Progetto Finale di Reti Logiche

A.A. 2020/21

Ottavia Belotti -
Javin Barone -

Prof. William Fornaciari

POLITECNICO
MILANO 1863

Indice

Introduzione

Il progetto riguarda l'equalizzazione dell'istogramma di un'immagine al fine di aumentarne il contrasto e rendere la distribuzione della gamma cromatica più equilibrata.

L'algoritmo ricerca i valori massimi e minimi dei pixel e computa un nuovo valore per ogni pixel secondo la procedura di equalizzazione riportata nella sezione successiva.
L'immagine equalizzata verrà salvata in memoria immediatamente dopo il contenuto dell'immagine originale.

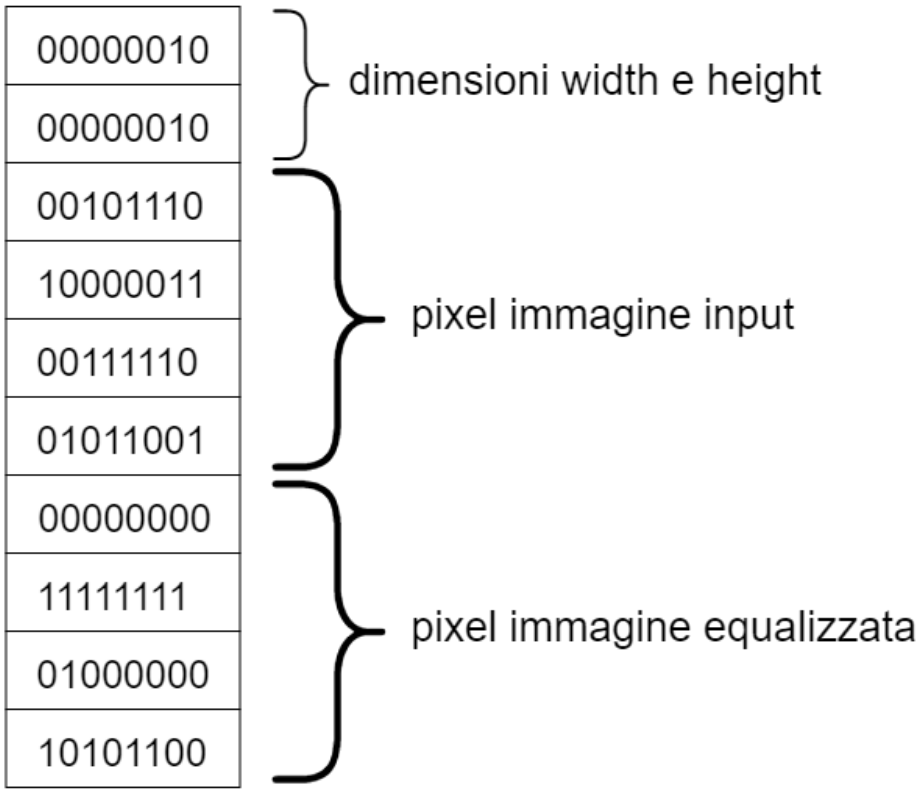

Fig. 1 - Diagramma delle celle di memoria

Premesse

- Ogni immagine è composta da pixel appartenenti alla scala di grigi a `256` livelli.
- Ogni pixel dell'immagine è rappresentato su 1 byte.
- Per limitazioni di indirizzamento a memoria (16 bit), le immagini hanno una grandezza massima di 128px * 128px.
- Le dimensioni delle immagini sono date agli indirizzi `0x0` e `0x1`, `0x0` per la dimensione colonna e `0x1` per riga.

Ipotesi progettuali

- Affinché la prima computazione possa essere avviata, il segnale di start deve essere preceduto dal segnale di reset. In caso di immagini successive, il segnale di reset non è più necessario.
- Se la macchina riceve un segnale di reset durante una computazione, essa si ferma e si prepara a ricevere una nuova immagine.
- Affinché il processo ricominci, il segnale `i_start` deve essere riportato a 0. Una volta fatto ciò, la macchina annulla `o_done` cosicché `i_start` possa essere nuovamente impostato a 1.
- Nel caso di un'immagine degenere, ovvero con una o entrambe le dimensioni nulle, l'algoritmo termina.

