PROVA FINALE
Progetto di Reti Logiche

Anno Accademico 2020/2021
Professore: Salice Fabio

Gruppo formato da:
ALESSANDRO MARANELLI (Codice persona: 10661029 -Matricola: 911967)
ELISA MARIANI (Codice persona:10632876 - Matricola:907287)

Politecnico di Milano

INDICE

1.Introduzione

1.1. Scopo del progetto

Lo scopo del progetto è implementare un componente che riproduca il metodo di equalizzazione dell'istogramma di una immagine, metodo che permette di ricalibrare e incrementare il contrasto cromatico della stessa, effettuando una distribuzione dei valori di intensità dei pixel su tutto l'intervallo di intensità.
Data la complessità dell'algoritmo standard, il componente sintetizzato deve riprodurre una versione semplificata dell'algoritmo, andando a trattare solamente immagini in scala di grigi a 256 livelli.

1.2. Specifiche generali

Le immagini, la cui dimensione massima è di 128x128 pixel, sono in scala di grigi a 256 livelli, pertanto ogni pixel assume un valore di intensità di grigio compreso tra 0 e 255 (estremi inclusi), esprimibile in binario con 1 byte.
Le dimensioni dell'immagine sono salvate in una memoria RAM che contiene anche i valori di intensità di ciascun pixel dell'immagine (paragrafo 1.3).
L'algoritmo da implementare prevede che si trovino il massimo e il minimo valore di intensità dei pixel all'interno dell'immagine di partenza, rispettivamente MAX_PIXEL_VALUE e MIN_PIXEL_VALUE, e che si eseguano al termine della ricerca le seguenti operazioni:

DELTA_VALUE= MAX_PIXEL_VALUE-MIN_PIXEL_VALUE
SHIFT_LEVEL=(8-FLOOR(LOG2(DELTA_VALUE+1)))

Per ogni pixel bisognerà poi calcolare il corrispondente nuovo pixel dell'immagine equalizzata seguendo questi passi:

TEMP_NEW_PIXEL=(CURRENT_PIXEL_VALUE-MIN_PIXEL_VALUE)<<SHIFT_LEVEL
NEW_PIXEL_VALUE=MIN(255,TEMP_NEW_PIXEL)

I nuovi valori di intensità dei pixel vengono salvati nella stessa memoria RAM da cui sono stati prelevati quelli dell'immagine originale, in coda a questi ultimi.

Il componente deve comunicare con l'esterno per mezzo di opportuni segnali e va inoltre progettato col fine di poter codificare più immagini. La prima immagine deve essere codificata sempre dopo un reset del modulo (il segnale *i_rst* va a '1'). Le successive codifiche potranno avvenire soltanto dopo il termine della codifica precedente. Il meccanismo di re-start delle codifiche è gestito tramite i segnali *i_start* e *o_done*: il primo è il segnale in input che determina l'inizio della codifica, il secondo è il segnale in output che comunica il termine dell'elaborazione.

Due ulteriori segnali, *o_en* (accesso alla memoria) e *o_we* (attivazione della modalità di scrittura in memoria), permettono l'interazione con la RAM. Lo scambio di dati avviene attraverso tre linee di segnali vettore: *i_data* arriva al componente in seguito ad una richiesta di lettura dalla memoria, *o_data* serve per inviare alla memoria i valori elaborati e o_address è il bus indirizzi in uscita che comunica un indirizzo specifico alla RAM.
Infine, *i_clk* è un segnale di ingresso generato dal Test Bench.

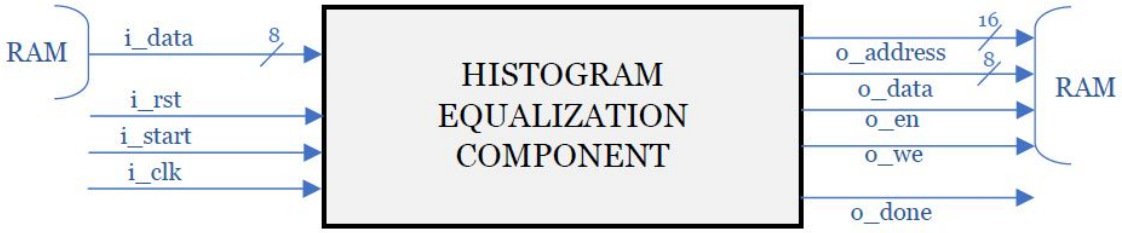

fig.1 : Rappresentazione dell' interfaccia del componente e dell'interazione con la memoria RAM

