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[1] 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

[1] https://it.wikipedia.org/wiki/Equalizzazione_dell%27istogramma

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 venuto 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)`

data	2	3	10	20	30	40	50	60	0	80	160	240	255	255	0	…	0
address	**0**	1	2	3	4	5	6	7	8	9	10	11	12	13	14	…	…

Fig. 2: esempio memoria con immagine 2 x 3, 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 aggiunto un segnale `register_next` oltre al solito `register`.

Il primo processo (`SYNC`) è sincrono ed allineato sul fronte di **salita** del clock che 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, infatti esso 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 e il cast a `unsigned` per eseguire operazioni di somma e differenza.

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

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

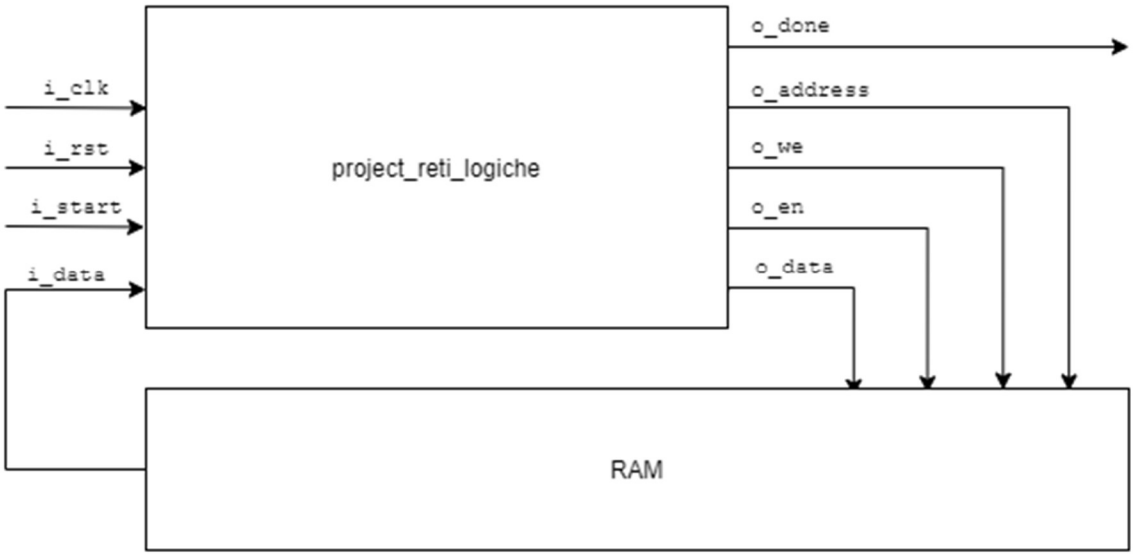


Fig. 3: Rappresentazione ad alto livello del componente

2.b – ARCHITETTURA FSM

Di seguito è riportato il cammino standard che percorre la macchina. Lo stato iniziale è quello di `IDLE` (vedi 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, controllata la loro validità e avviene la ricerca degli estremi. Nella fase seguente ha inizio il calcolo delle costanti, prima del `delta` e poi di `shiftLvl` attraverso dei controlli di soglia. Successivamente avviene il processo di equalizzazione dei pixel uno per uno. Dopo la scrittura in memoria dell'ultimo pixel equalizzato, viene alzato `o_done` e il componente viene trasferito nello stato `DONE` dove 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 di `i_start`, il modulo ripartirà con la fase di codifica.

