

# **Prova finale (Progetto di Reti Logiche)**

Prof. William Fornaciari anno 2020 – 2021

Enada Dervishi    10607448

Vladislav Burduja   10604480

# INDICE

1. Introduzione	
a. Analisi della specifica.....	3
b. Descrizione della memoria.....	4
2. Architettura	
a. Scelte progettuali.....	5
b. Architettura FSM.....	6
3. Sintesi	
a. Report di sintesi.....	8
b. Risultati dei test.....	9
4. Conclusioni .....	11

## 1.a – ANALISI DELLA SPECIFICA

La specifica del progetto richiede la progettazione in linguaggio VHDL di un componente in grado di leggere da memoria un'immagine, equalizzarne l'istogramma<sup>1</sup> ricalibrando così il contrasto ed infine scrivere in memoria l'immagine equalizzata.



Fig. 1: Esempi di immagini pre e post equalizzazione (sorgente Wikipedia)

Nella versione da sviluppare, l'algoritmo di equalizzazione sarà applicato solo ad immagini in scala di grigi a 256 livelli e deve trasformare ogni suo pixel nel modo seguente:

$$\Delta value = maxPixelValue - minPixelValue$$

$$shiftLevel = 8 - floor(log_2(\Delta value + 1))$$

$$tempPixel = (currPixelValue - minPixelValue) \ll shiftLevel$$

$$newPixelValue = min(255, tempPixel)$$

Dove *maxPixelValue* : massimo valore dei pixel dell'immagine

*minPixelValue* : minimo valore dei pixel dell'immagine

*floor()* : funzione che calcola l'intero inferiore

*currPixelValue* : valore del pixel da equalizzare

*newPixelValue* : valore del pixel equalizzato

<sup>1</sup>[https://it.wikipedia.org/wiki/Equalizzazione\\_dell'istogramma](https://it.wikipedia.org/wiki/Equalizzazione_dell'istogramma)

La specifica inoltre impone la seguente interfaccia del componente:

SEGNALE	I/O	DESCRIZIONE
i_clk	Input	Clock fornito dal TestBench (100ns)
i_rst	Input	Segnale di reset
i_start	Input	Segnale di inizio elaborazione
i_data	Input	Vettore di 8 bit in risposta a seguito di una richiesta di lettura
o_address	Output	Indirizzo di 16 bit dove voglio leggere / scrivere
o_done	Output	Segnale che indica la fine dell'elaborazione
o_en	Output	Segnale di enable, va tenuto alto per poter leggere / scrivere
o_we	Output	Segnale di write enable, va tenuto alto per poter scrivere
o_data	Output	Vettore di 8 bit da scrivere in RAM

## 1.b – DESCRIZIONE MEMORIA

La memoria è già istanziata all'interno del TestBench ed è progettata per rispondere ad una richiesta di lettura nel ciclo di clock **successivo** a quando viene fornito l'indirizzo.

### Dimensione dell'immagine

Il byte all'indirizzo **0** si riferisce alla dimensione di colonna, nell'esempio  $N-COL=2$ .

Il byte all'indirizzo **1** si riferisce alla dimensione di riga, nell'esempio  $N-RIG = 3$ .

Sia la dimensione colonna che la dimensione riga sono superiormente limitate a 128.

### Immagine originale

I pixel, ciascuno di 8 bit, sono memorizzati a partire dall'indirizzo **2**.

### Immagine equalizzata

I pixel, ciascuno di 8 bit, sono salvati in memoria a partire dall'indirizzo:

$$2 + (N-COL * N-RIG)$$

<b>data</b>	2	3	10	20	30	40	50	60	0	80	160	240	255	255	0	...	0
<b>address</b>	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	...	...

Fig. 2: esempio memoria con immagine 2 x 3 a intervalli regolari, post equalizzazione

## 2.a – SCELTE PROGETTUALI

Per implementare il componente richiesto, si è ritenuto opportuno descrivere quest'ultimo attraverso una **FSM a due processi**. Rispetto ad una FSM a singolo processo viene associato per ogni register un segnale `register next`.

