Relazione su Progetto Finale di Reti Logiche

Mattia Sironi (10614502), *Lea Zancani* (10608972)

A.A. 2020-2021

1 Introduzione

Il progetto di Reti Logiche 2020-2021 ha come obiettivo quello di realizzare, attraverso il linguaggio VHDL, un sistema di elaborazione per immagini digitali, che applicando un algoritmo standard ad ogni pixel ne ricalcoli un valore appartenente ad una scala più ampia. In questo modo alla fine del processo l'immagine complessiva risulta ricalibrata nel contrasto.

Esistono diversi metodi di equalizzazione per raggiungere il fine, quello da noi trattato è chiamato ***equalizzazione a istogramma dell'immagine***.

Un esempio è riportato di seguito.

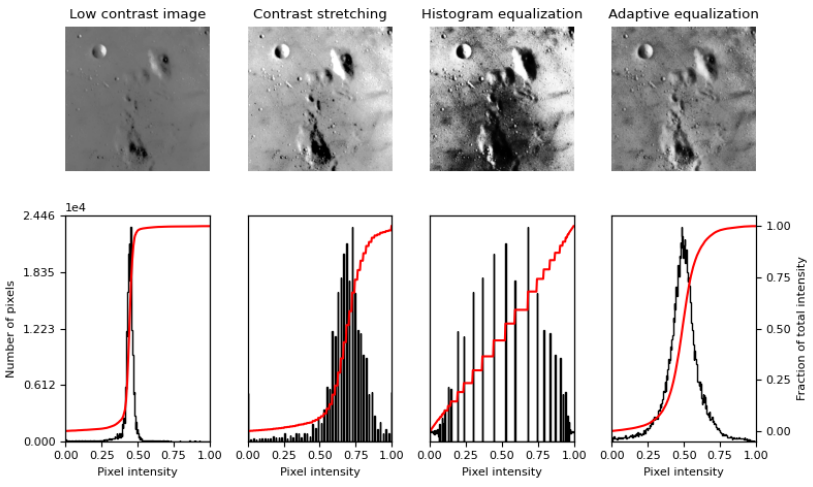

Figure 1: Diversi tipi di equalizzazione

2 Architettura

2.1 Specifica

L'implementazione si è basata sulla *entity*, sulla sua *behavioral architecture* e sul funzionamento del *testbench* ricevuti.

```
entity project_reti_logiche is
port (
      i_clk         : in  std_logic;
      i_start       : in  std_logic;
      i_rst         : 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 entity project_reti_logiche;
```

Ci è stata inoltre fornita una memoria con indirizzamento al byte.
I byte 0000_h e 0001_h erano destinati alle dimensioni dell'immagine. La dimensione massima prevista era di 128x128 pixel, su una scala di grigi a 256 livelli.
La specifica prevedeva la scrittura in memoria dell'immagine elaborata immediatamente dopo l'immagine ricevuta.

L'algoritmo per l'eleborazione del singolo pixel era fornito all'interno della specifica e si avvaleva di una serie di calcoli per valori intermedi.

DELTA_VALUE = MAX_PIXEL_VALUE – MIN_PIXEL_VALUE
SHIFT_LEVEL = (8 – FLOOR(LOG2(DELTA_VALUE +1)))
TEMP_PIXEL = (CURRENT_PIXEL_VALUE - MIN_PIXEL_VALUE) $\ll SHIFT\_LEVEL$
$NEW\_PIXEL\_VALUE = MIN(255, TEMP\_PIXEL)$

Al termine della scrittura abbiamo gestito i segnali *o_done, i_start e i_reset* secondo le indicazioni ricevute sul comportamento del *testbench*.

Figure 2: esempio di commutazione dei segnali *i_done* e *i_reset*, dalla presentazione del Prof. Palermo.

2.2 Scelta implementativa

Dopo aver analizzato la specifica abbiamo optato per la realizzazione di una macchina di Moore.

