Progetto finale di Reti Logiche

Pablo Giaccaglia

1 settembre 2021

Matricola: 913542
Codice Persona: 10626428
Docente: Fabio Salice

Indice

1 Introduzione

1.1 Requisiti del progetto

Il progetto richiesto consiste nell'implementazione in VHDL del metodo di equalizzazione dell'istogramma di un'immagine. Il metodo rientra nella famiglia di operazioni su immagini digitali classificate come **"operazioni globali"**. Queste operazioni, con il fine di trasformare un' immagine A in una nuova immagine B , alterano i pixel originali in modo tale che il valore di uscita nel punto (i,j) dipenda da tutti i valori dell'immagine di ingresso. In questo contesto l'equalizzazione mira a modificare la forma dell'istogramma, redistribuendo i valori dei livelli di grigio su tutto l'intervallo di intensità, in modo che sia quanto più uniforme possibile. Questa trasformazione implica un aumento del contrasto. L'obiettivo è quello di migliorare immagini a basso contrasto, quindi con un intervallo dei valori di intensità molto vicini.

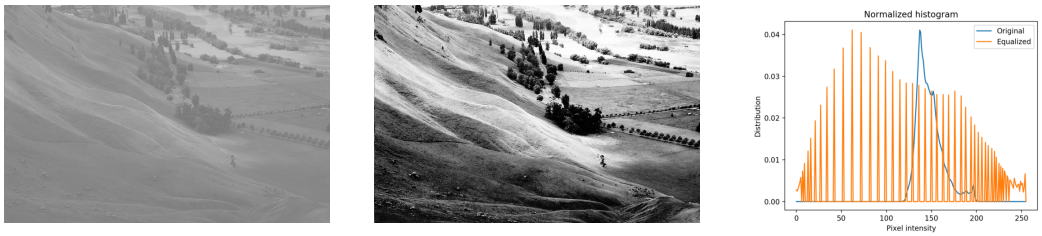


Figura 1: Esempio di immagine pre e post equalizzazione

1.2 Specifiche generali

Come da specifica funzionale del progetto, non è richiesta l'implementazione dell'algoritmo standard, bensì una sua versione semplificata. L'algoritmo di equalizzazione è applicato solo ad immagini in scala di grigi a 256 livelli e trasforma ogni suo pixel nel modo seguente :

Algorithm 1 Algoritmo di equalizzazione

procedure EQUALIZZAPIXEL

```
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 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 nuovo valore del pixel.

1.3 Interfaccia del componente

Il componente da descrivere ha un'interfaccia così definita :

```
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
```

In particolare:

- `i_clk` è il segnale di `CLOCK` in ingresso 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
- `o_we` è il segnale di `WRITE_ENABLE` da dover mandare alla memoria (=1) per poter scriverci. Per leggere la memoria deve valere 0.
- `o_data` è il segnale (vettore) di uscita del componente verso la memoria.

1.4 Dati e descrizione memoria

I dati, ciascuno di dimensione 8 bit, sono memorizzati in una memoria RAM con indirizzamento al byte. In particolare :

- L'indirizzo 0 è usato per memorizzare il numero di colonne;
- L'indirizzo 1 è usato per memorizzare il numero di righe;
- I pixel dell'immagine sono memorizzati a partire dall'indirizzo 2 e in byte contigui;
- I pixel dell'immagine equalizzata sono memorizzati in memoria a partire dall'indirizzo `2+(N_COLONNE*N_RIGHE)`
- La dimensione massima dell'immagine è 128x128 pixel;
- Ciascun pixel assume valori tra 0 e 255 (256 livelli di grigio).

2	2	46	131	62	89	0	255	64	172

Tabella 1: Rappresentazione della memoria a fine elaborazione per un'immagine 2x2

1.5 Ipotesi progettuali

a. La memoria è già istanziata all'interno dei Test Bench e non va sintetizzata;

b. Il componente funziona considerando che prima della prima codifica verrà sempre dato il reset al modulo;

c. Il componente è progettato per poter codificare più immagini;

d. Si assume che i contenuti delle celle della RAM non cambino durante l'elaborazione, finché non si resetta il componente.

e. Nel caso di reset a metà elaborazione, il componente si riporta nelle condizioni iniziali e il segnale `i_start` si abbassa. Senza il segnale di reset `i_start` non si abbassa mai durante l'elaborazione;

f. Una seconda elaborazione non deve attendere il reset del componente, infatti a fine elaborazione viene alzato il segnale `o_done`, si attende che il segnale `i_start` venga abbassato, infine si abbassa `o_done`. Una nuova elaborazione può avvenire seguendo il medesimo protocollo.

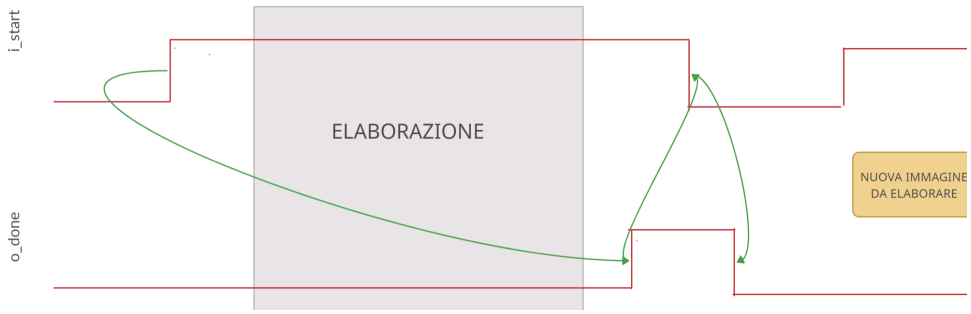


Figura 2: Protocollo di inizio e fine elaborazione.

2 Design

L'architettura è stata progettata in maniera modulare, in modo da specializzare i singoli componenti creati e separare le diverse funzionalità per il calcolo dei nuovi valori dei pixel dalla gestione della macchina a stati finiti. Nello specifico è stata adoperata una FSMD (*macchina a stati finiti con datapath*), in modo tale da delegare alla macchina a stati solamente il controllo del protocollo di comunicazione del circuito verso l'esterno e il coordinamento delle varie componenti.

2.1 Funzionamento in sintesi

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

a. Reset e attesa del segnale di start.

b. Inizializzazione degli output del componente mandando la richiesta di lettura del numero di colonne alla RAM.

c. Lettura e salvataggio del dato richiesto, richiesta di lettura del numero di righe alla RAM.

d. Controllo dei vincoli sulla dimensione dell'immagine: se almeno uno dei due valori letti è pari a 0 si termina l'elaborazione, in attesa che il segnale `i_start` venga abbassato. Se i vincoli sono rispettati si calcola il numero totale di pixel dell'immagine (`numero_righe*numero_colonne`), oltre che il primo e l'ultimo indirizzo di scrittura in RAM.

