POLITECNICO
MILANO 1863

# Progetto di Reti Logiche

William Zeni
matricola 10613915

Cristina Urso
matricola 10599689

Anno 2020/21

Progetto sostenuto presso il Politecnico di Milano - Dipartimento di Elettronica, Informazione e Bioingegneria. Corso diretto dal Prof. Gianluca Palermo.

# Indice

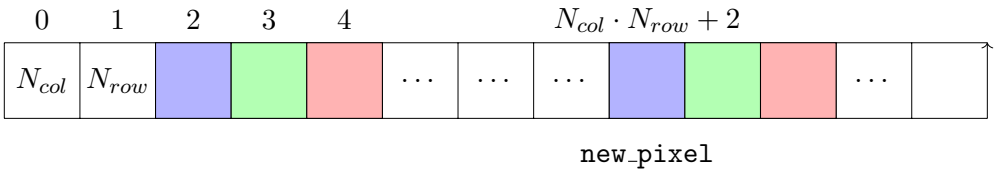
# 1 Introduzione

## 1.1 Scopo del progetto

Lo scopo del progetto è quello di creare un componente hardware sintetizzabile, in grado di equalizzare un immagine. L'algoritmo che ne prevede l'equalizzazione si ispira ad una versione semplificata del metodo di equalizzazione dell'istogramma, il quale applica un aumento nel contrasto di un immagine su scala di grigi. In generale l'elaborazione digitale dell'immagine risulta più evidente specialmente quando i dati raccolti sono rappresentati da valori di intensità molto vicini. Per cui, se una immagine contenesse una scala di grigi molto ampia, l'effeto dell'equalizzazione risulterebbe pressochè nullo.

## 1.2 Specifiche generali

Si definisca un immagine dalle dimensioni variabili, ma di massimo $128 \times 128$, e si definisca una memoria lineare nella quale i primi due valori siano le dimensioni dell'immagine e i restanti i valori assegnati ad ogni pixel, il componente hardware dovrà scrivere in coda alla memoria l'immagine equalizzata pixel per pixel. Il risultato sarà una memoria complessivamente lunga $2 \cdot N_{col} \cdot N_{row} + 2$, dove $N_{row}$ e $N_{col}$ sono rispettivamente il numero di righe e di colonne dell'immagine. A partire dalla posizione 2, per ogni pixel si avrà il corrispettivo pixel equalizzato ad una distanza $N_{col} \cdot N_{row}$ come mostrato in figura.




La memoria dialogherà in stretto contatto con il componente attraverso due segnali, che determineranno l'avvio della computazione, la sua terminazione e l'eventuale ripartenza. Si noti che la computazione di una singola immagine, una volta iniziata, non potrà mai essere interrotta, ma rimane possibile la computazione di più immagini.

Ogni pixel avrà un valore compreso tra 0 e 255 e verrà rielaborato dal componente nel seguente modo:

```
1: delta_value ← max_pixel_value − min_pixel_value
2: shift_level ← 8 − ⌊log2(delta_value + 1)⌋
3: temp_pixel ← curr_pixel_value − min_pixel_value
4: temp_pixel ← temp_pixel << shift_level
5: if temp_pixel > 255 then
6:     new_pixel ← 255
7: else
8:     new_pixel ← temp_pixel
9: end if
```

dove `max_pixel_value` e `min_pixel_value` sono rispettivamente il valore massimo e il valore minimo del pixel trovati all'interno dell'immagine, `curr_pixel_value` è il valore del pixel preso in considerazione e `new_pixel` è il valore del pixel da scrivere in memoria.

### 1.3 Interfaccia del componente

```
entity project_reti_logiche is
port (
    i_clk      : 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)
);
end project_reti_logiche;
```

L'interfaccia del componente, per potersi relazionare con la memoria, deve essere come quella sopra riportata. I segnali di input vengono evidenziati dal prefisso `'i_'`, mentre per i segnali di output è presente il prefisso `'o_'`. Nel dettaglio:

