Progetto di Reti Logiche
2020/2021

Prof. Fabio Salice

Davide Osimo

Matricola

Codice Persona

POLITECNICO
MILANO 1863

Indice

Introduzione

Obiettivo

È data la dimensione di un'immagine nei primi due byte della memoria e in quelli successivi i pixel dell'immagine. Lo scopo del progetto è quello di implementare un componente hardware tramite codice VHDL che fornisce l'immagine equalizzata nei byte immediatamente seguenti l'immagine originale.

Specifiche

Sono riportate di seguito le specifiche del progetto che si distinguono in: algoritmo di equalizzazione, dati e gestione della memoria, interfaccia del componente, considerazioni generali.

Equalizzazione

Per equalizzazione si intende la ricalibrazione del contrasto di un'immagine quando l'intervallo dei valori di intensità sono molto vicini, effettuandone una distribuzione su tutto l'intervallo di intensità, al fine di incrementare il contrasto. In questo progetto l'equalizzazione si applica a immagini in scala di grigi a 256 livelli.

Figura 1: Esempi di immagine pre e post equalizzazione

L’algoritmo di equalizzazione fornito è il 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)
```

dove `MAX_PIXEL_VALUE` e `MIN_PIXEL_VALUE` sono rispettivamente il massimo e il minimo valore dei pixel dell’immagine, `CURRENT_PIXEL_VALUE` è il valore del pixel da trasformare e `NEW_PIXEL_VALUE` è il valore del nuovo pixel. `FLOOR(LOG2(DELTA_VALUE + 1))` è un numero intero con valori tra 0 e 8 facilmente ricavabile da controlli a soglia e che approssima il logaritmo in base 2 ad un intero per difetto.

