Prova Finale di Reti Logiche

Rivi Gabriele 910564 10663569
Redaelli Mattia 907429 10622823

1 Aprile, 2021

Indice

1 Introduzione e Specifiche

1.1 Presentazione del progetto

L'obiettivo del progetto è quello di scrivere un programma VHDL che simuli il comportamento di una rete logica capace di equalizzare l'istogramma di immagini in toni di grigio, ovvero ricalibrare il contrasto.

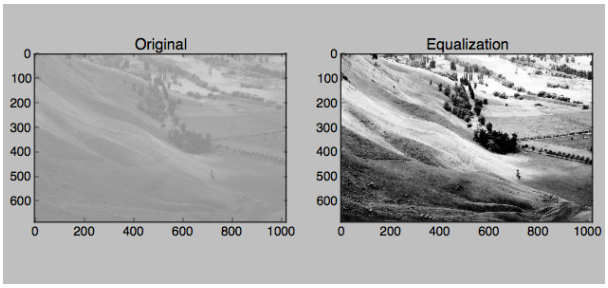

Fig. 1: Esempio di equalizzazione di una immagine.

1.2 Specifiche progettuali

L'algoritmo da implementare consiste di una versione semplificata del classico metodo di equalizzazione. Le immagini saranno su una scala di grigio a 256 livelli, rappresentate pixel per pixel, ognuno dei quali è identificato da un valore (tra 0 e 255) che indica l'intensità del suo tono di grigio. Ogni pixel viene modificato sulla base di un preciso algoritmo. Più precisamente, dopo aver valutato quali sono il massimo ed il minimo valore d'intensità dei pixel, si procede a calcolare il nuovo valore per ogni pixel seguendo questo algoritmo:

1. *DELTA_VALUE* = MAX_PIXEL_VALUE – MIN_PIXEL_VALUE
2. *SHIFT_LEVEL* = (8 – FLOOR(LOG2(DELTA_VALUE +1)))
3. *TEMP_PIXEL* = (CURRENT_PIXEL_VALUE – MIN_PIXEL_VALUE) « SHIFT_LEVEL
4. *NEW_PIXEL_VALUE* = MIN(255 , TEMP_PIXEL)

Successivamente, il nuovo valore del pixel viene riscritto in memoria.

0	Columns: 2
1	Rows: 2
2	46
3	131
4	62
5	89
6	0
7	255
8	64
9	172

Nella tabella rappresentata è posto un esempio di memoria alla fine dell'equalizzazione di un'immagine.
Agli indirizzi 0 ed 1 sono presenti le dimensioni dell'immagine.
Gli indirizzi da 2 a 5 (in grigio) corrispondono ai pixel iniziali, prima che vengano processati. In blu invece, dalla posizione 6 fino alla posizione 9, troviamo i pixel scritti in seguito all'equalizzazione.