e. Calcolo del `MAX_PIXEL_VALUE` e del `MIN_PUXEL_VALUE` leggendo la memoria dalla cella 2 alla cella (`numero_righe*numero_colonne`) + 1.

f. Calcolo del `DELTA_VALUE` con i dati precedentemente individuati.

g. Calcolo del `MAX_SHIFT_LEVEL` con l'ausilio di una LUT per il calcolo del $\log_2$. Questo procedimento verrà illustrato nella sezione dedicata al componente d'interesse (sezione 2.4).

h. Lettura della memoria dalla cella 2 alla cella (`numero_righe*numero_colonne`) + 1, calcolo del `NEW_PIXEL_VALUE` e scrittura in memoria all'indirizzo i + (`numero_righe*numero_colonne`).

i. Segnalazione di fine elaborazione usando il segnale `o_done`. Attesa del segnale `i_start = 0` che permetterà di tornare allo step a per una possibile successiva elaborazione.

Per gestire questo algoritmo si è scelta un'implementazione costituita da 5 tipi di entità: la macchina a stati finiti, un `Pixel Count Module`, un `Max & Min Module`, un `Shift Level Module` e un `New Pixel Module`. Segue una descrizione dettagliata di tutte le entità realizzate.

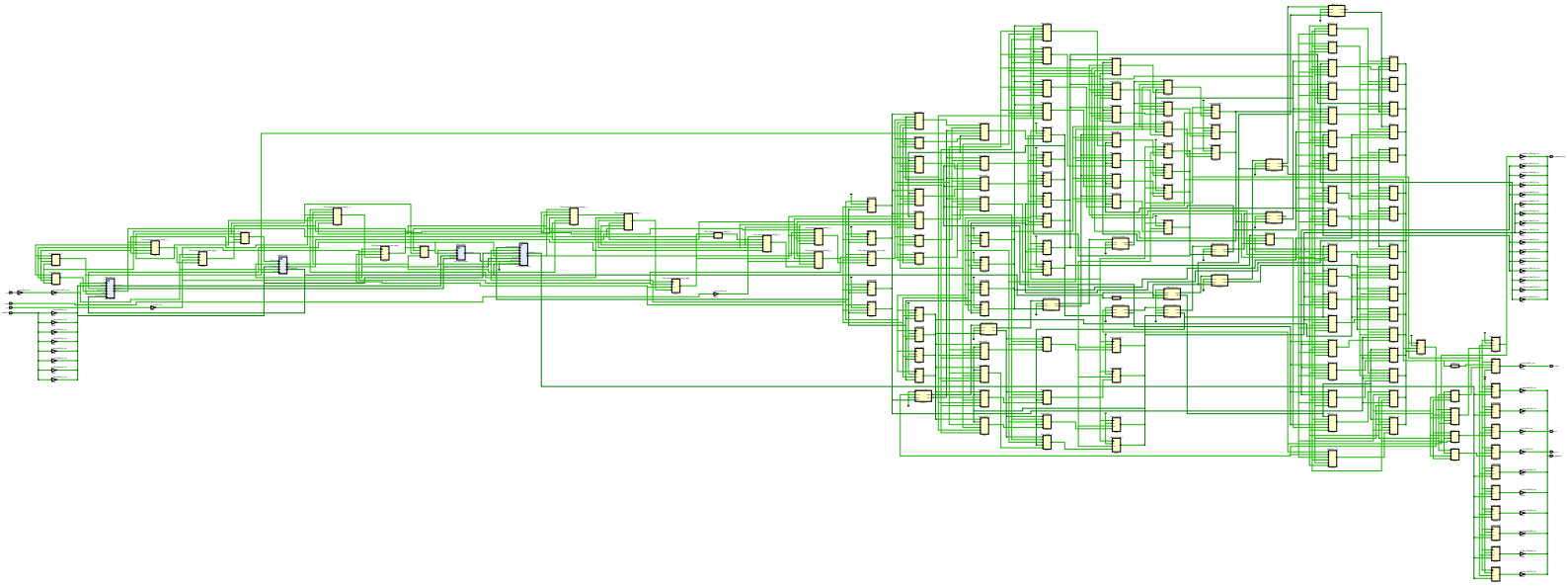


Figura 3: Schema funzionale del modulo

2.2 Pixel Count Module

Come suggerisce il nome, questo componente combinatorio, realizzato da 2 process secondo un approccio behavioral, è atto al calcolo del numero totale di pixel occupati dall'immagine in RAM, necessario per poter svolgere le successive operazioni sui dati in ingresso.

Ingressi :

`i_clk`	segnale di clock
`i_rst`	segnale di reset interno che inizializza il modulo
`i_data`	segnale proveniente dalla memoria contenente il numero di righe o colonne
`rows_load`	segnale di enable inviato dalla FSM al momento del salvataggio del numero di righe
`columns_load`	segnale di enable inviato dalla FSM al momento del salvataggio del numero di colonne

Uscite :

`total_pixels`	segnale inviato alla FSM contenente il numero totale di pixel occupati dall'immagine in RAM
`o_end`	segnale posto a 1 terminata la computazione del modulo
`idle`	segnale posto a 1 nel caso in cui le dimensioni dell'immagine non siano valide

Come approfondito nel paragrafo 2.6, questo modulo interessa tre stati della FSM : uno per il salvataggio del numero di colonne, uno per il salvataggio del numero di righe e infine uno stato per l'effettivo calcolo del numero totale di pixel, eseguito con una semplice moltiplicazione, operazione solitamente molto onerosa al crescere della dimensione degli operandi in termini di risorse hardware, ma in questo caso tollerabile per due motivi : l'operazione riguarda due segnali da 8 bit e quindi il tool di sintesi ha impegato un DSP slice, inoltre, rispetto alla FPGA scelta, le dimensioni del modulo implementato sono minime in termini di Flip-Flop e LUT.

