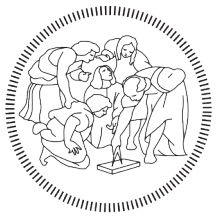
Prova Finale (Reti Logiche)

Prof. William Fornaciari - A.A. 2020/2021

Gabriele D'Angeli (Codice persona: 10631708 - Matricola: 912225)
Francesco Di Stefano (Codice persona: 10629747 - Matricola: 907024)



POLITECNICO
MILANO 1863

Indice

1 Introduzione

1.1 Scopo del progetto

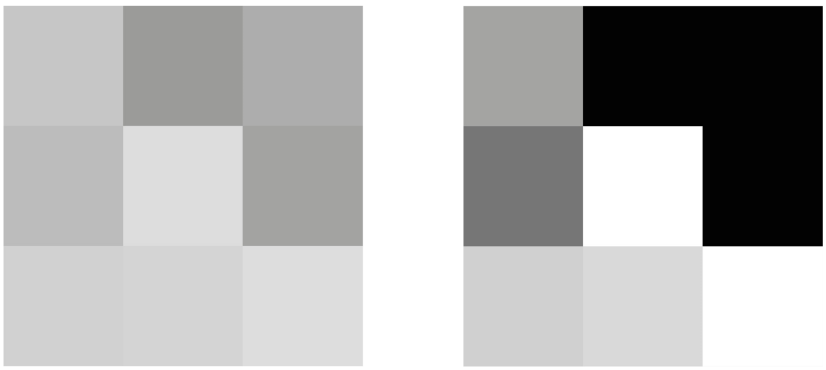
Obiettivo della prova finale è quello di implementare, utilizzando il linguaggio di descrizione hardware VHDL, una versione semplificata di un algoritmo standard di equalizzazione dell'istogramma di un'immagine in scala di grigi. Il metodo di equalizzazione è pensato per ricalibrare il contrasto di un' immagine quando l'intervallo dei valori di intensità sono molto vicini tra loro, effettuandone una ridistribuzione su tutto l'intervallo.

1.2 Specifiche generali

L'algoritmo di equalizzazione implementato può essere applicato solamente ad immagini in scala di grigi a 256 livelli e la cui dimensione massima sia 128x128 pixel. La trasformazione di ogni singolo pixel avviene 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, rispettivamente, il maggior e il minor valore dei pixel dell'immagine considerata, `CURRENT_PIXEL_VALUE` è il valore del pixel da trasformare e `NEW_PIXEL_VALUE` è il valore del nuovo pixel da scrivere in memoria.



Esempio: a sinistra l'immagine da equalizzare, a destra l'immagine equalizzata

1.3 Interfaccia del componente

Il componente da descrivere deve avere la seguente interfaccia:

```
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;
```

Dove:

- `i_clock` 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 per poter scriverci. Per leggere da memoria esso deve essere 0;
- `o_data` è il segnale (vettore) di uscita dal componente verso la memoria.

1.4 Dati e descrizione memoria

L'immagine da equalizzare viene letta sequenzialmente dalla memoria (con indirizzamento al byte), riga per riga. La dimensione dell'immagine è definita da due byte: il dato contenuto nell'indirizzo 0 della memoria si riferisce alla dimensione di colonna dell'immagine, quello contenuto nell'indirizzo 1 si riferisce alla dimensione di riga. A partire dall'indirizzo 2 è memorizzata l'immagine da equalizzare. Ogni byte corrisponde ad un pixel dell'immagine. Come già detto, la dimensione massima dell'immaginè è 128x128 pixel.
L'immagine equalizzata deve essere scritta in memoria immediatamente dopo l'immagine originale: il primo pixel dell'immagine equalizzata si troverà nella posizione `2 + (N_COL * N_RIG)` della memoria.

2 Architettura

2.1 Scelte progettuali

Per realizzare il componente richiesto abbiamo deciso di utilizzare un singolo processo che include tutto il funzionamento della macchina a stati finiti progettata. Il numero degli stati può essere senza dubbio ridotto ma abbiamo deciso di mantenere tutti gli stati illustrati per fare in modo che ogni stato avesse un singolo compito, così da semplificare eventuali modifiche al codice.

2.2 Stati della macchina

La macchina progettata è composta da nove stati, di seguito vengono illustrate le funzioni di ogni singolo stato:

2.2.1 `INIT` state

Questo è lo stato iniziale del componente, cioè quello in cui siamo quando non è stata elaborata ancora nessuna immagine. In questo stato attendiamo che il segnale *i_rst* venga alzato per la prima volta come da specifica.