Dati e gestione della memoria

Il modulo legge da una memoria l’immagine da elaborare la quale è memorizzata sequenzialmente e riga per riga.

La dimensione dell’immagine è definita dai primi 2 byte della memoria. Nel primo byte, a partire dall’indirizzo 0, si troverà il numero delle colonne. Nel secondo byte, all’indirizzo 1, avremo il numero delle righe dei pixel dell’immagine. La dimensione massima è di 128x128 pixel.

I pixel dell’immagine da elaborare vengono memorizzati a partire dal byte all’indirizzo 2 in avanti, un byte per ogni pixel. Dunque, il byte all’indirizzo 2 conterrà il primo pixel dell’immagine.

L’immagine equalizzata verrà scritta in memoria immediatamente dopo l’ultimo pixel dell’immagine originale, a partire quindi dall’indirizzo `(dim_col*dim_row) + 2`.

I byte contengono informazioni a 8 bit, quindi sia dimensioni che valore dei pixel delle due immagini saranno a 8 bit. Non ha senso scrivere la dimensione dell’immagine equalizzata perché quest’ultima e l’immagine originale hanno la stessa dimensione.

In *Figura 2* e *Figura 3* vediamo un esempio, per chiarire il concetto, di un’immagine di 3x2 pixel.

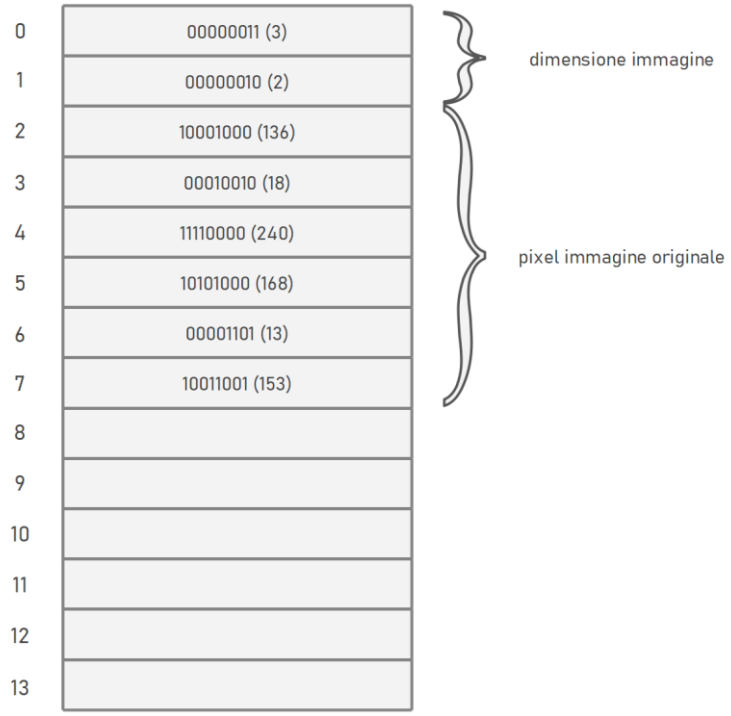


Figura 2: RAM prima dell'equalizzazione, tra parentesi i numeri decimali corrispondenti

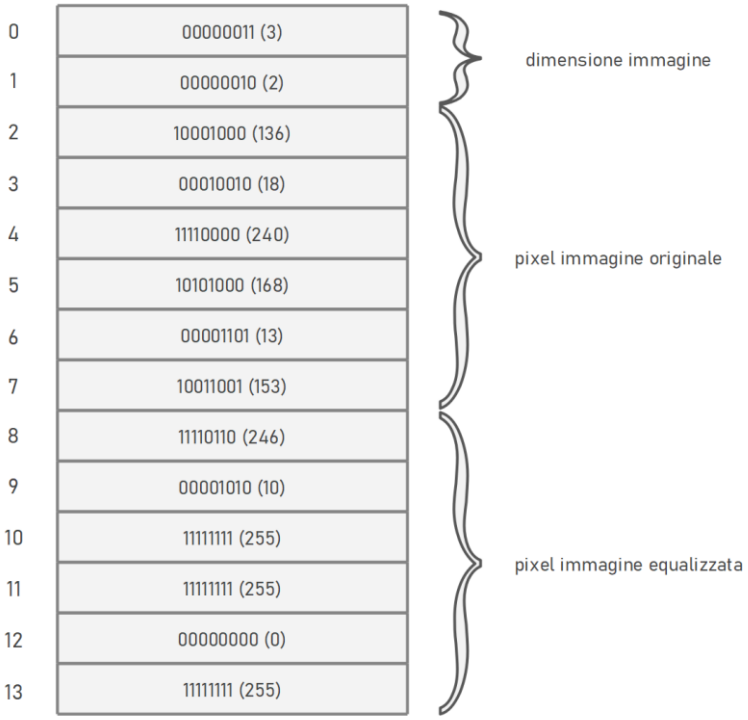


Figura 3: RAM dopo l'equalizzazione, tra parentesi i numeri decimali corrispondenti

Interfaccia del componente

Di seguito l'interfaccia del componente:

```
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;
```

Dove:

- `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 Test Bench;
- `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.

Considerazioni generali

Il modulo inizia l'elaborazione quando il segnale `i_start` viene portato a 1. Il segnale `i_start` rimane alto finché il segnale `o_done` non viene portato alto; al termine della computazione, dopo aver scritto il risultato in memoria, il modulo porta a 1 il segnale `o_done` che notifica la fine dell'elaborazione. Il segnale `o_done` rimane alto fino a che il segnale `i_start` non è riportato a 0. Un nuovo segnale `i_start` non può essere dato fin tanto che `o_done` non viene riportato a 0. Se a questo punto viene rialzato il segnale `i_start`, il modulo riparte con la fase di codifica, dato che esso è progettato per poter codificare più immagini. L'immagine da codificare, però, non viene mai cambiata all'interno della stessa esecuzione, ossia prima che il modulo abbia segnalato il completamento tramite il segnale `o_done`.

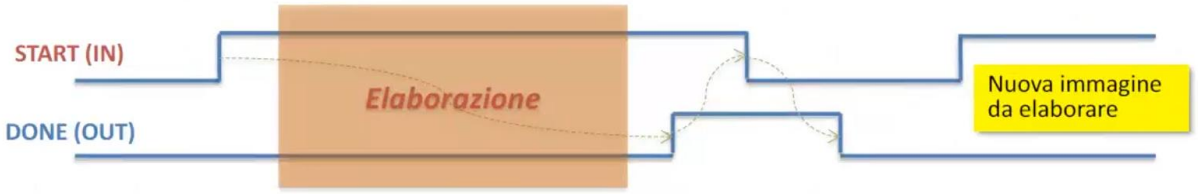


Figura 4: I segnali `i_start` e `o_done` durante la computazione

Il modulo è progettato considerando che prima della prima codifica viene sempre dato il segnale `i_rst` = '1' al modulo. Invece, come descritto nel protocollo precedente, una seconda elaborazione non attende il reset del modulo.

Architettura

L'architettura del modulo da me implementato prevede due processi distinti:

- processo di gestione dei segnali `i_rst` e `i_clk`;
- processo della Macchina a Stati Finiti (FSM).

`i_rst` e `i_clk` process

È il processo che gestisce l'inizializzazione e l'aggiornamento dei registri.

Per quanto riguarda `i_rst`, quando esso viene portato a 1, i registri vengono inizializzati. Il reset del modulo può avvenire in qualsiasi momento della computazione.

Quando, invece, il segnale `i_clk` commuta dal fronte basso a quello alto, i registri vengono aggiornati tramite la notazione `registro_curr <= registro_next` con il valore che hanno ricavato dal processo dell'FSM.

FSM process

È il processo che rappresenta la vera e propria parte algoritmica del progetto. Esso comprende 3 grandi fasi principali:

1. lettura delle dimensioni dell'immagine;
2. scorrimento di tutti i byte che contengono i pixel dell'immagine originale per poter trovare il `MAX_PIXEL_VALUE` e il `MIN_PIXEL_VALUE`
3. fase di equalizzazione che comprende la lettura ed equalizzazione dei pixel dell'immagine originale e la conseguente scrittura dei nuovi pixel sulla memoria a partire dall'indirizzo `(dim_col*dim_row) + 2`.

Il processo è descritto da una FSM a 10 stati, riportati di seguito con la propria descrizione.

START

È lo stato iniziale, ovvero quello che parte quando `i_start` è portato a 1. Se il segnale `i_rst` è portato a 1, si torna in questo stato.

ADDRESS

È lo stato in cui viene assegnato un indirizzo a `o_address` per la sola fase di lettura dei dati. Infatti,

o_en e o_we vengono portati rispettivamente a 1 e a 0 per poi poter leggere i dati dalla memoria. Inizia con gli indirizzi delle dimensioni dell'immagine e poi continua con quelli dei valori dei pixel: i segnali booleani distinguono le due fasi.

CHECK

