Prova Finale di Reti Logiche

Lucas José Manini

March 22, 2021

Matricola:	906915
Codice persona:	10625965
Docente:	Fabio Salice

1 Requisiti del progetto

Il progetto proposto consiste nell'implementazione in VHDL di un algoritmo ispirato al metodo di equalizzazione dell'istogramma di un'immagine, in modo da aumentarne il contrasto globale. In particolare, ne è stata presentata una sua versione semplificata, applicato solo ad immagini in scala di grigio a 256 livelli, di dimensione massima 128 x 128 pixel. Inoltre, il componente progettato deve essere in grado di equalizzare molteplici immagini consecutivamente, senza dover ricevere un segnale di reset.
Viene richiesto al componente di:

- Accedere ad una memoria RAM per recuperare i dati dell'immagine da equalizzare.
- Trasformare ogni pixel dell'immagine secondo la legge :

$$DELTA\_VALUE = MAX\_PIXEL\_VALUE - MIN\_PIXEL\_VALUE$$

$$SHIFT\_VALUE = 8 - \lfloor LOG_2(DELTA\_VALUE + 1) \rfloor$$

$$TEMP = (CURRENT\_PIXEL\_VALUE - MIN\_PIXEL\_VALUE) << SHIFT\_LEVEL$$

$$NEW\_PIXEL\_VALUE = MIN(TEMP, 255)$$

- Scrivere l'immagine equalizzata nella memoria RAM immediatamente dopo l'immagine originale.

2 Architettura

L'architettura del componente è stata progettata seguendo un approccio top-down, in modo da agevolare e semplificare il più possibile le prime fasi di progettazione, entrando nei dettagli implementativi solo quando effettivamente necessario. Il componente è descritto attraverso una **macchina a stati finiti**

(FSM), che detta lo stato della computazione e si interfaccia con la RAM per il trasferimento dei dati.

2.1 Descrizione ad alto livello

Da un'ottica di alto livello, l'implementazione esegue i seguenti passi:

1. Inizializza i segnali interni ai valori iniziali.
2. Legge la RAM, salvando il numero di colonne e righe che compongono l'immagine. Durante la lettura dei byte che compongono l'immagine, salva il valore minimo e massimo osservato fino a quel momento.
3. Calcola DELTA_VALUE e SHIFT_VALUE
4. Per ogni byte dell'immagine originale:
   (a) Ne legge il valore di grigio.
   (b) Calcola NEW_PIXEL_VALUE.
   (c) Scrive in memoria NEW_PIXEL_VALUE alla posizione corrispondente.
5. Se è richiesto un reset, torna a 1, altrimenti rimane in attesa del segnale di start, nel qual caso torna a 2.

Il processo è descritto dalla seguente FSM:

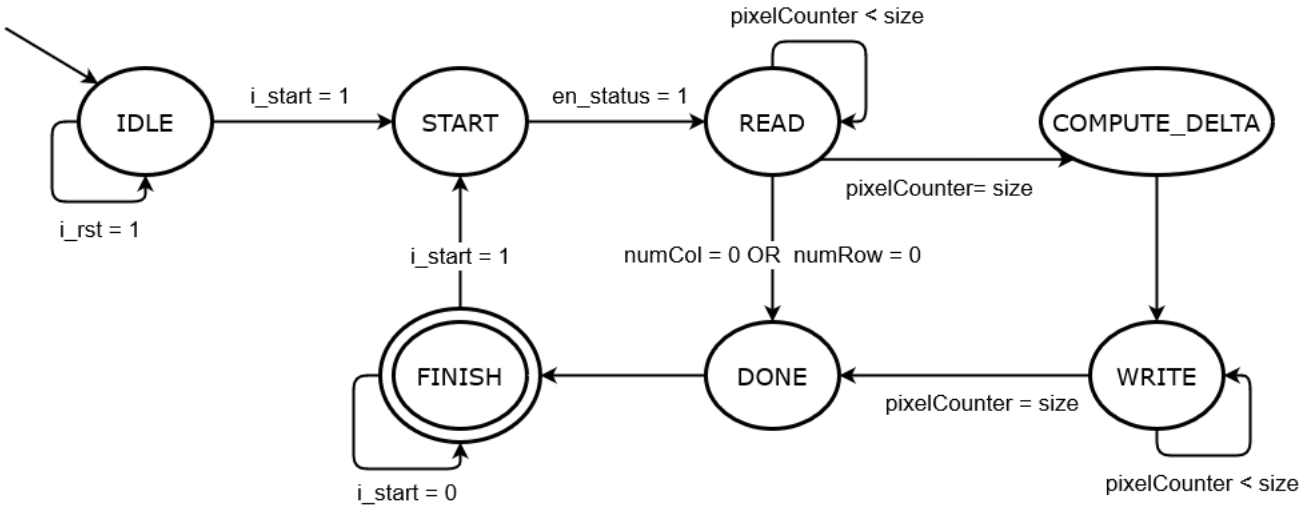

Figure 1: FSM semplificata

2.2 Architettura implementata

Entrando in maggior dettaglio, durante la fase di implementazione del codice VHDL, si è trovata la necessità di aumentare il numero di stati, per due motivi principali:

- In modo da rendere piú semplice la gestione dei segnali e delle operazioni.
- Si è manifestata la necessità di introdurre stati di stallo, posti tra una richiesta di lettura dalla memoria e l'effettivo utilizzo del dato ricevuto: questo è stato fatto sotto l'ipotesi che i ritardi di propagazione tra la RAM e il componente progettato siano inferiori a 1 ciclo di clock, che è il tempo occupato dagli stati di stallo.

Pertanto, la FSM effettivamente implementata è rappresentata dal seguente diagramma:

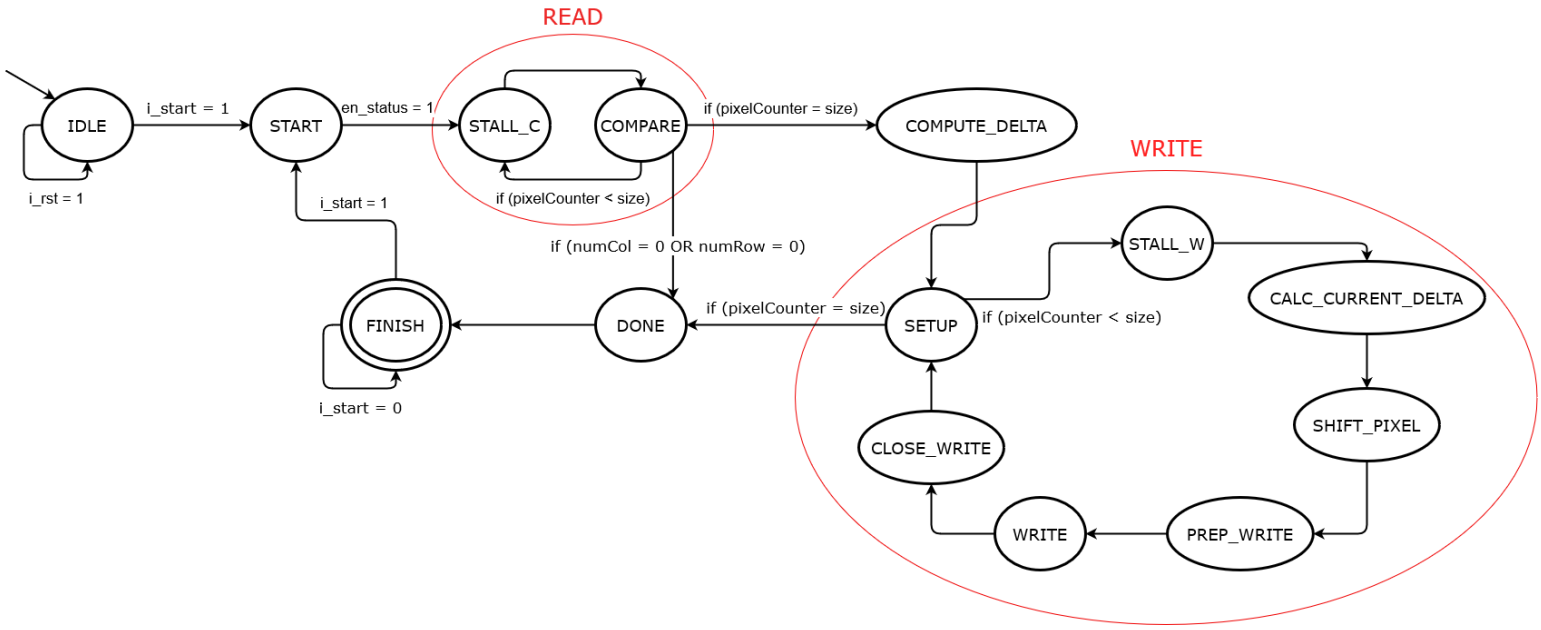

Figure 2: FSM implementata

Segue una descrizione degli stati formali della FSM:

- **IDLE** : Stato iniziale di idle in cui si posiziona la FSM al reset della computazione. La macchina attende che venga asserito il segnale *i_start*.
- **START** : Stato in cui si abilita la lettura attraverso un controllo sul segnale interno *en_status* che viene posto a 0 o 1 in contemporanea con il segnale *o_en*, allo stesso tempo si richiede in lettura il primo byte.

