Prova Finale (Progetto di Reti Logiche)

Prof. Gianluca Palermo - Anno 2020/2021

Christian Romeo (Codice Persona 10629231 - Matricola 908821)

Gabriele Perego (Codice Persona 10488414 - Matricola 853068)

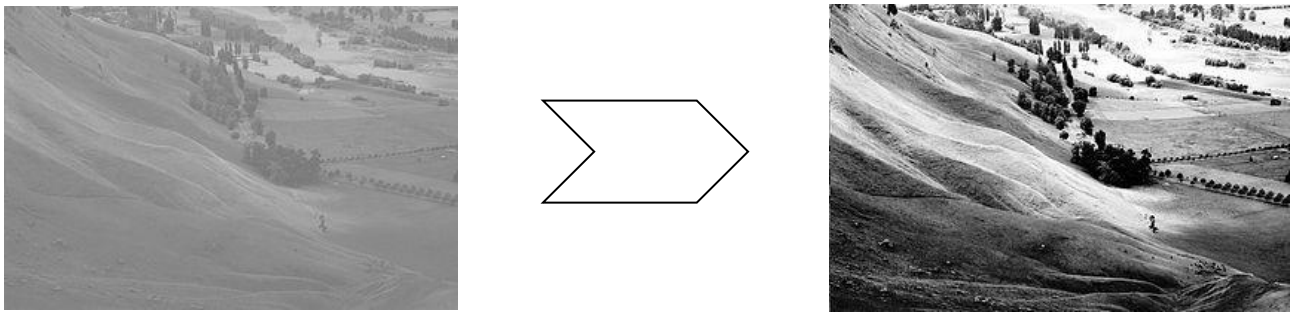


Figura 1 Equalizzazione Istogramma Immagine

ABSTRACT

Il metodo di equalizzazione dell'istogramma di un'immagine è un metodo di elaborazione digitale pensato per ricalibrare il contrasto di un'immagine quando l'intervallo dei valori di intensità sono molto vicini effettuandone una distribuzione su tutto l'intervallo, al fine di incrementare il contrasto.
Vedremo un'implementazione dell'algoritmo in una sua versione semplificata su di un componente hardware sintetizzato tramite il linguaggio VHDL.

INDICE

1. INTRODUZIONE

Il nostro obiettivo per questo progetto, oltre a creare un design funzionante in pre e post sintesi che rispetti le specifiche, è stato quello di creare un componente scalabile, con codice chiaro, semplice e di facile manutenzione.
Nella fase di implementazione del progetto sono stati anche affrontati aspetti dettati dalle limitazioni della specifica ed architettura utilizzata.

1.1 Specifiche di progetto

È richiesto di implementare l'algoritmo di equalizzazione applicato ad immagini digitali, in scala di grigi a 256 livelli, che deve trasformare ogni suo pixel nel modo seguente:

$$Delta\_Value = Max\_Pixel\_Value - Min\_Pixel\_Value$$
$$Shift\_Level = (8 - Floor(Log2(Delta\_Value + 1)))$$
$$Temp\_Pixel = (Current\_Pixel\_Value - Min\_Pixel\_Value) << Shift\_Level$$
$$New\_Pixel\_Value = Min(255, Temp\_Pixel)$$

Il modulo da implementare dovrà: leggere l'immagine sequenzialmente e riga per riga da una memoria già istanziata; equalizzarla; ed infine scrivere i risultati sulla stessa memoria negli opportuni indirizzi.
Ogni byte corrisponde ad un pixel dell'immagine.
La dimensione dell'immagine è definita da 2 byte, memorizzati a partire dall'indirizzo 0, il byte all'indirizzo 0 si riferisce alla dimensione di colonna (N_COL), mentre il byte all'indirizzo 1 si riferisce alla dimensione di riga (N_RIG). La dimensione massima dell'immagine è 128 x 128 pixel.
I pixel dell'immagine da elaborare, ciascuno di 8 bit, sono presenti in memoria con indirizzamento al byte partendo dalla posizione 2 e in byte contigui.
I pixel dell'immagine equalizzata, ciascuno di 8 bit, devono essere salvati in memoria con indirizzamento al byte partendo dalla posizione $2 + (N\_COL * N\_RIG)$.

1.2 Funzionamento in sintesi

Il funzionamento di base del componente può essere schematizzato in breve attraverso un numero definito di step, poi rappresentato architetturalmente attraverso una macchina a stati finiti:

1. Reset e attesa del segnale di start.
2. Inizializzazione dei segnali ed output del componente.
3. Lettura e salvataggio in un registro del numero di colonne e poi righe.
4. Lettura di tutti i pixel originali e ricerca del massimo e minimo valore.
   In base alla struttura della memoria mi interrompo all'indirizzo $2 + (N\_COL * N\_RIG)$.
5. Calcolo livello di shift da applicare ai pixel tramite casistiche a soglia per emulare semplicemente l'operazione di floor e logaritmo.
6. Ritorno a leggere il primo pixel dell'immagine ed applico l'equalizzazione, se risulta un valore superiore a 254 allora viene equalizzato a 255, in entrambi i casi viene salvato il valore equalizzato.

7. Scrivo sulla RAM il nuovo valore del pixel e controllo se ho terminato di esaminare tutti i pixel originali, in tal caso mi avvio allo stato di terminazione, altrimenti ritorno allo stato di lettura dei pixel passando al pixel successivo, ripetendo l'operazione per tutti gli altri pixel dell'immagine.
8. Per segnalare il termine delle operazioni di equalizzazione viene alzato il segnale o_done a '1'. Attendiamo ora il segnale di fine (i_start posto a '0') che ci permetterà di ritornare allo stato di inizializzazione per l'elaborazione di una nuova immagine.

