Prova Finale (Progetto di Reti Logiche)

Prof. William Fornaciari - Anno Accademico 2020/2021

Andrea Paolo Arbasino (Codice Persona 10622220 - Matricola 907000)
Daniele Azzarà (Codice Persona 10658437 - Matricola 911314)

Indice

1 Introduzione

1.1 Scopo del progetto

Lo scopo della prova finale è la realizzazione di un componente hardware in VHDL. Il componente in esame riceve in ingresso un'immagine in scala di grigi a 256 livelli di dimensione massima 128x128 pixel e la salva dopo aver applicato l'algoritmo di equalizzazione dell'istogramma. Tale algoritmo ha lo scopo di ricalibrare il contrasto dell'immagine aumentandolo significativamente nelle aree a basso contrasto locale.

1.2 Specifiche generali

Il componente si interfaccia con un chip RAM in cui è stata precaricata un'immagine. Il Byte all'indirizzo 0 contiene il numero di colonne, quello in posizione 1 il numero di righe. I pixel dell'immagine sono di dimensione 8 bit e memorizzati in memoria a partire dall'indirizzo 2.

N_COL
N_RIG
PRIMO_PIXEL
...
ULTIMO_PIXEL
PRIMO_PIXEL_RIELABORATO
...
ULTIMO_PIXEL_RIELABORATO

Tabella 1: Rappresentazione del chip RAM

I pixel della immagine equalizzata, sempre di 8 bit, sono salvati in memoria a partire dalla posizione 2+(N_COL*N_RIG).
L'algoritmo trasforma ogni pixel dell'immagine nel seguente modo:

DELTA_VALUE = MAX_PIXEL_VALUE – MIN_PIXEL_VALUE
SHIFT_LEVEL = (8 – FLOOR(LOG_2(DELTA_VALUE +1)))
TEMP_PIXEL = (CURRENT_PIXEL_VALUE - MIN_PIXEL_VALUE) $\ll$ SHIFT_LEVEL
NEW_PIXEL_VALUE = MIN(255 , TEMP_PIXEL)

Dove:

- MAX_PIXEL_VALUE e MIN_PIXEL_VALUE indicano rispettivamente il massimo e il minimo valore dei pixel dell'immagine;
- CURRENT_PIXEL_VALUE indica il valore del pixel da trasformare;
- NEW_PIXEL_VALUE indica il valore del nuovo pixel.

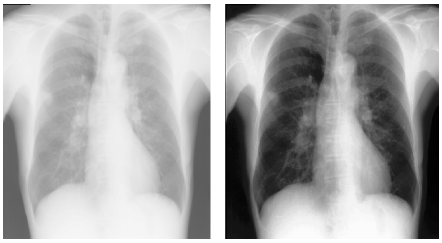

Figura 1: Esempio di immagine rielaborata secondo il metodo di equalizzazione dell'istogramma con particolare attenzione ad una sua applicazione in campo medico.