- **i_clk** scandisce il ciclo di clock della memoria, registrando '1' sul `rising_edge`.
- **i_rst** è il segnale di `RESET`, viene chiamato all'inizio della computazione di una o più immagini. Se si registra '1' su questo segnale la computazione deve essere terminata e ripristinata al momento di partenza.
- **i_start** determina l'inizio della coputazione. Questo segnale rimane alto fino a quando la computazione non termina. La computazione può iniziare solo se il segnale è basso.
- **i_data** raccoglie i dati inviati dalla memoria.
- **o_address** è il segnale attraverso il quale si possono fare richieste alla memoria. Sostanzialmente su questo segnale si pone l'indirizzo della memoria che si vuuole leggere o sulla quale si vuole scrivere.
- **o_done** determina la fine della computazione. Il segnale deve essere portato alto quando si vuole terminare la computazione e deve rimanere alto fino a quando la memoria non porti il segnale di `START` basso. Solo in quel caso il segnale viene riportato basso.
- **o_en** abilita la memoria alla lettura o alla scrittura.
- **o_we** abilita la memoria alla scrittura. Quando questo segnale è alto la memoria si aspetta un dato in ingresso da scrivere.
- **o_data** è il dato che la memoria si aspetta di leggere quando la scrittura è attiva.

## 1.4 Dati e Descrizione memoria

**: Fig.1**

```
architecture projecttb of project_tb is
  constant   c_CLOCK_PERIOD    : time := 15 ns;
  signal     mem_address       : std_logic_vector (15 downto 0);
  signal     mem_o_data        : std_logic_vector (7 downto 0);
  signal     mem_i_data        : std_logic_vector (7 downto 0);
  signal     tb_done           : std_logic;
  signal     tb_rst            : std_logic;
  signal     tb_start          : std_logic;
  signal     tb_clk            : std_logic;
  signal     enable_wire       : std_logic;
  signal     mem_we            : std_logic;

type ram_type is array (65535 downto 0) of std_logic_vector(7 downto 0);
```

**: Fig. 2**

```
begin
UUT: project_reti_logiche
port map (
          i_clk      => tb_clk,
          i_start    => tb_start,
          i_rst      => tb_rst,
          i_data     => mem_o_data,
          o_address  => mem_address,
          o_done     => tb_done,
          o_en       => enable_wire,
          o_we       => mem_we,
          o_data     => mem_i_data
          );
```

La memoria ha un'interfaccia che dialoga direttamente con il componente sintetizzato come indicato dalla *port map* rappresentata in `Fig.2`. È indirizzabile al *byte* e ha una dimensione massima di $N_{col} * N_{row} * 2 + 2$ dove nel caso pessimo $N_{col}$ e $N_{row}$ hanno un valore di 128 *byte*. Pertanto, la dimensione massima supportabile dalla memoria deve essere di $2^7 * 2^7 * 2 + 2$, ovvero di 32770 *byte*. Come si può notare dalla definizione della sua entity (`Fig.1`), per evitare eventuali errori, si è deciso di dedicare al componente uno spazio di $2^{16}$ *byte*. Ogni singolo *byte* è definito a sua volta da un *std_logic_vector* di 8 *bit* di tipo `unsigned` e può avere un valore compreso tra 0 e 255, dove 0 rappresenta il pixel nero e 255 rappresenta il pixel bianco. Qualora fosse prevista l'elaborazione di più immmagini, vi sarà la necessità di definire più memorie in cascata.
La costante `c_CLOCK_PERIOD` è quella che determina il ciclo di *clock* del componente e deve essere mantenuta per specifica sotto i 100 *ns*.

# 2 Desing Pattern

## 2.1 Scelte Progettuali

La struttura del progetto è stata suddivisa in due process principali: `UPDATE` e `STATES`. Il primo ha il compito di relazionarsi con la memoria e il secondo contiene gli stati della macchina impiegati nella equalizzazione delle immagini. In questo modo una porzione di codice è adibita esclusivamente alla computazione dei pixel (process `STATES`), mentre la rimanente si occupa dei segnali di output della *entity* (process `UPDATE`).
Per permette un corretto dialogo tra i due process è nata l'esigenza di avere dei segnali "duplicati". I segnali con suffisso `'_cp'` sono stati introdotti per mantere in memoria i valori computati, mentre i segnali con suffisso `'_next'` sono stati implementati allo scopo di permettere agli stati di far richieste alla memoria. Nel particolare, per ogni ciclo di clock, durante il *rising_edge*, il process `UPDATE` si risveglia, aggiornando i segnali. I segnali contenenti i valori da mantenere vengono reimpostati con i segnali `'_cp'`, mentre i segnali di output della *entity* sono aggiornati con i segnali `'_next'`.
Il process `STATES` è a tutti gli effetti un `FSM` (*Macchina a Stati Finiti*) rappresentabile attraverso lo schema al paragrafo 2.3. Il suo funzionamento è permesso sostanzialmente da tre segnali a lui dedicati: `next_state`, che identifica la chiamata allo stato prossimo, `curr_state`, che rileva lo stato corrente, e `prev_state`, che conserva lo stato precedente.

