# Prova Finale Progetto di Reti Logiche

Roger Aldair Veliz Sedano
Codice Persona: 10614456

Prof. Gianluca Palermo - A.A. 2020-2021

## Resume

Lo scopo del progetto è quello di realizzare un componente HW in VHDL che data un'immagine in ingresso, trasformi ogni pixel dell'immagine seguendo un algoritmo semplificato di equalizzazione.
Per calcolare il nuovo valore del pixel da trasformare è necessario prima calcolare il valore massimo e minimo dei pixel perciò per si farà un prima lettura di tutti i pixel per trovare il minimo e il massimo, trovati questi valori si esegue un seconda lettura dei pixel per poterli trasformare uno ad uno.

# 1 Specifica

L'algoritmo semplificato di equalizzazione è il 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.

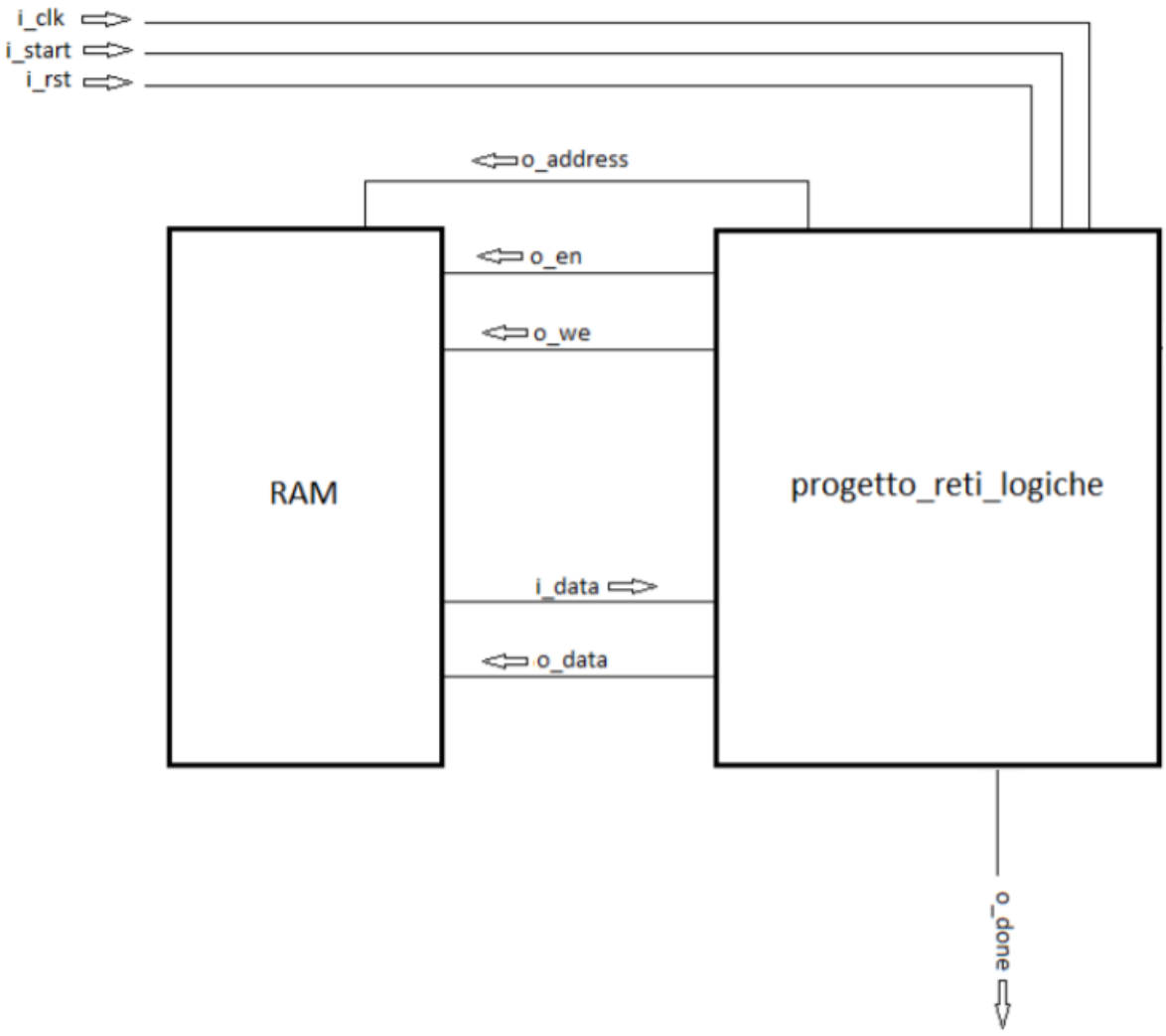
Mentre l'interfaccia del componente è definita come segue:

```
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;
```

