Prova finale
(Progetto di Reti Logiche)

Alessandro Barbiero
Matricola 913718
Codice persona 10692413

Anno Accademico 2020/2021

Politecnico di Milano
Professore: W. Fornaciari

SOMMARIO

1 INTRODUZIONE

1.1 OBIETTIVO

L'obiettivo del progetto è la creazione di un componente hardware il cui scopo è quello di realizzare l'equalizzazione dell'istogramma di un'immagine.

Tale procedura è atta a ricalibrare il contrasto di un'immagine effettuando una distribuzione dei valori di intensità lungo tutto l'intervallo di valori possibili.

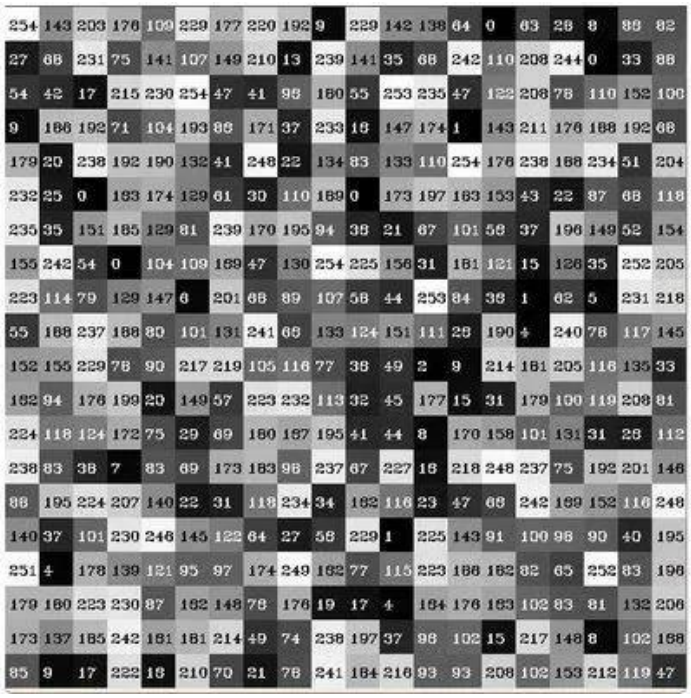

Figura 1: Intensità dei pixel di un'immagine

La figura soprariportata mostra un'immagine correttamente calibrata, che sfrutta cioè interamente l'intervallo di valori disponibile (da 0 a 255).

Nella figura seguente si può vedere l'effetto di una equalizzazione dell'istogramma dell'immagine su una foto in scala di grigi con un basso contrasto.

Figura 2: Immagine pre e post equalizzazione

1.2 Semplificazioni e Dettagli implementativi

La versione sviluppata presenta alcune semplificazioni rispetto all'algoritmo standard e segue le seguenti specifiche:

- Applicato solo ad immagini in scala di grigi a 256 livelli
- La trasformazione segue un algoritmo semplificato dove il nuovo valore del pixel viene calcolato nel modo 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)
```

- Il modulo legge l'immagine da una memoria sequenzialmente e riga per riga, ogni byte corrisponde ad un pixel dell'immagine
- Il byte all'indirizzo 0 si riferisce alla dimensione di colonna
- Il byte all'indirizzo 1 si riferisce alla dimensione di riga
- Dimensione massima dell'immagine = 128x128 pixel
- L'immagine memorizzata parte dall'indirizzo 2 e in byte contigui
- L'immagine equalizzata è scritta in memoria immediatamente dopo l'originale
- L'immagine da codificare non può essere cambiata all'interno della stessa esecuzione
- Il modulo può codificare più immagini in serie, sincronizzandosi con la memoria, sfruttando i segnali di START e DONE come illustrato nella macchina a stati finiti
- L'interfaccia utilizzata per il componente è data da specifiche

1.3 Strumentazione

Come strumento di sintesi si è utilizzato Xilinx Vivado WebPACK (versione 2020.2) e come FPGA target si è scelto la FPGA consigliata: *xc7a200tfbg484-1.*

La memoria usata e il protocollo seguito per i test bench seguono le specifiche fornite da Xilinx nella User guide per la memoria "*Single-Port Block RAM Write-First Mode",* come da specifiche.

Per il disegno di schema in moduli e FSM si è ricorso a Draw.io, strumento utilizzato per avere chiaro in mente come operi il componente. (Allego successivamente tali schemi)

2 ARCHITETTURA

2.1 SCELTE PROGETTUALI

Per la scrittura del codice si è scelto un approccio didattico, cercando di capire il funzionamento del componente, e cercando di sfruttare la concorrenza del linguaggio VHDL.

Per questa ragione si è scelto di dividere il progetto in due entity, una rappresentante il Datapath e una l'automa a stati finiti che controlla il susseguirsi dei segnali utili all'avanzamento del processo all'interno del datapath.

Si è scelto di utilizzare solamente due entity e un approccio Dataflow/Behavioural per descrivere i componenti facenti parte del Datapath invece che un approccio Structural in quanto i componenti base erano elementari e inserire più entity non avrebbe alleggerito la lettura del codice.

