Prova Finale(Progetto di Reti Logiche)

Prof. William Fornaciari, Prof. Federico Terraneo

Emanuele Bellini(Codice persona 10633744 - Matricola 908179)

POLITECNICO
MILANO 1863

Anno accademico 2020-2021

Indice

1

Introduzione

1.1 Scopo del progetto

Si vuole progettare ed implementare un componente hardware in grado di ricalibrare il contrasto di un' immagine, incrementandolo e distribuendolo su tutto l'intervallo di intensità di una scala di grigi a 256 livelli.

1.2 Specifiche generali

Il modulo da implementare leggerà l'immagine da elaborare sequenzialmente e riga per riga, ricevendo i byte da una memoria nella quale è memorizzata l'immagine. Il byte all'indirizzo 0 indica il numero di pixel per riga dell' immagine, mentre il byte all' indirizzo 1 indica il numero di pixel per colonna. I valori degli n pixel dell'immagine sono scritti sequenzialmente dall' indirizzo 2 in poi, quindi fino all'indirizzo 1+n, occupando un byte per ogni pixel. La dimensione massima dell'immagine da equalizzare è di 128x128 pixel. L' algoritmo di equalizzazione dell'immagine, ispirato al metodo di equalizzazione dell'istogramma, trasforma ogni pixel dell'immagine nel modo seguente:

DELTA_VALUE = MAX_PIXEL_VALUE – MIN_PIXEL_VALUE
SHIFT_LEVEL = (8 – FLOOR(LOG2(DELTA_VALUE +1)))
TEMP_PIXEL = (CURRENT_PIXEL_VALUE - MIN_PIXEL_VALUE) <<SHIFT_LEVEL
NEW_PIXEL_VALUE = MIN(255 , TEMP_PIXEL)

Dove MAX_PIXEL_VALUE e MIN_PIXEL_VALUE, sono il massimo e minimo valore dei pixel dell'immagine, CURRENT_PIXEL_VALUE è il valore del pixel da trasformare, e NEW_PIXEL_VALUE è il valore del nuovo pixel. La nuova immagine equalizzata verrà scritta in memoria immediatamente dopo l'immagine originale, con indirizzo del primo byte pari a 2+(numero_di_colonne*numero_di_righe).

1.3 Interfaccia del componente

Il componente da descrivere ha un'interfaccia così definita:

```
entity project_reti_logiche is
port(
i_clck : in std_logic;
i_rst : in std_logic;
i_start : in std_logic;
i_data : in std_logic_vector( 7 downto 0 );
o_address : out std_logic_vector( 15 downto 0 );
o_done : out std_logic;
o_en : out std_logic;
o_we : out std_logic;
o_data : out_std_logic_vector( 7 downto 0 );
```

Dove:

- i_clk è il segnale di CLOCK in ingresso generato dal TestBench;
- i_rst è il segnale di RESET che inizializza la macchina pronta per ricevere il primo segnale di START;
- i_start è il segnale di START generato dal Test Bench;
- i_data è il segnale (vettore) che arriva dalla memoria in seguito ad una richiesta di lettura;
- o_address è il segnale (vettore) di uscita che manda l'indirizzo alla memoria;
- o_done è il segnale di uscita che comunica la fine dell'elaborazione e il dato di uscita scritto in memoria;
- o_en è il segnale di ENABLE da dover mandare alla memoria per poter comunicare (sia in lettura che in scrittura);
- o_we è il segnale di WRITE ENABLE da dover mandare alla memoria (=1) per poter scriverci. Per leggere da memoria esso deve essere 0;
- o_data è il segnale (vettore) di uscita dal componente verso la memoria.

ESEMPIO:

Esempio del contenuto della memoria al termine di un' elaborazione di un'immagine di dimensione 4x3. I valori che qui sono rappresentati in decimale, sono memorizzati in memoria con l'equivalente codifica binaria su 8 bit senza segno.