Nello specifico:

- **i_clk** è il segnale di CLOCK in ingresso generato dal test bench;
- **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 che fa partire l'elaborazione;
- **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** 'e il segnale di WRITE ENABLE da alzare a 1 per poter scriverci. Per leggere da memoria esso deve essere 0;
- **o_data** 'e il segnale (vettore) di uscita dal componente verso la memoria.



# 2 Scelte Progettuali

Per prima cosa ho definito l'algoritmo da usare, il quale consisteva in due letture dei pixel dell'immagine. La prima lettura serviva per memorizzare la dimenzione delle righe, delle colonne, e per trovare il minimo e il massimo valore dei pixel letti.

Dopo aver trovato il minimo e il massimo, si calcola il valore di "shift" con dei controlli a soglia sul **DELTA_VALUE**(max-min). Per i controlli a soglia ho optato per uno switch-case essendo 8 i possibili valori che "shift" può avere.

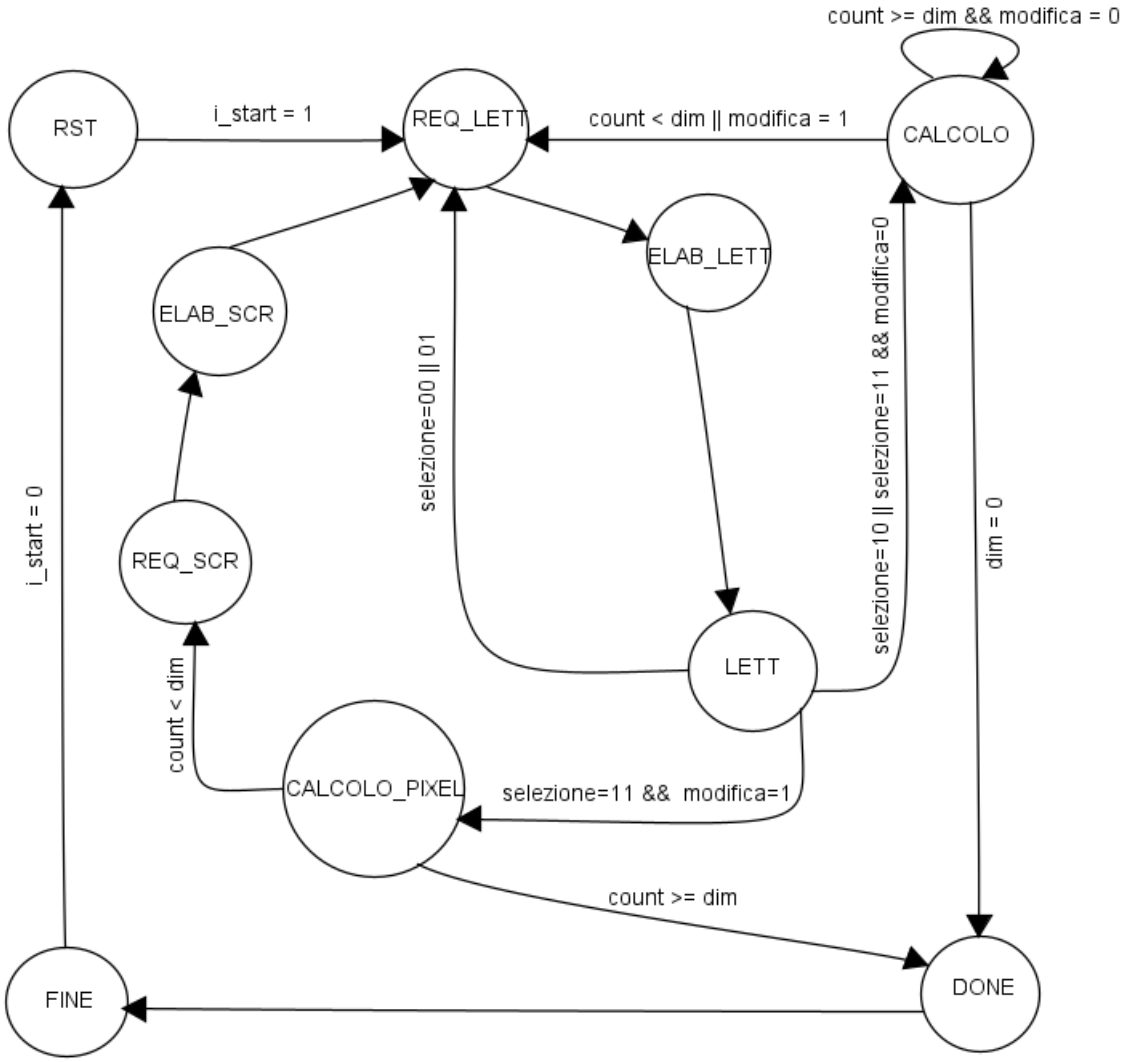
Trovato il valore di **"shift"** si prosegue con una seconda lettura nel quale si calcola il valore del nuovo pixel "shiftando" il vecchio valore e scrivendolo in memoria subito dopo la fine dei pixel originali.
Per entrambe le letture ho usato un contatore per tenere traccia di fine lettura e quindi di fine elaborazione(fine della seconda lettura).

