Prova Finale di Reti Logiche

Flavio Rizzoglio (10611773)
Gianluca Ruberto (10607366)

1 Aprile 2021

Sommario

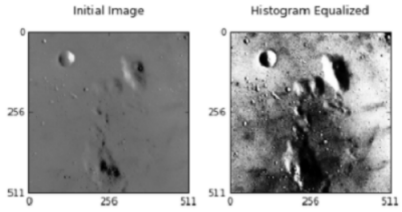
1 Introduzione

1.1 Scopo del Progetto

Il progetto è ispirato al metodo di equalizzazione dell'istogramma di una immagine. Questo metodo è pensato per ricalibrare il contrasto di una immagine quando l'intervallo dei valori di intensità sono molto vicini effettuandone una distribuzione su tutto l'intervallo di intensità, al fine di incrementare il contrasto. Di seguito in *Figura 1* e *Figura 2* alcuni esempi di funzionamento.

Figure 1: fonte Wikipedia

Figure 2: fonte towardsdatascience



1.2 Specifiche Progettuali

Nel progetto effettuato non è richiesta l'implementazione dell'algoritmo standard ma una sua versione semplificata. Infatti, l'algoritmo di equalizzazione semplificato usato verrà applicato ad immagini in scala di grigi a 256 livelli e funziona nel modo seguente:

```
DELTA_VALUE = MAX_PIXEL { MIN_PIXEL
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 il massimo e minimo valore dei pixel dell'immagine, `CURRENT_PIXEL_VALUE` è il valore del pixel da trasformare, e `NEW_PIXEL_VALUE` è il valore del nuovo pixel. Le dimensioni dell'immagine, ciascuna di dimensione di 8 bit, sono memorizzate in una memoria con indirizzamento al byte partendo dalla posizione zero: il byte in posizione

0 si riferisce al numero di colonne `(N-COL)`, il byte in posizione 1 si riferisce al numero di righe `(N-RIG)`. I pixel dell'immagine, ciascuno di 8 bit, sono memorizzati in memoria con indirizzamento al byte partendo dalla posizione 2. I pixel della immagine equalizzata, sempre di 8 bit, sono memorizzati in memoria con indirizzamento al Byte partendo dalla posizione calcolata nel modo seguente: `2+(N-COL*N-RIG)`.

1.3 Interfaccia componente

Il componente descritto ha 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;
```

In cui abbiamo i segnali:

- **i_clk**: Segnale di `CLOCK` in ingresso generato dal test bench corrente;
- **i_rst**: Segnale di `RESET` che inizializza la macchina, rendendola pronta a ricevere il primo segnale di `START`;
- **i_start**: Segnale di `START` generato dal test bench;
- **i_data**: Segnale (vettore) che arriva dalla memoria in seguito ad una richiesta di lettura;
- **o_address**: Segnale (vettore) di uscita che manda l'indirizzo alla memoria;
- **o_done**: Segnale di uscita che comunica la fine dell'elaborazione e il dato di uscita scritto in memoria;
- **o_en**: Segnale di `ENABLE` da dover mandare alla memoria per poter comunicare (sia in lettura che in scrittura);
- **o_we**: Segnale di `WRITE ENABLE` da dover mandare alla memoria per poterci scrivere. Deve essere uguale a 1 per scrittura in memoria e 0 per lettura da memoria;
- **o_data**: Segnale (vettore) di uscita dal componente verso la memoria.