Architettura

L’architettura del progetto prevede un unico modulo organizzato in due processi: `DELTA_LAMBDA` implementante la logica combinatoria della macchina a stati finiti (FSM) e `STATE_OUTPUT` la logica sequenziale. Lo scopo del primo è quello di guidare la computazione e di interfacciarsi con la RAM, quello del secondo è di aggiornare i segnali in modo sincrono al clock sul fronte di salita.

Descrizione ad alto livello del funzionamento

L'inizializzazione avviene nello stato di `IDLE`, la computazione è avviata mediante il segnale `i_start` = 1.

Una volta avviata la conversione:

- Legge da memoria i primi 2 byte per il calcolo della dimensione dell’immagine;
- Legge tutti i valori dei pixel dell’immagine al fine di trovarne il massimo e minimo;
- Calcola `delta_value`, ovvero la differenza tra il valore massimo e minimo dei pixel dell’immagine;
- Calcola `shift_level`, cioè la quantità di bit di cui `(current_pixel - min_pixel_value)` dovrà essere traslato;
- Rilegge i valori dei pixel (da indirizzo `0x2` a `last_address`), per ciascuno di essi:
    - Calcola il `temp_pixel` come riportato sotto;
    - Calcola il `new_pixel_value`, cioè il valore del pixel scelto tra il minimo della coppia `(255, TEMP_PIXEL);`
    - Salva in memoria, all’indirizzo corretto, il nuovo pixel.

```
DELTA_VALUE = MAX_PIXEL_VALUE – MIN_PIXEL_VALUE
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)
```

Segnali usati

- **`current_state`**: segnala lo stato della FSM in cui ci troviamo attualmente.
- **`last_address`**: tiene memorizzato l'ultimo indirizzo dell'immagine attualmente in RAM.
- **`width`**: prima dimensione dell'immagine da equalizzare.
- **`height`**: seconda dimensione dell'immagine da equalizzare.
- **`read_size`**: flag per tener traccia della lettura di `width` (0) o `height` (1).
- **`min`**: valore del pixel più piccolo dell'immagine.
- **`max`**: valore del pixel più grande dell'immagine.
- **`delta_value`**: differenza tra max e min.
- **`log`**: risultato del calcolo di $\log_2$(`delta_value +1`).
- **`shift_level`**: numero di shift verso sinistra da fare in seguito.
- **`temp_pixel`**: nuovo pixel temporaneo, da confrontare con il valore 255.
- **`counter`**: contatore da 16 bit.

- Ogni segnale ha un equivalente "`next`", utilizzato per aggiornare il valore alla salita del clock, implementando così dei registri per il mantenimento dei rispettivi segnali.

Logica Combinatoria

La Macchina a Stati Finiti è una macchina di Moore composta da undici stati: `IDLE, DIM_READING, LAST_ADDR_COMPUTE, MIN_MAX_FETCH, DELTA_VALUE_COMPUTE, SHIFT_COMPUTE, NEW_PIXEL_COMPUTE, MEM_WRITE_PIXEL, MEM_READ, END_COMPUTE, WAIT_CONFIRM`.

