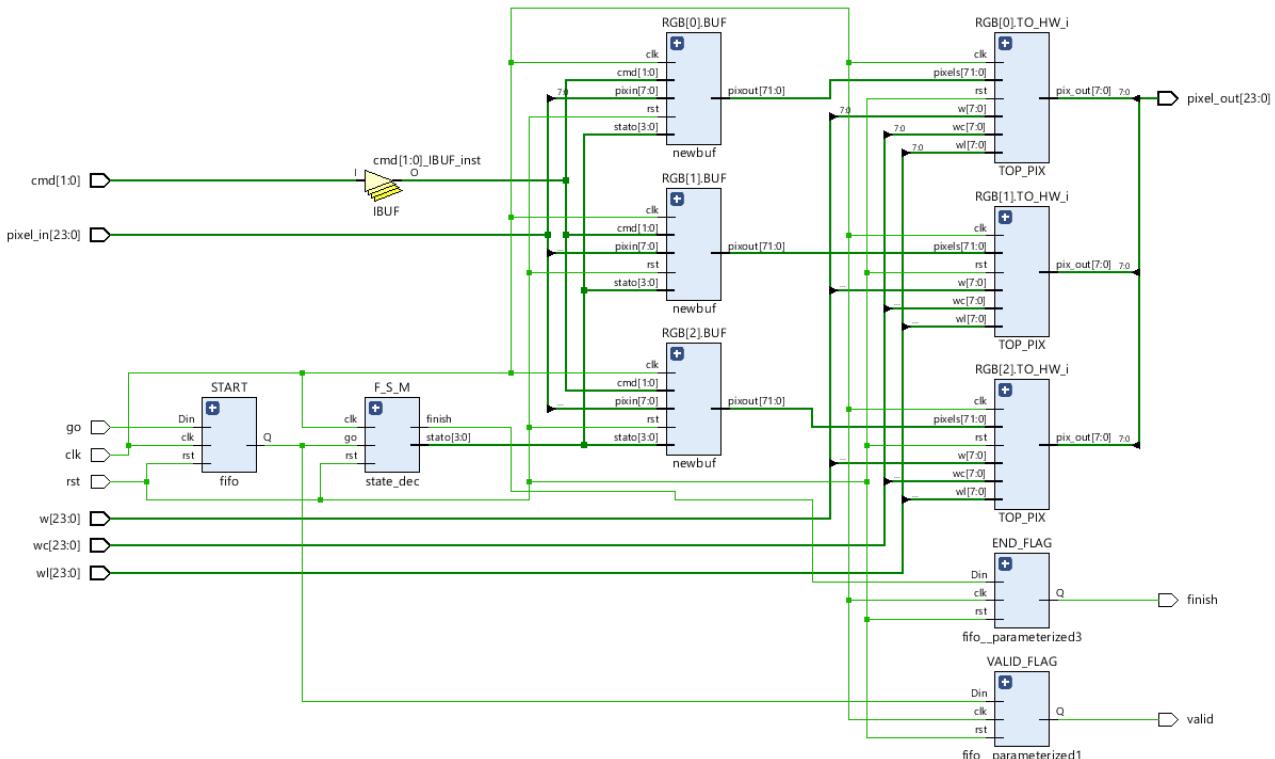
```

Questa terza tipologia di **circuito di filtraggio** mantiene invariate le caratteristiche funzionali della precedente versione, ma possiede un'architettura che consente di implementare più **stadi di pipeline** al suo interno, al fine di migliorare le performance in termini di massima frequenza di clock.

Inoltre, permette di selezionare dal modulo **Top** del circuito la dimensione dell'immagine da filtrare, attraverso l'inserimento opportuno di variabili di tipo "*generic*".

Tali variabili consentono di configurare l'hardware della scheda in funzione della dimensione inserita, prima della fase di implementazione.

Di seguito, è mostrato lo schema "gate level" del circuito.

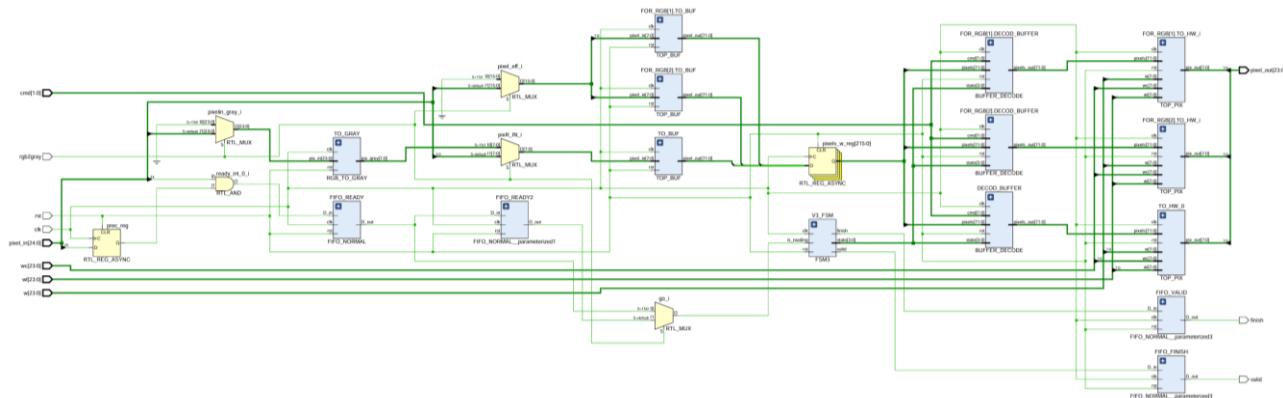


VERSIONE 4

```

entity TOP is
generic (
    DIM_FIFO_GO : integer := 30;
    DIM_FIFO_GO_GRAY: integer := 10;
    DIM_FIFO_VALID_FINISH: integer:= 12);
port(
    clk, rst, rgb2gray: in std_logic;
    cmd: in std_logic_vector(1 downto 0);
    pixel_in: in std_logic_vector(24 downto 0);
    wc,wl,w: in std_logic_vector(23 downto 0);
    pixel_out: out std_logic_vector(23 downto 0);
    valid, finish: out std_logic);
end TOP;

```



Questa versione utilizza la stessa **FSM** della **Versione 2** ma presenta il blocco aggiuntivo che permette la conversione a grigi ma ciò che vogliamo sottolineare qui sono i vari multiplexer presenti, i quali hanno una funzione di abbassare il consumo della potenza.

Questi multiplexer fanno in modo che, quando il segnale *rgb2gray* è basso, (non si vuole effettuare una conversione a grigi) vengano messi a zero i pixel in ingresso al blocco **RGB_TO_GRAY** riducendo il consumo di potenza dinamica; ma non è tutto. Quando *rgb2gray* è alto il blocco **RGB_TO_GRAY** dalla terna di pixel **RGB** in ingresso produce un singolo pixel in gray e quindi non sono necessari tutti i blocchi presenti, in quanto rimane da filtrare un'unica immagine e non tre, per questo motivo è presente un multiplexer il cui scopo è quello di porre a zero i pixel in ingresso ai **BUFFER** dei verdi e dei blu e utilizzare solo la sottostruttura che di solito è utilizzata per i rossi per effettuare il filtraggio dell'immagine in grigio.

Il segnale di *rgb2gray* funge anche da selettore per il trigger della **FSM** per un semplice motivo; il blocco **RGB_TO_GRAY**, per via degli stadi di pipeline al suo interno, ha una sua latenza che va sommata alla latenza generale del buffer, infatti, i pixel che vede in ingresso il buffer dei rossi quando il segnale *rgb2gray* è alto, sono proprio i pixel di uscita del blocco di conversione, per questo motivo bisogna eventualmente ritardare ulteriormente il trigger **FSM** di un tempo uguale alla latenza del blocco di **RGB_TO_GRAY**.

2. CARRY-SAVE-8BIT A 4 OPERANDI

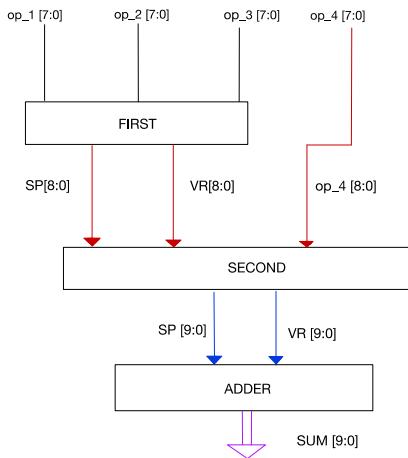


Figura 1.3: Struttura Carry-Save con 4 operandi a 8 bit

Siccome il circuito è predisposto per un filtro isotropico 3x3, i *Carry Save a 8 bit* ci servono per sommare i valori dei pixel agli angoli e ai lati della finestra da filtrare. Vanno sommati 4 pixel unsigned per cui abbiamo 4 operandi in ingresso. Nella descrizione dell'entity inoltre sono presenti un ulteriore ingresso “*pix_to_delay*” e un ulteriore uscita “*central_pix*”, con

l'obiettivo di prendere in ingresso il pixel centrale, che è l'unico da non sommare, e indirizzarlo in una catena di registri pari alla pipeline del CSA, per fare in modo che in uscita il pixel centrale sia sincronizzato con le somme dei pixel ai lati e ai corner.

```

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity CARRY_SAVE4_8 is --la somma è totalmente unsigned!
Port (clk, rst: in std_logic;
      OP1,OP2,OP3,OP4:IN STD_LOGIC_VECTOR(7 DOWNTO 0);
      pix_to_delay: in std_logic_vector(7 downto 0);
      FINAL_RIS, central_pix:OUT STD_LOGIC_VECTOR(9 DOWNTO 0));
end CARRY_SAVE4_8;

component FIRST is
Port (clk, rst: in std_logic;
      op1,op2,op3,op4:in std_logic_vector(7 downto 0);
      pix_to_delay: in std_logic_vector(7 downto 0);
      SP,VR:OUT STD_LOGIC_VECTOR(7 DOWNTO 0);
      op4_out,central_pix: out std_logic_vector(7 downto 0));
end component;

component SECOND is
Port (clk, rst: in std_logic;
      op1,op2,op3,pix_to_delay: in std_logic_vector(7 downto 0);
      SP,VR:OUT STD_LOGIC_VECTOR(8 DOWNTO 0);
      central_pix: out std_logic_vector(7 downto 0));
end component;

```

Essendo una somma a quattro operandi abbiamo necessariamente bisogno di tre livelli, chiamati nel nostro caso **FIRST** e **SECOND** e **PIPE_TO_ADDER**, per sommare i singoli bit degli operandi, eseguire uno shift del vettore dei riporti ed estendere il vettore di somma parziale.

Anche se **FIRST** e **SECOND** svolgono la stessa funzione è necessario distinguerli in quanto nel secondo livello trattiamo ingressi estesi di 1 bit.

```

component PIPE_TO_ADDER is
port ( A,B : in std_logic_vector (8 downto 0);
      Cin, clock, reset : in std_logic;
      pix_to_delay: in std_logic_vector(7 downto 0);
      Sum,central_pix : out std_logic_vector (9 downto 0));
end component;

SIGNAL VR1,SP1: STD_LOGIC_VECTOR(7 DOWNTO 0);
SIGNAL VR2,SP2: STD_LOGIC_VECTOR(8 DOWNTO 0);
signal op4_out: std_logic_vector(7 downto 0);
signal pix_temp:std_logic_vector(15 downto 0);

begin

BLOCCO1: FIRST port map(clk, rst, OP1, OP2, OP3, OP4,
                         pix_to_delay, SP1, VR1, op4_out,
                         pix_temp(7 downto 0));

BLOCCO2: SECOND port map(clk, rst, SP1, VR1, op4_out,
                         pix_temp(7 downto 0), SP2, VR2,
                         pix_temp(15 downto 8));

FINAL: PIPE_TO_ADDER port map(SP2, VR2, '0', clk, rst,
                               pix_temp(15 downto 8),
                               FINAL_RIS, central_pix);

end Behavioral;

```

L'ultimo stadio **PIPE_TO_ADDER** esegue l'effettiva somma tra l'ultima somma parziale e il vettore dei riporti; ovviamente il c_{in} è zero ed è ininfluente (*la dichiarazione di un carry in ingresso "Cin" nell'entity è data da una comodità di riciclare un sommatore già fatto in precedenza*).

Ogni blocco presente possiede per **ogni uscita** dei registri che svolgono la funzione di **pipeline** (il comportamento è descritto all'interno di un *process*, sensibile al fronte di salita del clock).

2.1.1 FIRST (1 LIVELLO CSA_8BIT)

```

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity FIRST is
Port (clk, rst: in std_logic;
      op1,op2,op3,op4:in std_logic_vector(7 downto 0);
      pix_to_delay: in std_logic_vector(7 downto 0);
      SP,VR:OUT STD_LOGIC_VECTOR(8 DOWNTO 0);
      op4_out,central_pix: out std_logic_vector(7 downto 0));
end FIRST;

architecture Behavioral of FIRST is
signal sp_int, vr_int: std_logic_vector(8 downto 0);

component FA is
Port (op1,op2,op3:in std_logic;
      SP,VR:OUT STD_LOGIC);
end component;

begin

FA_f: for i in 0 to 7 generate
FA_i: FA port map (op1(i), op2(i), op3(i), sp_int(i), vr_int(i+1));
end generate;

-- sp_int(8) <= sp_int(7);
sp_int(8) <='0';

vr_int(0) <= '0';

process(clk,rst)
begin
  if rst = '1' then
    SP <= (others => '0');
    VR <= (others => '0');
    op4_out <= (others=>'0');
    central_pix<=(others=>'0');
  elsif rising_edge(clk) then
    SP <= sp_int;
    VR <= vr_int;
    op4_out <= op4;
    central_pix<=pix_to_delay;
  end if;
end process;

end Behavioral;

```

I segnali di uscita sono estesi direttamente con zero in quanto stiamo sommando pixel dell'immagine che sappiamo essere unsigned per specifica data. Il reset è asincrono quindi fuori dalla condizione di rising edge del clock. In maniera analoga, a meno di un'estensione degli ingressi e dell'uscite, è descritto il blocco **SECOND**.

2.1.2 BLOCCO FA

```

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity FA is
  Port (op1,op2,op3:in STD_LOGIC;
        SP,VR:OUT STD_LOGIC);
end FA;

architecture Behavioral of FA is
  signal p,g,s,r: STD_LOGIC;
begin

  p<= op1 xor op2;
  g<= p and op2;
  s<= p xor op3;
  VR<=(p AND op3) OR (op1 and op2);

end Behavioral;

```

Nell'immagine è descritto un **Full Adder** che implementa la logica del **Carry Look Ahead**. Nel dettaglio, il componente esegue l'operazione di somma tra due bit e un carry in, che in questo caso sono denominati op1, op2 e op3. Viene generato un bit di somma e uno di carry out. Otto di questi componenti vengono utilizzati in parallelo per i blocchi FIRST e SECOND.

2.2 PIPE_TO_ADDER

```

entity PIPE_TO_ADDER is
port ( A,B : in std_logic_vector (8 downto 0);
      Cin, clock, reset : in std_logic;
      pix_to_delay: in std_logic_vector(7 downto 0);
      Sum,central_pix : out std_logic_vector (9 downto 0));
end PIPE_TO_ADDER;

architecture Behavioral of PIPE_TO_ADDER is
signal sum_p : std_logic_vector(12 downto 0);

component ADDER12 is
port ( A,B : in std_logic_vector (11 downto 0);
      Cin : in std_logic;
      Sum : out std_logic_vector (12 downto 0));
end component;

signal Aint,Bint: std_logic_vector(11 downto 0);

begin
  begin
    Aint<="000"&A;
    Bint<="00"&B&'0';

    dut: ADDER12 port map (Aint,Bint,Cin,sum_p);

    process(clock, reset)
    begin
      if (reset = '1') then
        sum <= (others=>'0');
        central_pix<=(others=>'0');
      elsif(rising_edge(clock)) then
        sum<= sum_p(9 downto 0);
        central_pix<="00"&pix_to_delay;
      end if;
    end process;
  end Behavioral;

```

Pipe_to_Adder è l'ultimo stadio del CSA che ci restituisce la somma complessiva tra il vettore somma e quello dei riporti. È stato necessario implementare un Adder a 12 bit che sarà spiegato in seguito. Ad ogni modo l'uscita finale di questa struttura non è sovradimensionata.

2.3 ADDER_12

```

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity ADDER12 is
port ( A,B : in std_logic_vector (11 downto 0);
Cin : in std_logic;
Sum : out std_logic_vector (12 downto 0);
end ADDER12;

architecture CSA of ADDER12 is
signal C_int : std_logic_vector(3 downto 0);

component FastRCA_4bit is
port(a,b : in std_logic_vector (3 downto 0);
Cin : in std_logic;
Sic : out std_logic_vector (3 downto 0);
Cout: out std_logic);
end component;

begin
    begin
        C_int(0) <= Cin;
        FOR_GEN: for i in 0 to 2 generate
            RCA4: FastRCA_4bit port map (A(3+4*i downto 4*i),
B(3+4*i downto 4*i),
C_int(i),
Sum(3+4*i downto 4*i),
C_int(i+1));
        end generate;
        Sum(12)<= C_int(3);
    end CSA;

```

In generale il risultato che ci si aspetta da un **Carry Save** a 4 operandi a n bit è **n+2 bit**, nel caso specifico quindi la somma finale deve uscire a **10 bit**. Tuttavia, anziché descrivere un sommatore a 10 bit per la somma finale è stato necessario descriverne uno più grande a **12 bit**, al fine di sfruttare le *carry chains*, strutture hardware dedicate nei chip FPGA della famiglia **Virtex7** che permettono delle performance migliori.

2.3.1 FastRCA_4BIT

Affinché l'ADDER12 sia implementato con le *carry chain*, al fine di **ridurre al minimo il ritardo del sommatore**, è stata usata la libreria “**UNISIM**” che consente di istanziare (usare come component senza dichiarare) la primitiva **CARRY4** della famiglia **FPGA Virtex7**.

```

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

library UNISIM;
use UNISIM.vcomponents.all;--CARRY4:FastCarryLogicComponent--7Series;

entity FastRCA_4bit is
port(a,b : in std_logic_vector (3 downto 0); |
Cin : in std_logic;
Sic : out std_logic_vector (3 downto 0);
Cout : out std_logic);
end FastRCA_4bit;

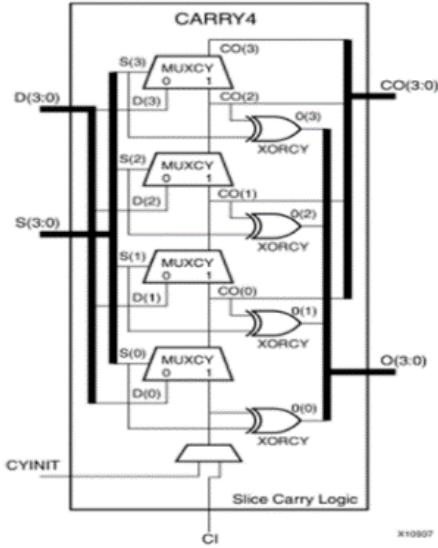
architecture Behavioral of FastRCA_4bit is
signal p_int, co_int: std_logic_vector (3 downto 0);

begin
    begin
        CARRY4_inst: CARRY4
        port map(
            CO=>co_int,--4-bitcarryout
            O=>Sic,--4-bitcarrychainXORda
            CI=>Cin,--1-bitcarrycascadein
            CYINIT=>'0',--1-bitcarryiniti
            DI=>a,--4-bitcarry-MUXdatain
            S=>p_int--4-bitcarry-MUXselectinput
        );--EndofCARRY4_instinstantiation

        Cout <= co_int(3);

    end Behavioral;

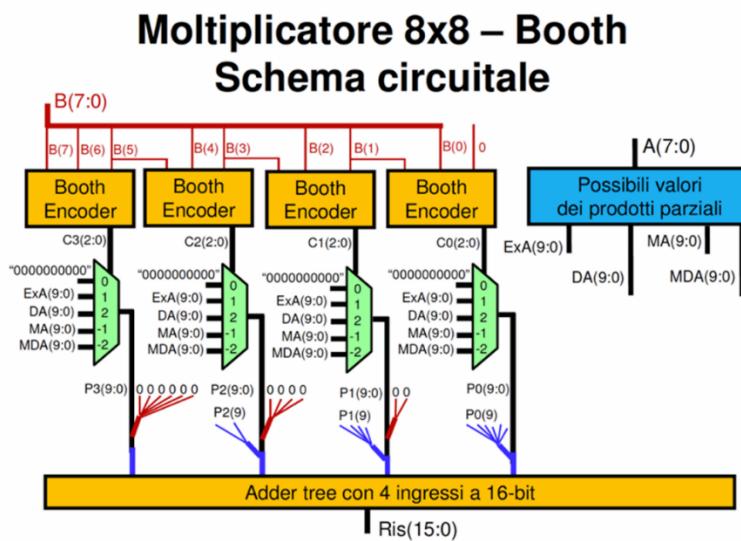
```



3. BOOTH MULTIPLIER

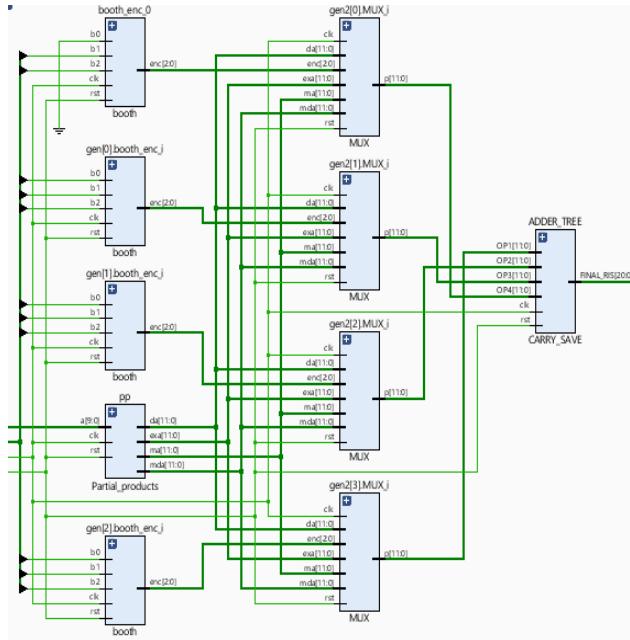
Per effettuare il prodotto tra ciascuna *somma di pixel* (operando A a 10 bit) e i *coefficienti del filtro* (operando B ad 8 bit) è stato utilizzato il *moltiplicatore di Booth*. La prima operazione che viene eseguita è codificare l'operando B; sono prese quindi delle terne opportune, ciascuna con 1 bit di overlap con la terna precedente.

Nel caso in questione sono necessari **4 booth encoder** che si occuperanno della codifica di ogni terna parallelamente. A partire dall'operando A, vengono estratti tutti i possibili *prodotti parziali* che andranno poi in ingresso a quattro *multiplexer* in parallelo, i cui selettori sono proprio i valori codificati dalle singole terne estratte dall'operando B attraverso i blocchi di booth encoder. Ogni multiplexer darà in uscita uno specifico prodotto parziale.



Ciascuna terna assume un determinato peso. Per rispettare il peso di queste terne, i quattro prodotti parziali da sommare dovranno essere opportunamente estesi e shiftati verso sinistra; perciò, nel blocco di somma **CARRY_SAVE** si esegue questo passaggio tramite *estensione con zeri* ed *estensione con segno*. Il modo in cui si effettua quest'operazione verrà spiegata nel **sotto-paragrafo 3.3**.

Successivamente, per ottenere il risultato finale, i prodotti parziali vengono sommati facendo uso di un *Carry Save*.



```

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.std_logic_unsigned.all;

entity TOP_MULTI is
port(b: in std_logic_vector(7 downto 0); --- b sono i coefficienti del filtro che sono signed
a: in std logic vector(5 downto 0);
clk,reset: in std logic;
P_out: out std_logic_vector(17 downto 0)
);
end TOP_MULTI;

architecture Behavioral of TOP_MULTI is

component CARRY_SAVE is
Port (clk, rst: in std_logic;
      OF1,OF2,OF3,OF4:IN STD_LOGIC_VECTOR(11 DOWNTO 0);
      FINAL_RES:OUT STD_LOGIC_VECTOR(20 DOWNT0 0));
end component;

component Partial_products is
Port(a: in std_logic_vector(9 downto 0);
exa, da, mda, ma: out std_logic_vector(11 downto 0);
clk,rst: in std logic);
end component;

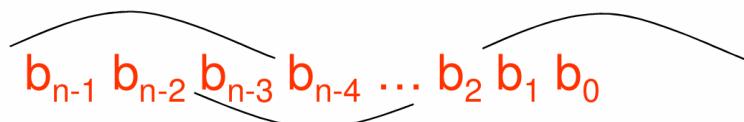
component MUX is
Port(exa, da, ma, mda: in std_logic_vector(11 downto 0);
enc: in std logic vector(2 downto 0);
clk,rst: in std logic;
p : out std_logic_vector(11 downto 0));
end component;

component booth is
Port (clk, rst: in std_logic;
      b0, b1, b2: in std_logic;
      enc: out std_logic_vector(2 downto 0));
end component;

```

3.1 BOOTH ENCODER

Questo componente, per ogni terna di bit dell'operando B, estrapola un digit rappresentato con 3 bit. Ogni terna è presa come illustrato di seguito:



Essendo l'operando signed ad 8 bit sono stati realizzati *quattro* booth encoder. Di seguito viene riportato il codice di ciascun codificatore utilizzato.

```

library IEEE;
use IEEE.STD_LOGIC_1164.all;

entity booth is
    Port (clk, rst: in std_logic;
          b0, b1, b2: in std_logic;
          enc: out std_logic_vector(3 downto 0));
end booth;

architecture Behavioral of booth is

    signal a_int, b_int, c_int, b0_p, b1_p, b2_p: std_logic;
    signal enc_int: std_logic_vector(2 downto 0);

begin
    a_int <= b0_p xor b1_p;
    b_int <= b1_p xor b2_p;
    c_int <= b1_p and b0_p;
    enc_int(0) <= a_int;
    enc_int(1) <= (not a_int) and b_int;
    enc_int(2) <= c_int and b2_p;

process(clk,rst)
begin
    if rst='1' then
        enc <= (others => '0');
        b0_p <= '0';
        b1_p <= '0';
        b2_p <= '0';
    elsif rising_edge(clk) then
        b0_p <= b0;
        b1_p <= b1;
        b2_p <= b2;
        enc <= enc_int;
    end if;
end process;

```

3.2 PARTIAL PRODUCTS

```

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use ieee.std_logic_unsigned.all;

entity Partial_products is
  Port( a: in std_logic_vector(9 downto 0);
  exa, da, mda, ma: out std_logic_vector(11 downto 0);
  clk,rst: in std_logic);
end Partial_products;

architecture Behavioral of Partial_products is

signal da_p,ma_p, mda_p,exa_p: std_logic_vector(11 downto 0);
signal a_p:  std_logic_vector(9 downto 0);
signal a_ext, comp_2, temp: std_logic_vector(10 downto 0);

component plus is
  Port(a : in std_logic_vector(10 downto 0);
  a_out: out std_logic_vector(10 downto 0));
end component;

begin
  a_ext <= '0' & a;
  temp <= a_ext xor "1111111111";
  plus_1: plus port map (temp,comp_2);
  mda_p <= comp_2 & '0';
  ma_p <= comp_2(10) & comp_2;
  exa_p<= '0' & a_ext;
  da_p <= a_ext & '0';

  | process(clk,rst)
  begin
    | if rst = '1' then
      exa<= (others => '0');
      da<= (others => '0');
      mda<= (others => '0');
      ma<= (others => '0');
      a_p <= (others=>'0');
    | elsif rising_edge(clk) then
      exa<= exa_p;
      mda<=mda_p;
      ma<=ma_p;
      da<=da_p;
      a_p<=a;
    | end if;
  | end process;
| end Behavioral;

```

L'operando A è a **10 bit**, poiché corrisponde all' uscita del *Carry Save* a 3 operandi da 8 bit. Le uscite del blocco “**Partial_products**” sono i *valori* che può assumere il prodotto parziale ovvero $-A$ (*ma*), A (*exa*), $-2A$ (*mda*) e $2A$ (*da*).

Lo 0 non viene inserito dal momento che viene semplicemente messo come ingresso nel multiplexer. Per la regola del Moltiplicatore di Booth, dato che l'operando A è a 10 bit, ogni prodotto parziale sarà a 12 bit.

Per ottenere i valori negativi si esegue:

$$-2 * A = 2 * (2's A)$$

Il valore in complemento a 2 è dato da:

$$2's(A) = \text{not}(A) + '1'$$

exa	A
mda	-2A
da	2A
ma	-A
a_p	0

L'operazione logica *not(a_ext)* si ottiene invertendo tutti i bit dell'operando, tramite l'operazione di *xor* bit a bit con “1111111111”. L'operazione di somma con 1 viene effettuata tramite il blocco plus che completa il 2's (A). Per ottenere tutti i prodotti parziali si utilizzano i segnali *a_ext* e *comp2* quest' ultimo è proprio il *2's(a_ext)*, infatti:

1. **-A (ma)** è ottenuto estendendo con segno *comp2*.
2. **A (exa)** è ottenuto estendo con segno il segnale *a_ext* in ingresso.
3. **-2A (mda)** si ricava shiftando verso sinistra *comp2*.
4. **2A (da)** si ricava shiftando verso sinistra il segnale *a_ext* in ingresso.

3.3 PLUS

```

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity plus is
Port(a : in std_logic_vector(10 downto 0);
a_out: out std_logic_vector(10 downto 0));
end plus;

architecture Behavioral of plus is
signal c : std_logic_vector(10 downto 0);

begin
c(0)<= a(0);
a_out(0) <= not a(0);
logic:
for i in 1 to 10 generate
c(i)<=a(i) and c(i-1);
a_out(i) <= a(i) xor c(i-1);
end generate;

end Behavioral;

```

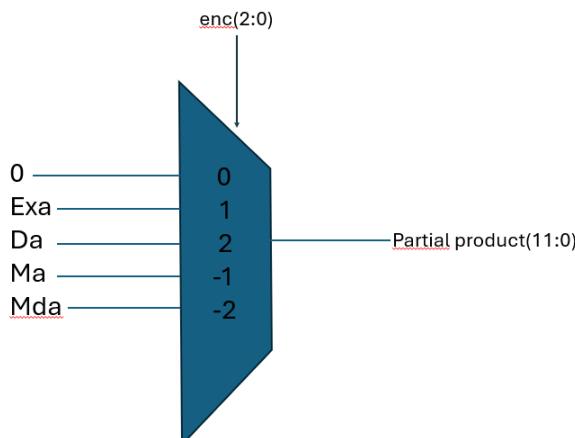
Il blocco "Plus" esegue la somma di +1 dell'operando in ingresso. Questa operazione è necessaria per completare la rappresentazione del valore A in complemento a due. La somma +1 avviene tramite una logica sviluppata appositamente per ottimizzare tale operazione. L'ingresso e l'uscita sono ad **11 bit**, questa cosa può sembrare un errore poiché la somma tra due numeri a **n bit** deve essere di **n+1 bit** per evitare l'overflow. In realtà qui il problema non si pone poiché in ingresso a questa struttura vi è l'operando *temp*, composto dai bit di *a_ext* flippati. L'overflow si verifica quando in ingresso al blocco plus il segnale *a* = (*others=>'1'*), ma ciò avviene solo se *a_ext* è zero poiché:

$$\text{temp} \leq a_{\text{ext}} \text{ xor } "11\dots1".$$

All'interno di questo blocco quindi a *temp* sarà sommato uno, ma se è composto da tutti uno allora il risultato sarà tutto zero, il che è corretto poiché il complemento a due di zero è proprio zero.

3.4 MULTIPLEXER PER I PRODOTTI PARZIALI

Tutti i possibili valori dei prodotti parziali vanno in ingresso al multiplexer, il cui seletore sarà un vettore di tre bit prodotto dal booth encoder a partire dalla terna in ingresso; pertanto, il numero di multiplexer utilizzati è pari al numero di codificatori di Booth.



```

entity MUX is
Port(exa, da, ma, mda: in std_logic_vector(11 downto 0);
enc: in std_logic_vector(2 downto 0);
clk, rst: in std_logic;
p : out std_logic_vector(11 downto 0));
end MUX;

architecture Behavioral of MUX is
signal p_P : std_logic_vector(11 downto 0);

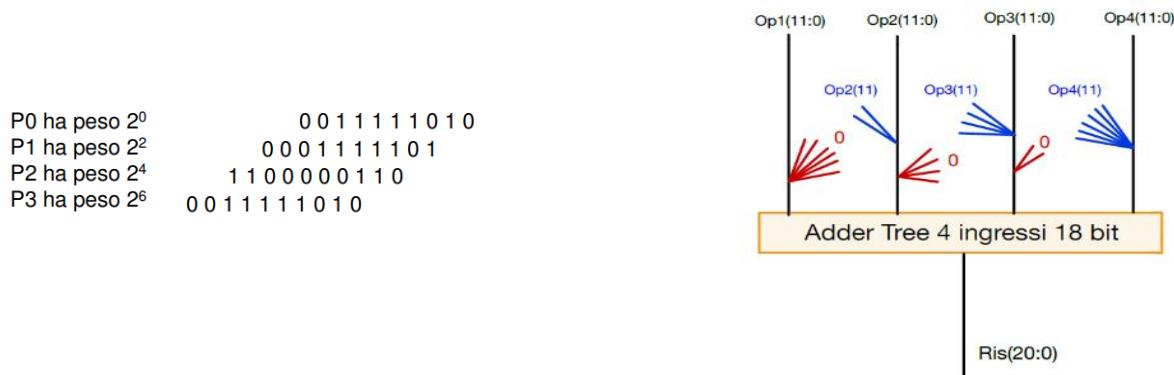
begin
with enc select
p_P<= exa when "001",
da when "010",
ma when "101",
mda when "110",
(others=>'0') when others;

process(clk, rst)
begin
if rst = '1' then
p<= (others=>'0');
elsif(rising_edge(clk)) then
p<= p_P;
end if;
end process;
end Behavioral;

```

3.5 SOMMA DEI PRODOTTI PARZIALI

Una volta scelti dai multiplexer, i diversi prodotti parziali vengono sommati. Prima però vanno estesi e shiftati. Quest'operazione è necessaria in quanto i prodotti parziali sono a 12 bit e per poter ottenere il risultato in maniera corretta vanno estesi fino a 18 bit. L'allineamento tiene conto del peso del bit centrale di ciascuna terna.



Ci si aspetta che il risultato finale della moltiplicazione sia a 18 bit, eppure l'uscita dell'ADDER finale è di 21 bit, questo perché internamente all'ultimo ADDER del moltiplicatore a valle è presente un ADDER20 costruito con cinque carry chain, le quali rappresentano un vincolo perché sono dei ripple carry a 4 bit ciascuno, ed è localmente necessaria un'estensione di un bit per adattare la dimensione degli ingressi all'ADDER20.

Si noti che nel grafico le *estensioni con zero* sono rappresentate dalle linee rosse e quelle *con segno* si identificano tramite le connessioni in blu. **Questa operazione viene eseguita nel blocco STEP1 del Carry Save** prima di ottenere il vettore dei riporti e quello delle somme parziali secondo l'approccio descritto nel paragrafo 2.1.1. Il valore op_4 è quello relativo alla terna con peso 2^0 , è il prodotto parziale meno significativo; pertanto, non deve essere shiftato verso sinistra ma solo esteso con segno di 6 bit, op_1 invece è il valore col peso maggiore (2^6), quindi il prodotto parziale più significativo; quindi, dovrà essere solo esteso con sei zeri verso sinistra rispetto ad op_4.

```

op1_int <= op1 & "000000";
op2_int <= op2(11) & op2(11) & op2 & "0000";
op3_int <= op3(11) & op3(11) & op3(11) & op3(11) & op3 & "00";
op: for i in 0 to 6 generate
    op4_int(i+12) <= op4(11);
end generate;
op4_int(11 downto 0)<= op4(11 downto 0);
  
```

Estensione degli operandi effettuata nel blocco STEP1 interno al CARRY_SAVE

Una volta allineati questi prodotti parziali, vengono sommati nel Carry Save Adder che funziona come quello descritto nel capitolo 2, l'unica differenza è che viene adattato alle

dimensioni opportune. L'uscita è a 21 bit ma la saturazione avviene banalmente utilizzando da questo blocco solo i primi 18 bit della sua uscita.

```

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity CARRY_SAVE is
  Port (clk, rst: in std_logic;
        OP1,OP2,OP3,OP4:IN STD_LOGIC_VECTOR(11 DOWNTO 0);
        FINAL_RIS:OUT STD_LOGIC_VECTOR(20 DOWNTO 0));
end CARRY_SAVE;

architecture Behavioral of CARRY_SAVE is

  SIGNAL VR1,SP1: STD_LOGIC_VECTOR(18 DOWNTO 0);
  SIGNAL VR2,SP2: STD_LOGIC_VECTOR(19 DOWNTO 0);

  component STEP1 is
    Port (clk, rst: in std_logic;
          op1,op2,op3,op4:in std_logic_vector(11 downto 0);
          SP,VR, op4_out:OUT STD_LOGIC_VECTOR(18 DOWNTO 0));
  end component;

  component STEP2 is
    Port (clk, rst: in std_logic;
          op1,op2,op3:in std_logic_vector(18 downto 0);
          SP,VR:OUT STD_LOGIC_VECTOR(19 DOWNTO 0));
  end component;

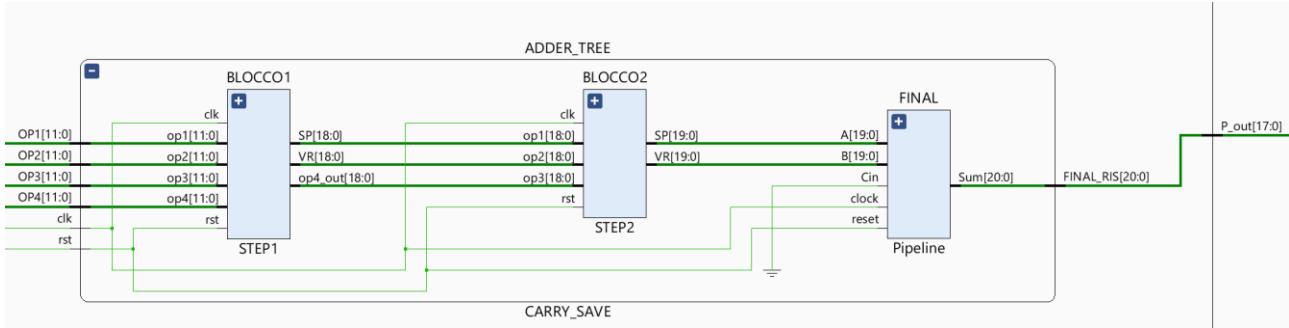
  component Pipeline is
    port ( A,B : in std_logic_vector (19 downto 0);
           Cin, clock, reset : in std_logic;
           Sum : out std_logic_vector (20 downto 0));
  end component;

  signal op4_out: std_logic_vector(18 downto 0);

begin

  begin
    BLOCCO1: STEP1 port map(clk, rst, OP1, OP2, OP3, OP4, SP1, op4_out);
    BLOCCO2: STEP2 port map(clk, rst, SP1, VR1, op4_out, SP2, VR2);
    FINAL: Pipeline port map(SP2, VR2, '0', clk, rst, FINAL_RIS);
  end Behavioral;

```



4. SOMMA FINALE

Una volta ottenuti i prodotti con i coefficienti del filtro, per completare la convoluzione i tre valori in uscita dai moltiplicatori vengono sommati attraverso un Carry Save a tre operandi da 18 bit. In uscita questo componente darà il pixel filtrato a 20 bit.

```

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

```

```

entity CARRY_SAVE_3 is
  Port (clk, rst: in std_logic;
        OP1,OP2,OP3:IN STD_LOGIC_VECTOR(17 DOWNTO 0);
        FINAL_RIS:OUT STD_LOGIC_VECTOR(19 DOWNTO 0));
  end CARRY_SAVE_3;

architecture Behavioral of CARRY_SAVE_3 is

  SIGNAL VR1,SP1: STD_LOGIC_VECTOR(18 DOWNTO 0);

  component STEP1_3 is
    Port (clk, rst: in std_logic;
          op1,op2,op3:in std_logic_vector(17 downto 0);
          SP,VR:OUT STD_LOGIC_VECTOR(18 DOWNTO 0));
  end component;

  component Pipeline_3 is
    port ( A,B : in std_logic_vector (18 downto 0);
           clock, reset, cin : in std_logic;
           Sum : out std_logic_vector (19 downto 0));
  end component;

  begin
    BLOCCO1: STEP1_3 port map(clk, rst, OP1, OP2, OP3, SP1, VR1);
    FINAL: Pipeline_3 port map(SP1, VR1, clk, rst,'0', FINAL_RIS);
  end Behavioral;

```

Nel caso specifico di tre operandi, il CSA è costituito da un singolo livello, chiamato **STEP1_3**, in cui viene effettuata la somma dei singoli bit e lo shift del vettore dei riporti ed è presente un registro di pipeline per ogni uscita.

È presente, infine, il blocco di somma finale con relativa pipeline, **Pipeline_3**.

5. BLOCCO DI SATURAZIONE

Il pixel filtrato ottenuto, come richiesto dalla specifica, va saturato ad 8 bit unsigned, ciò vuol dire che il suo valore deve essere compreso tra 0 e 255, ovvero il range di valori rappresentabili utilizzando la notazione ad 8 bit.

In particolare, **qualsiasi valore di “pix_in” (che è un numero signed) superiore a +255 unsigned deve essere saturato a 255 unsigned, mentre qualsiasi valore inferiore a 0 deve essere impostato a 0**. Durante questa operazione, è necessario convertire il valore signed a 20 bit in un valore unsigned a 8 bit.

Sono stati dichiarati due segnali interni “X” e “bool”.

Usiamo **bool** e **pix_in (19)** come discriminante per capire se bisogna saturare o meno.

```

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity SATURATION is
  Port( clk,rst: in std_logic;
        pix_in: in std_logic_vector(19 downto 0);
        pix_sat: out std_logic_vector(7 downto 0));
end SATURATION;

architecture Behavioral of SATURATION is

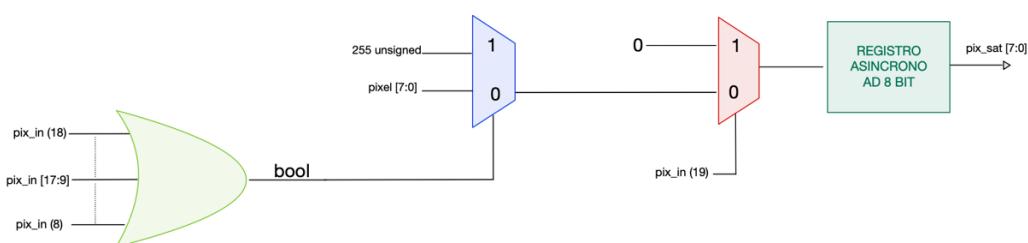
signal X: std_logic_vector(7 downto 0);
signal bool : std_logic;

begin
  process(clk,rst)
  begin
    if(rst='1') then
      pix_sat<=(others=>'0');
    elsif(rising_edge(clk)) then
      if(pix_in(19)='1') then
        pix_sat<=(others=>'0');
      elsif(bool='1') then
        pix_sat<=(others=>'1');
      else
        pix_sat<=X;
      end if;
    end if;
  end process;
end Behavioral;

```

La logica utilizzata è la seguente: inizialmente si verifica se il numero è positivo andando a controllare l'MSB del pixel in ingresso: se è 1 allora il pixel in uscita dall'Adder è negativo e va saturato a zero, altrimenti se uno dei bit che vanno dal diciannovesimo al nono è 1 allora bisogna saturare il valore a 255.

Per verificare quest'ultima condizione si fa la **OR** tra tutti i bit dal 19esimo al nono del pixel in ingresso e il risultato va nella variabile **bool**: se è 1 allora in uscita saturo a 255, se è 0 semplicemente non devo saturare nulla(il numero sarà sicuramente compreso tra 0 e 255), quindi gli 8 bit del pixel in ingresso vanno in uscita e gli altri non vengono connessi in uscita, altrimenti se non fosse né 0 né 1 il segnale X, che è indeterminato, andrà in uscita, ciò è necessario poiché vanno coperti tutti i possibili casi.



6. MACCHINA A STATI FINITI E DECODERS

Nel progetto sono state implementate tre diverse **Macchine a Stati Finiti** (FSM) per la gestione del filtraggio delle immagini.

In particolare, la prima FSM, denominata **FSM_V2** all'interno del progetto, non implementa alcuna gestione dei bordi e presenta una struttura minimale. La seconda FSM, denominata **FSM3**, offre invece la possibilità di gestire i bordi scegliendo tra quattro modalità differenti. La terza **FSM** permette di definire dal top la dimensione dell'immagine da filtrare.

6.1 FSM MINIMALE

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

entity FSM_V2 is
Port( clock, reset,go: in std_logic;
pixels: in std_logic_vector(215 downto 0);
pixels_out: out std_logic_vector(215 downto 0);
valid,finish: out std_logic
);
end FSM_V2;

architecture Behavioral of FSM_V2 is
type state_type is (idle,start);
signal state : state_type;
signal count: std_logic_vector(9 downto 0);
signal reset_int: std_logic;
begin
process (clock, reset)
begin
if(reset = '1') then
state <= idle;
pixels_out<=(others=>'0');
count<=(others=>'0');
reset_int<='0';
valid<='0';
finish<='0';
elsif (rising_edge(clock)) then
case state is
when idle => count<=(others=>'0');
if go='1' and reset_int = '0' then
state<= start;
pixels_out<=pixels;
else pixels_out<=(others=>'0');
end if;
when start => valid<='1';
pixels_out<=pixels;
count<=count+1;
if count="111111111" then
finish<='1';
state<=idle;
reset_int<='1';
end if;
end case;
end if;
end process;
end Behavioral;
```

La struttura descritta in alto viene impiegata per implementare una gestione dei bordi dell'immagine di tipo **toroidale-mista**.

La FSM è **minimale** in quanto questo tipo di gestione dei bordi è effettuata dalla struttura del buffer seriale che è adatta ad un filtro 3x3.

Questa FSM prende in ingresso tutte le uscite dei buffer dell'immagine RGB, insieme al segnale “**go**” e ai classici “**clock**” e “**reset**”.

Questa FSM comanda il segnale di “**valid**” che si alzerà non appena sarà disponibile la prima finestra da filtrare, ovvero quando il primo pixel raggiunge la posizione centrale del buffer.

Il segnale di **finish** viene alzato dopo **1024** colpi di clock, esattamente quando l'ultimo pixel da filtrare raggiunge la posizione centrale.

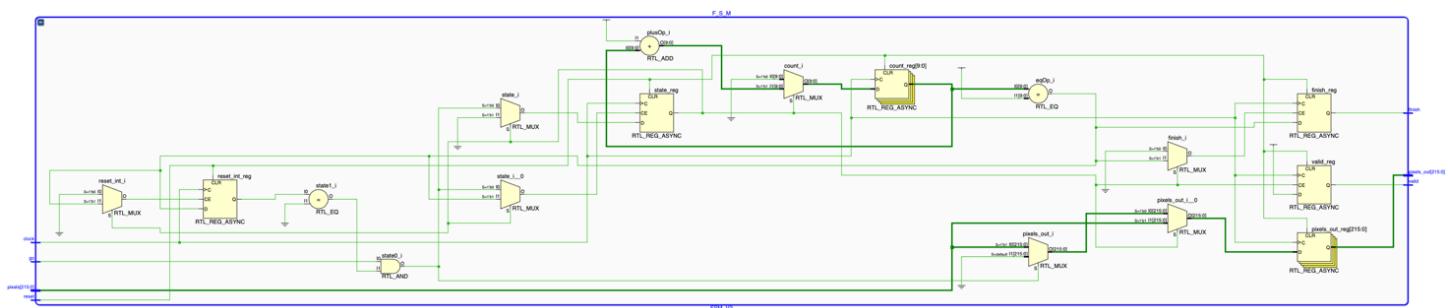
I segnali di “**valid**” e “**finish**” quando vengono generati dalla FSM vengono fatti ritardare con delle FIFO da **undici registri** per rispettare la latenza del circuito di filtraggio, in modo che in uscita siano sincronizzati rispettivamente col primo e ultimo pixel filtrato valido.

Per il conteggio dei colpi di clock sono stati utilizzati operatori aritmetici ad alto livello.

Sono presenti due stati:

- **Idle:** Stato di inizializzazione in cui gli ingressi e le uscite della FSM vengono impostati a zero. Il passaggio allo stato **Start** avviene quando il segnale "go" assume il livello logico alto. Questo segnale, generato ad inizio lettura del file contenente l'immagine, è ritardato tramite una FIFO in modo da abilitarsi esattamente quando il primo pixel raggiunge la posizione centrale del buffer.
 - **Start:** Stato in cui inizia il processo di filtraggio. Il segnale “**valid**” viene attivato e il contatore “**count**” inizia a incrementare ad ogni colpo di clock. Una volta che “**count**” raggiunge il valore **1023**, si è certi di aver letto tutti i dati. Il completamento dell'operazione viene indicato tramite il segnale “**finish**”, dopodiché lo stato ritorna a **Idle** e il segnale di “**reset_int**” viene alzato, azzerando tutti i pixels visti dall'hardware.

