Prova Finale - Reti Logiche (prof. William Fornaciari)

Corigliano Emilio 10627041 - 907936

A.A. 2020/21

1 Introduzione e spiegazione del problema

Il progetto consiste nella progettazione di un circuito per l'equalizzazione di immagini in scala di grigi a 256 valori mediante il linguaggio di definizione hardware VHDL, attraverso il software *Vivado*. L'equalizzazione è un processo di elaborazione digitale che consiste nell'elaborare l'immagine affinchè i suoi colori coprano tutto lo spettro disponibile. Nel nostro caso, dopo l'elaborazione vogliamo ottenere delle immagini che abbiano pixel di valore 0 e 255 rispettivamente nei punti di valore minimo e massimo nell'immagine originale; tutti gli altri pixel verranno ridistribuiti su tutto lo spettro in maniera coerente.

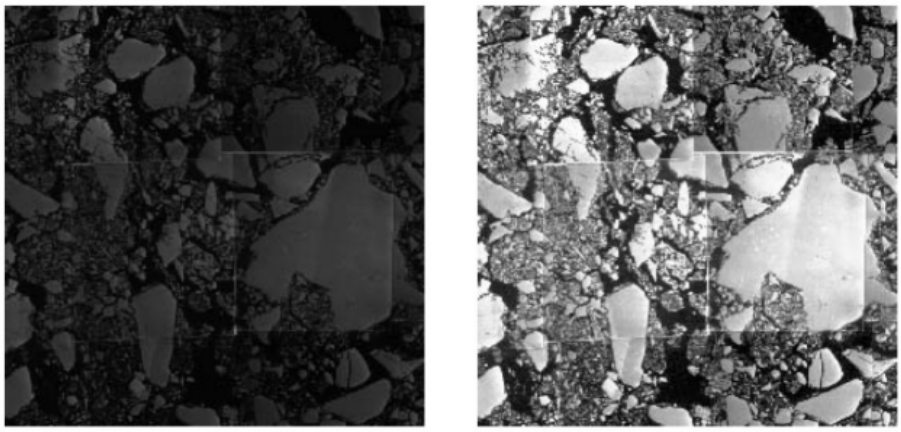

Figure 1: esempio di equalizzazione di un'immagine

2 Considerazioni, approccio e scelte implementative

Leggendo la specifica si possono fare delle prime considerazioni generali, per poi approfondire ogni aspetto particolare:

- Non è necessario distinguere le varie righe, si può trattare la memoria semplicemente come un vettore di dimensione $N = R \cdot C$ da analizzare sequenzialmente;

- Per le scansioni dell'immagine si è scelto di utilizzare un contatore che contasse N cicli di clock; Questo ha lo svantaggio di dover essere computato in una fase a sè antecedente alle altre ma il vantaggio di poter avere una corrispondenza immediata tra il contatore (i) e gli indirizzi di memoria da leggere $(2+i)$ o scrivere $(2+N+i)$, con $i \in [0, N)$;
- È necessario che si scansioni preliminarmente una volta tutta l'immagine così da trovare i byte di valore massimo (MAX_PV) e minimo (MIN_PV); questa fase deve necessariamente essere precedente alla vera e propria elaborazione dell'immagine poichè quest'ultima ha bisogno dell'elaborazione dello $SHIFT\_VALUE$ (computato a partire dalla differenza tra MAX_PV e MIN_PV).
- Infine, è necessario eseguire un'altra scansione dell'immagine per eseguire la vera e propria elaborazione.

2.1 Registri

Di seguito si elencano i diversi registri usati al fine di memorizzare importanti informazioni per l'elaborazione dell'immagine:

- rN: memorizza il numero totale di byte che compongono l'immagine
- $rMax$: memorizza il valore massimo dei pixel presente nell'immagine
- $rMin$: memorizza il valore minimo dei pixel presente nell'immagine
- rSL: memorizza lo $SHIFT\_LEVEL$ da adoperare per l'equalizzazione.