1.3. Descrizione della memoria

La memoria RAM (fig.2) con cui comunica il modulo ha un indice di indirizzamento al byte:

- Agli indirizzi 0 e 1 sono salvati rispettivamente la dimensione di colonna e di riga dell'immagine;
- Dall'indirizzo 2 fino all'indirizzo 2+(NumColonne*NumRighe) si trovano salvati i valori di intensità dei pixel dell'immagine iniziale.
- Gli indirizzi da 2+(NumColonne*NumRighe)+1 conterranno i valori di intensità dei pixel dopo l'equalizzazione (perciò l'ultimo valore sarà salvato in 2+2*(NumColonne*NumRighe))

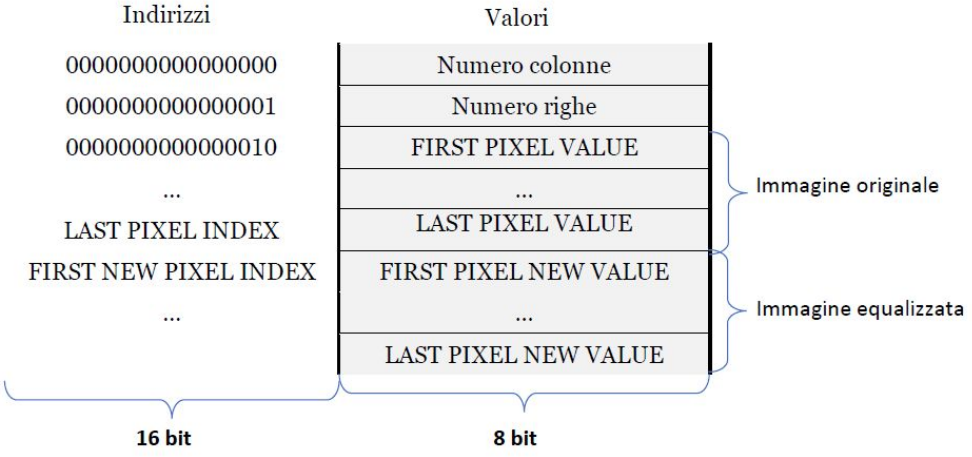

fig.2 : rappresentazione della memoria RAM comunicante con il modulo

Un semplice esempio del funzionamento del modulo può essere l'elaborazione di un'immagine 2x2. In fig.3, la RAM pre-elaborazione rappresenta lo stato iniziale della memoria. Inizialmente, infatti, essa contiene soltanto le dimensioni dell'immagine da equalizzare e i valori di intensità dei pixel della stessa. Successivamente, l'equalizzazione avviene applicando l'algoritmo descritto nel paragrafo 1.2 e il risultato della computazione è mostrato nella RAM post-elaborazione, dove i byte da 6 a 9 costituiscono l'immagine equalizzata.

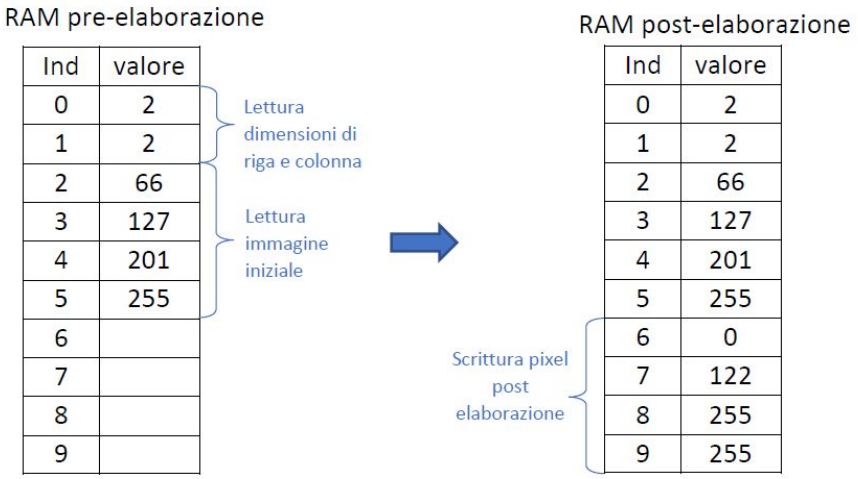

fig.3 : Esempio di memoria pre e post elaborazione

2. Architettura

2.1 Scelte progettuali

L'architettura è stata pensata e progettata come un unico modulo, il quale si occupa delle interazioni con la memoria RAM e con l'esterno e dell'elaborazione dell'algoritmo di computazione.