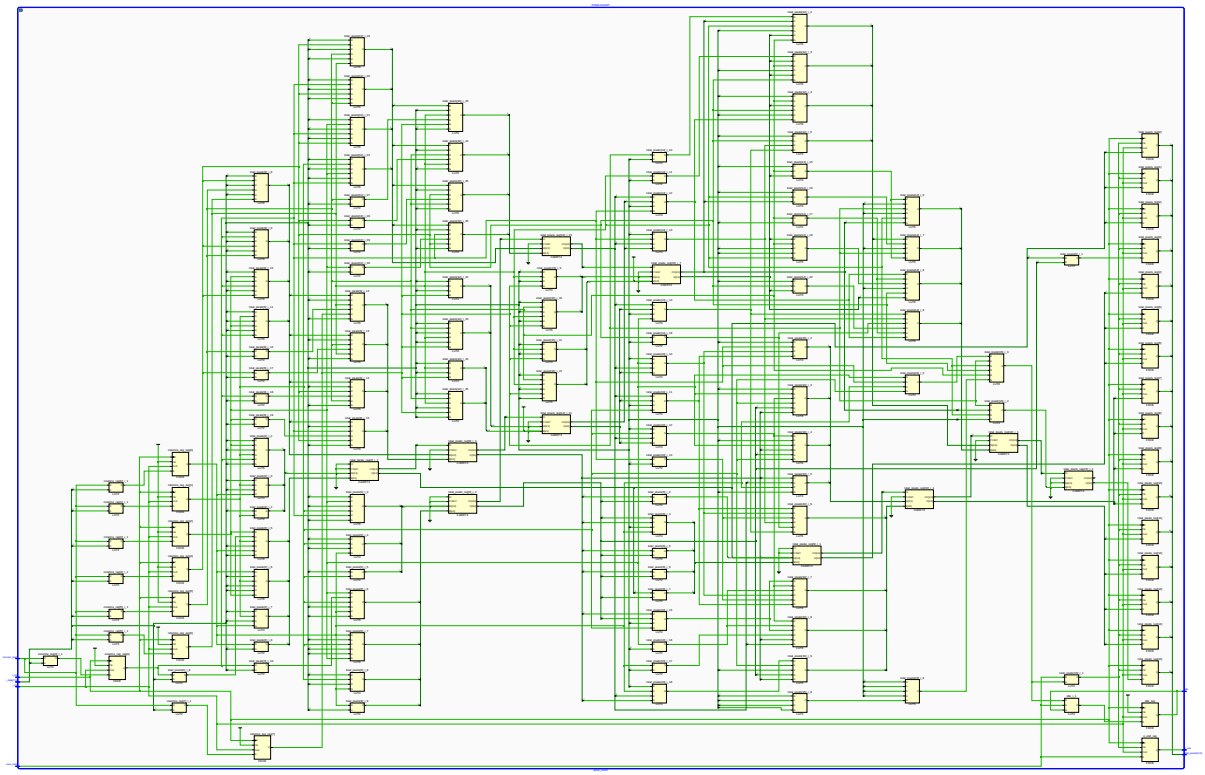


Figura 4: Schema funzionale del modulo

2.3 Max & Min Module

Questo modulo sequenziale adempie al compito di calcolare i valori del massimo e del minimo pixel tra quelli costituenti l'immagine in RAM, occupando due stati della FSM : uno per la richiesta del dato in memoria e uno per la lettura e i confronti necessari per aggiornare eventualmente i valori.

Ingressi :

`i_clk`	segnale di clock
`i_rst`	segnale di reset interno che inizializza il modulo
`i_data`	segnale proveniente dalla memoria contenente il pixel da confrontare
`max_min_load`	segnale di enable inviato dalla FSM al momento del salvataggio del pixel per il confronto
`rc_loader`	segnale di enable inviato dalla FSM al momento del salvataggio del numero totale di pixel
`total_pixels`	segnale inviato dalla FSM contenente il numero totale di pixel dell'immagine

Uscite :

`max_px`	segnale inviato alla FSM contenente il massimo valore dei pixel dell'immagine
`min_px`	segnale inviato alla FSM contenente il minimo valore dei pixel dell'immagine
`o_end`	segnale posto a 1 terminata la computazione del modulo

Realizzato tramite un'architettura behavioral, il componente prevede 3 process per la gestione della computazione : due process sono dedicati al calcolo del massimo e del minimo valore, mentre un terzo process gestisce le uscite del modulo.

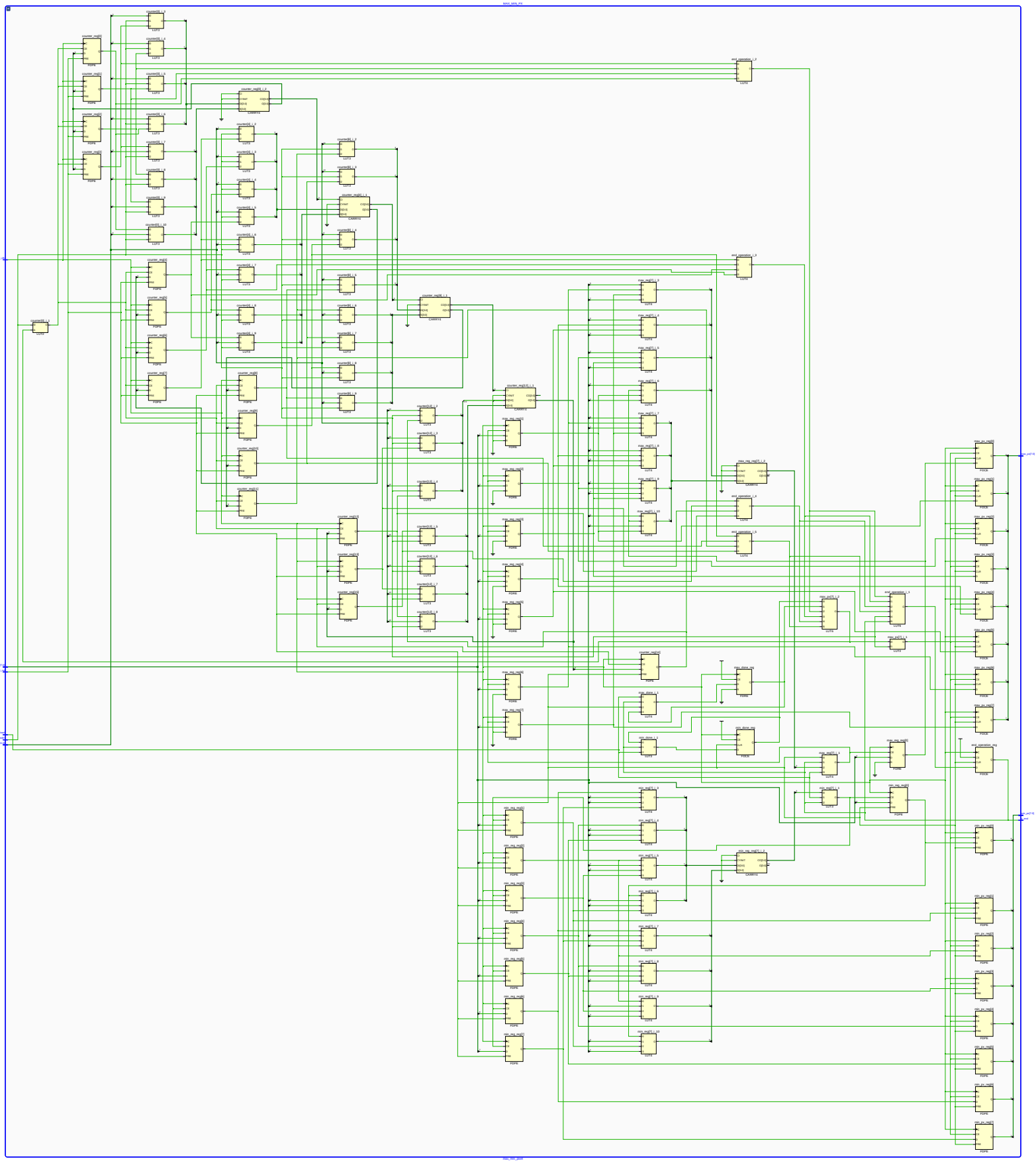


Figura 5: Schema funzionale del modulo

2.4 Shift Level Module

È un componente sequenziale realizzato tramite una semplice architettura behavioural a singolo process, attraverso il quale, impegando un solo stato della FSM, si "calcola" il valore dello `shift_level` consultando una LUT, realizzata mediante una serie di *if - elsif* a cascata. Questa scelta semplifica notevolmente il calcolo dal momento che, come riportato nel paragrafo 1.2, `8-FLOOR*`($\log_2$ `(DELTA_VALUE+1))` è un numero intero con valori tra 0 e 8 facilmente ricavabile da controlli a soglia nel modo seguente :

DELTA_VALUE	FLOOR(X)	SHIFT_LEVEL		DELTA_VALUE	FLOOR(X)	SHIFT_LEVEL
0	0	8		31	5	4
1	1	7		32	5	3
2	1	7		...	...	...
