Politecnico di Milano
Progetto di Reti Logiche

Corso di Laurea Triennale in Ingegneria Informatica

Relazione di progetto

Equalizzatore d'immagine

Febbraio 2021

Gruppo di lavoro:

Pietro Bernardelle, pietro.bernardelle@mail.polimi.it
Codice Persona: 10618579

Stefano Civelli, stefano.civelli@mail.polimi.it
Codice Persona: 10628574

Docenti:

Prof. William Fornaciari
Prof. Federico Terraneo

Anno Accademico 2020-2021

Indice

1 Introduzione

1.1 Descrizione del progetto

Lo scopo del progetto é quello di realizzare un equalizzatore di immagini in bianco e nero, ovvero un componente che permetta di ricalibrare il contrasto al fine di incrementarlo.

Questa operazione viene effettuata modificando i valori di intensitá dei singoli pixel (i quali assumono un valore compreso tra 0 e 255 che indica la quantitá di bianco presente) per redistribuire in maniera piú equa la composizione dell'istogramma[1] (come mostrato in Figura 1).

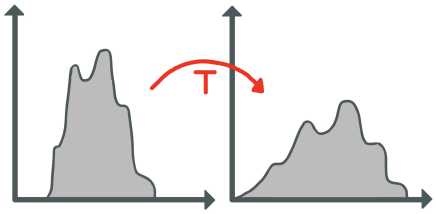

Figura 1: Trasformazione dovuta all'equalizzazione

1.2 Specifiche generali

L'immagine è letta sequenzialmente da memoria, dove si trova memorizzata come mostrato in Figura 2. Ogni pixel dell'immagine è trasformato per mezzo dell'algoritmo fornito e riscritto in memoria a partire dalla prima cella disponibile.

Un esempio di funzionamento del componente è mostrato in Figura 2 dove il blocco "Elaboratore" è quello realizzato dalla entity `"project_reti_logiche"`. Il modulo comunica alla memoria gli indirizzi da cui vuole leggere o su cui vuole scrivere (freccia nera), legge il valore dei pixel (frecce rosse), applica l'algoritmo e scrive in uscita i valori sulla memoria (frecce blu).

L'indirizzo x rappresenta il primo indirizzo di memoria nel quale si andrá a memorizzare l'immagine.

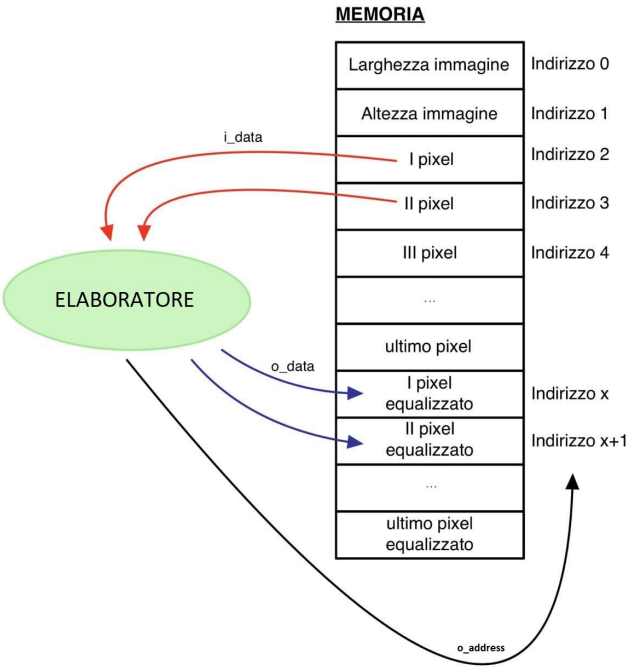

Figura 2: Funzionamento ad alto livello del componente (Elaboratore)

[1] Istogramma: rappresenta la distribuzione dei pixel chiari (sulla porzione di destra) e scuri (sulla sinistra) dell'immagine

2 Architettura

Per la progettazione del componente si è scelto di utilizzare un architettura modulare divisa in un'unità di elaborazione (Data path) e un'unità di controllo (FSM).

