Prova Finale di Reti Logiche

Gabriele Ginestroni 10687747 907770
Giacomo Gumiero 10610435 907829

10 Maggio 2021

Docente: Fabio Salice
Anno accademico: 2020/2021

Indice

1 Requisiti di Progetto

Oggetto della specifica del progetto è l'implementazione in VHDL di un algoritmo semplificato di equalizzazione dell'istogramma di un'immagine collocata in memoria. L'algoritmo è applicato a immagini in scala di grigi a 256 livelli.

Il valore di ogni pixel dell'immagine di output è ottenuto come segue:

(a) $\texttt{delta\_value} = \texttt{max\_pixel\_value} - \texttt{min\_pixel\_value}$

(b) $\texttt{shift\_level} = 8 - \lfloor \log_2(\texttt{delta\_value} + 1) \rfloor$

(c) $\texttt{temp\_pixel} = (\texttt{current\_pixel\_value} - \texttt{min\_pixel\_value}) << \texttt{shift\_level}$

(d) $\texttt{new\_pixel\_value} = \min(255, \texttt{temp\_pixel})$

L'implementazione deve leggere da memoria in modo sequenziale, riga per riga, l'immagine da elaborare.
In particolare, ogni byte rappresenta un pixel, ad eccezione dei primi due. Questi, memorizzati a partire dall'indirizzo 0, contengono infatti le dimensioni dell'immagine, come mostrato in figura (1).
Si suppone vera l'ipotesi che la dimensione massima dell'immagine sia pari a 128×128 pixel.

L'output deve essere scritto in memoria negli indirizzi immediatamente successivi a quelli dell'immagine originale.