È lo stato che verifica se siamo ancora nella fase che riguarda le dimensioni dell'immagine oppure quella che riguarda i pixel dell'immagine.

RAM

È lo stato che legge il dato all'indirizzo RAM precedentemente scelto in ADDRESS. Il dato può essere o una dimensione o il valore di un pixel. Inoltre, questo stato controlla se abbiamo un'immagine con 0 pixel: se sì, si va nello stato DONE_UP. Dato che, come detto prima, bisogna scorrere una volta tutti i pixel dell'immagine, se il booleano scrolled_ram_curr è *false* allora si va nello stato SCROLL, se è *true* si va nello stato EQUALIZE.

SCROLL

È lo stato che scorre i valori dei pixel dell'immagine originale per definire il MAX_PIXEL_VALUE e il MIN_PIXEL_VALUE.

EQUALIZE

È lo stato che equalizza il valore del pixel che si sta analizzando dell'immagine originale tramite apposito algoritmo. Inoltre, prepara l'indirizzo su cui scrivere il valore del pixel equalizzato.

WRITE

È lo stato che scrive il valore del pixel equalizzato all'indirizzo precedentemente scelto in EQUALIZE. Infatti, o_en e o_we sono posti tutti e due a 1. Se siamo giunti all'ultimo pixel dell'immagine allora il prossimo stato è DONE_UP, se no torniamo in ADDRESS per leggere il prossimo pixel.

DONE_UP

È uno stato che serve per non annullare la scrittura dell'ultimo pixel dell'immagine equalizzata e che porta il segnale o_done a 1.

WAIT_START

È uno stato intermedio per poter permettere a `i_start` di scendere a 0 e mantenere `o_done` ad 1 cosicché non scendano tutti e due a 0 nello stesso ciclo di clock, come da specifica.

DONE

È lo stato che, quando `i_start` è uguale a 0, inizializza i registri per poter permettere l'elaborazione di una nuova immagine. `o_done` va a 0 perché c'è l'assegnazione a inizio processo.

Scelte progettuali

Come già anticipato ho scelto di avere due processi distinti per clock/reset e FSM perché secondo me così la logica del funzionamento del programma è più facile da individuare e mi è risultato più comodo.

L'approccio che ho utilizzato per questo progetto è stato quello di avere il minor numero di stati possibile incorporando tra loro varie fasi. Infatti, la fase delle dimensioni dell'immagine e la fase della lettura dei pixel le ho unite negli stessi stati (`ADDRESS`, `CHECK`, `RAM`) con flag opportuni: ho eseguito delle verifiche sulle dimensioni, sul passaggio da dimensioni a pixel e sullo *scrolling* della memoria.

Per quanto riguarda lo stato `EQUALIZE`, anche qui ho unito tutti i passaggi, che l'algoritmo di equalizzazione richiedeva, in un unico stato. I vari step dell'algoritmo li ho differenziati tra loro con il segnale (vettore) `cont_curr` a 2 bit, dato che gli step sono 4.

Inoltre, ho dovuto creare gli stati `DONE_UP` e `WAIT_START` che servono per la conclusione della computazione. `DONE_UP` pone a 1 il segnale `o_done`, dato che provando a farlo nel `WRITE` non mi contava la scrittura dell'ultimo pixel dell'immagine equalizzata. Quando siamo in `WAIT_START` il segnale `i_start` è intanto sceso a 0, quindi per non far scendere a 0 sia `i_start` che `o_done` ho creato questo stato che mantiene `o_done` a 1. `o_done` scenderà a 0 nel prossimo stato.

Ho usato le seguenti librerie:

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;
use IEEE.NUMERIC_STD.ALL;
```