1.3 Interfaccia del Componente

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;
```

In particolare:

- `i_clk`: segnale di CLOCK in ingresso generato dal TestBench;
- `i_rst`: segnale di RESET che inizializza la macchina pronta per 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 (=1) per poter scriverci. Per leggere da memoria deve essere pari a 0;
- `o_data`: segnale (vettore) di uscita dal componente verso la memoria.

1.3.1 Descrizione Memoria

Il modulo implementato si interfaccia con una memoria indirizzata al byte, in cui ogni indirizzo corrisponde ad un pixel dell'immagine. La memoria è così costruita:

- `Indirizzo 0`: contiene il numero di colonne dell'immagine (*columns*).
- `Indirizzo 1`: contiene il numero di righe dell'immagine (*rows*).

Dall'indirizzo 2 in avanti iniziano ad essere presenti i valori dei pixel da processare. Una volta finito l'intero processo, i pixel dell'immagine equalizzata si troveranno anch'essi in memoria, scritti in sequenza a partire dall'indirizzo $2 + (rows \cdot columns)$

2 Design scelto per FSM

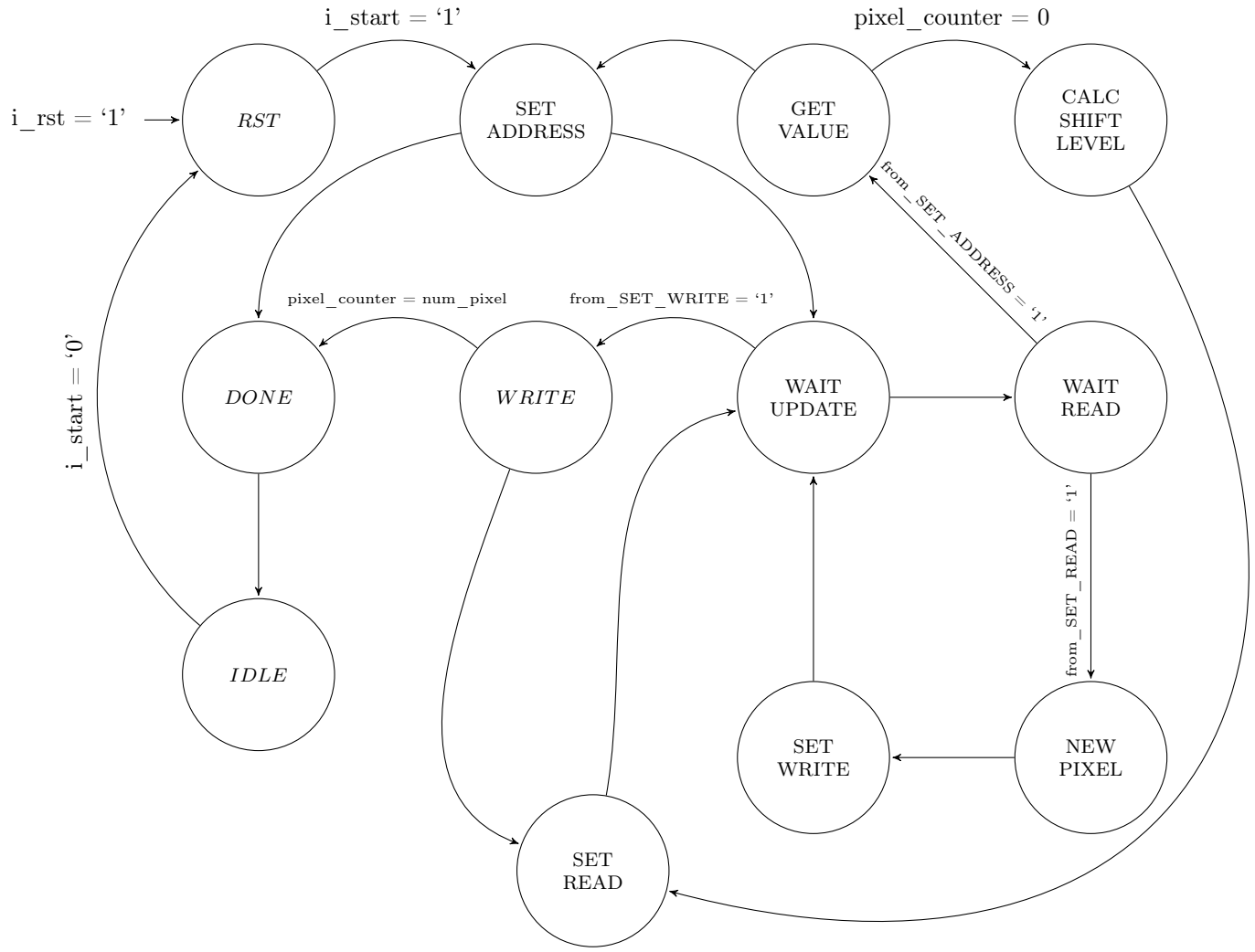

Fig. 2: Modello della macchina a stati finiti.

La macchina a stati è principalmente composta di due diversi cicli.
Il primo ciclo composto degli stati (SET_ADDRESS,WAIT_UPDATE,WAIT_READ, GET_VALUE) si occupa di leggere tutti i pixel dell'immagine così da poter trovare il valore massimo ed il minimo.
Il secondo ciclo composto degli stati (SET_READ,WAIT_UPDATE, WAIT_READ, NEW_PIXEL, SET_WRITE,WRITE) ripartendo dall'indirizzo del primo pixel in memoria, li processa singolarmente calcolando il nuovo valore del pixel, scrivendolo in memoria in coda all'immagine non equalizzata.
Una volta terminata l'equalizzazione dell'immagine, la macchina si riporta in posizione iniziale pronta ad una nuova esecuzione.

2.1 Descrizione degli stati

- `RST`: Stato di inizializzazione di segnali e variabili al valore di default. Prosegue in SET_ADDRESS quando il segnale *i_start* viene alzato ad '1' ;