Il componente da descrivere 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;
```

1.1 Esempio di funzionamento

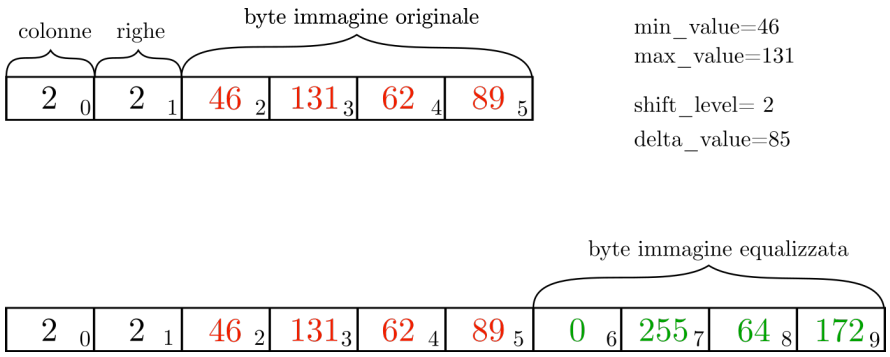

Fig. 1: Esempio di funzionamento dell'algoritmo di equalizzazione

2 Architettura

Per l'implementazione è stato scelto di realizzare un **datapath** affiancato da una **macchina a stati finiti** (FSM) che svolge il ruolo di unità di controllo.

2.1 Implementazione ad alto livello

L'elaborazione è costituita da due fasi principali.
La prima consiste in una lettura sequenziale di tutti i pixel dell'immagine originale per ottenere i valori massimo e minimo. Questo passo si conclude con la computazione del delta_value e dello shift_level.

La seconda e ultima fase consiste in un ciclo di lettura-scrittura durante la quale vengono riletti i pixel originali ed elaborati i rispettivi nuovi valori usando i risultati del passo precedente.

2.2 Datapath

Il datapath progettato è costituito da tre *component*, realizzati con specifica dataflow:

1. **DATAPATH_ZERO** è il componente combinatorio che si occupa di:
   (a) Computazione dei valori massimo e minimo dei byte dell'immagine originale presente in memoria. Questi valori vengono aggiornati a ogni iterazione del ciclo di lettura iniziale scandita dal datapath_address
   (b) Calcolo del *delta_value*

2. **DATAPATH_OUT** è il componente combinatorio che si occupa di:
   (a) Calcolo dello $shift\_level$
   (b) Lettura dei byte ed elaborazione dei rispettivi nuovi valori da scrivere in memoria durante la fase finale della computazione
3. **DATAPATH_ADDRESS** è il componente combinatorio che si occupa di:
   (a) Calcolo della dimensione totale dell'immagine presente in memoria
   (b) Fornire gli indirizzi di memoria durante la fase iniziale di lettura sequenziale
   (c) Generare gli indirizzi di memoria durante la fase finale di lettura-scrittura
   (d) Alzare il segnale *end_loop* alla fine del ciclo di lettura sequenziale
   (e) Alzare il segnale *tmp_done* al termine del ciclo di lettura-scrittura per segnalare alla FSM la fine della computazione

I component sopra descritti si interfacciano con registri realizzati per mezzo di un unico *process*.

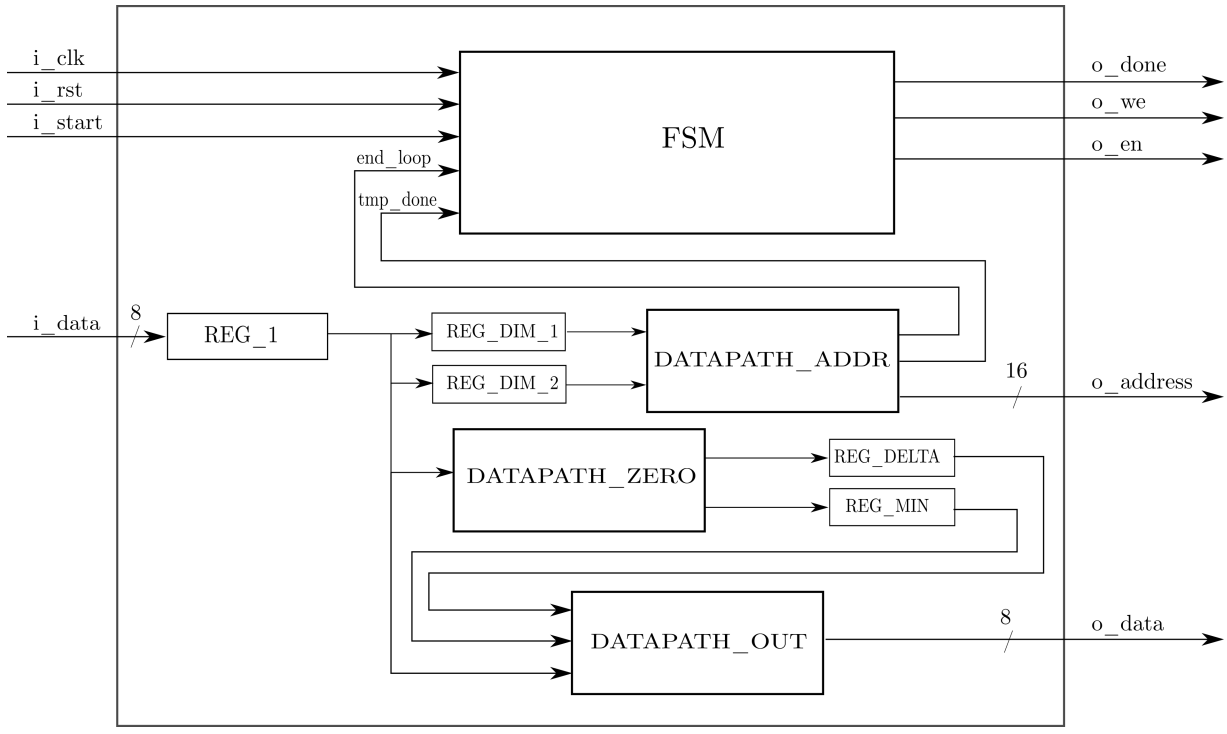

Fig. 2: Schema strutturale ad alto livello dell'implementazione

2.3 Finite State Machine