...	...	...		62	5	3
6	2	6		63	6	2
7	3	5		64	6	2
8	3	5		...	...	...
...	...	...		126	6	2
14	3	5		127	7	1
15	4	4		128	7	1
16	4	4		...	...	...
...	...	...		254	7	1
30	4	4		255	8	0

Tabella 2: Controlli a soglia

Ingressi :

`i_clk`	segnale di clock
`i_rst`	segnale di reset interno che inizializza il modulo
`max_px`	segnale inviato dalla FSM contenente il massimo valore dei pixel dell'immagine
`min_px`	segnale inviato dalla FSM contenente il minimo valore dei pixel dell'immagine

Uscite :

`shift_level`	segnale contenente il valore indicante lo shift da computare su ogni pixel dell'immagine

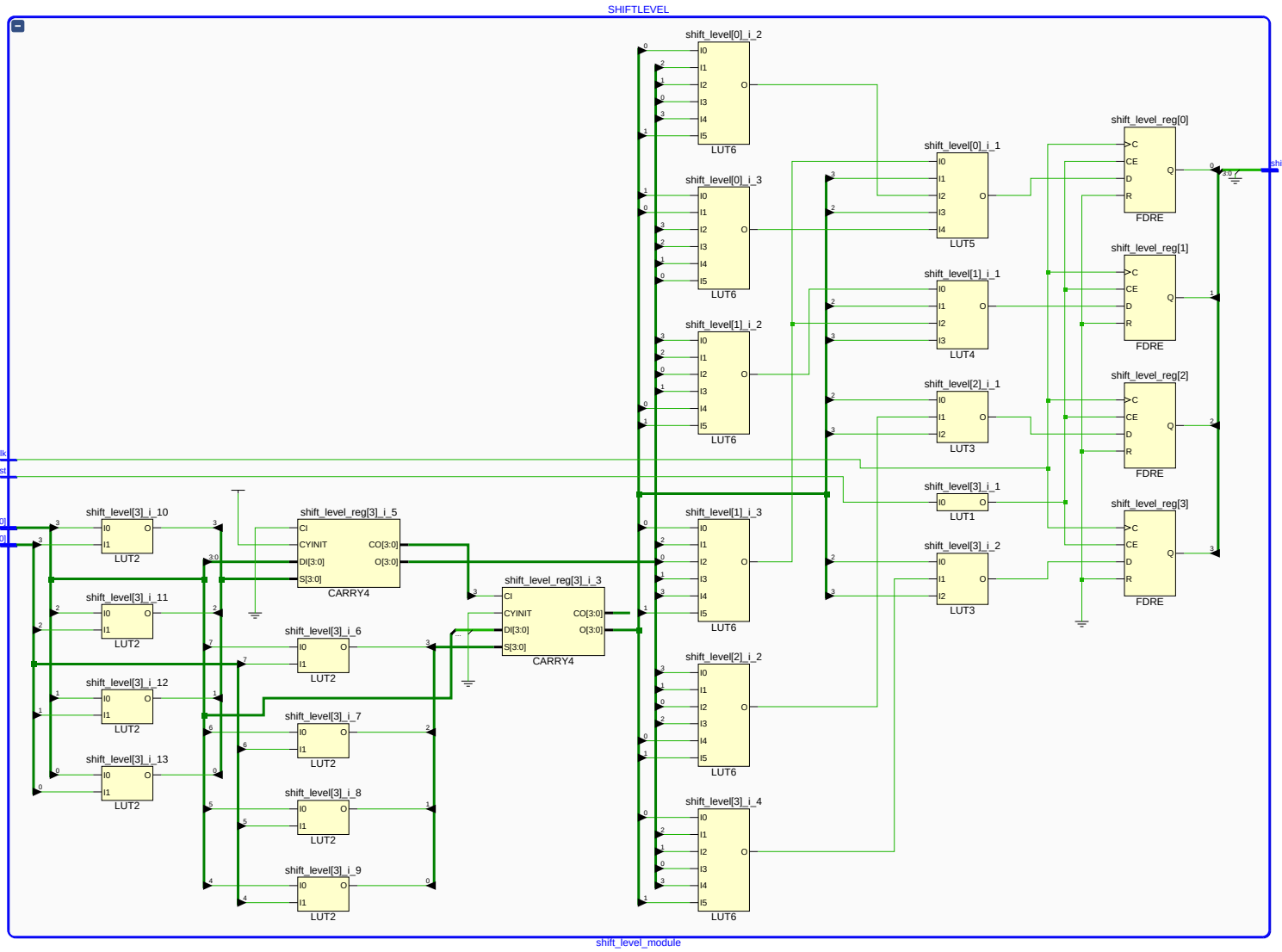


Figura 6: Schema funzionale del modulo

2.5 New Pixel Module

Componente sequenziale per il calcolo del `new_pixel_value` da scrivere in RAM, realizzato attraverso due process e un approccio behavioral. Il modulo richiede 4 stati della FSM : 2 per lettura e scrittura, uno per il calcolo del nuovo valore e uno per il calcolo dei nuovi indirizzi necessari per l'elaborazione.