Da un'ottica di alto livello, l'implementazione esegue i seguenti passi:

1. carico nel primo registro il valore della prima cella di memoria
2. carica nel secondo registro il valore della seconda cella di memoria
3. calcola la dimensione dell'immagine utilizzando un blocco che conta i pixel
4. legge 1 ad 1 tutti i pixel dell'immagine e ne calcola massimo e minimo
5. calcola il delta tra massimo e minimo, il log e lo shift_level come da specifica
6. esegue lo shift e scrive a partire dal primo indirizzo di memoria libero i valori dei nuovi pixel, calcolati come da specifica calcolando MIN(255 , TEMP_PIXEL)

Per la realizzazione del modulo sono stati utilizzati registri di tipo parallelo-parallelo che aggiornano il proprio contenuto sul fronte di salita del clock. Tutti i registri sono stati definiti in VHDL con approccio behaviuoral tramite Process e hanno tutti sensitivity list (i_rst, i_clk) e, in caso di reset (asincrono), vengono resettati al valore '0'.

2.1 Datapath

In questa sezione è descritto il DataPath dividendone la trattazione in moduli logici dedicati ad eseguire specifici tasks.

1. calcolo dimensione immagine
2. gestione indirizzi
3. esecuzione algoritmo

2.1.1 Calcolo dimensione immagine

Questo modulo è composto da 2 registri, `registro_1` e `registro_2` (realizzati come specificato precedentemente) che memorizzano rispettivamente il primo e secondo Byte di memoria. A seguire sono presenti un sommatore, un sottrattore, un comparatore e un registro (`regTS`) che combinati consentono di calcolare per somme successive il numero di pixel totali dell'immagine.

A cascata è presente un ulteriore sommatore che, alla fine del conteggio, incrementa di 1 il valore presente in `regTS`[2]. Questo valore è infine salvato nel registro `reg sum_1` che viene portato in ingresso ad un comparatore e ad un multiplexer usati, come descritto di seguito, nel modulo di gestione indirizzi.

In questo modulo vengono generati 3 segnali d'uscita: `finish`, `o_end` e `CZ`. I primi 2 hanno lo scopo di gestire 2 dei casi limite (verranno approfinditi di seguito) mentre il terzo é utilizzato per notificare la conclusione del processo che permette di contare i pixel presenti nell'immagine.

2.1.2 Gestione indirizzi

La gestione degli indirizzi è realizzata mediante 2 cicli di conteggio e un multiplexer. Un ciclo é adibito al conteggio degli indirizzi di lettura (quello che in Figura 3 comprende il `registro count`), l'altro agli indirizzi di scrittura (Figura 3 ciclo che comprende `reg sum_2)`. Tramite il multiplexer si può decidere da quale dei 2 cicli attingere l'indirizzo da mandare alla memoria per mezzo del segnale `addr_sel`.

[2] `la sensitivity list`, a differenza degli altri registri, comprende anche il segnale `i_start` per permettere la reinizializzazione di questo specifico registro ogni volta che viene sottoposta al modulo una nuova immagine

Il ciclo di conteggio degli indirizzi è composto di:

- un multiplexer per l'inizializzazione dei registri
- un sommatore a 16 bit che incrementa di 1 l'indirizzo ogni volta che `rco_load`='1' e `rsu_load`='1' rispettivamente
- un registro per memorizzare l'indirizzo

Il segnale `o_end`, che consente di terminare la fase di lettura e passare a quella di elaborazione, viene alzato nel momento in cui l'indirizzo corrente diventa uguale a quello memorizzato nel registro `reg sum_1`.

Tutti i multiplexer sono stati realizzati in VHDL utilizzando il costrutto with-select.

2.1.3 Esecuzione algoritmo

Questo terzo modulo é quello adibito all'esecuzione dell'algoritmo vero e proprio.