grazie alle quali ho potuto utilizzare la funzione `to_integer(unsigned(X))` per convertire uno `std_logic_vector` X `unsigned` in un `integer` e la funzione `std_logic_vector(to_unsigned(X, n))` per convertire un `integer` X in uno `std_logic_vector unsigned` con n bit. Ho usato anche la moltiplicazione (*) e l'elevamento a potenza (**) per lo `shift_level`.

Infine, ad inizio processo dell'FSM i segnali creati da me di tipo `registro_curr` vengono assegnati al corrispettivo `registro_next` per mantenere in modo pulito il segnale del ciclo di clock precedente se al ciclo corrente il segnale non viene cambiato. I segnali di output vengono inizializzati ogni volta che inizia il processo dell'FSM (ad ogni ciclo di clock), per avere un modo corretto con cui eseguire richieste alla RAM esclusivamente al momento giusto.

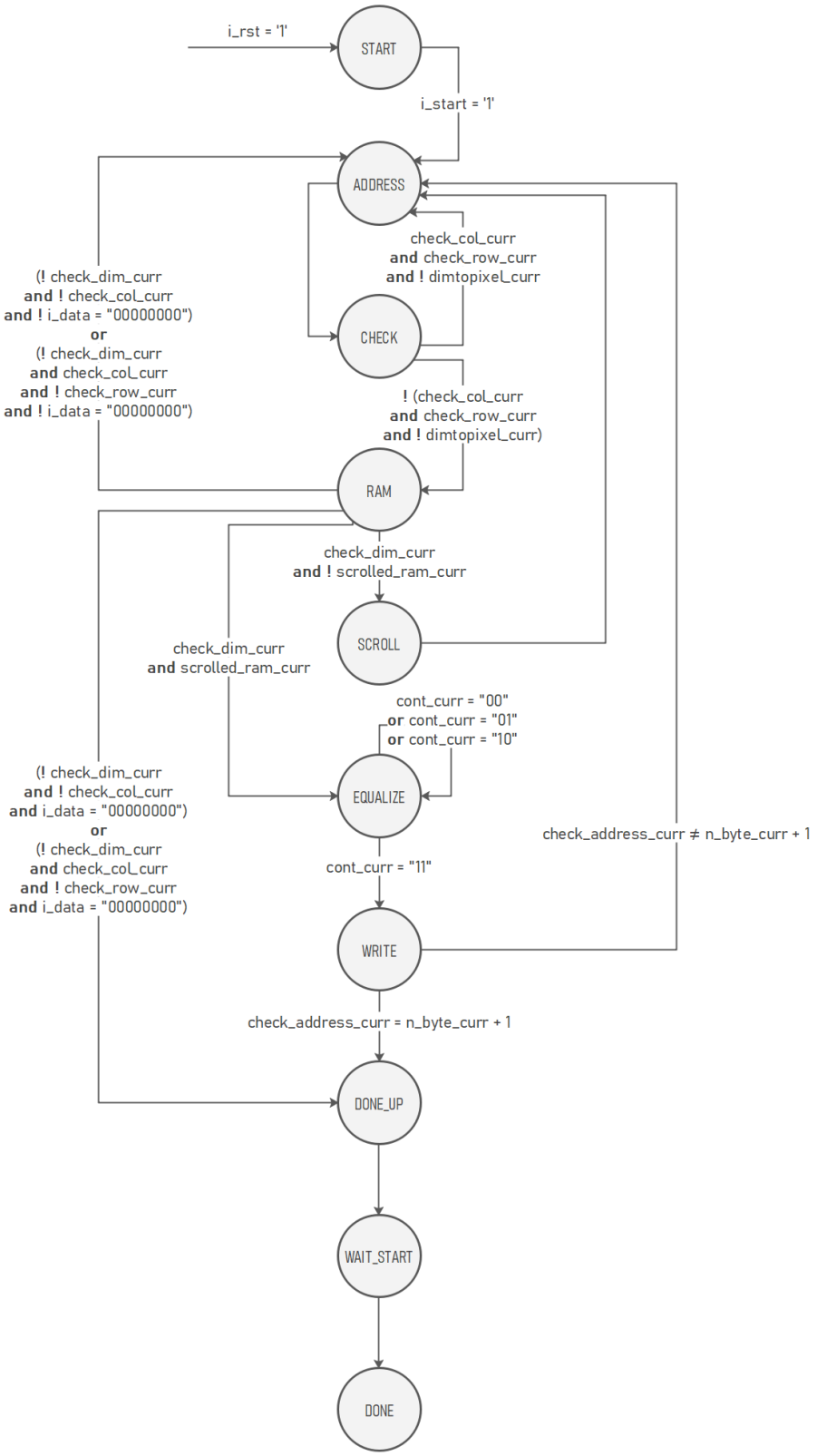
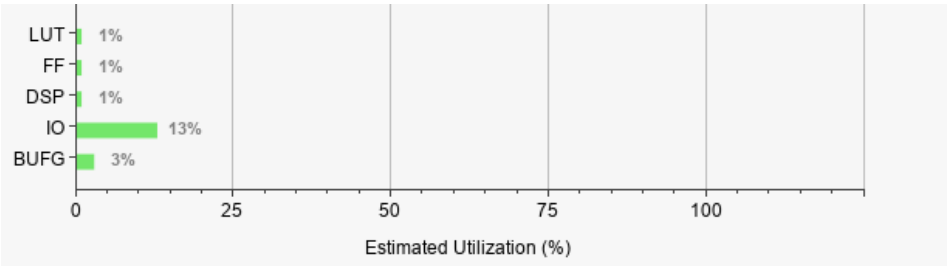


Figura 5: FSM del modulo

Risultati sperimentali: sintesi

Utilization report

Il dispositivo progettato è perfettamente sintetizzabile (ho usato una FPGA xc7k70tfbg676-2L) senza *inferred latch* e utilizza ben 240 *Look Up Table* e 176 *Flip Flop*.