Altri registri sono stati usati per poter realizzare varie parti del componente:

- $r1$: è usato per memorizzare il numero da aggiungere ad ogni ciclo per l'algoritmo delle somme ripetute per il prodotto di $N = R \cdot C$;
- $rC1$: è il registro che permette di contare quante volte eseguire l'operazione di somme ripetute per computare N;
- $rC2$: è il registro che permette di contare l'indice per la valutazione del valore del pixel nella fase in cui si ricercano il valore massimo e minimo;
- $rC3$: è il registro che permette di contare l'indice del pixel da elaborare;

2.2 Fasi

Ci sono 4 fasi fondamentali che occorre attraversare per elaborare l'immagine, ognuna delle quali ha bisogno di dati elaborati precedentemente. Per questo non sono ulteriormente parallelizzabili.

Si è scelto di affrontare ogni fase singolarmente per semplificare l'esposizione; questo approccio ha semplificato anche l'implementazione, di conseguenza le diverse fasi sono

indipendenti tra loro. Seguendo questo approccio, però, si è limitata l'ottimizzazione della progettazione, facendo sì che alcuni componenti (come per esempio i contatori) non fossero in quantità minima ma sono stati replicati, in maniera leggermente diversa, per ogni fase in cui erano necessari. Questo però non è stato ritenuto un problema data la dimensione del progetto e la qualità dell'hardware per il quale è stato progettato; si è preferita la qualità e semplicità espositiva alla ottimizzazione progettuale.

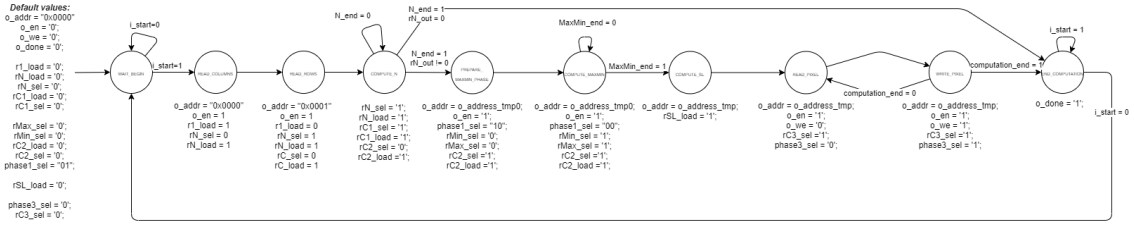

Figure 2: FSM dell'elaborazione

2.2.1 computazione della quantità di pixel

La prima fase è quella in cui si calcola la dimensione dell'immagine N da equalizzare. Questo valore è ottenuto moltiplicando $mem[0]$ e $mem[1]$ tramite un moltiplicatore per somme successive. Alla fine dell'operazione viene portato in alto il segnale N_end che indica la fine dell'operazione. Da questo momento fino alla prossima esecuzione nel registro rN ci sarà il valore $N = R \cdot C$.