2.2.2 `RST` State

In questo stato attendiamo che il segnale *i_start* venga alzato dando così il via all'elaborazione di un'immagine. Questo è lo stato in cui si torna alla fine dell'elaborazione di un'immagine o quando durante quest'ultima viene alzato il segnale *i_rst*.

2.2.3 `DIMENSION` State

Questa stato è introdotto per evitare l'utilizzo dell'operatore di moltiplicazione * per il calcolo del numero di pixel dell'immagine, tutto ciò che facciamo qui è quindi calcolare la dimensione dell'immagine utilizzando un segnale interno *img_dimension* come accumulatore delle somme ripetute e il segnale interno *img_height* come contatore.

2.2.4 `DELTA` State

Qui avviene il calcolo dei valori *MIN_PIXEL_VALUE* e *MAX_PIXEL_VALUE* ed in seguito quest'ultimi vengono utilizzati per il calcolo del *DELTA_VALUE*. La macchina rimane in questo stato finché il segnale interno *mem_read_position* è minore del segnale interno *img_dimension* sommato ad 1.

2.2.5 `SHIFT` State

Qui avviene il calcolo dello *SHIFT_LEVEL* attraverso un controllo a soglia, evitando così l'utilizzo di funzioni non sintetizzabili.

2.2.6 `TMP_PIXEL` State

Qui viene letto un pixel per volta dalla memoria e, utilizzando lo *SHIFT_LEVEL* calcolato precedentemente, viene calcolato anche il relativo *TMP_PIXEL*. Se il segnale interno *mem_write_position* è maggiore del segnale interno *img_dimension* moltiplicato per 2 e sommato ad 1, allora la macchina passa direttamente all'*END* State.

2.2.7 `WRITE` State

Il valore *TMP_PIXEL* precedentemente calcolato viene qui confrontato con il valore intero 255 ed il valore minore tra i due viene scritto in memoria.

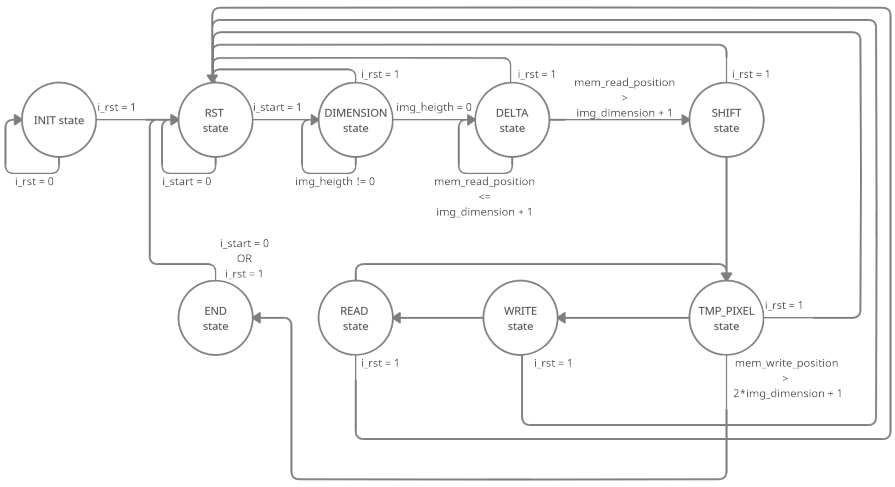
2.2.8 `READ` State

Questo stato è utilizzato per leggere il prossimo pixel dalla memoria da utilizzare poi nel calcolo del *TMP_PIXEL*, infatti se ci sono ancora pixel da leggere si torna nel *TMP_PIXEL* state e si iterano le operazioni eseguite.

2.2.9 `END` State

Questo stato rappresenta la fine dell'elaborazione di un'immagine. Tutto ciò che viene fatto qui è aspettare che il segnale *i_start* venga abbassato in modo da poter tornare nel *RST* State e fare in modo che il componente sia pronto per elaborare un'altra immagine.