Di seguito viene riportato lo **schema RTL**:



6.2 FSM CON GESTIONE DEI BORDI

```

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity FSM3 is
    Port ( clk, rst, is_reading: in std_logic;
            stato: out std_logic_vector(3 downto 0);
            valid, finish: out std_logic );
end FSM3;

architecture Behavioral of FSM3 is

type state_type is (prestart,temp, start, nordovest, up, nordest, left, center, right, sudovest, down, sudest, ending, idle);
signal state : state_type;
signal count_o, count_v: std_logic_vector(5 downto 0);
signal real_go: std_logic;
begin

process (clk, rst)
begin

if (rst = '1') then
    count_o <= (others=>'0');
    count_v <= (others=>'0');
    stato <= (others=>'0');
    valid <= '0';
    finish <= '0';
    state <= prestart;
    real_go<='0';
elsif (rising_edge(clk)) then

    case state is

        when idle =>
            count_o <= (others => '0');
            real_go<='0';
            state<=idle;

        when prestart =>
            if(is_reading='1') then
                state <= temp;
                stato<="0000";
                real_go<='1';
                end if;
        when temp =>
            state<=start;
        when start =>
            if(real_go='1') then
                state <= nordovest;
                stato <= "0001";
                end if;
        when nordovest =>
            stato <= "0010";
            state <= up;
            valid<='1';

        when up =>
            count_o <= count_o + 1;
            if (count_o = "011101") then -- se count = 29;
                stato <= "0011";
                state <= nordest;
            end if;
        when nordest =>
            stato <= "0100";
            state <= left;
            count_o <= (others => '0');

        when left =>
            state <= center;
            stato<="0101";

        when center =>
            count_o <= count_o+1;
            if (count_o = "011101") then -- se count = 29;
                stato <= "0110";
                state <= right;
            end if;

        when right =>
            count_v <= count_v + 1;
            count_o <= (others => '0');
            if (count_v = "011101") then -- se count_v = 29;
                stato <= "0111";
                state <= sudovest;
            else stato <= "0100";
                state <= left;
            end if;

        when sudovest =>
            stato <= "1000";
            state <= down;
            count_v <= (others => '0');

        when down =>
            count_o <= count_o + 1;
            if (count_o = "011101") then -- se count = 29;
                stato <= "1001";
                state <= sudest;
            end if;

        when sudest =>
            state <= ending;
            stato<="1010";
            count_o <= (others => '0');

        when ending =>
            valid <= '0';
            finish <= '1';
            state <= idle;
            stato<="0000";

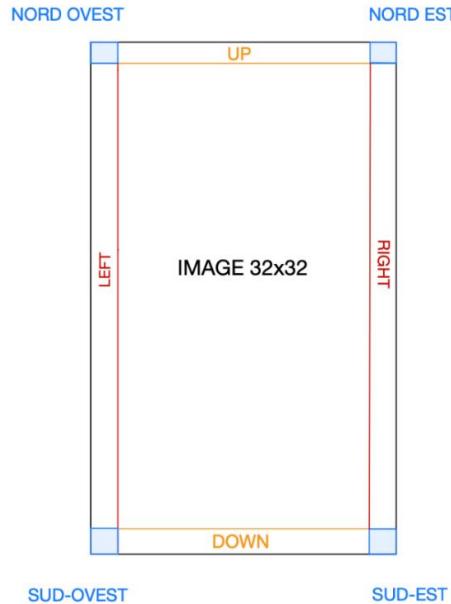
    end case;
end if;
end process;
end Behavioral;

```

L' entity di questa **FSM** si differenzia dalla precedente per il fatto che in ingresso non prende i pixel del buffer.

Il segnale di **trigger** è definito come "**is_reading**".

Se consideriamo un'immagine, a seconda di dove ci troviamo nel bordo, abbiamo otto casistiche diverse da gestire: quattro riguardano gli angoli dell'immagine (**NORD-OVEST**, **NORD-EST**, **SUD-OVEST**, **SUD-EST**) e quattro riguardano i lati dell'immagine (**LEFT**, **RIGHT**, **UP**, **DOWN**). In figura è riportato quanto detto:



La macchina a stati finiti deve comprendere quindi almeno otto stati specifici per la gestione dei bordi, uno stato “**center**” per tutti i pixel interni al bordo, uno stato “**idle**”, e uno stato “**ending**” che indica la fine del processo di filtraggio dell'immagine RGB.

Per determinare la posizione del pixel corrente nell'immagine in termini di righe e colonne, vengono utilizzati due segnali di conteggio:

- **count_o** per conteggiare il numero della colonna.
- **count_v** per conteggiare il numero della riga.

Attraverso il reset la **FSM** passa nello stato di “**prestart**” e vi permane fintanto che il segnale **is_reading** non diventa alto. Il segnale **is_reading** è inviato tramite delle FIFO due colpi di clock prima che il primo pixel valido si posizioni nella posizione centrale del buffer. Quando **is_reading** è alto si alza un segnale **real_go** e si passa allo stato di **temp**. Questo stato è uno stato satellite, e risolve un problema ricorrente della simulazione timing post-implementazione:

In sostanza il segnale di **is_reading** funge da trigger della **FSM** e nelle simulazioni post timing accadeva che tale segnale veniva catturato in ritardo andando a inficiare il corretto funzionamento del circuito. La **FSM** era sensibile al brusco passaggio dallo stato di “**start**” a quello di “**nordovest**”. Questo stato tampone **temp** risolve questa problematica garantendo il corretto funzionamento del circuito anche durante la post implementazione.

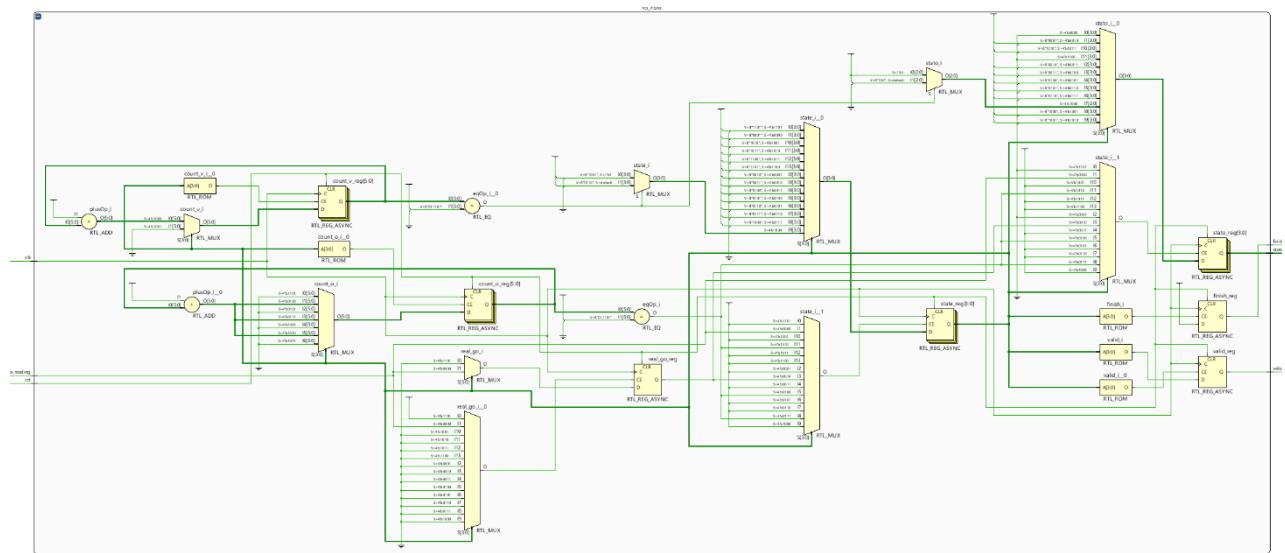
All'inizio, il circuito gestisce l'angolo in alto a sinistra dell'immagine. Il segnale **count_o** viene incrementato per ogni scorrimento fatto lungo le righe (esclusi tutti i pixel laterali dell'immagine) per tenere conto dell'indice di colonna e viene reimpostato a 0 ogni qual

volta si deve passare alla riga successiva e lo si fa esattamente prima, quindi nello stato **“nordest”**, **“right”** e anche alla fine nel caso **“sudest”**.

Il segnale **count_v** viene incrementato ogni volta passiamo in **“right”** per tenere conto dell'indice di riga.

Quando **count_v** raggiunge 29, si passa all'angolo in basso a sinistra dell'immagine, Una volta gestito l'ultimo pixel nello stato sudest, il circuito entra nello stato di **“ending”**, in cui si attiva il segnale **finish**, si resetta lo stato e si ritorna in **“idle”**.

Infine, è riportato lo schema RTL della **FSM** appena descritta.



BUFFER DECODE

Di base è un ampio multiplexer che riarrangia i pixel uscenti dal buffer principale in funzione di un selettori a 6 bit, dove i primi 2 bit rappresentano il comando (**cmd**) che indica il tipo di gestione dei bordi scelto, mentre i restanti 4 bit ci danno informazioni sullo stato della FSM, ovvero, sulla posizione del pixel da filtrare all'interno dell'immagine. Questo blocco è in costante dialogo con la FSM e ne permette l'effettiva gestione del bordo. Come detto in precedenza, è possibile selezionare tra quattro diverse modalità di gestione dei bordi tramite un segnale di comando asincrono denominato "**cmd**", come segue:

- **White Padding:** tutti i pixel esterni all'immagine confinanti con i bordi vengono virtualmente inizializzati a 255.
Si attiva se **cmd= "10"**.
- **Black Padding:** tutti i pixel esterni all'immagine confinanti con i bordi vengono virtualmente inizializzati a 0.
Si attiva se **cmd= "01"**.
- **Mirroring:** i pixel adiacenti ai bordi vengono specchiati.
Si attiva se **cmd= "11"**.
- **Toroidale mista:** Quando si applica un filtro su un'immagine con estensione toroidale, i pixel sul bordo destro dell'immagine vengono trattati come se fossero adiacenti ai pixel sul bordo sinistro, e viceversa; similmente avviene tra i bordi superiore e inferiore. Tuttavia, nel nostro caso, la gestione per il bordo superiore è effettuata tramite **padding di zeri** per via del reset iniziale, motivo per il quale parliamo di toroidale mista perché per i bordi superiore e inferiore la gestione è “intrinseca dell'Hardware” e si attiva se **cmd="00"**.

```

library IEEE;
use IEEE.STD_LOGIC_1164.all;

entity BUFFER_DECODE is
  Port ( clk, rst: in std_logic;
         cmd: in std_logic_vector(1 downto 0);
         state: in std_logic_vector(3 downto 0);
         pixels: in std_logic_vector(71 downto 0);
         pixels_out: out std_logic_vector(71 downto 0));
end BUFFER_DECODE;

architecture Behavioral of BUFFER_DECODE is

begin

  signal enc: std_logic_vector(5 downto 0);
  signal buf_gen,buf_p: std_logic_vector(71 downto 0);
  signal pix1,pix2,pix3,pix4,pix5,pix6,pix7,pix8,pix9,zero:white: std_logic_vector(7 downto 0);
  signal state_p: std_logic_vector(3 downto 0);

begin

  pix1 <= pixels(71 downto 64);
  pix2 <= pixels(63 downto 56);
  pix3 <= pixels(55 downto 48);
  pix4 <= pixels(47 downto 40);
  pix5 <= pixels(39 downto 32);
  pix6 <= pixels(31 downto 24);
  pix7<= pixels(23 downto 16);
  pix8<= pixels(15 downto 8);
  pix9<= pixels(7 downto 0);
  zero<=(others>>'0');
  white<=(others>>'1');

  --giro il buffer per semplicità visiva

  with cmd select
  buf_p<= pixels when "00",
  buf_gen when others;

  with enc select
  buf_gen<= --nord_ovest code 0001

  pix9&pix8&zero&pix6&pix5&pix4&pix3&pix2&pix1 when "010001", --black padding code 01
  pix9&pix8&white&pix6&pix5&white&white&white when "100001", --white padding code 10
  pix9&pix8&pix9&pix6&pix5&pix6&pix9&pix8&pix9 when "110001", --mirroring code 11

  --up code 0010
  pix9&pix8&pix7&pix6&pix5&pix4&zero&zero when "010010", --black padding
  pix9&pix8&pix7&pix6&pix5&pix4&white&white when "100010", --white padding
  pix9&pix8&pix7&pix6&pix5&pix4&pix9&pix8&pix7 when "110010", --mirroring

  --nord est code 0011
  zero&pix8&pix7&zero&pix5&pix4&pix3&pix2&pix1 when "010011", --black padding
  white&pix8&pix7&white&pix5&pix4&white&white when "100011", --white padding
  pix7&pix8&pix7&pix4&pix5&pix4&pix7&pix8&pix7 when "110011", --mirroring

  --left code 0100
  pix9&pix8&zero&pix6&pix5&zero&pix3&pix2&zero when "010100", --black padding
  pix9&pix8&white&pix6&pix5&white&pix3&pix2&white when "100100", --white padding
  pix9&pix8&pix9&pix6&pix5&pix6&pix6&pix3 when "110100", --mirroring

  (others>>'0') when "010000",
  (others>>'0') when "100000",
  (others>>'0') when "110000",

  --right code 0110
  zero&pix8&pix7&zero&pix5&pix4&zero&pix2&pix1 when "010110",
  white&pix8&pix7&white&pix5&pix4&white&pix2&pix1 when "100110",
  pix7&pix8&pix7&pix4&pix5&pix4&pix1&pix2&pix1 when "110110",

  --sud_ovest code 0111
  zero&zero&zero&pix6&pix5&zero&pix3&pix2&zero when "010111",
  white&white&white&pix6&pix5&white&pix3&pix2&white when "100111",
  pix3&pix2&pix3&pix4&pix5&pix6&pix3&pix2&pix3 when "110111",

  -- down code 1000
  zero&zero&zero&pix6&pix5&pix4&pix3&pix2&pix1 when "011000",
  white&white&white&pix6&pix5&pix4&pix3&pix2&pix1 when "101000",
  pix3&pix2&pix1&pix6&pix5&pix4&pix3&pix2&pix1 when "111000",

  --sud est code 1001
  zero&zero&zero&pix6&pix5&pix4&zero&pix2&pix1 when "011001",
  white&white&white&pix5&pix4&white&pix2&pix1 when "101001",
  pix4&pix2&pix1&pix4&pix5&pix4&pix1&pix2&pix1 when "111001",

  pixels when others;

process(clk,rst)
begin
  if rst = '1' then
    pixels_out<=(others>>'0');
    state_p<="0000";
  elsif rising_edge(clk) then
    pixels_out<=buf_p;
    state_p<=state;
  end if;
end process;

end Behavioral;

```

Nel **DECODE_BUFFER**, per semplicità di scrittura viene innanzitutto eseguita una rotazione del buffer.

Il buffer viene configurato, come detto in precedenza, a seconda dello stato della **FSM** e della gestione scelta. Abbiamo quindi 3x9 configurazioni del buffer.

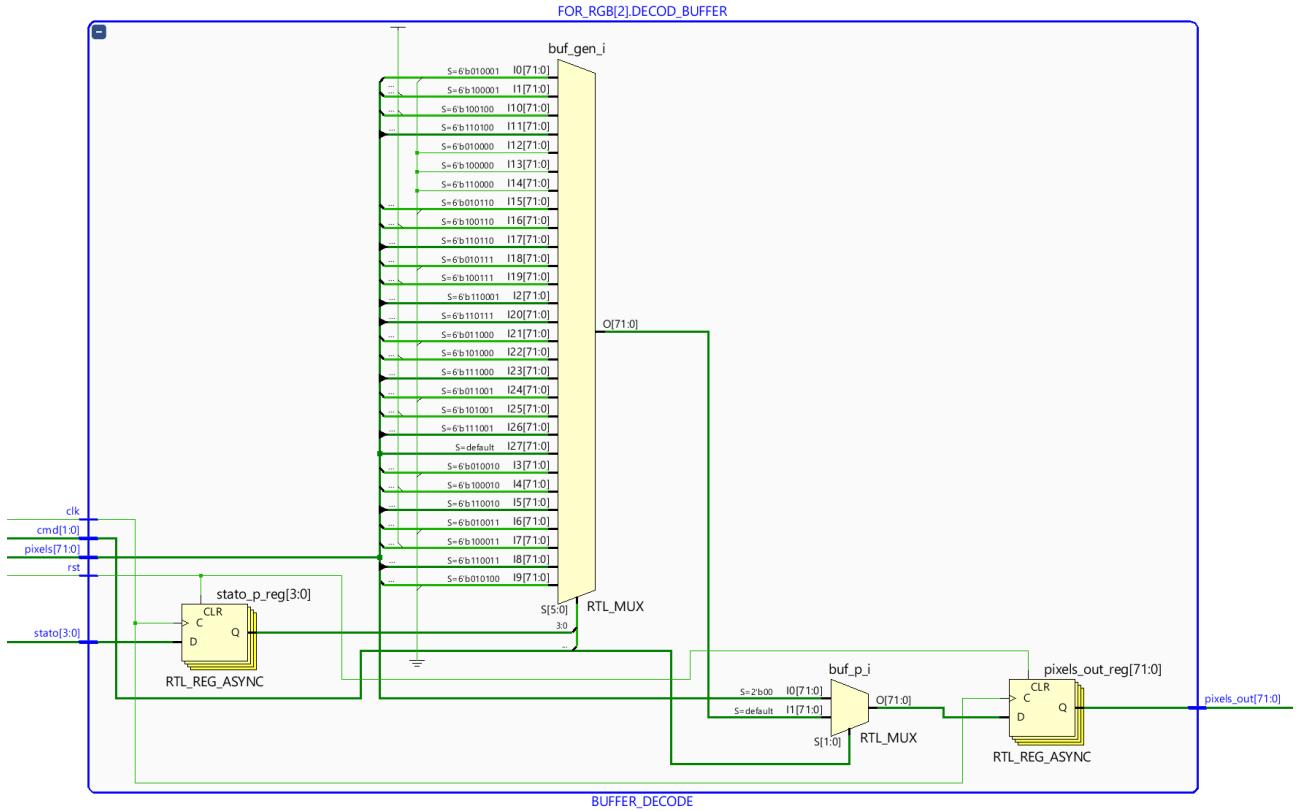
Per il white padding si riempiono con "11111111" (255) i pixel della finestra esterni all'immagine; Al contrario, nel caso di black padding, si riempiono con "00000000" (0).

Nel caso del Mirroring, a seconda della posizione, il buffer viene specchiato opportunamente.

Naturalmente per ogni gestione i pixel interni al bordo (centrali) rimangono invariati.

È stato possibile scrivere il tutto sfruttando la logica di un Mux e considerando il buffer un vettore di 72 bit (9 pixel).

Di seguito è riportato l'RTL del **DECOD_BUFFER**.



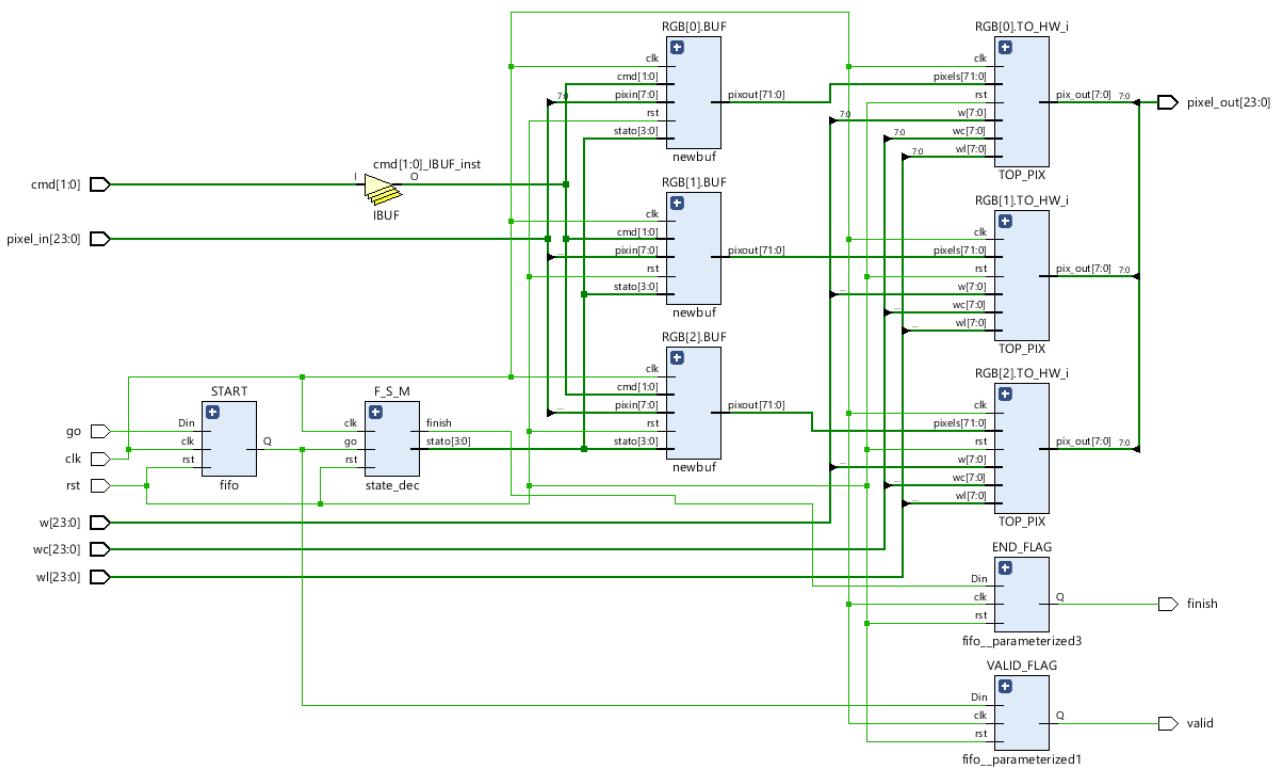
6.3 FSM PARAMETRICA CON GESTIONE DEI BORDI

Questa terza tipologia di **FSM** mantiene invariate le caratteristiche funzionali della precedente versione, ma possiede un'architettura di decodificazione del buffer che consente di implementare più stadi di pipeline, al fine di migliorare le performance in termini di massima frequenza di clock.

Inoltre, permette di selezionare dal modulo **Top** del circuito la dimensione dell'immagine da filtrare, attraverso l'inserimento opportuno di variabili di tipo “*generic*”.

Tali variabili consentono di configurare l'hardware della scheda in funzione della dimensione inserita, prima della fase di implementazione.

Di seguito, è mostrato lo schema “**gate level**” del circuito.



6.3.1 State Decode

```

library IEEE;
use IEEE.STD.LOGIC_1164.ALL;
use IEEE.STD.LOGIC_UNSIGNED.ALL;
use IEEE.STD.LOGIC_UNSIGNED.all;
use IEEE.NUMERIC_STD.ALL;

-- Uncomment the following library declaration if using
-- arithmetic functions with Signed or Unsigned values
--use IEEE.NUMERIC_STD.ALL;

-- Uncomment the following library declaration if instantiating
-- any Xilinx leaf cells in this code.
--library UNISIM;
--use UNISIM.VComponents.all;

entity state_dec is
    generic( ddim: integer;
             bdim: integer );
    Port ( clk, rst: in std_logic;
           go: in std_logic;
           stat0: out std_logic_vector(3 downto 0);
           finish: out std_logic );
end state_dec;

architecture Behavioral of state_dec is

type state_type is (start, nordovest, up, nordest, left, center, right, sudovest, down, sudest, idle);
signal state : state_type;
signal count_o, count_v: std_logic_vector(bdim-1 downto 0);

begin
process (clk, rst)
begin
    if (rst = '1') then
        count_o <= (others=>'0');
        count_v <= (others=>'0');
        state <= (others=>'0');
        finish <= '0';
        state <= start;
    elsif (rising_edge(clk)) then
        case state is
            when idle =>      state <= idle;
                                state <= "1111";
                                finish <= '1';

            when start =>    if (go = '1') then      -- se pixel è cambiato;
                                state <= nordovest;
                            end if;
                                state <= "1111";
                                finish <= '0';

            when nordovest => state <= "0000";
                                finish <= '0';
                                state <= up;

            when up =>       state <= "0001";
                                finish <= '0';
                                count_o <= count_o + 1;
                                if (count_o = conv_std_logic_vector(ddim-3, bdim)) then -- se count = 29;
                                    count_o <= (others => '0');
                                    state <= nordest;
                                end if;

            when nordest => state <= "0010";
                                finish <= '0';
                                state <= left;

            when left =>     state <= "0011";
                                finish <= '0';
                                state <= center;

            when center =>   state <= "1000";
                                finish <= '0';
                                count_o <= count_o+1;
                                if (count_o = conv_std_logic_vector(ddim-3, bdim)) then -- se count = 29;
                                    count_o <= (others => '0');
                                    state <= right;
                                end if;

            when right =>    state <= "0100";
                                finish <= '0';
                                count_v <= count_v + 1;
                                if (count_v = conv_std_logic_vector(ddim-3, bdim)) then -- se count_v = 29;
                                    count_v <= (others => '0');
                                    state <= sudovest;
                                else state <= left;
                                end if;

            when sudovest => state <= "0101";
                                finish <= '0';
                                state <= down;

            when down =>      state <= "0110";
                                finish <= '0';
                                count_o <= count_o + 1;
                                if (count_o = conv_std_logic_vector(ddim-3, bdim)) then -- se count = 29;
                                    count_o <= (others => '0');
                                    state <= sudest;
                                end if;

            when sudest =>    state <= "0111";
                                finish <= '1';
                                state <= idle;

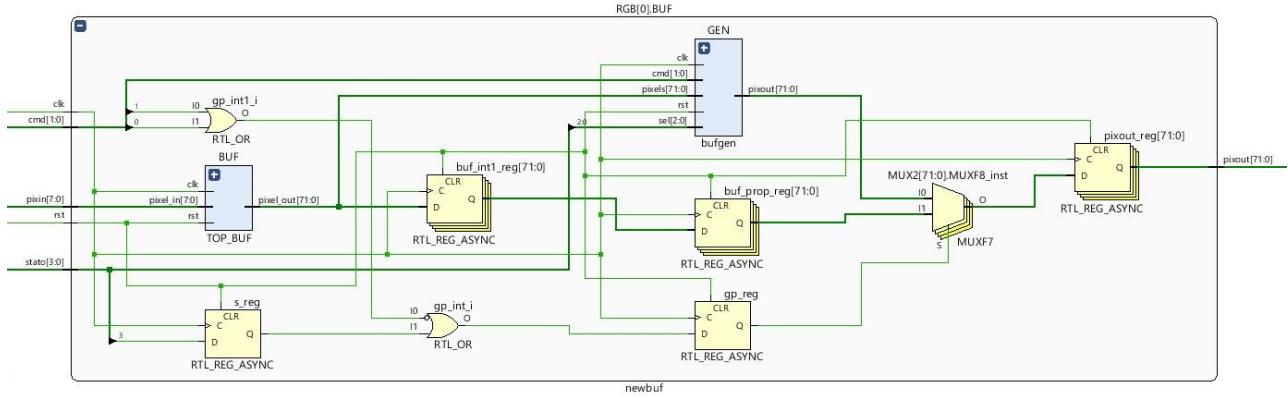
        end case;
    end if;
end process;
end Behavioral;

```

Il nucleo della FSM è definito dal blocco **state_dec**, analogamente alle altre versioni: dallo stato iniziale di “start”, un segnale di ‘**inizio lettura**’ permette di sincronizzare la successione dei 9 stati della FSM con le 9 suddivisioni dell’immagine da filtrare. La peculiarità è rappresentata dalla definizione di due variabili:

- **ddim**: esprime il numero di righe e colonne dell’immagine e costituisce un riferimento per il massimo numero raggiungibile dai contatori di riga e colonna, rispettivamente **count_v** e **count_o**;
- **bdim**: esprime il numero di bit da dedicare a ciascun contatore (pari a $\log_2 ddim$ approssimato per eccesso).

6.3.2 Output Buffer



L'architettura del blocco **newbuf** fornisce in uscita la **generazione** o la **propagazione** del buffer, selezionata opportunamente da un MUX a due ingressi tramite il selettoore **gp**.

Tale segnale è l'unione (tramite la funzione logica OR) di due informazioni fondamentali:

- la scelta della gestione dei bordi **toroidale mista** ($cmd = "00"$), identificabile dalla NOR dei bit del segnale *cmd*;
- la posizione del pixel all'interno della **parte centrale dell'immagine** ($stato = "1111"$), identificabile dal **MSB** del segnale *stato*.

Tramite questa struttura, il circuito è in grado di distinguere quando il buffer deve essere **generato** (in quanto il pixel da filtrare si trova nella cornice, con una gestione dei bordi tra *padding di zeri*, *padding di 255* e *mirroring*) oppure **propagato** (poiché il pixel da filtrare si trova al centro dell'immagine o la gestione selezionata è la *toroidale mista*).

6.3.3 Input Buffer

```

entity TOP_BUF is
generic (d: integer);
Port (pixel_in: in std_logic_vector(7 downto 0);
clk,rst: in std_logic;
pixel_out: out std_logic_vector(71 downto 0)
);
end TOP_BUF;

architecture Behavioral of TOP_BUF is

```

```

signal Qinternal: std_logic_vector(((d-3)*2+9)*8-1 downto 0);
signal pix_p: std_logic_vector(7 downto 0);
signal pixel_int: std_logic_vector(71 downto 0);

component REG8 is
Port (Din: in std_logic_vector(7 downto 0);
clk, rst: in std_logic;
Q: out std_logic_vector (7 downto 0)
);
end component;

```

viene stabilita la lunghezza dei registri FIFO; questi, infatti, hanno il compito di memorizzare temporaneamente le righe soggette all'operazione di filtraggio.

```

pix_p<=pixel_in;
pixel_int(7 downto 0)<=Qinternal(7 downto 0);
pixel_int(15 downto 8)<=Qinternal(15 downto 8);
pixel_int(23 downto 16)<=Qinternal(23 downto 16);
pixel_int(31 downto 24)<=Qinternal((d+1)*8-1 downto d*8);
pixel_int(39 downto 32)<=Qinternal((d+2)*8-1 downto (d+1)*8);
pixel_int(47 downto 40)<=Qinternal((d+3)*8-1 downto (d+2)*8);
pixel_int(55 downto 48)<=Qinternal((2*d+1)*8-1 downto 2*d*8);
pixel_int(63 downto 56)<=Qinternal((2*d+2)*8-1 downto (2*d+1)*8);
pixel_int(71 downto 64)<=Qinternal((2*d+3)*8-1 downto (2*d+2)*8);
pixel_out <= pixel_int;

REG8_0: REG8 port map(pix_p,clk,rst,Qinternal(7 downto 0));
FIFO: for i in 1 to 2*(d-3)+8 generate
REG8_i: REG8 port map (Qinternal(8*i-1 downto 8*(i-1)),clk,rst,Qinternal(8*(i+1)-1 downto 8*i));
end generate;

| end Behavioral;

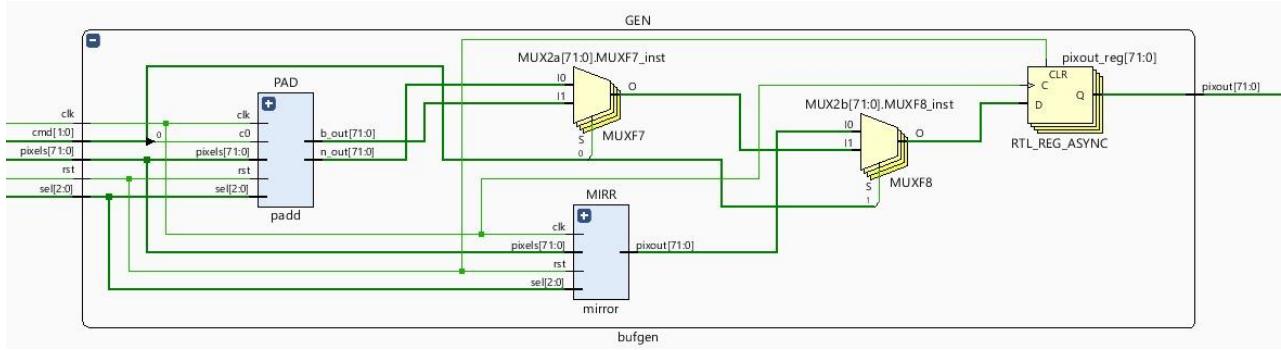
```

Le uscite di questo blocco costituiscono i pixels della finestra 3x3 da filtrare, uniti in un unico vettore da 72 bit. Il segnale interno è parametrizzato per essere correttamente ripartito nei segnali di output.

Il buffer di ingresso presenta una struttura congruente alle versioni già descritte, ma è risultato più opportuno includerlo all'interno del blocco *newbuf*.

La dimensione del buffer è configurabile tramite la variabile **generic** "d", in base alla quale

6.3.4 Buffer Generator



```

entity bufgen is
  Port (clk, rst: in std_logic;
        cmd: in std_logic_vector(1 downto 0);
        sel: in std_logic_vector(2 downto 0);
        pixels: in std_logic_vector(71 downto 0);
        pixout: out std_logic_vector(71 downto 0) );
end bufgen;

architecture Behavioral of bufgen is

component padd is
  Port (clk, rst, c0: in std_logic;
        sel: in std_logic_vector(2 downto 0);
        pixels: in std_logic_vector(71 downto 0);
        b_out, n_out: out std_logic_vector(71 downto 0) );
end component;

component mirror is
  Port (clk, rst: in std_logic;
        sel: in std_logic_vector(2 downto 0);
        pixels: in std_logic_vector(71 downto 0);
        pixout: out std_logic_vector(71 downto 0) );
end component;

signal mirr_int, black_int, white_int, padd_int, pix_int: std_logic_vector(71 downto 0);

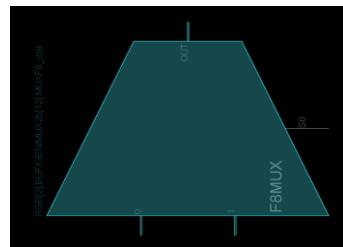
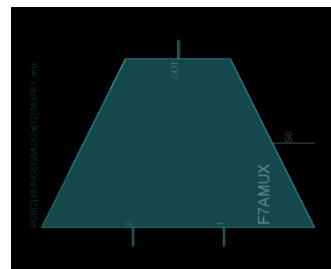
PAD: padd port map(clk, rst, cmd(0), sel, pixels, white_int, black_int);
MIRR: mirror port map(clk, rst, sel, pixels, mirr_int);

MUX2a: for i in 0 to 71 generate
  MUXF7_inst : MUXF7
  port map (
    O => padd_int(i),
    I0 => black_int(i),
    I1 => white_int(i),
    S => cmd(0)
  );
end generate;

MUX2b: for i in 0 to 71 generate
  MUXF8_inst : MUXF8
  port map (
    O => pix_int(i),
    I0 => mirr_int(i),
    I1 => padd_int(i),
    S => cmd(1)
  );
end generate;

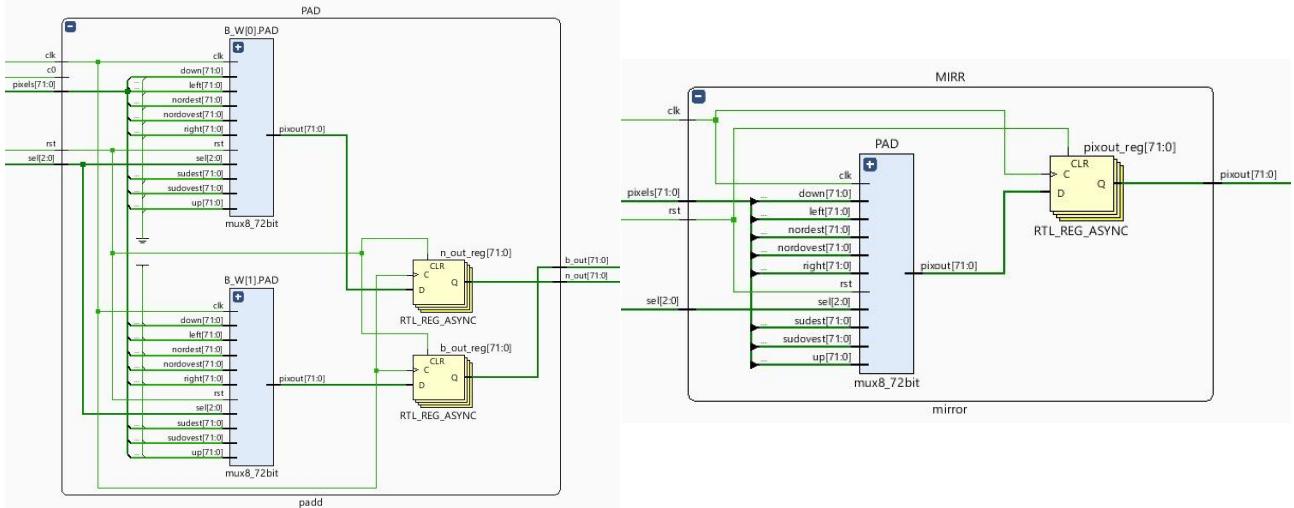
process(clk, rst)
begin
  if(rst = '1') then
    pixout <= (others=>'0');
  elsif(rising_edge(clk)) then
    pixout <= pix_int;
  end if;
end process;

```



La componente **bufgen** è formata da una struttura seriale di due multiplexers a due ingressi, terminato da uno stadio di pipeline. I selettori del primo e del secondo mux sono rispettivamente il **LSB** e il **MSB**. Nel presente blocco sono istanziate le primitive **MUXF7** e **MUXF8**, come analizzeremo più approfonditamente in seguito.

6.3.5 Padding di 0, 255 e Mirroring



La struttura circuitale della gestione dei bordi utilizza il componente **mux8_72bit** al fine di selezionare il vettore di buffer adeguato allo stato della FSM (che costituisce il selettore del mux), vale a dire alla posizione del pixel da filtrare.

Per il **white padding** e il **black padding**, descritti nello stesso blocco, vengono definiti dei vettori a 8 bit che custodiscono il valore del padding corrispondente (rispettivamente 0 e 255).

```

entity padd is
  Port (clk, rst, c0: in std_logic;
        sel: in std_logic_vector(2 downto 0);
        pixels: in std_logic_vector(71 downto 0);
        b_out, n_out: out std_logic_vector(71 downto 0) );
end padd;

pix1 <= pixels(71 downto 64);
pix2 <= pixels(63 downto 56);
pix3 <= pixels(55 downto 48);
pix4 <= pixels(47 downto 40);
pix5 <= pixels(39 downto 32);
pix6 <= pixels(31 downto 24);
pix7 <= pixels(23 downto 16);
pix8 <= pixels(15 downto 8);
pix9 <= pixels(7 downto 0);
n <= "00000000";
b <= "11111111";

state1 <= b & b & b & b & pix5 & pix6 & pix7 & pix8 & pix9 &
           n & n & n & n & pix5 & pix6 & pix7 & pix8 & pix9;
state2 <= b & b & b & pix4 & pix5 & pix6 & pix7 & pix8 & pix9 &
           n & n & n & pix4 & pix5 & pix6 & pix7 & pix8 & pix9;
state3 <= b & b & b & pix4 & pix5 & b & pix7 & pix8 & b &
           n & n & n & pix4 & pix5 & n & pix7 & pix8 & n;
state4 <= b & pix2 & pix3 & b & pix5 & pix6 & b & pix8 & pix9 &
           n & pix2 & pix3 & n & pix5 & pix6 & n & pix8 & pix9;
state5 <= pix1 & pix2 & b & pix4 & pix5 & b & pix7 & pix8 & b &
           pix1 & pix2 & n & pix4 & pix5 & n & pix7 & pix8 & n;
state6 <= b & pix2 & pix3 & b & pix5 & pix6 & b & b & b &
           n & pix2 & pix3 & n & pix5 & pix6 & n & n & n;
state7 <= pix1 & pix2 & pix3 & pix4 & pix5 & pix6 & b & b & b &
           pix1 & pix2 & pix3 & pix4 & pix5 & pix6 & n & n & n;
state8 <= pix1 & pix2 & b & pix4 & pix5 & pix6 & b & b & b &
           pix1 & pix2 & n & pix4 & pix5 & pix6 & n & n & n;
```

Per il *mirroring*, gli ingressi del **mux**, corrispondenti agli **otto stati** della **FSM** relativi ai bordi dell'immagine, sono creati da una opportuna operazione di “**mapping**” dei pixels appartenenti alla finestra di filtraggio. Tale operazione è effettuata **specchiando** fuori dai bordi i pixels all'interno della **cornice** dell'immagine.

È importante notare che, poiché la distinzione tra cornice e centro dell'immagine è gestita dall'architettura generale *newbuf*, il seletore dei mux presentati è il vettore dei **tre bit meno significativi dello stato**.

```

entity mirror  is
  Port (clk, rst:  in std_logic;
        sel:  in std_logic_vector(2 downto 0);
        pixels:  in std_logic_vector(71 downto 0);
        pixout:  out std_logic_vector(71 downto 0) );
end mirror;

architecture Behavioral of mirror is

component mux8_72bit is
  Port (clk, rst:  in std_logic;
        sel:  in std_logic_vector(2 downto 0);
        nordovest, up, nordest, left, right, sudovest, down, sudest:  in std_logic_vector(71 downto 0);
        pixout:  out std_logic_vector(71 downto 0) );
end component;

pix1 <= pixels(71 downto 64);
pix2 <= pixels(63 downto 56);
pix3 <= pixels(55 downto 48);
pix4 <= pixels(47 downto 40);
pix5 <= pixels(39 downto 32);
pix6 <= pixels(31 downto 24);
pix7 <= pixels(23 downto 16);
pix8 <= pixels(15 downto 8);
pix9 <= pixels(7 downto 0);

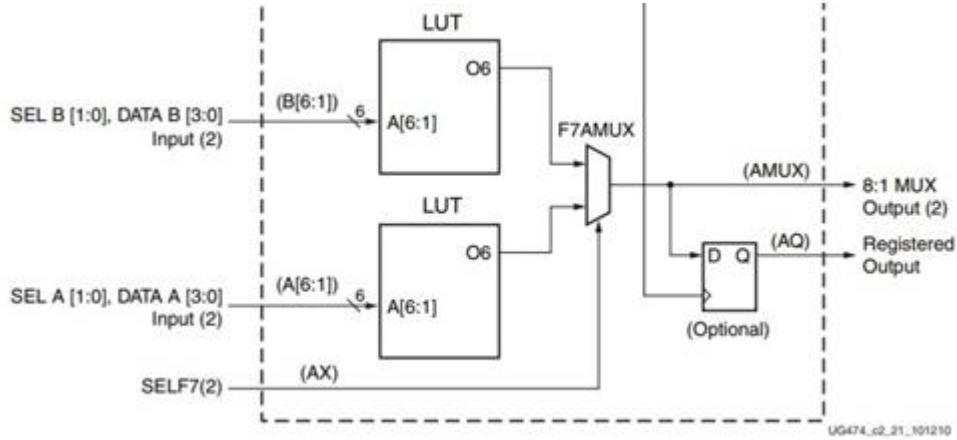
state1 <= pix9 & pix8 & pix9 & pix6 & pix5 & pix6 & pix9 & pix8 & pix9;
state2 <= pix7 & pix8 & pix9 & pix4 & pix5 & pix6 & pix7 & pix8 & pix9;
state3 <= pix7 & pix8 & pix7 & pix4 & pix5 & pix4 & pix7 & pix8 & pix7;
state4 <= pix3 & pix2 & pix3 & pix6 & pix5 & pix6 & pix9 & pix8 & pix9;
state5 <= pix1 & pix2 & pix1 & pix4 & pix5 & pix4 & pix7 & pix8 & pix7;
state6 <= pix3 & pix2 & pix3 & pix6 & pix5 & pix6 & pix3 & pix2 & pix3;
state7 <= pix1 & pix2 & pix3 & pix4 & pix5 & pix6 & pix1 & pix2 & pix3;
state8 <= pix1 & pix2 & pix1 & pix4 & pix5 & pix4 & pix1 & pix2 & pix1;

PAD: mux8_72bit port map(clk, rst, sel,
                           state1,
                           state2,
                           state3,
                           state4,
                           state5,
                           state6,
                           state7,
                           state8,
                           pixint);

process(clk, rst)
begin
  if(rst = '1') then
    pixout <= (others=>'0');
  elsif(rising_edge(clk)) then
    sel_in <= sel;
    pixout <= pixint;
  end if;
end process;

end Behavioral;
```

6.3.6 Multiplexer 8:1



Il seguente componente è stato descritto con l'obiettivo di essere implementato in un'unica SLICE della scheda FPGA, utilizzando le risorse dedicate già presenti.

```

entity mux8_72bit is
  generic(n: integer := 72);
  Port (clk, rst: in std_logic;
        sel: in std_logic_vector(2 downto 0);
        nordovest, up, nordest, left, right, sudovest, down, sudest: in std_logic_vector(n-1 downto 0);
        pixout: out std_logic_vector(n-1 downto 0) );
end mux8_72bit;

begin
  MUX8: for i in 0 to n-1 generate
    LUT6_0: LUT6
      generic map (
        INIT => "1111111000000000111100001111000011001100110011001010101010101010"
      )
      port map (
        I0 => nordovest(i),
        I1 => up(i),
        I2 => nordest(i),
        I3 => left(i),
        I4 => sel(0),
        I5 => sel(1),
        O  => state_int0(i)
      );
    LUT6_1: LUT6
      generic map (
        INIT => "111111100000000011110000111100001100110011001100101010101010"
      )
      port map (
        I0 => right(i),
        I1 => sudovest(i),
        I2 => down(i),
        I3 => sudest(i),
        I4 => sel(0),
        I5 => sel(1),
        O  => state_int1(i)
      );
  end;

```

In questo design, la primitiva **LUT6** viene istanziata e adoperata come **multiplexer a quattro ingressi**, inizializzando la tabella di verità contenuta nelle sue $2^6 = 64$ celle di memoria.

Prima di descrivere come mappare la logica del mux 4:1 in una *look up table*, è importante definire la gestione degli ingressi della nostra **LUT6**.

Inputs						Outputs
I5	I4	I3	I2	I1	I0	O
0	0	0	0	0	0	INIT[0]
0	0	0	0	0	1	INIT[1]
0	0	0	0	1	0	INIT[2]
0	0	0	0	1	1	INIT[3]
0	0	0	1	0	0	INIT[4]
0	0	0	1	0	1	INIT[5]
0	0	0	1	1	0	INIT[6]
0	0	0	1	1	1	INIT[7]
0	0	1	0	0	0	INIT[8]
0	0	1	0	0	1	INIT[9]

I due MSB's I5 e I4 rappresentano il **selettore** del multiplexer 4:1, mentre i restanti **4 bits** costituiscono gli **ingressi**.

```

MUXF7_inst : MUXF7
  port map (
    O => pixout(i),      -- Output of MUX to general routing
    I0 => state_int0(i), -- Input (tie to LUT6 O6 pin)
    I1 => state_int1(i), -- Input (tie to LUT6 O6 pin)
    S => sel(2)          -- Input select to MUX
  );
end generate;

end Behavioral;

```

La **stringa di inizializzazione** "INIT" deve seguire l'ordine mostrato in figura; perciò, la prima coppia di byte (16 bit) più a destra della stringa corrisponde all'output della **LUT** quando il selettore è "00"; similmente si procede per le successive coppie. In questo modo, ciascuna delle due **LUT6** instanziate costituisce un mux 4:1; pertanto, collegando in serie ad esse un multiplexer 2:1 come il **MUXF7** già presente nella *slice*, si ottiene un mux 8:1 ad un singolo bit di uscita. Tramite un *for generate* è possibile parallelizzare tale struttura per creare un mux 8:1 di dimensione 72 bit.

7. CONVERTITORE RGB2GRAY

Il seguente blocco effettua la conversione dell'immagine da RGB a grayscale.

```
entity RGB_TO_GRAY is
generic ( DIM_FIFO_EXTRA : integer := 3;
           DELAY_PIPE: integer := 3);
port(clk,rst: in std_logic;
pix_in: in std_logic_vector(23 downto 0);
pix_grey: out std_logic_vector(7 downto 0));
end RGB_TO_GRAY;
```

Il blocco riceve in ingresso una terna di pixel R, G, B e fornisce in uscita un singolo pixel, risultato della conversione da RGB a GRAY secondo la regola classica.

In generale la conversione si attua tramite la seguente formula:

$$Gray_{pixel} = 0,299 \cdot Pix_R + 0,587 \cdot Pix_G + 0,114 \cdot Pix_B$$

L'approccio sviluppato evita l'utilizzo di divisorî utilizzando una piccola approssimazione:

- $0,299 \approx \frac{1}{4} + \frac{1}{32} + \frac{1}{64} = 0,296875$ (sottostima dei rossi dello 0,2125%)
- $0,587 \approx \frac{1}{2} + \frac{1}{16} + \frac{1}{64} + \frac{1}{128} = 0,5859375$ (sottostima dei verdi dello 0,10625%)
- $0,114 \approx \frac{1}{16} + \frac{1}{32} + \frac{1}{64} + \frac{1}{128} = 0,1171875$ (sovraffima dei blu dello 0,31875% errore massimo)

In questo modo, è possibile effettuare la conversione da RGB a GRAY utilizzando solo operazioni di right-shift e somme cioè:

1. Si effettua l'operazione $\frac{pixel}{4} + \frac{pixel}{32} + \frac{pixel}{64} = S1$;
2. Si effettua l'operazione $\frac{pixel}{2} + \frac{pixel}{16} + \frac{pixel}{64} + \frac{pixel}{128} = S2$;
3. Si effettua l'operazione $\frac{pixel}{16} + \frac{pixel}{32} + \frac{pixel}{64} + \frac{pixel}{128} = S3$;
4. Si effettua l'operazione $Gray_{pixel} = S1 + S2 + S3$;

Eseguire questa approssimazione permette di avere un errore di calcolo su ogni pixel inferiore ad 1 (es: $255 * 0,31875\% = 0,81 < 1$). Poichè l'errore non tende esattamente a zero, l'operazione di RGB_TO_GRAY differisce in minima parte dalla conversione tramite tool Matlab. Nel dettaglio, trattandosi in uscita di numeri interi, l'eventuale oscillazione dovuta all'errore intrinseco di questa strategia intorno allo 0.5 farà sì che ci siano dei numeri approssimati per eccesso anziché per difetto e viceversa, per cui è previsto una precisione al netto di ± 1 su 256 possibili valori. Si noti che tutto ciò è normalizzato alla massima dimensione del pixel, infatti:

$$0,299 + 0,587 + 0,114 = 1$$

Ma anche la somma dei coefficienti approssimati adottati:

$$\frac{1}{4} + \frac{1}{32} + \frac{1}{64} + \frac{1}{2} + \frac{1}{16} + \frac{1}{64} + \frac{1}{128} + \frac{1}{16} + \frac{1}{32} + \frac{1}{64} + \frac{1}{128} = 1$$

Questo garantisce che qualunque sia la terna dei pixel in ingresso, la loro somma non sarà mai maggiore del pixel più intenso. Da notare che, nel caso in cui la terna sia (255,255,255), il risultato sarà proprio 255; questo ci permette di usare i carry save a 8 bit descritti in precedenza senza doverne creare di nuovi, in quanto fino all'ultima somma il risultato non supererà mai gli 8 bit.