L'implementazione realizza una macchina a stati finiti (FSM) e utilizza un solo processo. All'interno del processo, la determinazione dello stato corrente avviene nel seguente modo: mediante un costrutto 'if ' viene controllato, per ogni ciclo di clock, lo stato del segnale *i_rst* che, nel caso sia alto, riporta la macchina nello stato iniziale e, nel caso contrario, tramite un costrutto case, porta la macchina allo stato corrente.

In concorrenza al processo vengono utilizzate delle istruzioni combinatorie per gestire i segnali *o_we* e *o_en*. Essi sono segnali il cui valore dipende esclusivamente dallo stato della macchina e quindi possono essere gestiti in maniera concorrente al processo e non necessariamente in modo sequenziale all'interno di esso.

La FSM presenta alcuni stati di "wait" (WAIT_DIM, WAIT_OLD_PIX). Essi sono stati creati per evitare che possibili ritardi del segnale in lettura *i_data*, proveniente dalla memoria RAM, causassero errori di sincronizzazione. Mediante questi stati si può garantire il funzionamento corretto della macchina sino ad un ritardo massimo di 1 clock.

2.2 Segnali interni

L'architettura implementata utilizza i seguenti segnali interni:

- **current_state: state_type**
  Indica lo stato corrente della FSM. Il suo valore viene aggiornato ad ogni clock.
- **max_pixel_value: std_logic_vector (7 downto 0)**
  Utilizzato per memorizzare il valore massimo tra quelli dei pixel dell'immagine da equalizzare.
- **min_pixel_value: std_logic_vector (7 downto 0)**
  Utilizzato per memorizzare il valore minimo tra quelli dei pixel dell'immagine da equalizzare.
- **n_pixel: std_logic_vector (14 downto 0)**
  Utilizzato per memorizzare il numero totale di pixel dell'immagine.
- **n_col: std_logic_vector (7 downto 0)**
  Memorizza la dimensione di colonna dell'immagine.
- **counter: std_logic_vector (14 downto 0)**
  Utilizzato come contatore per tener conto dei pixel letti.
- **ram_index: std_logic_vector (15 downto 0)**
  Utilizzato per individuare il prossimo indirizzo di RAM in cui leggere o scrivere.
- **delta_value: std_logic_vector (7 downto 0)**
  Memorizza il valore DELTA_VALUE definito nel paragrafo 1.2.
- **shift_level: std_logic_vector (3 downto 0)**
  Memorizza il valore SHIFT_LEVEL definito nel paragrafo 1.2.
- **difference: std_logic_vector (7 downto 0)**
  Memorizza la differenza tra il pixel corrente e *min_pixel_value*, operazione necessaria nel calcolo di NEW_PIXEL_VALUE.

2.3 Macchina a stati

La FSM progettata, il cui diagramma è rappresentato in figura 4, codifica ogni immagine seguendo i seguenti step:

1. Lettura numero colonne e righe e conseguente calcolo del numero totale di pixel;
2. Lettura sequenziale di tutti i pixel dell'immagine originale e contemporanea identificazione del pixel massimo e minimo;
3. Calcolo *delta_value*
4. Calcolo *shift_level*
5. Per ogni pixel dell'immagine originale:
   i. Rilettura del pixel da memoria
   ii. Calcolo *difference (*CURRENT_PIXEL_VALUE-MIN_PIXEL_VALUE)
   iii. Determinazione nuovo pixel
   iv. Scrittura in memoria del nuovo pixel
6. Segnalazione del termine dell'esecuzione

Per la determinazione dello *shift_level*, è stato scelto di effettuare un controllo sulla posizione dell' '1' più significativo nel parametro '*delta_value+1*'.

L'applicazione dell'operazione di shift (<<) sul valore *difference* è stata invece implementata nel seguente modo: agli '(8-shift)' bit meno significativi del segnale viene concatenato in coda un numero 'shift' di zeri.

Nella progettazione si è cercato di parallelizzare le operazioni il più possibile, soprattutto per quanto riguarda il caricamento di dati dalla memoria.
Tuttavia, nei pochi casi in cui la parallelizzazione delle operazioni comportava l'utilizzo di strutture inutilmente complesse e non convenienti in termini di efficienza temporale e di occupazione di memoria della macchina, si è optato per l'aggiunta di uno stato ulteriore nella FSM. Queste considerazioni saranno approfondite nel paragrafo 3.2.