DELTA_VALUE = MAX_PIXEL_VALUE – MIN_PIXEL_VALUE = 131-46 = 85
SHIFT_LEVEL = (8 – FLOOR(LOG2(85+1))) = 2

INDIRIZZO MEMORIA	VALORE	COMMENTO
0	4	\\ Byte più significativo numero colonne
1	3	\\ Byte meno significativo numero righe
2	76	\\ primo Byte immagine
3	131	
4	109	
5	89	
6	46	
7	121	
8	62	
9	59	
10	46	
11	77	
12	68	
13	94	\\ ultimo Byte immagine
14	120	\\ primo Byte immagine equalizzata (risultato)
16	252	
17	172	
18	0	
19	255	
20	64	
21	52	
22	0	
23	124	
24	88	
25	192	

1.4 Memoria

La RAM con cui si interfaccia il componente è composta da 2^{16} byte, nello specifico è così descritta:

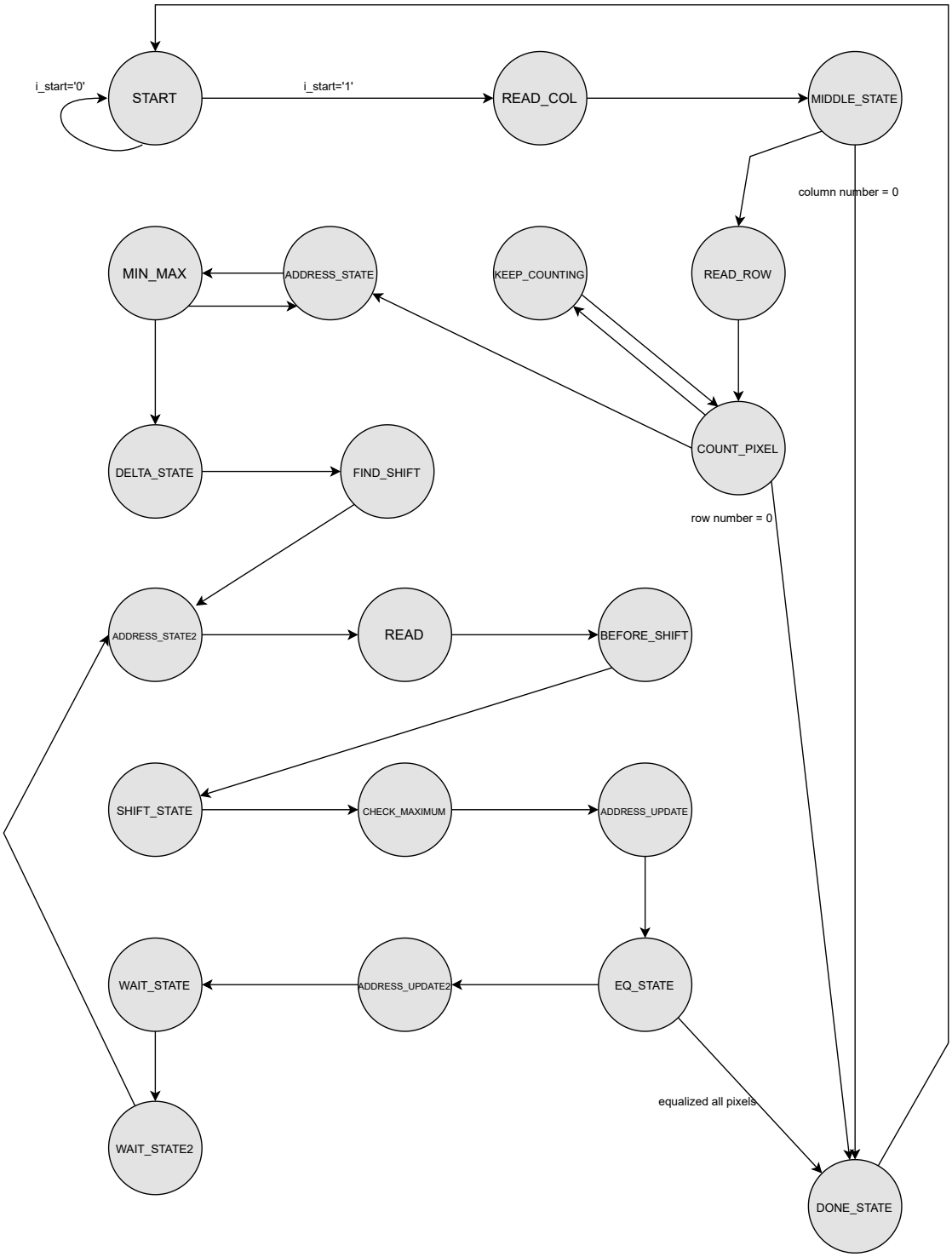
```
-- Single-Port Block RAM Write-First Mode (recommended template)
--
-- File: rams_02.vhd
--
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;
entity rams_sp_wf is
port(
  clk  : in  std_logic;
  we   : in  std_logic;
  en   : in  std_logic;
  addr : in  std_logic_vector(15 downto 0);
  di   : in  std_logic_vector(7 downto 0);
  do   : out std_logic_vector(7 downto 0)
);
end rams_sp_wf;

architecture syn of rams_sp_wf is
type ram_type is array (65535 downto 0) of std_logic_vector(7 downto 0);
signal RAM : ram_type;
begin
  process(clk)
    begin
    if clk'event and clk = '1' then
      if en = '1' then
        if we = '1' then
          RAM(conv_integer(addr)) <= di;
          do                      <= di after 2 ns;
        else
          do <= RAM(conv_integer(addr)) after 2 ns;
        end if;
      end if;
    end if;
  end process;
end syn;
```