1.3 Interfaccia del componente

Il componente da descrivere ha un'interfaccia così definita:

```
 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` è il segnale di `CLOCK` in ingresso generato dal Test Bench;
- `i_rst` è il segnale di `RESET` che inizializza la macchina pronta per ricevere il primo segnale di `START`;
- `i_start` è il segnale di `START` generato dal TestBench;
- `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 (=1) per poter scriverci. Per leggere da memoria esso deve essere 0;
- `o_data` è il segnale (vettore) di uscita dal componente verso la memoria.

2 Architettura

Data la natura della specifica (in particolare la presenza di segnali di start e di reset) è risultato naturale pensare ad un approccio risolutivo mediante una macchina a stati finiti. Per realizzare il componente abbiamo deciso di modellare una FSM gestita da un singolo processo.

Quando il segnale `i_start` in ingresso viene portato a 1, il componente inizia l'elaborazione. Al termine della computazione, dopo aver scritto il risultato in memoria, il segnale `o_done` viene portato a 1. A questo punto, il segnale `i_start` può essere abbassato e, una volta avvenuto, viene portato a 0 anche `o_done`, in modo da poter essere pronti ad elaborare un'altra immagine. Infatti, se il segnale `i_start` viene rialzato, il modulo riparte con la fase di codifica.

Inoltre, l'algoritmo utilizzato si ipotizza avere una complessità lineare $\theta(N)$ a causa della necessità di leggere l'immagine due volte: una per individuare MIN_PIXEL_VALUE e MAX_PIXEL_VALUE e una per rileggere i pixel e poterli successivamente rielaborare.

2.1 Stati della macchina

2.1.1 `RESET` state

Il `RESET` state indica lo stato iniziale in cui si attende che il segnale di start venga portato alto. Esso denota che si è pronti ad elaborare un'immagine. Quando i_start è pari a 1 vengono impostati ai valori di default alcuni segnali utili all'elaborazione. Qualunque sia lo stato corrente di elaborazione, se viene alzato il segnale i_rst si torna in questo stato.

2.1.2 `GETCOLONNE` state

Indica lo stato in cui viene letto il Byte all'indirizzo 0 contenente il numero di colonne dell'immagine.

2.1.3 `GETRIGHE` state

Indica lo stato in cui viene letto il Byte all'indirizzo 1 contenente il numero di righe dell'immagine.