## 2.2 Elenco Stati - Process STATES

### 2.2.1 START

Lo stato di `START` è il primo stato del process `STATES` ed è stato pensato come stato di attesa iniziale. Questo stato viene invocato in due situazioni differenti: se il segnale di `i_rst` viene portato alto, oppure quando il segnale `i_start` viene riportato basso dopo la computazione di un immagine. Lo stato `START` non cambia fino a quando il segnale `i_start` non viene portato alto. In quel momento lo stato successivo viene impostato a `INIT`.

### 2.2.2 INIT

`INIT` è uno stato di transizione nel quale il processore si assicura che i segnali siano inizializzati con i valori opportuni. Successivamente imposta lo stato prossimo a `ABILIT_READ`.

### 2.2.3 ABILIT_READ

Attraverso `ABILIT_READ` si abilita la memoria alla sola lettura. Viene richiamato in due momenti diversi del progetto: all'inizio della computazione, subito dopo `INIT`, per portare il segnale `o_en` a 1 e permettere agli altri stati di poter leggere dalla memoria, oppure dopo la scrittura di un pixel, al fine di disabilitare l'accesso alla scrittura. In base allo stato chiamante, instrada lo stato prossimo a quello opportuno.

### 2.2.4 ABILIT_WRITE

Lo stato `ABILIT_WRITE` abilita la memoria alla scrittura. Viene invocato subito dopo aver computato il valore del nuovo pixel e in nessun altro momento. Instrada poi lo stato prossimo a `WRITE_PIXEL`.

### 2.2.5 WAIT_MEM

`WAIT_MEM` è uno stato centrale durante la gestione del flusso di dati. La sua funzione è quella di far "sprecare" un ciclo di clock al processore. Ciò ci assicura sia in caso di scrittura, sia in caso di lettura, che i segnali in ingresso e in uscita siano letti o scritti correttamente. Nello specifico, questo stato evita che lo stato successivo lavori con dati relativi allo stato precedente. In base allo stato chiamante instrada lo stato prossimo in modo opportuno.

**N.B:** *Alcune chiamate a questo stato potevano essere evitate. Questa informazione è emersa durante lo stress test a cui il processore è stato sottoposto. Tuttavia, è stato scelto di forzare ugualmente l'attesa per ovviare eventuali errori e per permettere la corretta acquisizione dei dati indipendentemente dal periodo di clock scelto. Ciò dovrebbe permettere una maggior robustezza, sebbene un aumento nella latenza della computazione.*

### 2.2.6 GET_RC

`GET_RC` si occupa della lettura dei primi due pixel della memoria in cui sono contenute le dimensioni dell'immagine da modificare. Viene invocato dopo l'abilitazione della memoria alla lettura e aggiorna i segnali `n_col` e `n_row`. Lo stato richiama se stesso fino a quando non ha aggiornato i due segnali, altrimenti imposta lo stato prossimo a `GET_DIM`.

### 2.2.7 GET_DIM

Lo stato `GET_DIM` si preoccupa di aggiornare il segnale `dim_address` con l'indirizzo del primo *byte* libero per la scrittura. Il calcolo che ne determina il valore è: $n\_col \cdot n\_row + 2$. `GET_DIM` verifica inoltre il caso speciale in cui una delle dimensioni dovesse essere nulla. In quel caso impone lo stato prossimo a `DONE`.

**N.B:** *La moltiplicazione usa segnali std_logic_vector da* 8 *bit e produce un risultato su* 16 *bit. Essendo* `dim_address` *un std_logic_vector su* 16 *bit, il problema non si pone.*

### 2.2.8 READ_PIXEL

`READ_PIXEL` è uno stato strettamente connesso con lo stato `GET_MINMAX`. Richiede alla memoria il valore del pixel da leggere e aggiorna il segnale `curr_address` a quello successivo. In questo modo il valore del pixel sarà disponibile sul segnale `i_data` dopo due *rising_edge*. L'attesa è sempre lasciata allo stato `WAIT_MEM`.