Diagramma FSM

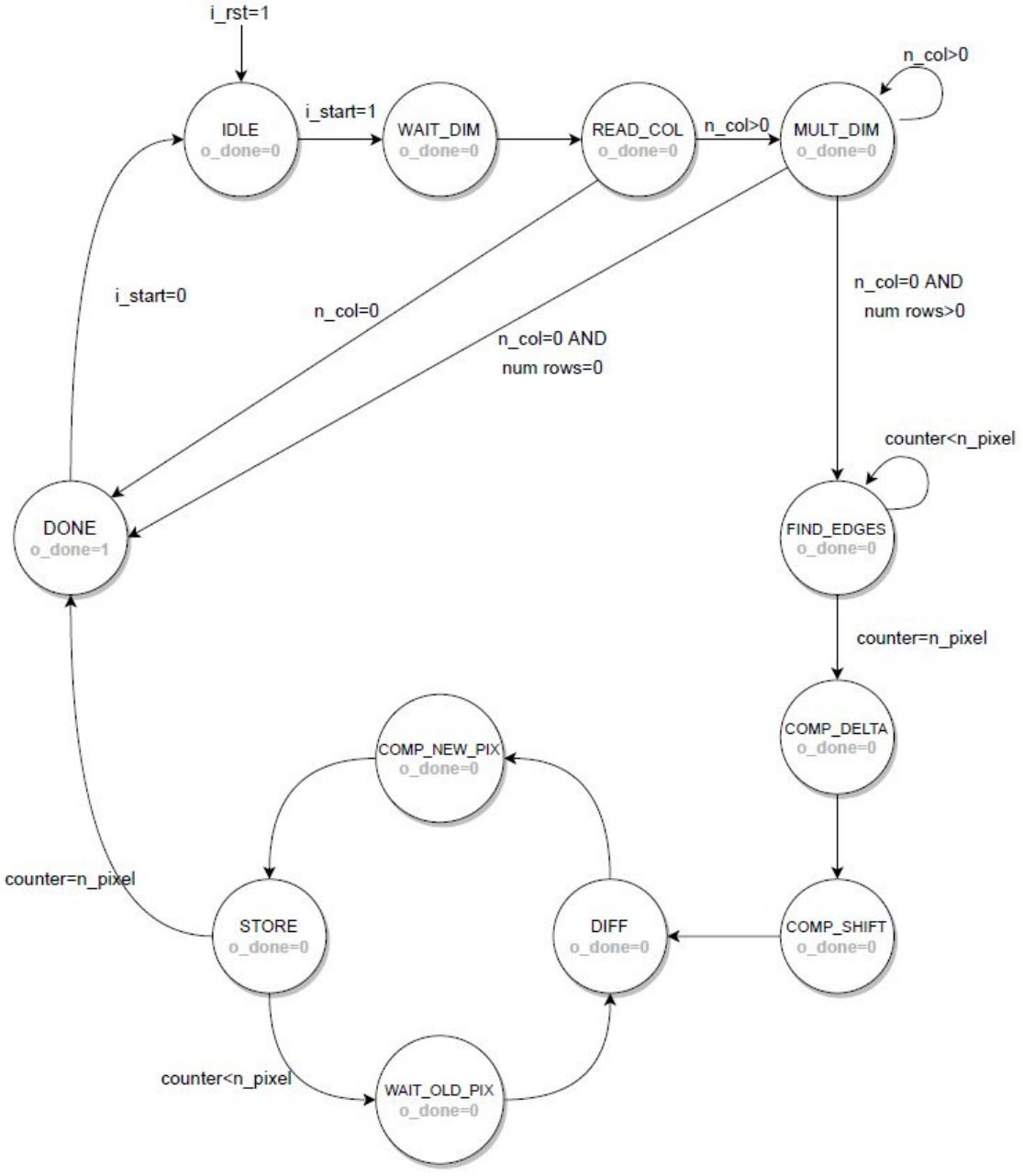

fig.4 : Diagramma degli stati della FSM

Descrizione degli stati

Di seguito una descrizione degli stati che compongono la FSM progettata:

- IDLE

  E' lo stato di partenza della FSM. In IDLE si inizializzano i valori di alcuni segnali interni (tra cui *o_address* e *ram_index*, posti uguali all'indirizzo '0' della ram, e *o_done*, posto a '0') utilizzati negli stati successivi. In questo stato la macchina rimane in attesa che *i_start* diventi '1': quando questo accade la computazione inizia e lo stato corrente viene posto uguale a WAIT_DIM.