- `SET_ADDRESS`: Stato adibito all'incremento dell'indirizzo di accesso in RAM per il primo ciclo, per una corretta lettura dei dati; al suo interno viene anche inizializzato un contatore al numero di pixel analizzati durante il processo (*pixel_counter*), e nel caso di valori di righe o colonne pari a 0, passa allo stato di DONE per terminare l'esecuzione; in tutti gli altri casi passa a WAIT_UPDATE;

- `GET_VALUE`: Stato adibito alla lettura da memoria. Nei primi due passaggi viene eseguito il fetch dei valori di righe e colonne, per poi passare alla scansione di tutti i pixel per trovare il valore minimo e massimo, utilizzando come supporto un contatore ai pixel processati. Se la scansione dei pixel è terminata si passa a CALC_SHIFT_LEVEL, altrimenti si ritorna a SET_ADDRESS;

- `CALC_SHIFT_LEVEL`: Tramite una funzione di *Threshold Control* si calcola quale sia lo shift_level. È in questo stato che vengono implementati i primi due passi dell'algoritmo visto in 1.2. L'indirizzo viene riportato in corrispondenza del primo pixel e si passa a SET_READ;

- `WAIT_UPDATE`: Stato di attesa per il giusto aggiornamento dei segnali. Usato sia dal primo ciclo che dal secondo; in base ai valori di alcuni segnali Booleani si passa o a WAIT_READ, oppure a WRITE;

- `WAIT_READ`: Stato di attesa per il giusto aggiornamento dei segnali. Usato sia dal primo ciclo che dal secondo; in base ai valori di alcuni segnali Booleani si passa o a GET_VALUE (*primo ciclo*), oppure a NEW_PIXEL (*secondo ciclo*);

- `SET_READ`: Si pone *o_we*='0' per accedere alla RAM in lettura e si imposta l'indirizzo al valore: $2 + pixel\_counter$, ovvero la posizione del pixel da analizzare; passa a WAIT_UPDATE;

- `NEW_PIXEL`: Si implementa qui il terzo passo dell'algoritmo descritto a 1.2; Si passa a SET_WRITE;

- `SET_WRITE`: Si pone l'indirizzo di memoria alla corretta posizione di scrittura del nuovo valore del pixel: $2 + pixel\_counter + rows \cdot columns$ e si esegue il quarto passo dell'algoritmo di cui a 1.2;

- `WRITE`: Si pone *o_we*='1' per accedere alla RAM in scrittura, si incrementa *pixel_counter*, si reimposta l'indirizzo in corrispondenza del primo pixel in memoria; Se tutti i pixel sono stati processati ($pixel\_counter = rows \cdot columns - 1$) si passa a DONE, altrimenti a SET_READ per procedere;