### 2.2.9 GET_MINMAX

`GET_MINMAX` aggiorna i segnali `max_pixel_value` e `min_pixel_value`, inizializzati rispettivamente con 0 e 255, con il valore di `i_data`. Questo avviene se e solo se si verifica che `i_data` è *maggiore* di `max_pixel_value` (in questo caso aggiorna `max_pixel_value`) oppure se `i_data` è *minore* di `min_pixel_value` (in questo caso aggiorna `min_pixel_value`). Infine, imposta lo stato prossimo a `READ_PIXEL`

### 2.2.10 GET_DELTA

`GET_DELTA` è uno stato banale. Calcola la *differenza* tra `max_pixel_value` e `min_pixel_value` e imposta lo stato prossimo a `WAIT_MEM`.

### 2.2.11 CALC_SHIFT

Lo stato `CALC_SHIFT` si preoccupa di effettuare la riga 2 dello speudo-codice riportato al paragrafo 1.2. Per effettuare il calcolo, richiama se stesso (attraverso lo stato `WAIT_MEM`) aggiornando dei segnali d'appoggio. Nel dettaglio si potrebbe semplificare il tutto con il seguente pseudo-codice:

```
1: m ← delta_value + 1
2: k ← −1
3: t ← 1
4: while t ≤ m do
5:     k ← k + 1
6:     t ← t ∗ 2
7: end while
8: shift_level ← 8 − k
```

Sostanzialmente attraverso il ciclo `WHILE` si ricava $k$, il quale non è altro che il valore di $\lfloor log_2(delta\_value + 1) \rfloor$. Questa procedura è forse quella più dispendiosa a livello di tempo e memoria, poichè per aggiornare ogni segnale si deve come minimo attendere due cicli di clock. Inoltre, la moltiplicazione tra due interi ($t$ e 2) produce un risultato su 64 *bit* che VHDL gestisce troncando a 32 *bit*. Per le specifiche del progetto, ci si aspetta che $k$ non possa assumere un valore maggiore di 8. Pertanto, si è certi che una variabile a 32 *bit* possa largamente contenere i valori desiderati.

### 2.2.12 GET_PIXEL

`GET_PIXEL` è uno stato omonimo di `READ_PIXEL`. La principale differenza del primo con il secondo è che mentre `READ_PIXEL` effettua dei controlli sul flusso della computazione, `GET_PIXEL` richiede solamente alla memoria il dato da leggere e imposta lo stato prossimo a `WAIT_MEM`.

### 2.2.13 CALC_NEWPIXEL

In `CALC_NEWPIXEL` si calcola il nuovo valore del pixel da scrive in memoria. Per farlo, viene utilizzata una variabile di appoggio `pixel_to_shift` definita come uno *std_logic_vector* da 16 *bit* e inizializzata come `i_data`−`min_pixel_value` con l'aggiunta di otto '0' in posizione più significativa. La variabile viene, quindi, shiftata e su di essa viene fatto il seguente controllo: se risulta essere minore di 255 allora aggiorniamo il segnale `new_pixel` al suo valore in *std_logic_vector* su 8 *bit*, altrimenti il segnale `new_pixel` viene impostato a 255 sempre in *std_logic_vector* su 8 *bit*.

### 2.2.14 WRITE_PIXEL

Lo stato `WRITE_PIXEL` si assicura che il valore del nuovo pixel calcolato sia scritto in memoria, incrementa tutti i contatori e controlla di non essere arrivato al termine della computazione. Il controllo usufruisce della varibile intera `last`, che assume il valore di $2 * \texttt{n\_col} * \texttt{n\_row} + 2$, ovvero l'indice dell'ultimo pixel scrivibile. Quando il contatore assume questo valore lo stato successivo viene impostato a `DONE` (sempre attravero `WAIT_MEM`), altrimenti viene impostato lo stato prossimo a `ABILIT_READ`, dove la scrittura viene disabilitata e viene riabilitata la lettura per il pixel successivo. Anche qui, si noti che il calcolo di `last` è sicuramente contenibile in una varibile a 32 *bit*.