1.3 Note aggiuntive sul codice

Ai fini di non complicare un codice in sostanza semplice è stato scelto di scriverlo in modo chiaro e lineare, quindi sono state utilizzate il meno possibile funzionalità algoritmiche (metodi), con l'obiettivo di non utilizzare il linguaggio VHDL come se fosse un linguaggio di programmazione.
Il codice è quindi organizzato in diverse funzionalità, inserite in un singolo modulo ($entity$).
Infine, si sono utilizzati nomi di variabili intuitivi per rendere il codice facilmente interpretabile ai fini di eventuali modifiche.

1.4 Note aggiuntive sulla specifica

Il modulo deve essere progettato per poter codificare più immagini, ma l'immagine da codificare non verrà mai cambiata all'interno della stessa esecuzione, ossia prima che il modulo abbia segnalato il completamento tramite il segnale o_done = 1.
Il modulo partirà nell'elaborazione quando un segnale i_start in ingresso verrà portato a 1.
Il segnale i_start rimarrà alto fino a che il segnale o_done non verrà portato alto; al termine della computazione (e una volta scritto il risultato in memoria) infatti, il modulo dovrà alzare il segnale o_done che notifica la fine dell'elaborazione.
Il segnale o_done dovrà rimanere alto fino a che il segnale i_start non è riportato a 0.
Un nuovo segnale start non può essere dato fin tanto che o_done non è stato riportato a 0.
Se a questo punto viene rialzato il segnale di i_start, il modulo dovrà ripartire con la fase di codifica.
Il modulo deve essere progettato considerando che prima della prima codifica verrà sempre dato il reset. Invece, una seconda elaborazione non dovrà attendere il reset del modulo.
Durante la lettura è sufficiente settare l'indirizzo per un solo ciclo di clock, ma il risultato non potrà essere utilizzato fino al successivo ciclo di clock.
Inoltre il componente deve funzionare almeno con un periodo di clock di 100 ns.

1.5 Note aggiuntive sull'applicativo

È stato utilizzato il tool "$Vivado\ 2020.2\ WebPACK\ Edition$" impostato coi parametri di default.
La FPGA Target utilizzata è la $xc7a200tfbg484\_1$.