Ingressi :

`i_clk`	segnale di clock
`i_rst`	segnale di reset interno che inizializza il modulo
`i_data`	segnale proveniente dalla memoria contenente il pixel da elaborare
`current_address`	segnale inviato dalla FSM contenente l'indirizzo corrente di lettura dalla RAM
`total_pixels`	segnale con il numero totale di pixel, usato per il calcolo dell'ultimo indirizzo di scrittura
`shift_level`	segnale inviato dalla FSM contenente il valore dello shift a cui sottoporre ciascun pixel
`min_px`	segnale inviato dalla FSM contenente il minimo valore dei pixel dell'immagine

Uscite :

`o_data`	segnale di uscita dal componente verso la memoria
`o_end`	segnale posto a 1 terminata la computazione del modulo, dopo che è stato elaborato l'ultimo pixel utile in RAM

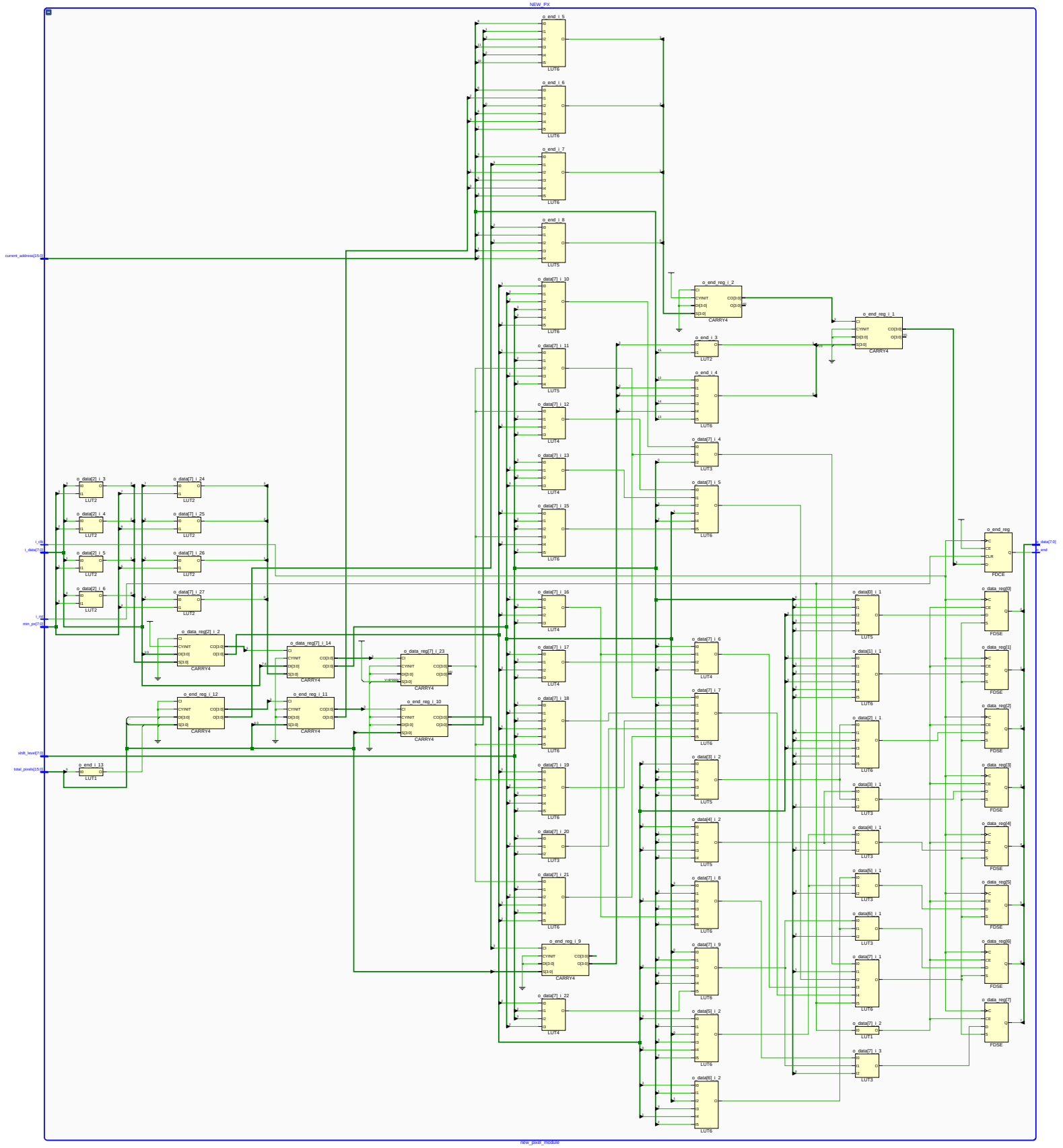


Figura 7: Schema funzionale del modulo

2.6 Macchina a Stati Finiti

La FSM è stata realizzata con specifica behavioural mediante 3 processi: `STATE_OUTPUT` , `DELTA` e `LAMBDA`. `STATE_OUTPUT` è il processo sequenziale che ha il compito di asserire le uscite e di cambiare lo stato interno sul fronte di salita del clock. `DELTA` è invece il processo combinatorio che calcola lo stato valido per il prossimo fronte di salita, mentre `LAMBDA` è il processo combinatorio che calcola le uscite per il prossimo fronte di salita del clock.

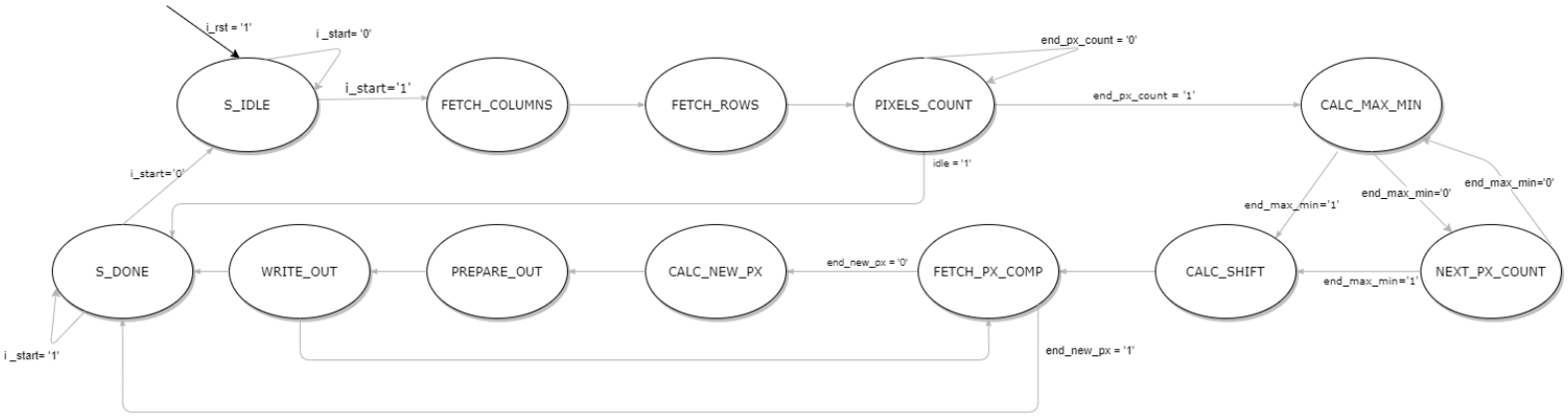


Figura 8: Macchina a stati finiti implementata

S_IDLE	Stato di idle della FSM e stato in cui si andrà in presenza del segnale i_rst ad '1'. In questo stato avviene un reset dei componenti, mediante apposito segnale dedicato. Quando viene ricevuto il segnale i_start la macchina passa allo stato FETCH_COLUMNS.