### 2.2.15 DONE

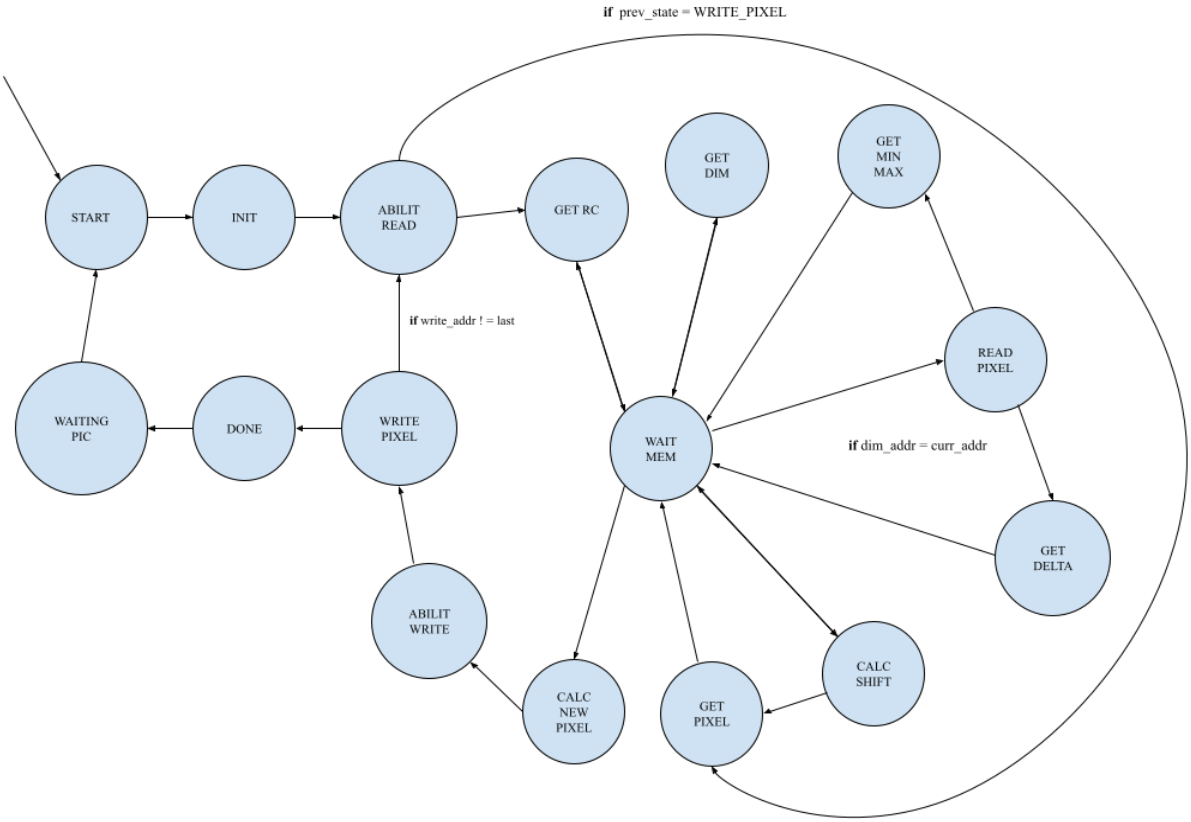
`DONE` chiude tutte le comunicazioni con la memoria, riportando a '0' sia `o_en` che `o_we` e pulendo il buffer di `o_address` e `o_data`. Successivamente il segnale `o_done` viene portato ad '1' e lo stato prossimo è impostato a `WAITING_PIC`.

### 2.2.16 WAITINGPIC

Ultimo stato del processo. Ha lo scopo di restare in attesa che il segnale di `i_start` venga riportato a '0'. Finche ciò non avviene, `WAITING_PIC` richiama se stesso. Quando sia `i_start` che `o_done` hanno il segnale a '0', imposta lo stato prossimo a `START` facendo ricominciare l'intera compilazione.