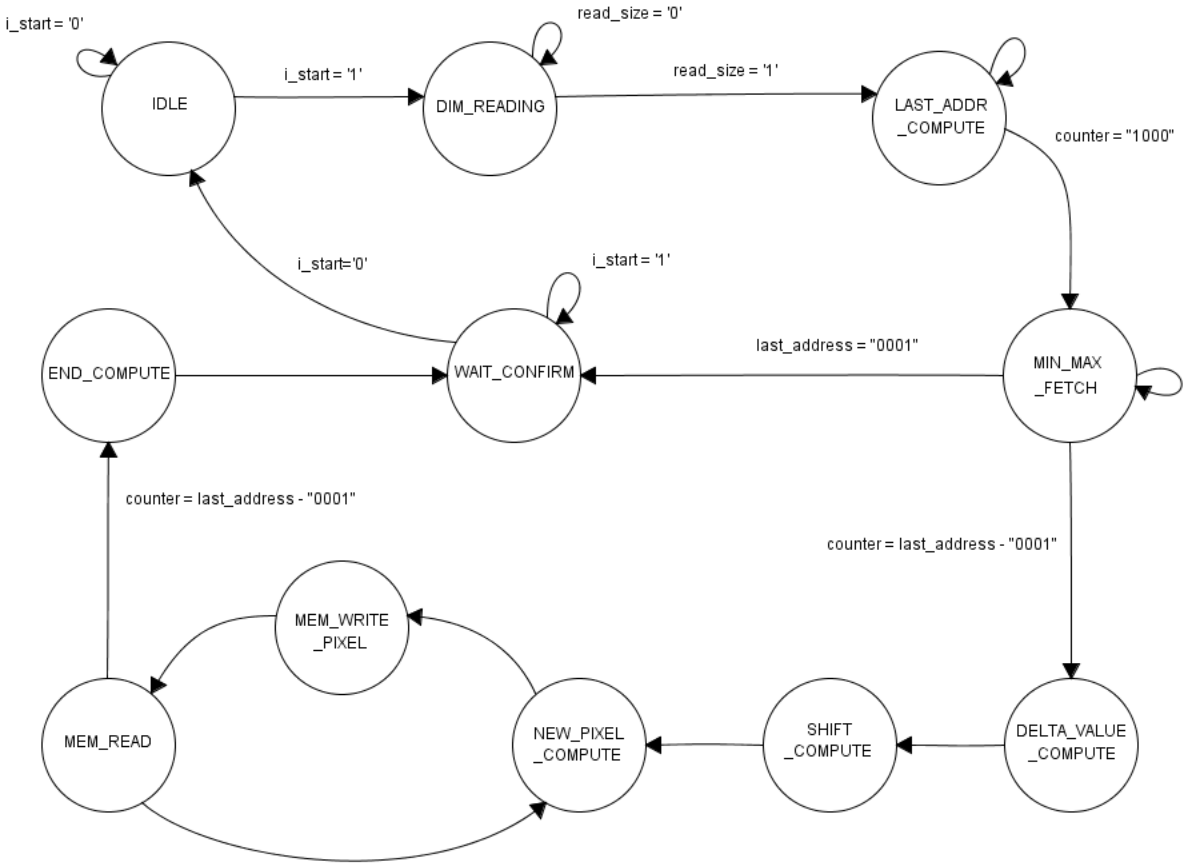

Fig. 2 - Diagramma della macchina a stati finiti (FSM)

Descrizione dei singoli stati

1. **`IDLE`:** stato di attesa e di reset per i segnali interni.
   Funzionamento: si attende che `i_start= '1'`, finché esso non arriva lo stato rimane in attesa.

2. **`DIM_READING`**: stato di lettura delle dimensioni dell'immagine.
   Funzionamento: facendo uso del bit `read_size`, che viene inizializzato a 0, si entra nella prima condizione pensata per la lettura di `width` dal primo indirizzo di memoria disponibile. Aggiornando `read_size` a 1 e ripetendo lo stato si passa alla lettura di `height` proveniente dal secondo indirizzo di memoria.