- WAIT_DIM

  Nello stato WAIT_DIM si attende che nella linea dati *i_data* si carichi la dimensione di colonna dell'immagine. Contemporaneamente si precarica l'indirizzo di memoria successivo, corrispondente al valore di dimensione di riga, in *o_address*. Lo stato corrente viene aggiornato a READ_COL.

- READ_COL

  In READ_COL si legge la dimensione di colonna dell'immagine, che viene memorizzata nel registro interno *n_col*. Se essa è positiva, lo stato corrente viene aggiornato a MULT_DIM. Altrimenti la computazione può già terminare, lo stato viene aggiornato a DONE ed il segnale *o_done* posto a '1'.

- MULT_DIM

  In MULT_DIM viene calcolato il numero totale di pixel che compongono l'immagine (prodotto righe*colonne).
  Ad ogni ciclo di clock il registro *n_col* viene decrementato di '1' e contemporaneamente viene sommata in *n_pixel* (registro precedentemente inizializzato a '0') la dimensione di riga, contenuta nel segnale *i_data*. Le operazioni vengono ripetute finché *n_col* non vale '0': questo momento coincide con l'aggiornamento dello stato corrente, che viene posto a FIND_EDGES.
  Inoltre, nei cicli di clock in cui *n_col* vale '1' e '0' (ultimi due cicli di clock che vedono MULT DIM come stato corrente), si iniziano a precaricare in *o_address* gli indirizzi rispettivamente del primo e del secondo pixel dell'immagine. In tal modo al cambio di stato saranno immediatamente pronti come valori in input.

- FIND_EDGES

  In FIND_EDGES si esaminano tutti i pixel dell'immagine originale, uno ad ogni ciclo di clock, partendo dal primo. L'obiettivo è individuare il pixel con il valore massimo e quello con il valore minimo di tutta l'immagine. Il pixel corrente viene valutato in questo modo: se il suo valore è inferiore al minimo corrente (*min_pixel_value*), il minimo corrente viene aggiornato, se invece è maggiore del massimo corrente (*max_pixel_value*) è il massimo corrente ad aggiornarsi.
  Nel caso in cui tutti i pixel siano stati esaminati (*counter* = *n_pixel*), lo stato si aggiorna a COMP_DELTA e viene caricato nuovamente in *o_address* e in *ram_index* l'indirizzo corrispondente al primo pixel dell'immagine. Ora min_pixel_value e *max_pixel_value* contengono rispettivamente il valore minimo e il valore massimo dell'immagine in input.

Se rimangono pixel da analizzare viene incrementato il contatore e si precarica l'indirizzo di RAM contenente il valore che verrà letto nel secondo ciclo di clock successivo.

- COMP_DELTA
  COMP_DELTA calcola il risultato di (*max_pixel_value* - *min_pixel_value* + 1) e lo salva in *delta_value*. Lo stato corrente viene aggiornato a COMP_SHIFT.

- COMP_SHIFT
  In COMP_SHIFT viene determinato il valore dello *shift_level*, utilizzando il valore di *delta_value* calcolato nello stato precedente (secondo la definizione di shift level presentata nel paragrafo 1.2). Lo stato viene portato a DIFF.

- DIFF
  In DIFF viene calcolata la differenza *difference* tra il pixel corrente (contenuto nel bus *i_data* perché precaricato negli stati precedenti) e il minimo valore tra tutti i pixel (*min_pixel_value* individuato in FIND_EDGES). Lo stato viene aggiornato a COMP_NEW_PIX.

- COMP_NEW_PIX
  In COMP_NEW_PIX viene calcolato, per il pixel corrente, il nuovo valore corrispondente al medesimo pixel nell'immagine equalizzata. In relazione alla definizione di NEW_PIXEL_VALUE (paragrafo 1.2), a seconda dello *shift_level* calcolato in precedenza si precarica in *o_data* il valore equalizzato del pixel corrente.
  Inoltre, viene incrementato un contatore necessario per verificare che tutti i pixel vengano esaminati. Contemporaneamente, si precarica in *o_address* l'indirizzo RAM dove, negli stati successivi, verrà salvato il valore del pixel equalizzato calcolato nel ciclo di clock corrente.
  Lo stato viene aggiornato a STORE per la scrittura effettiva del pixel in memoria.