7.1 RGB2GRAY: GESTIONE DEL RESTO

Le divisioni intere per potenze del due possono essere effettuate eseguendo dei right shift. In generale è vero che $\frac{k}{2^n}$, con k e 2^n in binario, è uguale a $right\ shift(n)$ di k, banalmente si tagliano le prime n cifre.

Un esempio: $\frac{10001}{10000} = right\ shift(4) \ di \ 10001 = 1$, Resto = 0001; $(\frac{17}{16} \approx 1)$. In questo caso l'operazione di right shift è sufficiente a garantire il corretto risultato, ma tale operazione da sola introduce degli errori, infatti:

$$\frac{01111}{10000} = right\ shift(4) \ di \ 01111 = 0, Resto = 1111; \frac{15}{16} \approx 1.$$

N.B: Con “right shift(4)” si intende testualmente “shiftare di quattro posizioni verso destra”

In questo caso l'operazione di right shift fornisce come risultato uno zero quando in realtà il risultato della divisione intera tra 15 e 16 è più prossimo a uno che a zero. Ne consegue che deve esistere una logica di gestione del resto, al fine di garantire una adeguata precisione. Il resto dell'operazione di right shift è tutta la parte di vettore che metto da parte, la quale è tanto maggiore quanto più grande è il divisore. L'idea è quella di immaginare un vettore da sette elementi attaccato al dividendo: tutto ciò che in teoria andrebbe messo da parte col right shift viene inserito a partire dall'**MSB** di questo “vettore dei resti”. Un esempio rende molto bene l'idea:

```

op1_R<='0'&'0'&pix_inp(7 downto 2);--pix/4
op2_R<='0'&'0'&'0'&'0'&pix_inp(7 downto 5);--pix/32
op3_R<='0'&'0'&'0'&'0'&'0'&pix_inp(7 downto 6);--pix/64
zero<=(others=>'0');
restoR4<='0'&pix_inp(1 downto 0)&"00000";
restoR32<='0'&pix_inp(4 downto 0)&"00";
restoR64<='0'&pix_inp(5 downto 0)&'0';

--VERDI
op1_G<='0'&pix_inp(15 downto 9); --/2
op2_G<='0'&'0'&'0'&'0'&pix_inp(15 downto 12);--/16
op3_G<='0'&'0'&'0'&'0'&'0'&pix_inp(15 downto 14);--/64
op4_G<='0'&'0'&'0'&'0'&'0'&'0'&pix_inp(15);--/128
restoG2<='0'&pix_inp(8)&"000000";
restoG16<='0'&pix_inp(11 downto 8)&"000";
restoG64<='0'&pix_inp(13 downto 8)&'0';
restoG128<='0'&pix_inp(14 downto 8);

--BLU
op1_B<='0'&'0'&'0'&'0'&pix_inp(23 downto 20);--/16
op2_B<='0'&'0'&'0'&'0'&'0'&pix_inp(23 downto 21);--/32
op3_B<='0'&'0'&'0'&'0'&'0'&pix_inp(23 downto 22);--/64
op4_B<='0'&'0'&'0'&'0'&'0'&'0'&pix_inp(23);--/128
restoB16<='0'&pix_inp(19 downto 16)&"000";
restoB32<='0'&pix_inp(20 downto 16)&"00";
restoB64<='0'&pix_inp(21 downto 16)&'0';
restoB128<='0'&pix_inp(22 downto 16);

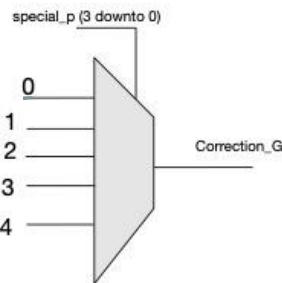
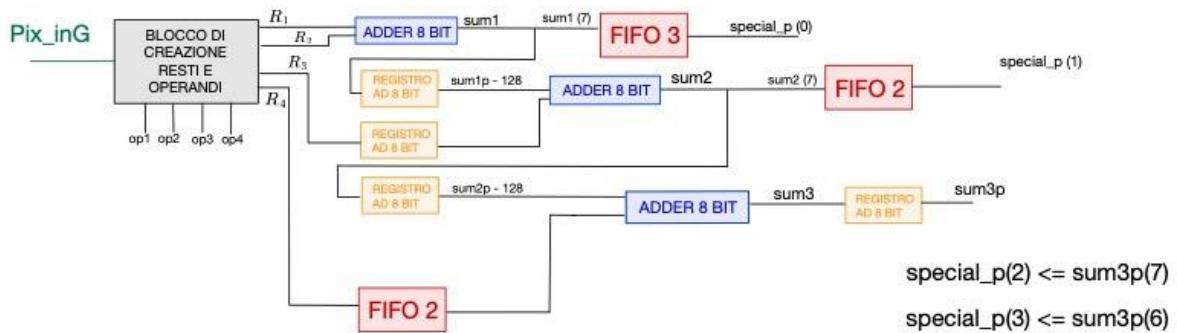
```

I segnali op1_R e simili contengono il **right shift** opportuno del pixel, mentre i segnali restoR4 e simili contengono appunto i resti. La dimensione dei segnali di resto è di 8 bit

poiché sono pensate per andare in ingresso a degli ADDER ad 8 bit: l'ottavo bit è messo a zero per effettuarne un'estensione al fine di utilizzarli negli ADDER.

Di fatto quello che stiamo facendo è di scomporre la scala da 0.00 a 1.00 non a passi di $\frac{1}{100}$ ma a passi di $\frac{1}{128}$. Sotto questo profilo lo 0.5 sarà rappresentato da 64, in binario 01000000, e l'unità da 128, in binario 10000000. A questo punto esiste una metrica che ci permette di stabilire quanto sia il resto e quindi è possibile sviluppare una logica di correzione.

Questa logica si sviluppa a partire da un semplice fatto: ogni *right shift* mi fornisce il risultato esatto, al più differisce di uno per difetto, ne consegue che una logica di correzione deve essere in grado di stabilire, dati n operandi, in funzione dei vari resti, se aggiungere al risultato ottenuto dalle somme dei *right shift* un numero che va da 0 a n . Di seguito è illustrato lo schema della logica utilizzata:



In questo caso facciamo riferimento alla gestione del resto dei verdi e blu, in quanto a differenza dei rossi hanno quattro operandi (quindi quattro resti), per cui la struttura è leggermente più estesa. La logica per i rossi è semplicemente quella appena illustrata in figura ma scalata a tre ingressi. Il funzionamento è spiegato in questi passi:

1. Si sommano i primi due resti, la loro somma non sarà mai superiore ad 8 bit in quanto l'ottavo bit è messo a zero e quindi è come se si stessero sommando due numeri a 7 bit.

2. Si utilizza l'**ottavo bit** come **flag** per la correzione, infatti se è uno, la somma dei due resti è maggiore di 128 e questo evento viene memorizzato nel segnale ***special(0)*** che sarà ritardato con delle **FIFO** di 3 **colpi di clock**, al fine di tener conto delle pipeline interne al blocco.
3. I primi 8 bit di questa prima somma sono inviati in un registro che svolge la funzione di pipeline.
4. All'uscita di questo registro è presente un **ADDER8** il quale eseguirà una seconda somma con protagonisti il terzo resto (ritardato tramite dei registri opportunamente) e la prima somma ridotta di 128 (per ridurre la somma di 128 basta porre a zero l'ottavo bit).
5. In uscita da questo secondo **ADDER** utilizziamo nuovamente l'ottavo bit della somma di uscita come flag chiamato ***special(1)***, il quale sarà ritardato di due colpi di clock con una **FIFO** e utilizzato col nome ***special_p(1)***.
6. I primi 8 bit di questa seconda somma vengono anch'essi mandati in un registro che svolge la funzione di pipeline.
7. All'uscita di questo registro è presente un ulteriore **ADDER8** dove si effettua una terza somma tra il quarto resto (ritardato tramite dei registri opportunamente) e la seconda somma ridotta di 128.
8. Quest'ultima somma va in un registro e in uscita da questo registro si usano come flag l'ottavo e il settimo bit. Banalmente se l'ottavo bit è uno vuol dire che il resto è **1.00**, se sia il settimo che l'ottavo sono uno allora il resto è **1.50**, se solo il settimo è uno allora il resto è **0.5**. Se nessuno tra l'ottavo e il settimo è uno allora il resto finale è minore di **0.5** per cui eventualmente non bisognerà aggiungere nulla.

In sostanza il risultato di questo processo è che dopo tre colpi di clock nel segnale ***special_p* (3 **downto 0**) il numero di “**uni**” presenti in questo vettore rappresenta la quantità da aggiungere per effettuare la correzione. Il codice VHDL per descrivere quanto detto è il seguente**

```

147 --GESTIONE RESTI VERDI
148 RESTI_G: ADDER8 port map(restoG2,restoG16,'0',SUMG);
149 special(3)<=SUMG(7);
150 pipeG: REG8 port map(SUMG(7 downto 0),clk,rst,SUMGp);
151 pipeG_s: FIFO_NORMAL generic map(3)
152 port map (clk,rst,special(3),special_p(3));
153 G_R64: REG8 port map(restoG64,clk,rst,restoG64p);
154
155 auxG(7 downto 0)<='0'&SUMGp(6 downto 0);
156 RESTI_G2:ADDER8 port map(restoG64p,auxG(7 downto 0),'0',SUMG2);
157 special(4)<=SUMG2(7);
158
159 pipeG2: REG8 port map(SUMG2(7 downto 0),clk,rst,SUMG2p);
160
161 FIFO_RESTOG: FIFO8_NORMAL generic map(2)
162 port map(clk,rst,restoG128,restoG128p);
163 pipeG_s1: FIFO_NORMAL generic map(2)
164 port map (clk,rst,special(4),special_p(4));
165
166 auxG(15 downto 8)<='0'&SUMG2p(6 downto 0);
167 RESTI_G3:ADDER8 port map(restoG128p,auxG(15 downto 8),'0',SUMG_F);
168 FIFO_G3: FIFO8_NORMAL generic map(1)
169 port map(clk,rst,SUMG_F(7 downto 0),SUMG_Fp);
170 special_p(5)<=SUMG_Fp(7);
171 special_p(6)<=SUMG_Fp(6);
172
173 with special_p(6 downto 3) select
174 correction_G<= conv_std_logic_vector(1,8) when "0001",
175 conv_std_logic_vector(1,8) when "0010",
176 conv_std_logic_vector(1,8) when "0100",
177 conv_std_logic_vector(2,8) when "0110",
178 conv_std_logic_vector(2,8) when "0101",
179 conv_std_logic_vector(2,8) when "0011",
180 conv_std_logic_vector(3,8) when "0111",
181
182 conv_std_logic_vector(1,8) when "1000",
183 conv_std_logic_vector(2,8) when "1001",
184 conv_std_logic_vector(2,8) when "1010",
185 conv_std_logic_vector(2,8) when "1100",
186 conv_std_logic_vector(3,8) when "1110",
187 conv_std_logic_vector(3,8) when "1101",
188 conv_std_logic_vector(3,8) when "1011",
189 conv_std_logic_vector(4,8) when "1111",
190 conv_std_logic_vector(0,8) when others;
191

```

Differisce leggermente la gestione del resto dei rossi:

```

114 --GESTIONE RESTI ROSSI
115 RESTI_R: ADDER8 port map(restoR4,restoR32,'0',SUMR);
116
117 special(0)<=SUMR(7);
118
119 pipeR: REG8 port map(SUMR(7 downto 0),clk,rst,SUMRp);
120 pipeR_s: FIFO_NORMAL generic map(2)
121 port map (clk,rst,special(0),special_p(0));
122 R_R64: FIFO8_NORMAL generic map(1)
123 port map(clk,rst,restoR64,restoR64p);
124 ausiliars(7 downto 0)<='0'&SUMRp(6 downto 0);
125 RESTI_R2:ADDER8 port map(restoR64p,ausiliars(7 downto 0),'0',SUMR_F);
126 pipeR2: REG8 port map(SUMR_F(7 downto 0),clk,rst,SUMR_Fp);
127
128 special_p(1)<=SUMR_Fp(7);
129 special_p(2)<=SUMR_Fp(6);
130
131 with special_p(2 downto 0) select
132 correction_R<= conv_std_logic_vector(1,8) when "001",
133 conv_std_logic_vector(1,8) when "010",
134 conv_std_logic_vector(1,8) when "100",
135 conv_std_logic_vector(2,8) when "110",
136 conv_std_logic_vector(2,8) when "101",
137 conv_std_logic_vector(2,8) when "011",
138 conv_std_logic_vector(3,8) when "111",
139 conv_std_logic_vector(0,8) when others;
140
141 FIFO_FINALER: FIFO8_NORMAL generic map(1)
142 port map(clk,rst,correction_R,CORRECTION_R_p);
143

```

A causa della mancanza del quarto resto i rossi completano tutto un colpo di clock in anticipo, per cui al fine di mantenere per la parte di gestione dei resti lo stesso stadio di pipeline, la correzione *correction_R* viene fatta aspettare per un colpo di clock. Infine, la parte finale è quella che unisce i risultati ottenuti in tutti i passaggi precedenti, la quale sarà illustrata di seguito:

```

239 : SUM_R_CORR: ADDER8 port map(correction_B,correction_G,'0',EXTRA);
240 :
241 ⊕ FIFO_OPR1: FIFO8_NORMAL generic map(DELAY_PIPE)
242 ⊕ port map(clk,rst,op1_R,op1_R_p);
243 ⊕ FIFO_OPR2: FIFO8 NORMAL generic map(DELAY_PIPE)
244 ⊕ port map(clk,rst,op2_R,op2_R_p);
245 ⊕ FIFO_OPR3: FIFO8_NORMAL generic map(DELAY_PIPE)
246 ⊕ port map(clk,rst,op3_R,op3_R_p);
247 ⊕ FIFO_OPG1: FIFO8 NORMAL generic map(DELAY_PIPE)
248 ⊕ port map(clk,rst,op1_G,op1_G_p);
249 ⊕ FIFO_OPG2: FIFO8_NORMAL generic map(DELAY_PIPE)
250 ⊕ port map(clk,rst,op2_G,op2_G_p);
251 ⊕ FIFO_OPG3: FIFO8 NORMAL generic map(DELAY_PIPE)
252 ⊕ port map(clk,rst,op3_G,op3_G_p);
253 ⊕ FIFO_OPG4: FIFO8_NORMAL generic map(DELAY_PIPE)
254 ⊕ port map(clk,rst,op4_G,op4_G_p);
255 ⊕ FIFO_OPB1: FIFO8_NORMAL generic map(DELAY_PIPE)
256 ⊕ port map(clk,rst,op1_B,op1_B_p);
257 ⊕ FIFO_OPB2: FIFO8 NORMAL generic map(DELAY_PIPE)
258 ⊕ port map(clk,rst,op2_B,op2_B_p);
259 ⊕ FIFO_OPB3: FIFO8 NORMAL generic map(DELAY_PIPE)
260 ⊕ port map(clk,rst,op3_B,op3_B_p);
261 ⊕ FIFO_OPB4: FIFO8_NORMAL generic map(DELAY_PIPE)
262 ⊕ port map(clk,rst,op4_B,op4_B_p);
263 ⊕ FIFO_EXTRA: FIFO8_NORMAL generic map(DIM_FIFO_EXTRA)
264 ⊕ port map(clk,rst,EXTRA(7 downto 0),FIX);
265 :
266 TIMES_0299: CARRY_SAVE4_8 port map(clk,rst,op1_R_p,op2_R_p,op3_R_p,correction_R_p,zero,results(9 downto 0));
267 TIMES_0587: CARRY_SAVE4_8 port map(clk,rst,op1_G_p,op2_G_p,op3_G_p,op4_G_p,zero,results(19 downto 10));
268 TIMES_0114: CARRY_SAVE4_8 port map(clk,rst,op1_B_p,op2_B_p,op3_B_p,op4_B_p,zero,results(29 downto 20));
269 :
270 SUM_EACH_OTHER: CARRY_SAVE4_8 port map(clk,rst,results(7 downto 0),results(17 downto 10),results(27 downto 20),FIX,zero,results(39 downto 30));
271 :
272 pix_grey<=results(37 downto 30);
273 :
274 ⊕ process(clk,rst)
275 begin
276 ⊕ if rst = '1' then
277 : pix_inp<=(others=>'0');
278 : elsif rising_edge(clk) then
279 : pix_inp<=pix_in;
280 : end if;
281 : end process;
282 ⊕ end Behavioral;
:

```

Per prima cosa (riga 239) effettuo la somma delle correzioni ottenute dai blu e dei verdi, la quale sarà rappresentata dal segnale *EXTRA*. Questo segnale andrà in ingresso ad una FIFO che lo ritarderà in modo opportuno, nel nostro caso di tre colpi di clock. Questa cosa viene fatta per un motivo ben preciso: i segnali di *right shift* riconoscibili dalla sigla "op" sono generati a tempo zero poiché sono solo una riconfigurazione dei pixel in ingresso. Quello che vogliamo è che le tre somme **TIMES_0299**, **TIMES_0587** e **TIMES_0114** avvengano in maniera sincrona tra di loro, ma nel farlo c'è da attendere il completamento della pipeline interna al blocco che genera *correction_R_p*, la quale è composta da tre stadi. Per questo motivo tutti gli operandi sono in ingresso ad una **FIFO** di dimensione generica, nel caso specifico la variabile intera **DELAY_PIPE** ha valore tre. Il segnale *EXTRA* va in ingresso come segnale **FIX** all'ultimo **CARRY_SAVE4_8** chiamato **SUM_EACH_OTHER**. Il segnale *EXTRA* è pronto allo stesso momento del segnale *correction_R*, ma va utilizzato nell'ultimo albero di somma, per cui per sincronizzarlo è stata inserita una **FIFO** con ritardo tre, poiché tre è la latenza dei carry save.

REPORT DI TIMING

Nel seguente capitolo vengono esaminati i report di timing delle quattro versioni progettate, che confrontano i path critici, i parametri di setup e di hold e il periodo di clock utilizzato.

VERSIONE 1																																																																																																																																				
Design Timing Summary																																																																																																																																				
Setup			Hold			Pulse Width																																																																																																																														
Worst Negative Slack (WNS):	0,080 ns		Worst Hold Slack (WHS):	0,051 ns		Worst Pulse Width Slack (WPWS):	0,220 ns																																																																																																																													
Total Negative Slack (TNS):	0,000 ns		Total Hold Slack (THS):	0,000 ns		Total Pulse Width Negative Slack (TPWS):	0,000 ns																																																																																																																													
Number of Failing Endpoints:	0		Number of Failing Endpoints:	0		Number of Failing Endpoints:	0																																																																																																																													
Total Number of Endpoints:	2624		Total Number of Endpoints:	2624		Total Number of Endpoints:	2723																																																																																																																													
All user specified timing constraints are met.																																																																																																																																				
Intra-Clock Paths - clk - Setup <table border="1"> <thead> <tr> <th>Name</th> <th>Slack ^{^ 1}</th> <th>Levels</th> <th>High Fanout</th> <th>From</th> <th>To</th> <th>Total Delay</th> <th>Logic Delay</th> <th>Net Delay</th> <th>Requirement</th> <th>Clock Uncertainty</th> </tr> </thead> <tbody> <tr><td>Path 1</td><td>0.080</td><td>5</td><td>2</td><td>FOR_RGB[0]...P_reg[0]/C</td><td>FOR_RGB[0].TO_HW_i/LAST_ADD/FINAL/Sum_reg[17]/D</td><td>2.370</td><td>1.767</td><td>0.603</td><td>2.4</td><td>0.035</td></tr> <tr><td>Path 2</td><td>0.088</td><td>5</td><td>2</td><td>FOR_RGB[0]...P_reg[0]/C</td><td>FOR_RGB[0].TO_HW_i/LAST_ADD/FINAL/Sum_reg[19]/D</td><td>2.362</td><td>1.759</td><td>0.603</td><td>2.4</td><td>0.035</td></tr> <tr><td>Path 3</td><td>0.154</td><td>5</td><td>2</td><td>FOR_RGB[1]...P_reg[0]/C</td><td>FOR_RGB[1].TO_HW_i/TO_M_EE/FINAL/Sum_reg[17]/D</td><td>2.207</td><td>1.788</td><td>0.419</td><td>2.4</td><td>0.035</td></tr> <tr><td>Path 4</td><td>0.160</td><td>2</td><td>8</td><td>FOR_RGB[1]...reg[18]/C</td><td>FOR_RGB[1].TO_HW_i/TO_SAT/pix_sat_reg[7]/D</td><td>2.225</td><td>0.704</td><td>1.521</td><td>2.4</td><td>0.035</td></tr> <tr><td>Path 5</td><td>0.160</td><td>0</td><td>1</td><td>FOR_RGB[1]...N_c_0/CLK</td><td>FOR_RGB[1].TO_HW_i/TO_S_5_FIFO_READY_GEN_c_1/D</td><td>2.174</td><td>1.606</td><td>0.568</td><td>2.4</td><td>0.035</td></tr> <tr><td>Path 6</td><td>0.164</td><td>5</td><td>2</td><td>FOR_RGB[0]...P_reg[0]/C</td><td>FOR_RGB[0].TO_HW_i/LAST_ADD/FINAL/Sum_reg[18]/D</td><td>2.286</td><td>1.683</td><td>0.603</td><td>2.4</td><td>0.035</td></tr> <tr><td>Path 7</td><td>0.177</td><td>1</td><td>148</td><td>F_S_M/finish_reg_rep/C</td><td>F_S_M/pixels_out_reg[151]/D</td><td>2.191</td><td>0.642</td><td>1.549</td><td>2.4</td><td>0.035</td></tr> <tr><td>Path 8</td><td>0.179</td><td>5</td><td>2</td><td>FOR_RGB[1]...P_reg[1]/C</td><td>FOR_RGB[1].TO_HW_i/TO_PRREE/FINAL/Sum_reg[17]/D</td><td>2.220</td><td>1.806</td><td>0.414</td><td>2.4</td><td>0.035</td></tr> <tr><td>Path 9</td><td>0.180</td><td>1</td><td>16</td><td>FOR_RGB[0]...c_reg[1]/C</td><td>FOR_RGB[0].TO_HW_i/TO_PR_en2[2].MUX_i/p_reg[5]/D</td><td>2.228</td><td>0.642</td><td>1.586</td><td>2.4</td><td>0.035</td></tr> <tr><td>Path 10</td><td>0.184</td><td>5</td><td>2</td><td>FOR_RGB[0]...P_reg[0]/C</td><td>FOR_RGB[0].TO_HW_i/LAST_ADD/FINAL/Sum_reg[16]/D</td><td>2.266</td><td>1.663</td><td>0.603</td><td>2.4</td><td>0.035</td></tr> </tbody> </table>												Name	Slack ^{^ 1}	Levels	High Fanout	From	To	Total Delay	Logic Delay	Net Delay	Requirement	Clock Uncertainty	Path 1	0.080	5	2	FOR_RGB[0]...P_reg[0]/C	FOR_RGB[0].TO_HW_i/LAST_ADD/FINAL/Sum_reg[17]/D	2.370	1.767	0.603	2.4	0.035	Path 2	0.088	5	2	FOR_RGB[0]...P_reg[0]/C	FOR_RGB[0].TO_HW_i/LAST_ADD/FINAL/Sum_reg[19]/D	2.362	1.759	0.603	2.4	0.035	Path 3	0.154	5	2	FOR_RGB[1]...P_reg[0]/C	FOR_RGB[1].TO_HW_i/TO_M_EE/FINAL/Sum_reg[17]/D	2.207	1.788	0.419	2.4	0.035	Path 4	0.160	2	8	FOR_RGB[1]...reg[18]/C	FOR_RGB[1].TO_HW_i/TO_SAT/pix_sat_reg[7]/D	2.225	0.704	1.521	2.4	0.035	Path 5	0.160	0	1	FOR_RGB[1]...N_c_0/CLK	FOR_RGB[1].TO_HW_i/TO_S_5_FIFO_READY_GEN_c_1/D	2.174	1.606	0.568	2.4	0.035	Path 6	0.164	5	2	FOR_RGB[0]...P_reg[0]/C	FOR_RGB[0].TO_HW_i/LAST_ADD/FINAL/Sum_reg[18]/D	2.286	1.683	0.603	2.4	0.035	Path 7	0.177	1	148	F_S_M/finish_reg_rep/C	F_S_M/pixels_out_reg[151]/D	2.191	0.642	1.549	2.4	0.035	Path 8	0.179	5	2	FOR_RGB[1]...P_reg[1]/C	FOR_RGB[1].TO_HW_i/TO_PRREE/FINAL/Sum_reg[17]/D	2.220	1.806	0.414	2.4	0.035	Path 9	0.180	1	16	FOR_RGB[0]...c_reg[1]/C	FOR_RGB[0].TO_HW_i/TO_PR_en2[2].MUX_i/p_reg[5]/D	2.228	0.642	1.586	2.4	0.035	Path 10	0.184	5	2	FOR_RGB[0]...P_reg[0]/C	FOR_RGB[0].TO_HW_i/LAST_ADD/FINAL/Sum_reg[16]/D	2.266	1.663	0.603	2.4	0.035
Name	Slack ^{^ 1}	Levels	High Fanout	From	To	Total Delay	Logic Delay	Net Delay	Requirement	Clock Uncertainty																																																																																																																										
Path 1	0.080	5	2	FOR_RGB[0]...P_reg[0]/C	FOR_RGB[0].TO_HW_i/LAST_ADD/FINAL/Sum_reg[17]/D	2.370	1.767	0.603	2.4	0.035																																																																																																																										
Path 2	0.088	5	2	FOR_RGB[0]...P_reg[0]/C	FOR_RGB[0].TO_HW_i/LAST_ADD/FINAL/Sum_reg[19]/D	2.362	1.759	0.603	2.4	0.035																																																																																																																										
Path 3	0.154	5	2	FOR_RGB[1]...P_reg[0]/C	FOR_RGB[1].TO_HW_i/TO_M_EE/FINAL/Sum_reg[17]/D	2.207	1.788	0.419	2.4	0.035																																																																																																																										
Path 4	0.160	2	8	FOR_RGB[1]...reg[18]/C	FOR_RGB[1].TO_HW_i/TO_SAT/pix_sat_reg[7]/D	2.225	0.704	1.521	2.4	0.035																																																																																																																										
Path 5	0.160	0	1	FOR_RGB[1]...N_c_0/CLK	FOR_RGB[1].TO_HW_i/TO_S_5_FIFO_READY_GEN_c_1/D	2.174	1.606	0.568	2.4	0.035																																																																																																																										
Path 6	0.164	5	2	FOR_RGB[0]...P_reg[0]/C	FOR_RGB[0].TO_HW_i/LAST_ADD/FINAL/Sum_reg[18]/D	2.286	1.683	0.603	2.4	0.035																																																																																																																										
Path 7	0.177	1	148	F_S_M/finish_reg_rep/C	F_S_M/pixels_out_reg[151]/D	2.191	0.642	1.549	2.4	0.035																																																																																																																										
Path 8	0.179	5	2	FOR_RGB[1]...P_reg[1]/C	FOR_RGB[1].TO_HW_i/TO_PRREE/FINAL/Sum_reg[17]/D	2.220	1.806	0.414	2.4	0.035																																																																																																																										
Path 9	0.180	1	16	FOR_RGB[0]...c_reg[1]/C	FOR_RGB[0].TO_HW_i/TO_PR_en2[2].MUX_i/p_reg[5]/D	2.228	0.642	1.586	2.4	0.035																																																																																																																										
Path 10	0.184	5	2	FOR_RGB[0]...P_reg[0]/C	FOR_RGB[0].TO_HW_i/LAST_ADD/FINAL/Sum_reg[16]/D	2.266	1.663	0.603	2.4	0.035																																																																																																																										

VERSIONE 2																																																																																																																																				
Design Timing Summary																																																																																																																																				
Setup			Hold			Pulse Width																																																																																																																														
Worst Negative Slack (WNS):	0,003 ns		Worst Hold Slack (WHS):	0,063 ns		Worst Pulse Width Slack (WPWS):	0,520 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:	2676		Total Number of Endpoints:	2676		Total Number of Endpoints:	2755																																																																																																																													
All user specified timing constraints are met.																																																																																																																																				
<table border="1"> <thead> <tr> <th>Name</th> <th>Slack ^{^ 1}</th> <th>Levels</th> <th>High Fanout</th> <th>From</th> <th>To</th> <th>Total Delay</th> <th>Logic Delay</th> <th>Net Delay</th> <th>Requirement</th> <th>Clock Uncertainty</th> </tr> </thead> <tbody> <tr><td>Path 1</td><td>0.003</td><td>3</td><td>33</td><td>FOR_RG...g[2]/C</td><td>FOR_RG...[6]/D</td><td>2.933</td><td>0.828</td><td>2.105</td><td>3.0</td><td>0.035</td></tr> <tr><td>Path 2</td><td>0.007</td><td>3</td><td>33</td><td>FOR_RG...g[3]/C</td><td>FOR_RG...[16]/D</td><td>2.926</td><td>0.828</td><td>2.098</td><td>3.0</td><td>0.035</td></tr> <tr><td>Path 3</td><td>0.013</td><td>3</td><td>31</td><td>FOR_RG...g[0]/C</td><td>FOR_RG...[50]/D</td><td>2.959</td><td>0.828</td><td>2.131</td><td>3.0</td><td>0.035</td></tr> <tr><td>Path 4</td><td>0.015</td><td>3</td><td>33</td><td>FOR_RG...g[1]/C</td><td>FOR_RG...[53]/D</td><td>2.979</td><td>0.828</td><td>2.151</td><td>3.0</td><td>0.035</td></tr> <tr><td>Path 5</td><td>0.022</td><td>3</td><td>33</td><td>FOR_RG...g[1]/C</td><td>FOR_RG...[70]/D</td><td>2.951</td><td>0.828</td><td>2.123</td><td>3.0</td><td>0.035</td></tr> <tr><td>Path 6</td><td>0.023</td><td>3</td><td>33</td><td>FOR_RG...g[1]/C</td><td>FOR_RG...[21]/D</td><td>2.911</td><td>0.828</td><td>2.083</td><td>3.0</td><td>0.035</td></tr> <tr><td>Path 7</td><td>0.028</td><td>3</td><td>33</td><td>FOR_RG...g[3]/C</td><td>FOR_RG...[23]/D</td><td>2.906</td><td>0.828</td><td>2.078</td><td>3.0</td><td>0.035</td></tr> <tr><td>Path 8</td><td>0.029</td><td>3</td><td>33</td><td>FOR_RG...g[1]/C</td><td>FOR_RG...[68]/D</td><td>2.910</td><td>0.828</td><td>2.082</td><td>3.0</td><td>0.035</td></tr> <tr><td>Path 9</td><td>0.030</td><td>3</td><td>33</td><td>FOR_RG...g[1]/C</td><td>FOR_RG...[71]/D</td><td>2.904</td><td>0.828</td><td>2.076</td><td>3.0</td><td>0.035</td></tr> <tr><td>Path 10</td><td>0.036</td><td>3</td><td>33</td><td>FOR_RG...g[1]/C</td><td>FOR_RG...[1]/D</td><td>2.902</td><td>0.828</td><td>2.074</td><td>3.0</td><td>0.035</td></tr> </tbody> </table>												Name	Slack ^{^ 1}	Levels	High Fanout	From	To	Total Delay	Logic Delay	Net Delay	Requirement	Clock Uncertainty	Path 1	0.003	3	33	FOR_RG...g[2]/C	FOR_RG...[6]/D	2.933	0.828	2.105	3.0	0.035	Path 2	0.007	3	33	FOR_RG...g[3]/C	FOR_RG...[16]/D	2.926	0.828	2.098	3.0	0.035	Path 3	0.013	3	31	FOR_RG...g[0]/C	FOR_RG...[50]/D	2.959	0.828	2.131	3.0	0.035	Path 4	0.015	3	33	FOR_RG...g[1]/C	FOR_RG...[53]/D	2.979	0.828	2.151	3.0	0.035	Path 5	0.022	3	33	FOR_RG...g[1]/C	FOR_RG...[70]/D	2.951	0.828	2.123	3.0	0.035	Path 6	0.023	3	33	FOR_RG...g[1]/C	FOR_RG...[21]/D	2.911	0.828	2.083	3.0	0.035	Path 7	0.028	3	33	FOR_RG...g[3]/C	FOR_RG...[23]/D	2.906	0.828	2.078	3.0	0.035	Path 8	0.029	3	33	FOR_RG...g[1]/C	FOR_RG...[68]/D	2.910	0.828	2.082	3.0	0.035	Path 9	0.030	3	33	FOR_RG...g[1]/C	FOR_RG...[71]/D	2.904	0.828	2.076	3.0	0.035	Path 10	0.036	3	33	FOR_RG...g[1]/C	FOR_RG...[1]/D	2.902	0.828	2.074	3.0	0.035
Name	Slack ^{^ 1}	Levels	High Fanout	From	To	Total Delay	Logic Delay	Net Delay	Requirement	Clock Uncertainty																																																																																																																										
Path 1	0.003	3	33	FOR_RG...g[2]/C	FOR_RG...[6]/D	2.933	0.828	2.105	3.0	0.035																																																																																																																										
Path 2	0.007	3	33	FOR_RG...g[3]/C	FOR_RG...[16]/D	2.926	0.828	2.098	3.0	0.035																																																																																																																										
Path 3	0.013	3	31	FOR_RG...g[0]/C	FOR_RG...[50]/D	2.959	0.828	2.131	3.0	0.035																																																																																																																										
Path 4	0.015	3	33	FOR_RG...g[1]/C	FOR_RG...[53]/D	2.979	0.828	2.151	3.0	0.035																																																																																																																										
Path 5	0.022	3	33	FOR_RG...g[1]/C	FOR_RG...[70]/D	2.951	0.828	2.123	3.0	0.035																																																																																																																										
Path 6	0.023	3	33	FOR_RG...g[1]/C	FOR_RG...[21]/D	2.911	0.828	2.083	3.0	0.035																																																																																																																										
Path 7	0.028	3	33	FOR_RG...g[3]/C	FOR_RG...[23]/D	2.906	0.828	2.078	3.0	0.035																																																																																																																										
Path 8	0.029	3	33	FOR_RG...g[1]/C	FOR_RG...[68]/D	2.910	0.828	2.082	3.0	0.035																																																																																																																										
Path 9	0.030	3	33	FOR_RG...g[1]/C	FOR_RG...[71]/D	2.904	0.828	2.076	3.0	0.035																																																																																																																										
Path 10	0.036	3	33	FOR_RG...g[1]/C	FOR_RG...[1]/D	2.902	0.828	2.074	3.0	0.035																																																																																																																										

VERSIONE 3 32X32

Design Timing Summary

Setup	Hold	Pulse Width
Worst Negative Slack (WNS): 0,032 ns	Worst Hold Slack (WHS): 0,048 ns	Worst Pulse Width Slack (WPWS): 0,420 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: 3687	Total Number of Endpoints: 3687	Total Number of Endpoints: 3766

All user specified timing constraints are met.

Intra-Clock Paths - clk - Setup

Name	Slack ^1	Levels	Routes	High Fanout	From	To	Total Delay	Logic Delay	Net Delay	Requirement	Clock Uncertainty
Path 1	0.032	2	3	23	F_S_M/FSM_onehot_state_reg[9]/C	F_S_M/FSM_o...e_reg[10]/CE	2.507	0.704	1.803	2.8	0.035
Path 2	0.032	2	3	23	F_S_M/FSM_onehot_state_reg[9]/C	F_S_M/FSM_o...e_reg[1]/CE	2.507	0.704	1.803	2.8	0.035
Path 3	0.032	2	3	23	F_S_M/FSM_onehot_state_reg[9]/C	F_S_M/FSM_o...e_reg[7]/CE	2.507	0.704	1.803	2.8	0.035
Path 4	0.040	2	3	24	F_S_M/FSM_onehot_state_reg[10]/C	F_S_M/FSM_o...e_reg[0]/CE	2.493	0.704	1.789	2.8	0.035
Path 5	0.040	2	3	24	F_S_M/FSM_onehot_state_reg[10]/C	F_S_M/FSM_o...e_reg[2]/CE	2.493	0.704	1.789	2.8	0.035
Path 6	0.040	2	3	24	F_S_M/FSM_onehot_state_reg[10]/C	F_S_M/FSM_o...e_reg[3]/CE	2.493	0.704	1.789	2.8	0.035
Path 7	0.040	2	3	24	F_S_M/FSM_onehot_state_reg[10]/C	F_S_M/FSM_o...e_reg[4]/CE	2.493	0.704	1.789	2.8	0.035
Path 8	0.040	2	3	24	F_S_M/FSM_onehot_state_reg[10]/C	F_S_M/FSM_o...e_reg[5]/CE	2.493	0.704	1.789	2.8	0.035
Path 9	0.040	2	3	24	F_S_M/FSM_onehot_state_reg[10]/C	F_S_M/FSM_o...e_reg[6]/CE	2.493	0.704	1.789	2.8	0.035
Path 10	0.040	2	3	24	F_S_M/FSM_onehot_state_reg[10]/C	F_S_M/FSM_o...e_reg[8]/CE	2.493	0.704	1.789	2.8	0.035

VERSIONE 3 64X64

Design Timing Summary

Setup	Hold	Pulse Width
Worst Negative Slack (WNS): 0,015 ns	Worst Hold Slack (WHS): 0,074 ns	Worst Pulse Width Slack (WPWS): 0,420 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: 3780	Total Number of Endpoints: 3780	Total Number of Endpoints: 3857

All user specified timing constraints are met.

Intra-Clock Paths - clk - Setup

Name	Slack	Levels	High Fanout	From	To	Total Delay	Logic Delay	Net Delay	Requirement	Clock Uncertainty
Path 1	0.015	2	23	F_S_M/FSM_on...te_reg[9]/C	F_S_M/FSM_o...e_reg[3]/CE	2.523	0.839	1.684	2.8	0.035
Path 2	0.015	2	23	F_S_M/FSM_on...te_reg[9]/C	F_S_M/FSM_o...e_reg[4]/CE	2.523	0.839	1.684	2.8	0.035
Path 3	0.015	2	23	F_S_M/FSM_on...te_reg[9]/C	F_S_M/FSM_o...e_reg[5]/CE	2.523	0.839	1.684	2.8	0.035
Path 4	0.015	2	23	F_S_M/FSM_on...te_reg[9]/C	F_S_M/FSM_o...e_reg[8]/CE	2.523	0.839	1.684	2.8	0.035
Path 5	0.037	2	23	F_S_M/FSM_on...te_reg[9]/C	F_S_M/FSM_o...e_reg[0]/CE	2.523	0.839	1.684	2.8	0.035
Path 6	0.037	2	23	F_S_M/FSM_on...te_reg[9]/C	F_S_M/FSM_o...e_reg[10]/CE	2.523	0.839	1.684	2.8	0.035
Path 7	0.037	2	23	F_S_M/FSM_on...te_reg[9]/C	F_S_M/FSM_o...e_reg[1]/CE	2.523	0.839	1.684	2.8	0.035
Path 8	0.037	2	23	F_S_M/FSM_on...te_reg[9]/C	F_S_M/FSM_o...e_reg[2]/CE	2.523	0.839	1.684	2.8	0.035
Path 9	0.037	2	23	F_S_M/FSM_on...te_reg[9]/C	F_S_M/FSM_o...e_reg[6]/CE	2.523	0.839	1.684	2.8	0.035
Path 10	0.037	2	23	F_S_M/FSM_on...te_reg[9]/C	F_S_M/FSM_o...e_reg[7]/CE	2.523	0.839	1.684	2.8	0.035

VERSIONE 4

Design Timing Summary

Setup

Worst Negative Slack (WNS): **0,001 ns**

Total Negative Slack (TNS): **0,000 ns**

Number of Failing Endpoints: **0**

Total Number of Endpoints: **2920**

Hold

Worst Hold Slack (WHS): **0,036 ns**

Total Hold Slack (THS): **0,000 ns**

Number of Failing Endpoints: **0**

Total Number of Endpoints: **2920**

Pulse Width

Worst Pulse Width Slack (WPWS): **0,470 ns**

Total Pulse Width Negative Slack (TPWS): **0,000 ns**

Number of Failing Endpoints: **0**

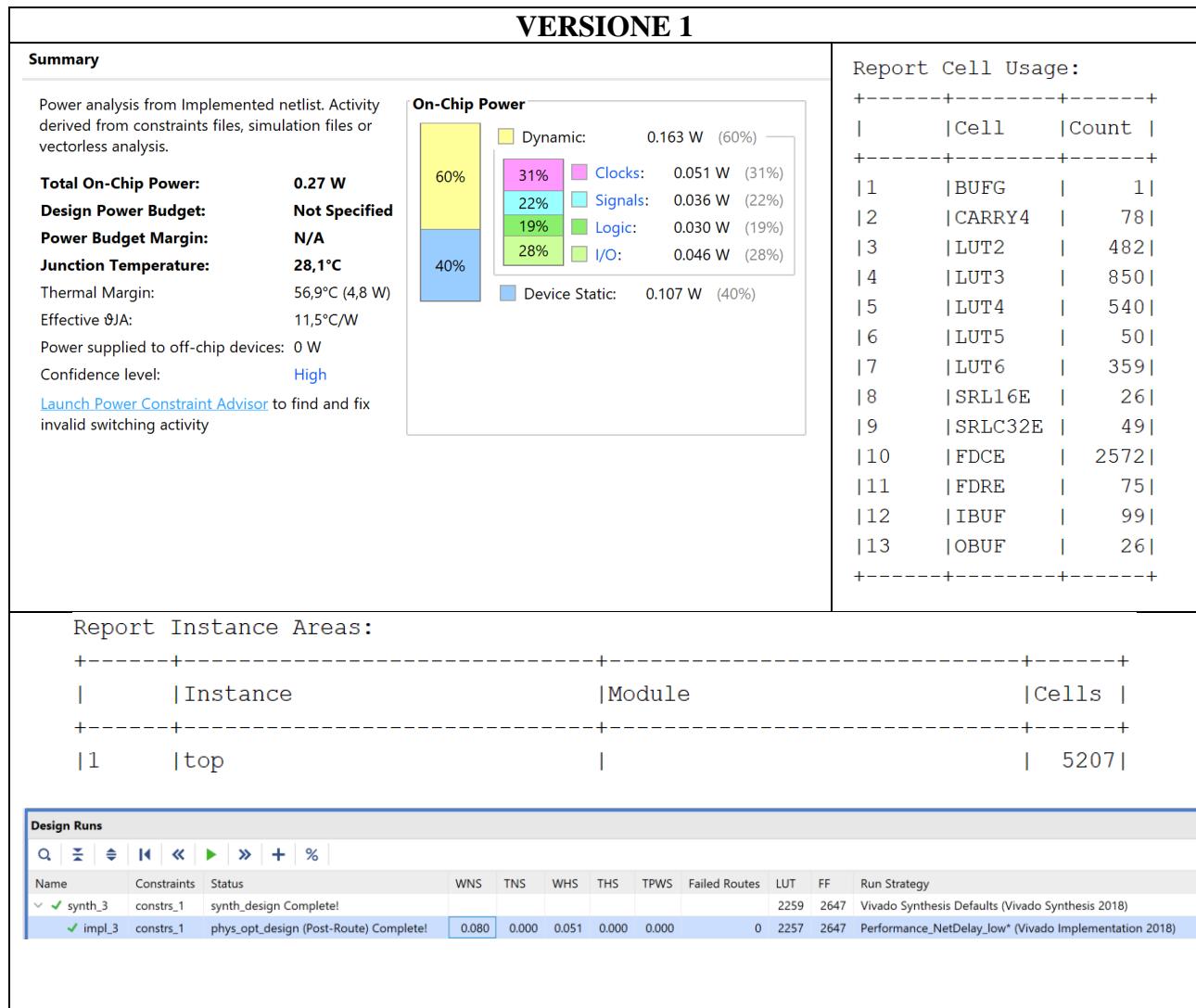
Total Number of Endpoints: **3016**

All user specified timing constraints are met.

Intra-Clock Paths - clk - Setup										
Name	Slack ^{^ 1}	Levels	High Fanout	From	To	Total Delay	Logic Delay	Net Delay	Requirement	Clock Uncertainty
Path 1	0.001	2	24	FOR_RG...ca_1/C	DECOD...[0]/D	2.775	0.704	2.071	2.9	0.035
Path 2	0.025	3	24	FOR_RG...g[2]/C	FOR_RG...[20]/D	2.809	0.828	1.981	2.9	0.035
Path 3	0.028	3	24	FOR_RG...g[2]/C	FOR_RG...[4]/D	2.806	0.828	1.978	2.9	0.035
Path 4	0.032	3	24	FOR_RG...llica/C	FOR_RG...[68]/D	2.841	0.828	2.013	2.9	0.035
Path 5	0.035	3	24	FOR_RG...g[2]/C	FOR_RG...[17]/D	2.849	0.828	2.021	2.9	0.035
Path 6	0.037	3	24	FOR_RG...g[2]/C	DECOD...[19]/D	2.835	0.828	2.007	2.9	0.035
Path 7	0.047	3	24	FOR_RG...g[2]/C	FOR_RG...[17]/D	2.789	0.828	1.961	2.9	0.035
Path 8	0.059	3	24	FOR_RG...llica/C	FOR_RG...[70]/D	2.833	0.828	2.005	2.9	0.035
Path 9	0.060	3	24	FOR_RG...llica/C	FOR_RG...[70]/D	2.882	0.828	2.054	2.9	0.035
Path 10	0.065	3	24	FOR_RG...g[2]/C	FOR_RG...[22]/D	2.768	0.828	1.940	2.9	0.035

REPORT DI POTENZA E RISORSE

Nel seguente capitolo vengono presentati i report di potenza delle diverse versioni del circuito, rivolgendo in particolare l'attenzione ai seguenti aspetti: la potenza dissipata statica e dinamica, le tipologie di risorse utilizzate e la frequenza di clock a cui opera il circuito. Questa analisi è necessaria per valutare l'impatto della velocità operativa del circuito e delle risorse utilizzate sulla dissipazione complessiva di potenza nel circuito.



Il circuito di filtraggio con la FSM minimale, caratterizzata da una gestione dei bordi unica (toroidale mista), si distingue per il suo basso consumo energetico, pari a **0.27 W**, dovuto all'uso di meno risorse rispetto agli altri circuiti (5207 celle). In tutti i circuiti realizzati si può osservare che la maggior parte della potenza è di tipo dinamico, principalmente a causa dei periodi di Clock molto brevi, in questo caso si lavora con un periodo di clock di 2.8 ns, ed è presente una elevata probabilità di transizioni. Questo è comprensibile, dato che si tratta di un circuito per il filtraggio di immagini, che esegue operazioni di somma e moltiplicazione.

VERSIONE 2

<p>Power analysis from Implemented netlist. Activity derived from constraints files, simulation files or vectorless analysis.</p> <p>Total On-Chip Power: 0.255 W Design Power Budget: Not Specified Power Budget Margin: N/A Junction Temperature: 27,9°C Thermal Margin: 57,1°C (4,8 W) Effective θJA: 11,5°C/W Power supplied to off-chip devices: 0 W Confidence level: High Launch Power Constraint Advisor to find and fix invalid switching activity</p>	<p>On-Chip Power</p> <table border="1"> <thead> <tr> <th>Category</th> <th>Power (W)</th> <th>Percentage</th> </tr> </thead> <tbody> <tr> <td>Dynamic</td> <td>0.149 W</td> <td>(58%)</td> </tr> <tr> <td>Clocks</td> <td>0.045 W</td> <td>(30%)</td> </tr> <tr> <td>Signals</td> <td>0.035 W</td> <td>(24%)</td> </tr> <tr> <td>Logic</td> <td>0.031 W</td> <td>(21%)</td> </tr> <tr> <td>I/O</td> <td>0.038 W</td> <td>(25%)</td> </tr> <tr> <td>Device Static</td> <td>0.107 W</td> <td>(42%)</td> </tr> </tbody> </table>	Category	Power (W)	Percentage	Dynamic	0.149 W	(58%)	Clocks	0.045 W	(30%)	Signals	0.035 W	(24%)	Logic	0.031 W	(21%)	I/O	0.038 W	(25%)	Device Static	0.107 W	(42%)	<p>Report Cell Usage:</p> <table border="1"> <thead> <tr> <th>Cell</th> <th>Count</th> </tr> </thead> <tbody> <tr><td> BUF</td><td>1</td></tr> <tr><td> CARRY4</td><td>60</td></tr> <tr><td> LUT1</td><td>2</td></tr> <tr><td> LUT2</td><td>481</td></tr> <tr><td> LUT3</td><td>879</td></tr> <tr><td> LUT4</td><td>429</td></tr> <tr><td> LUT5</td><td>153</td></tr> <tr><td> LUT6</td><td>521</td></tr> <tr><td> SRL16E</td><td>26</td></tr> <tr><td> SRLC32E</td><td>49</td></tr> <tr><td> FDCE</td><td>2604</td></tr> <tr><td> FDRE</td><td>75</td></tr> <tr><td> IBUF</td><td>101</td></tr> <tr><td> OBUF</td><td>26</td></tr> </tbody> </table>	Cell	Count	BUF	1	CARRY4	60	LUT1	2	LUT2	481	LUT3	879	LUT4	429	LUT5	153	LUT6	521	SRL16E	26	SRLC32E	49	FDCE	2604	FDRE	75	IBUF	101	OBUF	26
Category	Power (W)	Percentage																																																			
Dynamic	0.149 W	(58%)																																																			
Clocks	0.045 W	(30%)																																																			
Signals	0.035 W	(24%)																																																			
Logic	0.031 W	(21%)																																																			
I/O	0.038 W	(25%)																																																			
Device Static	0.107 W	(42%)																																																			
Cell	Count																																																				
BUF	1																																																				
CARRY4	60																																																				
LUT1	2																																																				
LUT2	481																																																				
LUT3	879																																																				
LUT4	429																																																				
LUT5	153																																																				
LUT6	521																																																				
SRL16E	26																																																				
SRLC32E	49																																																				
FDCE	2604																																																				
FDRE	75																																																				
IBUF	101																																																				
OBUF	26																																																				
<p>Report Instance Areas:</p> <table border="1"> <thead> <tr> <th>Instance</th> <th>Module</th> <th>Cells</th> </tr> </thead> <tbody> <tr><td> top</td><td></td><td>5407</td></tr> </tbody> </table>	Instance	Module	Cells	top		5407																																															
Instance	Module	Cells																																																			
top		5407																																																			

Design Runs											
		Constraints	Status	WNS	TNS	WHS	THS	TPWS	Failed Routes	LUT	FF
✓ synth_4	constrs_1		synth_design Complete!							2311	2679
✓ impl_4	constrs_1		phys_opt_design (Post-Route) Complete!	0.003	0.000	0.063	0.000	0.000	0	2306	2679

La seconda versione del circuito, che implementa varie possibilità di gestione dei bordi, rispetto alla prima spende 200 celle in più. Ciò nonostante, i consumi sono leggermente più bassi (0.255W) dovuto al fatto che in questa versione il periodo di clock è quello più alto(3.0 ns).