2

Architettura

Finché il segnale in ingresso i_start rimane a '0', il componente rimarrà nello stato START senza fare nulla. Quando il segnale in ingresso i_start viene portato a '1', vengono inizializzati i segnali necessari alla computazione ed il componente inizierà l'eleborazione dei dati in memoria, spostandosi dallo stato START allo stato READ_COL. Dopo aver scritto il risultato in memoria, una volta terminata l'elaborazione dell'immagine equalizzata, il componente porta a '1' il segnale 'o_done', per poi tornare nello stato START in attesa che il segnale in ingresso i_start diventi '1'. Infine è importante il ruolo del segnale in ingresso i_rst che, quando viene portato a '1', interrompe la computazione, inizializza i segnali necessari alla computazione e porta il componente nello stato START, il quale rimarrà in attesa del segnale alto i_start per elaborare un'eventuale ulteriore immagine.
I segnali forniti suggeriscono l'utilizzo di una macchina a stati per definire il componente. Di seguito vengono illustrati gli stati della FSM.

2.1 Macchina a stati



2.1.1 START

È lo stato iniziale della macchina. Attende che il test bench porti a '1' il segnale i_start per procedere all'inizializzazione dei segnali necessari e spostarsi poi nello stato READ_COL. Il componente torna su questo stato nel momento in qui il segnale i_rst viene portato a '1' oppure il segnale o_done diventa '1'.

2.1.2 READ_COL

In questo stato viene letto e memorizzato in un registro il numero di colonne. Il segnale o_address viene aggiornato al bit successivo. Lo stato successivo è MIDDLE_STATE.

2.1.3 MIDDLE_STATE

Qui viene controllato che il numero di righe non sia 0: se è zero viene alzato il segnale o_done e il componente torna nello stato START; altrimenti si procede con lo stato successivo, READ_ROW.

2.1.4 READ_ROW

Questo stato, analogo a READ_COL, memorizzerà il numero di colonne dell'immagine e chiederà l'indirizzo in memoria immediatamente successivo a quello attuale.

2.1.5 COUNT_PIXEL

Come prima cosa, qui viene controllato che il numero di colonne non sia 0: se è zero, il segnale o_done diventa '1' e il componente torna nello stato START, altrimenti si procede con la computazione. Questo stato si alternerà allo stato KEEP_COUNTING finché non verrranno contati tutti i pixel dell'immagine. Il numero totale di pixel viene salvato nel registro sum. Nel momento in cui è stato trovato il numero totale di pixel che compone l'immagine, vengono inizializzati i registri min_pixel e max_pixel, necessari per poter trovare e memorizzare il pixel con valore minimo e il pixel con valore massimo; lo stato successivo sarà ADDRESS_STATE.

2.1.6 KEEP_COUNTING

Stato intermedio che funge da contatore per lo stato COUNT_PIXEL.

2.1.7 ADDRESS_STATE

Questo stato funge da contatore per lo stato MIN_MAX, aggiornado il registro count ad ogni iterazione e rimandando il componente allo stato MIN_MAX, chiedendo l'indirizzo di memoria successivo, finché non sono stati letti i valori di ogni pixel.