Per sfruttare appieno il lungo periodo di clock fornito si è scelto di lavorare contemporaneamente su entrambi i fronti del clock:

- I registri all'interno del Datapath commutano sul fronte di discesa del clock
- La FSM varia lo stato sul fronte di salita del clock

Per implementare il meccanismo di shift variabile limitato superiormente si è scelto di procedere computando contemporaneamente il valore shiftato su 8 bit e la possibilità che lo shift esuli il valore massimo di 255. Così facendo si può scegliere il valore ottimale utilizzando solo segnali a 8 bit e parallelizzando parte della computazione. Per far ciò si è fatto ricorso a un *Priority encoder* per generare il segnale di *SHIFT_LEVEL* e a un altro encoder per generare una bitmask messa successivamente in AND con la differenza tra *CURRENT_PIXEL_VALUE* e *MIN_PIXEL_VALUE* in modo da estrapolare i bit più significativi del segnale che dovrà essere shiftato. Così facendo si può capire preventivamente se occorre tenere come valore di *o_data* quello ottenuto dallo shift oppure il valore limite 255. *(Il tutto è più chiaro vedendo lo schema dei componenti del Datapath)*

Un'altra scelta implementativa è stata quella della gestione dell'avanzamento nell'immagine. Per far ciò il datapath sfrutta due contatori, uno per le righe e uno per le colonne. Il contatore per le colonne viene incrementato ad ogni lettura e resettato a 1 ogni volta che viene letto l'ultimo pixel della riga, allo stesso tempo viene incrementato quello per le righe. Così facendo è possibile lavorare pixel per pixel fermandosi al termine dell'immagine (quando il numero di righe analizzate eguaglia il numero di righe totali) senza dover utilizzare un moltiplicatore computazionalmente più pesante.

Per immagini vuote (numero di righe o di colonne = 0) il componente termina immediatamente la computazione dell'immagine corrente e attende il comando per passare all'immagine successiva.

2.2 DATAPATH

Come detto precedentemente il Datapath è stato prima pensato e disegnato su applicativo per poi essere tradotto in codice, di seguito i componenti utilizzati:

- 11 registri paralleli di cui:
  - 1 a 4 bit
  - 8 a 8 bit
  - 2 a 16 bit
- 5 multiplexer di cui:
  - 4 a due scelte (2 to 1)
  - 1 a quattro scelte (4 to 1)
- 7 sommatori/sottrattori
- 7 comparatori
- 1 shifter limitato a 8 bit
- 1 AND logico
- 2 encoder

Il reset e il clock non compaiono nello schema funzionale ma sono collegati ad ogni registro.

- Il reset porta i valori dei registri a zero, eccezion fatta per *reg_min*, resettato a 255 (accortezza non necessaria per come è sviluppato il resto del progetto ma ritenuta sensata concettualmente)
- Il clock è collegato ai registri e permette la scrittura su di essi solo lungo il fronte di discesa

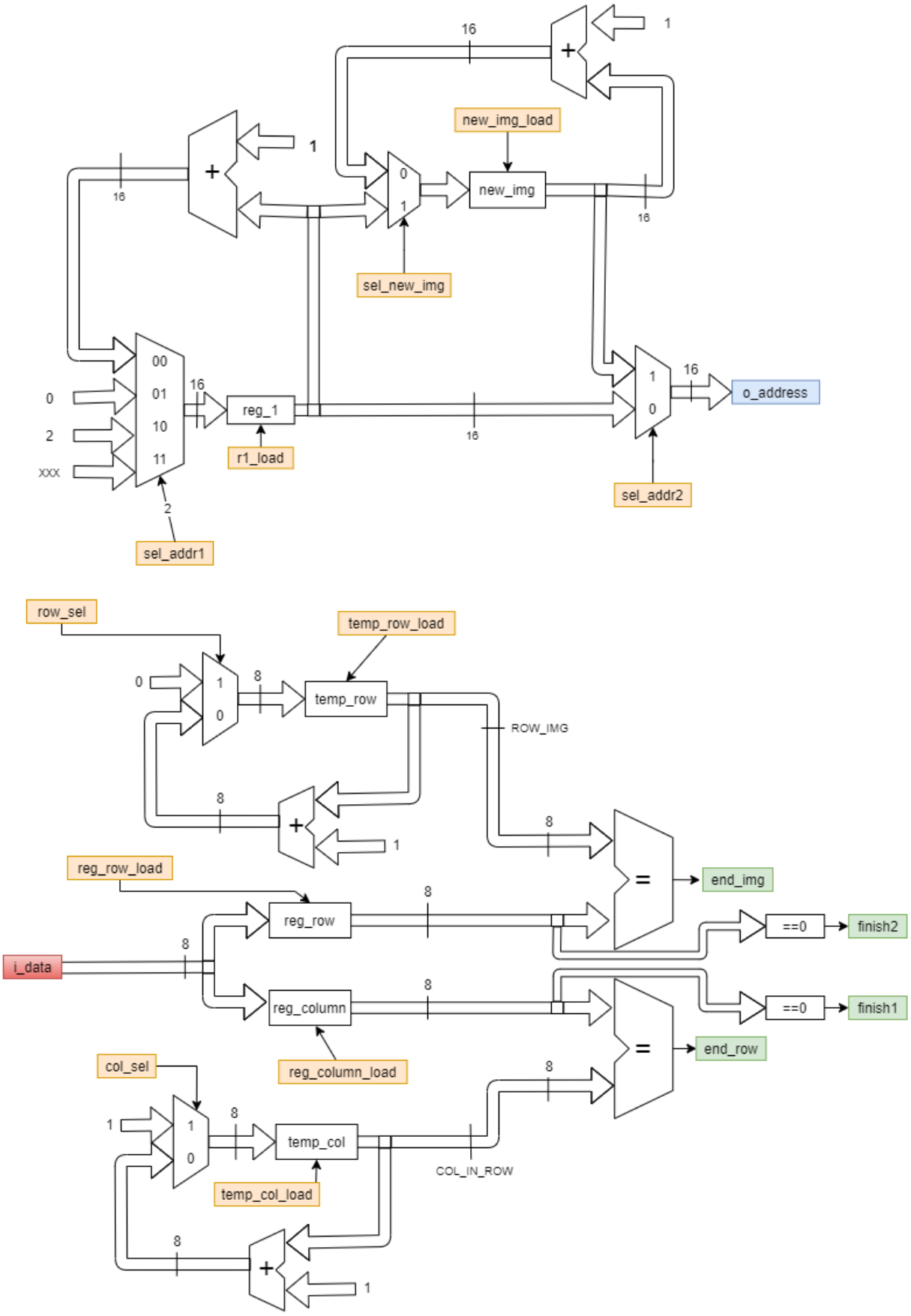

Figura 3: Datapath, prima parte (incremento indirizzi)

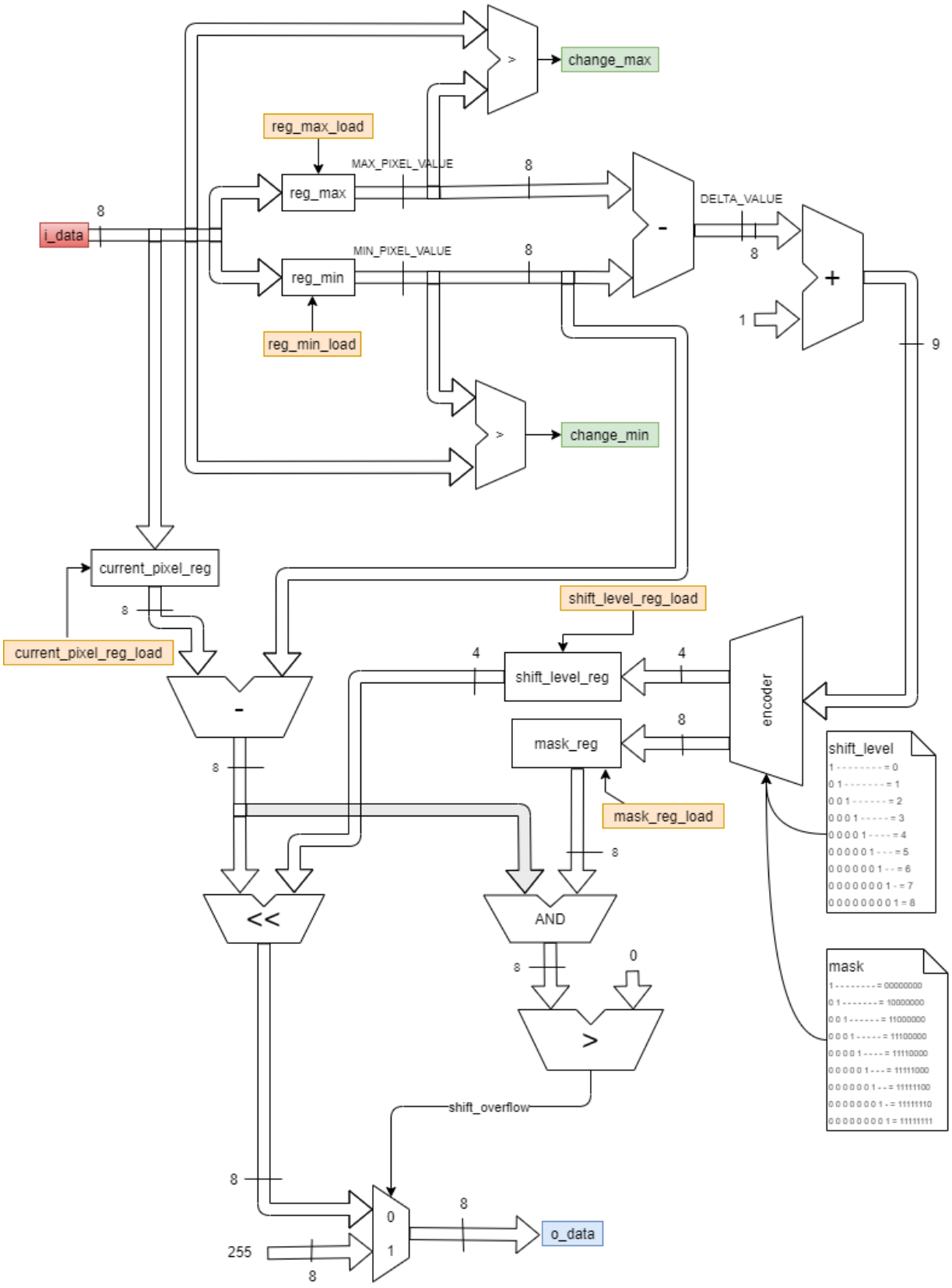

Figura 4: Datapath, seconda parte (pixel work)

2.3 FSM

La macchina a stati finiti fornisce i segnali utili al datapath (nello schema funzionale evidenziati in giallo) e varia il proprio stato in risposta a segnali inviati dal datapath (evidenziati in verde) e segnali esterni inviati al componente (*i_start*). La FSM funge inoltre da ponte per collegare i segnali *i_data, o_data, o_address* tra datapath (nello schema in rosso e blu) e memoria esterna ed è delegato alla creazione di segnali per regolare il flusso dei dati tra i due (*o_done, o_en, o_we*).

Lo schema della macchina in termini di diagramma degli stati è composto da 16 stati, di seguito analizzati singolarmente. Per chiarezza espositiva si omettono i segnali prodotti in ogni stato, sostituendo a questi una descrizione testuale.

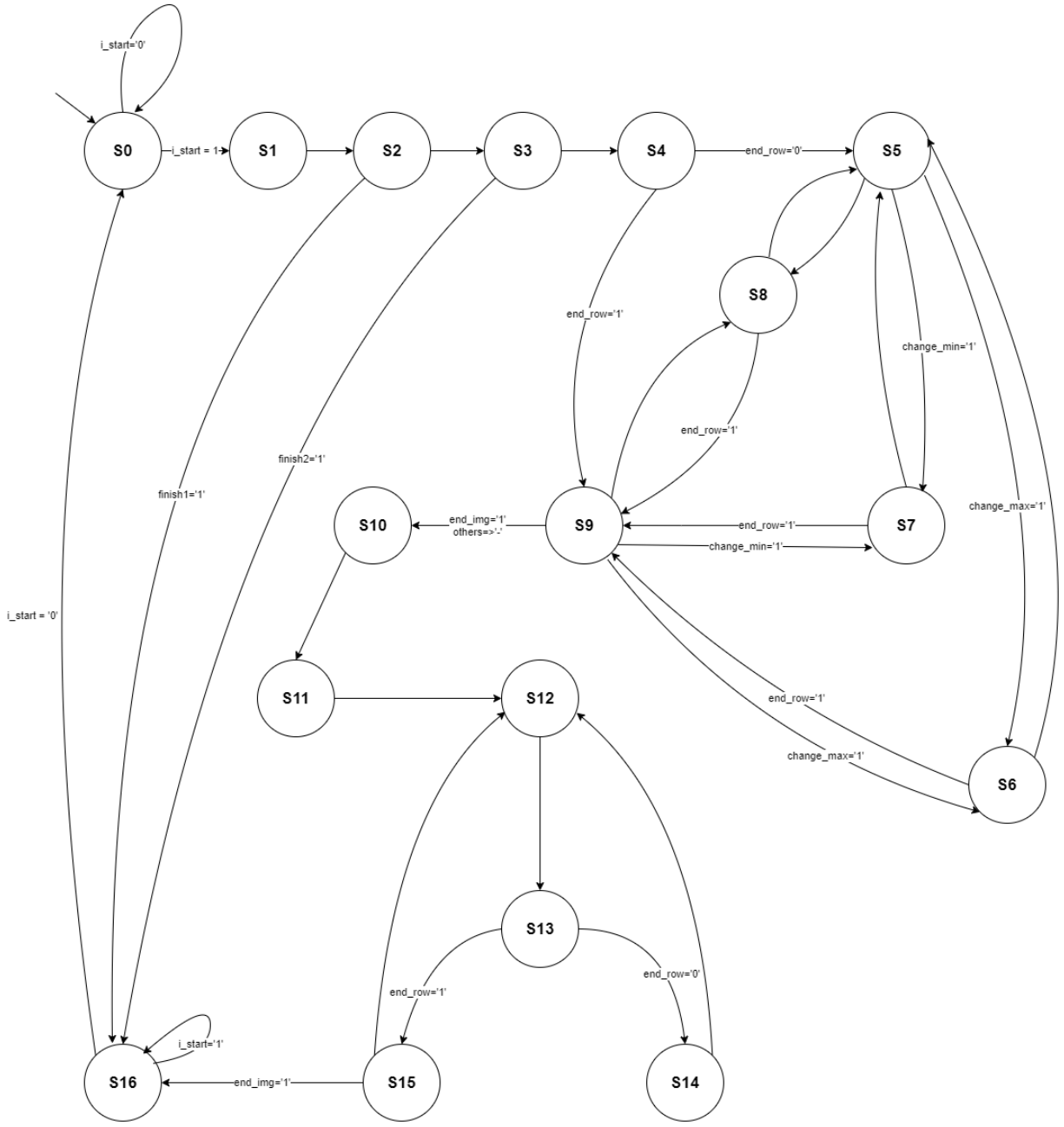

Figura 5: FSM

Il comportamento degli stati può essere così riassunto:

- S0: attende il segnale di *start*
- S1:
  - imposta l'indirizzo a 0
  - comunica all'esterno l'intenzione di leggere dalla memoria
- S2:
  - legge e memorizza il numero totale di colonne
  - incrementa l'indirizzo
- S3:
  - legge e memorizza il numero totale di righe
  - incrementa l'indirizzo
- S4:
  - legge il primo valore e lo salva sia come *MAX_PIXEL_VALUE* che come *MIN_PIXEL_VALUE*
  - resetta il conto di righe e colonne analizzate a 0/1