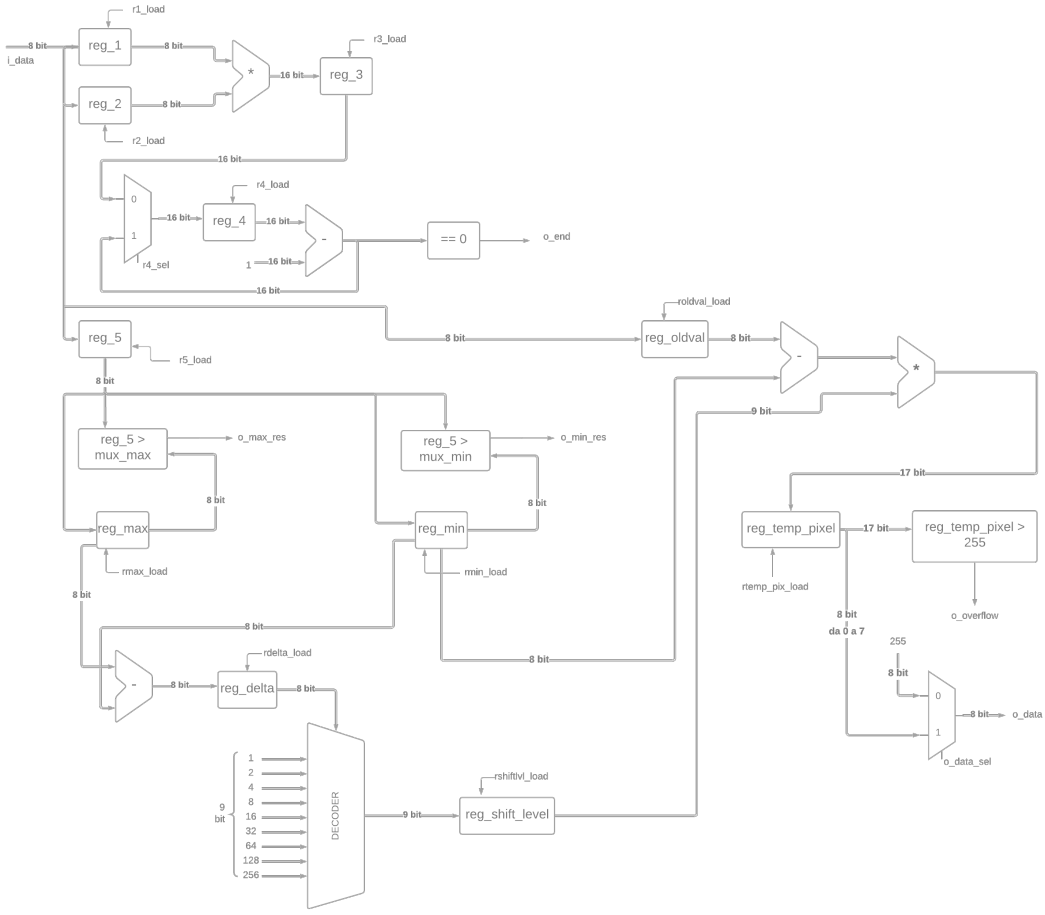
2 Architettura

Per realizzare il progetto è stato deciso di sfruttare una architettura con un datapath e una macchina a stati che ne controlli il flusso. Di seguito viene spiegato il funzionamento di tali componenti.

2.1 Datapath

Il datapath è così formato (*Figura 3*)

Figure 3: Datapath



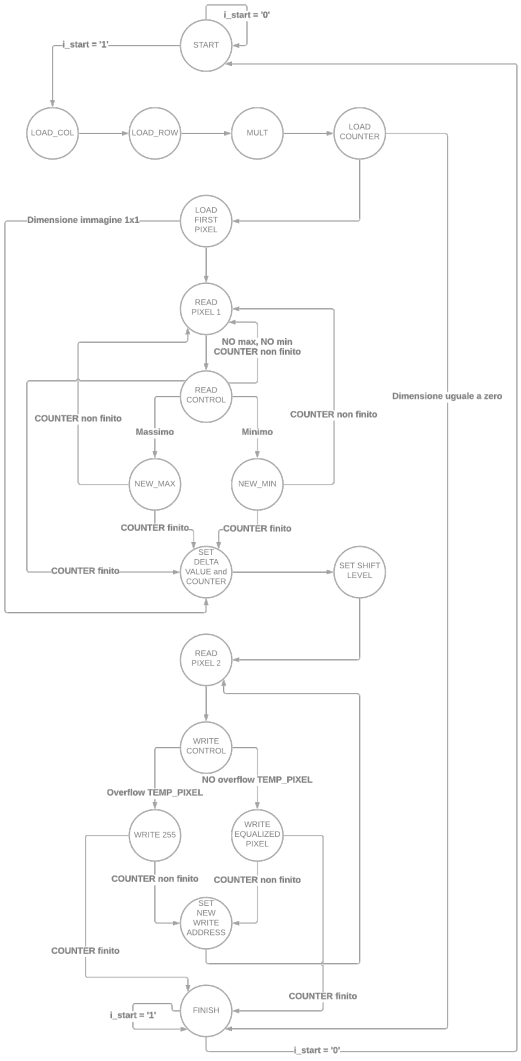
La sezione del datapath comprendente il registro `reg_4`, il multiplexer, il sommatore e un comparatore implementa un ciclo di conteggio usato sia in lettura che in scrittura.
La sezione composta da `reg_5`, `reg_max`, `reg_min` e i due comparatori im-

plementa la ricerca di massimo e minimo. Il loro risultato viene utilizzato per calcolare il `DELTA_VALUE` e `SHIFT_LEVEL`, come è visibile nella parte inferiore del datapath.
La logica di equalizzazione dei pixel, invece, è presente nella sezione del datapath a destra del registro `reg_old_val`.