- **STALL_C** : Stato di stallo, in modo da poter sopportare ritardi di propagazione, di al massimo 1 ciclo di clock, sul segnale *i_data*.
- **COMPARE** : Stato in cui, se si stanno effettuando le letture delle dimensioni dell'immagine, queste vengono salvate nei segnali interni *numCol* e *numRow*, in modo da poter calcolare *size = numCol * numRow*. Se invece si sta effettuando la lettura dei pixel dell'immagine, questi vengono confrontati con il minimo e massimo parziale trovato fino a quell'istante, salvati rispettivamente in *minPixelValue* (inizializzato a 255) e *maxPixelValue* (inizializzato a 0). Se il pixel considerato è minimo o massimo parziale, il suo valore è assegnato a *minPixelValue* o *maxPixelValue* a seconda del caso.
- **COMPUTE_DELTA** : Stato in cui si giunge una volta trovati minimo e massimo assoluti dell'immagine. Qui vengono calcolati *deltaValue = maxPixelValue - minPixelValue* e *shiftValue = 8 –* $\lfloor(log_2(deltaValue + 1))\rfloor$.
- **SETUP** : Stato in cui si richiede in lettura un pixel dell'immagine originale, fino a quando tutti i pixel dell'immagine originale sono stati considerati.
- **STALL_W** : Stato di stallo, ha lo stesso scopo dello stato **STALL_C**.
- **CALC_CURRENT_DELTA** : Stato in cui si calcola *temp = currentPixelValue – minPixelValue*.
- **SHIFT_PIXEL** : Stato in cui si calcola *temp = temp <<shiftValue*.
- **PREP_WRITE** : Stato in cui, contemporaneamente:
  1. Si alza il segnale *o_we*, in modo da abilitare la RAM in scrittura.
  2. Si setta *o_address* all'indirizzo del pixel considerato nell'iterazione corrente + *numCol * numRow*, in modo da ottenere il corretto indirizzo su cui va scritto il nuovo valore corrispondente del pixel preso in considerazione.
  3. Si setta *o_data = min(temp, 255)*.
- **WRITE** : Stato in cui si effettua la scrittura
- **CLOSE_WRITE** : Stato in cui si abbassa il segnale *o_we*, in modo da poter leggere correttamente il valore del prossimo pixel, evitando scritture non desiderate.
- **DONE** : Stato in cui si giunge una volta che è stato equalizzato ogni pixel dell'immagine. In questo stato si alza il segnale *o_done*, in modo che *i_start* possa essere abbassato.

- **FINISH** : Stato in cui, una volta che è stato abbassato *i_start*, viene abbassato *o_done*. La FSM rimarrà in questo stato fino a quando sarà alzato nuovamente il segnale *i_start*, segnalando l'inizio di una nuova computazione.

Si noti che, per semplicità di rappresentazione, sono stati omessi gli archi che portano da ciascuno stato allo stato di **IDLE**, quando il segnale *i_rst* è posto a 1. Ciò deriva dal fatto che se durante qualsiasi stadio della computazione, viene dato un segnale di reset, la macchina deve tornare allo stato di **IDLE**.

2.3 Scelte implementative

Durante la scrittura del codice VHDL, sono state fatte alcune scelte implementative che potrebbero far discutere sulla correttezza dello stesso, in particolare:

1. Riga 140 : *size <= std_logic_vector(unsigned(numCol) * unsigned(numRow));*

   Si è scelto di calcolare la dimensione dell'immagine attraverso l'operatore *, in modo da avere un controllo facilitato, attraverso il segnale interno *pixelCounter*, nei momenti in cui il componente deve ciclare su tutti i pixel dell'immagine.
   Questa scelta è stata presa consapevolmente del fatto che, se l'operazione di moltiplicazione è implementata in questo modo, si sta esprimendo la necessità di calcolare tale prodotto in un solo ciclo di clock, che comporta una crescita molto rapida dei registri necessari per calcolare tale prodotto, come visto durante il corso di "Reti Logiche" tenuto dal prof. Salice. In seguito ad un'attenta analisi delle FPGA moderne, si è comunque intrapresa questa strada, tenendo in mente il fatto che un prodotto di 8x8 bit, non comporta una problematica significativa per i tool di sintesi utilizzati.