Massimo e minimo sono implementati mediante 1 comparatore, 2 multiplexer e 1 registro. Il comparatore confronta ad ogni ciclo il valore in ingresso su i_data con quello memorizzato nel registro max/min e se i_data è rispettivamente maggiore/minore viene memorizzato nel registro corrispondente, altrimenti viene mantenuto il valore precedente. Alla fine della lettura dell'intera immagine, nei due registri (`registro max` e `registro min`) avrò memorizzato massimo e minimo, pronti per essere utilizzati per i calcoli successivi.

Vengono poi calcolati:

1. `DELTA_VALUE = MAX_PIXEL_VALUE - MIN_PIXEL_VALUE;`
2. `Y = FLOOR(LOG2(DELTA_VALUE +1));`
3. `SHIFT_LEVEL = (8 - Y);`

Questa fase si conclude calcolando per ogni pixel:

4. `TEMP_PIXEL = (CURRENT_PIXEL_VALUE - MIN_PIXEL_VALUE) « SHIFT_LEVEL;`

e assegnando il nuovo valore ad esso nel seguente modo

5. `NEW_PIXEL_VALUE = MIN( 255 , TEMP_PIXEL);`

per poi infine scriverlo in memoria.

Considerando la limitatezza del loro codominio, le funzioni di floor_log e shift ("«") sono state realizzate mediante dei process che mappano in maniera diretta il valore d'ingresso sul corrispondente valore in uscita.