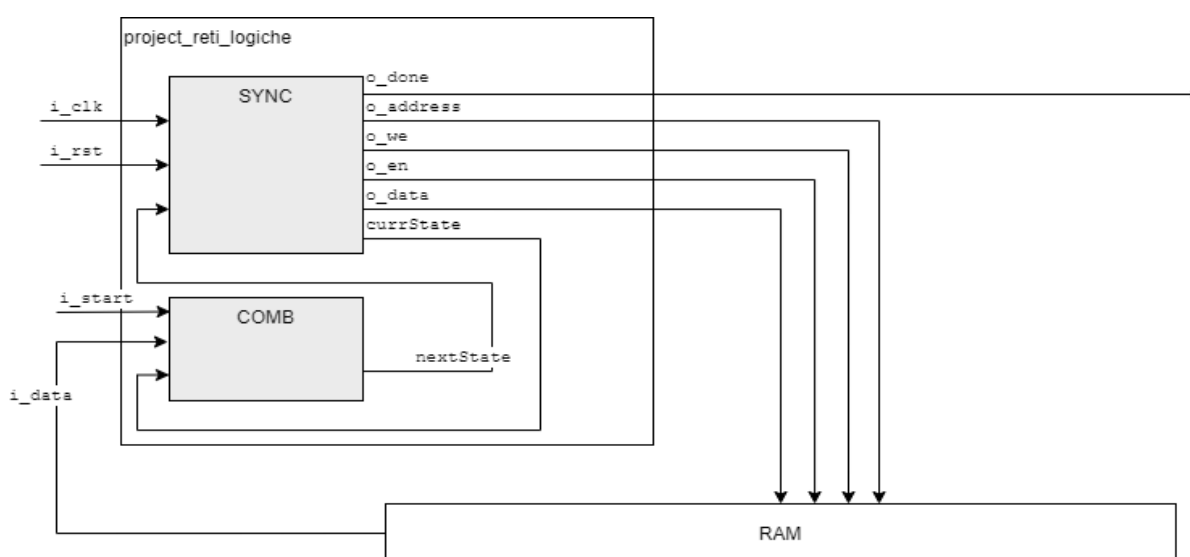
Il primo processo (**SYNC**), che è sincrono ed allineato sul fronte di **salita** del clock, carica i valori di `register_next` in `register` oppure riporta `register` ai valori di default in caso `i_rst` sia portato a **1**.

Il secondo invece (COMB) è combinatorio; esso infatti rappresenta la FSM che analizza i segnali in ingresso e lo stato corrente per determinare l'output ed il prossimo stato in cui evolverà il sistema.

L'algoritmo determina il risultato finale utilizzando come operazioni lo `shift_left`, il `cast to_integer` per eseguire lo shift, vari comparatori e il `cast` a `unsigned` per eseguire operazioni di somma e differenza.

Da una prima analisi dell'algoritmo si era pensato di memorizzare l'intera immagine così da evitare una seconda lettura; questo però sarebbe risultato dispendioso in termini di registri utilizzati considerate le potenziali dimensioni dell'immagine (128x128). Considerato ciò e il fatto che non ci sono restrizioni sul tempo di elaborazione ma solo sul periodo di clock, abbiamo deciso di far eseguire la seconda lettura.

Per quanto riguarda l'uso dell'operatore di moltiplicazione (\*) per il calcolo della dimensione (necessaria per determinare gli indirizzi di scrittura) abbiamo preferito utilizzare le somme. Data un'immagine **A** x **B**, la sua dimensione sarà calcolata in **min (A, B)** cicli di clock.



**Fig. 3: Rappresentazione ad alto livello del componente**

## 2.b – ARCHITETTURA FSM

Di seguito è riportato il cammino standard che percorre la macchina. Lo stato iniziale è quello di **IDLE** (fig. 4). Quando il segnale **i\_start** viene alzato, il componente inizia l'elaborazione passando al primo stato di computazione (**FETCH\_DIM**). A questo punto vengono caricate le dimensioni rispettivamente nei registri **N-COL** e **N-RIG**, controllata la loro validità e, se questa è valida, avviene la ricerca del massimo e minimo. Una volta trovati ha inizio il calcolo delle costanti, prima del **delta** e poi di **shiftLvl** attraverso controlli di soglia. Successivamente avviene la rilettura dei pixel originali e la loro equalizzazione uno ad uno. Dopo la scrittura in memoria dell'ultimo pixel equalizzato, viene alzato **o\_done** e il componente viene trasferito nello stato **DONE**. In questo stato i registri vengono portati ai loro valori di default in vista di una successiva elaborazione. Infine, una volta che **i\_start** viene abbassato, **o\_done** come da specifica viene portato a **0** e la macchina viene riportata al suo stato **IDLE**. Se a questo punto viene rialzato il segnale **i\_start**, il modulo ripartirà con la fase di codifica.