FETCH_COLUMNS	Stato in cui viene letto e salvato il dato richiesto. Richiesto alla RAM il dato corrispondente al numero di colonne.
FETCH_ROWS	Stato iniziale in cui viene richiesto alla RAM il dato corrispondente al numero di righe.
PIXELS_COUNT	Stato in cui il Pixel Count Module calcola il numero totale di pixel dell'immagine. Parallelamente la FSM richiede il valore del primo pixel utile per il calcolo del MAX_PIXEL e del MIN_PIXEL. Se la dimensione delle righe o delle colonne è pari a 0 da questo stato si passa a S_DONE.
CALC_MAX_MIN	Stato in cui calcola i valori di MAX_PIXEL e MIN_PIXEL, aggiornati ad ogni nuovo pixel letto dalla RAM. La macchina si porta in questo stato da NEXT_PX_COMP un numero di volte pari ai pixel totali, terminata la procedura la macchina si porta in CALC_SHIFT.
NEXT_PX_COMP	Stato in cui si chiede il valore del prossimo pixel utile per il calcolo del MAX_PIXEL e MIN_PIXEL. Successivamente la macchina si porta in CALC_MAX_MIN se sono presenti ancora pixel da leggere in RAM, altrimenti si porta in CALC_SHIFT.
CALC_SHIFT	Stato in cui il Shift Level Module calcola il valore dello SHIFT_LEVEL.
FETCH_PX_COMP	Stato in cui viene richiesto alla RAM il pixel di cui calcolare il nuovo valore. Nel caso in cui non siano presenti altri pixel in RAM da elaborare la macchina si porta nello stato S_DONE.
CALC_NEW_PX	Stato in cui il New Pixel Module calcola il NEW_PIXEL_VALUE.
PREPARE_OUT	Stato in cui viene richiesta alla memoria la scrittura del nuovo pixel, portando ad '1' i segnali o_en e o_we.
WRITE_OUT	Stato in cui viene scritto in memoria il NEW_PIXEL_VALUE. Successivamente la macchina si porta in FETCH_PX_COMP per una nuova elaborazione del pixel letto.
S_DONE	Stato in cui si segnala che l'elaborazione si è conclusa : o_done è portato ad '1'. Come illustrato nel portocollo al paragrafo 1.5, alla ricezione del segnale i_start uguale a '0' si riporta la macchina in S_IDLE per una possibile successiva elaborazione.

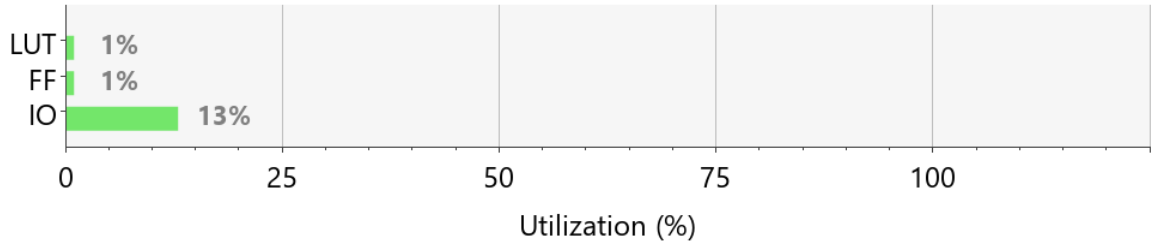
Tabella 3: Stati della FSM. I 12 stati di cui è costituita la FSM sono il risultato di diverse ottimizzazioni successive riguardanti sia le informazioni salvate che le operazioni svolte in ogni ciclo di clock. Ulteriori miglioramenti in termini di stati impegati non sono stati possibili a causa delle limitazioni imposte dalla RAM, ovvero una sola lettura o scrittura a ogni ciclo di clock.

3 Sintesi

3.1 Area occupata

Grazie al "Report Utilization", disponibile a seguito della sintesi del design, è possibile vedere l'area occupata dai vari moduli implementati.
Nel corso della progettazione non si è tenuto conto dell'area occupata in fase di ottimizzazione in quanto, come testimonia il report, i valori di utilizzo hanno svariati ordini di grandezza in meno rispetto alla disponibilità della FPGA.

Modulo	Utilizzo Look Up Table (41000 disponibili)	Utilizzo Look Up Table in %	Utilizzo Flip Flop (82000 disponibili)	Utilizzo Flip Flop in %
Pixel Count	78	0.2%	35	0.09%
Max&Min	36	0.09%	51	0.1%
Shift_Level	18	0.05%	4	0.001%
New_Pixel	43	0.1%	56	0.1%



3.2 Report di timing

Analizzando il *Design Timing Summary* è possibile vedere quanto è veloce in un singolo ciclo di clock il design sintetizzato, basandosi su un parametro che quantifica, rispetto al tempo totale a disposizione, il tempo del path peggiore, ovvero il *Worst Negative Slack*. Si è ottenuto, con il periodo di clock della specifica di 100ns, un Worst Negative Slack pari a 94,431ns. Da questo valore, sapendo anche il ritardo di risposta della RAM (T_{RAM}), possiamo calcolare il periodo minimo applicabile al design creato:

$$T_{min} = T_{curr} \text{ - WNS} + T_{RAM} = 100ns \text{ - } 94{,}431ns + 2ns = 7{,}569ns$$

Il design creato ha quindi una massima frequenza di clock pari a $f_{max} = 1/\ T_{min} \approx 132.11\ Mhz$

Design Timing Summary

Setup		Hold		Pulse Width	
Worst Negative Slack (WNS):	94,431 ns	Worst Hold Slack (WHS):	0,074 ns	Worst Pulse Width Slack (WPWS):	49,650 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:	393	Total Number of Endpoints:	393	Total Number of Endpoints:	175

All user specified timing constraints are met.

Figura 9: Design Timing Summary

3.3 Warning post synthesis

L'implementazione non riscontra né errori né warnings dopo la sintesi, essendo stati preventivamente risolti. Alcuni di questi warnings includono quelli causati dai latch inferiti e quelli causati da segnali presenti nel processo ma non inseriti nella sensitivity list.

4 Simulazioni

Una volta creato il design, esso è stato sottoposto a testing attraverso dei test bench appositi, con il fine di testare ogni funzionalità. Dopo aver testato il componente sintetizzato con il test bench di esempio, sono stati definiti altri 11 test, alcuni dei quali caratterizzati da transizioni critiche o configurazioni di memoria estreme, con l'intendo di cercare di causare errori, principalmente dei segnali di controllo.