  - incrementa l'indirizzo
- S5:
  - mantiene l'indirizzo precedente
  - aumenta il numero di colonna e mantiene quello di riga
  - confronta il valore del pixel all'indirizzo corrente con *MAX_PIXEL_VALUE* e *MIN_PIXEL_VALUE* salvati
- S6: il nuovo valore è maggiore del massimo salvato ->
  - salva il valore corrente come nuovo *MAX_PIXEL_VALUE*
  - incrementa l'indirizzo
- S7: il nuovo valore è minore del minimo salvato ->
  - salva il valore corrente come nuovo *MIN_PIXEL_VALUE*
  - incrementa l'indirizzo
- S8: il nuovo valore non è né massimo né minimo ->
  - incrementa l'indirizzo, prepara il prossimo valore su *'i_data'*
- S9:
  - mantiene l'indirizzo precedente
  - aumenta il numero di riga, numero di colonna riparte da 1
  - confronta il valore del pixel all'indirizzo corrente con *MAX_PIXEL_VALUE* e *MIN_PIXEL_VALUE* salvati
- S10: è finita l'immagine da analizzare ->
  - mantiene l'indirizzo precedente
  - salva il primo indirizzo dopo l'immagine in *'new_img'*
  - calcola e salva *shift_level* e la maschera associata
- S11:
  - riporta l'indirizzo a 2
  - resetta il conto di righe e colonne a 0/1
- S12:
  - mantiene gli indirizzi precedenti
  - salva il valore di *current_pixel_value*

- S13:
  - imposta l'indirizzo a cui puntare a *new_img*
  - incrementa l'indirizzo della prima immagine
  - invia il comando di salvare *o_data* contenente *NEW_PIXEL_VALUE* in memoria
- S14:
  - incrementa *new_img*
  - aumenta il numero di colonna e mantiene quello di riga
- S15:
  - incrementa *new_img*
  - aumenta il numero di riga, numero di colonna riparte da 1
- S16: computazione finita ->
  - alza il segnale di *'o_done'*
  - attende l'abbassamento di *'i_start'*

3 RISULTATI SPERIMENTALI

Il componente supera la fase di sintesi, così come tutti i test (Behavioural, Post-Synthesis Functional e Post-Synthesis Timing) a cui è stato sottoposto.

Si esaminano di seguito i risultati della sintesi e i test superati.

3.1 REPORT_UTILIZATION

Dall'analisi del report_utilization le informazioni più importanti possiamo trarle dalla seguente tabella:

Site Type	Used	Fixed	Available	Util%
Slice LUTs*	132	0	134600	0.10