2. DESIGN

2.1. Interfaccia del componente

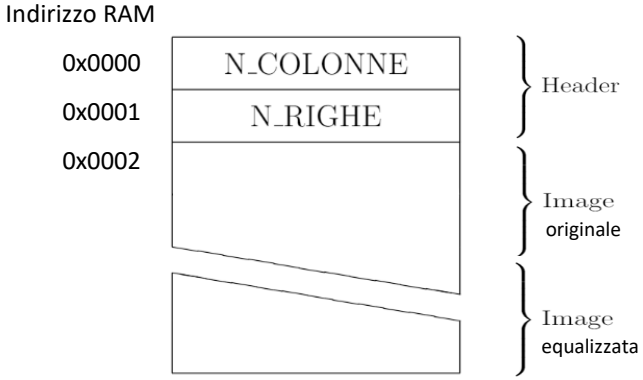
L'interfaccia del componente, così come presentata nelle specifiche, è la seguente:

```
entity project_reti_logiche is
  port (
    i_clk : in std_logic;
    i_start : in std_logic;
    i_rst : 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 particolare:

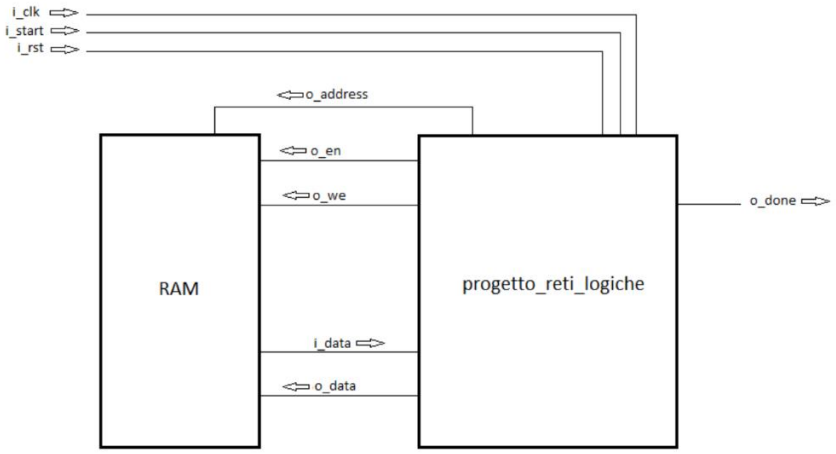
- i_clk è il segnale di $CLOCK$ in ingresso generato dal test bench;
- i_start è il segnale di $START$ generato dal test bench;
- i_rst è il segnale di $RESET$ che inizializza la macchina pronta per ricevere il primo segnale i_start;
- i_data è il segnale (vettore) che arriva dalla memoria in seguito ad una richiesta di lettura;
- $o\_address$ è il segnale (vettore) di uscita che manda l'indirizzo alla memoria;
- o_done è il segnale di uscita che comunica la fine dell'elaborazione e il dato di uscita scritto in memoria;
- o_en è il segnale di $ENABLE$ da dover mandare alla memoria per poter comunicare (sia in lettura che in scrittura);
- o_we è il segnale di $WRITE\ ENABLE$ da dover mandare alla memoria per scriverci. Per leggere da memoria esso deve essere $\mathbf{0}$;
- o_data è il segnale (vettore) di uscita dal componente verso la memoria.

Come già detto, il componente si interfaccia con un chip RAM sincrono ausiliario nel quale è stata precaricata un'immagine (ingresso del nostro componente, uscita della RAM) RAW in toni di grigio ed i relativi metadati, quali la larghezza colonne e righe in pixels.
Il formato dell'immagine è il seguente:



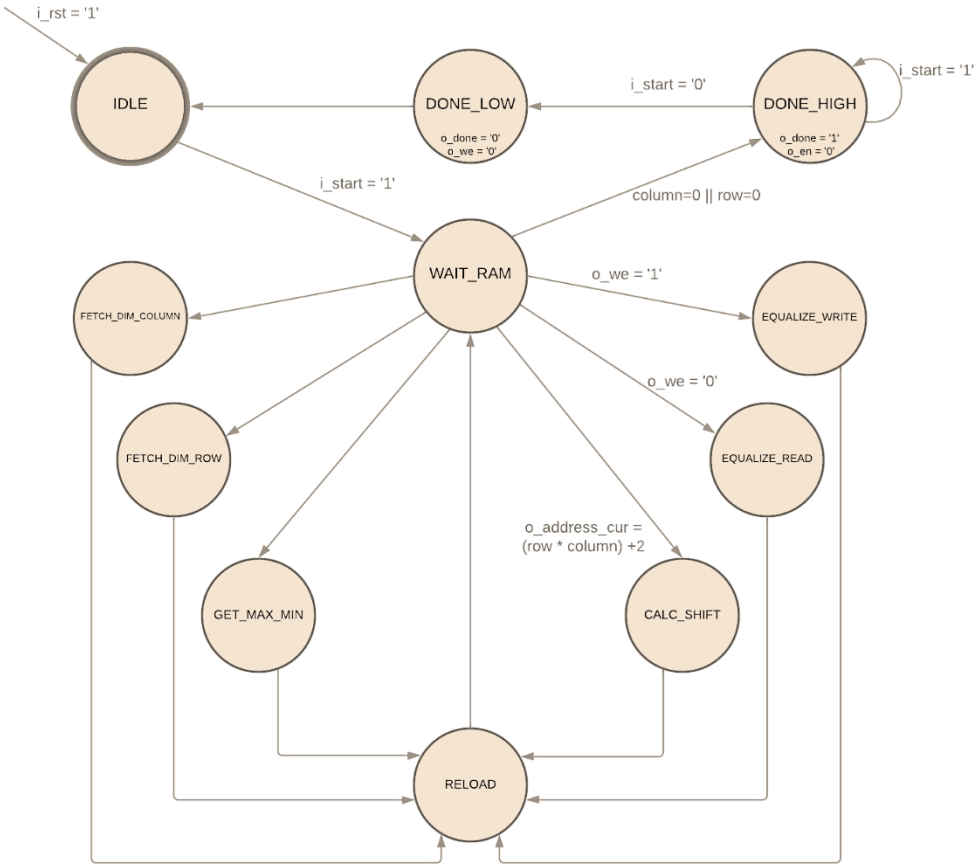
2.2. Architettura

Il funzionamento alla base del componente è stato implementato attraverso una FSM che usa come segnale di ingresso i_start.
Trattandosi di un problema di complessità ridotta abbiamo optato per una soluzione mono modulare e mono processo che scandisce i vari stati ed esegue le operazioni nel suo contesto.
La macchina sequenziale si presenta in questo modo:



2.3. Macchina a Stati Finiti

La FSM che si è deciso di implementare sulla base dell'algoritmo pensato è composta dai seguenti stati:



2.3.1. IDLE

Stato iniziale della macchina nella quale attende il segnale i_start dalla memoria RAM per iniziare una nuova elaborazione, ma non prima di aver inizializzato i segnali necessari.
Ogni altro stato ritorna qui in caso di segnale i_rst alzato.

2.3.2. RELOAD

Stato di attesa e preparazione dei segnali: stato in cui vengono aggiornati i segnali ed indirizzi, si stabilisce qual è lo stato successivo da considerare in base a determinati valori dei segnali.

2.3.3. WAIT_RAM

Stato in cui si attende la risposta della memoria in seguito alla richiesta di un dato.