- STORE
  In STORE avviene la scrittura effettiva del pixel in memoria, possibile grazie al segnale *o_we* che in corrispondenza di questo stato commuta a '1'. Se *counter* è minore di *n_pixel*, non tutti i pixel sono stati ancora equalizzati: si precarica in *o_address* l'indirizzo del successivo pixel da equalizzare e lo stato viene aggiornato a WAIT_OLD_PIX. Se invece tutti i pixel sono stati elaborati, lo stato si aggiorna a DONE ed il segnale *o_done* viene posto a '1'.

- WAIT_OLD_PIX
  WAIT_OLD_PIX è uno stato necessario al caricamento sul bus dati *i_data* del prossimo pixel da elaborare, il cui indirizzo è stato precaricato nello stato precedente. Lo stato corrente al termine del clock torna a DIFF per iniziare l'elaborazione di un nuovo pixel.

- DONE
  DONE è lo stato in cui termina la codifica. Il termine della codifica è comunicato all'esterno tramite il segnale *o_done* che commuta a '1' soltanto quando questo è lo stato corrente. Nello stato DONE, inoltre, si attende che il segnale che indica l'inizio di una nuova codifica (*i_start*) venga portato a '1'. In questo caso lo stato corrente si aggiornerà a IDLE e il segnale *o_done* verrà riportato a '0'.

3. Risultati sperimentali

3.1 Risultati della sintesi

Implementazione e sintesi sono state effettuate correttamente da Vivado, e le simulazioni dei test benches in post-sintesi, sia Functional che Timing, sono andate a buon fine.

I dati sperimentali hanno evidenziato che il periodo di clock necessario per il corretto funzionamento del componente sintetizzato è di molto inferiore al limite di 100 ns dato dalla specifica.

Il report di sintesi sottostante mostra l'utilizzo delle risorse sulla FPGA assegnata (xc7a200tfbg484-1). Si può notare come il componente necessiti di 166 LUT, che sono lo 0.12% di quelle disponibili, per implementare l'algoritmo di equalizzazione delle immagini, e di 119 registri (0.04% di quelli disponibili) i quali sono utilizzati come Flip-Flop.

Site Type	Used	Fixed	Available	Util%
Slice LUTs	166	0	134600	0.12
LUT as Logic	166	0	134600	0.12
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

fig. 5 report di sintesi riguardante l'utilizzo delle risorse

3.2 Ottimizzazioni

La versione finale del componente è il risultato di varie scelte progettuali che hanno portato ad un miglioramento della macchina in termini di tempi di esecuzione e di area di FPGA occupata.

Dalla prima versione funzionante del codice, tramite le ottimizzazioni applicate, si è riusciti a ridurre il numero di LUT complessive utilizzate da 265 a 166.
Anche il WNS, parametro che indica il margine minimo tra il tempo di setup del percorso critico ed il periodo di clock, impostato a 100 ns come da specifica, è aumentato da 92,217 ns a 94,897 ns.
Le principali ottimizzazioni effettuate sul codice sono le seguenti:

- Versioni precedenti del codice VHDL non prevedevano lo stato COMP_DELTA: la differenza *max_pixel_value-min_pixel_value* veniva calcolata all'interno di FIND_EDGES ogni qualvolta venivano aggiornati massimo o minimo corrente. L'aggiunta di un'ulteriore stato ha permesso di calcolare la differenza al termine della ricerca dei due valori estremi; questa modifica ha consentito di ridurre il tempo di computazione.

- La scelta di implementare il prodotto righe*colonne con una serie di somme successive ha reso possibile, data anche la dimensione massima delle immagini trattate (128*128 pixel), il fatto di poter riservare soltanto 15 bit ai registri *n_pixel* e *counter* (al posto dei 16 bit che un'operazione 8 bit*8 bit normalmente richiederebbe).
  La rimozione dell'operazione di moltiplicazione dal codice ha permesso di ridurre sensibilmente il numero di LUT impiegate. Sebbene questa scelta implichi un utilizzo di più cicli di clock per svolgere la medesima operazione, la dimensione massima delle colonne è fissata a 128, perciò in simulazione nel caso peggiore verranno impiegati "solamente" 127 clock in più rispetto all'uso dell'operatore "*". Questo tempo si può definire trascurabile rispetto a quello totale, necessario all'applicazione dell'algoritmo di equalizzazione a tutti i pixel componenti l'immagine.