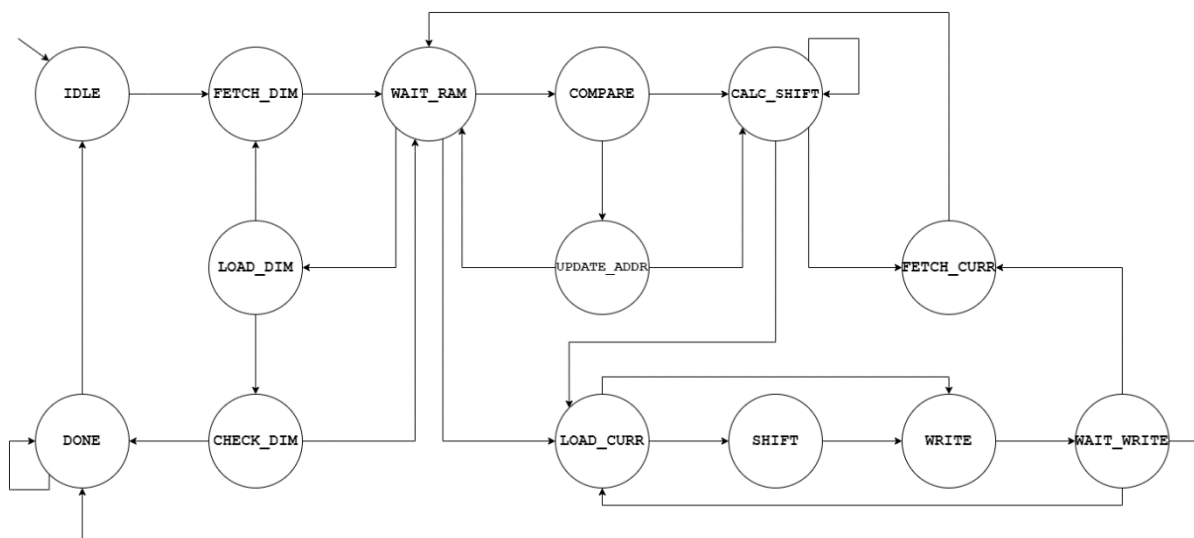


Fig. 4: Diagramma degli stati della FSM

Sono stati identificati due corner case in cui il componente si può imbattere durante la sua esecuzione.

Il primo, rappresentato dalla freccia che unisce **CALC\_SHIFT** a **LOAD\_CURR** (fig.4), si verifica quando si hanno tutti i pixel uguali: qui il **delta** risulta **0** e **shiftLvl** pari a **8**. Con i dovuti calcoli, è facile dimostrare che tutti i pixel equalizzati avranno valore pari a **0** indipendentemente dal valore del pixel originale, viene quindi saltata la fase di richiesta di quest'ultimo.

Il secondo caso invece è rappresentato in figura dalla freccia che collega lo stato **LOAD\_CURR** allo stato **WRITE**. Esso si verifica quando il **delta** risulta massimo (**255**) e quindi il valore di **shiftLvl** è pari a **0**; qui il pixel equalizzato rimane invariato rispetto a quello originale, perciò ha subito luogo la fase di scrittura senza passare per lo stato **SHIFT**.

La macchina a stati, il cui schema in termini di diagramma degli stati, ha 14 stati; qui di seguito una breve descrizione di ciascuno.