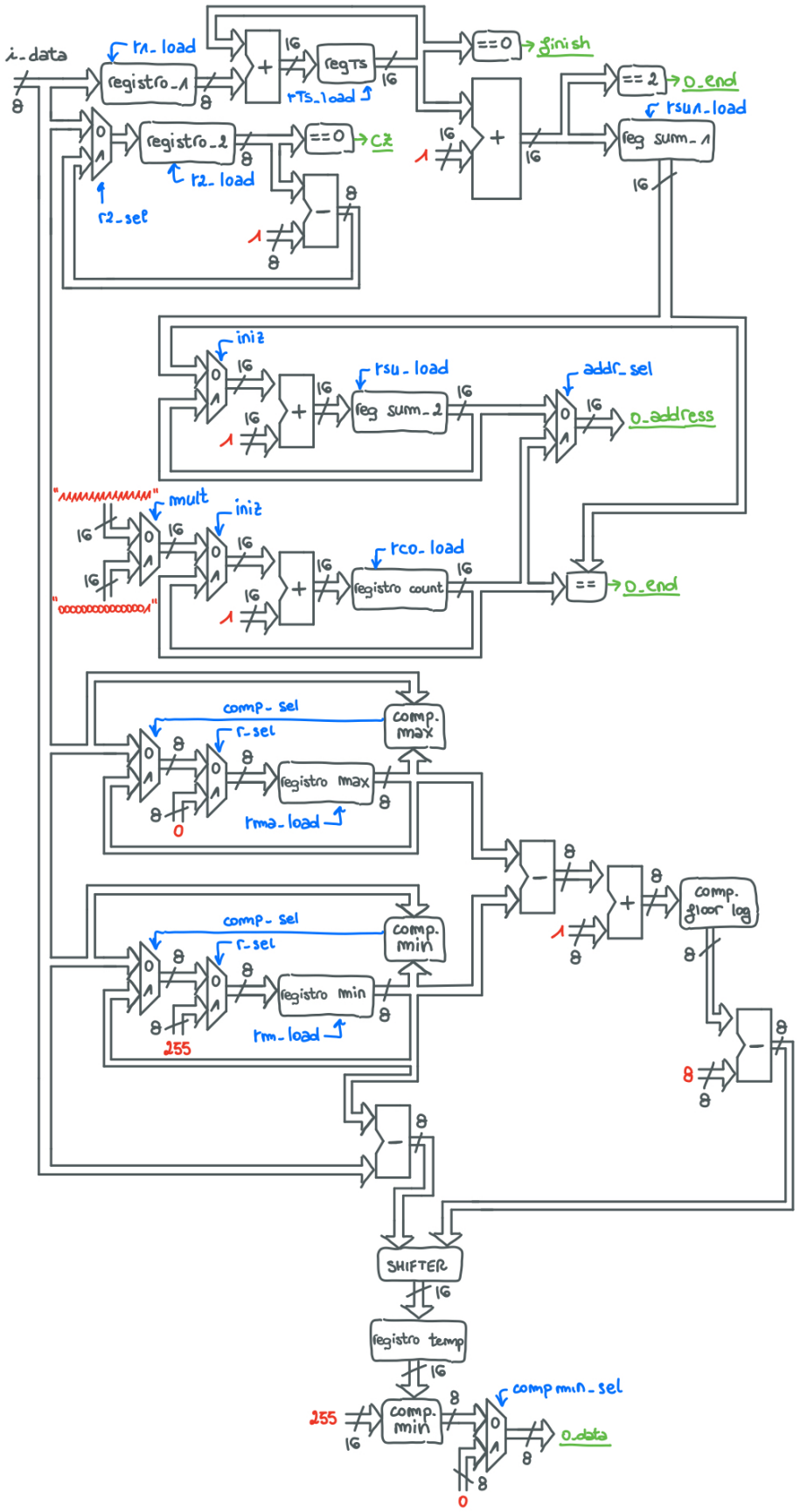

Figura 3: Datapath

2.2 Macchina a Stati

Si è scelto di utilizzare una sequenza di esecuzione separata per la gestione dei casi limite; questo per consentire il controllo di alcuni segnali parallelamente alla normale esecuzione, aumentandone la velocità nei suddetti casi.

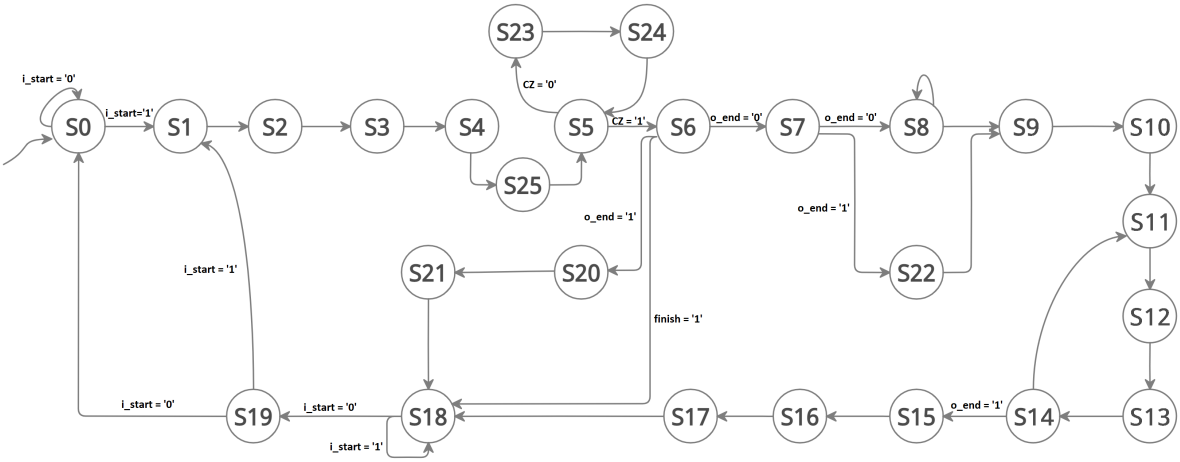

Figura 4: Macchina a stati

oss: E' stato assegnato il valore 0 come default a tutti i segnali della macchina a stati.

Di seguito é riportata una breve descrizione per ogni stato della FSM:

S0: Stato di reset. Permette di attendere che l'equalizzazione abbia inizio. Si rimane in questo stato finché `i_start` non viene alzato per passare quindi **S1**;

S1: Carica il registro `reg_count` con il primo indirizzo di memoria ponendo `rco_load`='1';

S2: Inizializza il registro `reg_max` a '0' e il registro `reg_min` a '255'. Inoltre incrementa di 1 il valore memorizzato in `reg_count` alzando il segnale `rco_load` (quest'ultimo passaggio verrá ripetuto per incrementare gli indirizzi quando necessario);