2.1.8 MIN_MAX

Se il valore del pixel memorizzato in memoria nell'attuale indirizzo è minore di min_pixel, allora min_pixel assumerà il valore di tale pixel; se è invece maggiore di max_pixel, il registro max_pixel assumerà il vaore di quel pixel. Se il valore di count corrisponde a quello di sum vuol dire che sono stati controllati tutti i pixel dell'immagine, in tal caso il componente andrà nello stato DELTA_STATE. Altrimenti tornerà in ADDRESS_STATE.

2.1.9 DELTA_STATE

Count viene reimpostato a 0, il delta value viene calcolato e salvato nel registro delta e viene chiesto l'indirizzo di memoria corrispondente al primo pixel dell'immagine. lo stato successivo è FIND_SHIFT.

2.1.10 FIND_SHIFT

Viene calcolato lo shift per eseguire l'algoritmo, in base al valore del registro delta precedentemente memorizzato. Questo valore viene salvato nel registro shift e il prossimo stato è ADDRESS_STATE2.

2.1.11 ADDRESS_STATE2

Stato intermedio per richiedere l'indirizzo di memoria corrispondente al valore memorizzato in save_address (è il primo pixel dell'immagine alla prima iterazione). Si passa allo stato READ.

2.1.12 READ

Viene memorizzato nel registro read_pixel il valore del pixel contenuto in memoria nell'attuale indirizzo. Si passa allo stato BEFORE_SHIFT.

2.1.13 BEFORE_SHIFT

Viene salvato nel registro eq_pixel la differenza tra il pixel appena letto e il pixel dell'immagine con valore minimo, poi si passa allo stato SHIFT_STATE.

2.1.14 SHIFT_STATE

Il valore di eq_pixel subisce uno shift pari al valore registato nel registro shift, il registro count aumenta di un'unità e si chiede l'indirizzo di memoria pari al valore di sum + il valore di save_address(ovvero il primo indirizzo di memoria libero, sul quale verrà memorizzato il valore del nuovo pixel appena equalizzato). Il prossimo stato è CHECK_MAXIMUM.

2.1.15 CHECK_MAXIMUM

Se il valore del pixel equalizzato supera 255, esso diventa 255. Il prossimo stato è ADDRESS_UPDATE.

2.1.16 ADDRESS_UPDATE

Viene alzato il segnale o_we per poter scrivere in memoria, poi il componente va allo stato EQ_STATE.

2.1.17 EQ_STATE

Il valore del pixel equalizzato viene scritto in memoria al primo indirizzo disponibile. Se count è uguale a sum, quindi sono stati processati tutti i pixel dell'immagine, il prossimo stato sarà DONE_STATE e o_done viene settato a '1'; altrimenti si passa allo stato ADDRESS_UPDATE2.

2.1.18 ADDRESS_UPDATE2

Stadio intermedio per aggiornare il regitro o_address, si passa a WAIT_STATE.

2.1.19 WAIT_STATE

Il registro save_address viene incrementato di 1, per poter salvare in meoria i pixel equalizzati immediatamente dopo i pixel dell'immagine, in maniera contigua. Si passa a WAIT_STATE2.

2.1.20 WAIT_STATE2

o_address diventa uguale a save_address, in modo tale da poter leggere e processare il prossimo bit dell'immagine. Dopodiché si torna a ADDRESS_STATE2.

2.1.21 DONE_STATE

In questo stato termina la computazione e la macchina torna allo stato START.

2.2 Scelte progettuali

La principale scelta progettuale consiste nella descrizione del componente mediante un unico modulo composto da due process:

1. Il primo processo rappresenta la parte sequenziale della macchina e serve per gestire il register transfer e quindi come vengono manipolati i registri.
2. Il secondo analizza i segnali in ingresso e lo stato corrente della FSM per determinare lo stato successivo in cui il sistema evolverà, quindi rappresenta la parte combinatoria della macchina.