2.3.4. FETCH_DIM_COLUMN e FETCH_DIM_ROW

Stati in cui viene richiesta la dimensione del numero di colonne e righe alla memoria con o_en alto.
I dati saranno resi disponibili su i_data al clock successivo.
Nel passaggio tra i 2 stati vengono incrementati gli indirizzi e si attraversa lo stato di $RELOAD$ nel quale viene controllato che la dimensione sia diversa da 0 ed atteso un ciclo di clock per la memoria, in caso contrario si passa a $DONE\_HIGH$.

2.3.5. GET_MAX_MIN

Stato dove viene scorsa tutta l'immagine per la prima volta alla ricerca del valore massimo e minimo di pixel ai fini della computazione dello $shift\_level$.
Ciò viene fatto ritornando in questo stato dopo essere passati dal $RELOAD$ per incrementare indirizzo ed attendere la memoria.

2.3.6. CALC_SHIFT

In questo stato, avendo a disposizione il valore massimo e minimo dei pixel dell'immagine si procede al calcolo dello $shift\ level$ tramite un semplice controllo a soglie in base alla differenza tra il valore massimo e minimo.
Prima di uscire si setta l'indirizzo della memoria di nuovo al primo pixel dell'immagine, per procedere all'equalizzazione.

2.3.7. EQUALIZE_READ

Stato dove avviene il calcolo del valore del pixel equalizzato.
Viene effettuato un confronto per stabilire se il risultante è maggiore di 254, in tal caso mando in scrittura allo stato seguente il valore 255, altrimenti quello equalizzato.
Mi muovo nel $RELOAD$ dove modifico l'indirizzo di scrittura alla posizione corretta e salvo nel registro temporaneo l'indirizzo di lettura successivo a quello attuale.

2.3.8. EQUALIZE_WRITE

Stato dove avviene la scrittura sulla memoria all'indirizzo $o\_address$ tramite o_data del valore determinato nello stato precedente.
Se l'indirizzo dove effettuare la prossima lettura è successivo all'ultimo pixel dell'immagine originale allora vado in $DONE\_HIGH$ altrimenti ripeto.

2.3.9. DONE_HIGH e DONE_LOW

Pongo il segnale o_done alto ed attendo la risposta $i\_start = 0$.
Allora mi sposto prima in $DONE\_LOW$, in cui abbasso il segnale di o_done, e successivamente in stato $IDLE$ aspettando il nuovo start.

2.4. Segnali interni

- $state\_cur$, $state\_next$ TIPO: $state$
  Segnali che tengono traccia dello stato corrente e prossimo che la FSM deve raggiungere.
- $o\_address\_cur$ TIPO: $std\_logic\_vector(15\ downto\ 0)$
  Segnale di supporto ad $o\_address$ che contiene un indirizzo di lettura della RAM.
- new_pixel_value TIPO: $std\_logic\_vector(7\ downto\ 0)$
  Segnale che contiene il nuovo valore del pixel equalizzato.
- $column$, row TIPO: $integer$
  Segnali che contengono rispettivamente il numero di colonne e righe dell'immagine.
- max , min TIPO: $integer$
