Prova Finale (Progetto di Reti Logiche)

Politecnico di Milano - A.A. 2020/21

Alberto Boffi[1] (C.P. 10xxxxxx)

Indice

[1] E-mail: alberto2.boffi@mail.polimi.it

1 Introduzione

Nell'ambito del *digital image processing* riveste particolare importanza il problema della regolazione del contrasto. Data un'immagine, l'algoritmo di *equalizzazione dell'istogramma* si occupa di incrementarne il contrasto, basandosi sull'analisi dell'*istogramma dell'immagine*. Quest'ultimo rappresenta il numero di pixels per ogni classe di valore tonale[2]: valori di luminosità molto vicini, dovuti ad un basso contrasto, sono ben visibili dall'istogramma, e il metodo dell'equalizzazione ha come obiettivo la distribuzione dei pixels lungo tutto l'asse tonale.

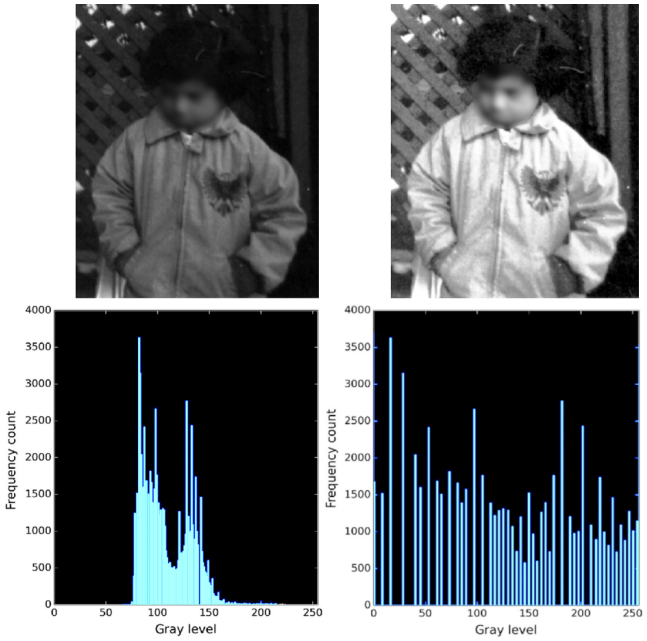

Figura 1: Esempio di applicazione della tecnica di equalizzazione dell'istogramma

1.1 Algoritmo

Si considera una versione semplificata dell'algoritmo, che sarà applicata ad un'immagine in scala di grigi, in cui ogni pixel è rappresentato su un livello da 0 a 255, attraverso un byte.

Per prima cosa si vuole misurare il contrasto dell'immagine, attraverso la differenza tra il massimo e minimo tono presenti:

$$\Delta val = max\_px\_val - min\_px\_val$$

A questo punto, ogni pixel verrà modificato moltiplicando la sua posizione all'interno del range di luminosità per un valore inversamente proporzionale al contrasto dell'immagine. Ciò equivale ad operare il seguente shift:

$$shift\_level = 8 - \lfloor log_2(\Delta val + 1) \rfloor$$
$$temp\_px = (curr\_px\_val - min\_px\_val) << shift\_level$$
$$new\_px\_val = min(255, temp\_px)$$

[2] Misura della percezione umana della luminosità del colore

Notare che la moltiplicazione può produrre un risultato superiore a 255, perciò la funzione *min* si occupa di riportare eventualmente il valore all'interno del range.

1.2 Implementazione

Si vuole realizzare un componente hardware digitale sincrono che implementi l'algoritmo di equalizzazione dell'istogramma, interfacciandosi ad una memoria che contiene l'immagine da elaborare.

Il modulo sarà programmato in codice VHDL attraverso la suite Xilinx Vivado v2020.2, e sintetizzato su FPGA `xc7a200tfbg484-1`.

Clock
Il segnale di clock dovrà avere una frequenza di almeno 10 MHz, con duty cycle 50%.
Affinché dunque il componente funzioni correttamente, i ritardi di propagazione devono essere inferiori al periodo di 100 ns.