Abbiamo utilizzzato due processi :

1. Il processo che gestiva i segnali di controllo con lista di sensibilità:

   `{i_clk , i_rst , i_start}`

2. Il processo che realizzava la FSM con tutti i suoi stati, "risvegliato" dalle modifiche dei segnali effettuate dal processo precedente

2.2.1 Funzionamento della FSM

La FSM realizzata si compone di 6 stati :

- **STATO INIZIALE**: indica 0000_h come indirizzo di memoria da cui iniziare a leggere e inizializza i segnali "accessori" da noi realizzati.
- **BEFORE READ**: mette a '1' il segnale che abilita la comunicazione con la memoria RAM e indica l'indirizzo a cui leggere.
- **READ**: legge la memoria ed effettua operazioni differenti a seconda dell'indirizzo corrente:
  1. dopo aver ricevuto la dimensione verticale e orizzontale dell'immagine (indirizzo 0000_h e 0001_h) calcola la dimensione totale. Se almeno una delle due dimensioni è nulla, la FSM termina l'elaborazione andando direttamente allo stato DONE.
  2. legge la memoria una prima volta per cercare il massimo e il minimo. Una volta identificati calcola il Delta.
  3. rilegge la memoria ed elabora ogni elemento calcolando il nuovo valore di ogni pixel dopo lo shift.
- **SHIFT**: calcola il valore dello shift.
- **WRITE**: scrive in memoria il pixel elaborato.
- **DONE**: mette a '1' il segnale *o_done* per notificare la fine dell'elaborazione.

2.2.2 Considerazioni

La FSM così implementata resetta i segnali solo se riceve una seconda immagine, prima di iniziare a leggerla ed elaborarla.

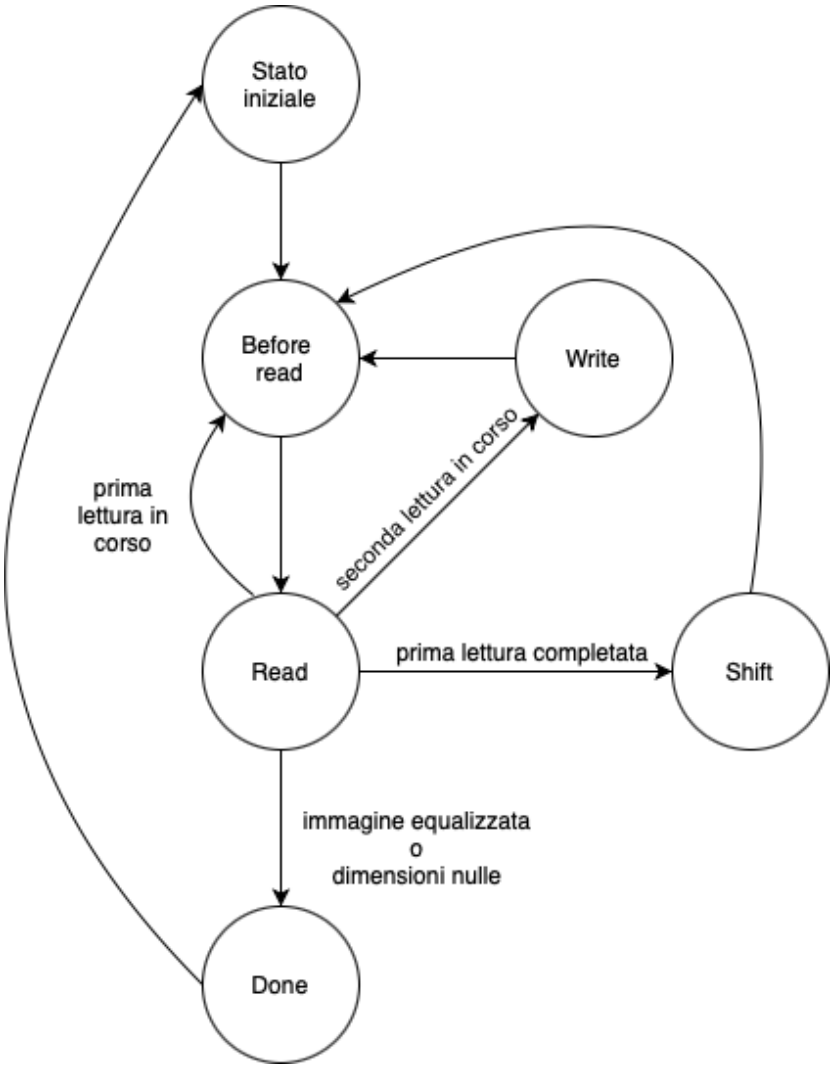

Figure 3: Grafico comportamento FSM

3 Risultati sperimentali

Abbiamo effettuato numerosi test, alcuni generati casualmente con l'utilizzo del generatore fornito dal prof. Salice, altri realizzati ad hoc per testare condizioni limite. In ogni caso il comportamento pre e post sintesi è sempre risultato coerente.