  Segnale che contiene il massimo / minimo valore dei pixel dell'immagine.
- $shift\_level$ TIPO: $integer$
  Segnale che contiene il valore dello $shift\_level = (8 - FLOOR(LOG2(delta + 1)))$.

2.5. Ottimizzazioni e scelte progettuali

- Ai fini dell'ottimizzazione si sono applicate alcune strategie, certe già citate nelle note aggiuntive sul codice [1.3], per esempio, tenendo memorizzate nel componente meno informazioni possibili ha favorito un'occupazione d'area del modulo, in termini di FF e LUT, molto più piccola rispetto alla FPGA scelta.
- Il progetto realizzato risulta anche relativamente scalabile, poiché se si aumentassero le dimensioni dell'immagine o la quantità di esse, sarebbe sufficiente apportare minori modifiche che non andrebbero ad impattare molto sull'area del modulo.
- La FSM realizzata non presenta la soluzione a numero di stati minimo in quanto sarebbe possibile ridurre ulteriormente il diagramma per ottenere una soluzione più compatta, ad esempio, accorpando $FETCH\_DIM\_COLUMN$ e $FETCH\_DIM\_ROW$ in un unico stato $FETCH\_DIM$ dove si torna 2 volte, oppure accorpando gli stati $DONE\_HIGH$ e $DONE\_LOW$ in un unico stato $DONE$, ottenendo una soluzione più ottimizzata.
  Tuttavia, abbiamo ritenuto che lasciare separati questi stati fosse la scelta migliore in termini di operabilità e chiarezza del codice, in quanto più facilmente interpretabile per ottimizzazioni future e facilmente leggibile nei casi in cui occorra identificare e modificare delle funzionalità.
- Tutto ciò si è quindi tradotto nella progettazione di una FSM con anche un attento utilizzo dei bus per comunicare con la RAM. La RAM impone infatti un limite massimo alla velocita raggiungibile, dal momento che ad ogni ciclo di clock corrisponde solo una lettura o una scrittura.
- Inoltre, il fatto di non poter accorpare più operazioni in un unico stato è dovuto proprio a questa limitazione della RAM poiché l'aggiornamento corretto di un segnale o la possibilità di scegliere correttamente il prossimo stato della macchina non poteva essere deciso immediatamente, occorreva aspettare un ulteriore ciclo di clock.
  Questa limitazione ci ha portato ad aggiungere uno stato per controllare i risultati e selezionare gli stati prossimi della FSM, lo stato $RELOAD$, ed un ulteriore stato di sincronizzazione degli indirizzi, lo stato $WAIT\_RAM$.

3. RISULTATO DELLA SINTESI

Il componente è stato sintetizzato con successo, di seguito alcuni dei risultati ottenuti.

3.1. Componenti Sintetizzati

Analizzando il $Vivado\ Synthesis\ Report$ troviamo che sono stati sintetizzati dei registri, 17, per un totale di 95 FF:

Num. Bit	Quantità	Descrizione
16	2	$o\_address$ e $o\_address\_cur$, indirizzi RAM a 16 bit.