- `DONE`: *o_done* viene alzato a '1' , reset dei segnali di accesso a RAM e si passa a IDLE;

- `IDLE`: quando *i_start* passa a '0' , *o_done* viene abbassato e si passa a RST in attesa di un nuovo start.

3 Scelte progettuali

Si è scelto di implementare la macchina sopra descritta utilizzando un'architettura di tipo behavioural composta di un solo processo.
Ad ogni ciclo di clock il segnale $CURR\_S$ determina lo stato di esecuzione corrente in base all'assegnamento eseguito al ciclo di clock precedente. Tutti i segnali vengono aggiornati in questo modo e ciò ha reso necessario l'inserimento di due stati adibiti alla semplice attesa affinchè i segnali possano aggiornarsi correttamente (WAIT_UPDATE, WAIT_READ).
Inoltre, per lo stesso motivo, si è deciso di utilizzare delle variabili a supporto di alcune operazioni (queste sono infatti aggiornate immediatamente dopo l'assegnamento di valore), tra cui:

- `pixel_counter:` Un contatore per tenere traccia dello stato di avanzamento dell'esecuzione;
- `delta_value, delta_int, temp:` Variabili a supporto del calcolo del valore nuovo pixel.

Per la lettura e la scrittura in memoria si utilizza un segnale di supporto ADDRESS, che ogni volta viene inizializzato all'indirizzo di memoria necessario in quel momento dell'esecuzione.
Per quanto riguarda l'implementazione dell'algoritmo per il calcolo del nuovo valore del pixel, visto ad 1.2, sono state fatte alcune considerazioni tenendo conto della necessità di sintetizzare il componente.
In primis, per implementare il secondo passo, è stato usato un *controllo a soglia* che sostituisce a pieno la funzione descritta sopra e calcola lo shift_level;
Mentre per effettuare l'operazione di $shift - left$ ed ottenere temp_pixel, per poter rappresentare a pieno tutta la scala di valori possibili, è stato utilizzato un segnale di tipo vettore di 16 bit (caso pessimo: pixel valore 255 shiftato di 8 bit), che viene "composto" concatenando i singoli bit.
Poi, per la funzione di $min(255, temp\_pixel)$, viene controllato se il valore di temp_pixel sia superiore a 255 (in quel caso viene utilizzato 255), oppure si utilizza temp_pixel(7 down to 0).