## 2.3 Approfondimento sul process STATES

All'interno del process `STATES` si possono notare due macro momenti della computazione. Il primo è quello descritto dalla prima lettura completa della memoria (da `START` a `GET_PIXEL`), attraverso la quale si determinano tutti i valori necessari per la computazione del nuovo pixel. Si noti inoltre che all'interno del primo momento abbiamo due cicli ben distinti: uno per il calcolo del *max_value* e *min_value* e uno per il calcolo dello *shift_level*. Il secondo momento (fino a `DONE`), invece, è descritto dalla seconda lettura semi-completa della memoria (difatto si esclude la lettura dei primi 2 *byte*) ed è il momento nel quale il pixel appena letto viene rielaborato e salvato in memoria nel *byte* opportuno. Questa doppia lettura della memoria, sebbene aumenti la complessità temporale, è strettamente necessaria, per via del calcolo del *delta*. Iniziamente, si era pensato di ottimizzare il secondo momento con due proeccess paralleli: il primo avrebbe elaborato i pixel pari, mentre il secondo i pixel dispari, e sarebbero stati gestiti da opportuni 'semafori'. Tuttavia, la struttura attuale è risultata essere molto più stabile nei test, permettendoci anche di variare a nostra discrezione il tempo di clock della computazione.



# 3 Risultati dei Test

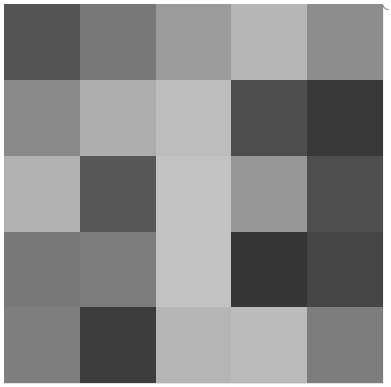
I test eseguiti sono stati pensati appositamente per evitare eventuali errori di programmazione, cercando di coprire il maggior numero di casi possibili. Per farlo, non solo sono stati creati dei test manualmente (dal `N.1` al `N.5` ), ma si è fatto ricorso sia al linguaggio `C` sia al linguaggio `Python` per la creazione di veri e propri generatori di test. Entrambi i generatori sono in grado di creare un numero di immagini (di dimensione diversa) in base alla decisione dell'utente.
I test creati dai generatori sono stati a loro volta controllati manualmente. Difatto, dal controllo manuale, è emerso che le funzioni delle librerie randomiche nei due linguaggi, essendo uniformemente distribuite, rendevano praticamente certa la presenza dei valori 0 e 255 in una qualsiasi immmagine dalle dimensioni maggiori di 20x20. Immagini che hanno all'interno valori molto distanti tra loro rendono l'equalizzazione dell'immagine inutile. Un *delta* molto grande porta inevitabilmente ad uno *shift* nullo, il che significa che l'immagine non viene equalizzata ma semplicemente copiata. Per evitare, quindi, che i test fossero dei semplici copia-incolla, si è deciso di adottare una distribuzione Gaussiana, avendo così immagini con *shift* di livello 8 molto più probabili di quelli con *shift* nullo. Dopo questa verifica e vista la semplicità nella scrittura di quest'algoritmo in un linguaggio ad alto livello si è supposta veritiera la correttezza dei test prodotti dai generatori.
I test in questione hanno ottenuto i seguenti risultati:

| N. | Nome Test | Descrizione | Tempi |
|---|---|---|---|
| 1 | tb_2x2.vhd | Test base con una immagine 2x2 | 1487500 ps |
| 2 | tb10.vhd | Test con 10 immagini 2x2 o 4x1, con caso particolare immagine con una dimensione nulla. Questo test usa tutti i valori di *shift* possibili. | 12392500 ps |
| 3 | tb5_delta.vhd | Test con 5 immagini 3x2 con valori di pixel molto vicini tra loro e caso particolare immagine 1x1. | 23522500 ps |
| 4 | tbreset.vhd | Test con 8 immagini 2x2 nel quale viene chiamato il segnale `i_rst` a fine di ogni compilazione d'immagine. | 1021300 ps |
| 5 | tbAsync.vhd | Test con 8 immagini 2x2 nel quale viene chiamato il segnale `i_rst` ogni due o ogni tre immagini. | 1021300 ps |
| 6 | tb128.vhd | Test con una singola immagine 128x128. Questo test è stato ottenuto con un generatore appositamente creato. | 2212818 ns |
| 7 | tbHARD.vhd | Stress test con 100 immagini 128x128 senza chiamata al segnale `i_rst`. Questo test è stato ottenuto con un genaratore reso disponibile da Davivde Merli su Github: https://github.com/davidemerli/RL-generator-2020-2021.git | 405796650 ns |