S3-S4: Consentono di immagazzinare il contenuto della prima e della seconda cella di memoria rispettivamente nei registri `reg_1` e `reg_2`. Per farlo poniamo ad '1' i seguenti segnali: `r1_load`, `r2_load`, `o_en`, `iniz`, `rco_load`;

S25: Introdotto per il corretto funzionamento del ciclo che permette di calcolare la dimensione dell'immagine. Consente al registro `reg2` di propagare il valore memorizzato nello stato precedente in uscita;

S5-S23-S24: Il ciclo composto da questi stati permette di calcolare il numero di pixel presenti nell'immagine. Viene portato avanti fino a che il contenuto del registro `reg_2` é diverso da zero, appena questa condizione viene violata il segnale `CZ` viene alzato e da **S5** si passa ad **S6**. Nel caso limite di immagini di dimensione 0 il segnale di controllo `CZ` é subito posto alto e dallo stato **S5** passo direttamente a **S6**;

S6: Lo stato piú importante dell'intera FSM; permette di individuare 2 dei 3 possibili casi limite, quello delle immagini con dimensione 0 e dimensione 1. Per prima cosa memorizza il valore dell'ultimo indirizzo di memoria da leggere in `reg sum_1`, che consentirá (come descritto nel datapath) di terminare la lettura. Avendo giá in questo stato a disposizione il valore del primo pixel in memoria, lo confronta con il valore contenuto nei registri max e min.

In caso di una immagine di dimensione 0, avendo il segnale `finish`='1', passerei direttamente nello stato **S18**.

In caso invece di un'immagine di dimensione 1, avendo `o_end`='1', percorrerei il ramo predisposto per questo caso specifico andando nello stato **S20**;

S7: Consente di individuare l'eventuale terzo caso limite(immagini di dimensione 2). In caso affermativo (se il segnale `o_end` è alto) salto direttamente allo stato **S22**;

S8: Legge ciclicamente grazie all'auto-anello tutti i pixel dell'immagine salvando il loro valore, se necessario (vedi descrizione Datapath), nei registri `reg_max` e `reg_min`. Rimango in questo stato fino a che il segnale `o_end` é tenuto basso;

S9: Terminata la lettura, in questo stato vengono inizializzati i registri che si occupano della memorizzazione degli indirizzi ai rispettivi valori iniziali. Per farlo occorre porre ad '1' i segnali `mult`, `rsu_load`, `rco_load`;

S10: Stato ponte che mi permette di attendere che sia disponibile il valore sull'uscita dei registri citati nello stato precedente;

S11-S12-S13-S14: Il ciclo composto da questi stati é quello che ci permette di alternare l'output degli indirizzi (`o_address`) usando il multiplexer comandato dal segnale `addr_sel`. Questo consente di alternare operazioni di scrittura e lettura. In caso di scrittura è fondamentale portare alti i segnali `o_we` e `o_address`.

Se nello stato **S14** `o_end` é alto, ovvero se siamo arrivati a leggere l'ultimo pixel dell'immagine, esco dal ciclo passando allo stato **S15**;

S15-S16-S17: Hanno il compito di completare elaborazione e scrittura in memoria dell'ultimo pixel. É stato necessario introdurre questi stati in quanto, per come é stato progettata la FSM, l'uscita dal ciclo sopra descritto avviene con la lettura dell'ultimo pixel dell'immagine;

S18: Pone `o_done`='1' e attende, con un auto-anello, che il segnale `i_start` venga abbassato. Quando questo accade si passa allo stato **S19**;

S19: Abbassa il segnale `o_done` e a seconda che il segnale `i_start` sia alto o basso (quindi a seconda che si voglia equalizzare subito un'altra immagine o meno) si passa rispettivamente negli stati **S1** o **S2**;

S20-S21: Sequanza di stati che gestisce il caso particolare delle immagini di dimensione 1;

S22: Permette di gestire il caso particolare delle immagini di dimensione 2.