STATO	DESCRIZIONE
IDLE	Stato iniziale/di reset in cui si attende che il segnale <code>i_start</code> sia portato a <b>1</b>
FETCH_DIM	Vengono chiesti i valori di <code>N-COL</code> e <code>N-RIG</code> impostando <code>o_address</code>
WAIT_RAM	Stato di hold per permettere alla memoria di rispondere a seguito di una richiesta di lettura
LOAD_DIM	Vengono salvate le dimensioni ( <code>i_data</code> ) dell'immagine data in risposta dalla memoria
CHECK_DIM	Stato in cui viene controllata la validità delle dimensioni, se queste sono valide proseguo con la lettura
COMPARE	Qui avviene la ricerca di un massimo e un minimo attraverso lo scorrimento dei valori presenti
CALC_SHIFT	Stato in cui si ricava prima la costante <code>delta</code> e poi, attraverso i valori di soglia la costante <code>shiftLvl</code>
FETCH_CURR	Viene impostato l'indirizzo al pixel da equalizzare
LOAD_CURR	Viene salvato in un registro il pixel da equalizzare
UPDATE_ADRR	Stato in cui <code>o_address</code> viene incrementato di <b>1</b>
SHIFT	Qui si calcola il nuovo pixel con l'operatore <code>shift_left</code>
WRITE	Stato in cui si scrive in memoria impostando <code>o_address</code> e <code>o_data</code>
WAIT_WRITE	Stato di hold per permettere alla memoria di memorizzare il dato
DONE	Stato finale in cui si riportano ai valori di default i registri

## 3.a – REPORT DI SINTESI

Il componente sintetizzato supera correttamente tutti i test specificati nelle simulazioni: *Behavioral*, *Post-Synthesis Functional* e *Post-Synthesis Timing*.

Qui di seguito è possibile vedere un confronto tra i tempi di simulazione dei due corner case che portano la macchina verso la più breve e la più lunga esecuzione con un periodo di clock di 100ns:

- 2450 ns, *Post-Synthesis Functional* con dimensione immagine = 1 pixel
- 14747550 ns, *Post-Synthesis Functional* con dimensione = 16384 pixel

### Timing\_report

Il timing report mostra un WNS di circa 95 ns, ciò vuol dire che il componente impiega circa 5 ns per eseguire le singole operazioni. Ciò permetterebbe di diminuire moderatamente il suo periodo senza incorrere in errori.

#### Setup

Worst Negative Slack (WNS): 95,798 ns  
Total Negative Slack (TNS): 0,000 ns  
Number of Failing Endpoints: 0  
Total Number of Endpoints: 369

### Report\_utilization

Dai report di sintesi inoltre risulta un'occupazione di registri sotto l'1% (fig. 5) coerente con la scelta di non salvare nessuno dei pixel dell'immagine originale.

```
Detailed RTL Component Info :
+---Adders :
  4 Input    16 Bit    Adders := 1
  3 Input    16 Bit    Adders := 1
  2 Input    16 Bit    Adders := 3
  2 Input     8 Bit    Adders := 3
  3 Input     8 Bit    Adders := 2
+---Registers :
  16 Bit    Registers := 6
  8 Bit     Registers := 11
  1 Bit     Registers := 9
+---Muxes :
  14 Input   16 Bit    Muxes := 6
  14 Input   14 Bit    Muxes := 1
  2 Input    14 Bit    Muxes := 15
  2 Input     8 Bit    Muxes := 8
  14 Input   8 Bit     Muxes := 10
  9 Input    4 Bit     Muxes := 1
  2 Input    2 Bit     Muxes := 1
  2 Input    1 Bit     Muxes := 7
  3 Input    1 Bit     Muxes := 1
  14 Input   1 Bit     Muxes := 25
```

Site Type	Used	Fixed	Available	Util%
Slice LUTs*	312	0	134600	0.23
LUT as Logic	312	0	134600	0.23
LUT as Memory	0	0	46200	0.00
Slice Registers	195	0	269200	0.07
Register as Flip Flop	195	0	269200	0.07
Register as Latch	0	0	269200	0.00
F7 Muxes	0	0	67300	0.00
F8 Muxes	0	0	33650	0.00

Fig. 5: Utilizzo di LUT, registri e Mux

Fig. 6. Descrizione register transfer level del componente



### 3. b – RISULTATI DEI TEST

Per verificare il corretto funzionamento del nostro componente vengono usati TestBench. Alcuni di questi sono ufficiali, fornitici dai docenti, altri invece sono stati definiti da noi per massimizzare la copertura dei possibili cammini che la macchina può effettuare durante la computazione. I TestBench usati sono:

#### TestBench ufficiale

- Si tratta di test forniti dai docenti, semplici con immagini di piccole dimensioni

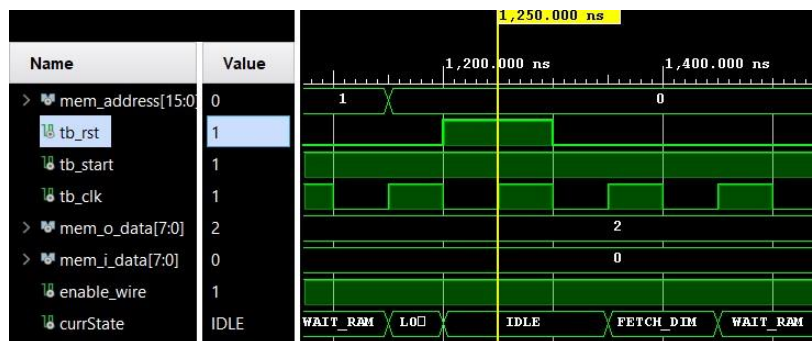
#### TestBench generator

- Generatore automatico di immagini. Permette di testare centinaia di immagini senza il bisogno di crearle singolarmente

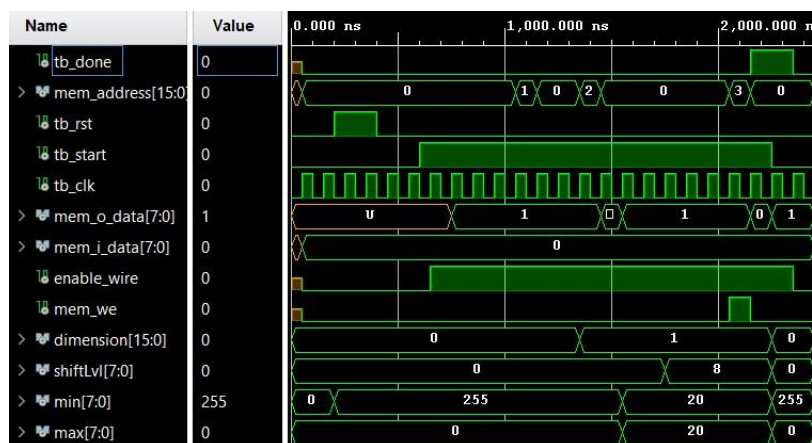
#### TestBench specifici

- Il loro fine è quello di testare cammini singolari. Di seguito i test con le rispettive *waveform* delle fasi critiche

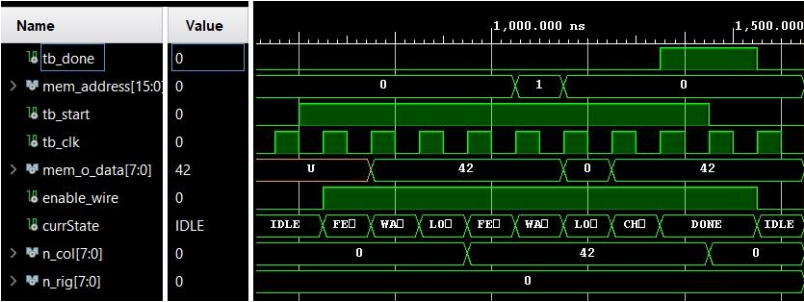
*Reset*: testiamo il comportamento del componente con un segnale di reset asincrono durante l'esecuzione che riporta la macchina allo stato `IDLE` e i registri ai valori di default



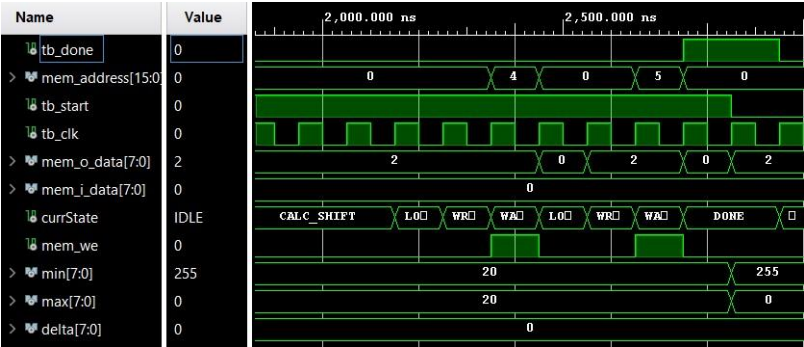
*Dimensione 1x1*: immagine da 1 pixel; rappresenta il cammino minimo