La macchina a stati finiti progettata coordina l'interazione tra la RAM e il datapath, gestendo lo stato dell'elaborazione. Si occupa inoltre di abilitare/inibire la scrittura di ogni registro. La FSM è stata realizzata tramite tre *process*, con specifica behavioral.
Segue una descrizione dei process che ne fanno parte:

1. **STATE_REG** si occupa di commutare lo stato corrente sul fronte di salita del clock e gestire un eventuale reset asincrono, riportando la macchina allo stato iniziale.

2. **NEXT_STATE** computa lo stato prossimo della FSM, in funzione dello stato attuale e degli ingressi del circuito.

3. **FSM_OUT** calcola le uscite della FSM in funzione del solo stato corrente. Segue che la macchina implementata è di MOORE .

Si descrive ora il diagramma sintetico degli stati della FSM.

- `WAIT_FOR_START`: Stato iniziale in cui la macchina attende che venga alzato il segnale **i_start** per iniziare una nuova elaborazione. In caso di reset asincrono la FSM viene riportata in questo stato.

- `COMPUTE_IMG_SIZE`: Stato in cui vengono lette da memoria le dimensioni dell'immagine e computata la dimensione totale.

- `READ_BYTE`: Stato in cui viene letto da RAM un singolo byte dell'immagine originale (a partire dall'indirizzo 2 della memoria).

- `COMPUTE_MAX_MIN`: Stato in cui vengono aggiornati i valori massimo e minimo dei byte dell'immagine, confrontando il valore appena letto con massimo e minimo attuali.

- `COMPUTE_DELTA`: Stato in cui viene calcolato il $delta\_value$.

- `COMPUTE_SHIFT`: Stato in cui viene calcolato lo $shift\_value$.

- `SECOND_READ_BYTE`: Stato in cui viene riletto da RAM un singolo byte (a partire dall'indirizzo 2 della memoria) dell'immagine originale per consentire l'elaborazione successiva del nuovo valore.

- `COMPUTE_NEW_PIXEL_VALUE`: Stato in cui viene elaborato il new_pixel_value, tramite i valori di delta e shift precedentemente calcolati.

- `WRITE_NEW_PIXEL`: Stato in cui viene accodato in memoria il nuovo valore del pixel, in posizione $i+img\_size$, con i indirizzo del byte letto.

- `DONE`: Stato in cui termina la computazione e viene asserito **o_done** a 1. La macchina resta in attesa che venga abbassato il segnale **i_start** a 0 per riportare la macchina allo stato iniziale, in attesa di un'eventuale nuova elaborazione.

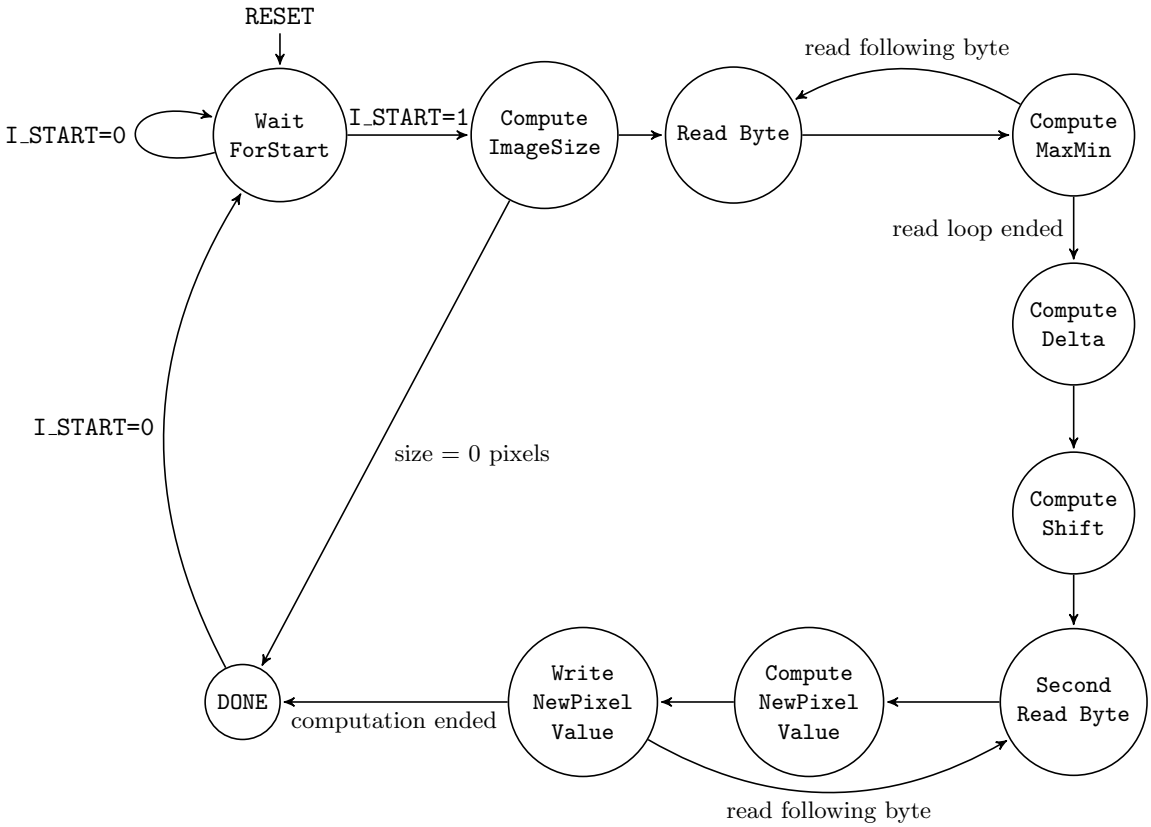