3 Risultati Sperimentali

3.1 Sintesi

Una volta eseguita la sintesi con successo sono stati analizzati 2 principali report:

- **report_timing** - risultati discussi nella Sezione 4
- **report_utilization** - discusso di seguito

La rete ottenuta ha le seguenti caratteristiche:

- 138 Flip Flop - utilizzati per registri ad 8 e 16 bit e per memorizzare i 26 stati della macchina a stati con una codifica One-Hot
- 188 LUT
- 0 Latch - per evitare l'inferenza di Latch sono sempre stati inizializzati i segnali con un valore di default

Di seguito é riportato lo schema della rete realizzato da vivado:

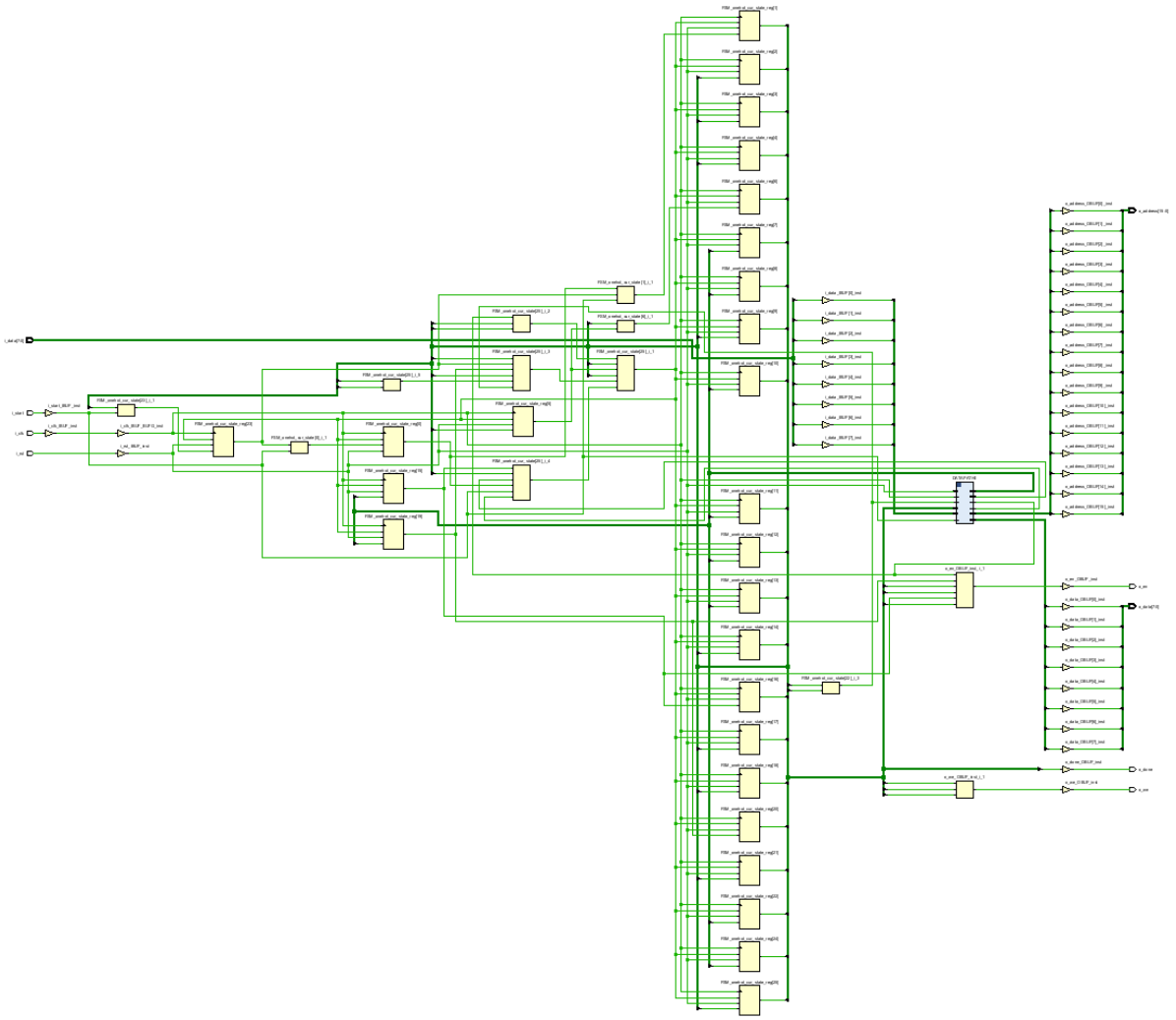

Figura 5: Schema del circuito sintetizzato

3.2 Simulazione

Per verificare il corretto funzionamento del componente realizzato sono stati utilizzati diversi *test bench* al fine di testare il comportamento, in Pre e Post sintesi, nella normale esecuzione e nei possibili casi limite.

Dopo aver appurato che il componente riuscisse a gestire semplici *test bench* è stato sottoposto a 2 test per stressarne funzionalità e comportamento:

- il primo composto da una sequenza di 10 000 immagini di dimensione massima prefissata di 16 pixel per stressare il componente sotto il punto di vista dei reset;
- il secondo composto da una sequenza di 2000 immagini di dimensione massima prefissata di 128 pixel per stressare lettura e scrittura di grandi quantitá di dati in memoria e la gestione degli indirizzi in uscita.