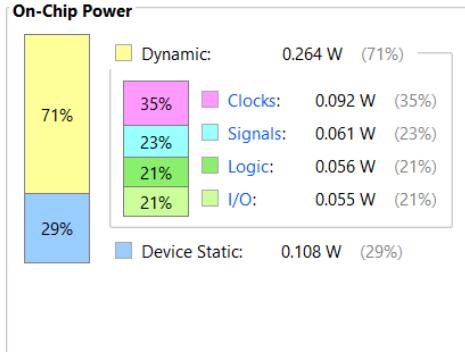
Di seguito viene riportato un confronto di consumo per la versione parametrizzata del circuito. Viene analizzato in questo caso il consumo del circuito implementato per filtrare una immagine RGB 32X32 e un circuito per immagini 64x64.

VERSIONE 3 32X32																																																																							
<p>Power analysis from Implemented netlist. Activity derived from constraints files, simulation files or vectorless analysis.</p> <p>Total On-Chip Power: 0.364 W Design Power Budget: Not Specified Power Budget Margin: N/A Junction Temperature: 29,2°C Thermal Margin: 55,8°C (4,7 W) Effective 8JA: 11,5°C/W Power supplied to off-chip devices: 0 W Confidence level: High Launch Power Constraint Advisor to find and fix invalid switching activity</p> <table border="1"> <caption>On-Chip Power</caption> <thead> <tr> <th>Category</th> <th>Percentage</th> <th>Power (W)</th> <th>Percentage</th> </tr> </thead> <tbody> <tr> <td>Dynamic</td> <td>70%</td> <td>0.256 W</td> <td>(70%)</td> </tr> <tr> <td>Clocks</td> <td>34%</td> <td>0.087 W</td> <td>(34%)</td> </tr> <tr> <td>Signals</td> <td>23%</td> <td>0.059 W</td> <td>(23%)</td> </tr> <tr> <td>Logic</td> <td>21%</td> <td>0.054 W</td> <td>(21%)</td> </tr> <tr> <td>I/O</td> <td>22%</td> <td>0.055 W</td> <td>(22%)</td> </tr> <tr> <td>Device Static</td> <td>30%</td> <td>0.108 W</td> <td>(30%)</td> </tr> </tbody> </table>	Category	Percentage	Power (W)	Percentage	Dynamic	70%	0.256 W	(70%)	Clocks	34%	0.087 W	(34%)	Signals	23%	0.059 W	(23%)	Logic	21%	0.054 W	(21%)	I/O	22%	0.055 W	(22%)	Device Static	30%	0.108 W	(30%)	<pre>Report Cell Usage: +-----+-----+ Cell Count +-----+-----+ 1 BUF6 1 2 CARRY4 78 3 LUT1 2 4 LUT2 481 5 LUT3 851 6 LUT4 321 7 LUT5 55 8 LUT6 1674 9 MUXF7 1080 10 MUXF8 216 11 SRL16E 26 12 SRLC32E 49 13 FDCE 3607 14 FDPE 1 15 FDRE 75 16 IBUF 101 17 OBUF 26 +-----+-----+</pre> <p>Report Instance Areas:</p> <table border="1"> <thead> <tr> <th>Instance</th> <th>Module</th> <th>Cells</th> </tr> </thead> <tbody> <tr> <td> 1 top</td> <td> </td> <td> 8644 </td> </tr> </tbody> </table> <table border="1"> <thead> <tr> <th>Name</th> <th>Constraints</th> <th>Status</th> <th>WNS</th> <th>TNS</th> <th>WHS</th> <th>THS</th> <th>TPWS</th> <th>Total Power</th> <th>Failed Routes</th> <th>LUT</th> <th>FF</th> </tr> </thead> <tbody> <tr> <td>synth_7</td> <td>constrs_1</td> <td>synth_design Complete!</td> <td>0.032</td> <td>0.000</td> <td>0.048</td> <td>0.000</td> <td>0.000</td> <td>0.421</td> <td>0</td> <td>3455</td> <td>3683</td> </tr> <tr> <td>impl_7</td> <td>constrs_1</td> <td>phys_opt_design (Post-Route) Complete!</td> <td></td> <td></td> <td></td> <td></td> <td></td> <td></td> <td></td> <td></td> <td></td> </tr> </tbody> </table>	Instance	Module	Cells	1 top		8644	Name	Constraints	Status	WNS	TNS	WHS	THS	TPWS	Total Power	Failed Routes	LUT	FF	synth_7	constrs_1	synth_design Complete!	0.032	0.000	0.048	0.000	0.000	0.421	0	3455	3683	impl_7	constrs_1	phys_opt_design (Post-Route) Complete!									
Category	Percentage	Power (W)	Percentage																																																																				
Dynamic	70%	0.256 W	(70%)																																																																				
Clocks	34%	0.087 W	(34%)																																																																				
Signals	23%	0.059 W	(23%)																																																																				
Logic	21%	0.054 W	(21%)																																																																				
I/O	22%	0.055 W	(22%)																																																																				
Device Static	30%	0.108 W	(30%)																																																																				
Instance	Module	Cells																																																																					
1 top		8644																																																																					
Name	Constraints	Status	WNS	TNS	WHS	THS	TPWS	Total Power	Failed Routes	LUT	FF																																																												
synth_7	constrs_1	synth_design Complete!	0.032	0.000	0.048	0.000	0.000	0.421	0	3455	3683																																																												
impl_7	constrs_1	phys_opt_design (Post-Route) Complete!																																																																					

Questa versione presenta un numero relativamente elevato di LUT e FLIP FLOP rispetto agli altri circuiti di filtraggio 32x32, il chè porta il consumo totale del chip a **364 mW**, ripartiti in:

- consumo di potenza statico = **108 mW (30%)**;
- consumo di potenza dinamico = **256 mW (70%)**;

L'aumento percentuale del consumo dinamico è conseguente all'aumento delle LUT. Ciò è dovuto principalmente alla frequenza di clock pari a 2.8 ns; esso infatti è il segnale che consuma da solo il **34%** della potenza dinamica.

VERSIONE 3 64 X 64																																								
<p>Power analysis from Implemented netlist. Activity derived from constraints files, simulation files or vectorless analysis.</p> <p>Total On-Chip Power: 0.372 W Design Power Budget: Not Specified Power Budget Margin: N/A Junction Temperature: 29,3°C Thermal Margin: 55,7°C (4,7 W) Effective θJA: 11,5°C/W Power supplied to off-chip devices: 0 W Confidence level: High</p>	 <table border="1"> <thead> <tr> <th>Category</th> <th>Power (W)</th> <th>Percentage</th> </tr> </thead> <tbody> <tr> <td>Dynamic</td> <td>0.264 W</td> <td>(71%)</td> </tr> <tr> <td>Clocks</td> <td>0.092 W</td> <td>(35%)</td> </tr> <tr> <td>Signals</td> <td>0.061 W</td> <td>(23%)</td> </tr> <tr> <td>Logic</td> <td>0.056 W</td> <td>(21%)</td> </tr> <tr> <td>I/O</td> <td>0.055 W</td> <td>(21%)</td> </tr> <tr> <td>Device Static</td> <td>0.108 W</td> <td>(29%)</td> </tr> </tbody> </table>	Category	Power (W)	Percentage	Dynamic	0.264 W	(71%)	Clocks	0.092 W	(35%)	Signals	0.061 W	(23%)	Logic	0.056 W	(21%)	I/O	0.055 W	(21%)	Device Static	0.108 W	(29%)																		
Category	Power (W)	Percentage																																						
Dynamic	0.264 W	(71%)																																						
Clocks	0.092 W	(35%)																																						
Signals	0.061 W	(23%)																																						
Logic	0.056 W	(21%)																																						
I/O	0.055 W	(21%)																																						
Device Static	0.108 W	(29%)																																						
	<p>Report Cell Usage:</p> <table border="1"> <thead> <tr> <th>Cell Type</th> <th>Count</th> </tr> </thead> <tbody> <tr><td> BUF</td><td>11</td></tr> <tr><td> CARRY4</td><td>78</td></tr> <tr><td> LUT1</td><td>2</td></tr> <tr><td> LUT2</td><td>482</td></tr> <tr><td> LUT3</td><td>851</td></tr> <tr><td> LUT4</td><td>321</td></tr> <tr><td> LUT5</td><td>45</td></tr> <tr><td> LUT6</td><td>1685</td></tr> <tr><td> MUXF7</td><td>1080</td></tr> <tr><td> MUXF8</td><td>216</td></tr> <tr><td> SRL16E</td><td>26</td></tr> <tr><td> SRLC32E</td><td>98</td></tr> <tr><td> FDCE</td><td>3641</td></tr> <tr><td> FDPE</td><td>11</td></tr> <tr><td> FDRE</td><td>75</td></tr> <tr><td> IBUF</td><td>101</td></tr> <tr><td> OBUF</td><td>26</td></tr> </tbody> </table>	Cell Type	Count	BUF	11	CARRY4	78	LUT1	2	LUT2	482	LUT3	851	LUT4	321	LUT5	45	LUT6	1685	MUXF7	1080	MUXF8	216	SRL16E	26	SRLC32E	98	FDCE	3641	FDPE	11	FDRE	75	IBUF	101	OBUF	26			
Cell Type	Count																																							
BUF	11																																							
CARRY4	78																																							
LUT1	2																																							
LUT2	482																																							
LUT3	851																																							
LUT4	321																																							
LUT5	45																																							
LUT6	1685																																							
MUXF7	1080																																							
MUXF8	216																																							
SRL16E	26																																							
SRLC32E	98																																							
FDCE	3641																																							
FDPE	11																																							
FDRE	75																																							
IBUF	101																																							
OBUF	26																																							
	<p>Report Instance Areas:</p> <table border="1"> <thead> <tr> <th>Instance</th> <th>Module</th> <th>Cells</th> </tr> </thead> <tbody> <tr><td> 1 top</td><td> </td><td> 8729 </td></tr> </tbody> </table>	Instance	Module	Cells	1 top		8729																																	
Instance	Module	Cells																																						
1 top		8729																																						
<p>Design Runs</p> <table border="1"> <thead> <tr> <th>Name</th> <th>Constraints</th> <th>Status</th> <th>WNS</th> <th>TNS</th> <th>WHS</th> <th>THS</th> <th>TPWS</th> <th>Total</th> <th>Failed Routes</th> <th>LUT</th> <th>FF</th> <th>Run Strategy</th> </tr> </thead> <tbody> <tr> <td>synth_6 (active)</td> <td>constrs_1</td> <td>synth_design Complete!</td> <td>0.015</td> <td>0.000</td> <td>0.074</td> <td>0.000</td> <td>0.000</td> <td>0.426</td> <td></td> <td>3510</td> <td>3717</td> <td>Vivado Synthesis Defaults (Vivado Synthesis 2018)</td> </tr> <tr> <td>Impl_6 (active)</td> <td>constrs_1</td> <td>phys_opt_design (Post-Route) Complete!</td> <td></td> <td></td> <td></td> <td></td> <td></td> <td></td> <td>0</td> <td>3822</td> <td>3717</td> <td>Performance_NetDelay_low (Vivado Implementation 2018)</td> </tr> </tbody> </table>	Name	Constraints	Status	WNS	TNS	WHS	THS	TPWS	Total	Failed Routes	LUT	FF	Run Strategy	synth_6 (active)	constrs_1	synth_design Complete!	0.015	0.000	0.074	0.000	0.000	0.426		3510	3717	Vivado Synthesis Defaults (Vivado Synthesis 2018)	Impl_6 (active)	constrs_1	phys_opt_design (Post-Route) Complete!							0	3822	3717	Performance_NetDelay_low (Vivado Implementation 2018)	
Name	Constraints	Status	WNS	TNS	WHS	THS	TPWS	Total	Failed Routes	LUT	FF	Run Strategy																												
synth_6 (active)	constrs_1	synth_design Complete!	0.015	0.000	0.074	0.000	0.000	0.426		3510	3717	Vivado Synthesis Defaults (Vivado Synthesis 2018)																												
Impl_6 (active)	constrs_1	phys_opt_design (Post-Route) Complete!							0	3822	3717	Performance_NetDelay_low (Vivado Implementation 2018)																												

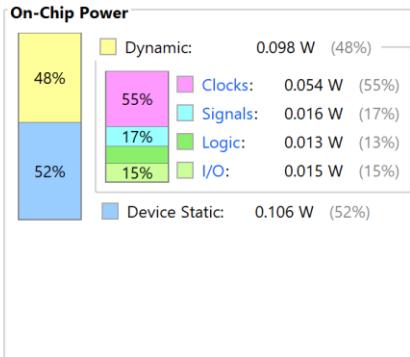
La configurazione della terza versione per il filtraggio di un'immagine 64x64 ha comportato un incremento del numero di risorse in confronto alla precedente configurazione, rispettivamente di 57 LUT e 34 FLIP FLOP.

VERSIONE 4 *rgb2gray* ALTO

Summary

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

Total On-Chip Power: 0.204 W
Design Power Budget: Not Specified
Power Budget Margin: N/A
Junction Temperature: 27,3°C
 Thermal Margin: 57,7°C (4,8 W)
 Effective θJA: 11,5°C/W
 Power supplied to off-chip devices: 0 W
 Confidence level: High
[Launch Power Constraint Advisor](#) to find and fix invalid switching activity



Report Cell Usage:

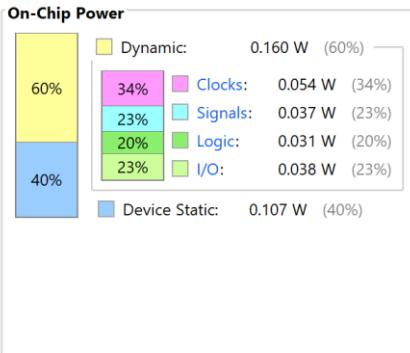
	Cell	Count
1	BUFG	1
2	CARRY4	60
3	LUT2	587
4	LUT3	938
5	LUT4	452
6	LUT5	171
7	LUT6	531
8	SRL16E	35
9	SRLC32E	49
10	FDCE	2838
11	FDRE	84
12	IBUF	102
13	OBUF	26

VERSIONE 4 *rgb2gray* BASSO

Summary

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

Total On-Chip Power: 0.267 W
Design Power Budget: Not Specified
Power Budget Margin: N/A
Junction Temperature: 28,1°C
 Thermal Margin: 56,9°C (4,8 W)
 Effective θJA: 11,5°C/W
 Power supplied to off-chip devices: 0 W
 Confidence level: High
[Launch Power Constraint Advisor](#) to find and fix invalid switching activity



x PW_WITHOUTGRAY x

Report Instance Areas:

	Instance	Module	Cells
1	top		5874

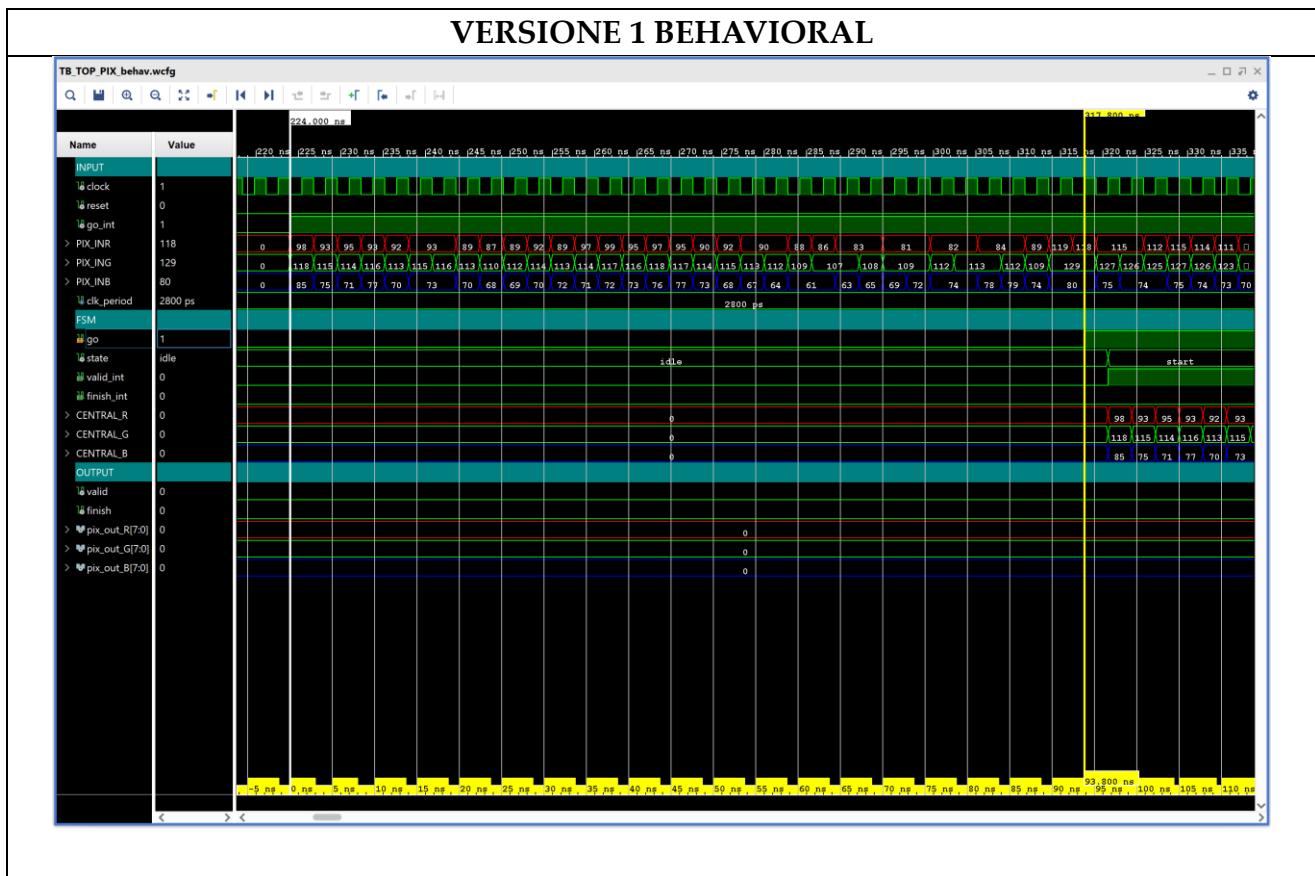
Design Runs		Run Summary									
Name	Status	WNS	TNS	WHS	THS	TPWS	Failed Routes	LUT	FF	Run Strategy	
synth_2 (active)	constrs_1 synth_design Complete!	0.001	0.000	0.036	0.000	0.000	0	2489	2922	Vivado Synthesis Defaults (Vivado Synthesis 2018)	
impl_2 (active)	constrs_1 phys_opt_design (Post-Route) Complete!						0	2482	2922	Vivado Implementation Defaults* (Vivado Implementation 2018)	

Nell'ultima versione è possibile osservare come le potenze in gioco varino significativamente a seconda che il segnale **RGB2GRAY** sia alto o basso. Quando è alto, per il calcolo dell'immagine filtrata in scala di grigi si utilizza virtualmente solo un terzo dell'hardware necessario per le immagini **RGB**, con conseguente minore consumo di potenza. È interessante notare che, nonostante la versione 4 sia pressoché identica alla versione 2, a meno del blocco aggiuntivo per la conversione a grigi, essa risulta più veloce; infatti, vanta un periodo di clock di 2.9 ns contro i 3 ns della versione 2. L'aumento della frequenza di clock e le maggiori risorse impiegate sono tuttavia causa del leggero aumento del consumo di potenza massima.

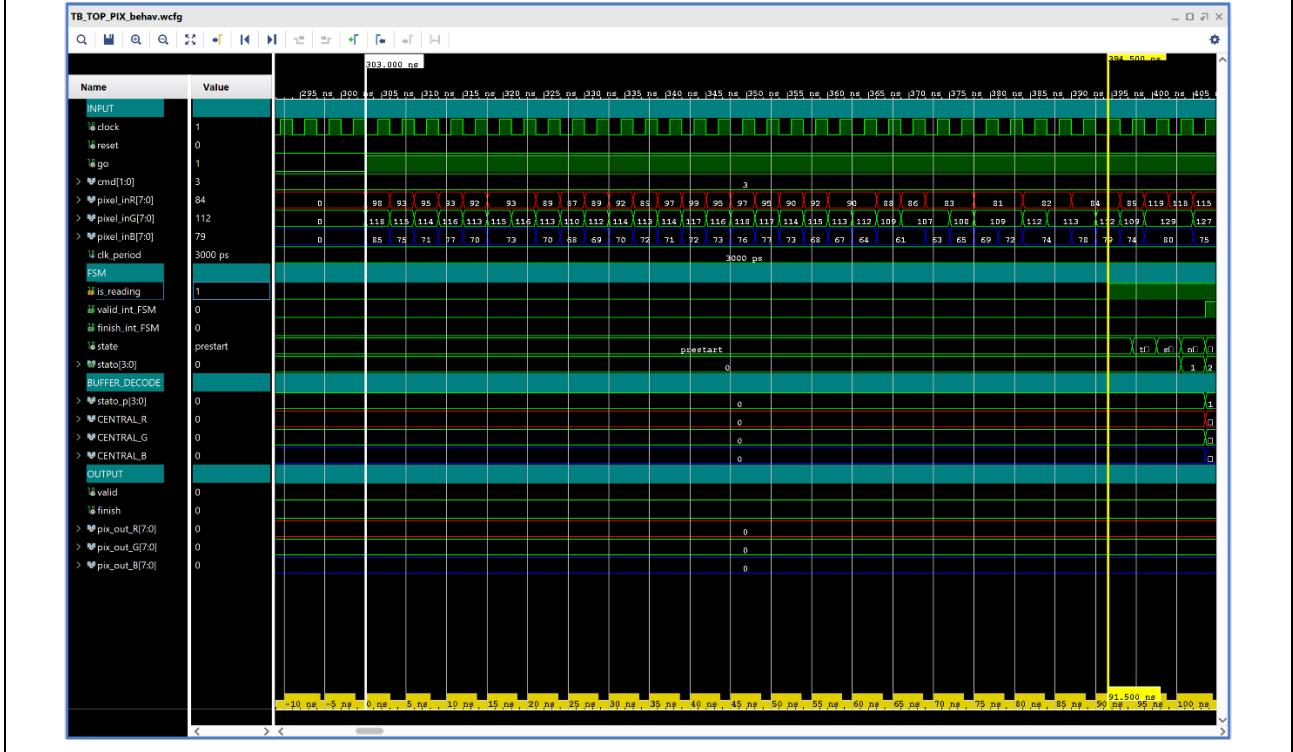
SIMULAZIONE BEHAVIORAL DEI SEGNALI

Nel seguente capitolo andremo a osservare le simulazioni delle varie versioni concentrando sui punti fondamentali. A scopo squisitamente didattico l'attenzione sarà focalizzata sulle simulazioni comportamentali, in modo tale da visualizzare esattamente ciò che è stato descritto. Saranno ovviamente presentate e discusse anche le simulazioni post timing, le quali, come vedremo in seguito, produrranno risultati coerenti con le simulazioni comportamentali.

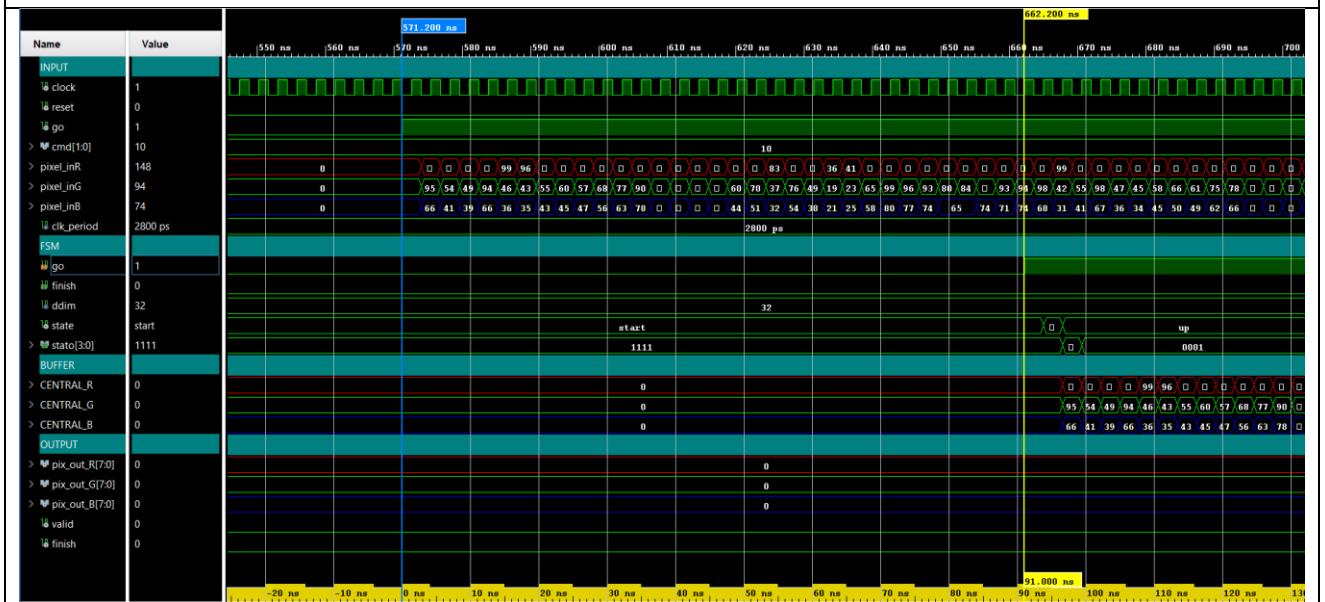
SEGNALE DI INIZIO LETTURA

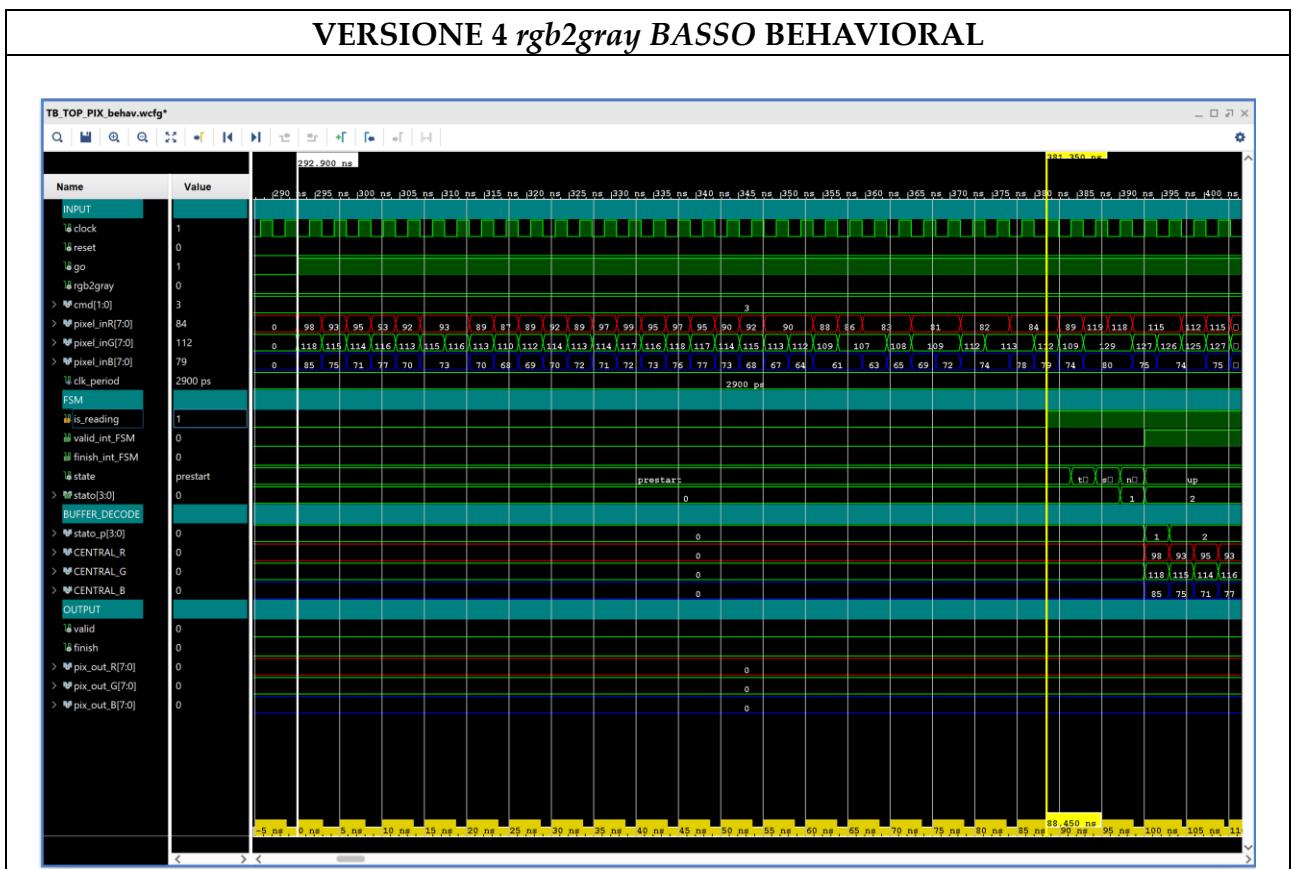
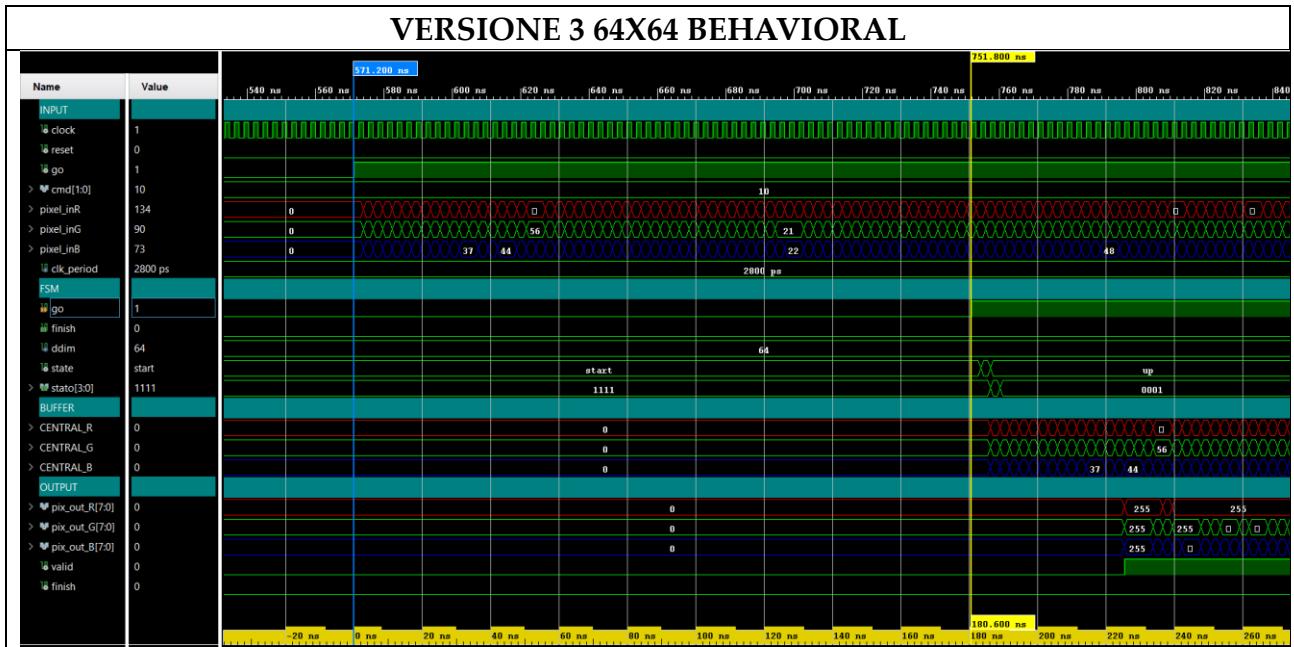


VERSIONE 2 BEHAVIORAL

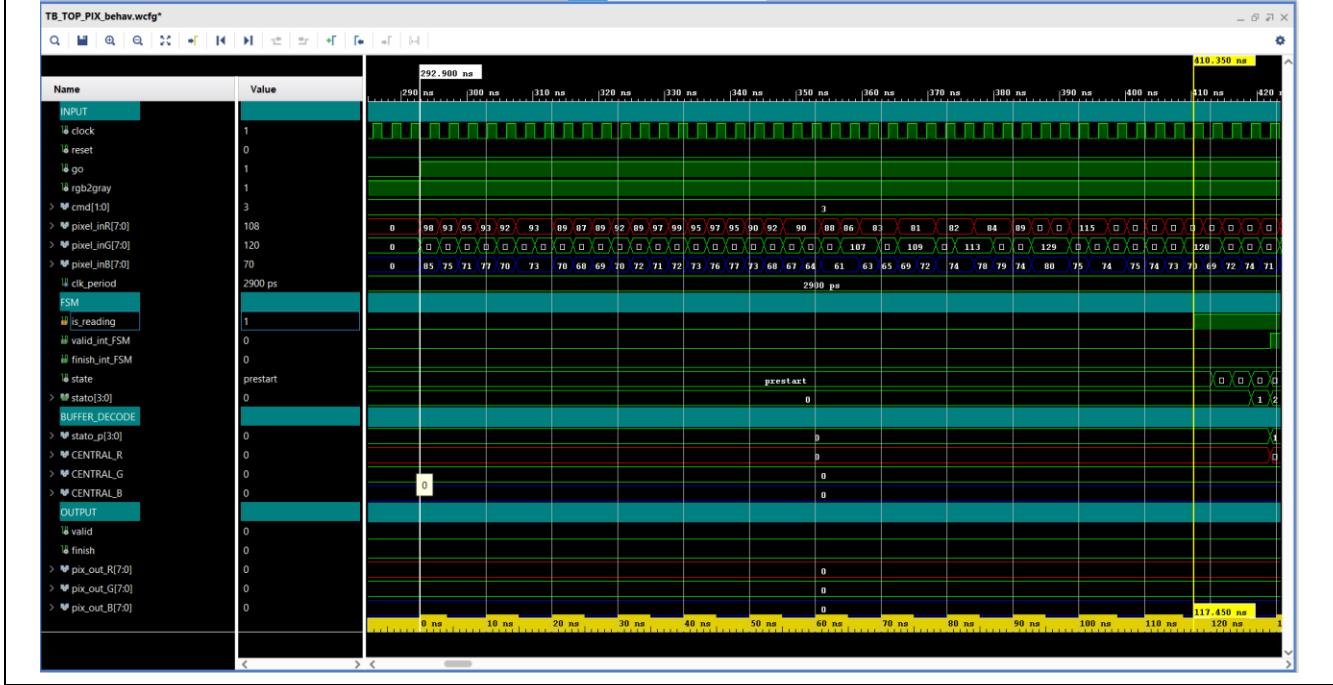


VERSIONE 3 32X32 BEHAVIORAL





VERSIONE 4 *rgb2gray* ALTO BEHAVIORAL



- **SEGNALE GO**

Nella prima versione il segnale **go_int** è generato alla lettura del primo pixel da file e funge da *trigger* per la **FSM** col nome di **go**.

Possiamo notare dall'immagine come il segnale di **go_int** e il segnale di **go** siano sfasati di 93.8 ns che sono esattamente $33.5 * \text{clk_period}$. Il segnale di **go** proveniente da test bench viene inserito in ingresso ad una FIFO di dimensione 34 (l'uscita di tale FIFO è visibile dopo $(\text{DIM_FIFO} - 1) * \text{clk_period}$ dopo il primo fronte di salita del clock).

Per garantire i tempi di setup e di hold, il segnale **go_int** passa allo stato logico alto metà colpo di clock in anticipo.

Analoga cosa succede nelle successive versioni.

Nella seconda versione il segnale **go_int** è generato alla lettura del primo pixel da file e funge da *trigger* per la **FSM** col nome di **is_reading**; quindi, la logica rimane invariata rispetto alla prima versione.

Tuttavia, in questo caso i due segnali sono sfasati di 91,5 ns ovvero $30.5 * \text{clk_period}$.

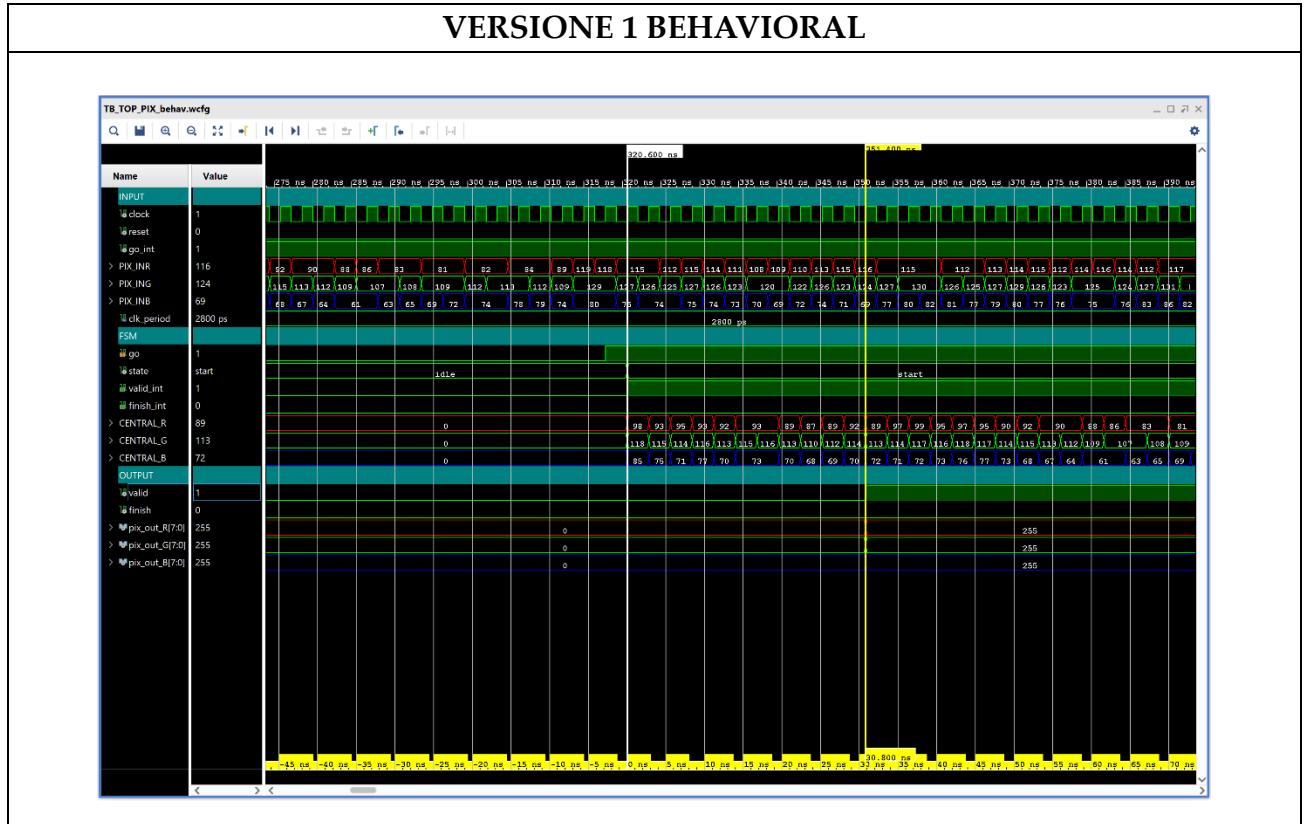
La **FSM** viene avviata con tre colpi di clock in anticipo per evitare il passaggio immediato dallo stato **go**, allo stato di **nordovest**.

Per raggiungere lo stato di **nordovest**, vengono utilizzati degli stati intermedi che permettono alla **FSM** di gestire l'inizio delle operazioni di filtraggio, senza riscontrare errori nella visualizzazione della prima finestra da filtrare nel buffer.

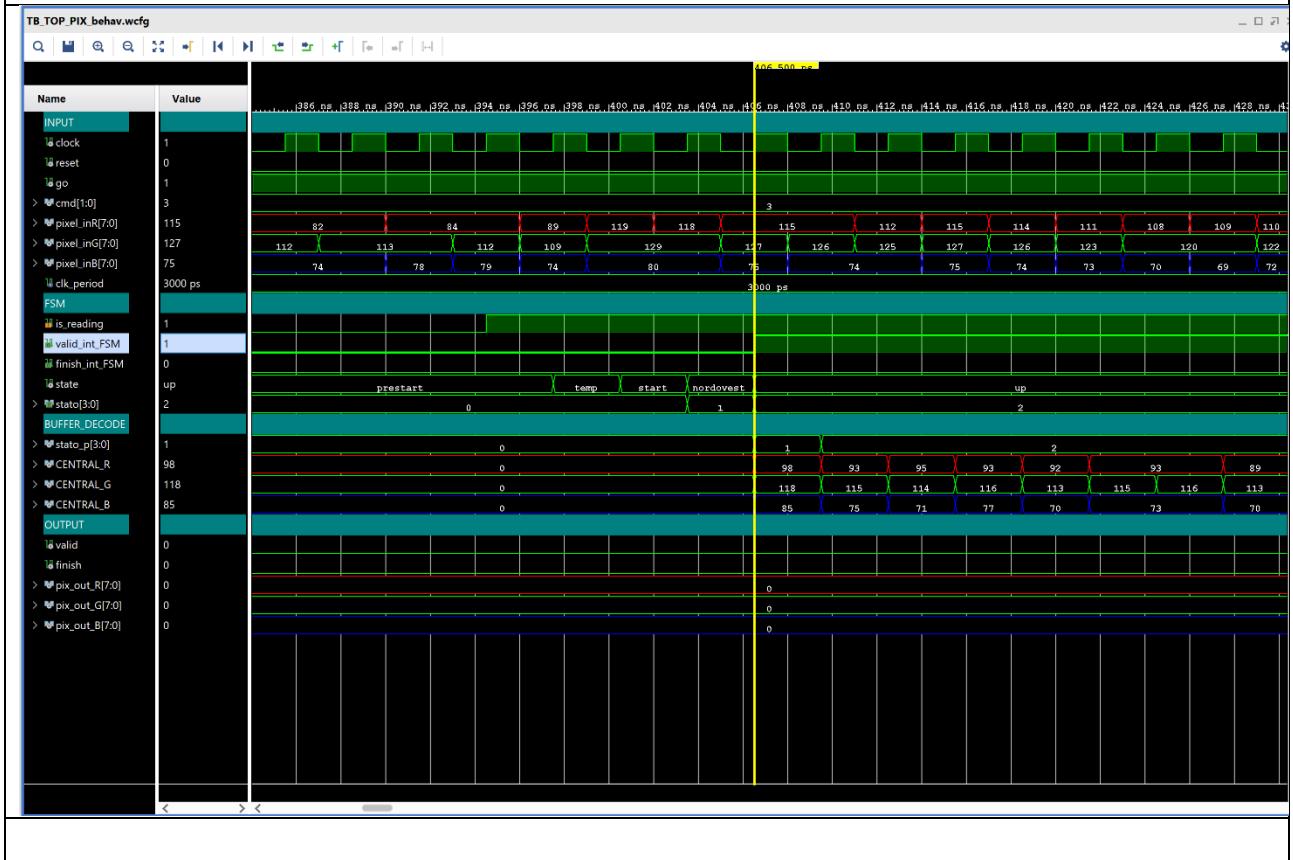
Nella terza versione il segnale di inizio lettura dei pixels da testbench è inviato da una terna di pixel aggiuntiva, settata a 0. Come ci si aspettava, la latenza tra inizio lettura a inizio filtraggio di un'immagine 64x64 è circa il doppio di quella per un'immagine 32x32.

La quarta versione include due simulazioni comportamentali: una con il blocco di conversione da immagini RGB a immagini grayscale attivo e una con il blocco di conversione disattivato. La simulazione con il blocco di conversione disattivato è analoga a quella della seconda versione. Attivando il blocco RGB2GRAY il tempo necessario per attivare il go interno alla FSM aumenta, dal momento che bisogna attendere non più $30.5 * clk_{period}$, ma bensì $40.5 * clk_{period}$, per rispettare la latenza dell'omonimo blocco di conversione.