Struttura modulo hardware
Il componente da realizzare è caratterizzato dalla seguente interfaccia:

```
entity project_reti_logiche is
  port (
    i_clk, i_rst, i_start: in std_logic;
    o_done: out std_logic;
    -- MEMORY INTERFACE
    i_data: in std_logic_vector(7 downto 0);
    o_address : out std_logic_vector(15 downto 0);
    o_en, o_we: out std_logic;
    o_data: out std_logic_vector (7 downto 0)
  );
end project_reti_logiche;
```

Il segnale di clock sarà connesso all'ingresso *i_clk*, mentre quello di reset, corrispondente al pin *i_rst*, è asincrono e consente di portare il modulo allo stato iniziale.
Si suppone che venga sempre dato un segnale di reset prima della prima codifica.

i_start e *o_done* dettano i tempi della computazione: quando *i_start* va a 1, il modulo inizia ad elaborare l'immagine. Il segnale rimane alto fino a che l'uscita *o_done* non verrà portata alta, per comunicare il termine dell'elaborazione, e *o_done* dovrà rimanere alto fino a che *i_start* non viene riportato a 0. Non può essere dato un nuovo segnale di *i_start* mentre *o_done* è alto.

Per quanto riguarda la comunicazione con la memoria: *i_data* e *o_data* sono rispettivamente il byte in arrivo in seguito ad una lettura e il byte in uscita da scrivere; *o_address* contiene l'indirizzo della cella a cui accedere, mentre *o_en* deve essere alto ogni qualvolta il modulo voglia fare un accesso. In quest'ultimo caso, la modalità d'accesso è descritta dal bit *o_we*: se alto, l'accesso è in scrittura, viceversa in lettura.

Il modulo leggerà e scriverà ogni byte sequenzialmente.

Struttura memoria
La memoria è una RAM sincrona, sensibile al fronte di salita del clock: in risposta ad un accesso, il dato letto o scritto sarà prodotto in uscita solo al ciclo successivo.

Il primo byte, all'indirizzo 0, contiene il numero di righe dell'immagine, mentre il byte all'indirizzo 1 il numero di colonne. Ognuno dei successivi byte corrisponde ad un pixel dell'immagine. Al termine della computazione, le successive celle (dall'indirizzo $n\_row \cdot n\_col + 2$) conterranno la nuova immagine con contrasto aumentato.

L'immagine da elaborare non cambierà mai fino al termine della computazione.

2 Architettura

2.1 Struttura generale

Il modulo da implementare è stato diviso in un'*unità di elaborazione* (*datapath*) e un'*unità di controllo (cu)*. Quest'ultima, descritta come macchina a stati, controllerà il datapath attraverso un opportuno scambio di segnali.

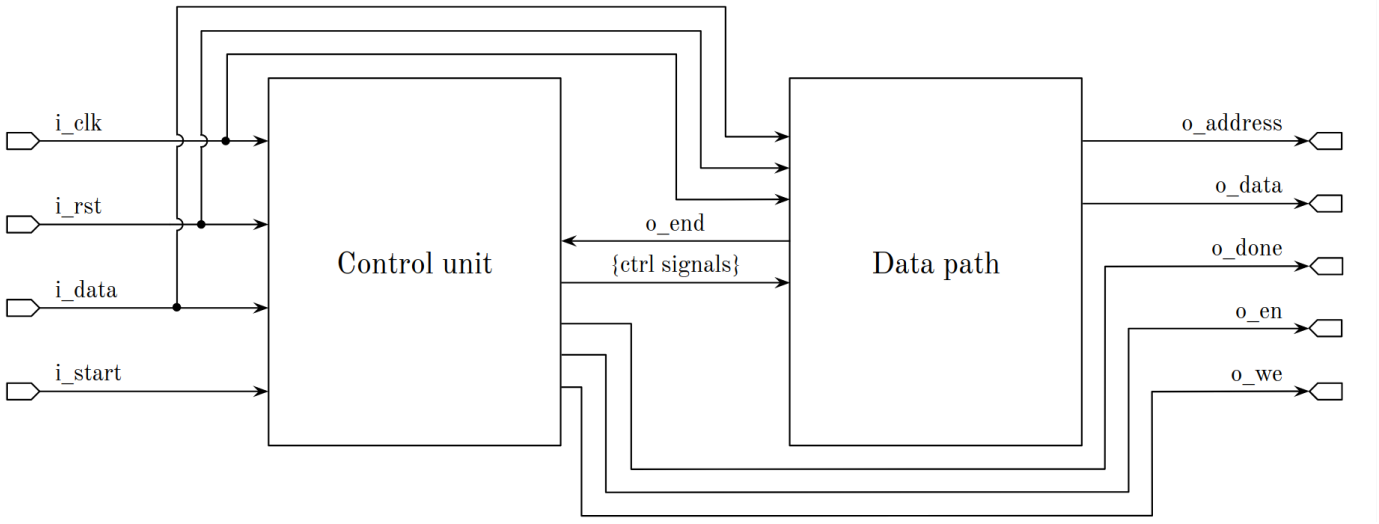

Figura 2: Architettura generale
{ctrl signals} è un insieme di segnali di controllo sul datapath che verrà discusso in seguito.