2.2 Macchina a Stati

La macchina a stati utilizzata è così formata (*Figura 4*):

Figure 4: Macchina a stati



Innanzitutto va precisato che la macchina a stati non è certamente minima e potrebbe essere ulteriormente ottimizzata (vedi sezione Ottimizzazioni). Passiamo dunque alla descrizione del ruolo e del funzionamento di ogni singolo stato:

- **START**: Attendo che il segnale `i_start` diventi uguale a uno per poter iniziare ad equalizzare l'immagine;
- **LOAD_COL**: Carico nel registro apposito (`reg_1`) la dimensione delle colonne;
- **LOAD_RIG**: Carico nel registro apposito (`reg_2`) la dimensione delle righe;
- **MULT**: Calcolo la dimensione totale dell'immagine e la carico in un apposito registro (`reg_3`) che terrò salvato per utilizzi successivi;
- **LOAD_COUNTER**: Carico la dimensione totale dell'immagine (per il ciclo di conteggio) e il primo pixel in due specifici registri (rispettivamente `reg_4` e `reg_5`). A questo punto, se la dimensione dell'immagine è uguale a zero (edge case) passo direttamente allo stato `FINISH`;
- **LOAD_FIRST_PIXEL**: Prendo il primo pixel caricato e lo imposto come massimo e minimo corrente in modo da facilmente effettuare i controlli successivi. Inoltre se l'immagine è composta da un solo pixel, passo allo stato `SET_DELTA_and_COUNTER`;
- **READ_PIXEL_1**: In questo stato leggo i pixel e li carico nel registro `reg_5` per poi effettuarne il confronto con i massimi e minimi correnti;
- **READ_CONTROL**: Effettuo il controllo con i massimi e minimi correnti e a seconda del risultato, scelgo se proseguire nello stato `NEW_MAX` in caso di massimo, `NEW_MIN` in caso di minimo. In caso non debba aggiornare massimi o minimi, passo a `READ_PIXEL_1` o in `SET_DELTA_and_COUNTER` quando ho letto tutti i pixel;
- **NEW_MAX**: Nel caso in cui raggiunga questo stato, carico il pixel corrente nel registro apposito che salva il pixel massimo (`reg_max`). Se il ciclo di conteggio è terminato, passo allo stato `SET_DELTA_and_COUNTER`, altrimenti torno torno in `READ_PIXEL_1`;
- **NEW_MIN**: Nel caso in cui raggiunga questo stato, carico il pixel corrente nel registro apposito che salva il pixel minimo (`reg_min`). Se il ciclo di conteggio è terminato, passo allo stato `SET_DELTA_and_COUNTER`, altrimenti torno torno in `READ_PIXEL_1`;
- **SET_DELTA_and_COUNTER**: Calcolo il `DELTA_VALUE` e lo carico nel registro `reg_delta_value`. Imposto inoltre i parametri per reiniziare il ciclo di conteggio così da effettuare correttamente la scrittura in memoria;

- **SET_SHIFT_LEVEL**: Calcolo lo `SHIFT_LEVEL` necessario per la equalizzazione e lo carico nel registro `reg_shift_level`. Carico inoltre il primo pixel nel registro `reg_old_val`, pronto per l'equalizzazione;
- **READ_PIXEL_2**: Equalizzo il pixel precedentemente caricato nel registro `reg_old_val` e assegno il valore calcolato al registro `reg_temp_pixel`. Contemporaneamente (alla fine del ciclo di clock) carico il pixel successivo in `reg_old_val`;
- **WRITE_CONTROL**: Controllo che il pixel appena equalizzato non sia maggiore di 255 (overflow) e di conseguenza scelgo in quale stato proseguire (guardando il segnale `o_overflow`);
- **WRITE_255**: Scrivo su `o_data` 255 a causa dell'overflow. Se ho finito i pixel da equalizzare vado allo stato `FINISH`, altrimenti proseguo in `SET_NEW_WRITE_ADDRESS`;
- **WRITE_EQUALIZED_PIXEL**: Scrivo su `o_data` gli 8 bit meno significativi del contenuto di `reg_temp_pixel`. Se ho finito i pixel da equalizzare vado allo stato `FINISH`, altrimenti proseguo allo statto successivo `SET_NEW_WRITE_ADDRESS`;
- **SET_NEW_WRITE_ADDRESS**: Imposto il giusto `o_address` per effettuare correttamente l'operazione di lettura e contemporaneamente proseguo con il ciclo di conteggio;
- **FINISH**: Imposto il segnale `o_done` a uno e aspetto che il segnale `i_start` vada a zero per tornare allo stato `START` ed essere pronto a leggere la prossima immagine o terminare l'esecuzione.

2.3 Ottimizzazioni

Nella prima versione del progetto, `o_address` veniva aggiornato in ogni stato basandosi sul valore dello stesso all'indirizzo precedente. Come si può intuire questo dava origine a diversi latch e quindi abbiamo implementato una funzione che le lega l'indirizzo allo stesso allo stato.

Ulteriori ottimizzazioni si sono concentrate sulla riduzione del numero di stati (soprattutto all'interno dei cicli) nella macchina a stati e sulla riduzione del numero di componenti di memoria nel datapath.