2. Riga 163 : *deltaValue <= std_logic_vector(unsigned(maxPixelValue) - unsigned(minPixelValue));*

   Si è scelto di calcolare tale sottrazione tra i vettori *currentPixel* e *minPixelValue* a 8 bit, utilizzando direttamente la loro codifica binaria naturale, senza doverli convertire in complemento a 2.
   Questa scelta è stata presa in virtú del fatto che, per ogni possibile input ben formato, vale l'ipotesi che i valori dei pixel non sono mai modificati durante l'esecuzione, e perció il minimo trovato e salvato in *minPixelValue* è tale per cui, per ogni pixel dell'immagine originale vale che :

   $$currentPixel \geq minPixelValue$$

   Per questo motivo, è sufficiente sottrarre i due vettori in codifica binaria naturale.

3 Test benches

I test effettuati hanno cercato di verificare il corretto funzionamento del componente, ponendo particolare attenzione a situazioni di memoria estreme. Sono stati creati testbench comprendenti le seguenti situazioni:

3.1 Test selezionati

- Immagine in ingresso tale per cui la sua dimensione è minima, ovvero *numCol = 0* e/o *numRow = 0*.
- Immagine in ingresso tale per cui la sua dimensione è massima, ovvero *numCol = numRow = 128* (**WORST_CASE IMAGE**).
- Immagine in ingresso tale per cui *deltaValue* è minimo, cioè *deltaValue = 0*, ovvero immagini tali per cui tutti i loro pixel hanno lo stesso valore di grigio. In particolare, sono state testate immagini monotone con valore di grigio posto a 0, 255, e altri casi in cui il tono di grigio dell'immagine è stato generato in maniera casuale.
- Immagine in ingresso tale per cui *deltaValue* è massimo, cioè *deltaValue = 255*, ovvero un'immagine che è giá distribuita su tutto l'insieme di valori assumibili e che pertanto non deve essere modificata.
- Test di computazione in cui, una volta che la macchina ha giá ricevuto il segnale di reset e di inizio, durante la computazione la macchina riceve nuovamente il segnale di reset e di inizio.
- Test di 10 immagini da equalizzare consecutivamente, senza mai ricevere un segnale di reset dopo quello iniziale, generate in modo aleatorio. (**10_CONSECUTIVE IMAGES**).
- Test di 10000 immagini generate in modo aleatorio, attraverso un tool autoprodotto (**10K_INDEPENDENT IMAGES**).

Questi test hanno evidenziato alcune criticitá nel codice iniziale e hanno guidato lo sviluppo della soluzione fino alla sua versione finale.
Alcune criticitá delle versioni iniziali del componente, evidenziate dai test selezionati comprendevano:

- Lettura completamente errata, per via dell'assenza degli stati di stallo.
- Nel caso di test con immagini consecutive senza reset, solo la prima immagine veniva equalizzata in modo corretto mentre l'output correlato con le rimanenti immagini era difficilmente comprensibile.

I risultati rilevanti sono riportati nella sezione 4.

4 Risultati sperimentali

4.1 Report di sintesi

Dal punto di vista dell'area, la sintesi riporta il seguente utilizzo dei componenti:

LUT :	252 (0.61% del totale)
FF :	173 (0.21% del totale)

Si è fatta particolare attenzione nella scrittura del codice in modo da evitare l'utilizzo di latch.

4.2 Risultato dei test bench

Si riportano i risultati di tempo legati a 2 dei 3 test di cui si specifica il nome nella sezione 3, nel caso di simulazione post-synthesis funtcional.

Simulazione **WORST_CASE IMAGE** :	14'747'550'100 ps $= \sim 15ms$
Simulazione **10_CONSECUTIVE IMAGES** :	4'386'750'100 ps $= \sim 4.3ms$

Per tutti i test riportati e i test successivi è stata effettuata la simulazione behavioural e successivamente la simulazione functional post-synthesis utilizzando un clock di periodo a 10ns, tutte portate a termine con successo.

Inoltre, si ritiene rilevante specificare che il test **10K_INDEPENDENT IMAGES**, ha testato il componente con 10'000 immagini di diverse dimensioni generate in modo aleatorio. Questo test è stato di particolare successo, in quanto il componente ha equalizzato correttamente il 100% delle immagini proposte.

5 Conclusione

Si ritiene che il componente progettato rispetti le specifiche, fatto che è stato verificato mediante estensivo testing sia casuale, che con testbench scritti manualmente.