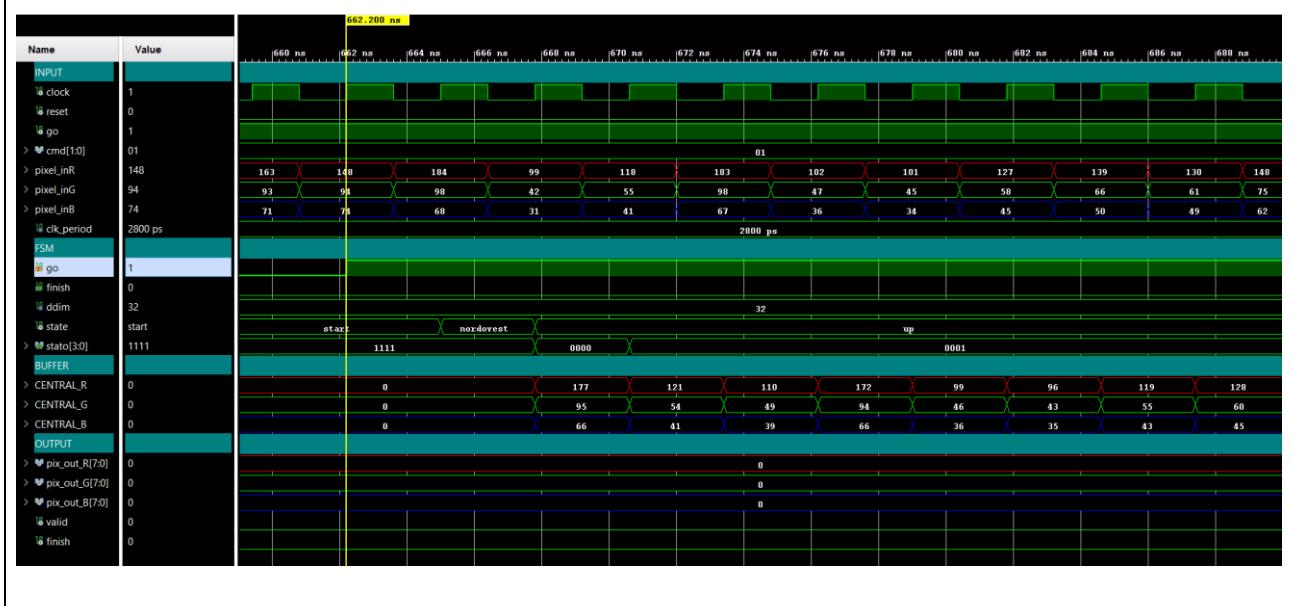
SEGNALE DI ATTIVAZIONE DELLA FSM



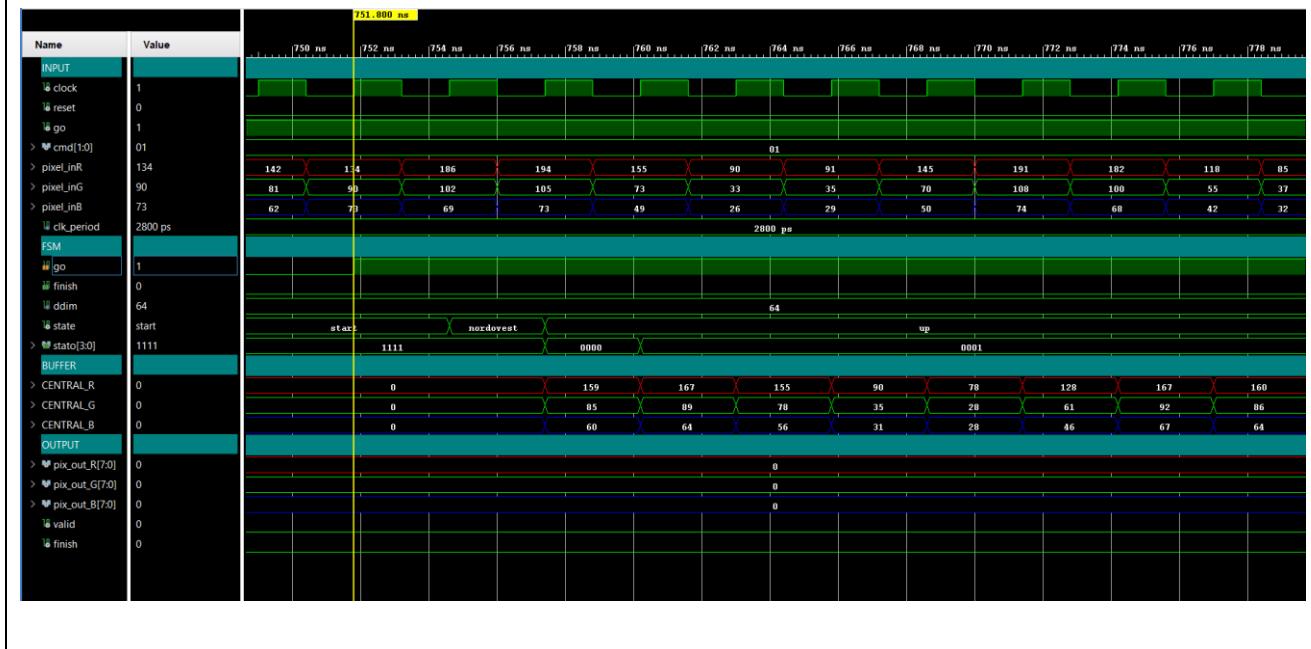
VERSIONE 2 BEHAVIORAL



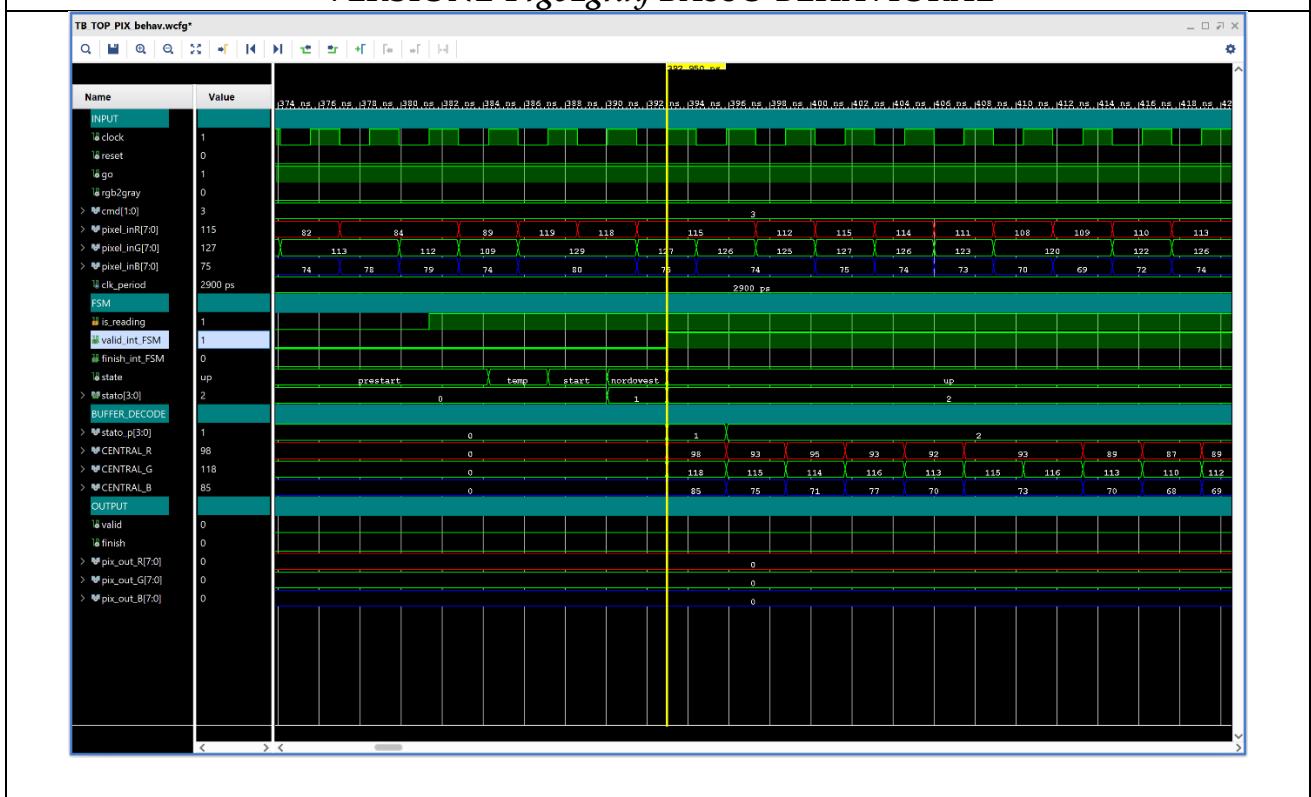
VERSIONE 3 32X32 BEHAVIORAL



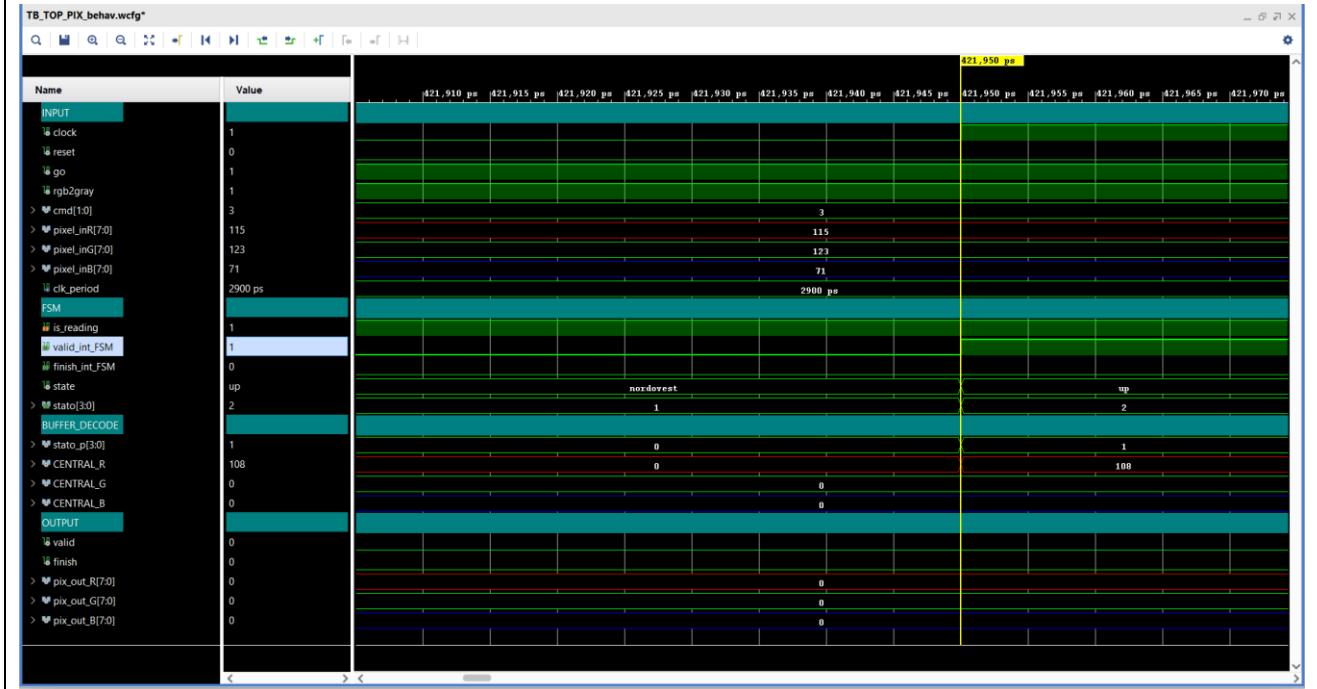
VERSIONE 3 64X64 BEHAVIORAL



VERSIONE 4 rgb2gray BASSO BEHAVIORAL



VERSIONE 4 *rgb2gray* ALTO BEHAVIORAL



- **VALID INT:** Nella prima versione il segnale di valid interno alla FSM, *valid_int*, è un segnale che si alza quando in uscita dalla FSM il primo pixel inviato da test bench si trova nella posizione centrale del buffer.

Dall'immagine è possibile osservare come il valid interno sia sincronizzato con i pixel centrali dei buffers R, G e B. La transizione al livello alto di questo segnale coincide anche col cambio di stato della **FSM**. Il *valid_int* è inserito in una **FIFO** di dimensione 11 per sincronizzare il **valid** visualizzato in uscita con i primi risultati validi del filtraggio; infatti, **passano 11 colpi di clock** dal momento in cui la prima finestra utile al filtraggio è pronta quando l'hardware produce il primo risultato valido, giustificati dagli **11 stadi di pipeline** inseriti nel circuito hardware.

Dal primo all'ultimo segnale sopra citati passano 30.8 ns, pari ad **$11 * clk_{period}$** , in questo caso **$DIM_FIFO - 1 = 11$** e non 10.

Il motivo di questi risultati è che il *valid_int* è pronto all'istante 320.6 ns ma è visibile alla **FIFO** solo al colpo di clock successivo, mentre per il caso del *go_int* esso è prodotto sul fronte di discesa del clock e quindi visto subito al primo fronte di salita.

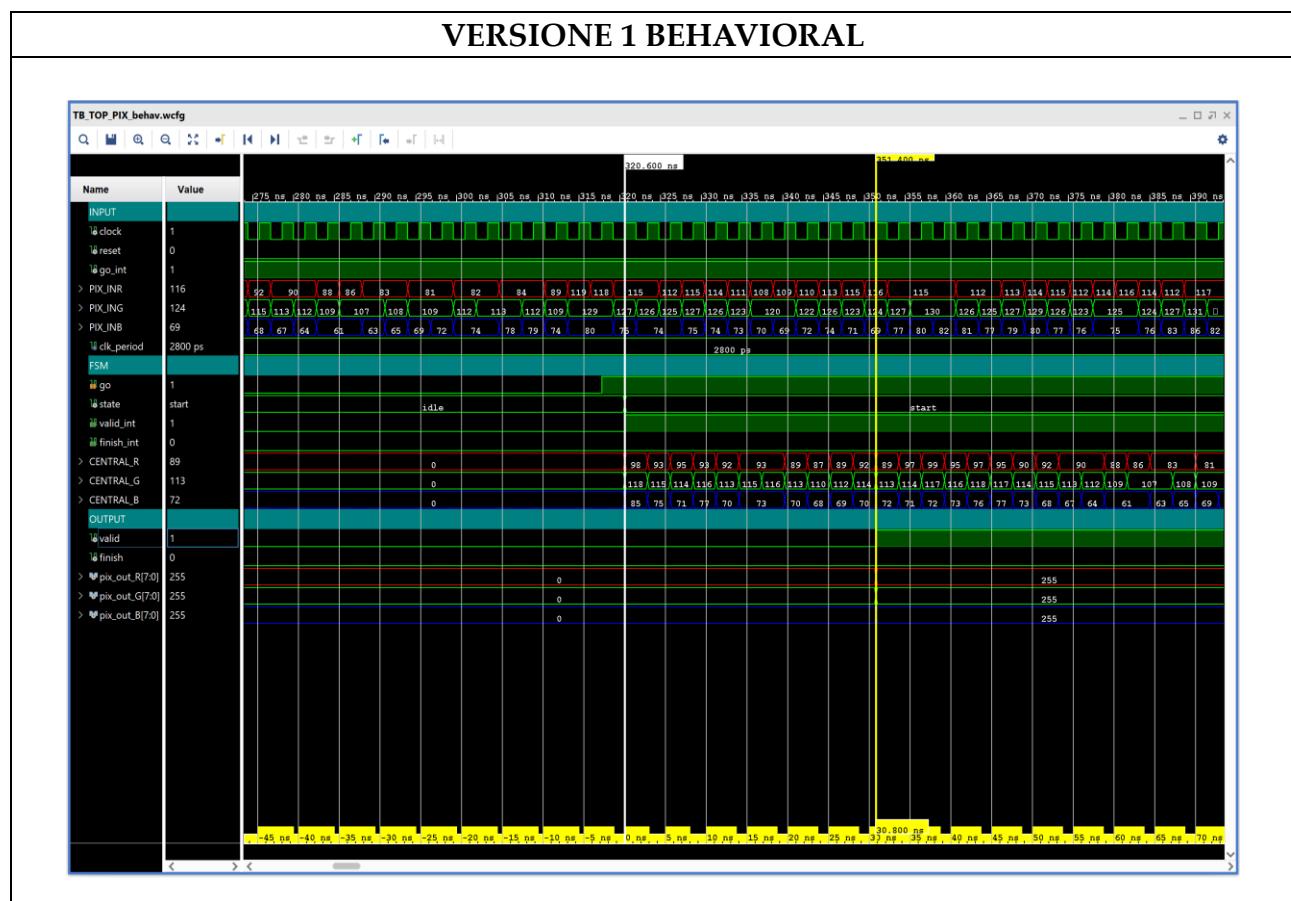
Nella seconda versione, analogamente alla prima, il segnale di valid interno dell'**FSM**, *valid_int_FSM*, passa allo stato logico alto quando il primo pixel inviato da test bench si trova nella posizione centrale del buffer.

Nella terza versione si osserva un ritardo di un colpo di clock in più rispetto alle altre, poiché il segnale di inizio operazione è effettuato leggendo da file un valore standard (posto a zero per non perturbare lo stato di reset del buffer) che precede i pixel dell'immagine.

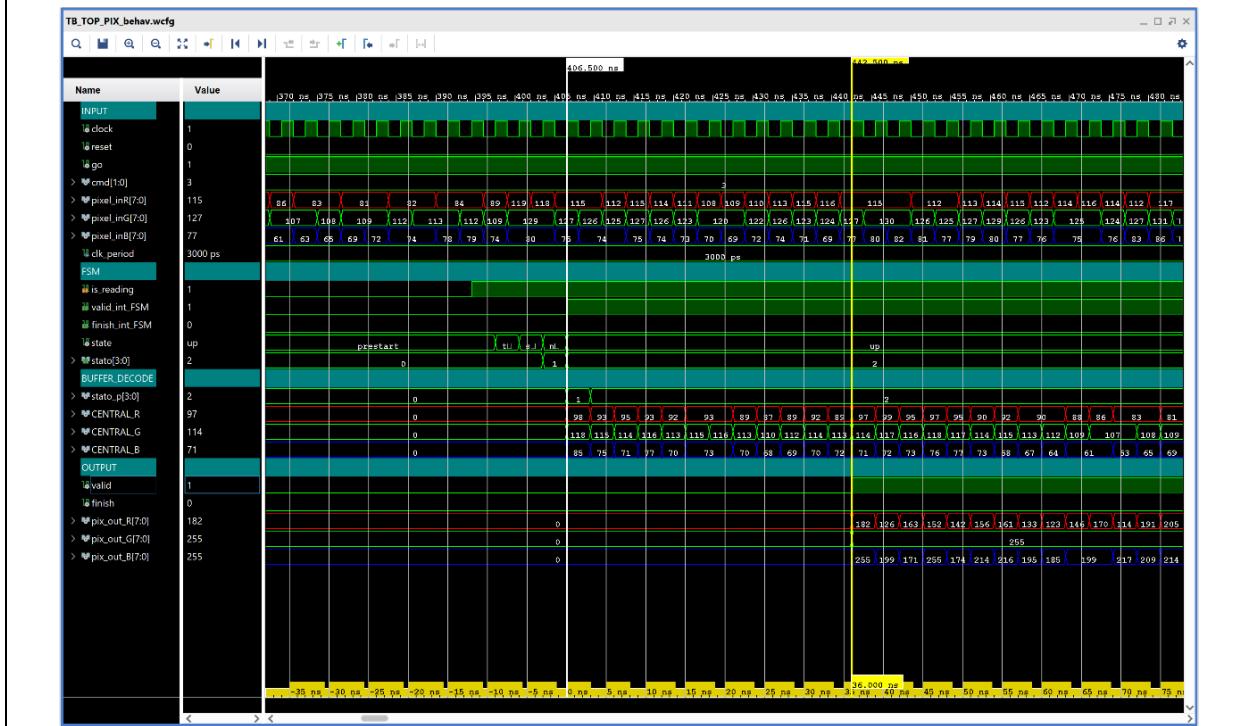
Nella quarta versione, la simulazione comportamentale con il blocco di conversione disattivato è completamente identica a quella della seconda versione descritta in precedenza.

Con il blocco di conversione attivo, si osserva che il **valid_int_FSM** è sincronizzato sia con lo stato dell'FSM, sia con il pixel in posizione centrale nel Buffer Decode.

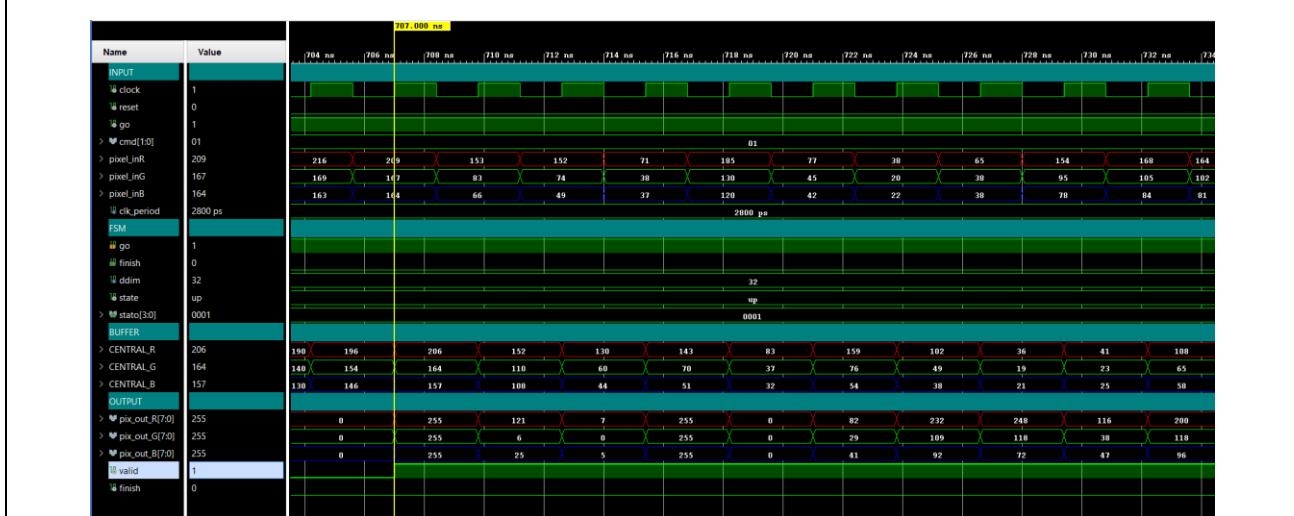
SEGNALE DI VALIDAZIONE OUTPUT



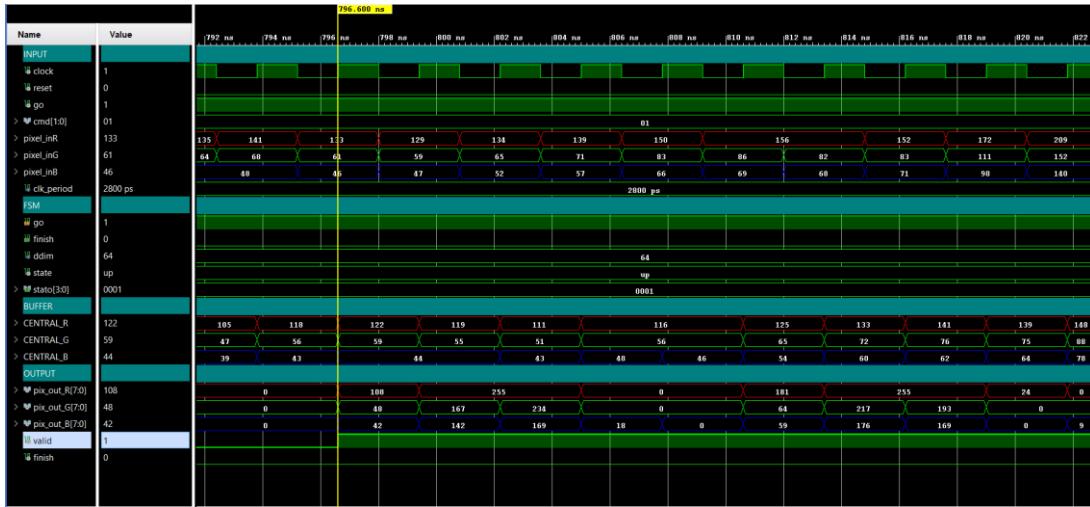
VERSIONE 2 BEHAVIORAL



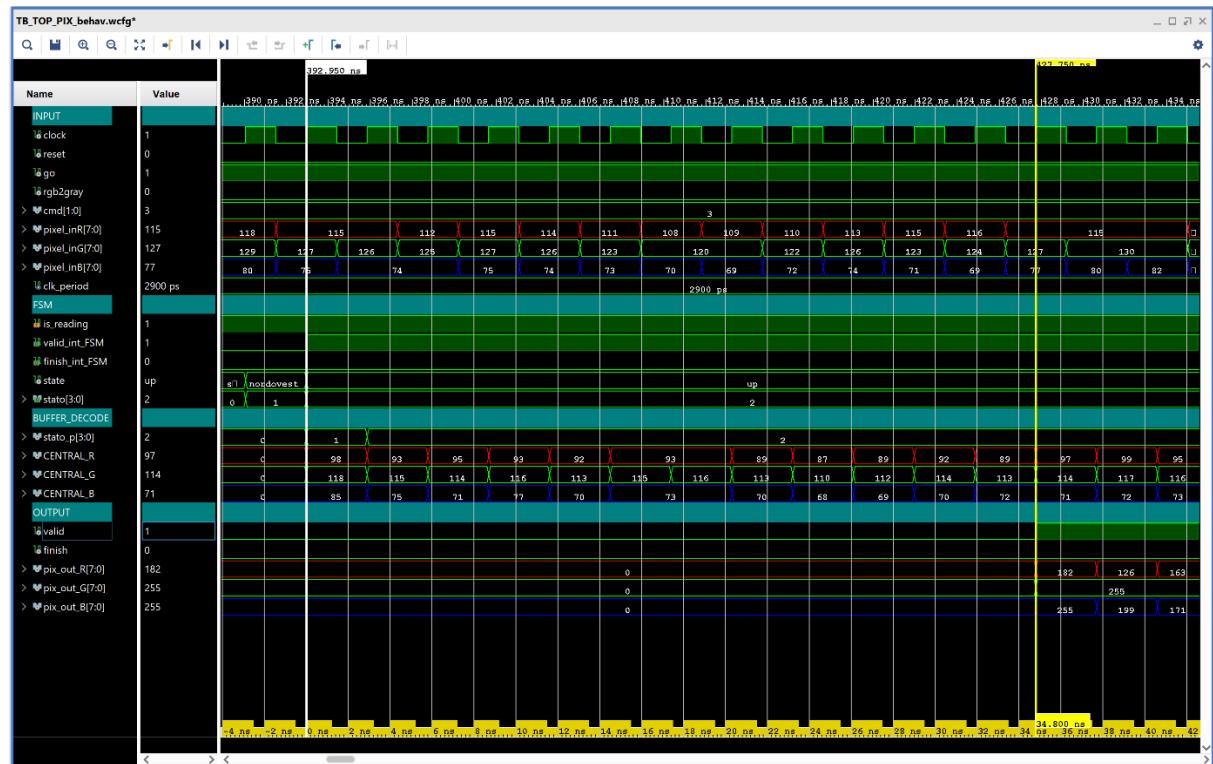
VERSIONE 3 32X32 BEHAVIORAL



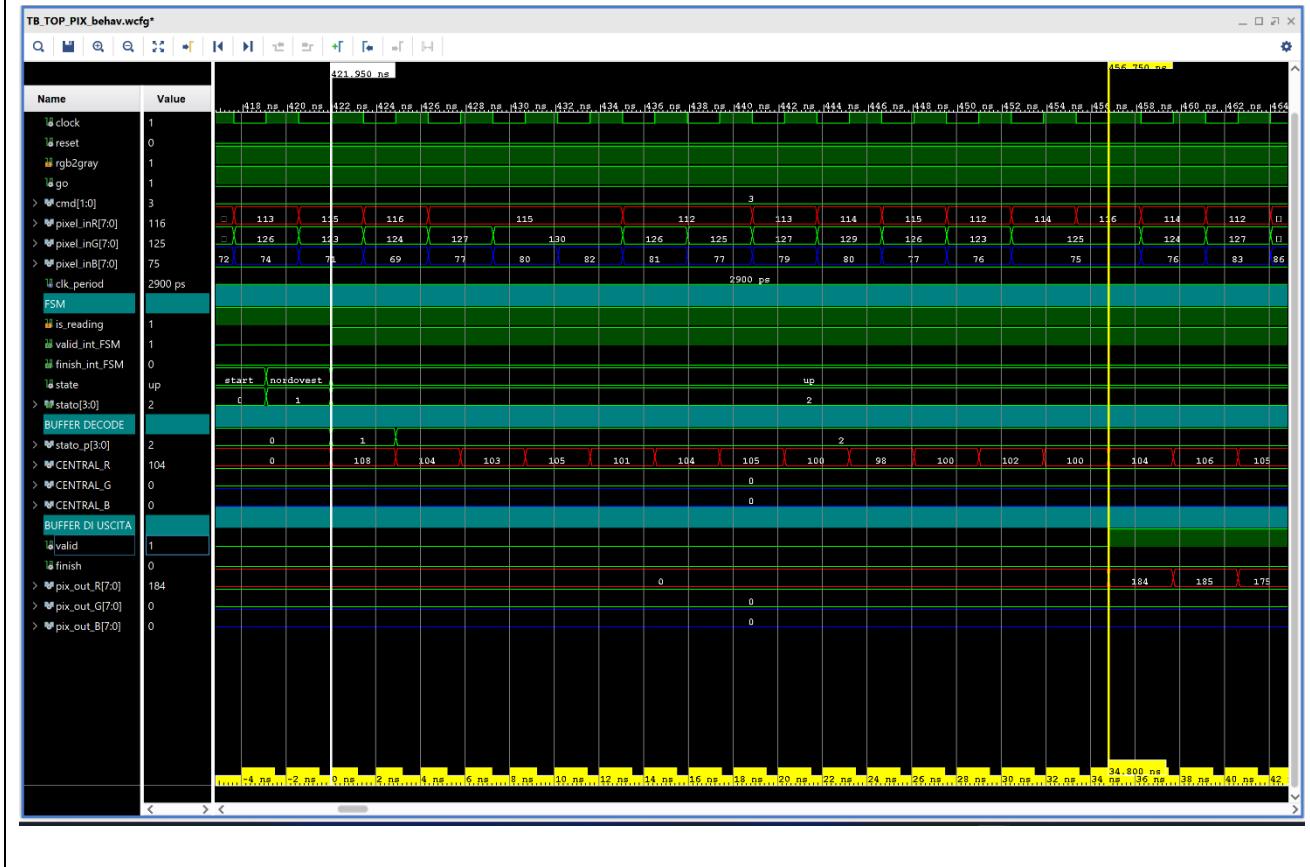
VERSIONE 3 64X64 BEHAVIORAL



VERSIONE 4 *rgb2gray* BASSO BEHAVIORAL

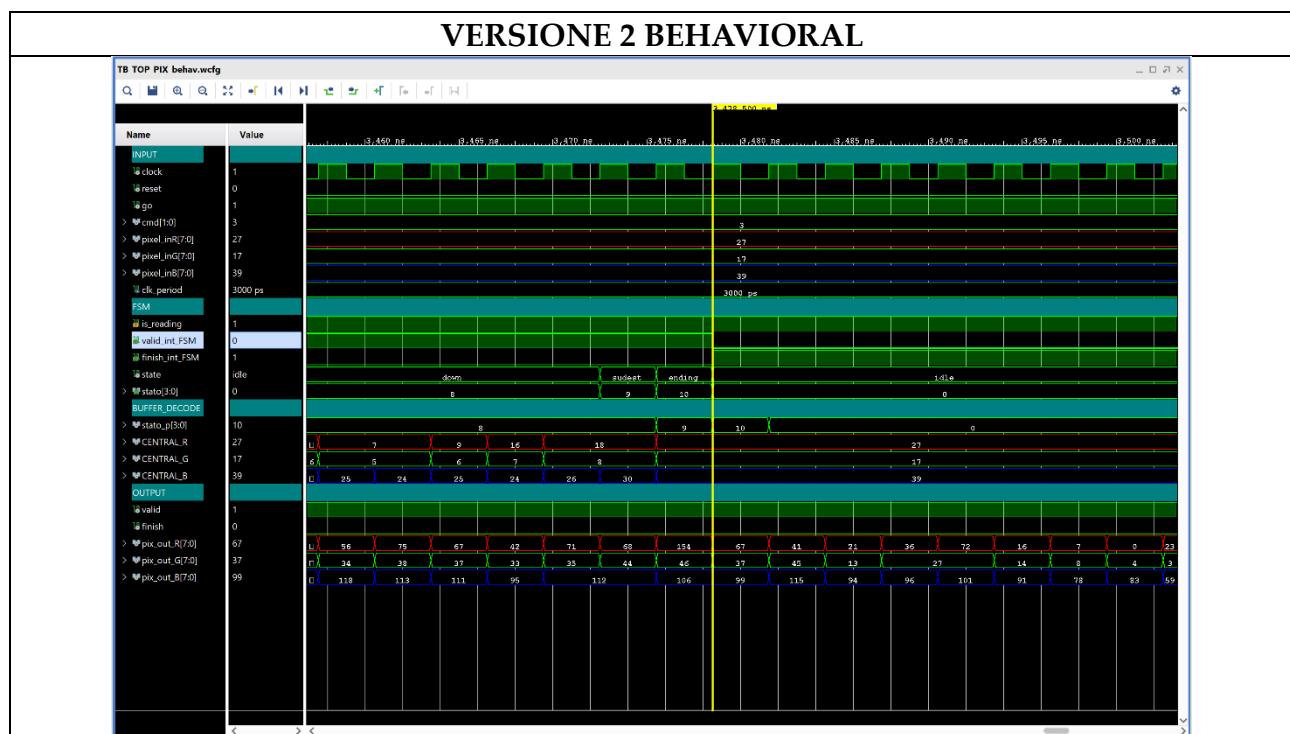
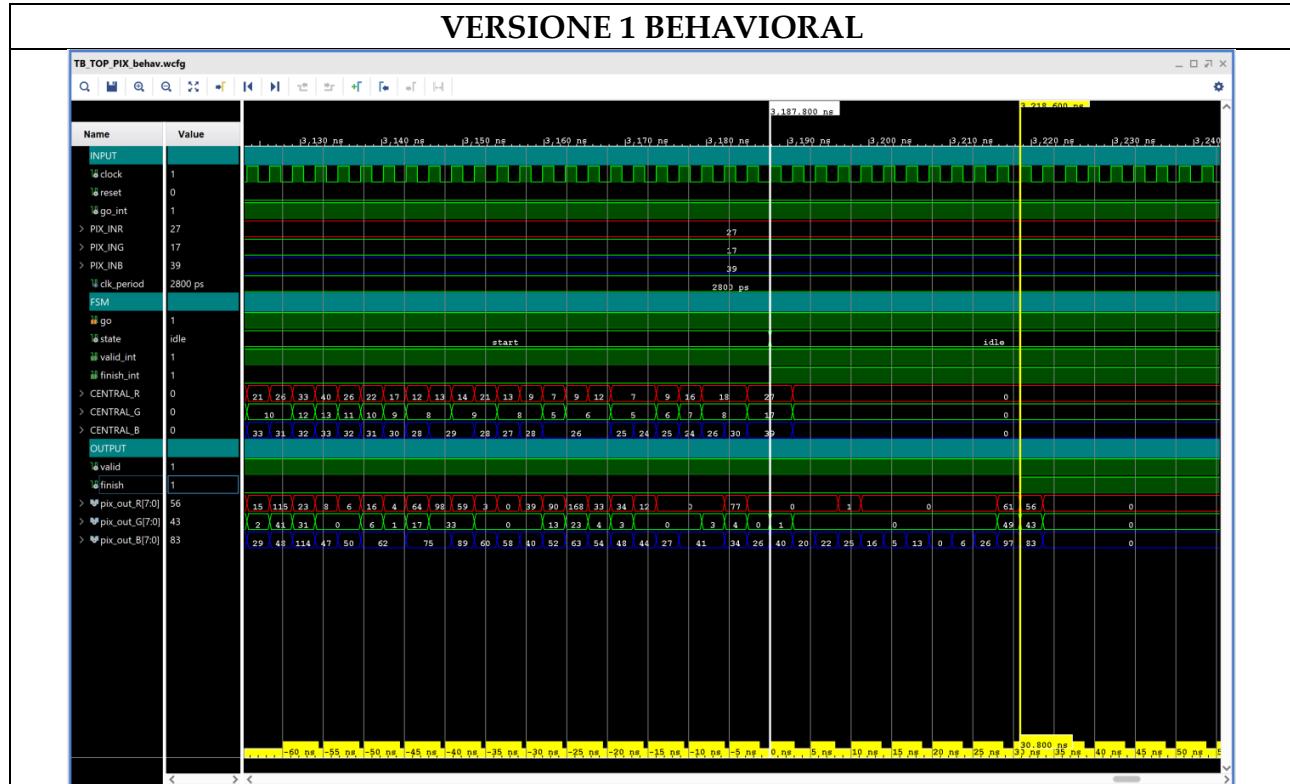


VERSIONE 4 *rgb2gray* ALTO BEHAVIORAL

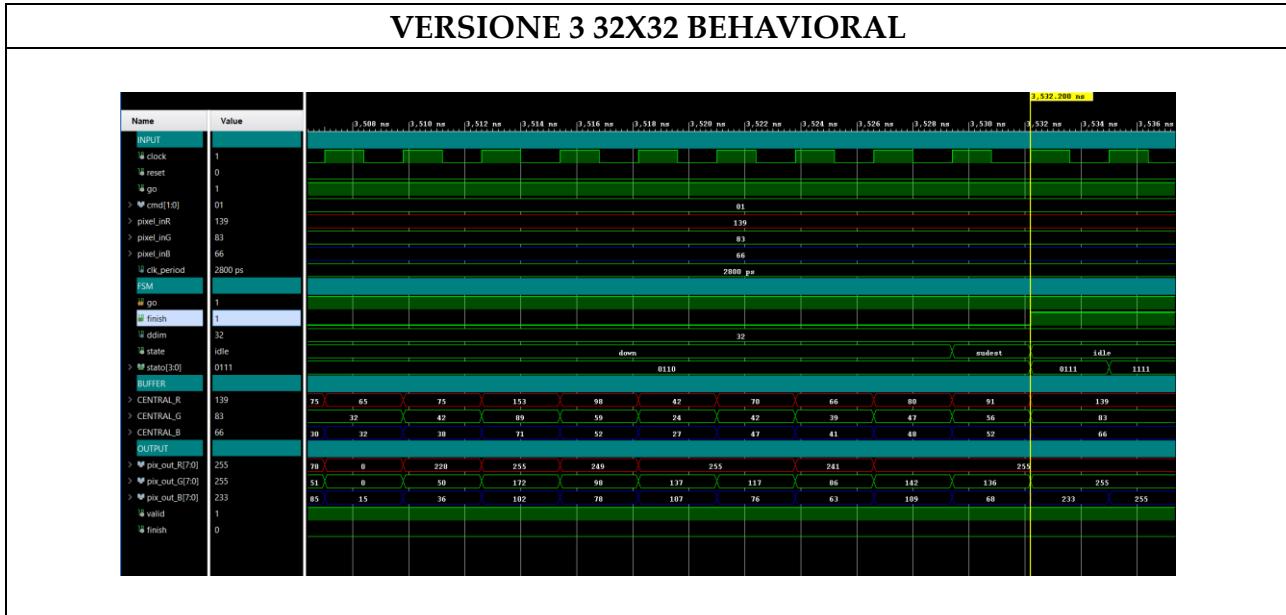


- **VALID:** Questo segnale indica l'uscita del primo pixel filtrato valido dal circuito, generato quando la **FSM** comunica quest'informazione, rispettando la latenza del blocco di filtraggio dovuta agli stadi di pipeline come descritto in precedenza.

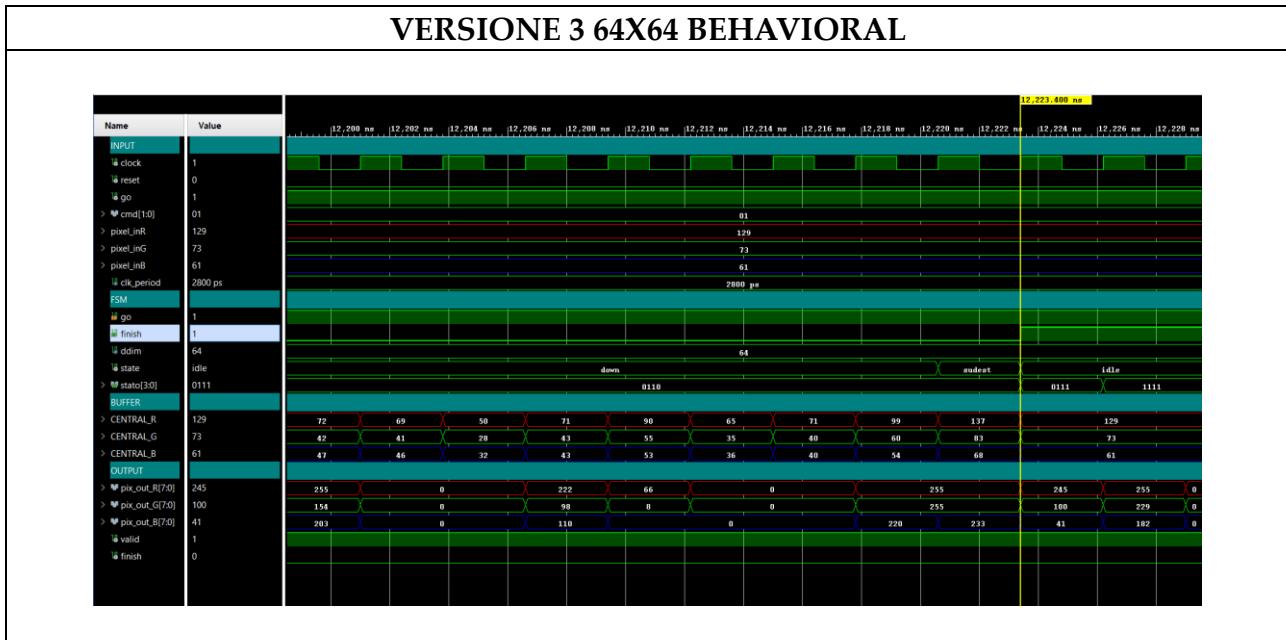
SEGNALE DI FINISH INTERNO



VERSIONE 3 32X32 BEHAVIORAL



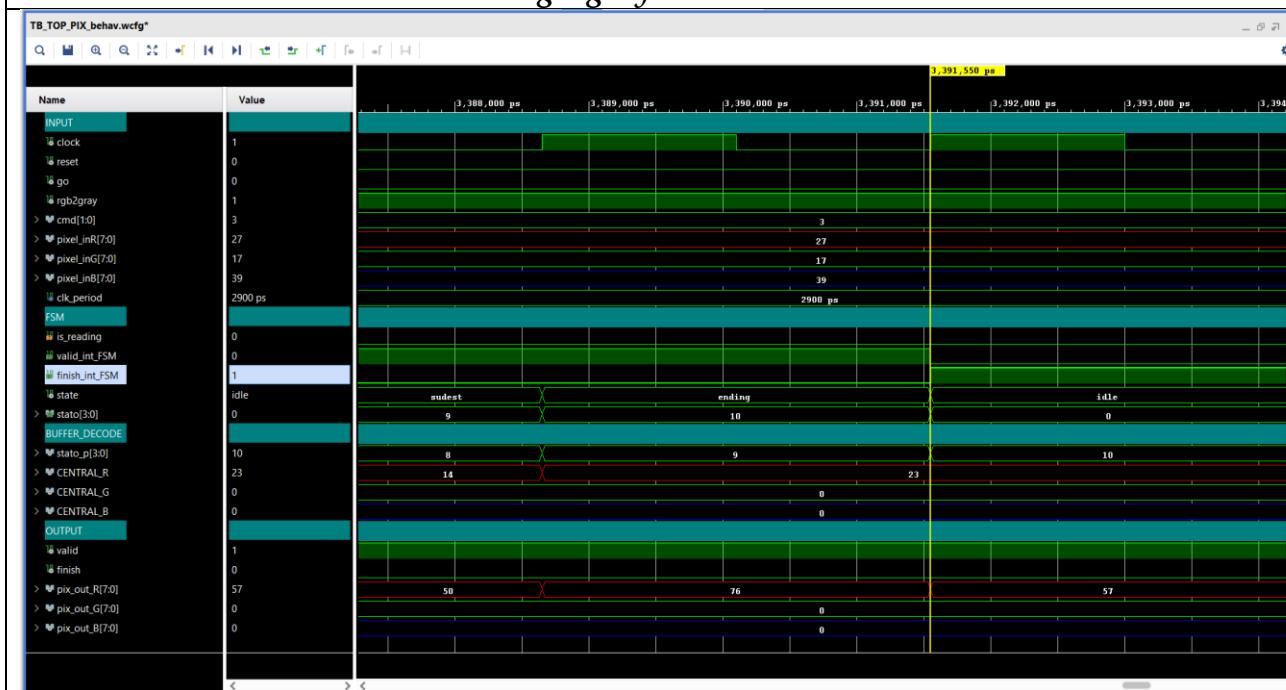
VERSIONE 3 64X64 BEHAVIORAL



VERSIONE 4 *rgb2gray* BASSO BEHAVIORAL



VERSIONE 4 *rgb2gray* ALTO BEHAVIORAL

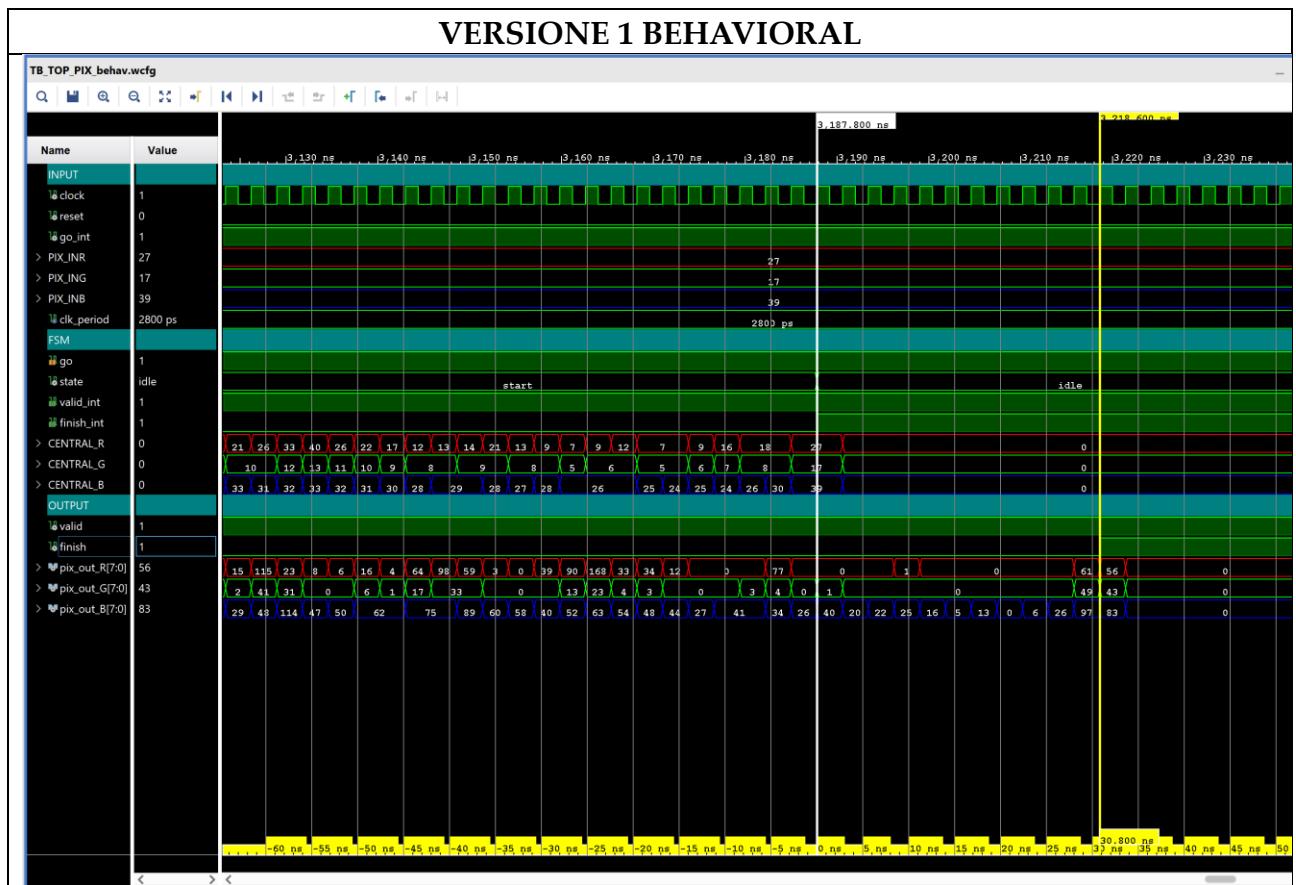


- **FINISH_FSM:** Il segnale di finish interno alla FSM, *finish_int*, è un segnale che si alza quando tutti i pixel dell'immagine sono stati letti, passando ad uno stato di idle.

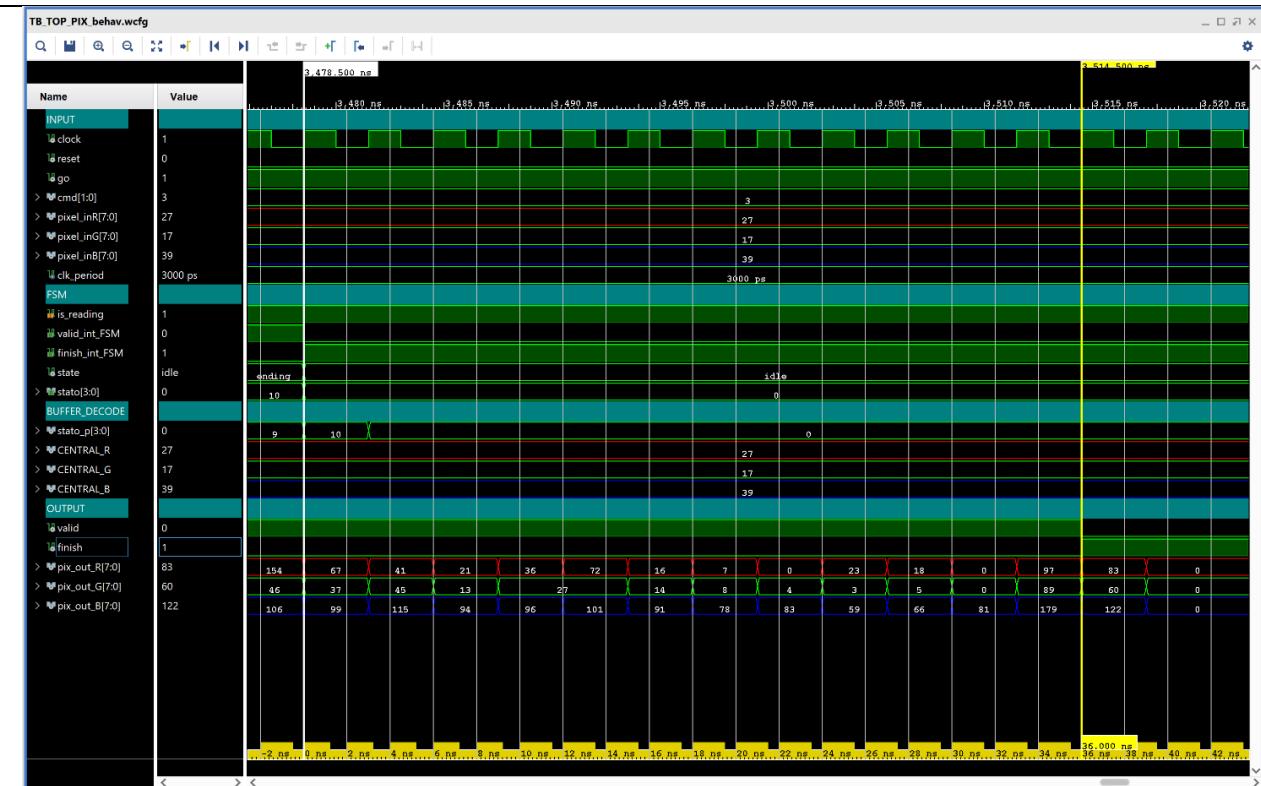
Il *finish_int* è sincronizzato con la fine della lettura dell'immagine e in tutti i circuiti viene rispettato il tempo di filtraggio secondo le strutture FIFO opportunamente inserite.

Nella prima versione questo segnale scatta esattamente una volta cambiato lo stato da start ad **idle**, nella seconda e nella quarta versione scatta quando termina lo stato di **ending** e si passa a quello di **idle**, nella terza invece si alza una volta terminato lo stato di sudest, scandendo il passaggio allo stato di **idle**.