2.3 Diagramma degli stati

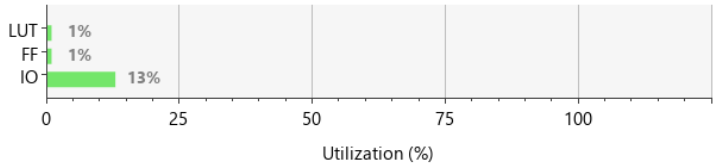


3 Sintesi

3.1 Utilization Report

Dall'utilization report generato da Vivado possiamo osservare che sono stati utilizzati 143 FF (Flip Flop) e 182 LUT (Look Up Table), mentre non sono stati utilizzati latch. Anche se questi numeri possono sembrare elevati, essi rappresentano, rispettivamente, lo 0.14% delle LUT e lo 0.05% dei FF totali disponibili nel FPGA, ciò è dovuto alla relativa semplicità del componente sintetizzato.

Resource	Utilization	Available	Utilization %
LUT	182	134600	0.14
FF	143	269200	0.05
IO	38	285	13.33



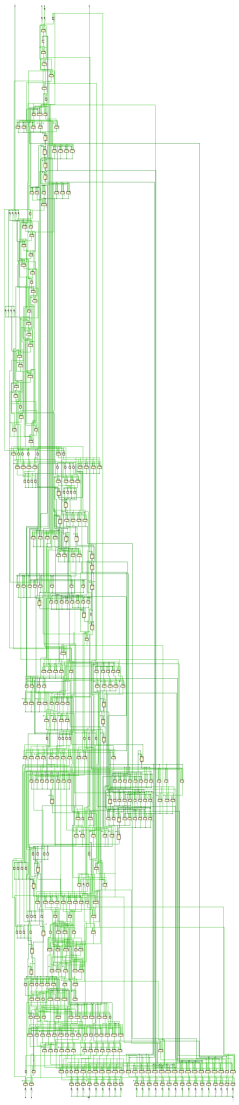
3.2 Timing Report

Dal timing report invece possiamo osservare che, ponendo un clock pari a 100 ns, si ottiene un Worst Negative Slack pari a 92.511 ns. Per osservare invece quanto periodo del clock sia effettivamente utilizzato dal componente per la computazione, possiamo avvalerci della Tcl Console fornita da Vivado ed utilizzare il comando 'report_timing' ottenendo così anche il Data Path Delay.

Setup		Hold		Pulse Width	
Worst Negative Slack (WNS):	92,511 ns	Worst Hold Slack (WHS):	0,080 ns	Worst Pulse Width Slack (WPWS):	49,500 ns
Total Negative Slack (TNS):	0,000 ns	Total Hold Slack (THS):	0,000 ns	Total Pulse Width Negative Slack (TPWS):	0,000 ns
Number of Failing Endpoints:	0	Number of Failing Endpoints:	0	Number of Failing Endpoints:	0
Total Number of Endpoints:	338	Total Number of Endpoints:	338	Total Number of Endpoints:	144

```
Slack (MET) :             92.511ns  (required time - arrival time)
  Source:                 img_dimension_reg[4]/C
                            (falling edge-triggered cell FDRE clocked by clock  {rise@0.000ns fall@50.000ns period=100.000ns})
  Destination:            mem_read_position_reg[10]/R
                            (falling edge-triggered cell FDRE 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:        6.879ns  (logic 3.295ns (47.899%)  route 3.584ns (52.101%))
```

3.3 Schema di sintesi



4 Risultati dei test

4.1 Informazioni generali

Il componente supera tutti i test a cui è stato sottoposto sia in Behaviorial Simulation che in Post-Synthesis Functional Simulation. In particolare, abbiamo suddiviso i test in due categorie: i test di casi limite ed i test di funzionamento. Per realizzare la prima categoria di test, abbiamo scritto un semplice generatore di test di singole immagini in Python utilizzabile attraverso un'interfaccia a riga di comando. La seconda categoria di test è stata invece realizzata manualmente. Infine, grazie ad un ulteriore generatore di test scritto da alcuni nostri colleghi, siamo stati in grado di testare il componente su un numero enorme di immagini casuali consecutive. I test utilizzati in entrambe le categorie ed il generatore da noi realizzato possono essere scaricati al seguente indirizzo: https://github.com/fdistefano99/Digital-Logic-Design-Project-2021. Qui sotto è riportato un esempio di funzionamento del nostro generatore.