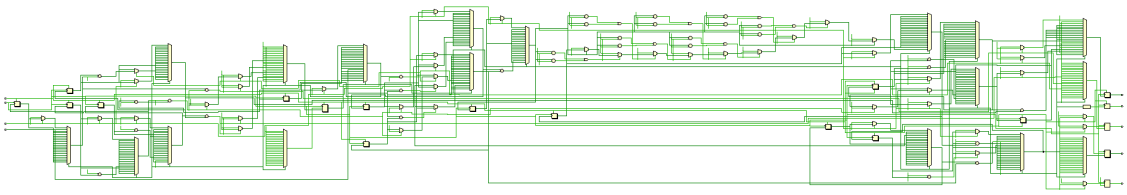
Le operazioni logiche utilizzate nei process sono l'addizione e la sottrazione tra due registri, lo SHIFT su un registro e gli orperatori relazionali per comparare i valori dei registri.

3

Risultati sperimentali

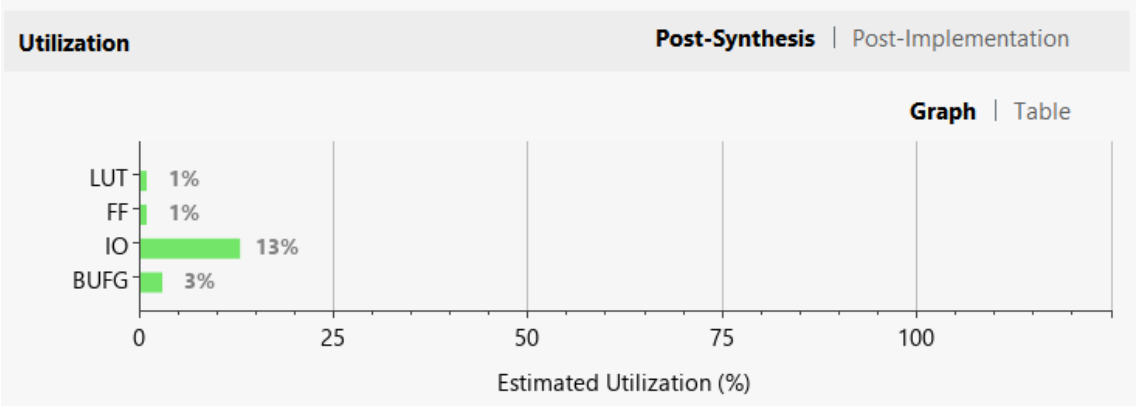
3.1 Schematic

Viene di seguito riportato lo schematic del componente sintetizzato, elaborato da Vivado:



3.2 Report di sintesi

Di seguito sono riportati i principali dati ottenuti in post sintesi:
-Report di utilizzo:



Site Type	Used	Fixed	Available	Util%
Slice LUTs*	391	0	134600	0.29
LUT as Logic	391	0	134600	0.29
LUT as Memory	0	0	46200	0.00
Slice Registers	171	0	269200	0.06
Register as Flip Flop	171	0	269200	0.06
Register as Latch	0	0	269200	0.00
F7 Muxes	0	0	67300	0.00
F8 Muxes	0	0	33650	0.00

-Report temporale:

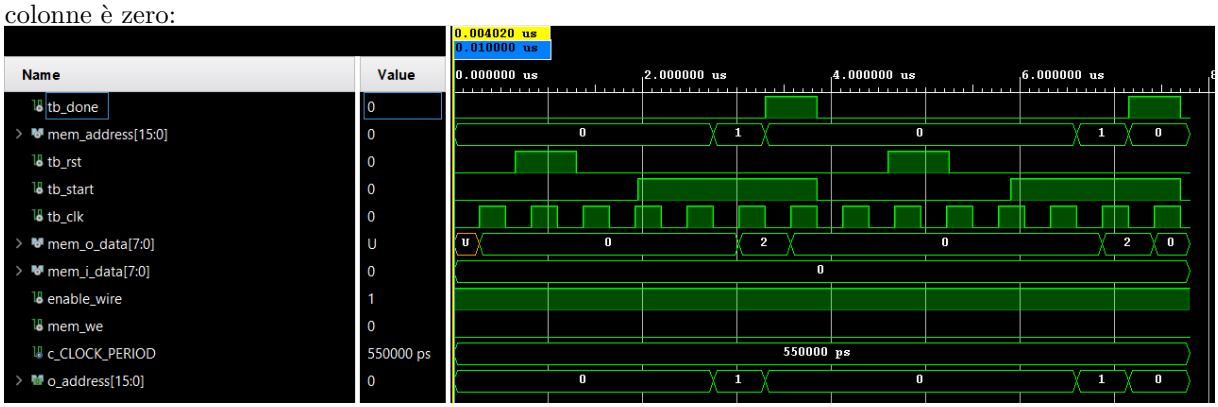
```
Timing Report

Slack (MET) :             89.914ns  (required time - arrival time)
  Source:                 FSM_sequential_cur_state_reg[2]/C
                            (falling edge-triggered cell FDCE clocked by clock  {rise@0.000ns fall@50.000ns period=100.000ns})
  Destination:            eq_pixel_reg[7]/D
                            (falling edge-triggered cell FDCE clocked by clock  {rise@0.000ns fall@50.000ns period=100.000ns})
  Path Group:             clock
  Path Type:              Setup (Max at Slow Process Corner)
  Requirement:            100.000ns  (clock fall@150.000ns - clock fall@50.000ns)
  Data Path Delay:        9.938ns  (logic 3.847ns (38.710%)  route 6.091ns (61.290%))
  Logic Levels:           14  (CARRY4=4 LUT3=1 LUT5=7 LUT6=2)
  Clock Path Skew:        -0.145ns (DCD - SCD + CPR)
    Destination Clock Delay (DCD):    2.100ns = ( 152.100 - 150.000 )
    Source Clock Delay      (SCD):    2.424ns = ( 52.424 - 50.000 )
    Clock Pessimism Removal (CPR):    0.178ns
  Clock Uncertainty:      0.035ns  ((TSJ^2 + TIJ^2)^1/2 + DJ) / 2 + PE
    Total System Jitter     (TSJ):    0.071ns
    Total Input Jitter      (TIJ):    0.000ns
    Discrete Jitter          (DJ):    0.000ns
    Phase Error              (PE):    0.000ns
```

4

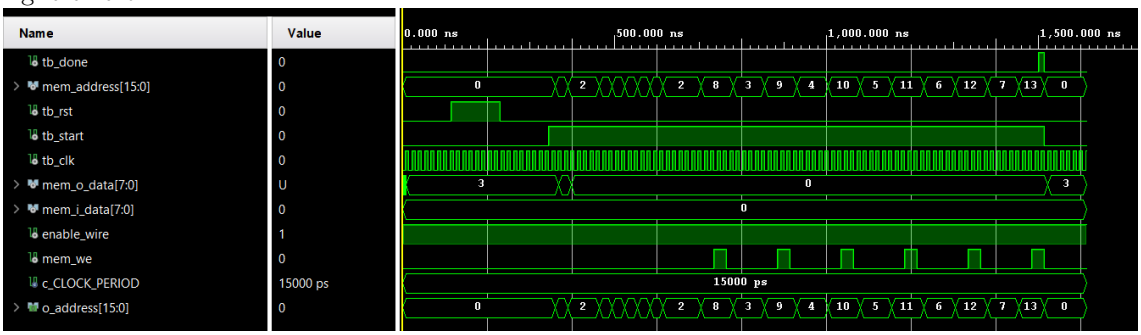
Simulazioni

Per verificare il corretto funzionamento del componente, esso è stato sottoposto a molteplici simulazioni effettuate con diversi test bench, utilizzando i test bench di base che sono stati forniti e generandone di nuovi per verificarne il corretto comportamento nel maggior numero di situazioni possibile. Inoltre sono stati utilizzati test bench ad-hoc per poter verificare che il componente funzionasse correttamente anche nei corner cases.
Di seguito vengono riportati la descrizione del test bench e il relativo comportamento del componente (in post-synthesis simulation functional), di qelli più significativi.