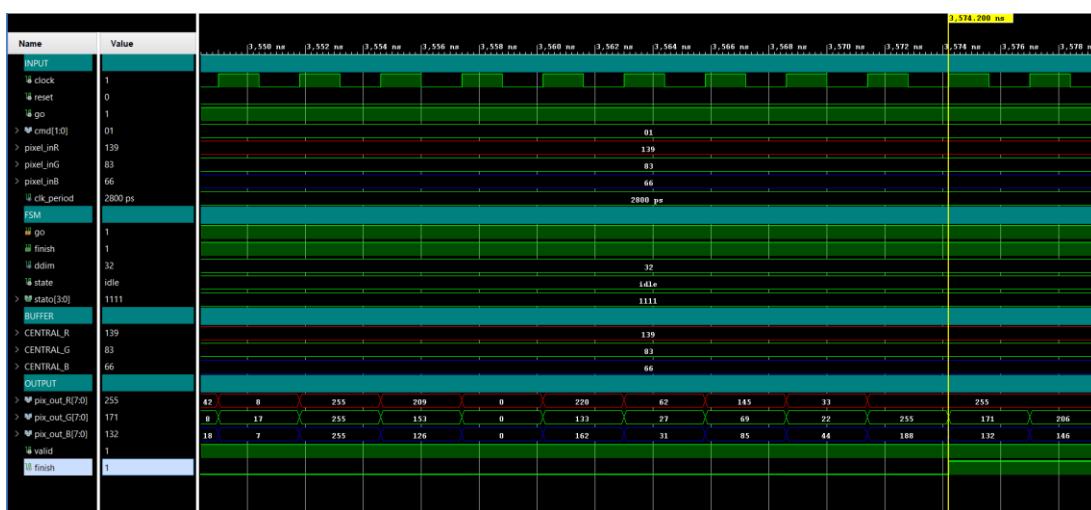
SEGNALE DI FINE FILTRAGGIO



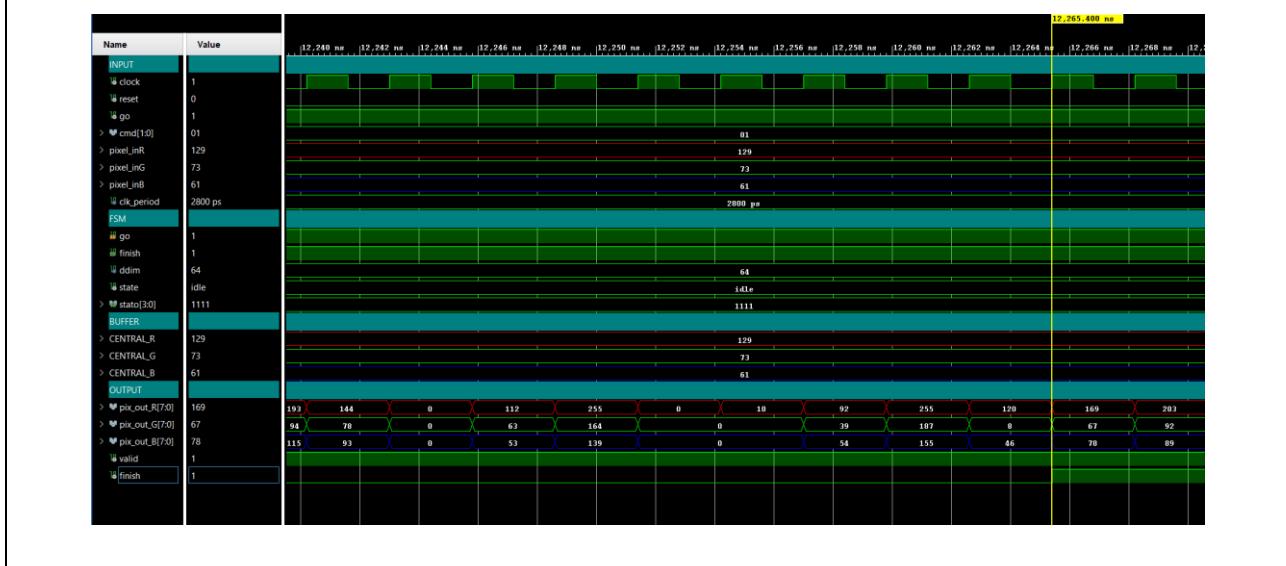
VERSIONE 2 BEHAVIORAL



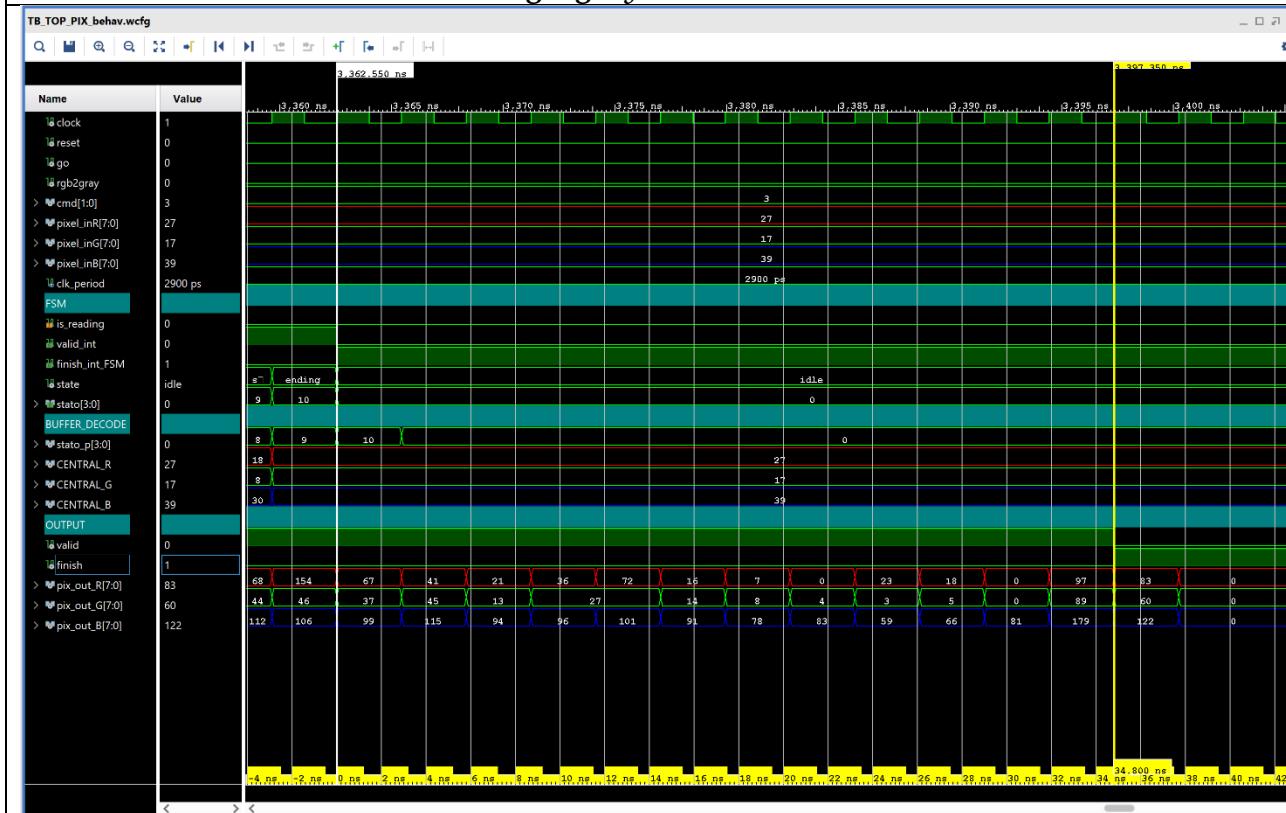
VERSIONE 3 32X32 BEHAVIORAL

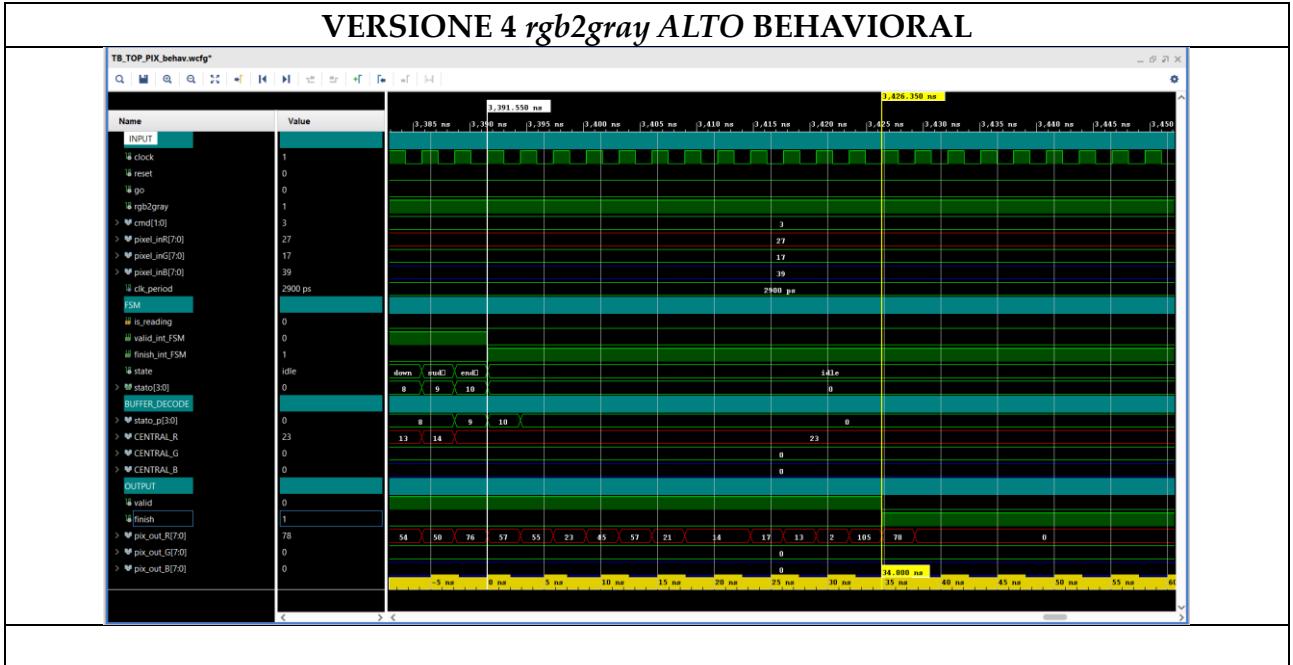


VERSIONE 3 64X64 BEHAVIORAL



VERSIONE 4 *rgb2gray* BASSO BEHAVIORAL





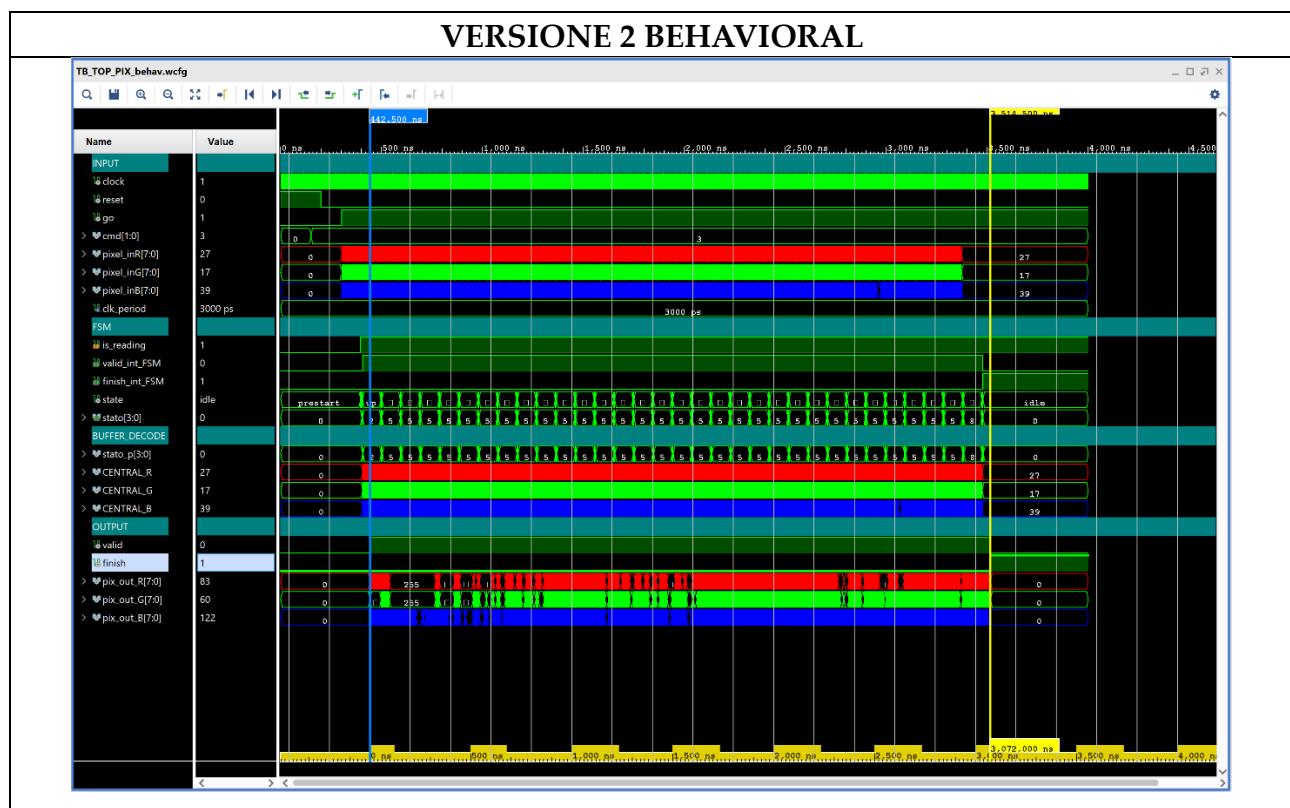
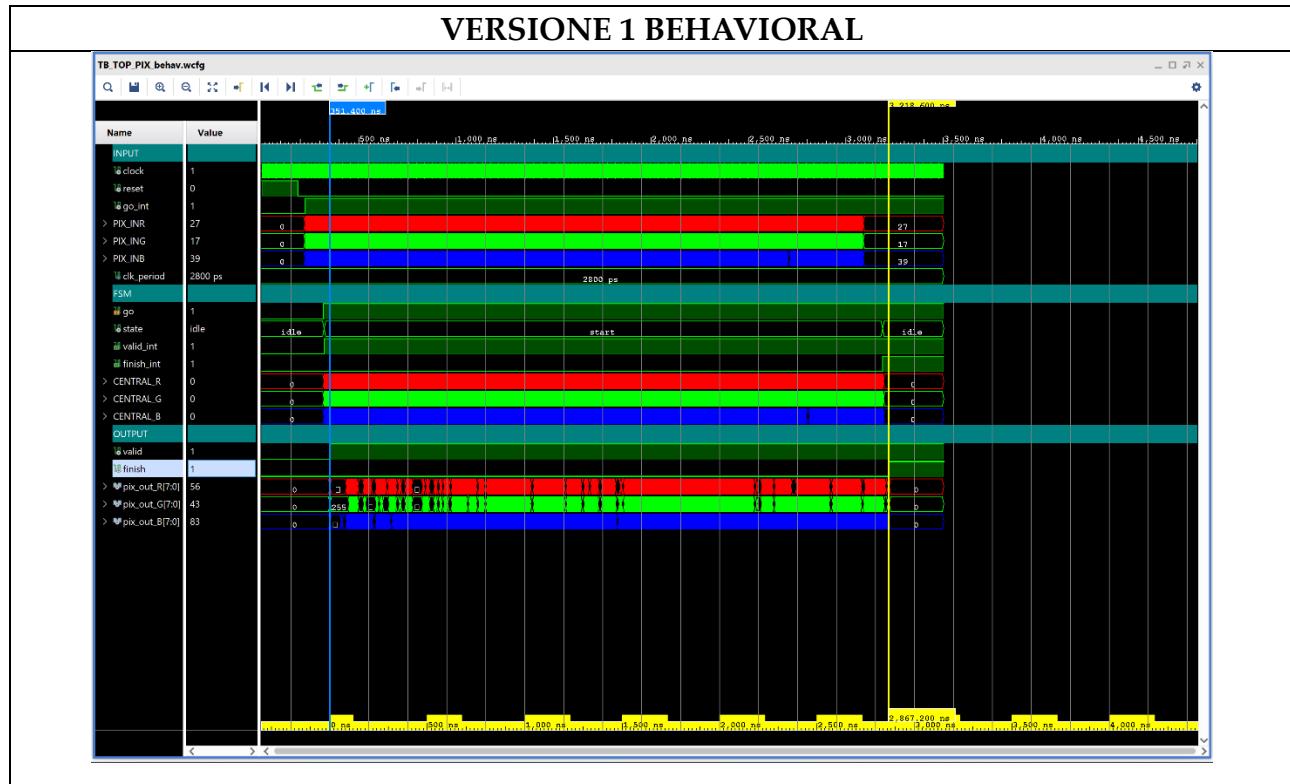
- **OUTPUT: finish**

Il segnale di *finish* in uscita sancisce il termine dell'operazione di filtraggio dell'immagine.

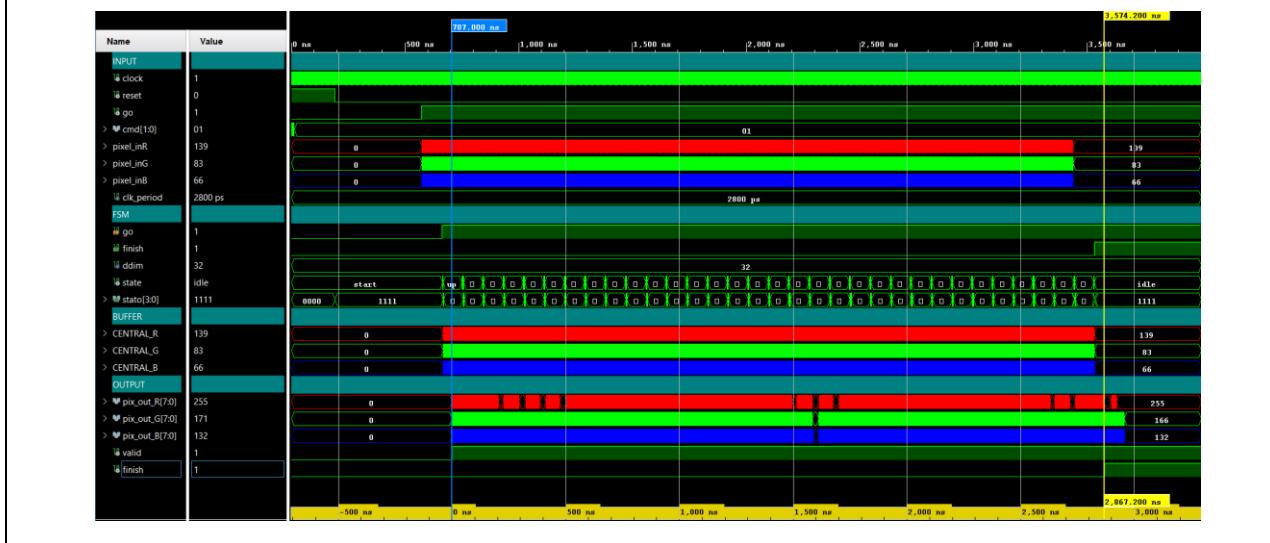
La distanza temporale tra *finish_int* e *finish* dipende dalle FIFO in cui vanno in ingresso dimensionate appositamente in funzione delle differenti latenze dei circuiti. Va sottolineato il fatto che l'ultima terna di uscita visibile nella V1 (56,43,83) si trova in corrispondenza del *finish* con valore logico alto, per cui quella terna non è valida.

Viene scelto appositamente di non settare i pixel a zero nello stato di **ending** in quanto tale scelta avrebbe comportato un peggioramento delle prestazioni, in particolare del WNS.

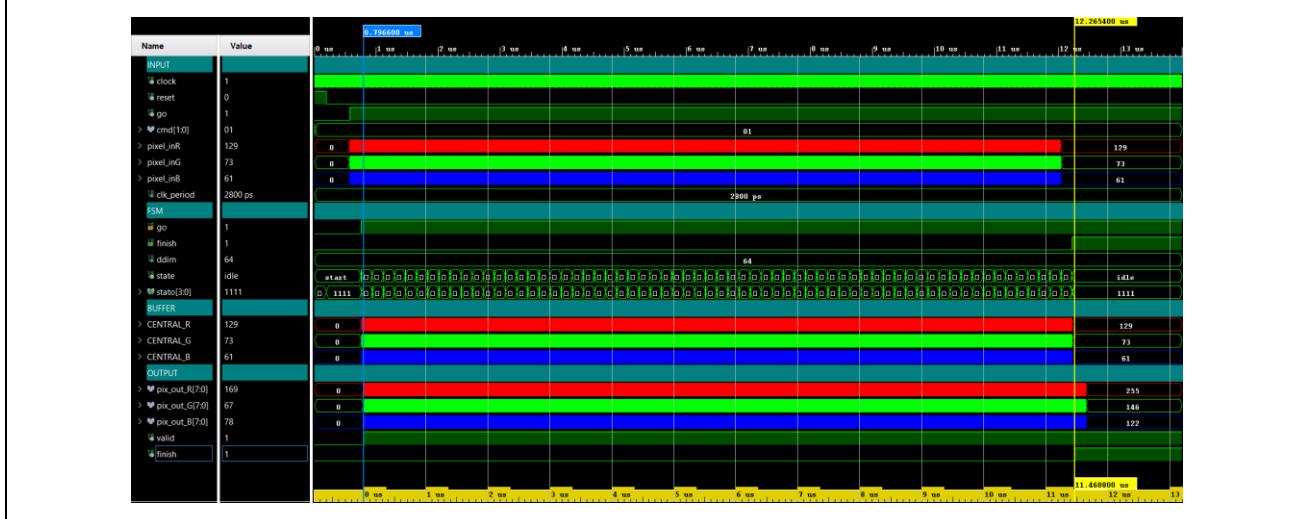
TEMPO DI FILTRAGGIO



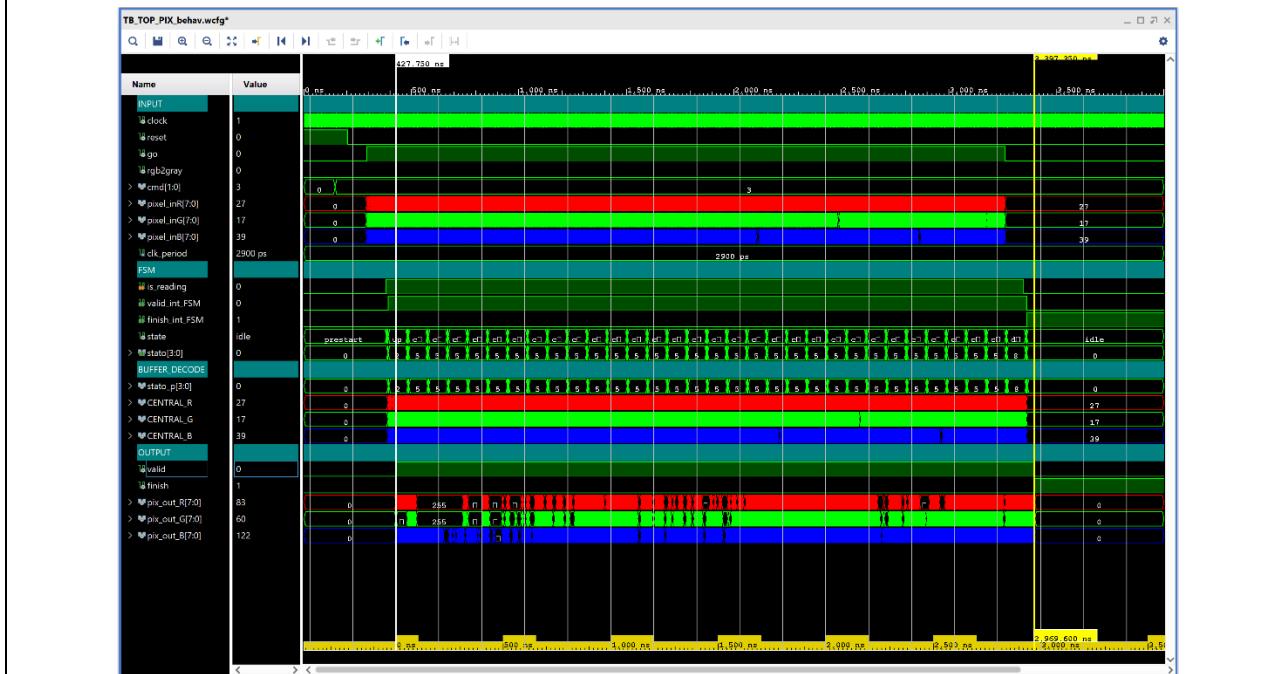
VERSIONE 3 32X32 BEHAVIORAL



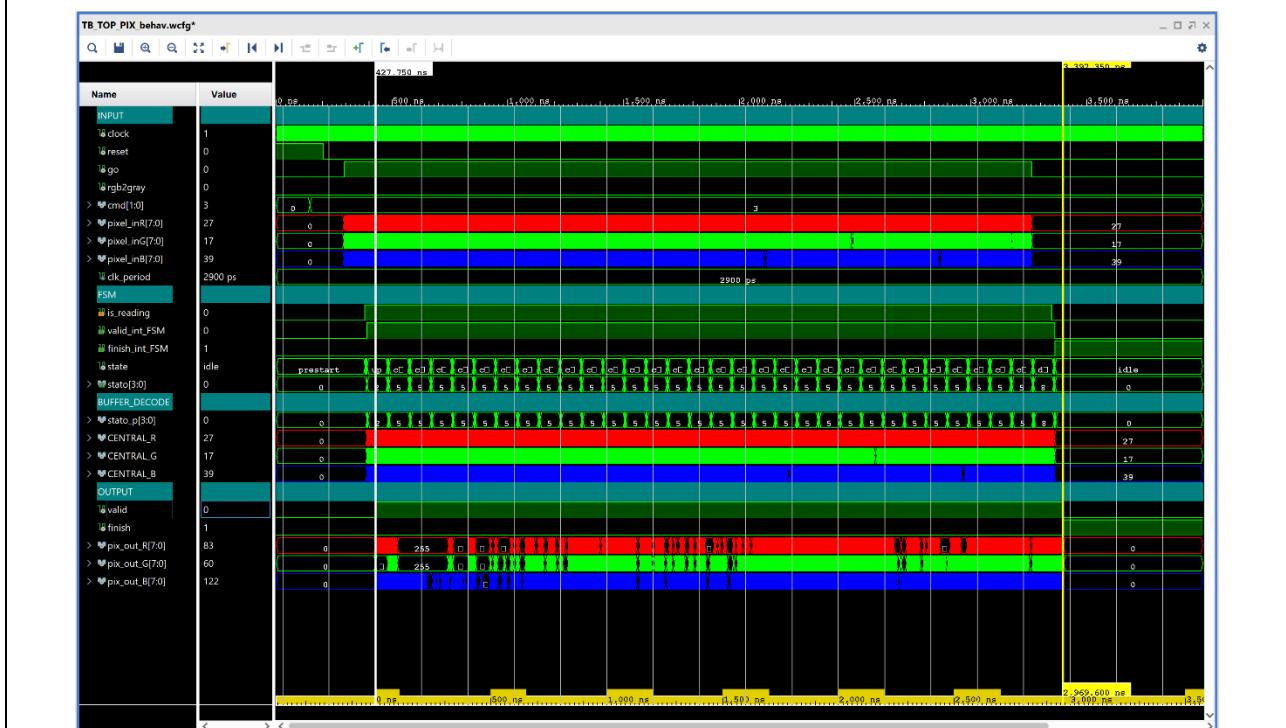
VERSIONE 3 64X64 BEHAVIORAL



VERSIONE 4 *rgb2gray* BASSO BEHAVIORAL



VERSIONE 4 *rgb2gray* ALTO BEHAVIORAL



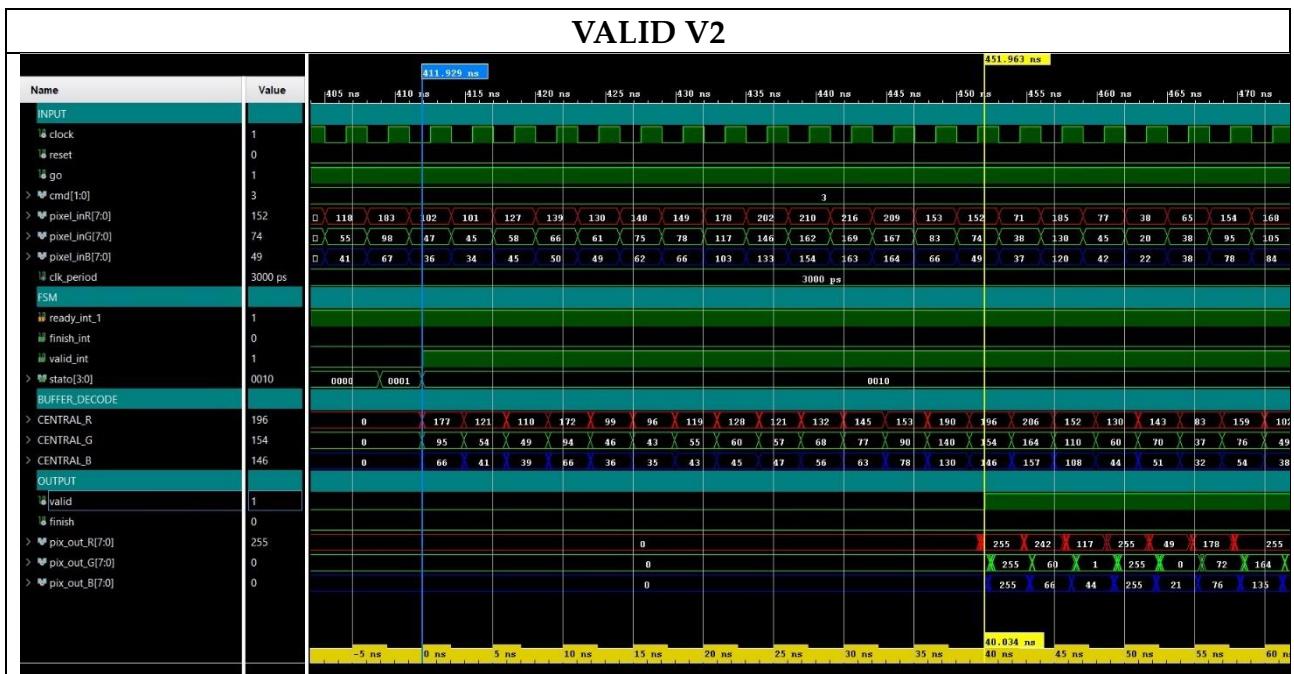
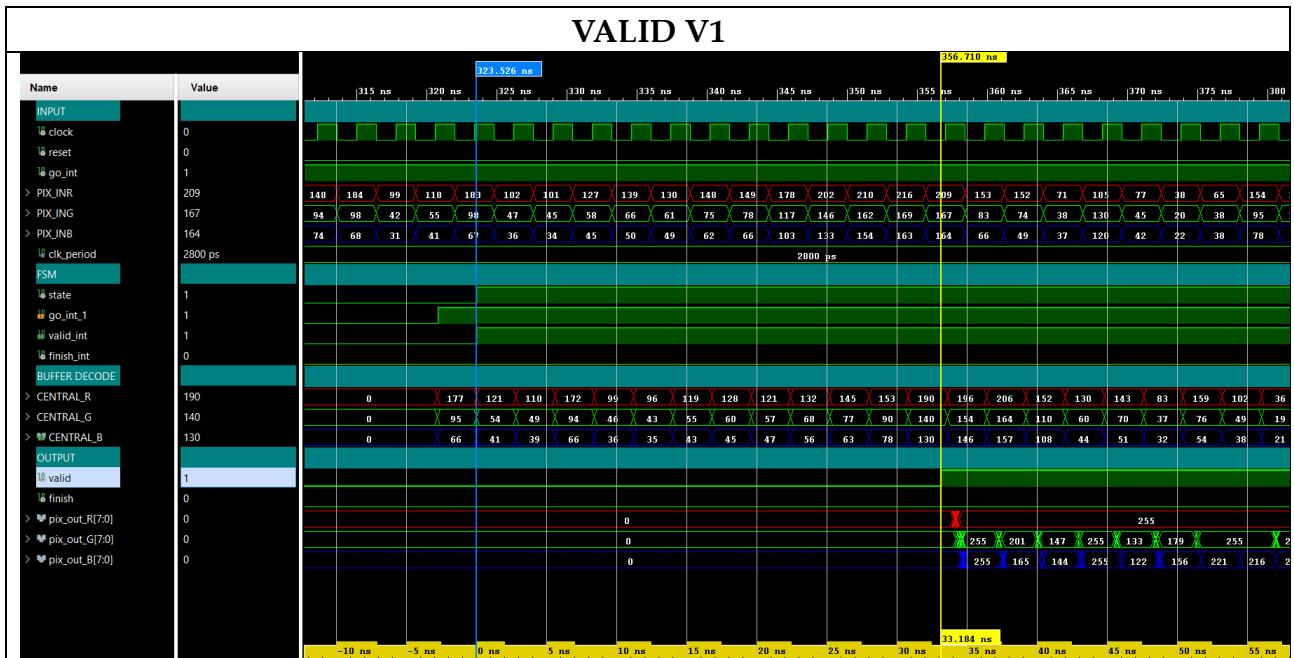
- **TEMPO DI FILTRAGGIO:** questo parametro è ricavato dalla differenza temporale tra gli istanti in cui il *valid* e il *finish* assumono il valore logico alto. Ci si aspetta che questo valore sia pari a **1024*clk_period**.

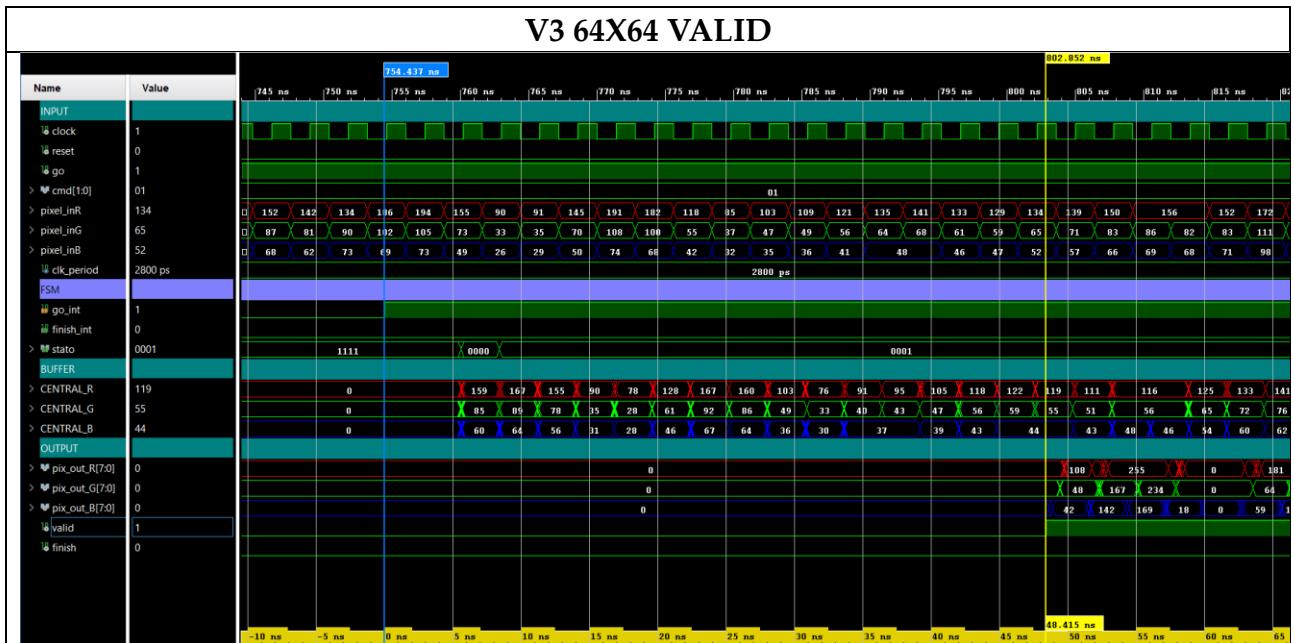
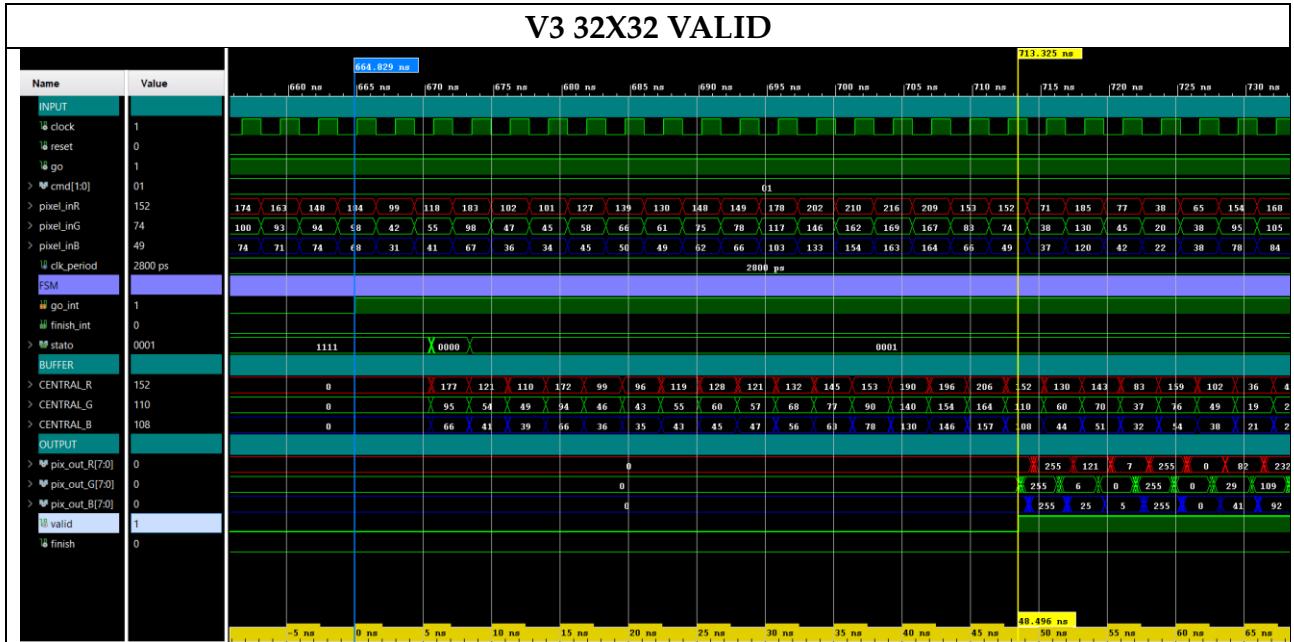
Il tempo di esecuzione risulta essere proprio **2867.2 ns**, esattamente **1024*clk_period**: questo dato stabilisce la correttezza di **valid** e **finish**.

Questo parametro dipende strettamente dalla frequenza di clock utilizzata per ogni versione, si può concludere che le versioni che impiegano meno tempo a filtrare sono la **Versione 1** e la **Versione 3** con immagine **32x32**.

SIMULAZIONI POST IMPLEMENTATION TIMING

LATENZA TRA INPUT E OUTPUT





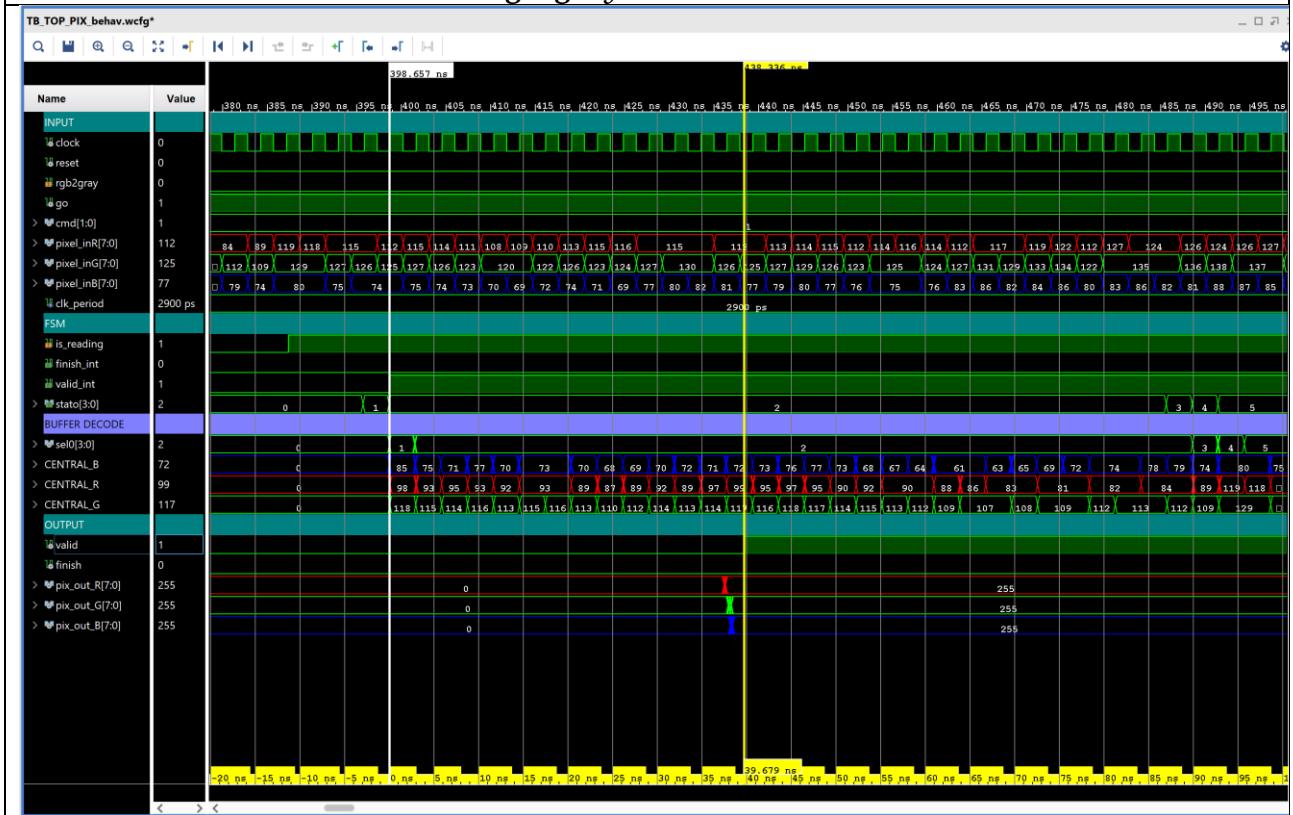
V4 VALID *rgb2gray ALTO*



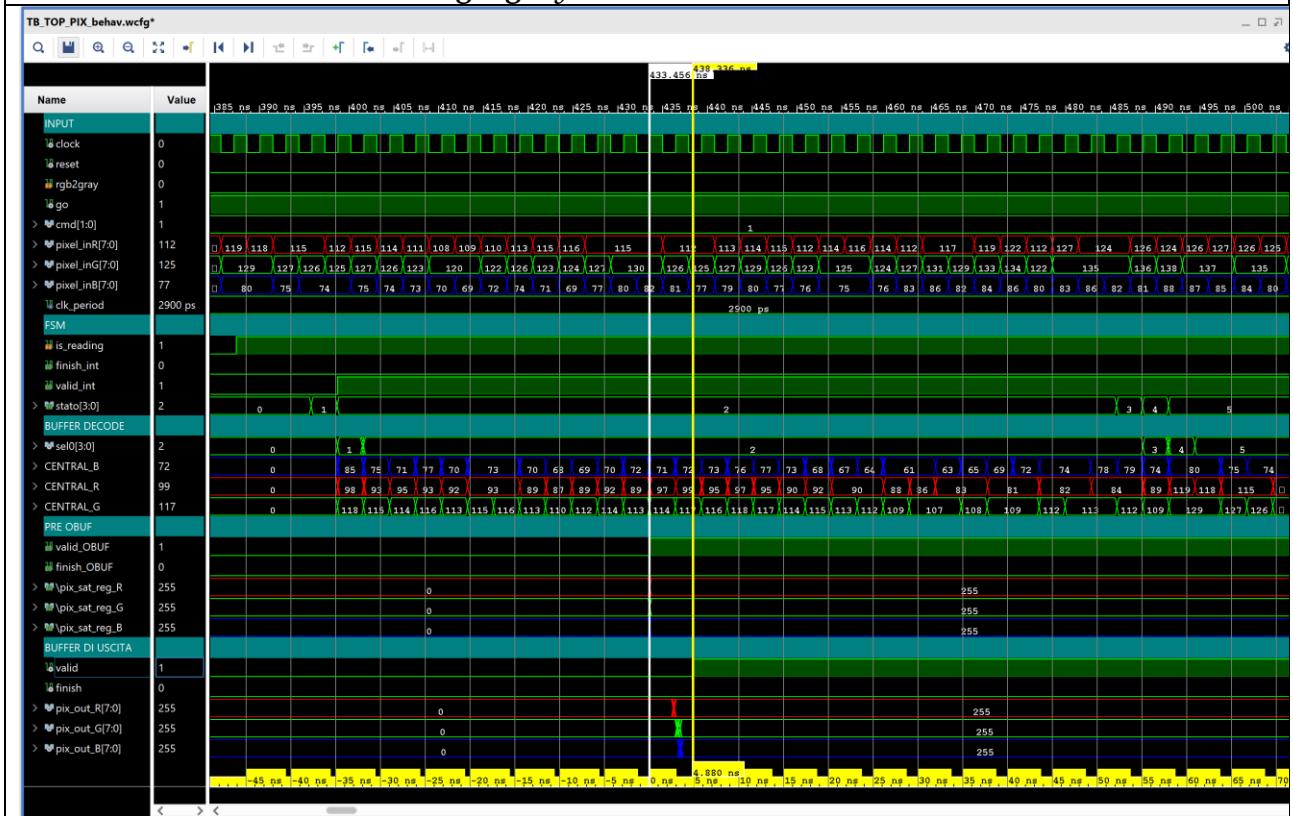
V4 *rgb2gray ALTO VALID OBUF*



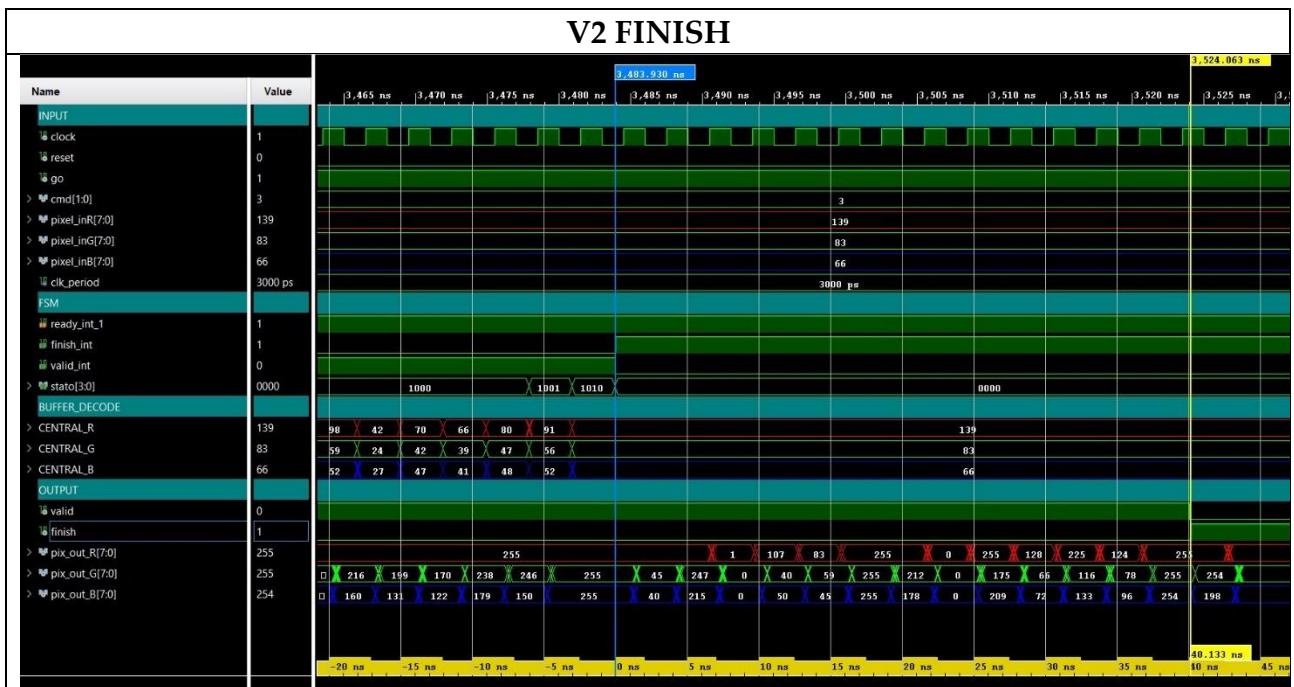
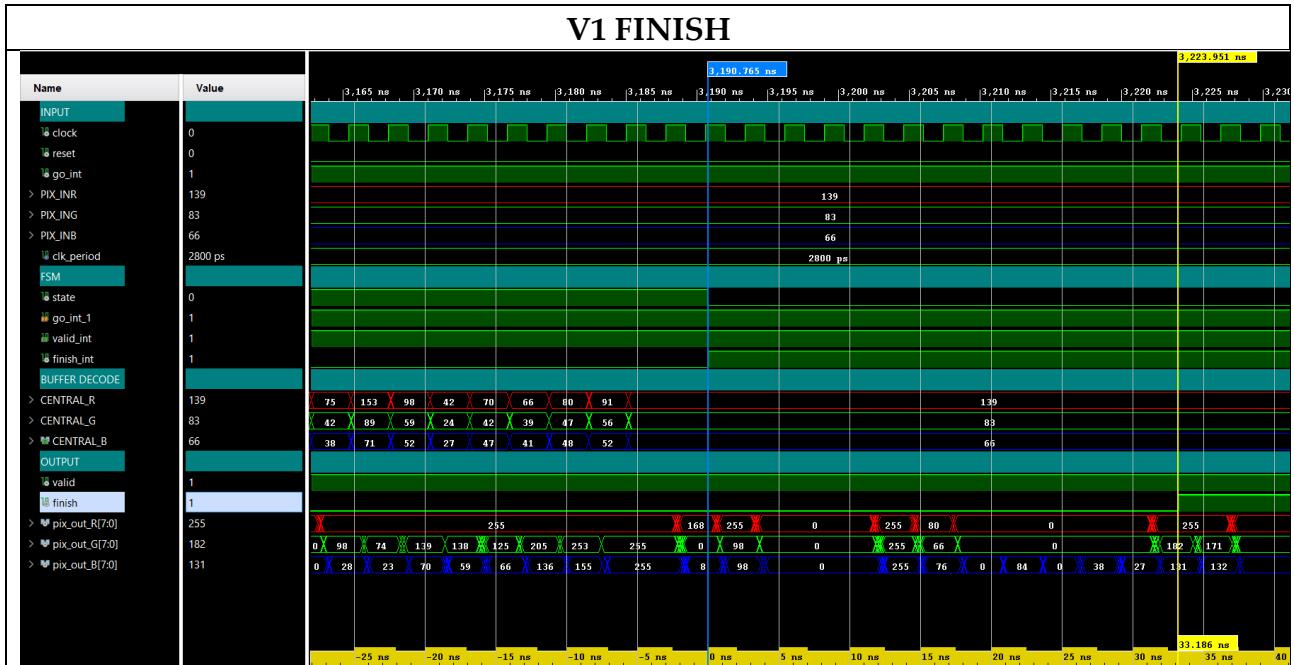
V4 *rgb2gray* BASSO VALID