2.1.4 `CHECKZERO` state

Indica lo stato in cui si gestisce il caso limite di dimensioni nulle dell'immagine.

2.1.5 `PRODOTTODIMENSIONE` state

Indica lo stato in cui viene svolto il calcolo del numero totale di pixel dell'immagine. Questo dato risulta essere molto importante, in quanto permette di calcolare l'ultimo indirizzo dell'immagine in ingresso e anche l'indirizzo in cui viene scritto l'ultimo Byte dell'immagine rielaborata.

2.1.6 `LETTURA` state

Indica lo stato in cui viene salvato in un buffer il pixel corrente dell'immagine.

2.1.7 `MAXMIN` state

Indica lo stato utilizzato per individuare i pixel di minima e massima intensità di grigio presenti nell'immagine.

2.1.8 `CALCOLODELTA` state

Indica lo stato utilizzato per calcolare il DELTA_VALUE, ossia MAX_PIXEL_VALUE - MIN_PIXEL_VALUE.

2.1.9 `CALCOLOSHIFT` state

Indica lo stato in cui viene calcolato lo SHIFT_LEVEL dell'immagine da elaborare.

2.1.10 `RILETTURA` state

Indica lo stato utilizzato per leggere nuovamente i pixel dell'immagine da elaborare.

2.1.11 `SOTTRAZIONE` state

Indica lo stato ausiliario utilizzato per velocizzare la computazione complessiva della macchina, alleggerendo l'onere di calcolo dello stato `RILETTURA`.
In questo stato viene calcolato CURRENT_PIXEL_VALUE - MIN_PIXEL_VALUE.

2.1.12 `CALCOLOTMP` state

Indica lo stato utilizzato per calcolare TEMP_PIXEL.

2.1.13 `NUOVOPIXEL` state

Indica lo stato utilizzato per calcolare NEW_PIXEL_VALUE.

2.1.14 `SALVA` state

Indica lo stato utilizzato per salvare il NEW_PIXEL_VALUE nel corretto indirizzo di memoria.

2.1.15 `SCORRIMENTO state`

Indica lo stato utilizzato per agevolare la gestione e lo scorrimento degli indirizzi utilizzati per manipolare l'immagine.

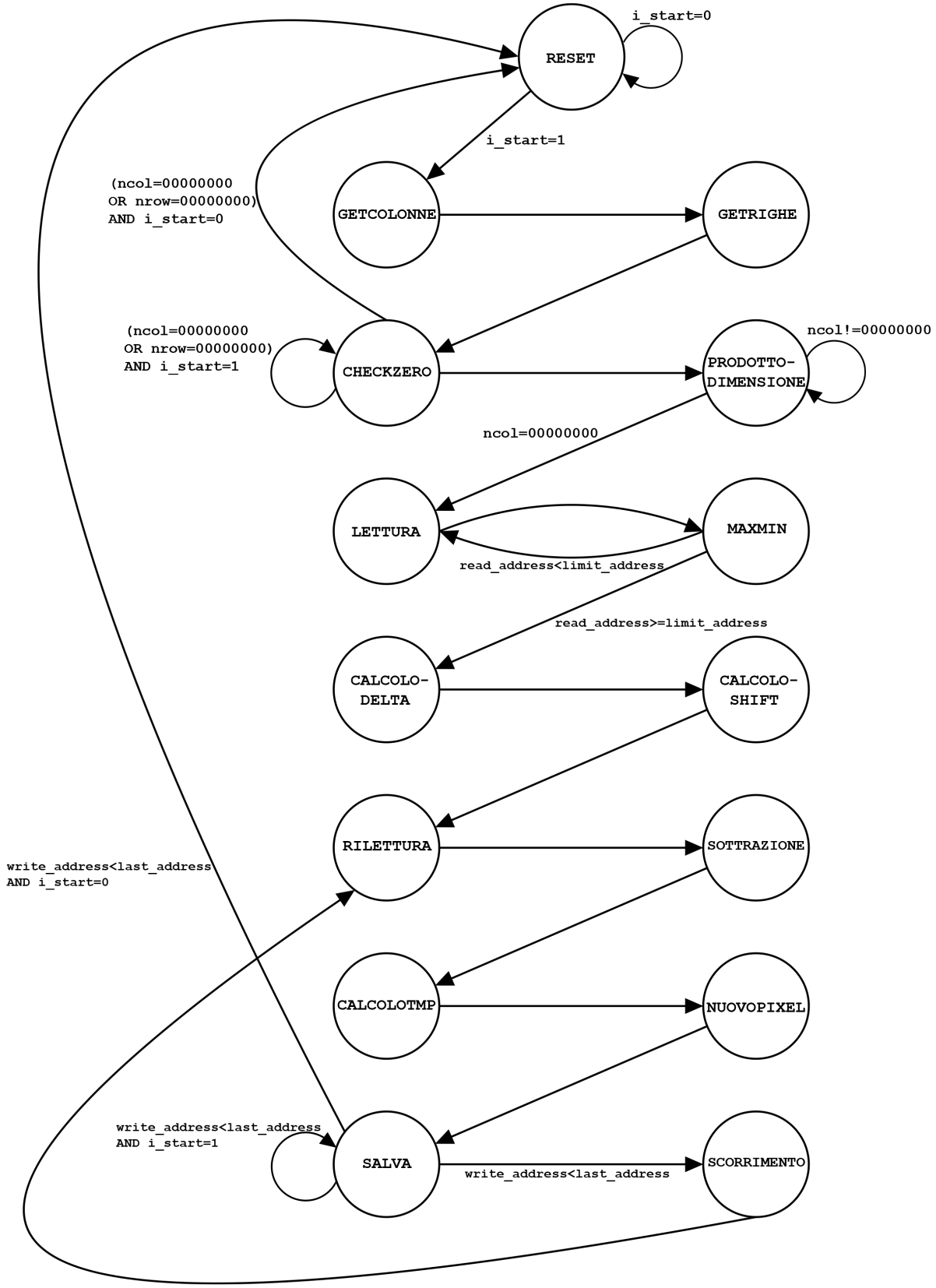

Figura 2: FSM con le principali condizioni di transizione

3 Risultati sperimentali

3.1 Sintesi

Il componente viene sintetizzato correttamente. Vivado fornisce la possibilità di avere la rappresentazione grafica dello schematic. Di seguito se ne riporta l'immagine.

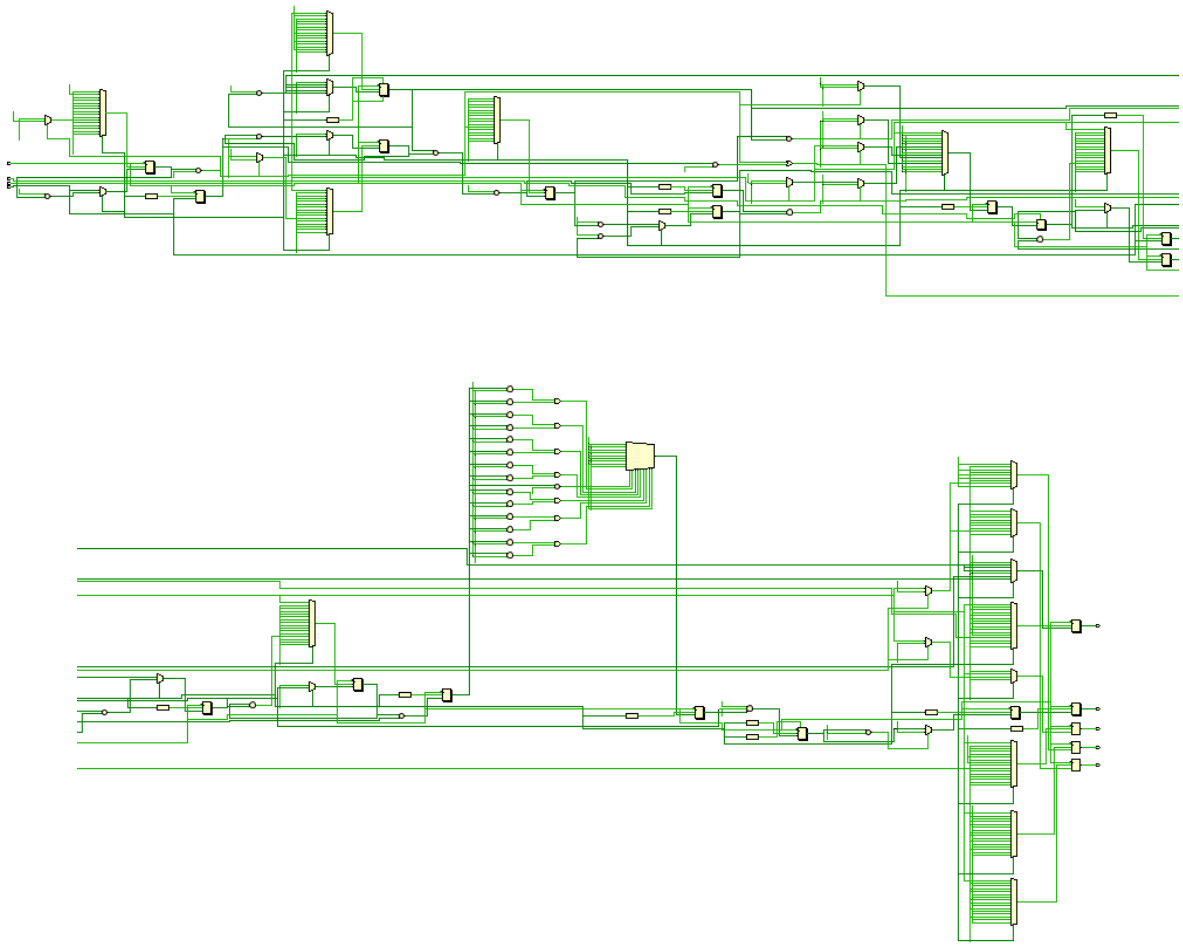

Figura 3: Schematic componente

3.2 Simulazioni

3.2.1 Test Bench pubblico