3. **`LAST_ADDR_COMPUTE`**: stato di calcolo dell'indirizzo a cui si trova l'ultimo pixel dell'immagine in ingresso: `last_address = width * height + 1`.
   Funzionamento: `last_address` e `counter` sono inizializzati a 0.
   La moltiplicazione tra `width` e `height` viene svolta mediante shift successivi a sinistra del valore di `width` conformemente al peso che assumono i bit `'1'` all'interno del vettore `height`. Ad ogni iterazione *t*, il contatore viene incrementato di uno e il segnale `height` viene consumato di un bit (shift destro), in modo da poter leggere il contributo degli `'1'` sempre in posizione 0 (LSB). Pertanto, `counter` tiene traccia del peso associato al bit in posizione 0 all'iterazione *t*, permettendo di effettuare uno shift a sinistra su `width` che corrisponde ad una moltiplicazione parziale. Così facendo `last_address` viene costruito iterativamente per moltiplicazioni successive. All'ottava iterazione, `last_address` è pronto e viene incrementato di 1 per essere consistente con l'indirizzamento a memoria dell'immagine in quanto il primo pixel si trova all'indirizzo `0x2`.

4. **`MIN_MAX_FETCH`**: stato di ricerca dei valori di pixel minimo e massimo dell'immagine e calcolo della loro differenza.
   Funzionamento: si riutilizza `counter` che tiene traccia dell'indirizzo in RAM al quale leggere il valore da confrontare con i min e max trovati alle iterazioni precedenti. La condizione di fuoriuscita canonica dallo stato è `counter = last_address - 1` (ovvero sono stati letti tutti i valori), tuttavia sono stati implementati ulteriori controlli per ridurre al minimo il tempo passato sullo stato:
   - Se la condizione (`max = 255 and min = 0`) è verificata allora è inutile proseguire nella ricerca trattandosi degli estremi massimi;
   - Se `last_address = '0001'` allora l'immagine in RAM è degenere, ovvero ha dimensione `0x0/0xn/nx0` e non ha senso continuare la computazione.

5. **DELTA_VALUE_COMPUTE**: stato di calcolo del logaritmo in base 2 di `delta_value`.
   Funzionamento: si confronta il valore di `delta_value` (ovvero `delta_value + 1`) con ogni potenza del 2 esprimibile su 9 bit per trovare il floor del logaritmo equivalente. Indicativamente, in ordine: a `delta_value` ≥ 256 corrisponde log $= 8$, a `delta_value` ≥ 128 corrisponde `log` $= 7$ e così via.

6. **SHIFT_COMPUTE**: stato di calcolo dello `shift_level`.
   Funzionamento: si finalizza il calcolo di `shift_level`, eseguendo la differenza tra 8 e `log`.

7. **NEW_PIXEL_COMPUTE**: stato di calcolo di `temp_pixel`.
   Funzionamento: si calcola la differenza tra il valore del pixel attualmente letto e `min`. Si effettua uno shift a sinistra di una quantità pari a `shift_level`, questa operazione avrà come risultato `temp_pixel`.

8. **MEM_WRITE_PIXEL**: stato di scrittura del nuovo pixel.
   Funzionamento: si confronta `temp_pixel` con 255, nel caso sia più piccolo allora il nuovo valore del pixel da scrivere sarà `temp_pixel`, altrimenti 255.

9. **MEM_READ**: stato intermedio di lettura.
   Funzionamento: si richiede alla RAM di preparare su `i_data` il valore del pixel successivo a quello appena gestito.

10. **END_COMPUTE**: stato di fine.
    Funzionamento: si imposta `o_done` a 1 dando segnale di terminazione processo per l'immagine corrente.

11. **WAIT_CONFIRM**: stato di attesa del segnale di start.
    Funzionamento: si attende che venga portato `i_start` a 0. Una volta letto, `o_done` viene riportato a 0 per permettere il rinnalzamento di `i_start` a 1. In quest'ultimo caso si torna ad `IDLE` per la prossima immagine.

Ottimizzazioni
e Workflow eccezionali

Durante la fase di progettazione della macchina a stati finiti sono state realizzate alcune ottimizzazioni per migliorare la gestione dei segnali, riducendone il numero, e migliorare le prestazioni temporali.