I casi di test appena descritti hanno anche consentito di testare i casi limite, sia all'inizio del programma che all'interno di una serie di numerose altre operazioni, per assicurarsi della loro stabilità.

Abbiamo inoltre realizzato dei TB appositi per testare in maniera più specifica i casi limite di:

- immagini di dimensione 0;
- immagini di dimensione 1;
- immagini di dimensione 2;
- immagini completamente bianche ;
- immagini completamente nere;
- immagini con pixel massimo '255' e pixel minimo '0' (con una conseguente non-necessità di equalizzare l'immagine);

che come spiegato in precedenza sono stati gestiti con un percorso di controllo alternativo della macchina a stati.

Di seguito riportiamo immagini e discussione di alcuni casi:

3.2.1 Immagini di dimensione 0

Nell'immagine (Figura 6) si può osservare che, come ci si aspetta, il segnale `finish` (che assume valore 0 solo nel momento in cui l'uscita di `registro_TS` risulta essere != 0) rimane costante ad 1. Questo consente di percorrere, partendo da S6, la sequenza di stati appositamente pensata per questo caso specifico.

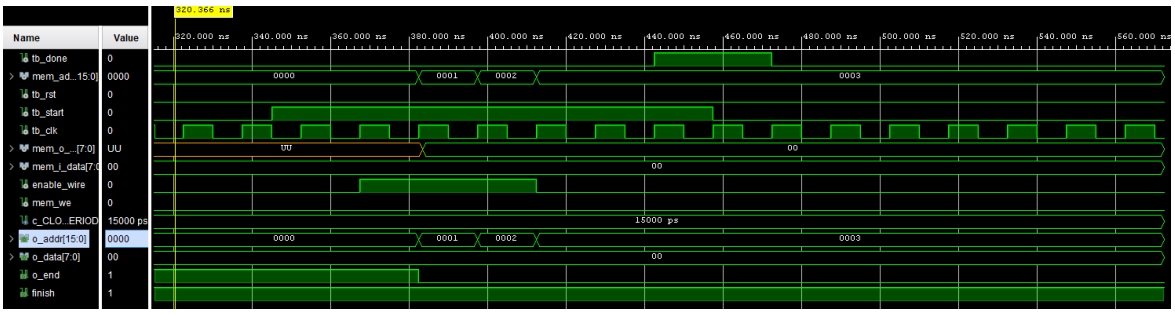

Figura 6: Screenshot caso particolare 1

3.2.2 Immagini di dimensione 1

Nell'immagine (Figura 7) si può osservare che, appena dopo aver letto i primi 2 indirizzi di memoria e calcolato il numero di pixel, il segnale `o_end` rimane alto (per via del comparatore che controlla che l'uscita di `regTS+1` sia uguale a 2) consentendo anche in questo caso di seguire il percorso di controllo dedicato.