LUT as Logic	132	0	134600	0.10
LUT as Memory	0	0	46200	0.00
Slice Registers	116	0	269200	0.04
Register as Flip Flop	116	0	269200	0.04
Register as Latch	0	0	269200	0.00
F7 Muxes	0	0	67300	0.00
F8 Muxes	0	0	33650	0.00

Come programmato vengono utilizzati 100 flip flop per realizzare i registri parallelo-parallelo del datapath e altri 16 per realizzare la codifica one-hot degli stati della FSM; non vengono inferiti Latch.

Altra informazione che si può ricavare è il basso grado di occupazione della FPGA, di diversi ordini di grandezza inferiore alla disponibilità, risultato scontato essendo la FPGA scelta appositamente sovradimensionata.

3.2 REPORT_TIMING

Analizzando il report_timing è chiaro che il limite di periodo di clock di 100ns è ampiamente rispettato, con un Data Path Delay pari a 3.368ns che, considerando una lieve incertezza del clock, porta a uno Slack rispetto al semiperiodo di 50ns pari a 46.481ns.

Senza contare ritardi dovuti ai collegamenti esterni, si può fissare un limite inferiore alla durata del periodo di clock che consenta al componente di funzionare correttamente:

$$T_{clock} = 2\left(T_{semiperiodoDato} - T_{slack}\right) \cong 7.038ns$$

E quindi una relativa frequenza di clock di:

$$f_{clock} = \frac{1}{T_{clock}} \cong 142\ MHz$$

Tali valori sono in realtà difficilmente realistici in quanto va sommato un delay dovuto alla memoria.

Utilizzando il componente con un periodo di clock pari a 9.7ns / 10ns, tenendo quindi un 10% di margine rispetto al minimo e aggiungendo circa 2ns di delay dovuti alla memoria, il componente dovrebbe funzionare correttamente senza problemi; valori inferiori sono possibili a discrezione del contesto.

3.3 TESTING

Il componente è stato testato attraverso diversi tipi di test per verificarne l'affidabilità sia in condizioni di grandi quantità di dati da analizzare, sia in condizioni di corner case.

Per realizzare i test bench sono state apportate modifiche a quelli forniti in fase di consegna. Tutti i test forniscono al componente una o più immagini e controllano l'esito del risultato attraverso degli *assert.*

Tra tutti si riportano:

- Test immagine completamente bianca
- Test immagine completamente nera