- È stato scelto di sfruttare un unico segnale contatore comune a più stati della macchina, poiché l'utilizzo di counter specifici per le operazioni di ricerca di `min/max`, di moltiplicazione per calcolo di `last_address` e di scorrimento della RAM per lettura/scrittura dei pixel risultava essere ridondante, in quanto tali contatori sarebbero stati sfruttati in mutua esclusione.

- È stato implementato un controllo che consente la terminazione immediata del processo (direttamente allo stato di `WAIT_CONFIRM`) subito dopo il calcolo di `last_address` nel caso in cui l'immagine abbia dimensioni nulle.

- Nella ricerca di `min` e `max`, se vengono trovati il minimo e il massimo coincidenti con gli estremi del dominio (ovvero 0 e 255 rispettivamente), allora si termina il ciclo di ricerca indipendentemente dal suo progresso e si passa direttamente allo stato successivo.

- È stato scelto di non implementare la classica operazione di moltiplicazione per il calcolo `last_address = width * height + 1`, perchè in un contesto reale verrebbero richieste eccessive risorse. Allo stesso modo però, un approccio mediante somme successive avrebbe inficiato troppo le prestazioni temporali in quanto nel caso limite di un'immagine 128x128, sarebbero necessari 128 cicli di clock per finalizzare il calcolo.
  Si è dunque optato per un algoritmo di moltiplicazione mediante shift successivi (vedi stato `LAST_ADDRESS_COMPUTE` nella sezione FSM), così si può calcolare il valore richiesto in otto cicli di clock.

Risultati Sperimentali

Versione Vivado: **`2020.1`**
FPGA target: **`xc7a200tfbg484-1`**

Test	Time	Descrizione
default	917500 ps	Singola immagine da 2x2 pixel
128x128	744122600 ps	Singola immagine da 128x128 pixel (caso limite: grandezza massima)
nullDim	632600 ps	Singola immagine da 2x0 pixel (caso degenere)
pixelTutti255	1157600 ps	Singola immagine da 8x8 con tutti i pixel aventi valore 255 (caso limite: estremo del dominio)
pixelTutti1	1157600 ps	Singola immagine da 8x8 con tutti i pixel aventi valore 1
pixelTutti0	917600 ps	Singola immagine da 2x2 con tutti i pixel aventi valore 0 (caso limite: estremo del dominio)
deltaMax	887600 ps	Singola immagine da 2x2 con valore `min` $= 0$ e `max` $= 255$ (caso limite: `shift_level` nullo)
multiStart	2712600 ps	Tre immagini consecutive da 35x35, 36x36, 12x12 pixel rispettivamente
rstAsincrono	1547600 ps	Singola immagine 4x4, in questo particolare testbench viene dato un secondo reset mentre la macchina sta terminando di scrivere i nuovi pixel
10k	6177799100 ps	Diecimila immagini