Grazie alla fase di testing, sono emerse numerose criticità nel codice iniziale che hanno permesso variazioni e ottimizzazioni fino alla versione finale del componente.. Per tutti i test riportati e i test successivi sono state effettuate le simulazioni *Behavioural* , *Post-Synthesis Functional* e *Post-Synthesis Timing*, tutte con successo.

Di seguito è riportato un breve elenco dei test utilizzati e per i più significativi vengono riportate le forme d'onda di alcuni segnali utili, accompagnati da una breve descrizione associata.

- Immagine 2x2 (Test Bench fornito con la specifica).
- Immagine composta da un solo pixel.
- Immagine con tutti i pixel settati a 255 (immagine bianca).
- Immagine con dimensione nulla (0px x 0px)
- Immagine con numero di colonne nullo.
- Reset asincrono durante l'elaborazione.
- Reset asincrono ripetuto durante l'elaborazione.
- Tre immagini con reset intermedio.
- Test su dimensione massima: viene testato il componente con un'immagine 128x128.
- Primo test finale : 10.0000 immagini di dimensione massima 16x16px
- Secondo test finale : 2.0000 immagini di dimensione massima 128x128px

4.1 Test fornito dal docente

2	2	46	131	62	89	0	255	64	127

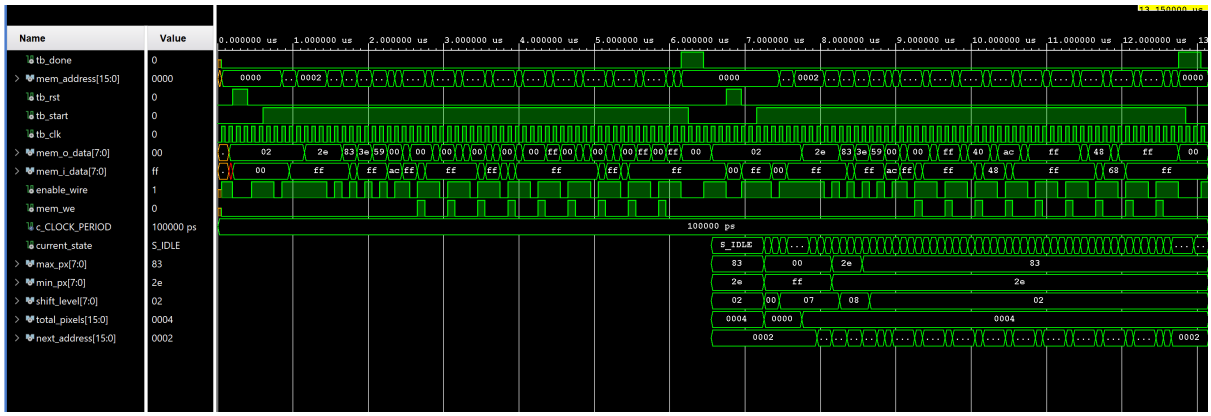


Figura 10: Waveform della Behavioural Simulation

4.2 Test Zero Pixel

Test che verifica il corretto funzionamento del componente quando la dimensione dell'immagine risulta uguale a 0.

0	2	46

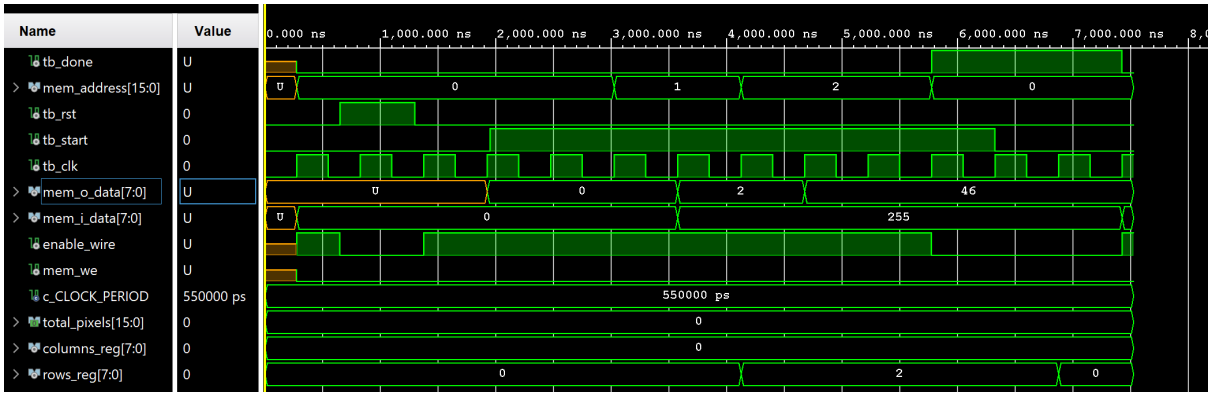


Figura 11: Waveform della Behavioural Simulation

4.3 Reset asincrono

Test che verifica il corretto funzionamento del componente quando l'elaborazione viene interrotta da un segnale di reset, seguito da una nuova elaborazione.

2	2	46	131	62	89	0	255	64	172

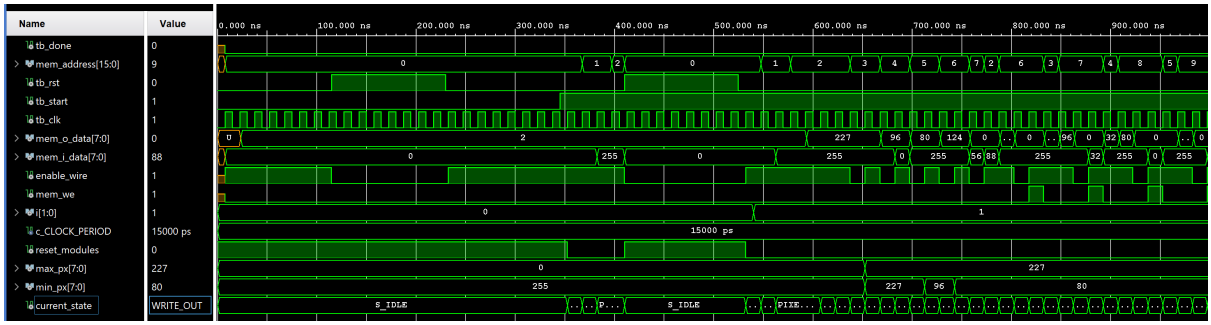


Figura 12: Waveform della Behavioural Simulation

4.4 Test Immagine bianca

Test che verifica il corretto funzionamento del componente quando l'immagine è costituita da soli pixel bianchi.