Fig. 3: Diagramma semplificato della macchina a stati finiti

3 Risultati sperimentali

3.1 Report utilization

Table 1: Slice Logic report utilization

Site type	Used	Fixed	Available	Utilization
Slice LUTs	248	0	134600	0.18%
LUT as Logic	248	0	134600	0.18%
LUT as Memory	0	0	46200	0.00%
Slice Registers	105	0	269200	0.04%
Register as Flip Flop	105	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%

In tabella sono riportati i risultati del *report_utilization* riguardanti il circuito sintetizzato.
Si può osservare l'assenza di latch inferiti e come l'implementazione occupi un'esigua porzione di FPGA.

3.2 Report timing

Table 2: Report timing

Slack(MET)	92.899 ns
Data Path Delay	6.950 ns
Requirement	100.000 ns

Dal report timing emerge che il componente sintetizzato rispetta ampiamente i vincoli di clock richiesti dalla specifica di progetto (periodo di clock al più pari a 100 ns).
Dai test eseguiti il circuito è risultato in grado di lavorare anche a frequenze più elevate.

4 Testbench

Il componente è stato esaurientemente testato tramite specifici testbench che coprono sia casi d'uso più frequenti sia edge case. In particolare, sono stati testati i seguenti casi limite:

- 1 pixel
- 0 pixel
- 128x128 pixel
- immagini multiple con reset
- immagini multiple senza reset
- reset asincrono
- particolari distribuzioni di valori dei pixel:
  (a) tutti pixel uguali
  (b) pixel con valori molto vicini
  (c) pixel con valori molto distanti
  (d) tutti pixel con valore 255
  (e) tutti pixel con valore 0
  (f) pixel con valore massimo in prima/ultima posizione
  (g) pixel con valore minimo in prima/ultima posizione

I test manuali eseguiti sfruttano sia tecnica white-box che black-box.
Per una fase di testing ancora più esaustiva è stato usato anche un generatore di test casuali, in grado di coprire una vasta gamma di casi d'uso.

5 Conclusioni

Il componente sviluppato ha superato con successo i test sia in simulazione *Behavioral* che in simulazione *Post-Synthesis Functional*, rispettando i vincoli della specifica.

Inoltre, dato l'approccio strutturale usato in fase di progettazione, dovrebbe risultare particolarmente facile un'eventuale espansione del componente.
Questo metodo di sviluppo, in aggiunta, ci ha permesso di scrivere il codice in modo modulare e indipendente, ottimizzando il lavoro del gruppo.