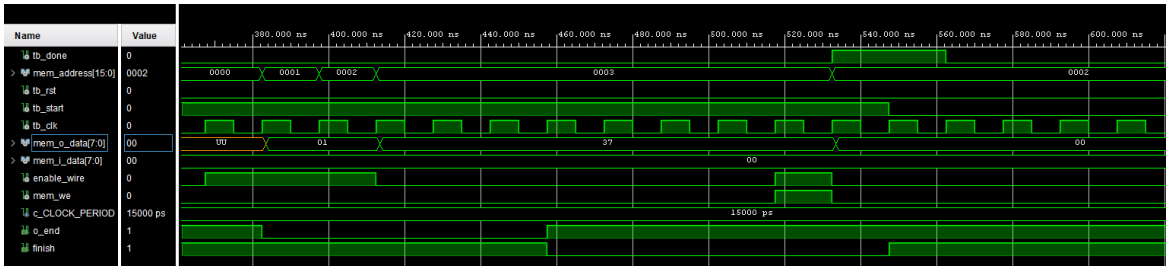

Figura 7: Screenshot caso particolare 2

3.2.3 Immagini di dimensione 2

In questo ultimo caso (Figura 8) é possibile notare che il segnale `o_end` viene portato alto prima rispetto al caso standard[3] in modo tale da evitare la lettura delle celle di memoria immediatamente successive (che sono le prime a non contenere l'immagine) per poi riabbassarlo e collegarsi al flow di controllo standard.

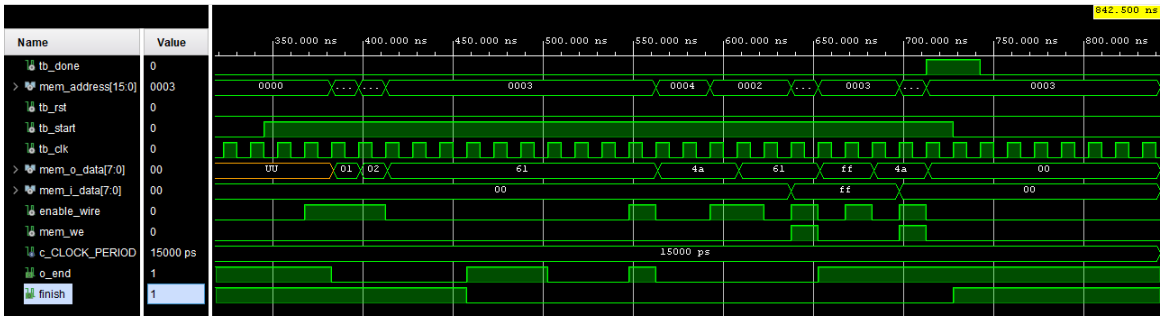

Figura 8: Screenshot caso particolare 3

[3] `o_end` viene posto a 1 almeno un ciclo di CK dopo la lettura del secondo pixel

4 Conclusioni

É stato osservato che il componente si comporta in maniera corretta dopo averlo sottoposto ad una notevole quantitá di casi di test, sia in simulazioni Pre-Synthesis che in Functional-Post-Synthesis. Possiamo quindi supporre con buon livello di confidenza che il componente è esente da errori di natura progettuale.

Abbiamo inoltre analizzato il report temporale prodotto dalla Post-Synthesis osservando i seguenti risultati:

- Data Path Delay: 5.304 [ns] (37.406% Logico e 62.594% di Routing)
- Requirement: 100.000 [ns]
- Slack (MET): 94.545 [ns]

Dai dati sopra riportati ci é possibile dedurre che il componente potrebbe funzionare ad una frequenza decisamente maggiore rispetto a quella imposta in quanto lo Slack compone la quasi totalitá del ciclo di clock. É stato possibile ottenere questo risultato optando per una soluzione con un maggior numero di registri; questo ha permesso di suddividere le operazioni su più cicli di clock, consentendo dunque di ridurre il tempo di lavoro nel singolo ciclo del componente.