9	1	Registro temporaneo per il confronto tra il pixel equalizzato e 254.
8	9	Tutti gli altri registri utilizzati.
4	2	Servono per i 2 segnali di stato ($state\_cur$, $state\_next$), 2^4 bit per gli 11 stati.
1	3	o_en, o_we, o_done.

Altri componenti sintetizzati: 7 Adder e 37 Muxer.

3.2. Area utilizzata sulla FPGA

Con le scelte progettuali si è riusciti, anche se non era l'obiettivo principale, a mantenere limitata l'area occupata dal componente sulla FPGA target.
Pensando che potesse essere utilizzato per equalizzazioni più complesse, dopo adeguate modifiche, non sarebbe stato saggio sprecare risorse che potrebbero essere impiegate per altro.
Effettuando un $report\_utilization$ fornito da Vivado vediamo l'area occupata dal design sintetizzato, ottenendo anche le percentuali di occupazione rispetto alla FPGA target:

Tipo	Utilizzati	Disponibili	Percentuale
LUT (usate come logica)	245	134600	0.18%
FF (registri)	95	269200	0.04%

3.3. Nota aggiuntiva sulla sintesi

Notiamo che la percentuale di utilizzo talmente bassa non giustifica un'ulteriore ottimizzazione dell'area utilizzata.
Il ritardo del Datapath col circuito da noi scritto sull'FPGA da noi selezionato durante il percorso critico, dopo che tutte le porte logiche hanno commutato, è di $11.442\ ns$.
Il nostro circuito può funzionare sulla FPGA scelta fino ad una frequenza (quindi con Slack di $0\ s$) di $3.36\ Mhz$ (= $\frac{1}{11{,}442ns}$) con il giusto lavoro di routing del componente.

4. SIMULAZIONI & TESTING

Il corretto funzionamento del componente sintetizzato è stato verificato tramite l'utilizzo di molteplici test bench randomici, generati automaticamente attraverso l'uso di un tool scritto in Python.
Le modalità di simulazione eseguite sono state: ***Behavioral*** e ***Post Synthesis Functional***.
A partire dai test bench di esempio ne sono stati costruiti altri per testare la robustezza del componente, soprattutto per testare il corretto funzionamento nei casi limite (corner case) di esecuzione.
Il componente ha risposto positivamente a tutti i test a cui è stato sottoposto.
Sono di seguito riportati i casi di test ritenuti più significativi in ***Post Synthesis Functional*** ai fini del corretto funzionamento del componente.

4.1. Test bench generico fornito dal docente

Andiamo a studiare il comportamento del componente in una situazione di carattere generale su un'immagine 2 x 2.

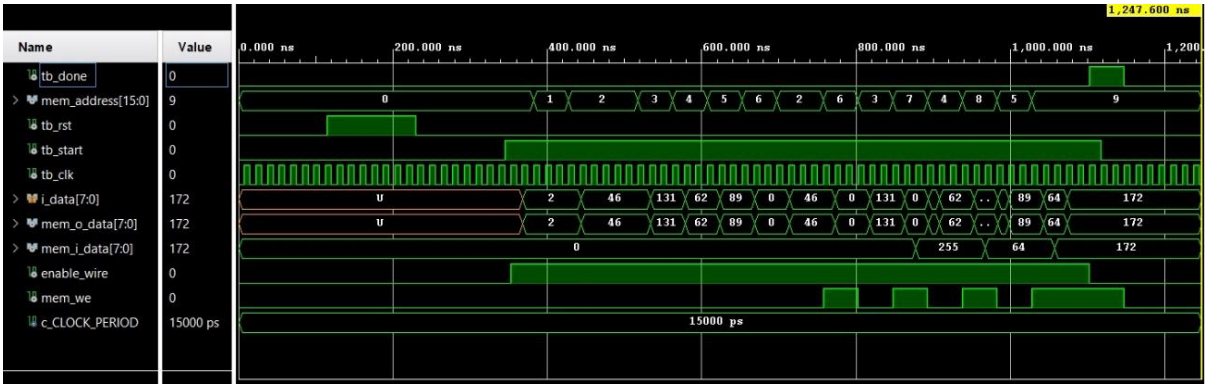


Figura 4.1: Test bench generico immagine 2x2, waveform dei segnali in Post Synthesis Functional Simulation

La seguente sequenza di numeri mostra il contenuto della memoria al termine dell'elaborazione. I valori che qui sono rappresentati in decimale, sono memorizzati in memoria con l'equivalente codifica binaria su 8 bit senza segno:

Indirizzo memoria	Valore	
0	2	\\ Byte più significativo numero colonne
1	2	\\ Byte più significativo numero righe
2	46	\\ Primo Byte immagine
3	131	
4	62	
5	89	
6	0	\\ Primo Byte immagine equalizzata
7	255	
8	64	
9	172	

4.2. Verifica corner case su pixel e SHIFT_LEVEL

4.2.1. Tutti i pixel a 0 con SHIFT_LEVEL massimo

Il test verifica il corretto funzionamento dell'equalizzazione di un'immagine 3 x 2 con tutti i pixel al valore 0. Con tutti i valori a 0, si avrà uno SHIFT_LEVEL pari a 8, valore massimo che può assumere.
Di conseguenza, si otterrà un'immagine con tutti i pixel equalizzati a 0.

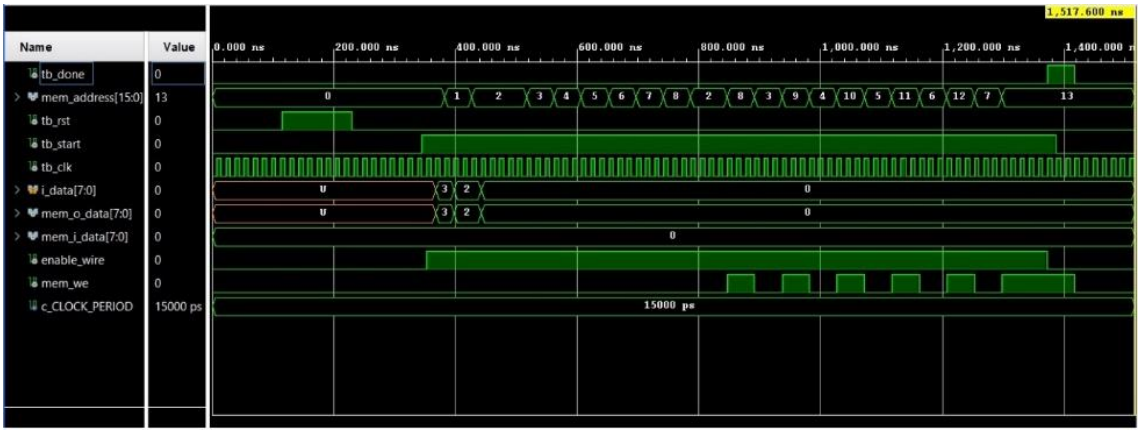


Figura 4.2.1: Test bench pixel a 0, waveform dei segnali in Post Synthesis Functional Simulation

4.2.2. Tutti i pixel a 255 con SHIFT_LEVEL massimo

Il test verifica il corretto funzionamento dell'equalizzazione di un'immagine 3 x 2 con tutti i pixel al valore 255. Con tutti i valori a 255, si avrà ancora uno SHIFT_LEVEL pari a 8.
Di conseguenza, si otterrà un'immagine con tutti i pixel equalizzati a 0.

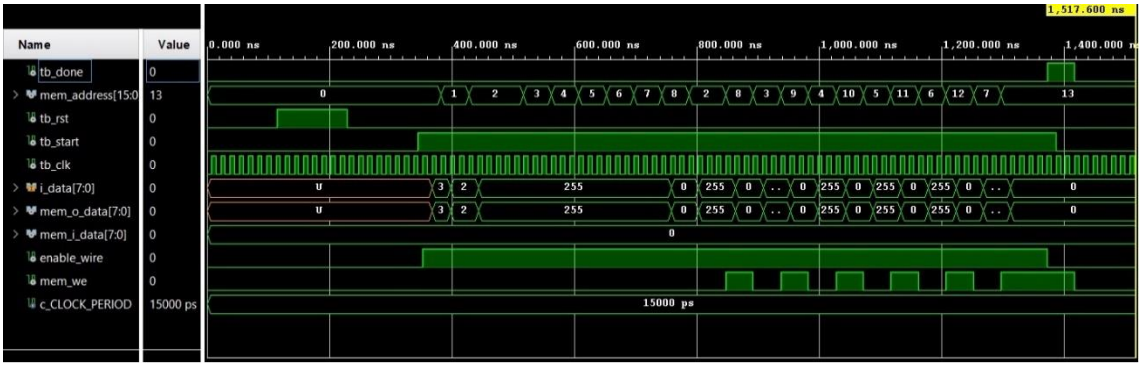


Figura 4.2.2: Test bench pixel a 255, waveform dei segnali in Post Synthesis Functional Simulation

4.2.3. Valori dei pixel compresi tra 255 e 0 con SHIFT_LEVEL minimo

Il test verifica il corretto funzionamento dell'equalizzazione di un'immagine 3 x 2 con pixel aventi valori 0 o 255. In questo modo si ottiene uno SHIFT_LEVEL pari a 0, valore minimo che può assumere.
Di conseguenza, si otterrà un'immagine equalizzata con valori uguali a quelli dell'immagine originale nelle loro rispettive posizioni.

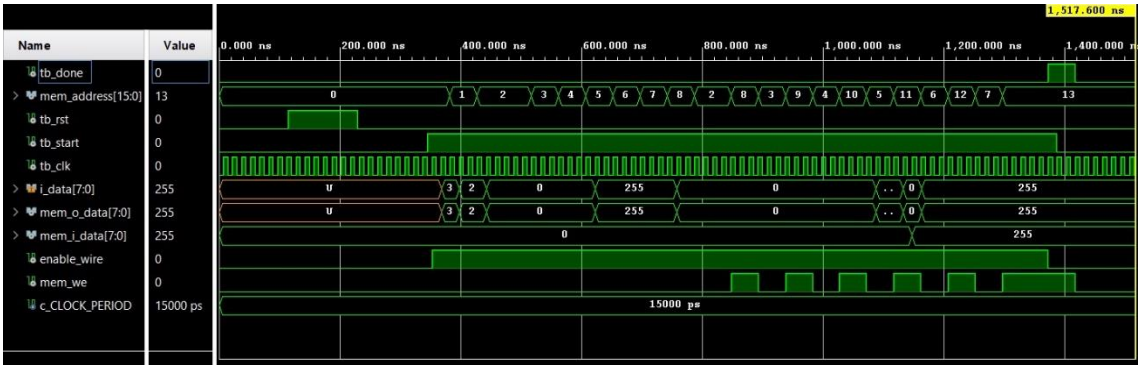


Figura 4.2.3: Test bench pixel compresi tra 0 e 255 , waveform dei segnali in Post Synthesis Functional Simulation

4.3. Verifica corner case sulle dimensioni

4.3.1. Test dimensione minima 1 x 1 pixel

Il test verifica il corretto funzionamento dell'equalizzazione di un'immagine di 1 pixel, dimensione 1 x 1.

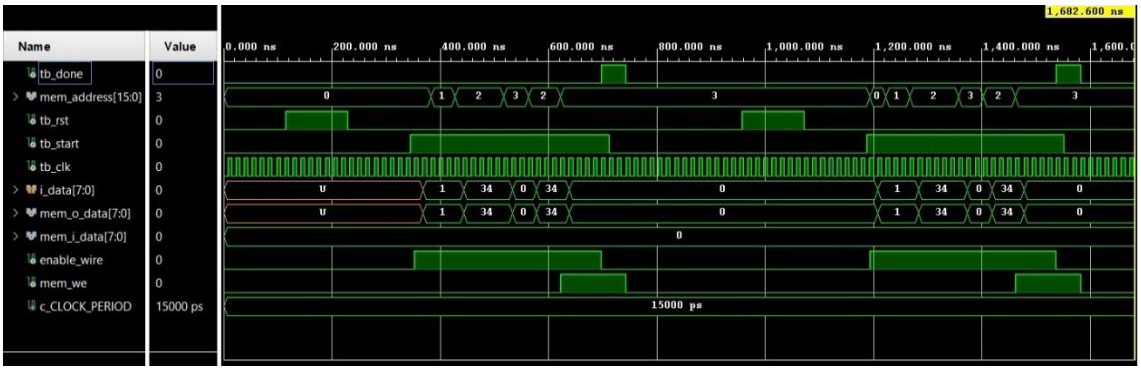


Figura 4.3.1: Test bench imagine 1x1, waveform dei segnali in Post Synthesis Functional Simulation

4.3.2. Test 0 pixel

Il test verifica il corretto funzionamento dell'equalizzazione di un'immagine con 0 pixel.
Dopo la lettura del numero di colonne pari a 0 (equivalentemente accadrebbe se il numero di righe fosse 0) la FSM finisce di computare, alza o_done a '1' e si prepara successivamente ad una nuova computazione abbassando o_done a '0' una volta che i_start viene portato a '0'.

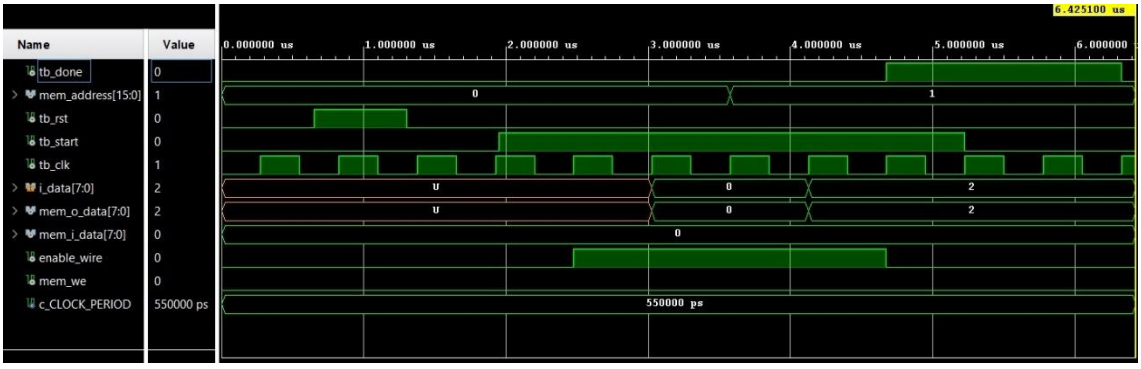


Figura 4.3.2: Test bench 0 pixel, waveform dei segnali in Post Synthesis Functional Simulation