- Test immagine monocromatica
- Test immagine in bianco e nero (solo pixel di valori 0/255)
- Test immagini multiple separate da un segnale di reset
- Test immagini multiple senza reset
- Test immagini vuote (dimensione di riga o colonna = 0)
- Test immagini di dimensione massima (128x128)

Si omettono ulteriori descrizioni dei vari test in quanto si ritiene il loro nome auto esplicativo e un'ulteriore trattazione o visione delle varie waveform risulterebbe ripetitiva.

A titolo di esempio si riporta la Waveform relativa al test bench fornito in fase di consegna.

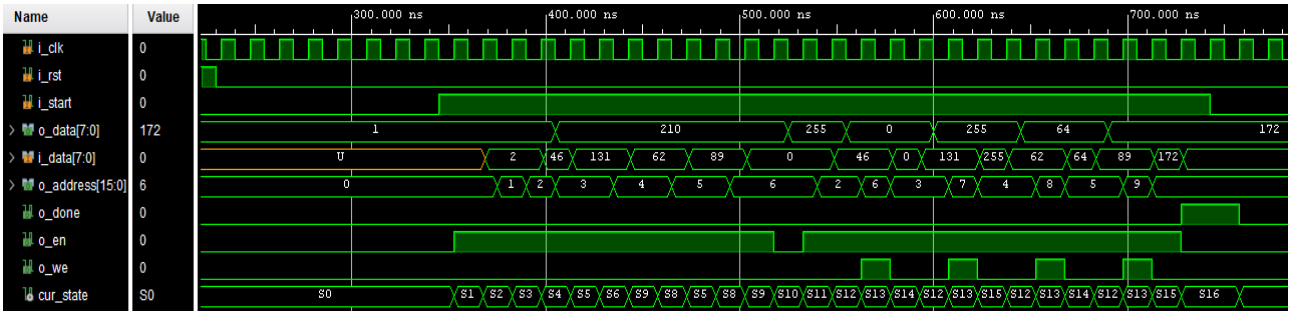

Figura 6: Waveform 1° test bench

Osservando i segnali si può apprezzare come *o_address* cambi sul fronte di discesa del clock mentre lo stato corrente della FSM e i segnali dipendenti da questo variano sul fronte di salita.

4 CONCLUSIONI

Si è realizzato un componente sintetizzabile, che risponde correttamente ai vari test forniti attraverso Behavioural Simulation, Post-Synthesis Functional Simulation e Post-Synthesis Timing Simulation con un ampio margine per aumentare la frequenza di clock di lavoro.

Osservazioni:

- Seguendo l'algoritmo dato, una figura monocromatica viene trasformata in una figura completamente bianca. Si potrebbe facilmente ovviare a questo problema in una successiva revisione.
- Non è stata controllata la minimalità dell'automa a stati finiti, questo potrebbe essere uno spunto per futuri miglioramenti.
- Il componente richiede quasi in ogni momento il collegamento diretto con la memoria tenendo alto il segnale di *o_en* per quasi tutta l'elaborazione. Si potrebbe ridurre questo tempo nella prima fase di lettura dell'immagine salvando il valore letto da memoria in un registro. Nel ciclo successivo tale valore potrà essere letto da quel registro nel caso debba essere salvato come nuovo massimo o minimo, eventualmente effettuando il cambio di valore di *o_en* sul fronte di discesa del clock. Così facendo si potrebbe lasciare libera la memoria di dialogare con eventuali altri componenti. Non si è attuata questa miglioria in quanto il componente è stato creato avendo accesso esclusivo e totale alla memoria esterna.