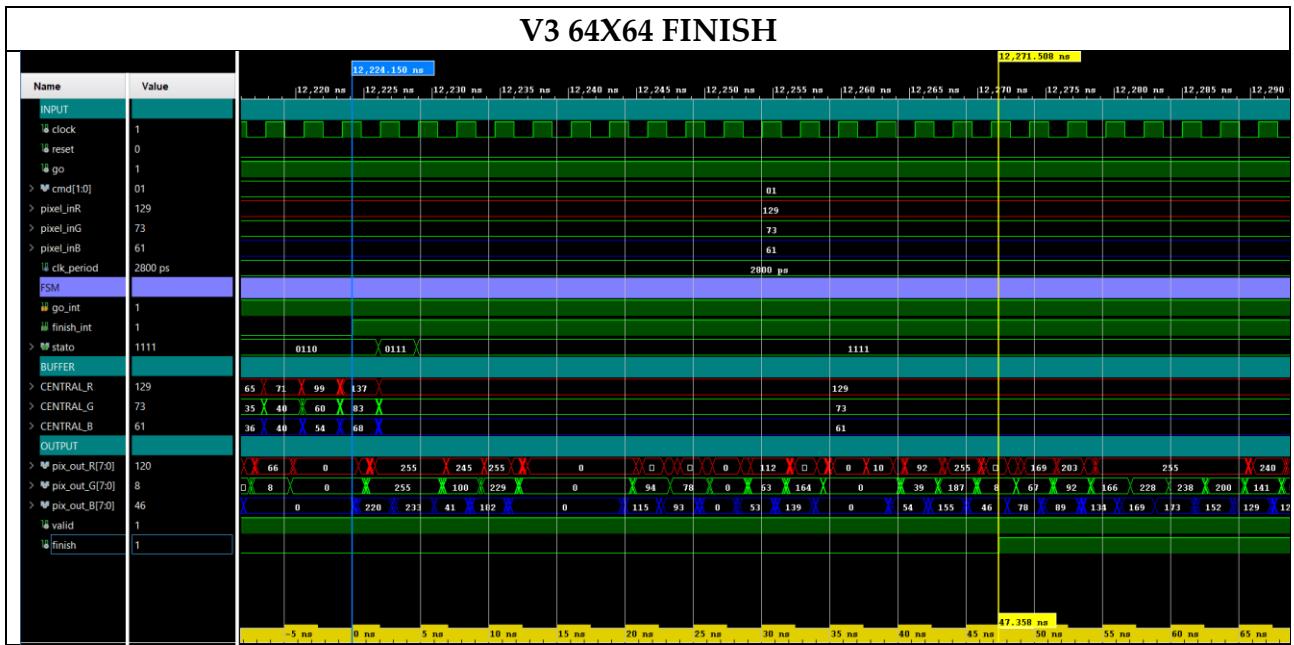
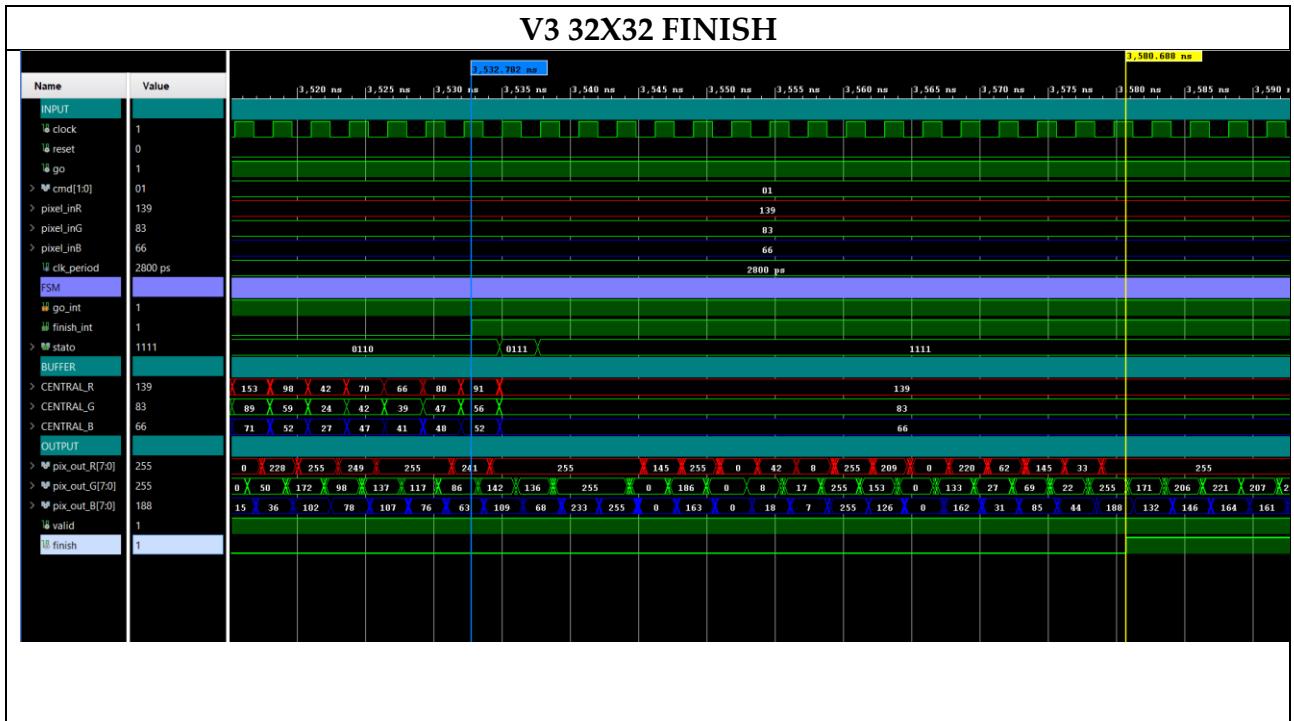


V4 *rgb2gray* BASSO VALID OBUF

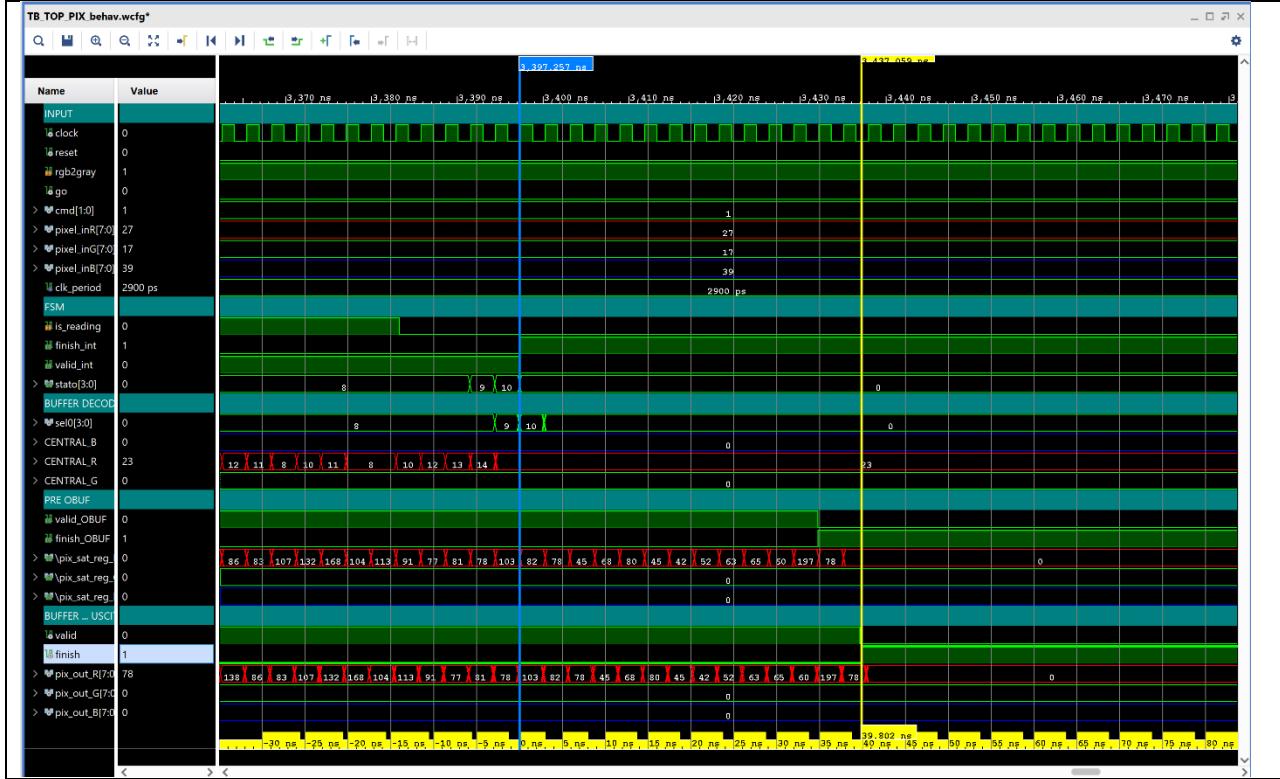


SEGNALI DI FINISH

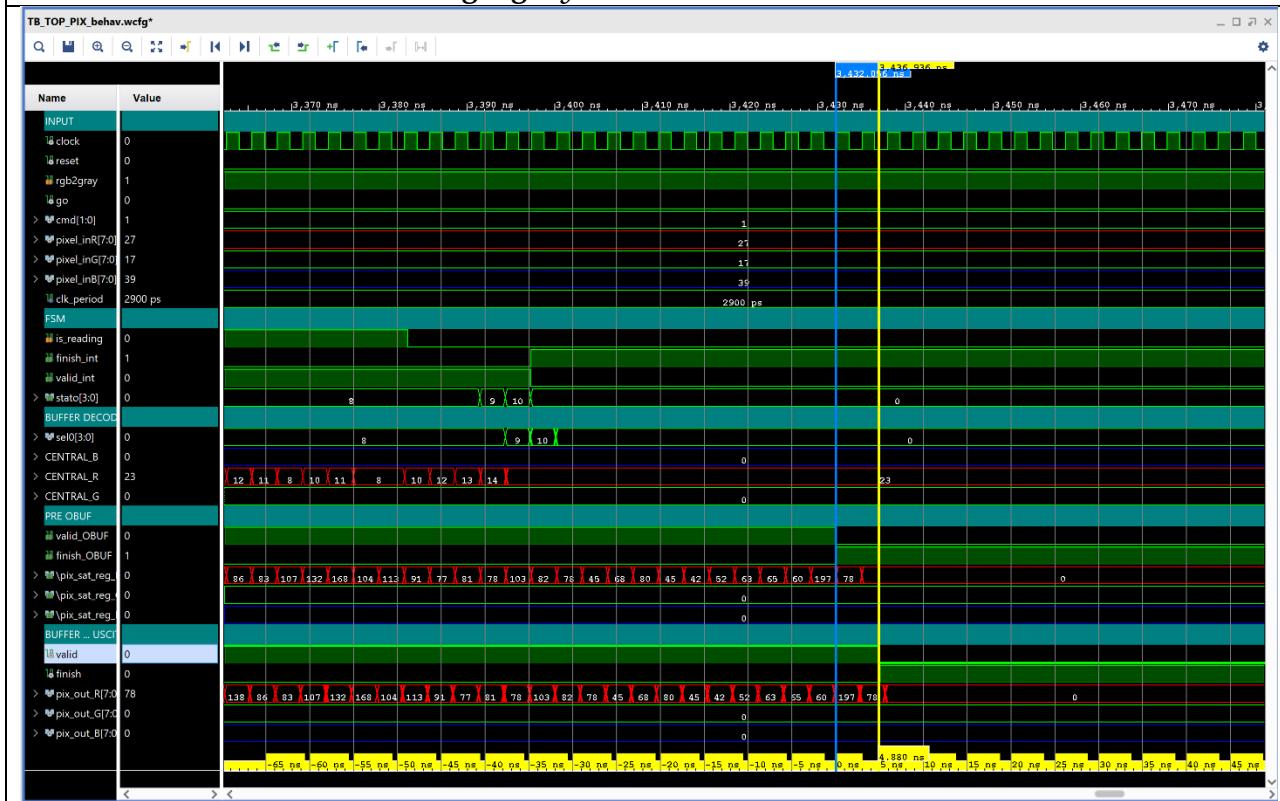


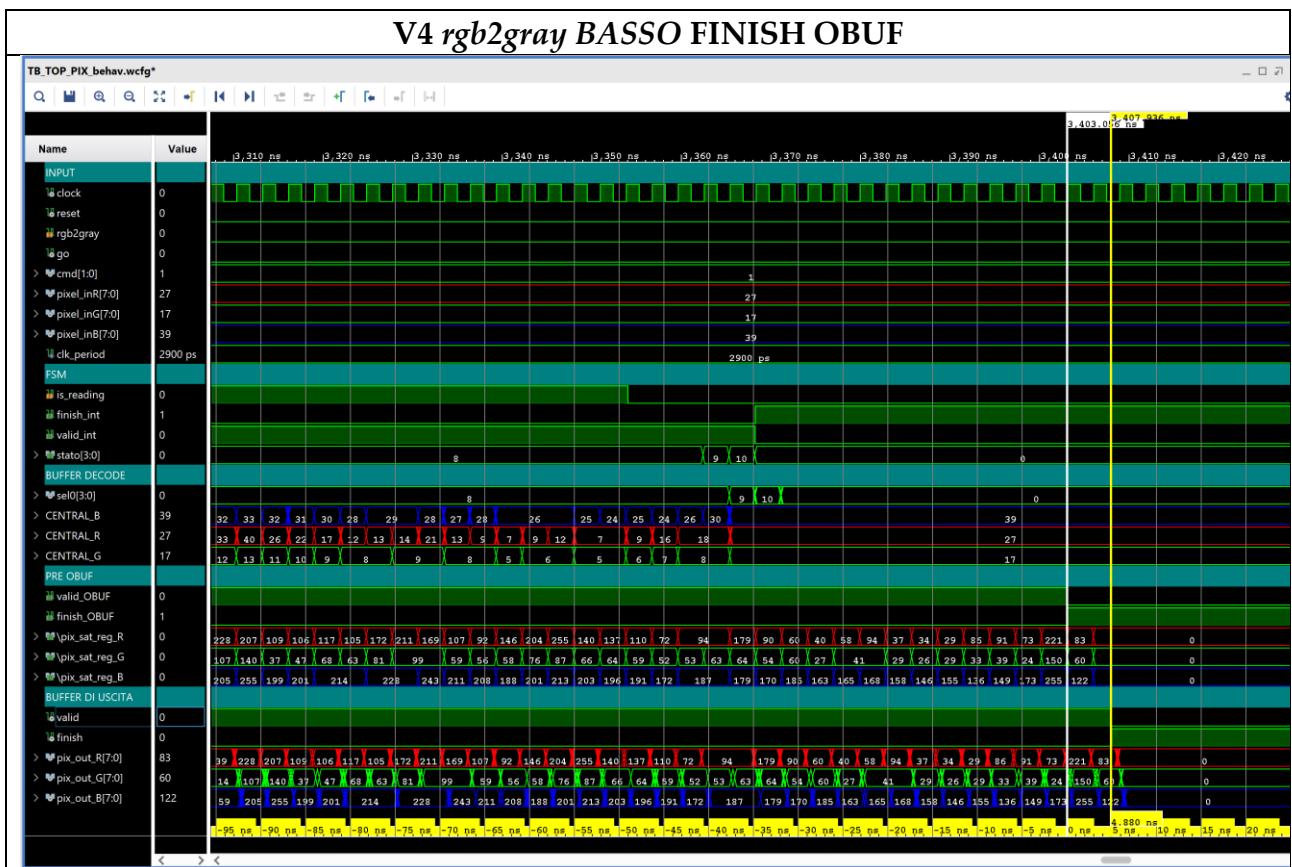
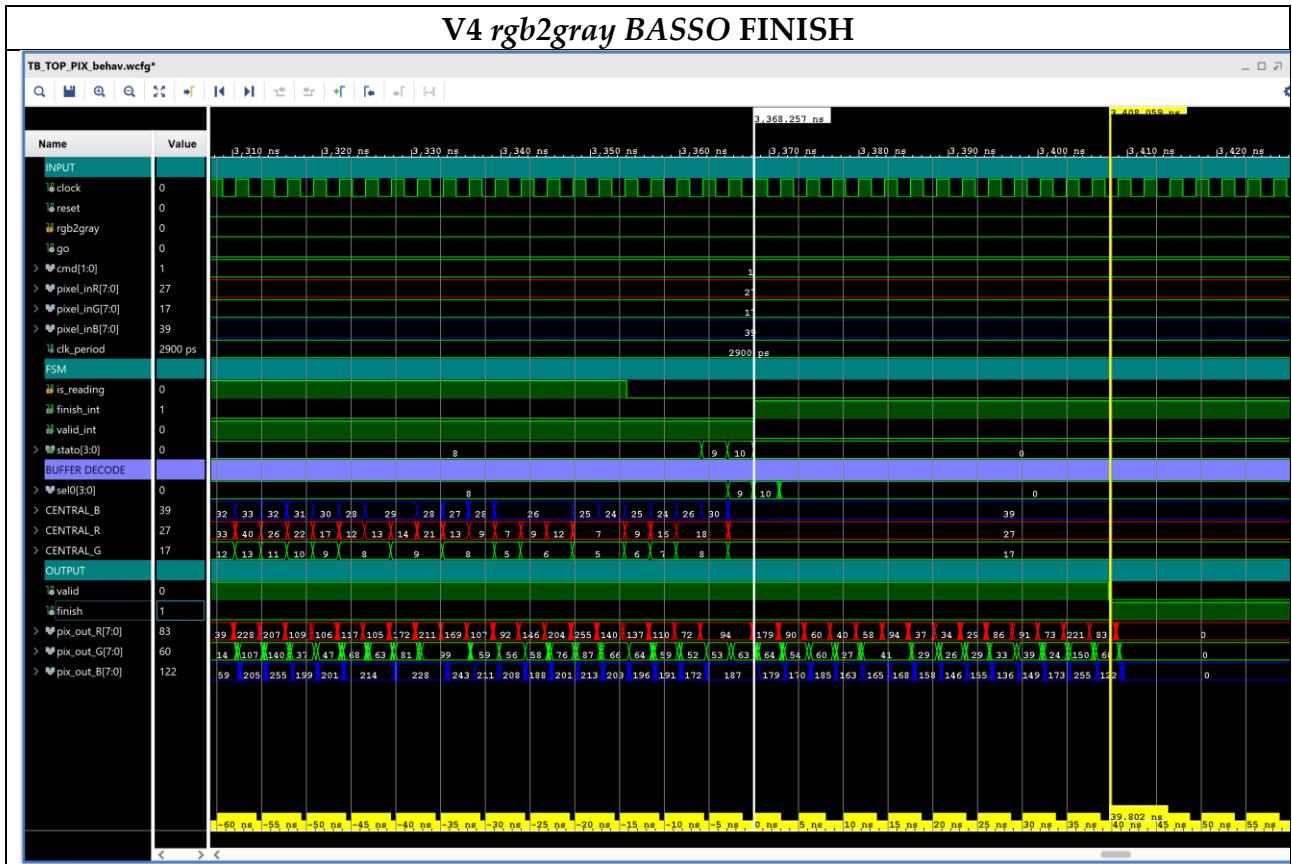


V4 rgb2gray ALTO FINISH

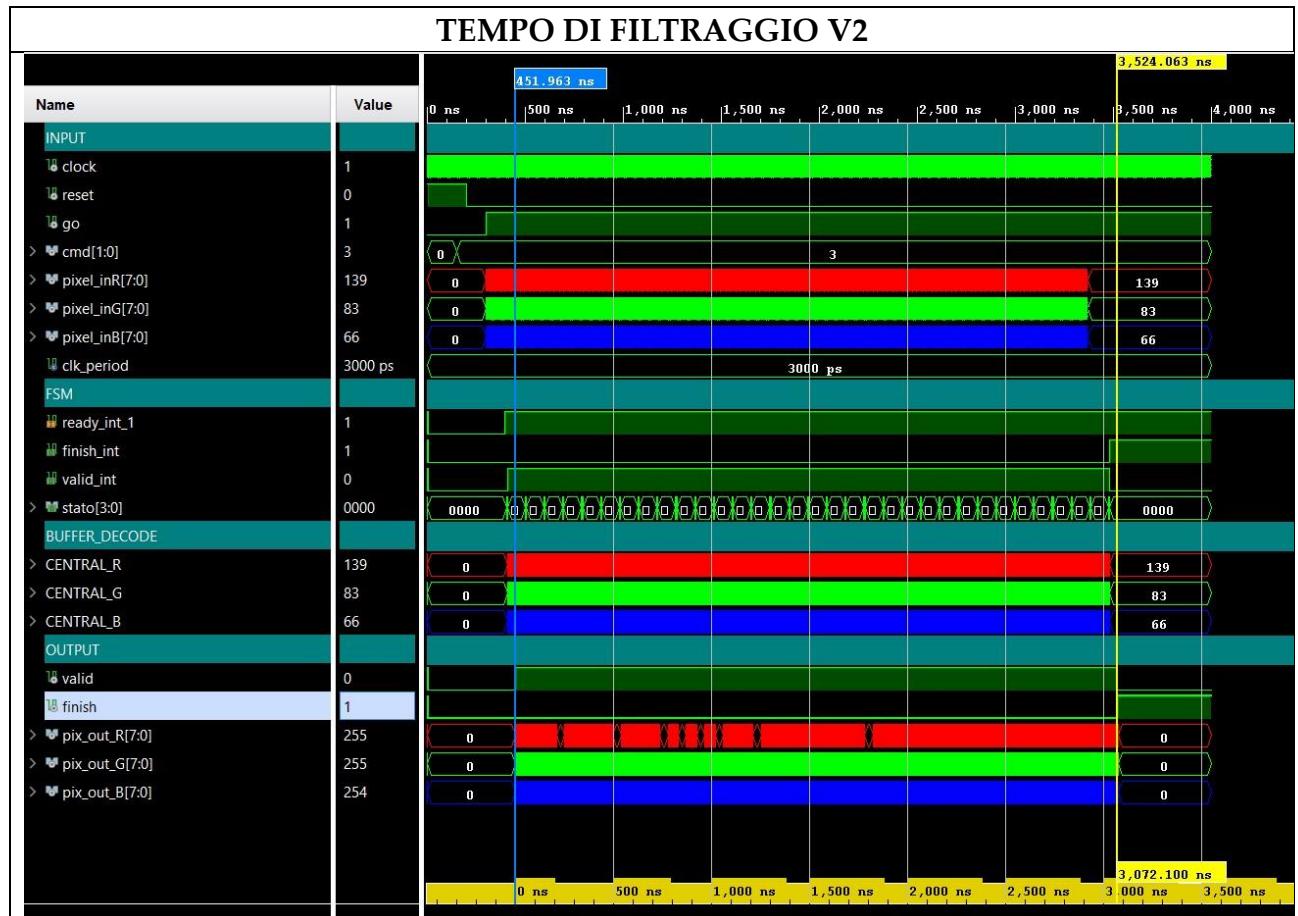
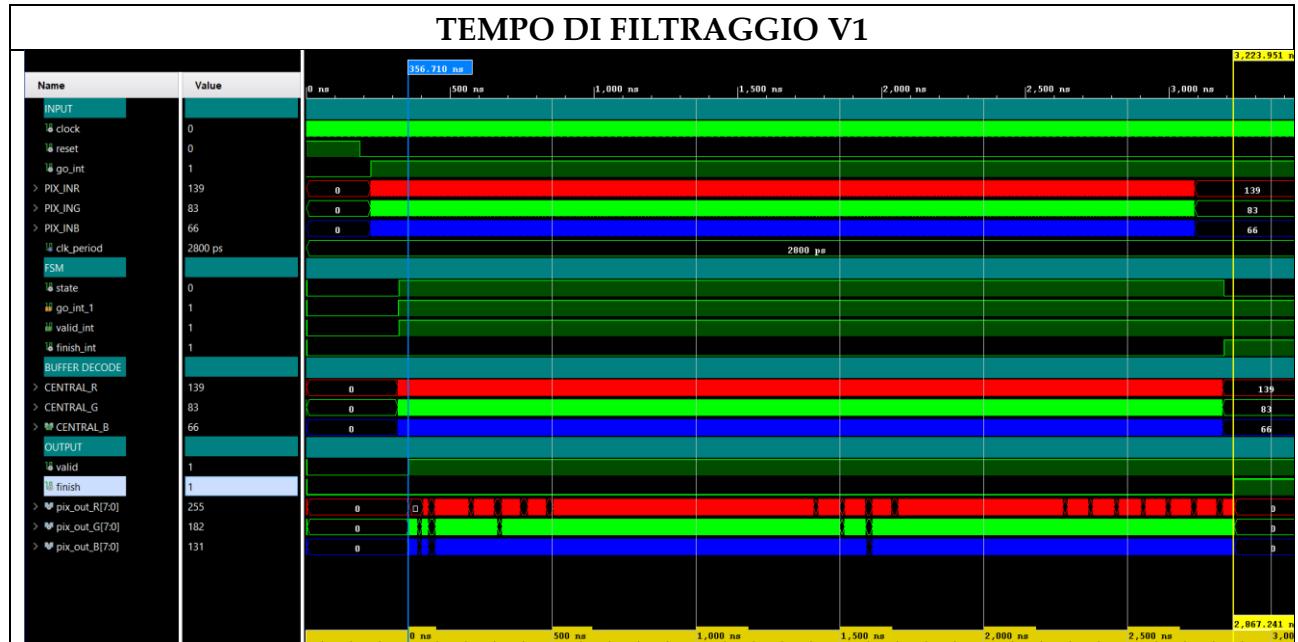


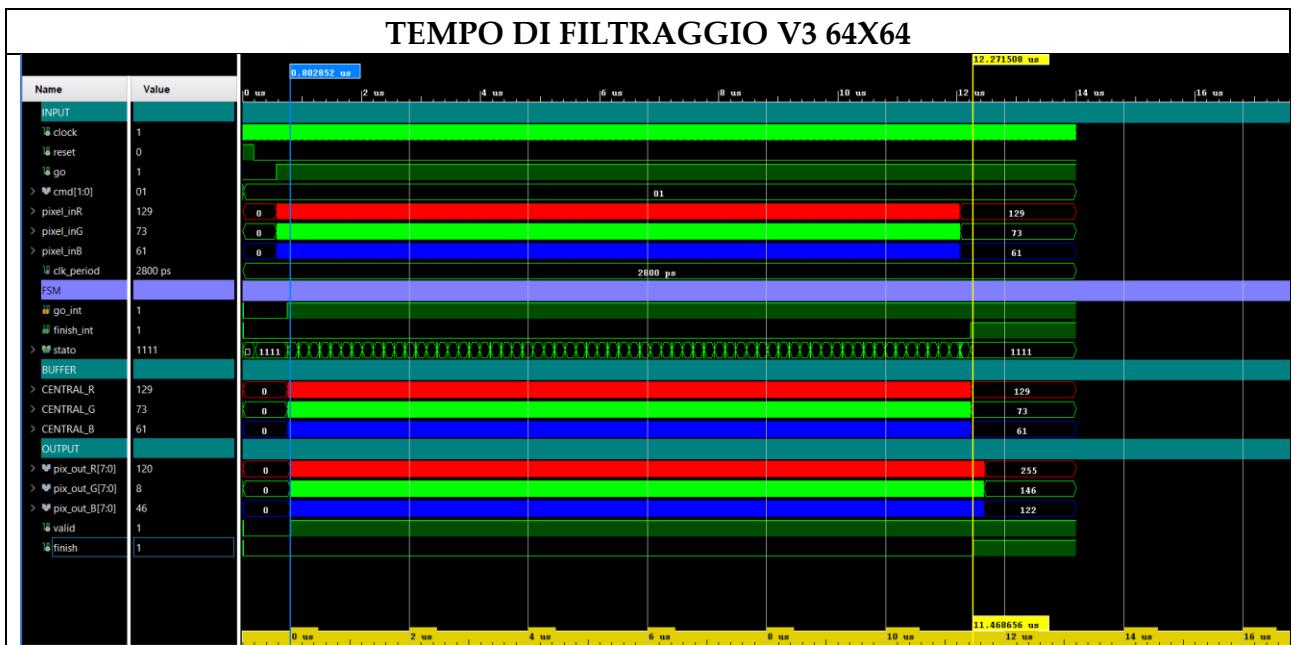
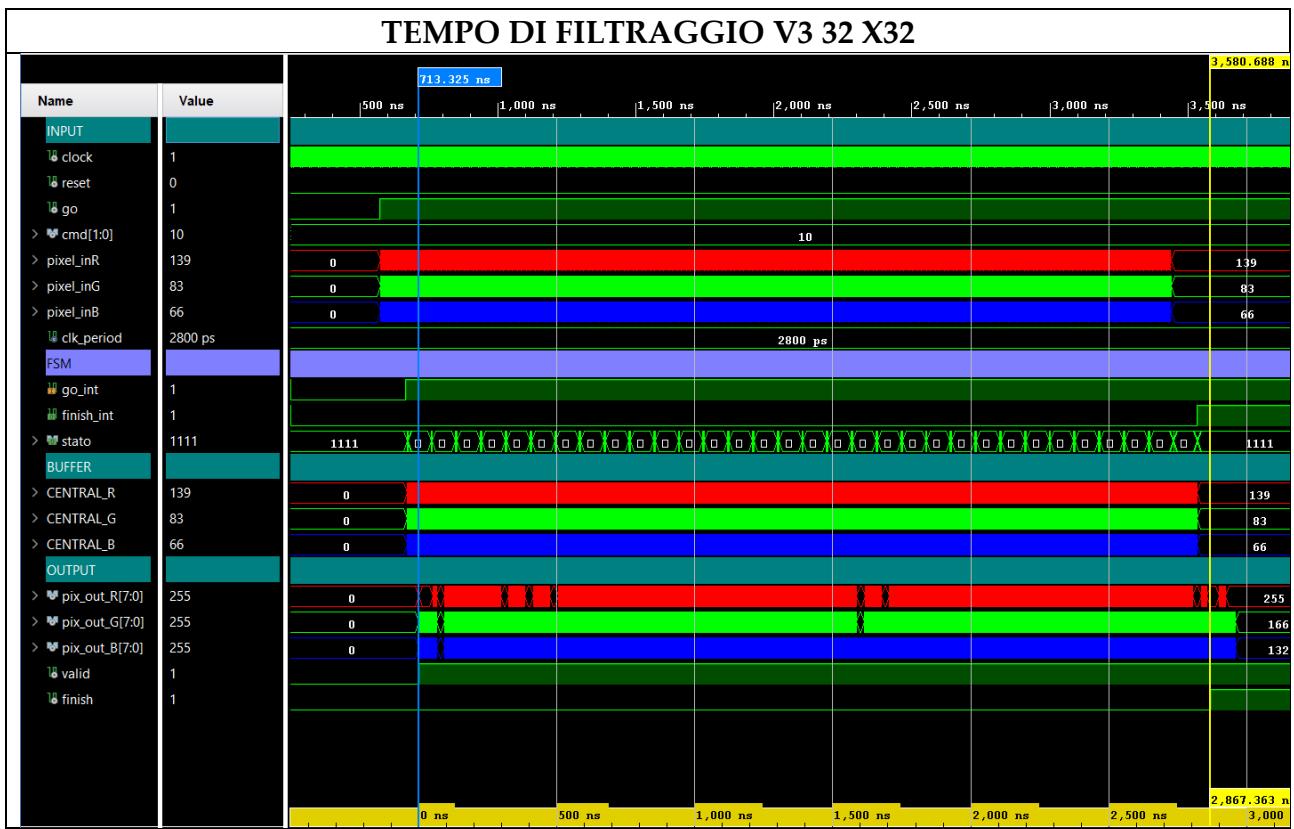
V4 rgb2gray ALTO FINISH OBUF



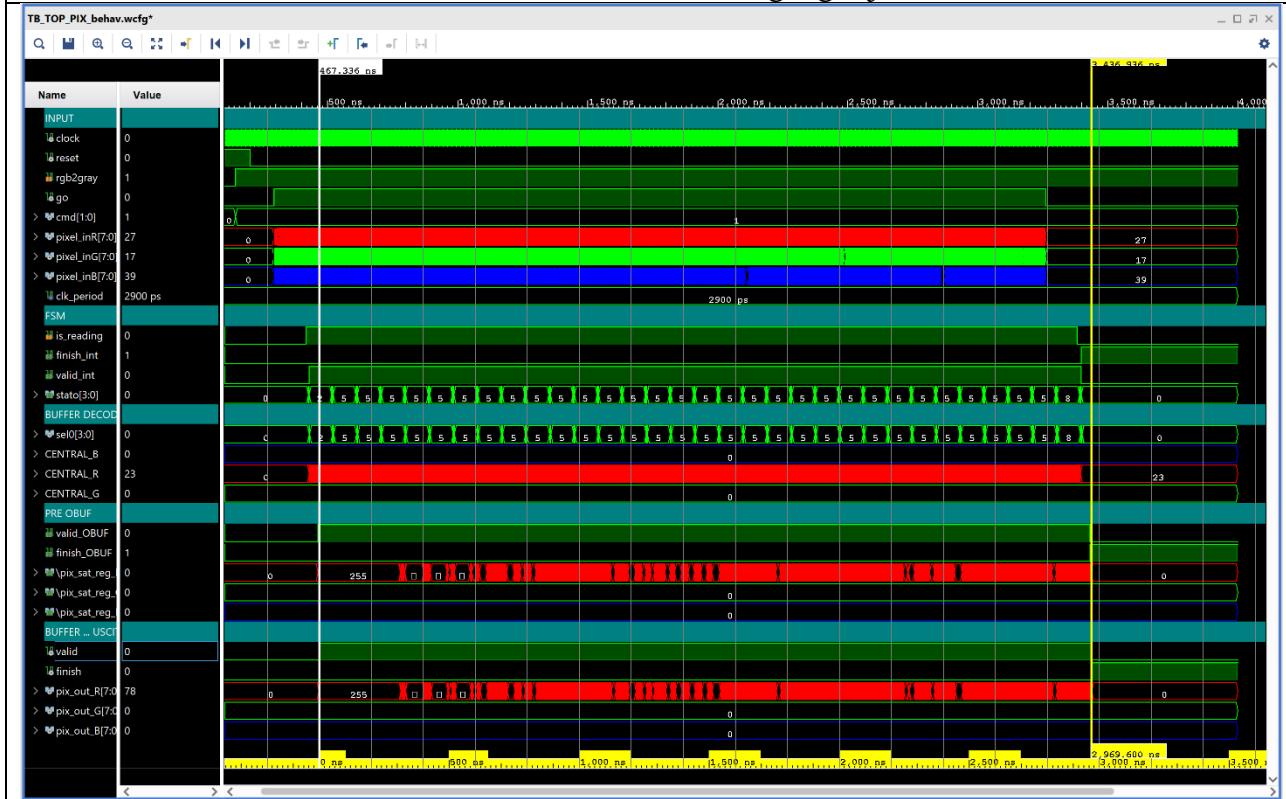


TEMPI DI FILTRAGGIO

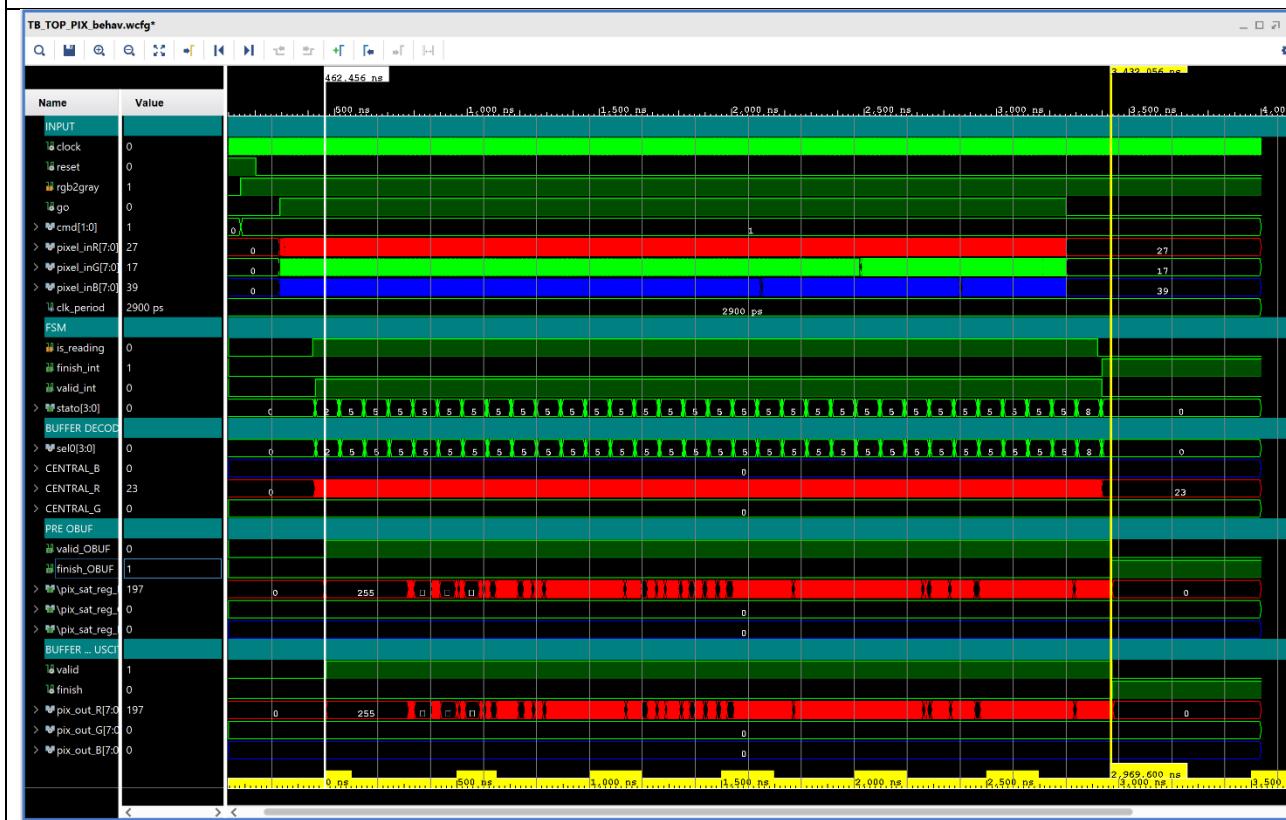




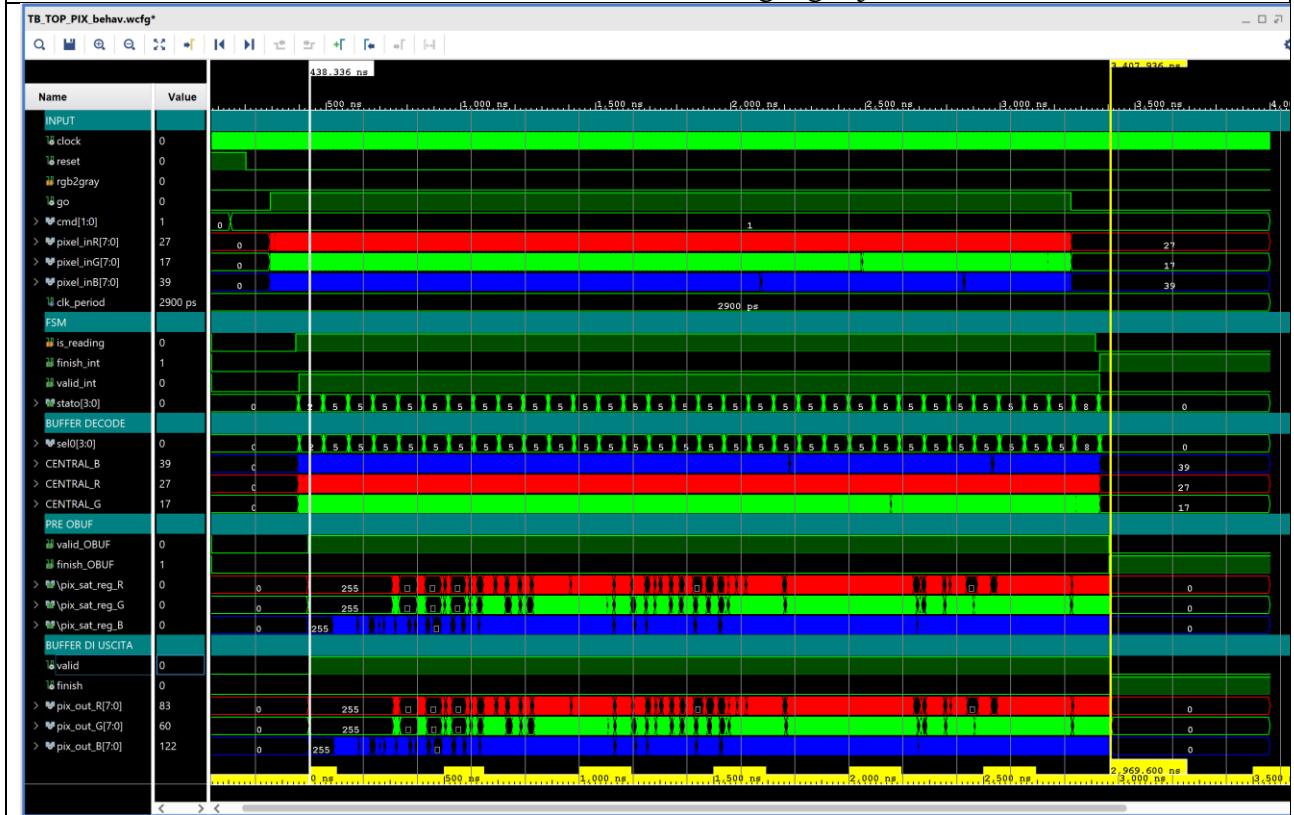
TEMPO DI FILTRAGGIO V4 *rgb2gray* ALTO



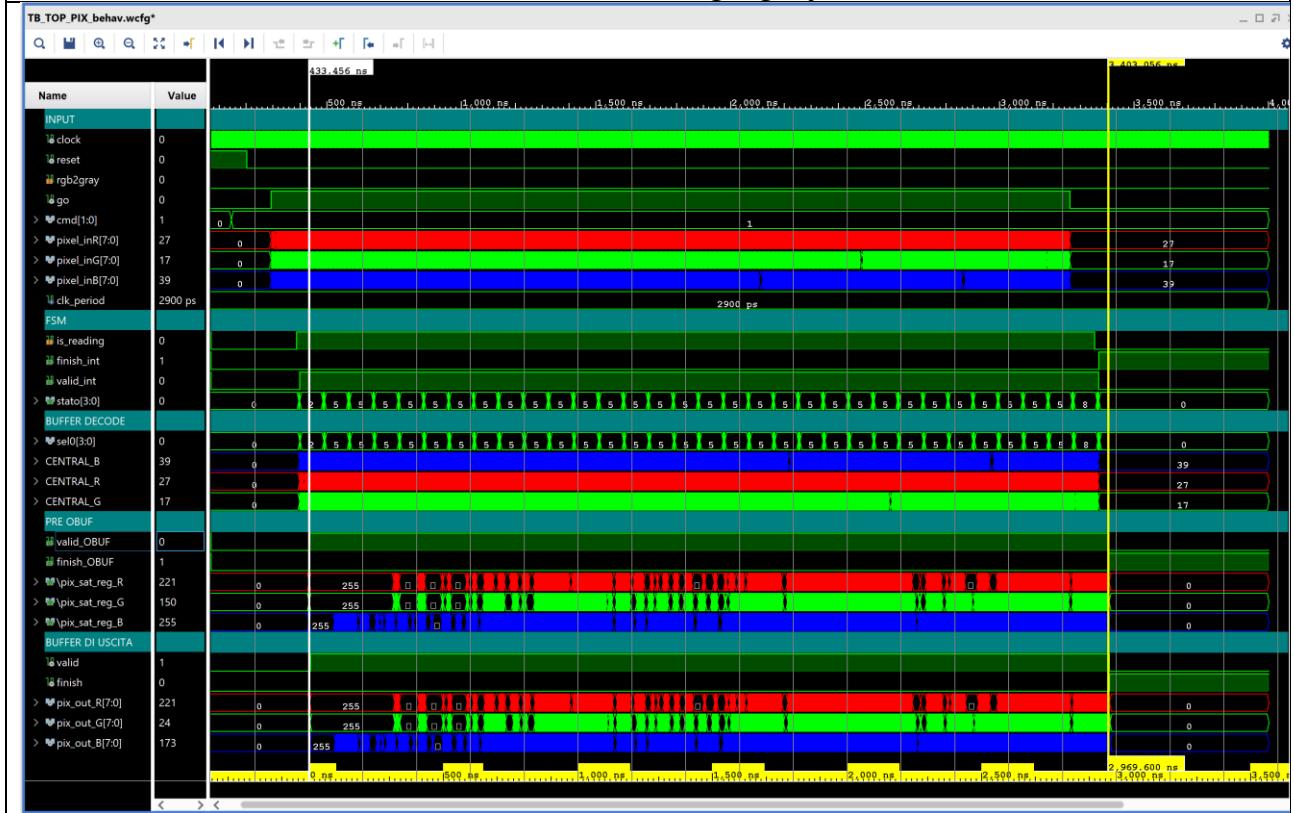
TEMPO DI FILTRAGGIO V4 GREY OBUF



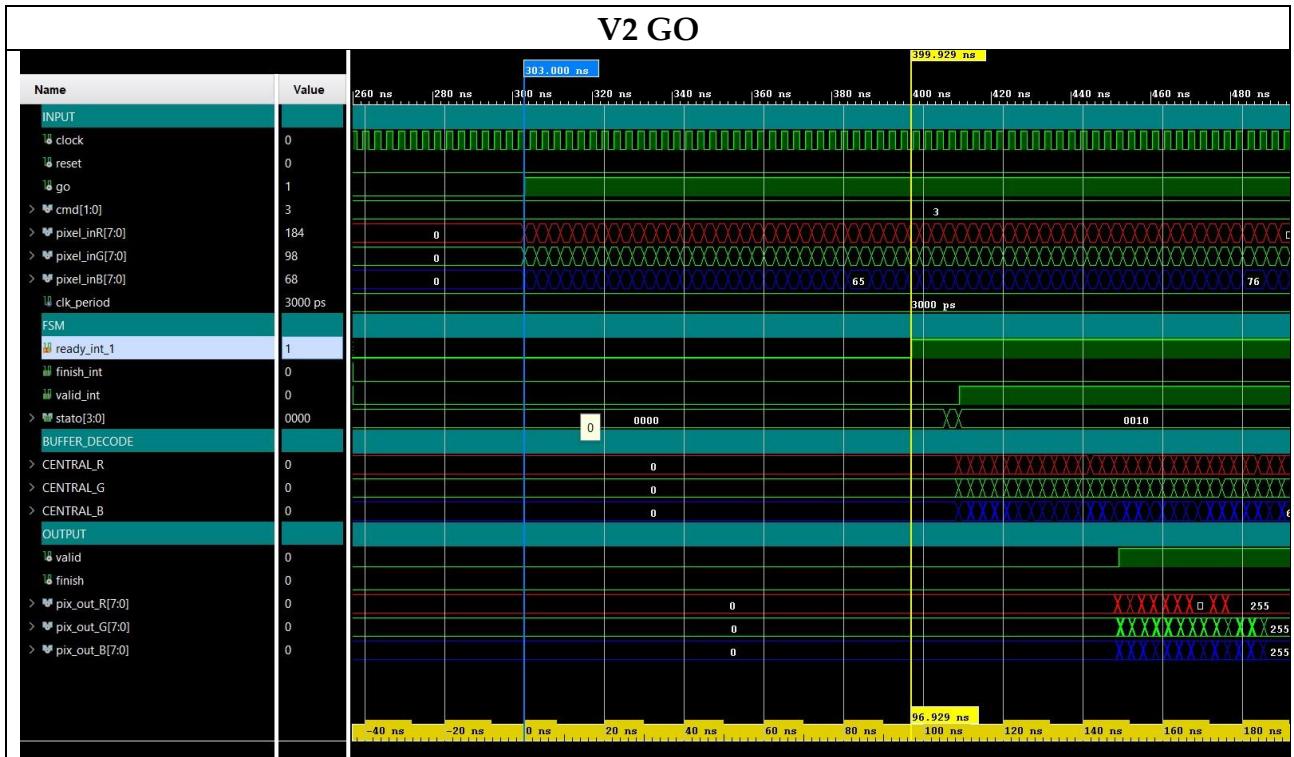
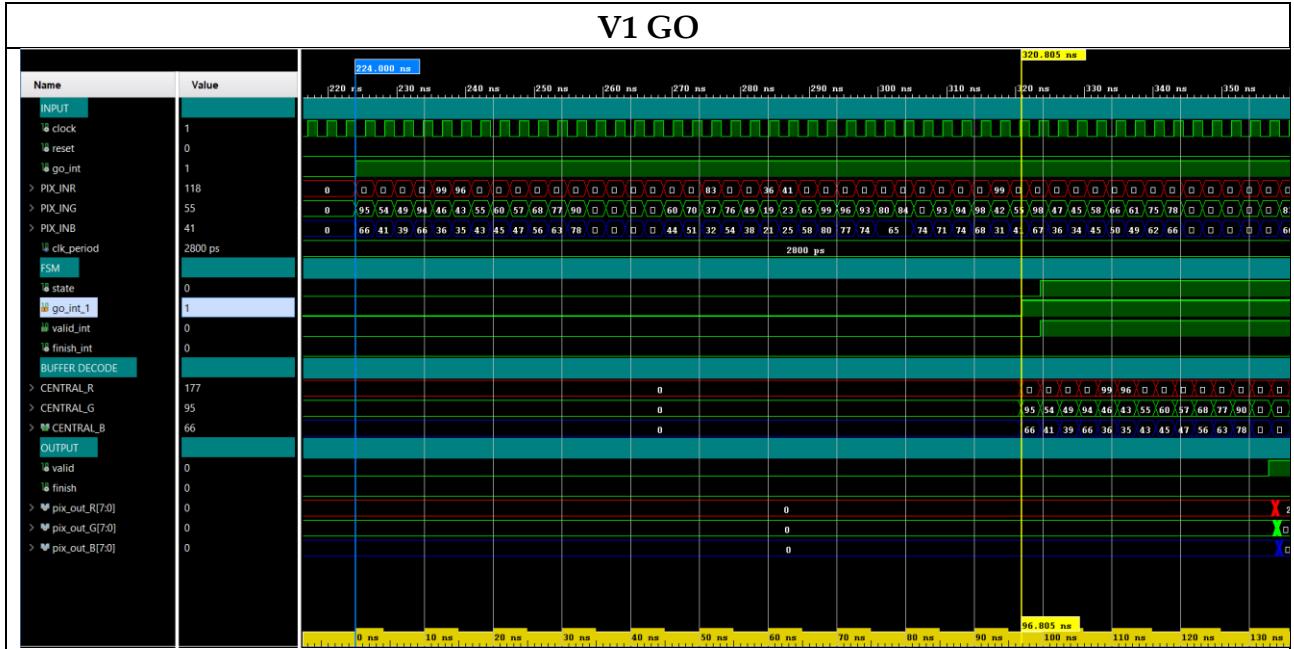
TEMPO DI FILTRAGGIO V4 *rgb2gray* BASSO