- Nel corso di una prima analisi era stato scelto di riservare alle immagini di dimensione '1'x'1' un trattamento 'ad hoc': uno stato apposito gestiva questa casistica e in questo modo venivano evitate tutte le operazioni per il calcolo di *delta* e *shift* necessarie per il calcolo del nuovo pixel in condizioni 'normali'. Per il trattamento del singolo caso la soluzione era certamente più efficiente ma comportava uno spreco di risorse sicuramente non necessario: il calcolo di *delta* e *shift (*in particolar modo in questo caso particolare) sono operazioni effettuabili in un tempo finito e molto limitato (complessità temporale O(1)). Inoltre, il caso di un'immagine '1'x'1' equivale ad avere un'immagine di un solo pixel che, in un'applicazione realistica del componente, è sicuramente un caso insolito. Alla luce di queste considerazioni si è scelto quindi di gestire l'analisi di queste particolari immagini allo stesso modo di tutte le altre.

4. Simulazioni

Per verificare il corretto funzionamento del componente in tutte le possibili casistiche che si possono presentare sono stati pensati e costruiti appositi test bench. È stata posta particolare attenzione ad opportuni casi limite.

4.1 Valori limite di Delta_value

Un possibile caso critico è quello in cui il *delta_value* vale '255', caso che si verifica quando l'intensità massima dei pixel è 255 mentre quella minima è 0. In questo caso lo shift corrispondente è '0': ogni pixel iniziale dovrà rimanere invariato nell'immagine finale.

Un altro caso limite è quello in cui tutti i pixel dell'immagine hanno la stessa intensità: *max_pixel_value* e *min_pixel_value* hanno lo stesso valore, perciò il *delta_value* è '0' e lo *shift_value* è '8'. L'immagine equalizzata dovrà avere tutti i pixel a '0'.

4.2 Altri valori particolari di Delta_value

Sono stati effettuati dei test in cui il segnale *delta_value* assume valori di soglia in relazione al calcolo dello *shift_level*.
Come si può notare in tab.1, ad ogni valore di *shift_level* corrisponde un intervallo di più *delta_value* diversi.
I casi testati hanno fatto in modo che *delta_value* fosse pari ai valori estremi di tali intervalli e si è potuto verificare che il valore di *shift_level* fosse assegnato correttamente.

SHIFT_LEVEL	intervallo DELTA_VALUE corrispondente
0	255
1	127-254
2	63-126
3	31-62
4	15-30
5	7-14
6	3-6
7	1-2
8	0

tab.1:La tabella mostra il valore di shift_level corrispondente ai diversi valori di delta_value

Sono stati testati anche valori intermedi di *delta_value* (valori compresi strettamente negli intervalli riportati in tabella) e anche per essi si è verificato che il valore di *shift_level* fosse quello corretto.

4.3 Immagini con dimensioni al limite del dominio di applicazione

È stata testata l'equalizzazione di immagini di dimensioni '1'x'1', dove l'unico pixel caratterizzante l'immagine deve essere portato a '0'.

E' stato verificato anche il funzionamento dell'architettura nel caso particolare di immagini in cui le due dimensioni, di riga e di colonna, erano pari a '0'. In questa particolare situazione la macchina commuta il segnale *o_done* a '1' subito dopo aver letto le dimensioni dell'immagine.

Si è scelto di gestire anche i casi in cui soltanto una delle due dimensioni è pari a zero, nonostante si ritenga siano casi che possono avere luogo soltanto come conseguenza di errori di salvataggio delle dimensioni nella memoria. Il comportamento della macchina risulta analogo al caso di immagine '0'x'0'.

Immagini di dimensioni 128x128 invece non hanno dato, come previsto, alcun problema: i 16 bit di indirizzo permettono di contenere con ampio margine un numero di celle che nel caso peggiore risulta di 32770 (2 celle per le dimensioni dell'immagine, 128*128 celle per l'immagine originale e 128*128 celle per l'immagine equalizzata).

4.4 Test composti da più immagini

Con più immagini da elaborare all'interno dello stesso test bench è stato testato il corretto sincronismo tra i segnali *done* e *start*: quest'ultimo deve rimanere alto sin quando la computazione dell'immagine in elaborazione non sarà terminata, ovvero finché il segnale *done* non viene portato a '1'.

È stata inoltre verificata la corretta inizializzazione dei registri interni al componente tra la computazione di una immagine e della successiva.