Questa soluzione è stata implementata attraverso una FSM che scandisce i vari stati che corrispondono agli step dell'algoritmo illustrato sopra. Il componente è descritto con un singolo processo.

## 2.1 Segnali interni

| NOME | TIPO | VAL. INIZIALE | DESCRIZIONE |
|---|---|---|---|
| STATE_CURR | STATUS | U | stato corrente |
| MEM_ADDRESS | std_logic_vector (15 downto 0) | (others => '0') | indirizzo di memoria da accedere |
| count | std_logic_vector (15 downto 0) | (others => '0') | contatore numero di pixel letti |
| dim | std_logic_vector (15 downto 0) | (others => '0') | numero totale di pixel |
| temp_pixel | std_logic_vector (15 downto 0) | (others => '0') | nuovo valore pixel |
| min | std_logic_vector (7 downto 0) | (others => '1') | valore pixel minimo |
| max | std_logic_vector (7 downto 0) | (others => '0') | valore pixel massimo |
| current | std_logic_vector (7 downto 0) | (others => '0') | valore pixel corrente |
| dim_x | std_logic_vector (7 downto 0) | (others => '0') | numero righe immagine |
| dim_y | std_logic_vector (7 downto 0) | (others => '0') | numero colonne immagine |
| delta_value | std_logic_vector (7 downto 0) | (others => '0') | max - min |
| shift | std_logic_vector (3 downto 0) | (others => '0') | indica di quanto shiftare |
| selezione | std_logic_vector (2 downto 0) | (others => '0') | registro utilizzato per gestire il salvataggio dei dati |
| modifica | std_logic | (others => '0') | indica se si è alla prima lettura o alla seconda lettura |

## 2.2 FSM




### 2.2.1 RST

E' lo stato di destinazione quando il segnale **i_rst** viene portato a 1. In questo stato iniziale vengono inizializzati tutti i segnali interni e si aspetta il segnale di **i_start** per iniziare l'elaborazione

### 2.2.2 REQ_LETT

Stato nel quale viene settato il segnale **o_address** all'indirizzo corretto della memoria RAM per poter leggere il dato