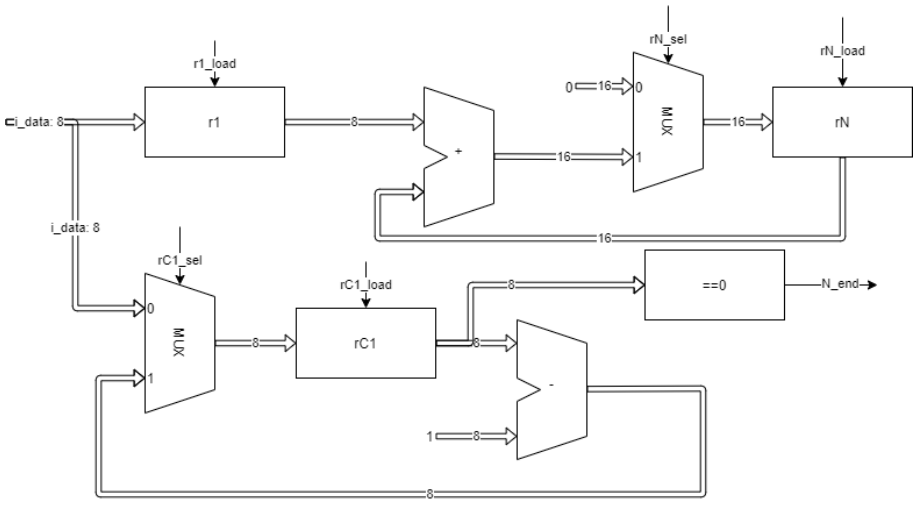

Figure 3: computazione della quantità di pixel

2.2.2 Ricerca dei valori massimi e minimi dei pixel

Nella seconda fase si cercano il valore massimo (MAX_PV) e il valore minimo (MIN_PV): si scorre una prima volta tutta l'immagine e si cercano contemporaneamente il valore massimo e minimo dei pixel presenti nell'immagine. Per tenere conto dei pixel letti e per generare l'indirizzo al quale leggere il valore si usa un contatore che parte da 0 e ad ogni ciclo di clock incrementa di 1, fino a raggiungere N; a questo punto verrà portato in alto il segnale di $MaxMin\_end$.

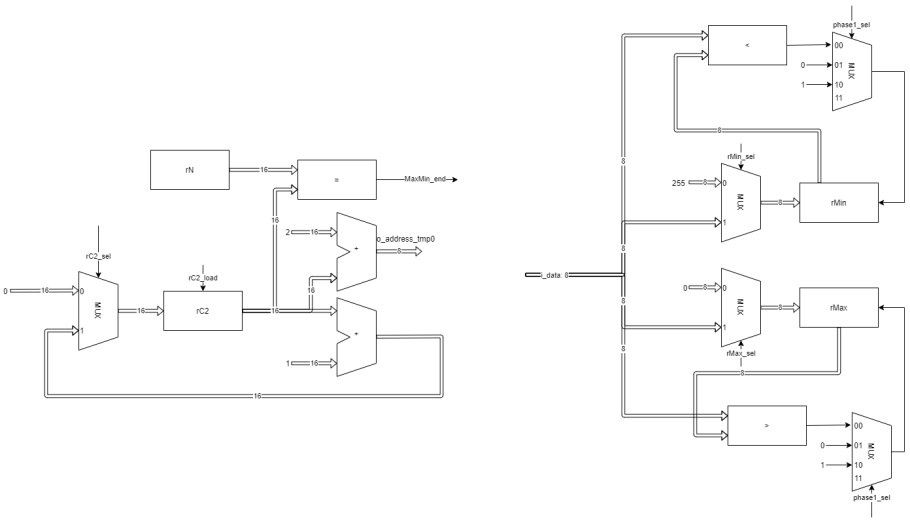

Figure 4: Ricerca dei valori massimi e minimi dei pixel

2.2.3 Computazione del valore SHIFT_LEVEL

La terza fase prevede il calcolo dello $SHIFT\_LEVEL$; questo viene ottenuto implementando una funzione combinatoria che esegue un controllo a soglie (nell'immagine rappresentata con il blocco $f(x)$ avente come input 8 bit e come output 4 bit). La funzione combinatoria prende in input $MAX\_PV - MIN\_PV + 1$ (quindi il $DELTA\_VALUE$ incrementato di 1) perchè si era pensato che sarebbe stato più semplice la generalizzazione. Infatti visualizzando gli shift level corrispondenti a tutti i valori di delta value possibili incrementati di uno (per esempio grazie allo script 1), si nota una semplice correlazione tra la posizione in cui compare il primo 1 e il valore dello $SHIFT\_LEVEL$ corrispondente.

Listing 1: Generazione di soglie

```
for i in range(256):
        print(bin(i) + ": " + str(8 - math.floor(math.log2(i+1))))
```

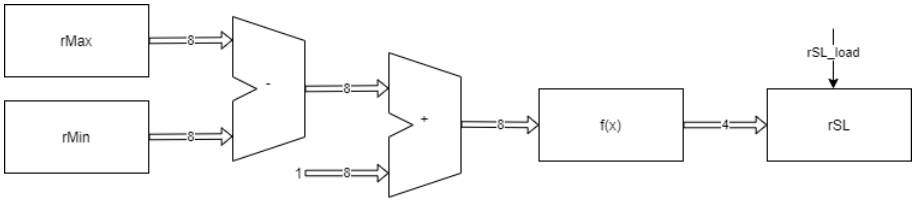

Figure 5: Computazione dello shift level

2.2.4 Equalizzazione

La quarta fase, quella finale, provvede a equalizzare l'immagine tramite l'algoritmo fornitoci. Per ogni pixel che compone l'immagine (il cui indirizzo è computato grazie ad un contatore che parte da $N-1$ e decrementa fino a 0) si legge il valore originale. Al ciclo successivo si provvede a scrivere in memoria il valore computato all'indirizzo del pixel originale incrementato di N. Sono necessari due cicli per ogni pixel da equalizzare poichè la memoria prevede un solo segnale per codificare l'indirizzo su cui leggere o scrivere; visto che l'immagine originale non va sovrascritta si deve alternare una fase di lettura con una di scrittura, nelle quali cambierà *o_address*.