4	2	255	255	255	255	255	255	255	255	0	0	0	0	0	0	0	0

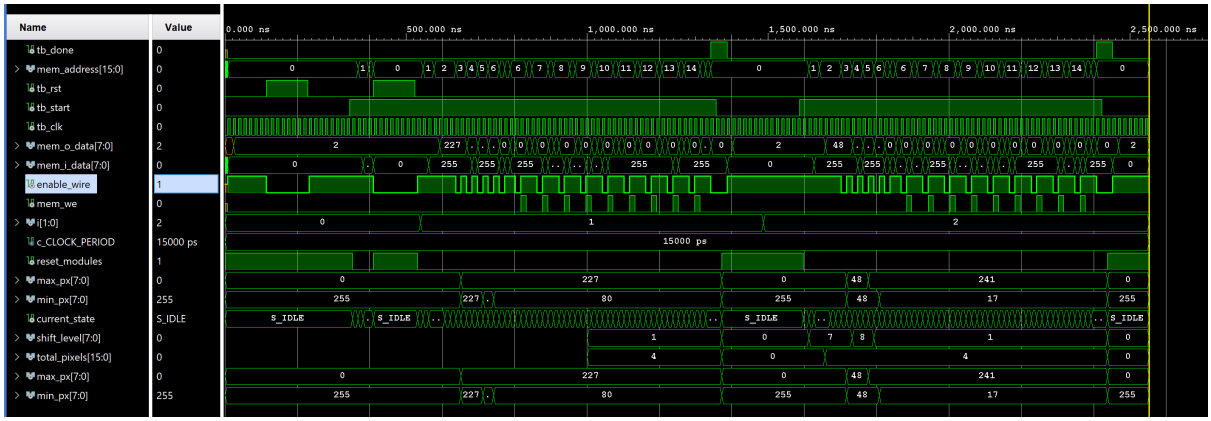


Figura 13: Waveform della Post-Synthesis Functional Simulation

4.5 Test singolo Pixel

Test che verifica il corretto funzionamento del componente quando l'immagine è costituita da un solo pixel.

1	1	34	0

4.5.1 Behavioural Simulation

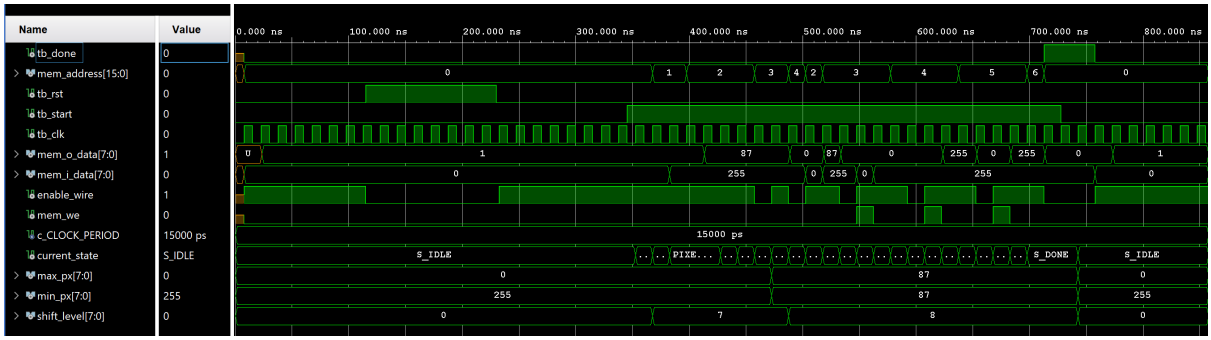


Figura 14: Waveform della Behavioural Simulation

4.5.2 Post-Synthesis Functional Simulation

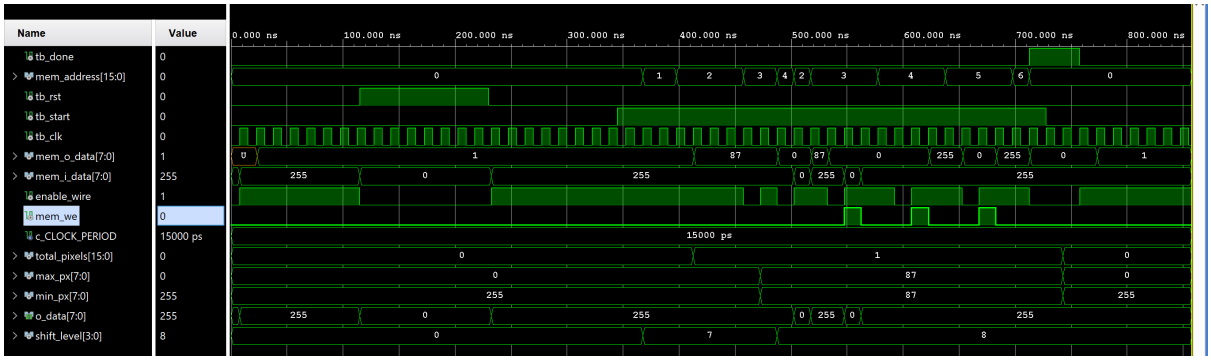


Figura 15: Waveform della Behavioural Simulation

4.5.3 Post-Synthesis Timing

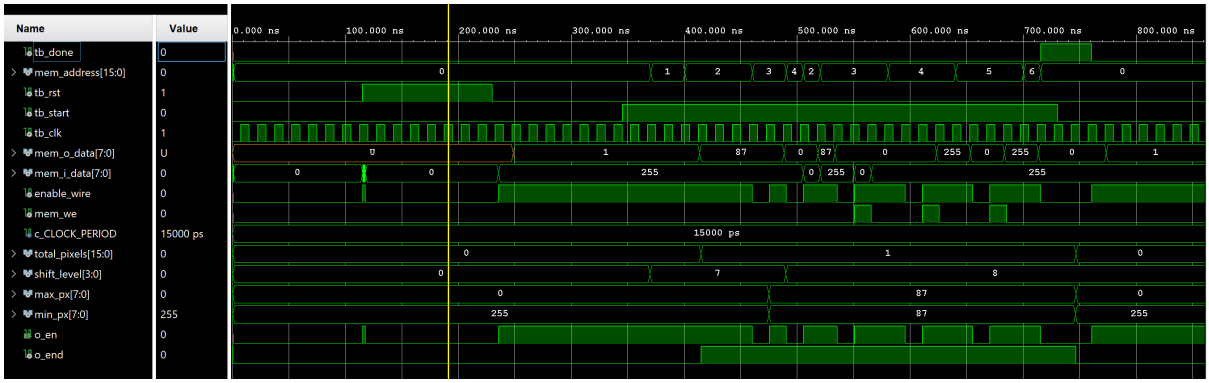


Figura 16: Waveform della Post-Synthesis Timing Simulation

5 Conclusioni

Le fasi di ottimizzazione e testing hanno permesso di produrre un'architettura che soddisfa i requisiti di progetto. L'estensivo testing mediante test benches sia casuali che manualmente scritti e l'organizzazione modulare a componenti hanno permesso di giungere a un design robusto e facilmente estendibile.