### 2.2.3 ELAB_LETT

E' uno stato intermedio in cui si attende la risposta dalla RAM alla richiesta di un dato

### 2.2.4 LETT

In questo stato i dati letti dalla RAM vengono memorizzati in **3 registri** a seconda del valore contenuto nel registro **selezione**. In particolare viene memorizzata la **dimensione delle righe**,la **dimensione delle colonne** e la **dimensione totale(righe * colonne).** Questo stato è attraversato ogni volta che si deve leggere dalla memoria e perciò viene memorizzato anche il valore del pixel corrente. Lo stato successivo dipende dal valore del segnale **modifica**, se quest'ultimo è =0 lo stato successivo è **CALCOLO** altrimenti è **CALCOLO_PIXEL**

### 2.2.5 CALCOLO

Lo scopo di questo stato è quello di trovare il minimo e il massimo dei dati letti dalla RAM. Per prima cosa si controlla che la **dim**(righe*colonne) sia >0 e che **count** sia < **dim** altrimenti si passa allo stato di fine elaborazione **DONE**. Se la dimensione è >0 e se **count** è minore di **dim** allora si confronta il dato corrente con il valore contenuto in **min** e **max**, si incrementa il contatore **count** e si ritorna allo stato **REQ_LETT** per leggere il prossimo pixel. Una volta letti tutti i valori in ingressi si porta a 1 il valore di **modifica**, si calcola lo **shift** attraverso dei controlli a soglia sul **delta_value** e si ritorna allo stato **REQ_LETT** per poter rileggere l'immagine dal primo pixel

### 2.2.6 CALCOLO_PIXEL

All'interno di questo stato per ogni pixel si calcola il valore del nuovo pixel dell'immagine equalizzata, successivamente si passa allo stato **REQ_SCR**. Se sono stati calcolati tutti i nuovi pixel dell'immagine equalizzata si passa allo stato **DONE**.

### 2.2.7 REQ_SCR

In questo stato prima di procedere con la scrittura si controlla se il nuovo valore del pixel è superiore o meno a "255" e in caso positivo lo si imposta a "255". Come da specifica si alza il segnale **o_we** a 1 per poter scrivere sulla RAM, si imposta l'indirizzo in cui scrivere e si passa allo stato **ELAB_SCR**

### 2.2.8 ELAB_SCR

Dopo aver scritto il nuovo pixel in memoria si deve tornare a calcolare il successivo pixel da scrivere. In questo stato, come da specifica, si abbassa il segnale **o_we** per poter cosi tornare a rileggere il successivo pixel e proseguire l'elaborazione

### 2.2.9 DONE

Questo è lo stato di fine elaborazione nel quale alziamo a 1 il segnale **o_done** per indicare la corretta fine dell'elaborazione

### 2.2.10 FINE

In questo stato si attende che **i_start** venga messo a 0 per poter tornare allo stato di RST e attendere un nuovo segnale di start.

# 3 Risultati

## 3.1 Sintesi

Il componente è correttamente sintetizzabile e implementabile.

| Name | Constraints | Status | Total Power | Failed Routes | LUT | FF |
|---|---|---|---|---|---|---|
| ✓ synth_1 | constrs_1 | synth_design Complete! | | | 221 | 150 |
| ✓ impl_1 | constrs_1 | route_design Complete! | 7.088 | 0 | 221 | 150 |

Usa 221 LUT e 150 FF.

| Name 1 | Slice LUTs (134600) | Slice Registers (269200) | Bonded IOB (285) | BUFGCTRL (32) |
|---|---|---|---|---|
| N project_reti_logiche | 221 | 150 | 38 | 1 |

Dopo aver effettuato la sintesi del componente ho effettuato un serie di test casuali e specifici per verificarne il comportamento anche i casi particolari.
Tutti i test sono stati eseguiti in Behavioral Simulation, Post-Synthesis Functional Simulation e Post-Implementation Functional Simulation.