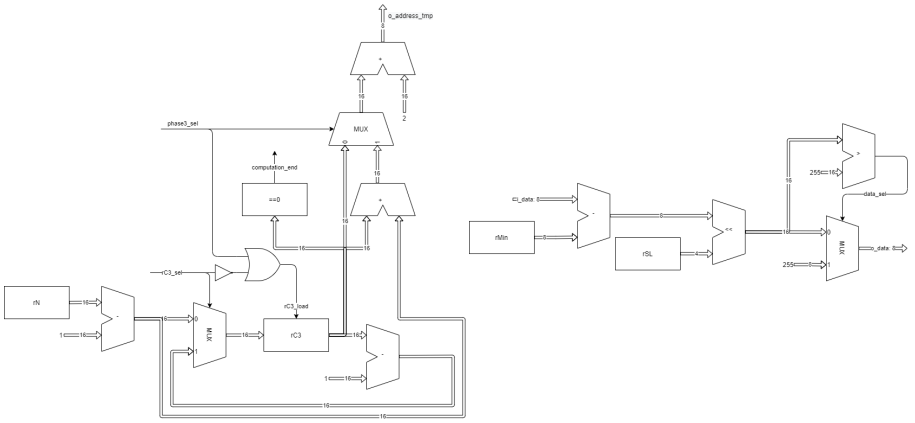

Figure 6: Equalizzazione

3 Risultati

Il componente è stato sviluppato, sintetizzato e testato sia con simulazioni behavioural che post-synthetis.

3.1 testing

Per testare il componente sviluppato sono stati scritti (con l'aiuto di script in python) dei testbanch. Mentre alcuni erano generati con dati random, altri sono stati creati per

controllare dei corner-cases o dei punti critici del codice. Di seguito sono leggermente approfonditi vari testbanch usati sul componente:

- dati casuali: diversi test sono stati implementati con dimensione e immagine iniziale casuali; in questi si includono anche tutti i testbanch fornitoci in allegato alla specifica;
- immagini grandi: vari test con dimensione dell'immagine molto grande (uno pure con dimensione 128x128, cioè la dimensione massima) e immagine generata casualmente; questo serve a testare in particolar modo se le dimensioni dei collegamenti tra i vari componenti sono sufficientemente grandi e se la fase di calcolo della dimensione dell'immagine gestisce il caso limite della dimensione massima.
- immagine di un pizel: test per analizzare il comportamento nel caso in cui ci sia solo un pixel;
- zero righe o zero colonne: due test per provare il funzionamento del componente un dato tra righe o colonne è 0. Utile per testare un caso limite per la fase di calcolo della dimensione dell'immagine;
- stesso valore in tutta l'immagine: usato per testare il comportamento nel caso di un solo valore uguale per tutti i pixel dell'immagine. Sollecita un corner case nel calcolo dello $SHIFT\_LEVEL$;
- valori immagine da 0 a 255: sollecita un altro corner case nel calcolo dello $SHIFT\_LEVEL$
- più immagini consecutive: testato il caso in cui ci sono più immagini consecutive, la seconda caricata in memoria dopo la prima elaborazione e così via

3.2 Report di sintesi

Analizzando il report post sintesi sui timing si evince che il componente rispetta i constraints imposti: di 10ns se ne usano 4.610ns, questo ci suggerisce che il componente potrebbe tranquillamente funzionare al doppio dela frequenza, lasciando comunque margine per far stabilizzare tutte le uscite prima della fine del ciclo di clock. Inoltre, analizzando il report sugli utilizzi, non si trovano latch inferiti.