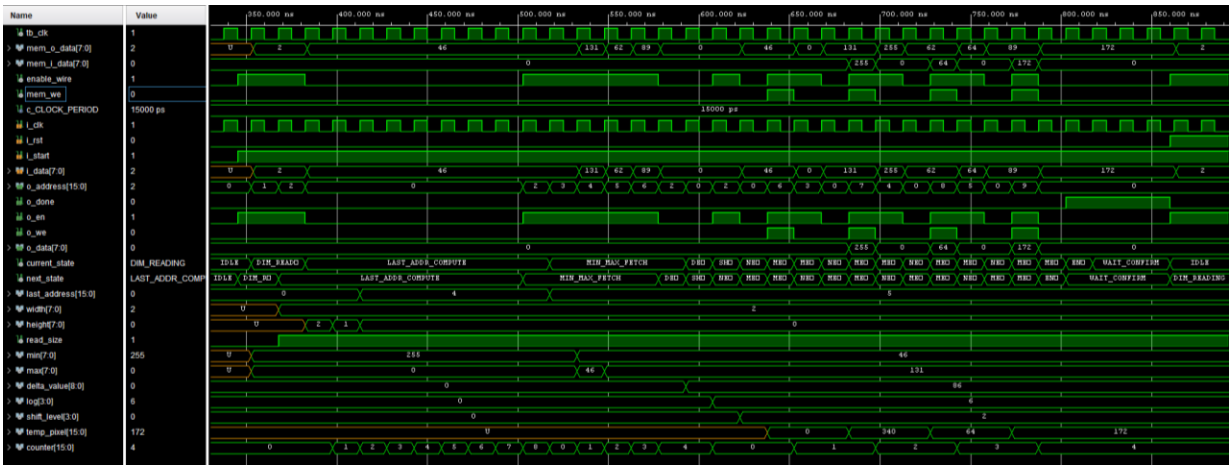

Fig. 3 - Forme d'onda relative al testbench default

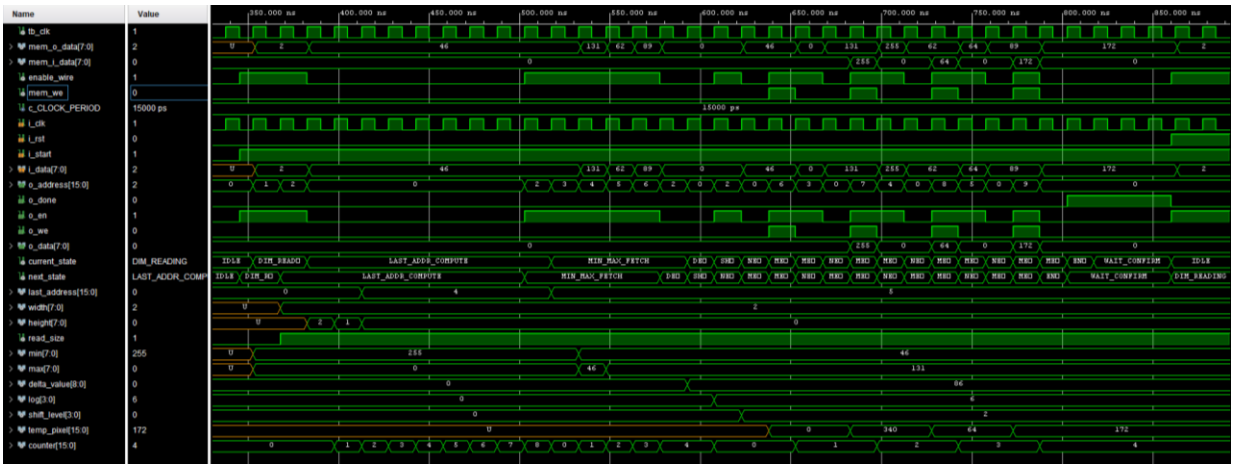

Fig. 4 - Forme d'onda relative al testbench rstAsincrono

Timing Report

Slack (MET): 93.569 ns
Data Path Delay: 5.818 ns
Logic: 2.691 ns (46.253%)
Route: 3.127 ns (53.747%)

Utilization Report

Site Type	Used	Fixed	Available	Util%
Slice LUTs	218	0	134600	0.16
LUT as Logic	218	0	134600	0.16
LUT as Memory	0	0	46200	0.00
Slice Registers	108	0	269200	0.04
Register as Flip Flop	108	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

Conclusioni

L'architettura progettata è pensata per affrontare il problema dell'equalizzazione delle immagini cercando di minimizzare il numero di cicli di clock e di stati, riutilizzare segnali ove possibile senza inficiare la chiarezza del codice VHDL. La stabilità dell'architettura è confermata dai risultati sperimentali di test che sono stati pensati per stressare la macchina su casi limite o grandi quantità di immagini da processare.

Si è inoltre sperimentalmente verificato che, quando sintetizzata sulla FPGA di riferimento, l'architettura proposta è in grado di operare a frequenze di clock maggiori permettendo quindi ulteriori guadagni nel tempo di esecuzione. Infatti, modificando il clock del testbench di riferimento ufficiale di un ordine di grandezza inferiore ai 100 ns, l'esecuzione in post sintesi viene portata a termine con esito positivo.