Resource	Estimation	Available	Utilization %
LUT	240	41000	0.59
FF	176	82000	0.21
DSP	1	240	0.42
IO	38	300	12.67
BUFG	1	32	3.13

Figura 6: Utilization report, sintesi

Timing report

Un dato importante riguardante il tempo di esecuzione è il *Worst Negative Slack* che ci dice quanto tempo rimane al completamento del ciclo di clock nel peggiore dei *paths*. Con un ciclo di clock di 100ns abbiamo i seguenti risultati:

Setup		Hold		Pulse Width	
Worst Negative Slack (WNS):	94,869 ns	Worst Hold Slack (WHS):	0,066 ns	Worst Pulse Width Slack (WPWS):	4,650 ns
Total Negative Slack (TNS):	0,000 ns	Total Hold Slack (THS):	0,000 ns	Total Pulse Width Negative Slack (TPWS):	0,000 ns
Number of Failing Endpoints:	0	Number of Failing Endpoints:	0	Number of Failing Endpoints:	0
Total Number of Endpoints:	329	Total Number of Endpoints:	329	Total Number of Endpoints:	177

Figura 7: Timing report, sintesi, clock 100ns

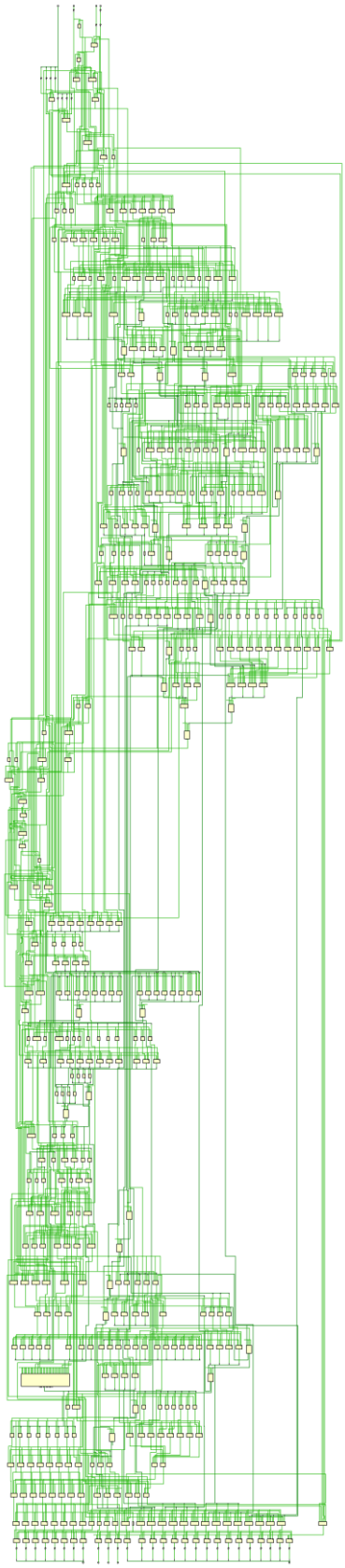


Figura 8: Schema di sintesi del componente

Simulazioni

Tutti i Test Bench forniti sono andati a buon fine superando la *Behavioral Simulation*, la *Post-Synthesis Functional Simulation* (e la *Post-Synthesis Timing Simulation*). Il modulo ha passato anche i test che ho creato attraverso il generatore. Riporto di seguito alcuni risultati di quei test che lavorano in condizioni di *borderline* e che ho scelto per questo motivo.

3 immagini

È un test che simula l'elaborazione di 3 immagini consecutivamente. Ho scelto di porlo come primo in questa relazione perché inizialmente mi aveva dato dei problemi con il segnale `o_done`: visto che `o_done` andava a 0, non mi scriveva l'ultimo pixel della prima immagine. Allora ho aggiunto lo stato `DONE_UP` per risolvere il problema.