4 Risultati Sperimentali

4.1 Report di Sintesi

Il componente sviluppato é sintetizzabile.
La sintesi é terminata con 0 Errori, 0 Avvisi Critici, 0 Avvisi.
Di seguito sono elencati lo schema e i risultati del report di sintesi:

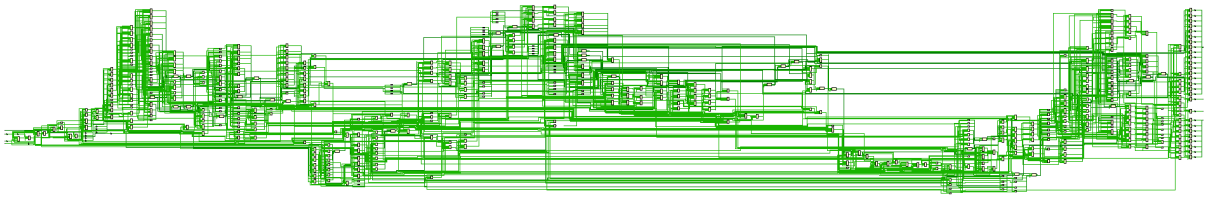

Fig. 3: Schema componente sintetizzato.

Risultati del report di utilizzo:

Utilization — **Post-Synthesis** | Post-Implementation

Graph | **Table**

Resource	Estimation	Available	Utilization %
LUT	305	134600	0.23
FF	145	269200	0.05
IO	38	285	13.33
BUFG	1	32	3.13

Fig. 4: Tabella report Post-Sintesi.

4.2 Simulazioni

Per effettuare la verifica della corretta computazione del componente si sono eseguiti svariati casi di test, tutti funzionanti sia in behavioural sia in post-sintesi funzionale.
La maggior parte di essi sono stati generati randomicamente da un programma python, sia sotto l'ipotesi di non avere reset, sia supponendo di avere dei reset asincroni alla fine di ogni computazione. Si sono inoltre considerati alcuni casi limite.
Di seguito sono elencati i piu significativi:

- **RST Asincrono**:
  `i_rst` posto a '1' nel mezzo di una esecuzione.
  Il componente avverte il cambiamento nel segnale i_rst, termina la computazione e si pone in attesa di un nuovo `i_start` = '1' necessario per ricominciare l'esecuzione dal primo indirizzo in RAM. Nel testbench è stato anche considerato che il segnale i_start passa a '0' in seguito al cambiamento di i_rst (come da specifica).

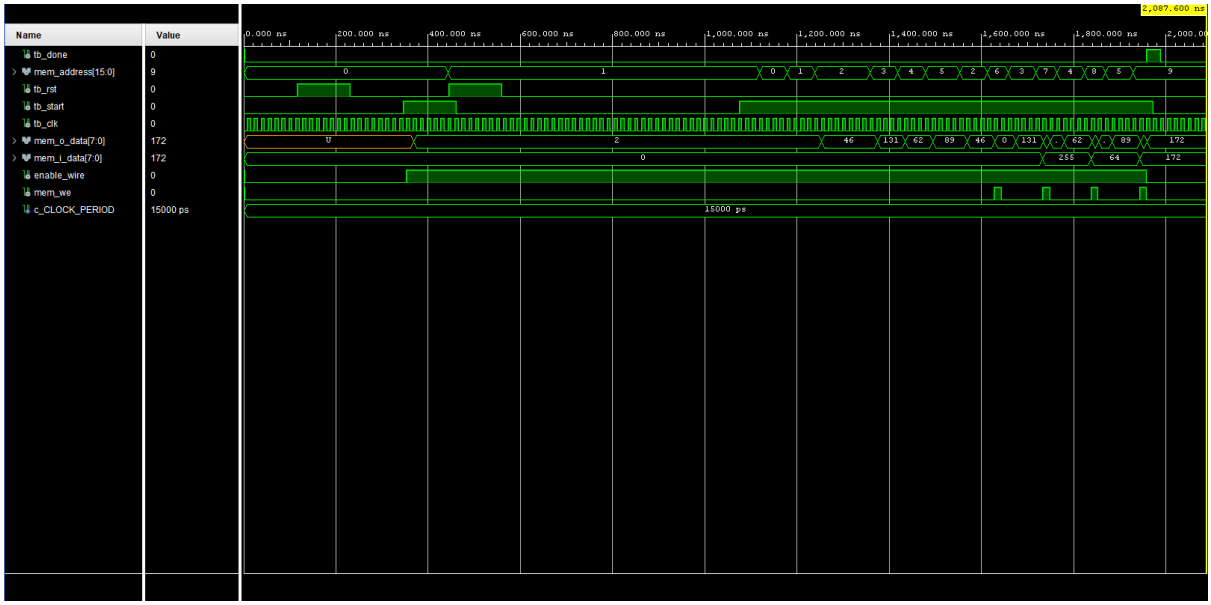

Fig. 5: Risultato test reset asincrono.

- **1 Pixel**:
  Il componente viene testato per verificare che sia in grado di gestire un'immagine composta da un solo pixel.

- **128x128 Pixel**:
  Test effettuato per verificare i limiti superiori delle dimensioni dell'immagine.

- **Tutti i Pixel con valore 0**:
  Eseguito per testare la capacitá di processare un'immagine con tutti i pixel con valore pari al limite inferiore, il risultato del test é analogo al test: *Tutti i Pixel con valore* 255, nel dettaglio *Fig.*6.

- **Tutti i Pixel con valore 255**:
  Questo test verifica la corretta esecuzione di un test bench con tutti i valori dei pixel pari a 255. Oltre a testare il limite superiore per valore del pixel, testa anche il caso limite in cui tutti i valori all'interno della RAM siano uguali. In questo caso il componente alla fine dell'equalizzazione semplificata stamperà tutti 0 in memoria.

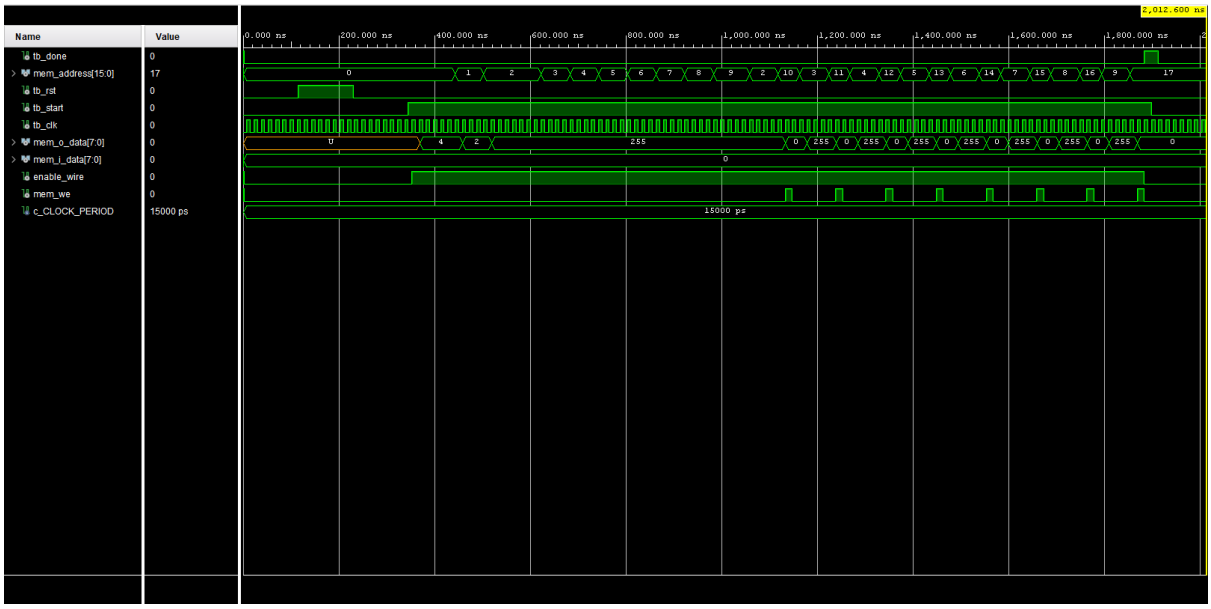

Fig. 6: Risultato test con tutti i pixel pari a 255.

- **0 Pixel**:
  Viene testata l'effettiva capacitá del componente di terminare l'esecuzione nel caso in cui le colonne o le righe abbiano valore pari a 0. Nel caso specifico una volta processate le prime due celle di RAM, la computazione termina passando allo stato DONE, essendo il valore del numero di colonne pari a 0.

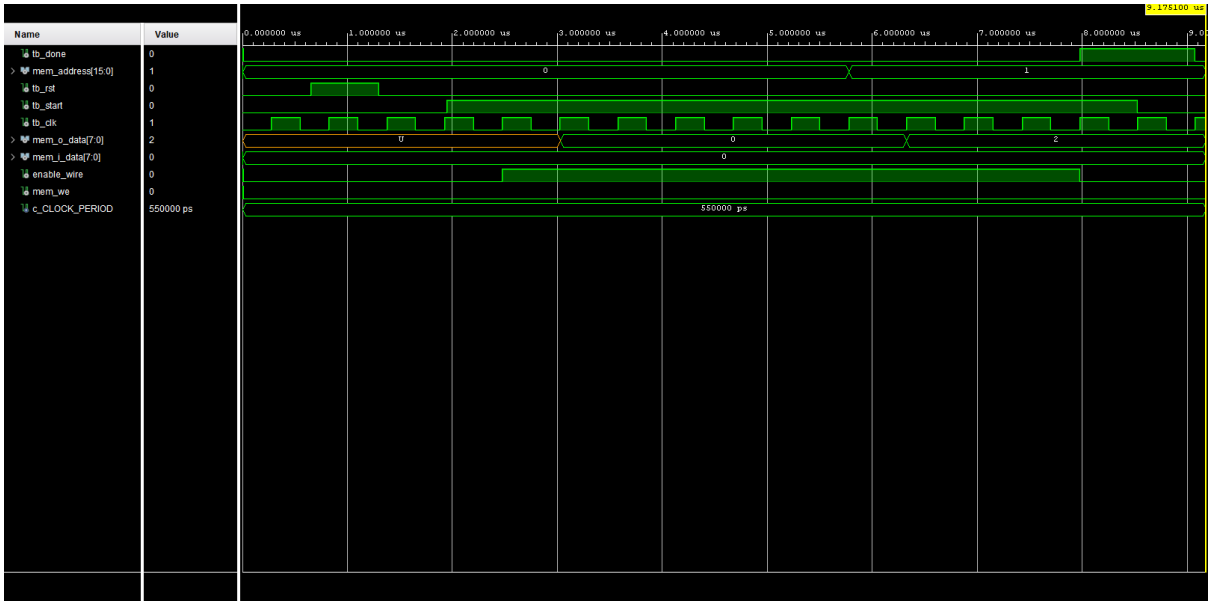

Fig. 7: Risultato test con una dimensione pari a 0.

5 Ottimizzazioni

Si é cercato di ottimizzare il codice il piu possibile al fine di ottenere un minor tempo di esecuzione. Come prima cosa si è introdotto un controllo affinchè si possa evitare di continuare a leggere ed analizzare i pixel dell'immagine nel caso in cui il massimo valore trovato sia il massimo possibile (255) e il minimo valore trovato sia il minimo possibile (0). Di conseguenza nello stato di GET_VALUE, nelle condizioni appena descritte, si passerebbe direttamente allo stato di CALC_SHIFT_LEVEL evitando diversi cicli di clock nel caso di immagini di grandi dimensioni.
Inoltre, ove possibile, si é cercato di riutilizzare gli stati di WAIT per diversi casi.
Ad esempio WAIT_UPDATE viene utilizzato da SET_ADDRESS, SET_WRITE e SET_READ, per determinare lo stato successivo in base a ogni chiamante viene utilizzato un valore boolean FROM_* (*=SET_ADDRESS, SET_WRITE, SET_READ in base al chiamante).

6 Conclusione

In conclusione, il risultato del lavoro é un componente sintetizzabile e correttamente simulabile in post-sintesi, con un totale di 305 *Look Up Tables* (copertura del 0.23%) e un totale di 145 *Flip Flop* (copertura del 0.05%).
La macchina a stati finiti composta da 12 stati, si ritiene esegua correttamente l'implementazione dell'algoritmo descritto nelle specifiche, avendo passato i casi di test limite e la computazione di decina di migliaia di immagini in sequenza.