Data la semplicità e la brevità del test bench fornitoci dai committenti, abbiamo deciso di utilizzarlo come punto di partenza nella fase di testing del componente. Esso, infatti, è risultato essere particolarmente utile per correggere iniziali errori di progettazione ed ottimizzare il funzionamento del componente.

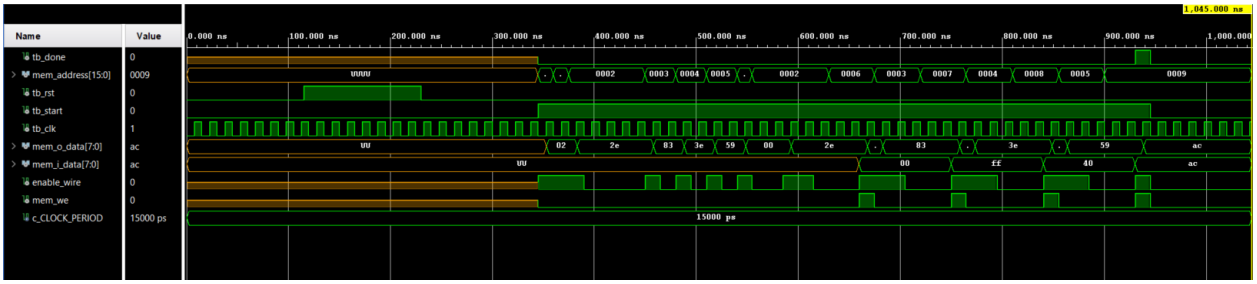

Figura 4: Simulazione Test Bench pubblico

3.2.2 Test Bench pubblico con reset asincrono

Per testare il corretto funzionamento del componente, abbiamo deciso di modificare il caso di test pubblico aggiungendo un reset asincrono. Come mostrato in figura 5, è possibile osservare come si venga immediatamente riportati nello stato di reset.

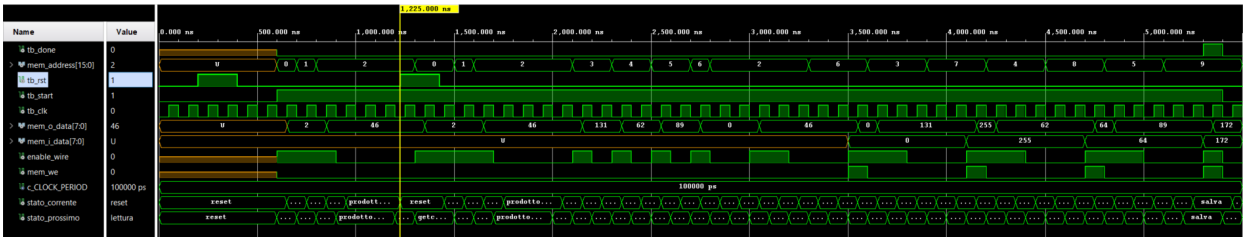

Figura 5: Simulazione Test Bench pubblico con reset asincrono

3.2.3 Ulteriori Test Bench

Per verificare ulteriormente il corretto funzionamento del componente sintetizzato, abbiamo deciso di testarlo mediante approccio di testing sia funzionale (black-box), che strutturale (white-box).

Per quanto riguarda il testing funzionale, abbiamo deciso di utilizzare diverse batterie di test con immagini in numero e dimensione via via crescenti, in modo tale da testare il comportamento del componente anche in situazioni stressanti.

Per quanto riguarda il testing strutturale, abbiamo creato diversi test bench che vanno a stimolare le situazioni "critiche" e i casi limite del componente in esame.

1. `DELTA MINIMO` o `MONOCOLORE`: il test verifica la corretta esecuzione del componente in caso di immagine in ingresso con tutti i pixel aventi stesso livello di intensità e conseguentemente un delta pari a 0.

2. `DELTA MASSIMO`: il test verifica la corretta esecuzione del componente in caso di immagine in ingresso con almeno un pixel di intensità 0 e uno di intensità 255 e conseguentemente un delta pari a 255.

3. `DIMENSIONE NULLA`: il test verifica la corretta esecuzione del componente nel caso in cui almeno una delle due dimensioni dell'immagine sia pari a 0. N.B: per gestire questo corner case è stato aggiunto lo stato `CHECKZERO`.

4. `DIMENSIONE MASSIMA`: il test verifica la corretta esecuzione del componente nel caso di immagine di dimensione 128x128 pixel. NB: per come è stato progettato il componente e a causa della dimensione della RAM in dotazione, si è in grado di elaborare immagini con una dimensione massima pari a 181x181 pixel.