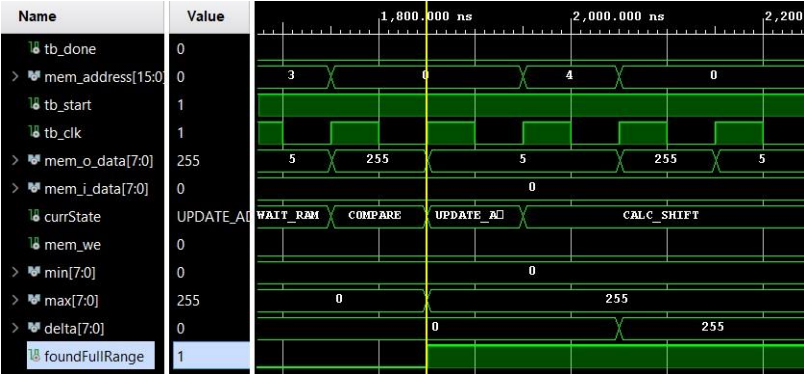
*Dimensione 0:* una o entrambe le dimensioni sono nulle; termina quindi senza una porta per poter comunicare l'errore



*Immagine mono valore:* l'immagine presenta gli stessi valori per tutti i pixel; viene saltata la parte di riletture dei pixel originali; o\_data è costante a 0



*FoundFullRange:* ottenimento di 0 e 255 prima della fine della lettura di tutti i pixel; il segnale foundFullRange passa a 1 e si procede al calcolo delle costanti



## 4 – CONCLUSIONE

### Ottimizzazioni

Abbiamo deciso di ottimizzare l'algoritmo di modo che la ricerca di massimo e minimo venga arrestata quando i valori **0** e **255** sono stati trovati, purché sia stata precedentemente trovata la dimensione. Così facendo non c'è bisogno di continuare la lettura di tutti i pixel e, si può procedere con l'equalizzazione vera e propria.

### Limiti del componente

Dato che la dimensione è calcolata attraverso somme ad ogni ciclo di clock, il componente ha bisogno di **min(N-COL, N-RIG)** cicli di clock per calcolarla. Nell'ipotetico caso di un'immagine **128x128**, la dimensione sarebbe quindi calcolata in **128** cicli di clock. Considerando che in questo numero di cicli di clock vengono analizzati circa **42** pixel, se tra questi ci fossero sia il valore **0** sia **255**, ci troveremmo in un caso di mancata ottimizzazione. Data la rara natura del caso e il problema trascurabile che provoca (perdita di circa 120 cicli di clock), si è deciso di accettarlo come un limite del componente.

### Altre considerazioni

Dalla struttura predefinita del TestBench (fig. 7) si denota che all'inizio della computazione gli indirizzi non coinvolti presentano valori pari a zero. Questa osservazione ci porta a pensare che, nella situazione descritta a pagina n.6, dove **delta** risulta **0** così come i valori dei pixel equalizzati, si potrebbe ottenere lo stesso identico risultato anche senza intervenire ulteriormente. Abbiamo comunque deciso di procedere alla riscrittura dei pixel poiché, nel caso in cui il TestBench fosse strutturato diversamente, il componente potrebbe non funzionare correttamente.

```
signal RAM: ram_type := (0 => std_logic_vector(to_unsigned( 2 , 8)),
1 => std_logic_vector(to_unsigned( 2 , 8)),
2 => std_logic_vector(to_unsigned( 46 , 8)),
3 => std_logic_vector(to_unsigned( 131 , 8)),
4 => std_logic_vector(to_unsigned( 62 , 8)),
5 => std_logic_vector(to_unsigned( 89 , 8)),
others => (others => '0'));
```

Fig. 7: Struttura TestBench