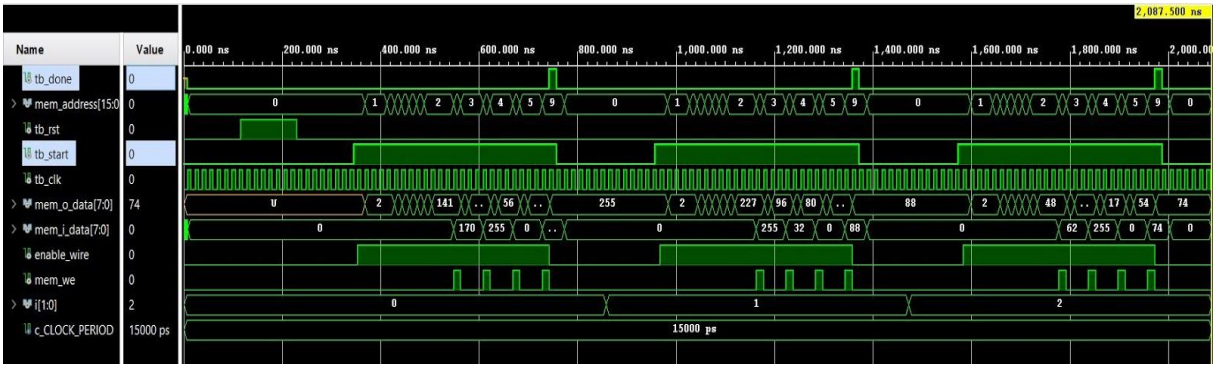

Fig.6 Forma d'onda di un test bench composto da tre immagini. Come si può notare, il segnale di start rimane alto finchè done non viene messo a 1, dopodichè l'indirizzo della memoria torna quello della prima cella e quando start torna ad 1 inizia la computazione dell'immagine successiva.

4.5 Reset asincrono

In caso di *reset* asincrono il modulo deve interrompere qualunque cosa stia facendo, perdere memoria del passato e mettersi in attesa di un nuovo *start*.

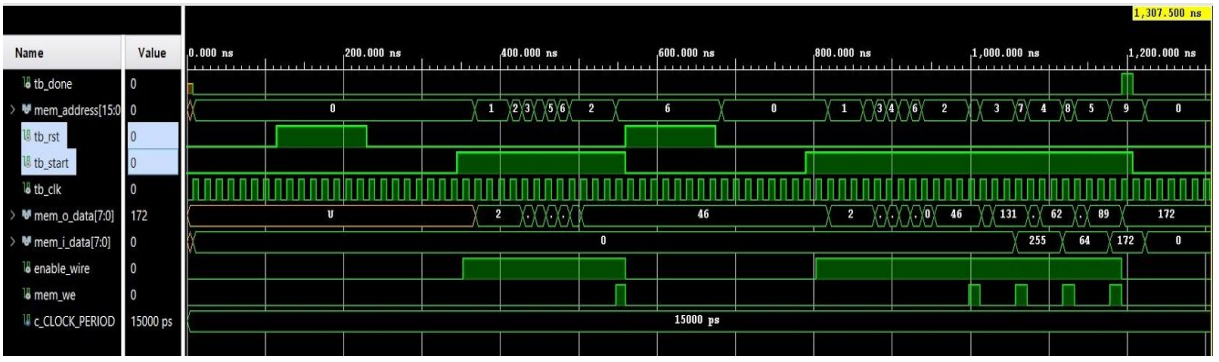

Fig.7 Da questa forma d'onda si può notare come l'arrivo del secondo segnale di reset faccia interrompere la computazione. Soltanto ad un nuovo segnale di start essa ricomincia da capo

Infine sono stati generati diversi test, ciascuno composto di più immagini di diverse dimensioni, per verificare il corretto funzionamento del componente in condizioni di utilizzo realistiche.

5. Conclusioni

I risultati del testing permettono di concludere che l'architettura progettata funzioni correttamente nelle simulazioni Behavioral, Post-Synthesis Functional e Post-Synthesis Timing, rispettando le specifiche di progetto.
Essa è stata estensivamente collaudata sia con test generati automaticamente e in modo casuale, che con test manualmente scritti, mirati a testare casistiche particolari.
I report di sintesi evidenziano una buona gestione delle risorse tempo e area della FPGA.
Alla luce degli esiti di tutte le prove effettuate, si ritengono vincenti le scelte progettuali e le ottimizzazioni effettuate.