3 Risultati Sperimentali

3.1 Sintesi

La sintesi procede senza problemi, senza errori o warning di alcun tipo. Immettendo il comando **report utilization** nella TCL Console di Vivado è possibile ottenere un report comprensivo di diversi dettagli utili sul progetto:

Figure 5: comando report_utilization

Site Type	Used	Fixed	Available	Util%
Slice LUTs*	314	0	134600	0.23
LUT as Logic	314	0	134600	0.23
LUT as Memory	0	0	46200	0.00
Slice Registers	119	0	269200	0.04
Register as Flip Flop	119	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

La sezione Slice Logic del **report utilization** illustra il tipo di componenti usati: è possibile notare, come facilmente prevedibile data la specifica del progetto, che la percentuale di utilizzo dei vari tipi di componenti è molto bassa e che il circuito non ha Latch.

Il comando **report timing**, invece, fornisce informazioni riguardo al timing del circuito. Di seguito in *Figura 6* un riassunto dell'output del comando:

Figure 6: riassunto comando report_timing

	Slack (MET) :	Requirement:	Data Path Delay:
Max Delay Paths	92.408ns	100.000ns	7.474ns
Min Delay Paths	0.167ns	0.000ns	0.295ns

Abbiamo scelto come required time 100ns in accordo con le specifiche. Nella sezione Max Delay Path è possibile trovare informazioni riguardanti il percorso critico. Lo slack è di 92.408ns e di conseguenza il percorso critico risulta essere 7.474ns, perfettamente in linea con il periodo di clock richiesto.

3.2 Simulazioni

Le prime simulazioni effettuate sono state quelle presenti nella documentazione, successivamente prima ancora dei corner case abbiamo testato oltre dodicimila immagini diverse, di varie dimensioni e con distribuzioni di pixel differenti, utilizzando uno script per generare il tutto. Successivamente sono stati analizzati i seguenti casi limite:

- Immagine con dimensioni massime;
- Immagine con dimensioni nulle;
- Immagine con pixel tutti uguali;
- Immagine con un solo pixel;
- Immagine con `DELTA_VALUE` 255;
- Reset asincrono del circuito.

È bene precisare che le immagini sono state testate consecutivamente, a dimostrazione che il circuito è in grado di continuare a lavorare dopo aver processato un'immagine.

3.2.1 Immagine con dimensioni massime

Per quanto un'immagine con dimensioni massime non sia un caso effettivamente complicato è necessario testare anche questa condizione poiché ci ritroviamo al limite del dominio di input.

3.2.2 Immagine con dimensioni nulle

In questo testbench sono state testate tutte e tre le possibilità che permettono di avere un'immagine a dimensioni nulle, ovvero:

- Immagine 0x0;
- Immagine 0xN;
- Immagine Nx0.

Anche in questo caso abbiamo scelto di testare questa casistica poiché al limite del dominio di input.

3.2.3 Immagine con pixel tutti uguali

Questo testbench rappresenta un caso particolare poiché è al limite del dominio del `DELTA_VALUE`. Sono stati testati diversi sotto casi:

- Immagine con pixel tutti uguali a 0;

- Immagine con pixel tutti uguali a 255;
- Immagine con pixel tutti uguali a N (dove N è un numero generico tra 0 e 255).

Ovviamente ogni volta l'immagine equalizzata risultava di tutti 0, risultato della versione semplificata dell'algoritmo di equalizzazione dell'istogramma.

3.2.4 Immagine con un solo pixel

Abbiamo scelto di testare immagini di dimensione 1x1 poiché rappresentano il limite del dominio che viene effettivamente processato e, in quanto sotto caso banale dell'immagine con pixel tutti uguali, è necessario che il risultato sia un unico pixel di valore 0.

3.2.5 Immagine con `DELTA_VALUE` 255

L'immagine contenente sia pixel a 0 che pixel a 255 rappresenta un caso limite del dominio di `DELTA_VALUE`, in questo caso l'immagine equalizzata è uguale all'immagine di partenza.

3.2.6 Reset asincrono del circuito

Questo testbench verifica che durante l'equalizzazione di un'immagine generica, in caso di reset, il circuito è in grado di riportarsi allo stato di partenza, riuscendo poi a ripartire successivamente in tranquillità.

4 Conclusioni

Il componente passa tutti i testbench precedentemente descritti sia in fase di pre-sintesi che un fase di post-sintesi, realizzando quindi un circuito in grado di equalizzare immagini in scala di grigi a 256 livelli.

Come conclusione finale, il VHDL ci è sembrato un linguaggio molto particolare, infatti non è un semplice linguaggio di programmazione ma un linguaggio di descrizione hardware. Questo ci ha obbligato a ragionare in modo molto diverso e sarà sicuramente utile per il futuro.