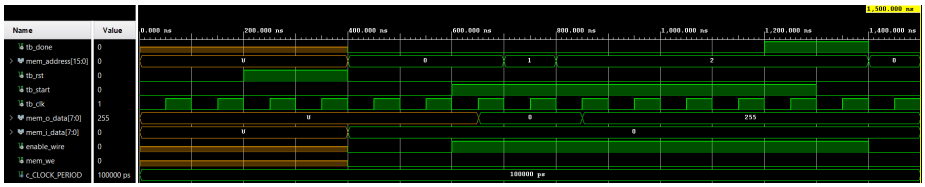
```
Inserire il numero di colonne(compreso tra 1 e 128) dell'immagine: 128
Inserire il numero di righe(compreso tra 1 e 128) dell'immagine: 128
Inserire la tipologia di test('RANDOM', 'ALL0', 'ALL255' o 'MANUAL') da generare: RANDOM
Testbench generato.
```

4.2 Test dei casi limite

Abbiamo individuato 9 casi limite per il componente, di seguito sono riportati gli andamenti dei segnali durante la simulazione dei 3 test di maggiore interesse.

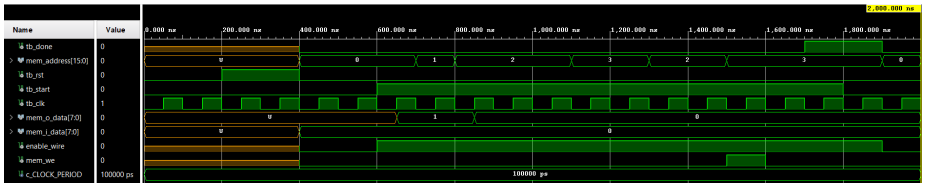
4.2.1 Immagine nulla

L'immagine in ingresso ha dimensione 0x0 e l'obiettivo del test è quello di verificare che il componente non scriva nulla in memoria.



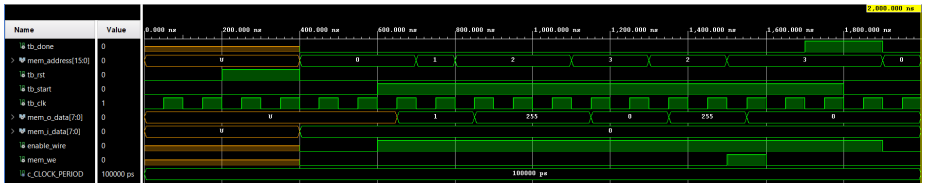
4.2.2 Immagine con un solo pixel a 0

L'immagine in ingresso ha dimensione 1x1 e l'unico pixel presente ha valore 0, l'obiettivo del test è quello di verificare che il componente mantenga a 0 il valore del pixel nell'immagine equalizzata.



4.2.3 Immagine con un solo pixel a 255

L'immagine in ingresso ha dimensione 1x1 e l'unico pixel presente ha valore 255, l'obiettivo del test è quello di verificare che il componente ponga a 0 il valore del pixel nell'immagine equalizzata.



4.2.4 Altri test di casi limite

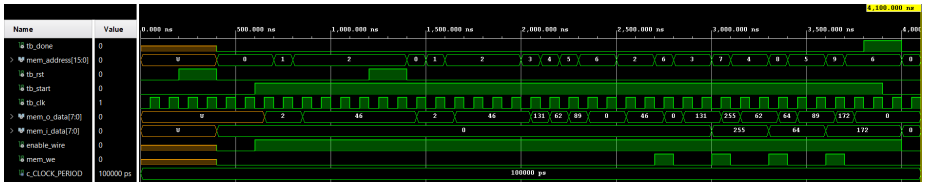
Oltre a quelli sopra riportati, sono stati generati anche test con immagini di dimensioni 128x1, 1x128 e 128x128 con valori tutti a 0 o tutti a 255. Abbiamo deciso di non riportare gli andamenti dei segnali relativi a quest'ultimi test poiché, avendo un numero elevato di pixel da elaborare, il numero di segnali sullo schermo è tale da non far risultare comprensibile l'andamento di quest'ultimi. Questi test possono essere comunque scaricati dal repository GitHub inserito poco sopra.

4.3 Test di funzionamento

Per studiare il comportamento del componente al di fuori dei casi limite abbiamo individuato 2 casi di particolare interesse, di seguito sono riportati gli andamenti dei segnali delle relative simulazioni.

4.3.1 Reset asincrono

In questo test viene alzato il segnale *i_rst* durante l'elaborazione di un'immagine ed in seguito si controllo che il componente riinizi la computazione dell'immagine e la porti a termine correttamente.



4.3.2 Elaborazioni multiple

In questo test vengono elaborate 3 immagini in successione, l'obiettivo è quindi quello di capire se il componente sia in grado di elaborare un'immagine subito dopo averne elaborata un'altra.