5. `IMMAGINI IN SERIE`: il test verifica la corretta esecuzione del componente nel caso di più immagini in ingresso.

3.3 Ottimizzazioni

La principale ottimizzazione è stata realizzata nel calcolo del logaritmo presente nello stato `CALCOLOSHIFT`, in cui viene ricavato lo `SHIFT_LEVEL`. Infatti, grazie alla ridotta dimensione dei domini dei valori di ingresso e di uscita, è stato possibile alleggerire il carico del componente calcolando in fase di progettazione i risultati per ogni range di valori in ingresso. Inoltre abbiamo accorpato la differenza nel calcolo dello shift per rendere il tutto ancora più veloce.

Abbiamo realizzato una lookup table seguendo la seguente logica:

INTERVALLO DELTA	LOG_2(DELTA_VALUE + 1)	SHIFT
0	0	8
$1 \rightarrow 2$	1	7
$3 \rightarrow 6$	2	6
$7 \rightarrow 14$	3	5
$15 \rightarrow 30$	4	4
$31 \rightarrow 62$	5	3
$63 \rightarrow 126$	6	2
$127 \rightarrow 254$	7	1
255	8	0

Tabella 2: Rappresentazione controlli a soglia utilizzati per calcolare il logaritmo

In fase di progettazione iniziale era stata ipotizzata una FSM con un numero inferiore di stati. In seguito a delle osservazioni circa l'algoritmo adottato abbiamo optato per un aumento del numero degli stati.

L'algoritmo utilizzato infatti ha una complessità $\theta(N)$ dovuta alla necessità di scorrere l'immagine due volte e di conseguenza la fase più onerosa risulta essere la fase di lettura. Aggiungendo stati ausiliari, come ad esempio lo stato `SOTTRAZIONE`, è stato possibile ottenere una frequenza di lavoro maggiore con conseguente riduzione dei tempi di esecuzione complessivi.

4 Conclusioni

Il componente sintetizzato supera correttamente tutti i test descritti in precedenza sia in *Behavioral Simulation* che *Post-Synthesis Functional Simulation* e *Post-Synthesis Timing Simulation*.

I risultati ottenuti dai report di Timing hanno prodotto come risultato un periodo di clock minimo pari a 6.630 ns, cui corrisponde una frequenza massima di circa 150.829 MHz.

Infine si è notato che all'aumentare della dimensione e del numero di immagini si misura una crescita lineare del tempo di esecuzione (come ipotizzato in precedenza) a causa della necessità di leggere due volte l'intera immagine.