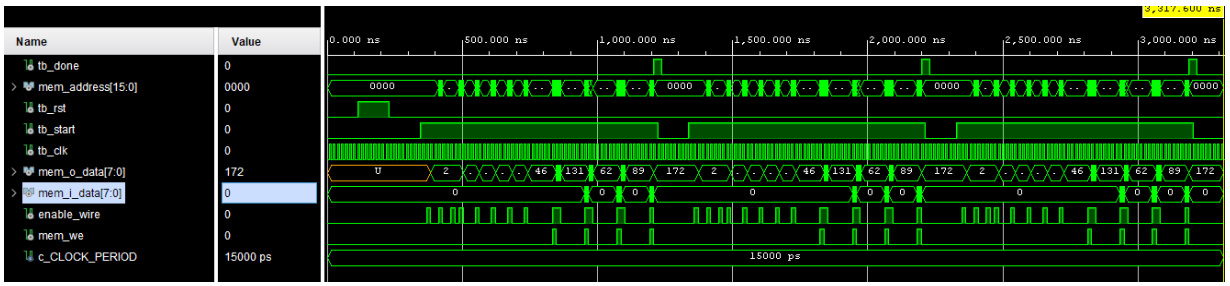


Figura 9: Waveform del Test 3 immagini

Tutti i pixel a 0

È un test che simula l'elaborazione di un'immagine con tutti i valori dei pixel a 0. L'ho scelto perché verifica una condizione limite.

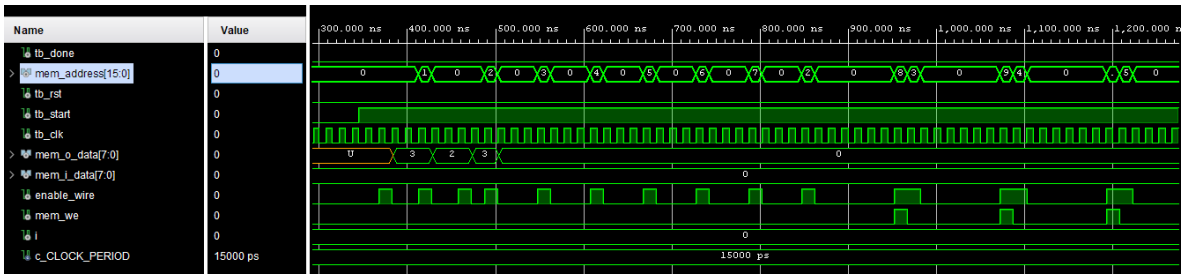


Figura 10: Waveform del Test Tutti i pixel a 0

Tutti i pixel a 255

È un test che simula l'elaborazione di un'immagine con tutti i valori dei pixel a 255. L'ho scelto perché verifica una condizione limite.

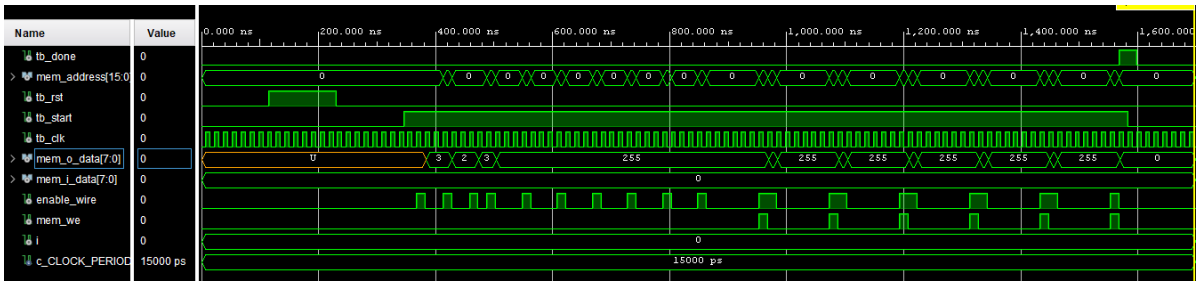


Figura 11: Waveform del Test Tutti i pixel a 255

Reset asincrono

Il test simula l'elaborazione di un'immagine con segnali di reset in momenti casuali durante la computazione. L'ho scelto per verificare il corretto funzionamento del segnale di reset. Notare come `mem_o_data` dopo il reset torna ad assumere il valore 2 che sarebbe la dimensione delle colonne e delle righe (immagine 2x2).