## 3.2 Test specifici e casuali

L'obbiettivo dei test specifici è quello di individuare le criticità dell'algoritmo nei seguenti casi limiti:

- Dimensione immagine = 0 pixel
- Tutti pixel uguali dell'immagine
- Reset asincrono
- Elaborazioni di più immagini

### 3.2.1 Dimensioni immagine=0 pixel

In questo test il numero di colonne o il numero delle righe era posto a 0 e una volta calcolato che la **dim**=0 il componente doveva segnalare la fine dell'elaborazione senza nemmeno preocupparsi delle altre celle di memoria se fossere vuoto o meno.

### 3.2.2 Tutti pixel uguali dell'immagine

L'obbiettivo di questo test era quello di verificare se l'algoritmo si comporta o meno come da specifica. In ingresso si ha un'immagine di dimensione arbitraria(anche dimensione = 1) con tutti i valori dei pixel uguali(in particolare tutti uguali =0 e tutti =255). Da specifica il risultato atteso è un'immagine della stessa dimensione contenente tutti pixel =0.

### 3.2.3 Reset asincrono

Questo test verifica che la macchina quando il segnale **i_rst** viene alzato a 1 torni allo stato iniziale indipendentemente dagli altri ingressi, come da specifica.

### 3.2.4 Elaborazioni di più immagini

Per questo tipo di test si è usato un numero di immagini grande fino a 10k in modo da osservarne il comportamento del componente sotto stress.

### 3.2.5 Test casuali

Testing di un numero elevato di test generati casualmente per verificare il comportamento.

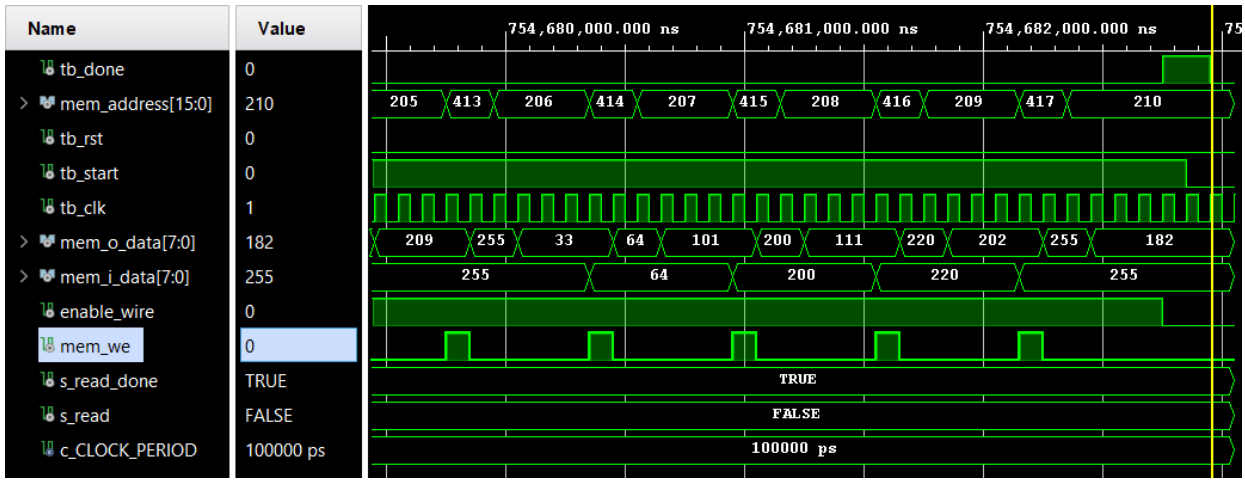



Figura 1: Simulazione test di 10k immagini

## 3.3 Conclusioni

I test specifici sono stati proggettati per portare la simulazione verso i corner case mentre i test casuali testavano il funzionamento in condizioni normali. Il componente ha risposto correttamente a tutti i test effettuati per cui ho ritenuto di aver coperto quasi la totalità dei possibili cammini che si può avere durante la computazione.
Il diagramma della FSM non è la soluzione minimale tuttavia è molto chiara e facilmente modificabile per eventuali ottimizzazioni future.