3.1 Test critici

- **Dimensione nulla** : Fornendo un'immagine con dimensione nulla abbiamo riscontrato che il nostro modulo funzionava correttamente, ma leggeva un byte non necessario e non appartenente all'immagine (Byte 0002_h) . Abbiamo quindi rivisto il codice per ottimizzare questo caso.

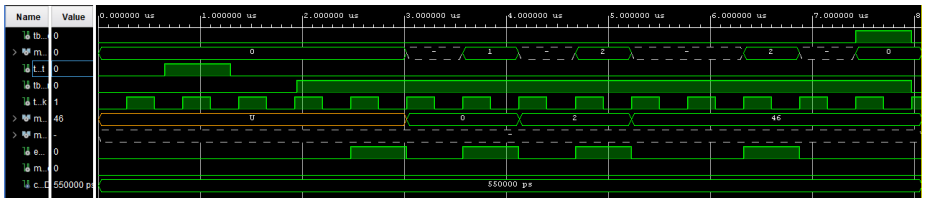

Figure 4: Prima dell'ottimizzazione

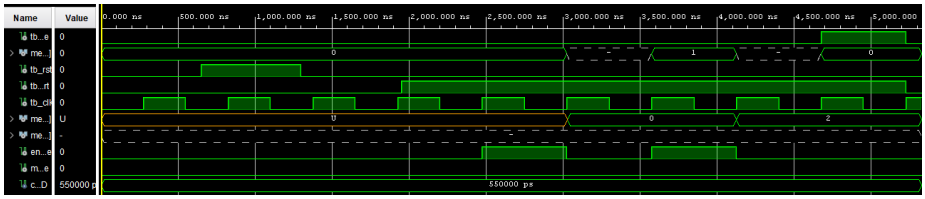

Figure 5: Dopo l'ottimizzazione

- **Casi limite controlli a soglia per il delta** : Abbiamo realizzato ed eseguito diversi test per verificare il corretto funzionamento dei controlli a soglia per il calcolo del delta, riscontrando qualche errore causato da un'operazione intermedia nel calcolo scritta in modo scorretto, che abbiamo poi corretto.

3.2 Caratteristiche della sintesi

In fase di sintesi ci siamo accorti di aver non volutamente realizzato qualche latch.

Con un'analisi del codice siamo riusciti a capire quali segnali fossero responsabili e abbiamo fatto in modo di risolvere il problema.

A questo proposito abbiamo inserito all'inizio del processso della macchina a stati un blocco di codice che si occupa di assegnare ad ogni segnale un valore almeno una volta. Questo non compromette il funzionamento della FSM, infatti se un certo segnale viene assegnato più volte all'interno della stessa esecuzione del processo, esso assume l'ultimo valore assegnatogli.

1. Slice Logic

Site Type	Used	Fixed	Available	Util%
Slice LUTs*	208	0	134600	0.15
LUT as Logic	208	0	134600	0.15
LUT as Memory	0	0	46200	0.00
Slice Registers	96	0	269200	0.04
Register as Flip Flop	96	0	269200	0.04
Register as Latch	0	0	269200	0.00
F7 Muxes	0	0	67300	0.00
F8 Muxes	0	0	33650	0.00

Figure 6: Slice Logic dopo l'eliminazione dei latch

4 Conclusioni

La principale difficoltà riscontrata nello svolgimento del progetto è stata quella di entrare nella logica di programmazione a livello più basso di quello a cui siamo abituati. In particolar modo, all'inizio abbiamo cercato di affrontare il problema con una programmazione sequenziale incompatibile con la gestione dei segnali di *clock* e il funzionamento sincrono del modulo.

Inoltre all'inizio la gestione dei segnali, che non si comportano come variabili, non è stata per niente intuitiva, ma una volta compreso il funzionamento di VHDL la realizzazione dell'algoritmo è stata relativamente semplice.