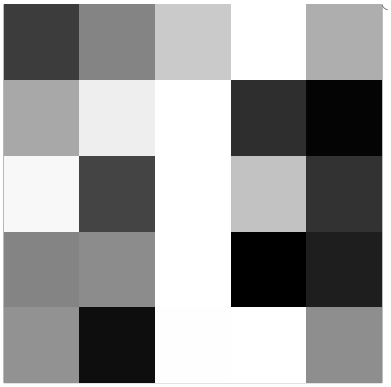
In realtà sono stati fatti ulteriori test, con diverso numero di immagini, di varie dimensioni e con *reset* "casuale", tuttavia riportare ulteriori test a quelli presentati risulterebbe ridondante. Si noti inoltre che, per verificare la robustezza del codice, il periodo di *clock* è stato impostato sempre in modo casuale per un qualsiasi valore compreso tra l' 1.1 *ns* e i 100 *ns*. Il periodo di *clock* minimo supportato dal nostro processore è risultato essere 1.1 *ns*. Per completezza i test sono stati fatti sia in pre-sintesi che in post-sintesi.

### 3.1 Esempi di equalizzazione

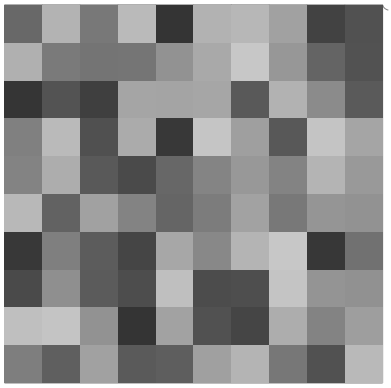
Lo scopo del progetto era quello di equalizzare un'immagine con il metodo dell'istogramma. Pertanto, potrebbe essere totalizzante verificare in maniera grafica che il processore elabori in maniera opportuna le immagini fornitogli. In allegato è stata proposta l'elaborazione di tre immagini di diversa risoluzione. Le immagini sono state create attraverso il generatore di test e sono state equalizzate modificando il progetto `VHDL` per ottenere su file testuale i pixel rielaborati. Successivamente, attravero un programma in `C`, i dati sono stati raccolti per la creazione delle immagini in `LaTeX`.



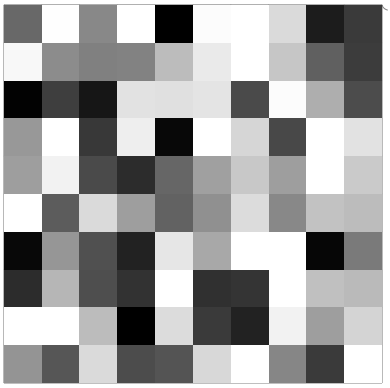
Pre-equalizzazione 5x5



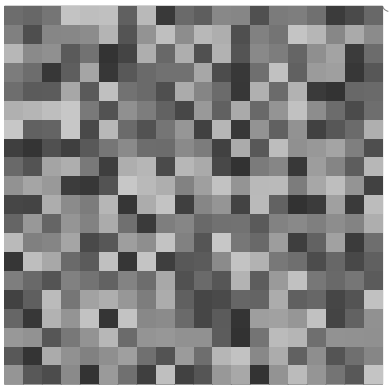
Post-equalizzazione 5x5



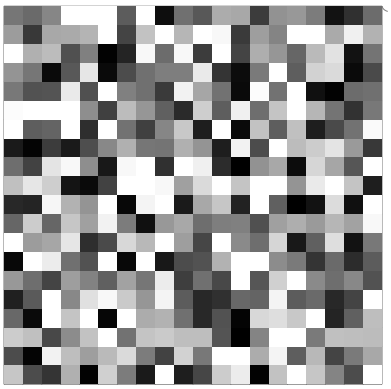
Pre-equalizzazione 10x10



Post-equalizzazione 10x10



Pre-equalizzazione 20x20



Post-equalizzazione 20x20

# 4 Conclusioni

Write somenthing here