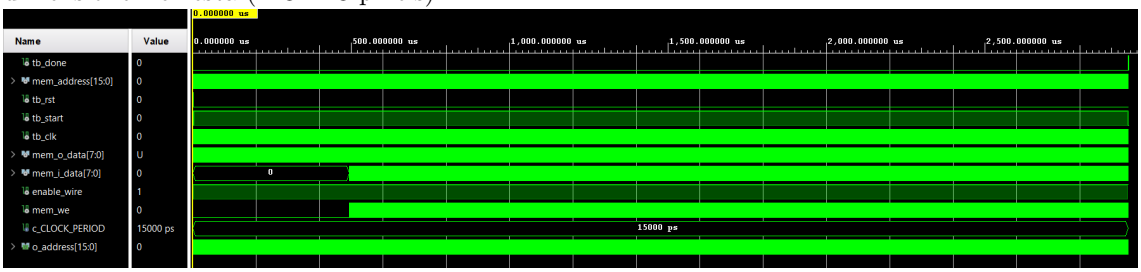
- Zero Colonne
Il test verifica che il componente si comporti correttamente quando il numero di colonne è zero:



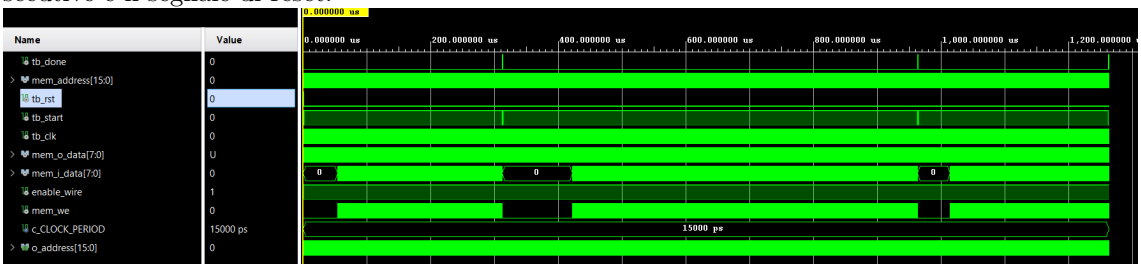
- Zero Righe
Il test verifica che il componente si comporti correttamente quando il numero di righe è zero:



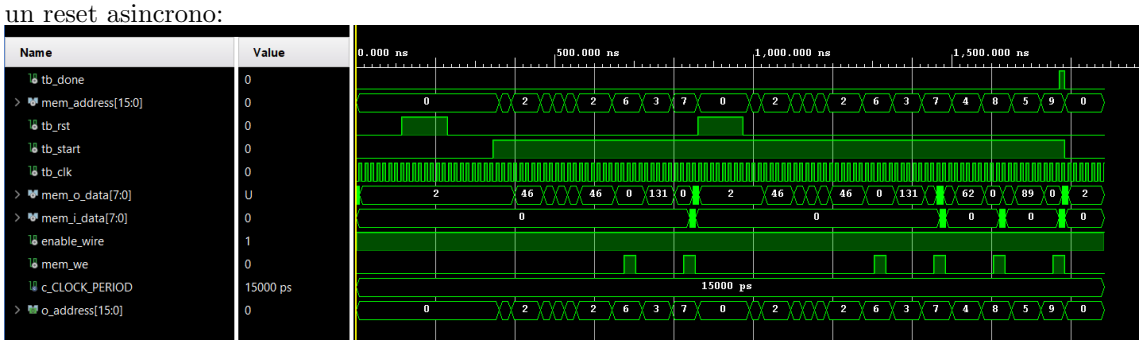
- Dimensione massima dell'immagine
Il test verifica che il componente riesca ad elaborare un' immagine della massima dimensione richiesta (128x128 pixels):



- Più immagini consecutive e reset
Il test verifica che il componente riesca a gestire correttamente più immagini consecutive e il segnale di reset:



- Reset asincrono
Il test verifica che l'immagine venga processata correttamente anche in presenza di un reset asincrono:



Infine è stato utilizzato un generatore random di casi di test per poter testare ulteriormente il componente.

5

Conclusioni

Il componente è risultato funzionante per tutti i test eseguiti, in pre sintesi e in post sintesi. Per sintetizzarlo sono stati utilizzati 391LUTs e 171FFs. Il componente riesce ad equalizzare correttamente immagini in scala di grigi, di dimensione massima 128x128 pixels, secondo il metodo dell'equalizzazone dell'istogramma, salvando i nuovi pixel equalizzati in memoria.