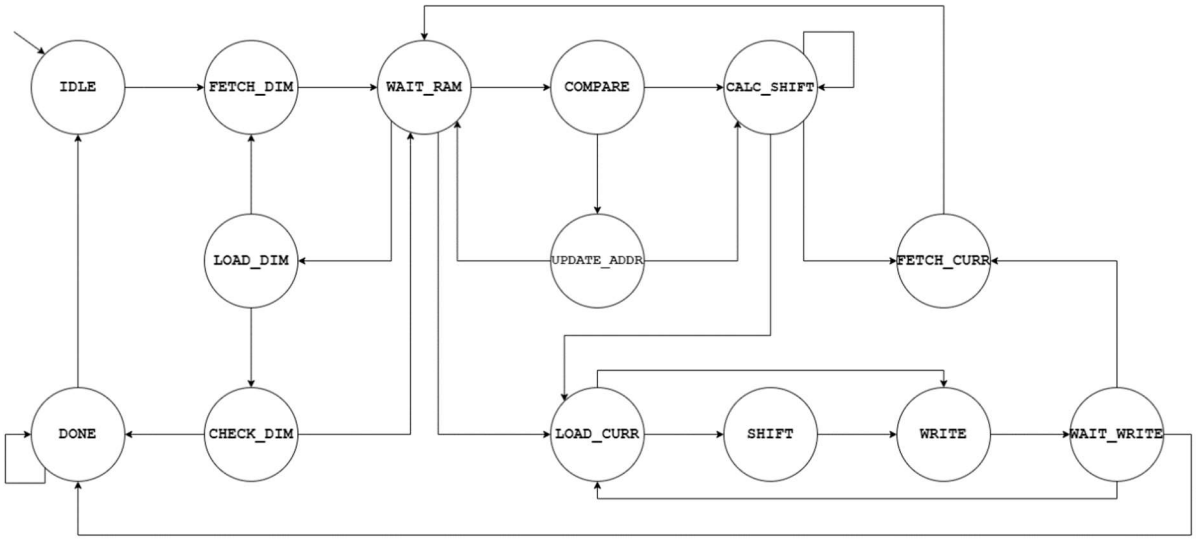


Fig. 4: Diagramma degli stati della FSM

Durante l'esecuzione del componente ci si può imbattere in due corner case. Nel primo caso, rappresentato in figura dalla freccia che collega lo stato `LOAD_CURR` allo stato `WRITE`, si verifica quando il `delta` risulta massimo (**255**) e quindi il valore di `shiftLvl` è pari a **0**, ha quindi subito luogo la fase di scrittura (siccome il pixel equalizzato rimane invariato rispetto a quello originale). Il secondo caso invece, rappresentato dalla freccia che unisce `CALC_SHIFT` a `LOAD_CURR`, 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 dell'ultimo (di esso?).

La macchina a stati il cui schema in termini di diagramma degli stati realizzata è composta da 14 stati, qui di seguito una breve descrizione di ciascuno.

STATO	DESCRIZIONE
IDLE	Stato iniziale/di reset in cui attendo che il segnale i_start sia portato a **1**
FETCH_DIM	Chiedo i valori di N-COL e N-RIG impostando o_address
WAIT_RAM	Stato di hold per permettere alla memoria di rispondere a seguito di una richiesta di lettura
LOAD_DIM	Salvo le dimensioni (i_data) dell'immagine date in risposta dalla memoria
CHECK_DIM	Stato in cui controllato la validità della dimensione, se questa è valida 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 delta e poi, attraverso i valori di soglia la costante shiftLvl
FETCH_CURR	Imposto l'indirizzo al pixel da equalizzare
LOAD_CURR	Salvo in un registro il pixel da equalizzare
UPDATE_ADRR	Stato in cui o_address viene incrementato di 1
SHIFT	Calcolo il nuovo pixel con l'operatore shift_left
WRITE	Stato in cui viene scrivo in memoria impostando o_address e o_data
WAIT_WRITE	Stato di hold per permettere alla memoria di memorizzare il dato
DONE	Stato finale in cui riporto ai valori di default i registri

3.a – REPORT DI SINTESI

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

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 100 ns:

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

- 14747550 ns, *Post-Synthesis Functional* con dimenzione = 16384 pixel

Timing_report

Il timing report mostra un WNS di 95ns, ciò vuol dire che il componente ci impiega circa 5 ns per eseguire le singole operazioni. Questo vuol dire che possiamo in teoria abbassare il periodo (in modo non drastico) 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% (vedi 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
```

Fig. 6. Descrizione register transfer level del componente

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

3. b – RISULTATI DEI TEST

Per verificare il corretto funzionamento del nostro componente vengono usati TestBench. Alcuni di questi sono ufficiali, forniteci dai docenti, altre invece sono state definite 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

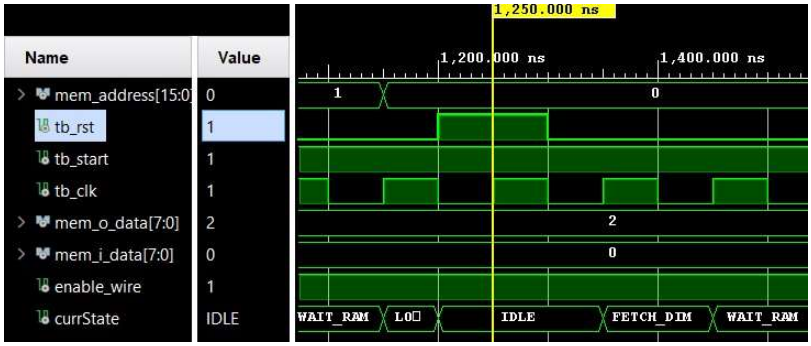
TestBench generator

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

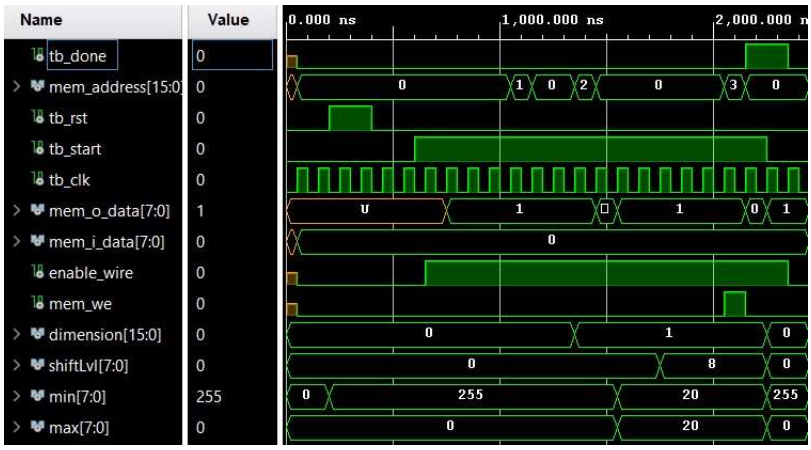
TestBench specifici

- Il loro fine è quello di testare cammini singolari. Di seguito i test con le rispettive *waveform* delle parti salienti/interessanti

Reset: testiamo il comportamento del componente con un segnale di reset asincrono durante l'esecuzione



Dimensione 1x1: immagine da 1 pixel, cammino minimo



Dimensione 0: una o entrambe le dimensioni sono nulle, il componente si arresta però non ha porte per comunicare l'errore

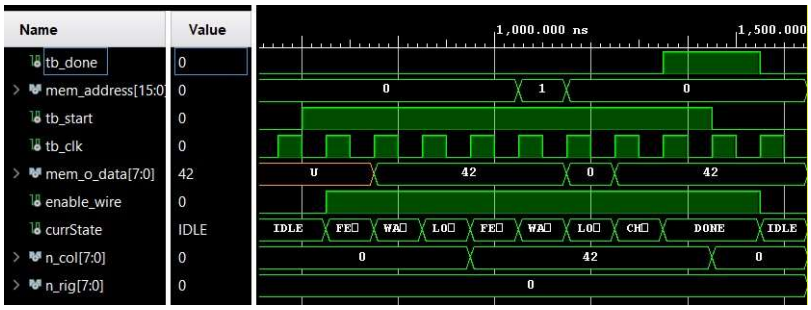
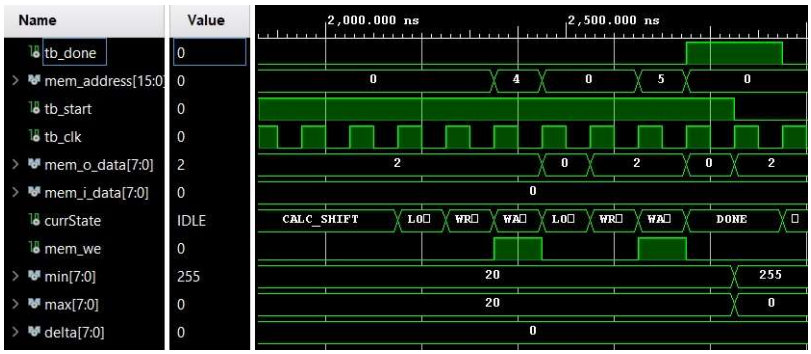
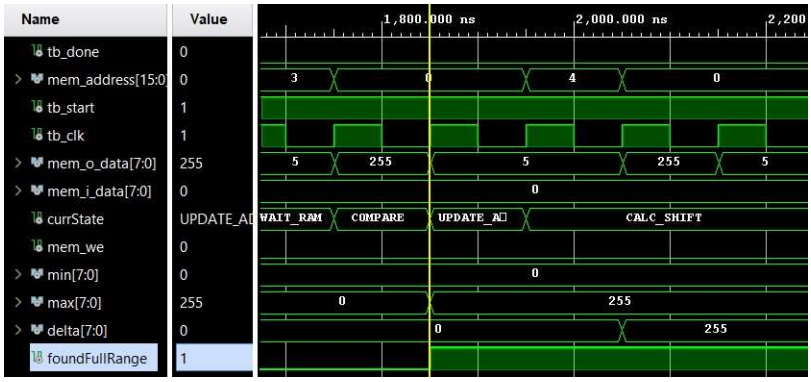


Immagine mono valore: viene saltata la parte di rilettura dei pixel originali



FoundFullRange: ottenimento di 0 e 255 prima della fine della lettura di tutti i pixel



4 – CONCLUSIONE

Ottimizzazioni

Abbiamo deciso inoltre di ottimizzare l'algoritmo di modo che la ricerca di massimo e minimo venga arrestata, nel momento in cui 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

Siccome la dimensione è calcolata attraverso somme ad ogni ciclo di clock, il componente ha bisogno di min(`N-COL`, `N-RIG`) ciclo di clock per calcolare la dimensione. 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 fosse sia il valore **0** che **255**, ci troveremmo in un caso di mancata ottimizzazione. Data la rara natura del caso e il piccolo problema 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 (vedi 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, in una situazione descritta meglio a pagina n.6 dove `delta` risulta **0** cosi come i valori dei pixel equalizzati. si potrebbe ottenere lo stesso identico risultato anche senza intervenire ulteriormente. Abbiamo comunque deciso di procedere alla ri-scrittura dei pixel poiché, nel caso 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