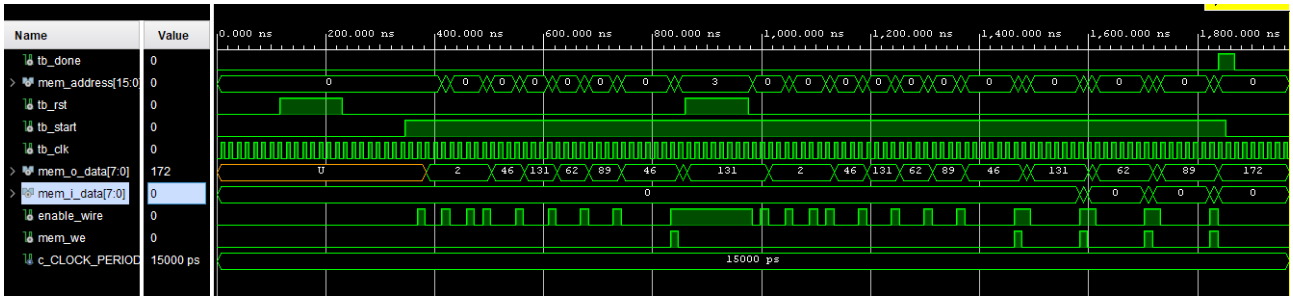


Figura 12: Waveform del Test Reset asincrono

Immagine 128x128 pixel

Il test simula l'elaborazione di un'immagine 128x128 pixel che è la dimensione massima che l'immagine può avere. Serve per verificare come il modulo si comporta con una grande quantità di

dati da elaborare. Vale la pena riportare i tempi della *Behavioral Simulation* e della *Post-Synthesis Functional Simulation* dato che è un test che impiega molto tempo per essere eseguito.

Behavioral Simulation: 2.949.737.500 ps

Post-Synthesis Functional Simulation: 2.949.737.600 ps

Un pixel

È un test che simula l'elaborazione di un'immagine 1x1 pixel. Serve per verificare il corretto comportamento del modulo.

Zero pixel

È un test che simula l'elaborazione di un'immagine che non ha nessun pixel. Serve per verificare che il modulo non scriva niente in memoria dato che la dimensione dell'immagine è 0.

0-255

Il test simula l'elaborazione di un'immagine che ha un `DELTA_VALUE` uguale a 255. I risultati di questo test daranno gli stessi valori dei pixel dell'immagine originale.

Tutti i pixel a 1

Il test simula l'elaborazione di un'immagine che ha tutti i pixel con valore uguale a 1. È un altro *corner case*.

Conclusioni

Per questo progetto è stato difficile inizialmente capire come procedere, dal codice VHDL all'implementazione vera e propria del modulo. Dopo alcuni giorni per capire il funzionamento di VIVADO e del VHDL, sono riuscito a procedere con la stesura del codice. Ho riscontrato alcuni problemi nello stabilire gli stati e le loro funzioni e molte volte mi ritrovavo ad aggiungere segnali booleani inutili per verificare certe condizioni. Superati i primi scogli, sembrava tutto regolare fino alle simulazioni coi Test Bench. All'inizio non passavo i test, dunque ho analizzato la waveform cercando di capire dove fosse il problema. I problemi sorti riguardavano errori di distrazione presenti nel codice e la non scrittura dell'ultimo pixel dell'immagine a causa del segnale `o_done`, concetto espresso più volte all'interno di questa relazione. Dopo tutto ciò sono riuscito a far funzionare il modulo correttamente. Sono soddisfatto dei risultati ottenuti, infatti il modulo svolge correttamente la funzione ad esso assegnata con il superamento della sintesi e un timing prettamente veloce.

È stata una bella esperienza svolgere questo progetto perché mi ha fatto capire più nel dettaglio come si implementano delle reti sequenziali ed il loro funzionamento con il segnale di clock. Un'altra cosa interessante è stato vedere il proprio circuito sintetizzato nell'FPGA, ho capito meglio la loro importanza e vedere il proprio progetto realmente sintetizzabile è stata una grande soddisfazione.