4.3.3. Test dimensione massima 128 x 128 pixel

Il test verifica il corretto funzionamento dell'equalizzazione di un'immagine avente dimensione massima 128 x 128 pixel, presentando una raffigurazione parziale del testing per chiarezza.

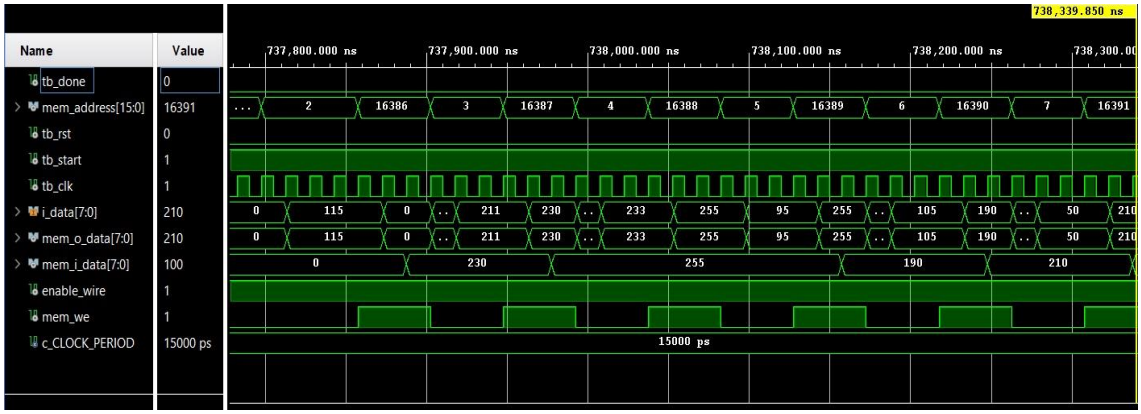


Figura 4.3.3: Test bench imagine 128x128 inizio della lettura-scrittura dei pixel equalizzati, waveform dei segnali in Post Synthesis Functional Simulation

4.4. Verifica corner case sul funzionamento dei segnali

4.4.1. Reset asincrono

Il test verifica che il trigger asincrono del segnale di reset non comprometta la computazione e che questa ricominci ritornando allo stato iniziale $IDLE$.

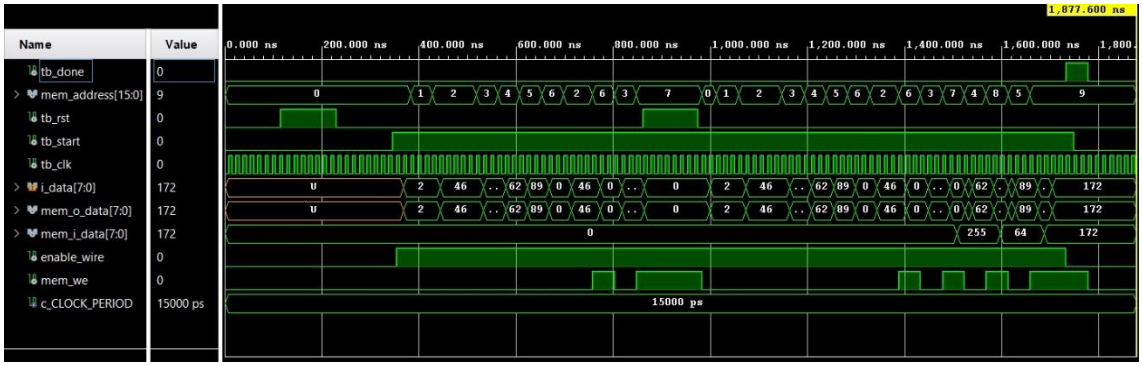


Figura 4.4.1: Test bench reset asincrono, waveform dei segnali in Post Synthesis Functional Simulation

4.4.2. Test computazione multipla

Il test verifica la corretta equalizzazione di più immagini consecutive (in questo esempio tre immagini consecutive).

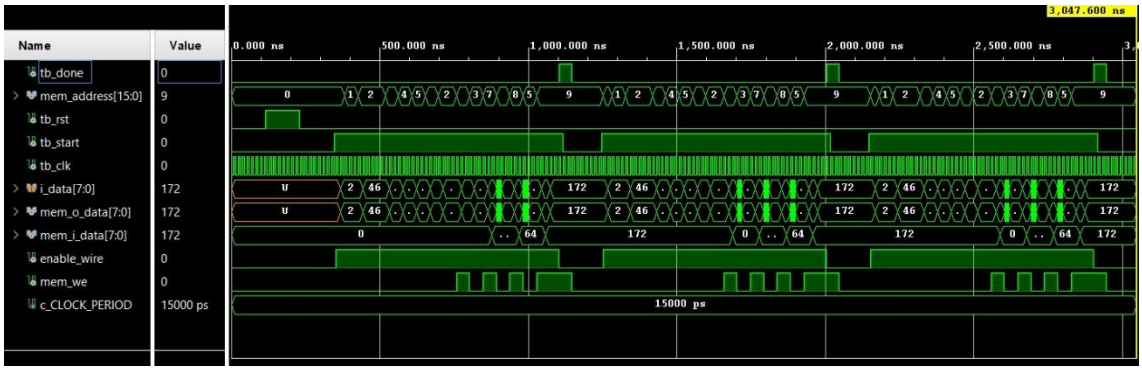


Figura 4.4.2: Test bench computazione multipla, waveform dei segnali in Post Synthesis Functional Simulation

4.5. Verifica su test bench generici

4.5.1. Test su immagini con massimo e minimo agli estremi

Il test verifica il corretto funzionamento per un'immagine 3 x 2 che presenta il valore massimo 235 al primo pixel e il valore minimo 0 all'ultimo pixel.

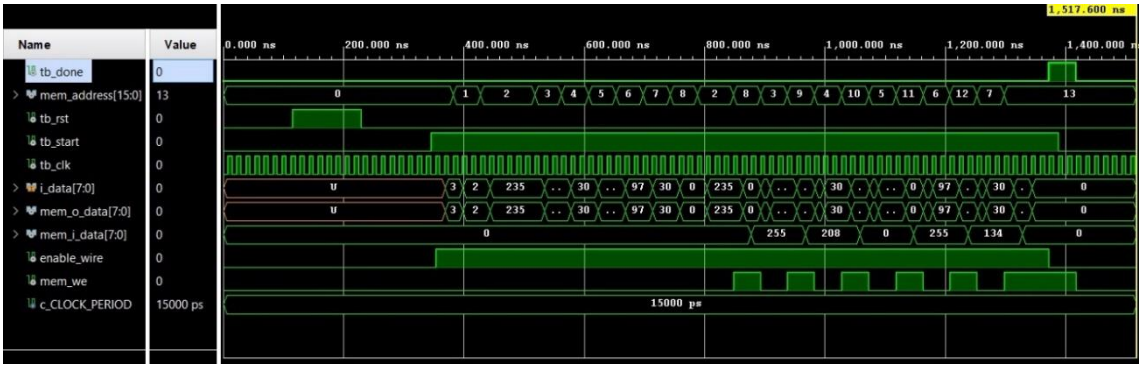


Figura 4.5.1: Test bench immagine max e min agli estremi, waveform dei segnali in Post Synthesis Functional Simulation

4.5.2. Test su immagine generica

Il test verifica il corretto funzionamento per un'immagine 4 x 3 con valore massimo 133 e valore minimo 122.

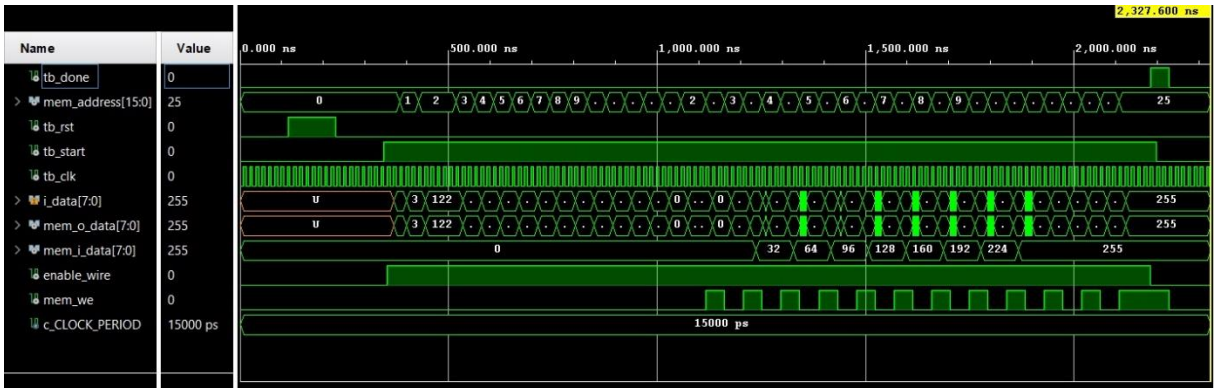


Figura 4.5.2: Test bench immagine generica, waveform dei segnali in Post Synthesis Functional Simulation

5. Conclusioni

Riassumendo si è creato un design con le seguenti caratteristiche:

- Funzionante in pre e post-sintesi.
- Un codice facilmente interpretabile e di funzionamento sequenziale per agevolare eventuali modifiche future del componente.
- Ottimizzato in modo che ogni lettura della RAM venga eseguita solo se strettamente necessaria (una lettura per trovare il valore massimo e il minimo e una lettura per calcolare il nuovo valore del pixel equalizzato).
- Uso limitato di risorse
- Frequenza massima di clock impostabile: $3.36\ Mhz$.
- Utilizzo di LUT pari al 0.18%.
- Utilizzo di FF pari al 0.04%.

Possiamo concludere di aver implementato un progetto, fedele alla nostra idea iniziale, in cui si è trovato un compromesso tra ottimizzazione e chiarezza del codice di un componente che funziona a dovere.