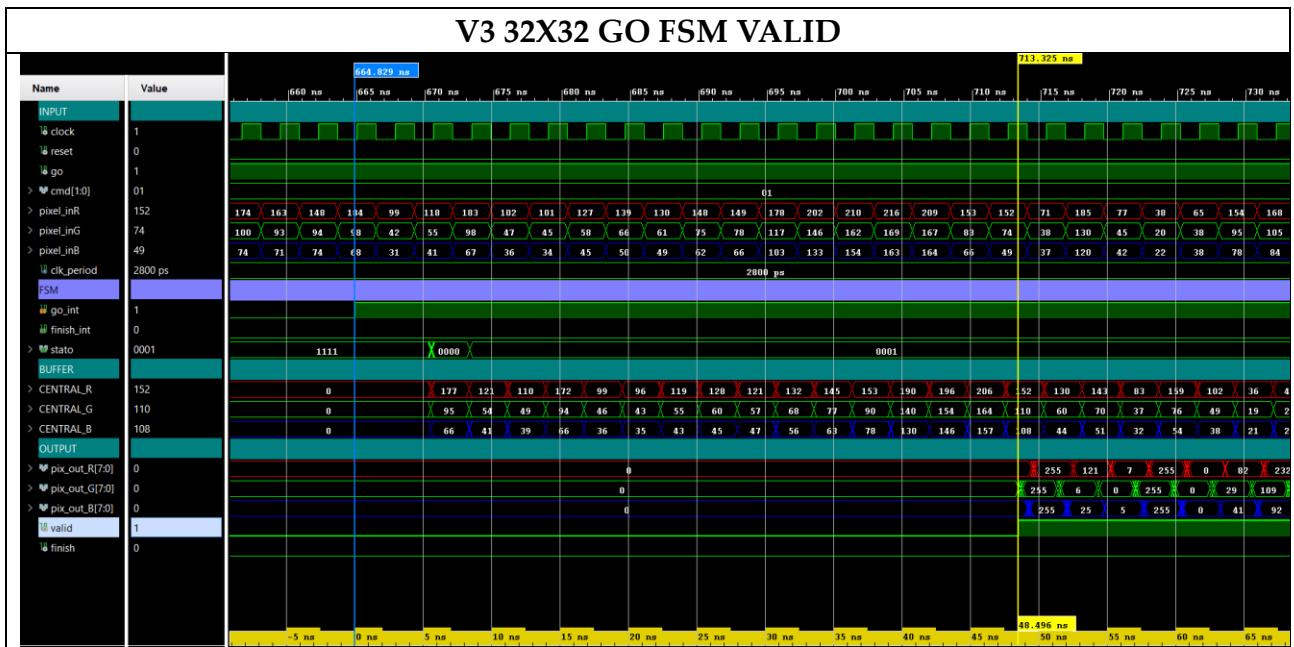
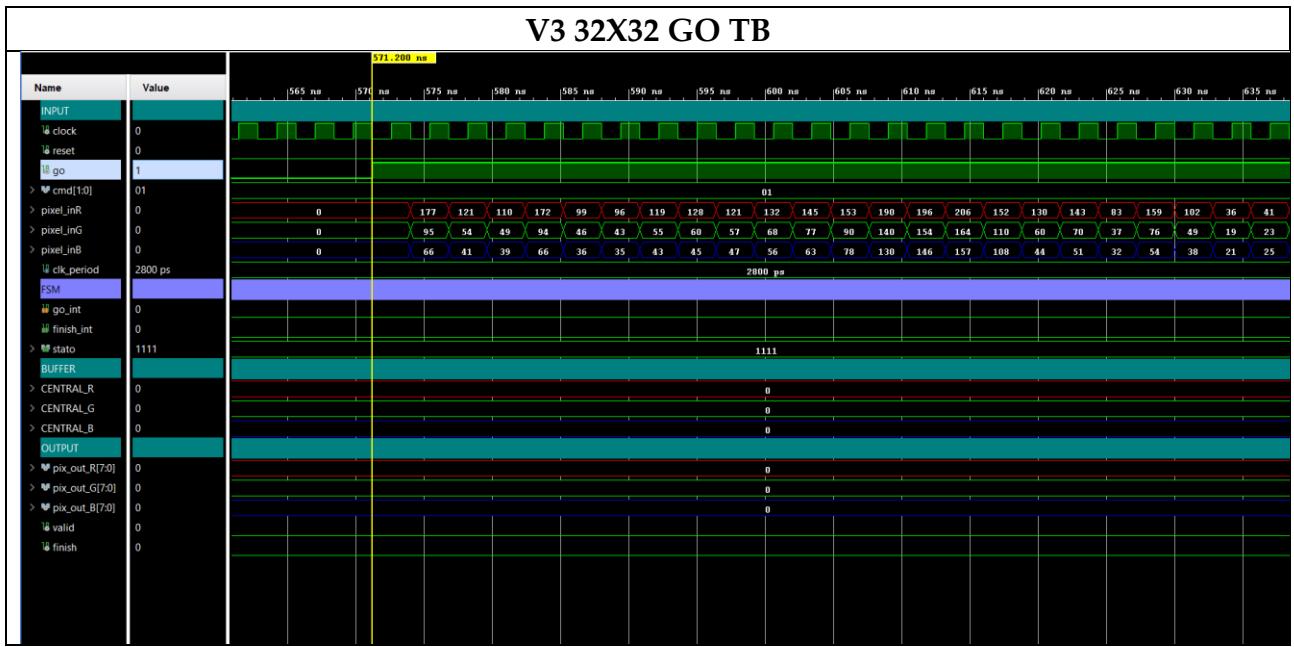


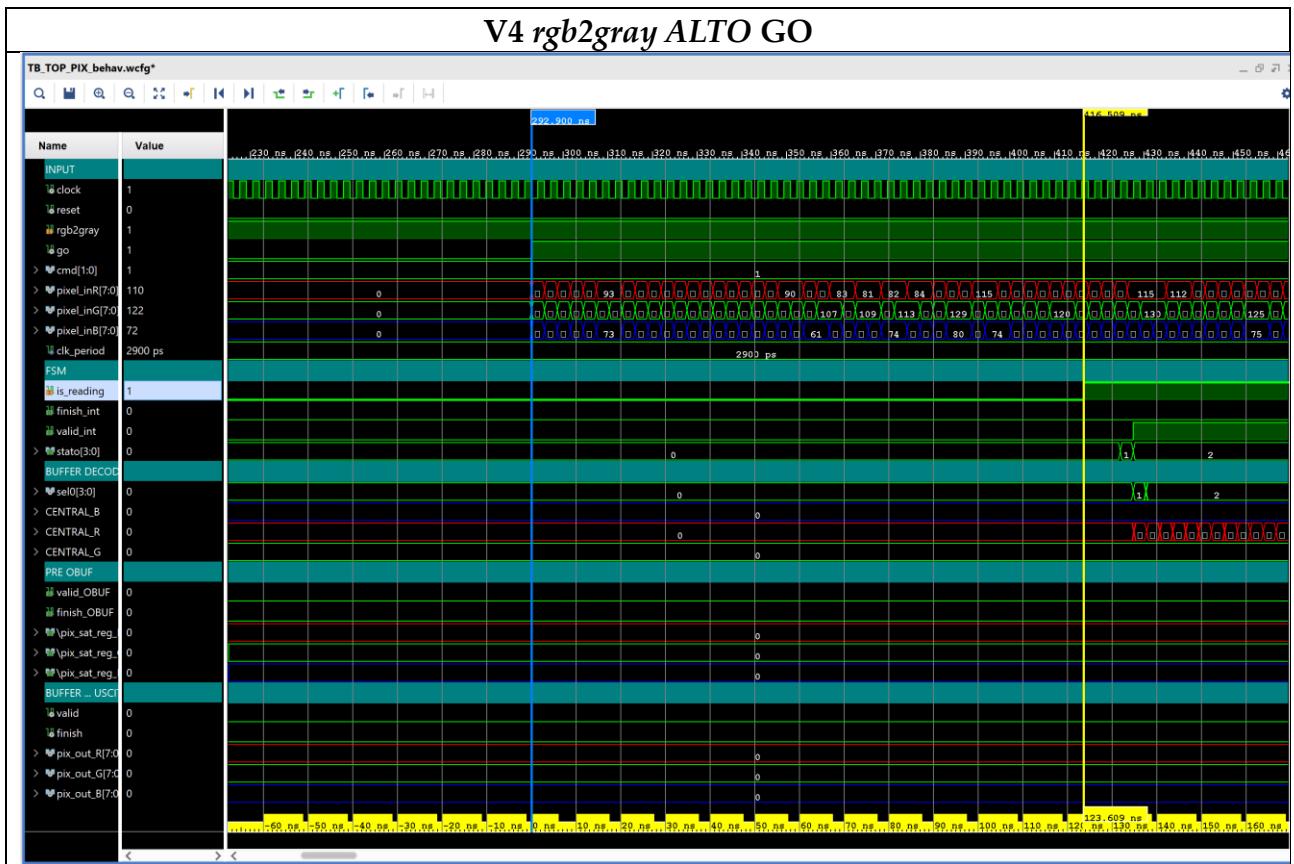
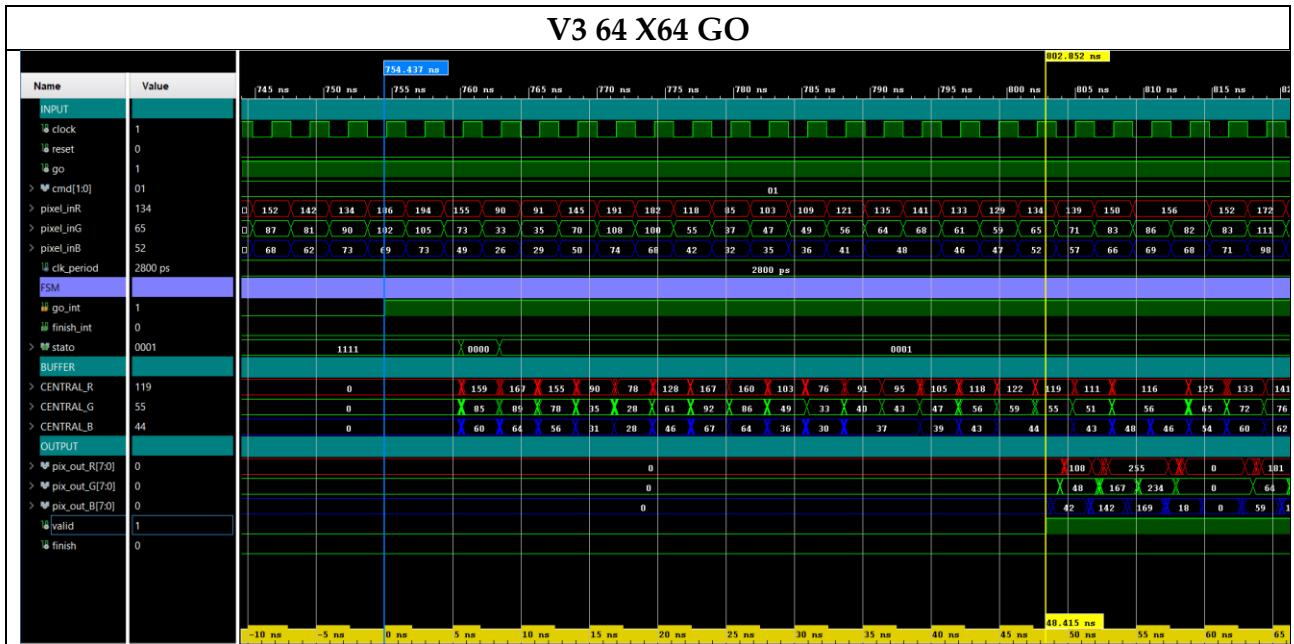
TEMPO DI FILTRAGGIO V4 *rgb2gray* BASSO OBUF

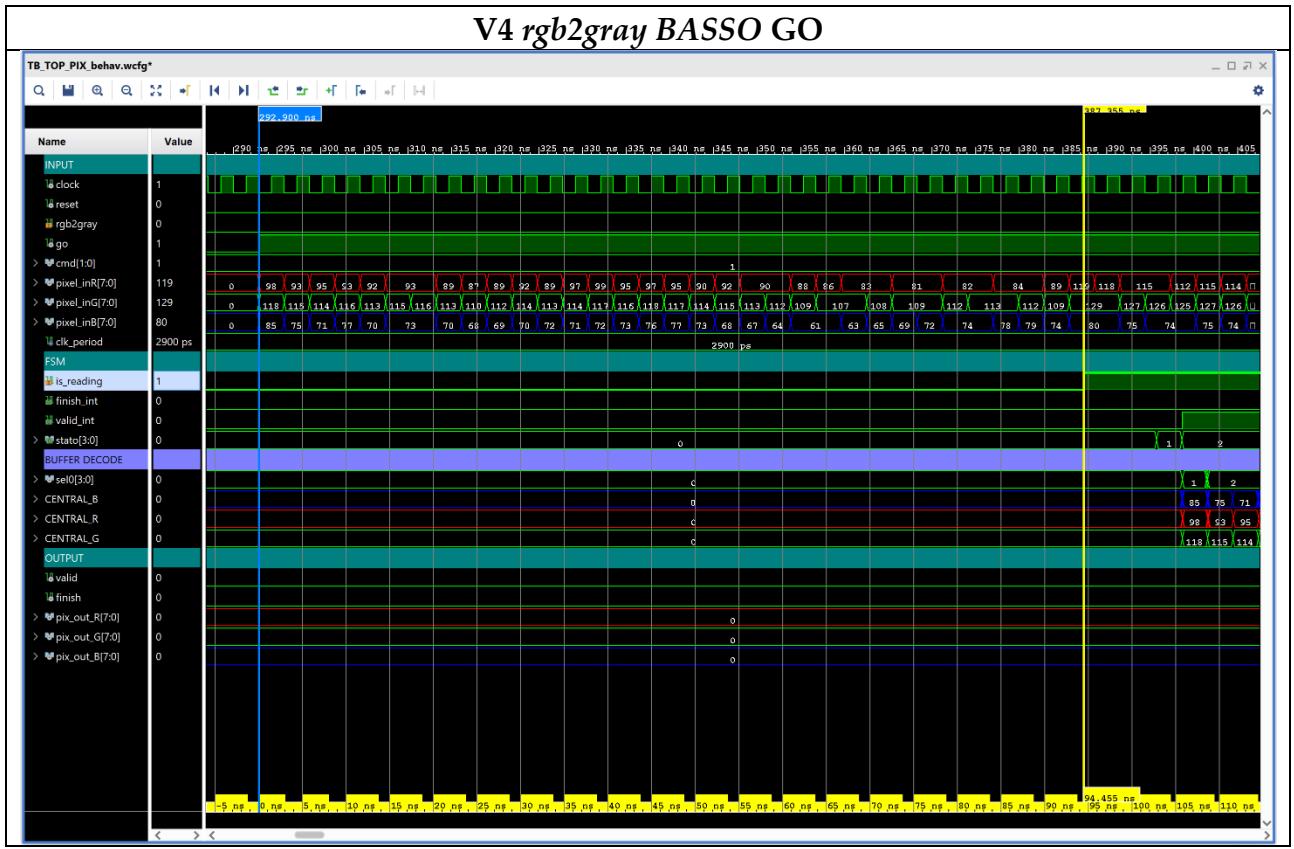


SEGNALI DI INIZIO LETTURA









CONCLUSIONI SULLE SIMULAZIONI POST IMPLMENTATION

Dalle simulazioni **post-implementation** è possibile notare come esse siano fedeli alle simulazioni **behavioral**. Naturalmente per queste altre simulazioni è stato necessario abbassare la frequenza del clock per garantire l'esattezza di tutti i pixel filtrati.

La **V3** fornisce il primo risultato valido dopo **17 colpi di clock** questo perché sono presenti più stadi di pipeline rispetto alle altre versioni.

Inoltre, i tempi di filtraggio di tutte le versioni per immagini **32x32** rientrano nell'intorno dei **3 µs**, differentemente dalla **V3** per immagini **64x64** che impiega per filtrare **11 µs**.

D'altronde:

$$T_{filtraggio} = \text{Dim}(image) * T_{clk} = (n \times n) * T_{clk}$$

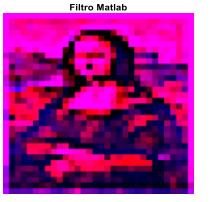
$$T_{filtraggio} = (64 \times 64) * 2.8 \text{ ns} = 11,47 \mu\text{s}$$

Negli screen per la **V4** riportati è evidenziato il ritardo degli **OBUF** per i segnali di uscita di **valid** e **finish**, essi sono più lenti rispetto agli **OBUF** dei pixel di uscita. Questo comporta avere i segnali di **valid** e **finish** non perfettamente sincronizzati con i pixel in uscita. Tuttavia, considerando tali segnali **pre-OBUF** è possibile notare la bontà del sistema implementato.

RISULTATI IN MATLAB

Di seguito sono riportati le verifiche e i confronti delle varie simulazioni ottenute tramite il tool Matlab.

VERSIONE 1

Coefficienti R, G e B $\leq w = 10, w_l = -1, w_c = -1$	
	
 Filtro FPGA "behavioral" MSE = 0.000000	 Filtro FPGA "post timing implementation" MSE = 0.000000
Coefficienti R e B $\leq w = 10, w_l = -1, w_c = -1$ Coeffienti G $\leq w = 0, w_l = 0, w_c = 0$	
	
 Filtro FPGA "behavioral" MSE = 0.000000	 Filtro FPGA "post timing implementation" MSE = 0.000000

Sottolineiamo il fatto che il valore MSE è pari a 0. Questo parametro, chiamato *errore quadratrico medio*, nel nostro caso quantifica la differenza tra le immagini prodotte da MATLAB e dal nostro circuito FPGA. Se questo valore è zero le immagini sono state filtrate correttamente senza errori.

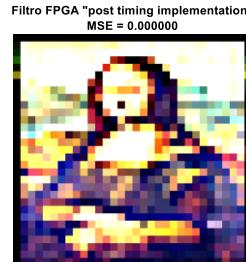
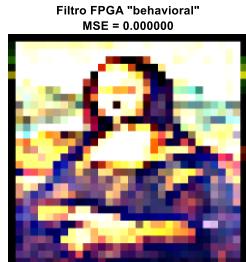
In aggiunta va tenuto presente che il calcolo dell'MSE, nel caso di gestione del bordo **"toroidale mista"**, esclude i bordi. In tutti gli altri casi il valore è calcolato su tutta l'immagine. Nella V1 l'unica gestione consentita è quella toroidale mista per cui nel caso sopra riportato l'MSE è calcolato escludendo il bordo.

VERSIONE 2

$\text{Coefficients } R, G, B \leq w = 11, w_l = -1, w_c = -1$ 	Gestione dei bordi usata: toroidale mista
	Filtro Matlab
	Filtro FPGA "behavioral" MSE = 0.000000
	Filtro FPGA "post timing implementation" MSE = 0.000000

$\text{Coefficients } R, G, B \leq w = 11, w_l = -1, w_c = -1$ 	Gestione dei bordi usata: padding zeri
	Filtro Matlab
	Filtro FPGA "behavioral" MSE = 0.000000
	Filtro FPGA "post timing implementation" MSE = 0.000000

Coefficienti R, G, B <= $w = \mathbf{11}, w_l = -\mathbf{1}, w_c = -\mathbf{1}$	Gestione dei bordi usata: Padding 255
---	---



Coefficienti R, G, B <= $w = \mathbf{11}, w_l = -\mathbf{1}, w_c = -\mathbf{1}$	Gestione dei bordi usata: mirroring
---	---



Coefficienti R, G <= $w = 11, w_l = -1, w_c = -1$
Coefficienti B <= $w = 0, w_l = 0, w_c = 0$

Gestione dei bordi usata: **padding zeri**

Immagine 32 x 32



Filtro Matlab



Filtro FPGA "behavioral"
MSE = 0.000000



Filtro FPGA "post timing implementation"
MSE = 0.000000



VERSIONE 3

Padding 0 di immagine 32x32



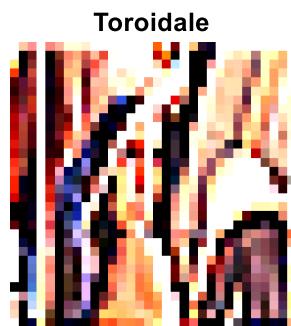
Filtro FPGA "behavioral"
MSE = 0



Filtro FPGA "post timing implementation"
MSE = 0



Gestione dei bordi



Mirroring di immagine 64x64

Immagine 64x64



Filtro Matlab



Filtro FPGA "behavioral"
MSE = 0



Filtro FPGA "post timing implementation"
MSE = 0



Gestione dei bordi

Toroidale



Padding 0



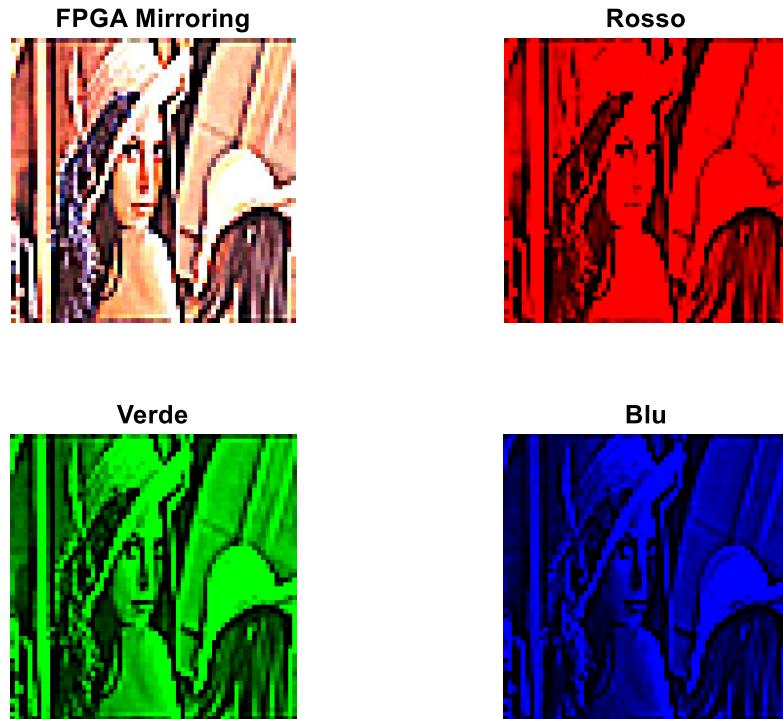
Padding 255



Mirroring



Separazione dei colori



Codice MATLAB per il calcolo e la visualizzazione dei risultati

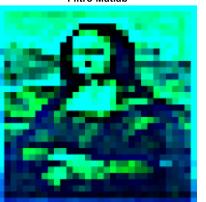
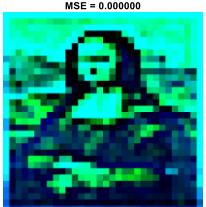
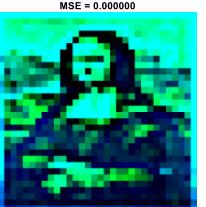
```
% MSE
ref = reshape(double(filtered_image), 1, []);
I_behav = reshape(double(myfilt_image_behav), 1, []);
I_impl = reshape(double(myfilt_image_impl), 1, []);
mse_behav = norm((ref-I_behav))^2/(3*dim^2);
mse_impl = norm((ref-I_impl))^2/(3*dim^2);
mse = norm((I_behav-I_impl))^2/(3*dim^2);

% Visualizza l'immagine originale e l'immagine filtrata
caption_behav = sprintf('Filtro FPGA "behavioral"\nMSE = %d', mse_behav);
caption_impl = sprintf('Filtro FPGA "post timing implementation"\nMSE = %d', mse_impl);
subplot(2,2,1), imshow(resized_image), title(sprintf('Immagine %dx%d', dim, dim))
subplot(2,2,2), imshow(filtered_image), title('Filtro Matlab')
subplot(2,2,3), imshow(myfilt_image_behav), title(caption_behav)
subplot(2,2,4), imshow(myfilt_image_impl), title(caption_impl)
```

$$MSE = \frac{|M_{REF} - M_{FILTRO}|^2}{3 \cdot DIM^2} = 0$$

$$PSNR = 20 \cdot \log_{10} \left(\frac{255}{\sqrt{MSE}} \right) \rightarrow \infty$$

VERSIONE 4

Coefficienti R <= $w = 0, w_l = 0, w_c = 0$ Coefficienti G, B <= $w = 11, w_l = -1, w_c = -1$ RGB2GRAY <= '0'	Gestione dei bordi usata: Padding zeri
  	

La **V4** quando il segnale *rgb2gray* è basso si comporta esattamente come la **V2**, per questo motivo presentiamo una sola immagine di prova. Maggiori dettagli saranno forniti sulla parte che differisce dalla **V2**, ovvero l'utilizzo di un filtro **RGB2GRAY** a monte del filtraggio. Ribadiamo che il prefiltraggio da **RGB** a **GRAY** avviene *real time* tramite un apposito segnale mandato da test bench, a discrezione dell'utente, il quale attraverso dei multiplexer fa in modo che i pixel in ingresso prima di entrare nella zona hardware che ne effettua le operazioni matematiche siano pretrattati in un blocco apposito chiamato **RGB_TO_GRAY**, il quale da una terna di pixel **RGB** produce un singolo pixel convertito in scala di grigi.

PURE GRAY

Coefficienti R <= $w = 1, w_l = 0, w_c = 0$
 Coefficienti G, B <= **Don't care**
 RGB2GRAY<= '1'

Gestione dei bordi usata: **Don't care**



L'MSE in questo caso non è zero. La differenza tra il pixel in gray valutato dal blocco di **RGB TO GRAY** e quello valutato da **MATLAB** è sempre compresa tra -1 e 1. Per questo motivo, a livello visivo, le immagini sembrano identiche. In totale i pixel che differiscono dal valore esatto, in questa immagine sono 340 quindi il 33.2%; tuttavia questo valore è da contestualizzare al fatto che queste differenze, seppur tante siano solo di +1 o -1 su una scala da 0 a 255.

```

num_pixel_diversi = 0;
for i = 1:32
    for j = 1:32
        gray_pixel_value = double(r_image(i, j));
        filtered_pixel_value = double(Rmyfilt_image(i, j));
        A = gray_pixel_value - filtered_pixel_value;
        if abs(A) > 1 % Use abs to check for both positive and negative differences
            num_pixel_diversi = num_pixel_diversi + 1;
        end
    end
end
fprintf('Il numero di pixel diversi tra le due immagini è: %d\n', num_pixel_diversi);

```

Command Window

```

>> rgb2grey
Il numero di pixel diversi tra le due immagini è: 0

```

Con il codice riportato sopra verifichiamo quanti siano i pixel che differiscono di una quantità maggiore di uno rispetto a quelli valutati da **MATLAB**. Per dare un po' di contesto specifichiamo che la variabile *r_image* contiene l'immagine filtrata da **MATLAB**, mentre la variabile *Rmyfilt_image* contiene l'immagine filtrata dal nostro circuito **FPGA**. Il risultato ottenuto è **zero** quindi vuol dire che l'errore commesso è in valore assoluto minore o uguale ad uno. Infatti, sostituendo ad *if abs(A) > 1 => if abs(A) > 0* otteniamo

Command Window

```
>> rgb2grey
```

Il numero di pixel diversi tra le due immagini è: 340

Coefficienti R <= w = 10, w_l = -1, w_c = -1 Coefficienti G, B <= Don't care RGB2GRAY<= '1'	Gestione dei bordi usata: Padding zeri
	
Filtro FPGA "behavioral" MSE = 18.328125 	Filtro FPGA "post timing implementation" MSE = 18.328125 

Coefficienti R $\leq w = 10, w_l = -1, w_c = -1$
 Coefficienti G, B \leq Don't care
 RGB2GRAY $\leq '1'$

Gestione dei bordi usata: **Padding 255**

Originale Matlab



matlab filter gray



Filtro FPGA "behavioral"

MSE = 16.758789



Filtro FPGA "post timing implementation"

MSE = 16.758789



Coefficienti R $\leq w = 2, w_l = 1, w_c = -1$
 Coefficienti G, B \leq Don't care
 RGB2GRAY $\leq '1'$

Gestione dei bordi usata: **Padding zeri**

Originale Matlab



matlab filter gray



Filtro FPGA "behavioral"
 MSE = 8.658203



Filtro FPGA "post timing implementation"
 MSE = 8.658203



Coefficienti R <= $w = 2$, $w_l = 1$, $w_c = -1$
Coefficienti G, B <= **Don't care**
RGB2GRAY<= '1'

Gestione dei bordi usata: **Mirroring**

Originale Matlab



matlab filter gray



Filtro FPGA "behavioral"
MSE = 23.699219



Filtro FPGA "post timing implementation"
MSE = 23.699219



TEST BENCH

Di seguito sono forniti i codici VHDL dei Testbench utilizzati per le simulazioni.

TESTBENCH V1

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.std_logic_arith.all;
use ieee.numeric_std.all;
use STD.textio.all;
use ieee.std_logic_textio.all;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity TB_TOP_PIX is

end TB_TOP_PIX;

architecture Behavioral of TB_TOP_PIX is

component TOP is
port(clk,rst,go: in std_logic;
pixel_in: in std_logic_vector(23 downto 0);
wc,wl,w: in std_logic_vector(23 downto 0);
pixel_out: out std_logic_vector(23 downto 0);
valid,finish: out std_logic);
end component;

signal wc,wl,w :std_logic_vector(23 downto 0):=(others=>'0');
signal pixel : std_logic_vector(23 downto 0):=(others=>'0');
signal clock,reset,go_int:std_logic := '0';
signal valid, finish: std_logic;
signal pix_out: std_logic_vector(23 downto 0);
signal pix_out_R,pix_out_G,pix_out_B: std_logic_vector(7 downto 0);
constant clk_period : time := 2.8 ns;

file file_RESULTS_R, file_RESULTS_G, file_RESULTS_B, R,G,B : text;

begin

uut: TOP port map(clock,reset,go_int,pixel,wc,wl,w,pix_out,valid,finish);

pix_out_R<=pix_out(7 downto 0);
pix_out_G<=pix_out(15 downto 8);
pix_out_B<=pix_out(23 downto 16);

process
begin
wait for clk_period/2;
clock <=not clock;
end process;
```

```

    process
    begin
    reset <= '1';
    wait for 150 ns;
    --R
    w(7 downto 0)<=conv_std_logic_vector(10,8);
    wc(7 downto 0)<=conv_std_logic_vector(-1,8);
    wl(7 downto 0)<=conv_std_logic_vector(-1,8);
    --
    --G
    w(15 downto 8)<=conv_std_logic_vector(0,8);
    wc(15 downto 8)<=conv_std_logic_vector(0,8);
    wl(15 downto 8)<=conv_std_logic_vector(0,8);
    --
    --B
    w(23 downto 16)<=conv_std_logic_vector(10,8);
    wc(23 downto 16)<=conv_std_logic_vector(-1,8);
    wl(23 downto 16)<=conv_std_logic_vector(-1,8);

    wait for 38 ns;
    -- wait for clk_period/2;
    reset <= '0';
    wait;
    end process;

process
    variable rdline : line;
    variable tmp : integer;
    file R : text open read_mode is
    "C:\Users\raffy\Documents\MATLAB\prog_perri\RMonaLisa.txt";
    begin
    wait for 80*clk_period;
    while not endfile(R) loop
        readline(R, rdline);
        read(rdline, tmp);
        go_int<='1';
        pixel(7 downto 0) <= CONV_STD_LOGIC_VECTOR(tmp,8);
        wait for CLK_period;
    end loop;
    WAIT;
end process;

process
    variable rdline : line;
    variable tmp : integer;
    file G : text open read_mode is
    "C:\Users\raffy\Documents\MATLAB\prog_perri\GMonaLisa.txt";

```

```

begin
wait for 80*clk_period;
) while not endfile(G) loop
    readline(G, rdline);
    read(rdline, tmp);
    pixel(15 downto 8) <= CONV_STD_LOGIC_VECTOR(tmp,8);
    wait for CLK_period;
) end loop;
WAIT;
) end process;

) process
variable rdline : line;
variable tmp : integer;
file B : text open read_mode is
"C:\Users\raffy\Documents\MATLAB\prog_perri\BMonaLisa.txt";
begin
wait for 80*clk_period;
) while not endfile(B) loop
    readline(B, rdline);
    read(rdline, tmp);
    pixel(23 downto 16) <= CONV_STD_LOGIC_VECTOR(tmp,8);
    wait for CLK_period;
) end loop;
WAIT;
) end process;

) process
variable vILINE      : line;
variable vOLINE       : line;
variable vSPACE       : character;
begin
file_open(file_RESULTS_R, "Routput_results.txt", write_mode);
file_open(file_RESULTS_G, "Goutput_results.txt", write_mode);
file_open(file_RESULTS_B, "Boutput_results.txt", write_mode);

--wait for 126*clk_period+ 1 ns;
wait for 128.5*clk_period; --for impl
) for i in 0 to 1023 loop

    write(vOLINE, pix_out_R, right, 8);
    writeline(file_RESULTS_R, vOLINE);

    write(vOLINE, pix_out_G, right, 8);
    writeline(file_RESULTS_G, vOLINE);

    write(vOLINE, pix_out_B, right, 8);
    writeline(file_RESULTS_B, vOLINE);

```

```
    wait for clk_period;
end loop;
file_close(file_RESULTS_R);
file_close(file_RESULTS_G);
file_close(file_RESULTS_B);
wait;
end process;

end Behavioral;
```

TESTBENCH V2

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.std_logic_arith.all;
use ieee.numeric_std.all;
use STD.textio.all;
use ieee.std_logic_textio.all;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity TB_TOP_PIX is
end TB_TOP_PIX;

architecture Behavioral of TB_TOP_PIX is

component TOP is
port(clk,rst: in std_logic;
cmd: in std_logic_vector(1 downto 0);
pixel_in: in std_logic_vector(24 downto 0);
wc,wl,w: in std_logic_vector(23 downto 0);
pixel_out: out std_logic_vector(23 downto 0);
valid, finish: out std_logic);
end component;

signal wc,wl,w,temp:std_logic_vector(23 downto 0):=(others=>'0');
signal pixel: std_logic_vector(24 downto 0):=(others=>'0');
signal clock,reset,ena:std_logic := '0';
signal cmd: std_logic_vector(1 downto 0):=(others=>'0');
signal valid, finish: std_logic;
signal pix_out: std_logic_vector(23 downto 0);
signal pix_out_R,pix_out_G,pix_out_B,pixelR,pixelG,pixelB: std_logic_vector(7 downto 0);
constant clk_period : time := 3 ns;

file file_RESULTS_R, file_RESULTS_G, file_RESULTS_B, R,G,B : text;
begin
uut: TOP port map(clock,reset,cmd,pixel,wc,wl,w,pix_out,valid,finish);

pix_out_R<=pix_out(7 downto 0);
pix_out_G<=pix_out(15 downto 8);
pix_out_B<=pix_out(23 downto 16);

pixelR<=pixel(7 downto 0);
pixelG<=pixel(15 downto 8);
pixelB<=pixel(23 downto 16);
```

```

    ) process
        begin
            wait for clk_period/2;
            clock<= not clock;
    ) end process;

    ) process
        begin
            reset <= '1';
            wait for 50*clk_period;
            --R
            w(7 downto 0)<=conv_std_logic_vector(11,8);
            wc(7 downto 0)<=conv_std_logic_vector(-1,8);
            wl(7 downto 0)<=conv_std_logic_vector(-1,8);
    ) --
    ) --G
            w(15 downto 8)<=conv_std_logic_vector(11,8);
            wc(15 downto 8)<=conv_std_logic_vector(-1,8);
            wl(15 downto 8)<=conv_std_logic_vector(-1,8);
    ) --
    ) --B
            w(23 downto 16)<=conv_std_logic_vector(0,8);
            wc(23 downto 16)<=conv_std_logic_vector(0,8);
            wl(23 downto 16)<=conv_std_logic_vector(0,8);
    ) --
    ) -- comando gestione bordi
        cmd <= "01";      -- "00" => toroidale (non fa nulla)
    )                 -- "01" => padding di 0 (nero)
                    -- "10" => padding di 255 (bianco)
    )                 -- "11" => mirroring

        wait for 17*clk_period;
        reset <= '0';
        wait;

    ) end process;

    ) process
        variable rdline : line;
        variable tmp : integer;
        file R : text open read_mode is
            "C:\Users\raffy\Documents\MATLAB\prog_perri\RMonalisa.txt";
        begin
            wait for 101*clk_period;
    ) while not endfile(R) loop
            readline(R, rdline);
            read(rdline, tmp);
            pixel(7 downto 0) <= CONV_STD_LOGIC_VECTOR(tmp,8);

```

```

        wait for CLK_period;
    end loop;
    WAIT;
end process;

process
    variable rdline : line;
    variable tmp : integer;
    file G      : text open read_mode is
"C:\Users\raffy\Documents\MATLAB\prog_perri\GMonaLisa.txt";
begin
    wait for 101*clk_period;
    while not endfile(G) loop
        readline(G, rdline);
        read(rdline, tmp);
        pixel(15 downto 8) <= CONV_STD_LOGIC_VECTOR(tmp,8);
        wait for CLK_period;
    end loop;
    WAIT;
end process;

process
    variable rdline : line;
    variable tmp : integer;
    file B      : text open read_mode is
"C:\Users\raffy\Documents\MATLAB\prog_perri\BMonaLisa.txt";
begin
    wait for 101*clk_period;
    while not endfile(B) loop
        readline(B, rdline);
        read(rdline, tmp);
        pixel(24 downto 16) <= '1'& CONV_STD_LOGIC_VECTOR(tmp,8);
        wait for CLK_period;
    end loop;
    WAIT;
end process;

process
variable v_ILINE      : line;
variable v_OLINE      : line;
variable v_SPACE      : character;
begin
file_open(file_RESULTS_R, "Routput_results.txt", write_mode);
file_open(file_RESULTS_G, "Goutput_results.txt", write_mode);
file_open(file_RESULTS_B, "Boutput_results.txt", write_mode);

```

```
    wait for 148.5*clk_period;
    --wait for 151.5*clk_period;

    for i in 0 to 1023 loop

        write(v_OLINE, pix_out_R, right, 8);
        writeline(file_RESULTS_R, v_OLINE);

        write(v_OLINE, pix_out_G, right, 8);
        writeline(file_RESULTS_G, v_OLINE);

        write(v_OLINE, pix_out_B, right, 8);
        writeline(file_RESULTS_B, v_OLINE);
        wait for clk_period;
    end loop;
    file_close(file_RESULTS_R);
    file_close(file_RESULTS_G);
    file_close(file_RESULTS_B);
    wait;
end process;

end Behavioral;
```

TESTBENCH V3

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.std_logic_arith.all;
use ieee.numeric_std.all;
use STD.textio.all;
use ieee.std_logic_textio.all;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity TB_TOP_PIX is

end TB_TOP_PIX;

architecture Behavioral of TB_TOP_PIX is

component TOP is
port(clk,rst, go: in std_logic;
cmd: in std_logic_vector(1 downto 0);
pixel_in: in std_logic_vector(23 downto 0);
wc,wl,w: in std_logic_vector(23 downto 0);
pixel_out: out std_logic_vector(23 downto 0);
valid, finish: out std_logic);
end component;

signal wc,wl,w,pixel :std_logic_vector(23 downto 0):=(others=>'0');
signal clock,reset:std_logic := '0';
signal go: std_logic :='0';
signal cmd: std_logic_vector(1 downto 0):=(others=>'0');
signal valid, finish: std_logic;
signal pix_out: std_logic_vector(23 downto 0);
signal pix_out_R,pix_out_G,pix_out_B: std_logic_vector(7 downto 0);
constant clk_period : time := 2.8 ns;

file file_RESULTS_R, file_RESULTS_G, file_RESULTS_B, R,G,B : text;

begin

uut: TOP port map(clock,reset,go,cmd,pixel,wc,wl,w,pix_out,valid,finish);
```

```

pix_out_R<=pix_out(7 downto 0);
pix_out_G<=pix_out(15 downto 8);
pix_out_B<=pix_out(23 downto 16);

process
begin
wait for clk_period/2;
clock<= not clock;
end process;

process
begin
reset <= '1';
wait for 15 ns;
-- wait for 1.8 ns; --impl_3
--R
w(7 downto 0)<=conv_std_logic_vector(10,8);
wc(7 downto 0)<=conv_std_logic_vector(-1,8);
wl(7 downto 0)<=conv_std_logic_vector(-1,8);
--
--G
w(15 downto 8)<=conv_std_logic_vector(10,8);
wc(15 downto 8)<=conv_std_logic_vector(-1,8);
wl(15 downto 8)<=conv_std_logic_vector(-1,8);
--
--B
w(23 downto 16)<=conv_std_logic_vector(10,8);
wc(23 downto 16)<=conv_std_logic_vector(-1,8);
wl(23 downto 16)<=conv_std_logic_vector(-1,8);
--
-- comando gestione bordi
cmd <= "10";    -- "00" => toroidale (non fa nulla)
                  -- "11" => padding di 255 (bianco)
                  -- "10" => padding di 0 (nero)
                  -- "01" => mirroring

-- wait for 38 ns;
-- wait for clk_period/2;
wait for 63*clk_period;
reset <= '0';
wait;

```

```

end process;

process
--      function start_cmd(i : in integer) return std_logic is
--      begin
--          if i = 0 then
--              return '1';
--          end if;
--          return '0';
--      end function;
variable rdline : line;
variable tmp : integer;
file R    :  text open read_mode is
"RLena64x64.txt";
begin
wait for 68*3*clk_period;
readline(R, rdline);
read(rdline, tmp);
go <= '1';
wait for CLK_period;
while not endfile(R) loop
readline(R, rdline);
read(rdline, tmp);
pixel(7 downto 0)  <= CONV_STD_LOGIC_VECTOR(tmp,8);
wait for CLK_period;
end loop;
WAIT;
end process;

process
variable rdline : line;
variable tmp : integer;
file G    :  text open read_mode is
"GLena64x64.txt";
begin
wait for 68*3*clk_period;
readline(G, rdline);
read(rdline, tmp);
wait for CLK_period;
while not endfile(G) loop
readline(G, rdline);
read(rdline, tmp);
pixel(15 downto 8)  <= CONV_STD_LOGIC_VECTOR(tmp,8);
wait for CLK_period;
end loop;
WAIT;
end process;

process
variable rdline : line;
variable tmp : integer;
file B    :  text open read_mode is
"BLena64x64.txt";
begin
wait for 68*3*clk_period;
readline(B, rdline);
read(rdline, tmp);
wait for CLK_period;
while not endfile(B) loop
readline(B, rdline);
read(rdline, tmp);
pixel(23 downto 16)  <= CONV_STD_LOGIC_VECTOR(tmp,8);
wait for CLK_period;
end loop;
WAIT;
end process;

```

```

process
variable v_ILINE      : line;
variable v_OLINE       : line;
variable v_SPACE       : character;
variable DIM           : integer := 64;
begin
  file_open(file_RESULTS_R, "Routput_results_padd0_32.txt", write_mode);
  file_open(file_RESULTS_G, "Goutput_results_padd0_32.txt", write_mode);
  file_open(file_RESULTS_B, "Boutput_results_padd0_32.txt", write_mode);

  wait for (221+DIM)*clk_period; --behav
--  wait for (223+DIM)*clk_period + clk_period/2; --impl_6

  for i in 0 to DIM*DIM-1 loop

    write(v_OLINE, pix_out_R, right, 8);
    writeline(file_RESULTS_R, v_OLINE);

    write(v_OLINE, pix_out_G, right, 8);
    writeline(file_RESULTS_G, v_OLINE);

    write(v_OLINE, pix_out_B, right, 8);
    writeline(file_RESULTS_B, v_OLINE);
    wait for clk_period;
  end loop;
  file_close(file_RESULTS_R);
  file_close(file_RESULTS_G);
  file_close(file_RESULTS_B);
  wait;
end process;

end Behavioral;

```

TESTBENCH V4

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.std_logic_arith.all;
use ieee.numeric_std.all;
use STD.textio.all;
use ieee.std_logic_textio.all;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity TB_TOP_PIX is
end entity TB_TOP_PIX;

architecture Behavioral of TB_TOP_PIX is

component TOP is
port(clk,rst,rgb2gray: in std_logic;
cmd: in std_logic_vector(1 downto 0);
pixel_in: in std_logic_vector(24 downto 0);
wc,wl,w: in std_logic_vector(23 downto 0);
pixel_out: out std_logic_vector(23 downto 0);
valid, finish: out std_logic);
end component;

signal wc,wl,w,temp:std_logic_vector(23 downto 0):=(others=>'0');
signal pixel: std_logic_vector(24 downto 0):=(others=>'0');
signal clock_reset,ena,rgb2gray:std_logic := '0';
signal cmd: std_logic_vector(1 downto 0):=(others=>'0');
signal valid, finish: std_logic;
signal pix_out: std_logic_vector(23 downto 0);
signal pix_out_R,pix_out_G,pix_out_B,pixelR,pixelG,pixelB: std_logic_vector(7 downto 0);
constant clk_period : time := 2.9 ns;

file file_RESULTS_R, file_RESULTS_G, file_RESULTS_B, R,G,B : text;

begin
    begin
        uut: TOP port map(clock,reset,rgb2gray,cmd,pixel,wc,wl,w,pix_out,valid,finish);

        pix_out_R<=pix_out(7 downto 0);
        pix_out_G<=pix_out(15 downto 8);
        pix_out_B<=pix_out(23 downto 16);

        pixelR<=pixel(7 downto 0);
        pixelG<=pixel(15 downto 8);
        pixelB<=pixel(23 downto 16);
    end;
end;
```

```

begin
wait for clk_period/2;
clock<= not clock;
) end process;

) process
begin
reset <= '1';
wait for 50*clk_period;
--R oppure il gray dell'immagine
w(7 downto 0)<=conv_std_logic_vector(2,8);
wc(7 downto 0)<=conv_std_logic_vector(-1,8);
wl(7 downto 0)<=conv_std_logic_vector(1,8);
) --
) --G
w(15 downto 8)<=conv_std_logic_vector(10,8);
wc(15 downto 8)<=conv_std_logic_vector(-1,8);
wl(15 downto 8)<=conv_std_logic_vector(-1,8);
) --
) --B
w(23 downto 16)<=conv_std_logic_vector(10,8);
wc(23 downto 16)<=conv_std_logic_vector(-1,8);
wl(23 downto 16)<=conv_std_logic_vector(-1,8);
) --
) -- comando gestione bordi
cmd <= "11";    -- "00" => toroidale (non fa nulla)
)           -- "01" => padding di 0 (nero)
           -- "10" => padding di 255 (bianco)
)           -- "11" => mirroring
rgb2gray<='1';

wait for 20*clk_period;
reset <= '0';
wait;

) end process;

) process
variable rdline : line;
variable tmp : integer;
file R  :  text open read_mode is
"C:\Users\raffy\Documents\MATLAB\prog_perri\RMonalisa.txt";
begin
wait for 101*clk_period;
) while not endfile(R) loop
readline(R, rdline);
read(rdline, tmp);
pixel(7 downto 0)  <= CONV STD LOGIC VECTOR(tmp,8);

```

```

        wait for CLK_period;
    end loop;
    WAIT;
end process;

process
    variable rdline : line;
    variable tmp : integer;
    file G      : text open read_mode is
"C:\Users\raffy\Documents\MATLAB\prog_perri\GMonaLisa.txt";
    begin
        wait for 101*clk_period;
        while not endfile(G) loop
            readline(G, rdline);
            read(rdline, tmp);
            pixel(15 downto 8) <= CONV_STD_LOGIC_VECTOR(tmp,8);
            wait for CLK_period;
        end loop;
        WAIT;
    end process;

process
    variable rdline : line;
    variable tmp : integer;
    file B      : text open read_mode is
"C:\Users\raffy\Documents\MATLAB\prog_perri\BMonaLisa.txt";
    begin
        wait for 101*clk_period;
        while not endfile(B) loop
            readline(B, rdline);
            read(rdline, tmp);
            pixel(24 downto 16) <= '1' & CONV_STD_LOGIC_VECTOR(tmp,8);
            wait for CLK_period;
        end loop;
        pixel(24)<='0';
        WAIT;
    end process;

process
    variable vILINE      : line;
    variable vOLINE      : line;
    variable vSPACE      : character;
    begin
        file_open(file_RESULTS_R, "Routput_results.txt", write_mode);
        file open(file RESULTS G, "Goutput_results.txt", write mode);

```

```

        file_open(file_RESULTS_B, "Boutput_results.txt", write_mode);
--wait for 148*clk_period;
--wait for 158*clk_period;
wait for 161*clk_period; -- post_impl con rgb2gray = '1'
--wait for 151*clk_period; --post_impl con rgb2gray = '0'
for i in 0 to 1023 loop

    write(v_OLINE, pix_out_R, right, 8);
    writeline(file_RESULTS_R, v_OLINE);

    write(v_OLINE, pix_out_G, right, 8);
    writeline(file_RESULTS_G, v_OLINE);

    write(v_OLINE, pix_out_B, right, 8);
    writeline(file_RESULTS_B, v_OLINE);
    wait for clk_period;
end loop;
file_close(file_RESULTS_R);
file_close(file_RESULTS_G);
file_close(file_RESULTS_B);
wait;
end process;

end Behavioral;

```