2.2 Unità di elaborazione

2.2.1 Logica implementativa

Sono stati assegnati al datapath diversi compiti, in modo da meccanizzare il processo di calcolo, lasciando all'unità di controllo solo il ruolo di "controller" del datapath.

È stato dunque necessario modularizzare la struttura, per avere un maggiore controllo su ogni componente, aumentare la comprensibilità e la garanzia di funzionamento.

Per prima cosa è stato creato un modulo per ognuno dei due componenti maggiormente usati:

- **Multiplexer** a 2 ingressi dati e 1 ingresso di selezione
- **Registro** parallelo-parallelo sensibile al fronte di salita, con ingresso di load e reset asincrono prioritario

In entrambi, la dimensione dei segnali d'ingresso viene assegnata durante l'istanziazione attraverso generic mapping.

La loro creazione ha semplificato enormemente la costruzione del data path, che si è così ridotta alla semplice connessione di segnali, oltre che alla realizzazione di operazioni secondarie. Non da meno, il testing preventivo di questi componenti ha garantito la correttezza del circuito in tutti i punti in cui sono stati instanziati.

2.2.2 Struttura

Il datapath è composto da 3 moduli sconnessi che operano in parallelo.

Per ragioni di semplicità, sono stati omessi nei circuiti gli ingressi di reset e di clock dei flip flop.

Algorithm
Il primo modulo si occupa di implementare l'algoritmo.

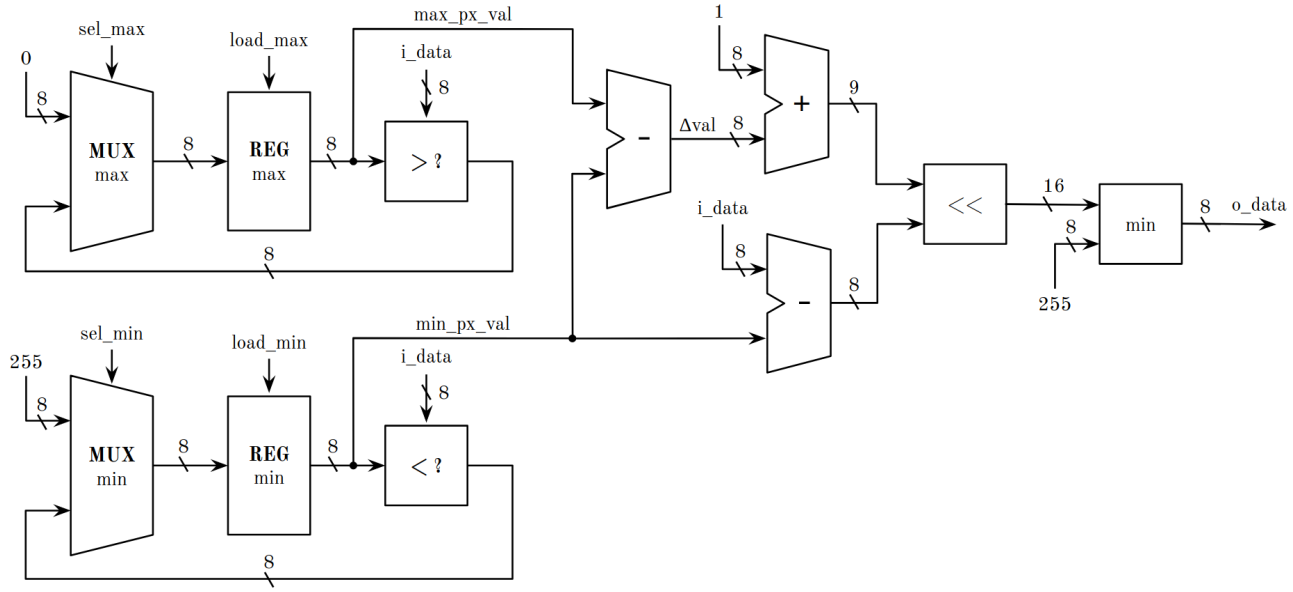

Figura 3: Schema circuitale del modulo *algorithm*

La ricerca di *max_px_val* e *min_px_val* è stata fatta confrontando ogni byte letto con il massimo/minimo byte trovato fino quel momento. Tale byte è memorizzato un registro inizializzato a 0 (nel caso di ricerca pixel massimo) o 255 (nel caso di ricerca pixel minimo).
Il componente | >? | produce in uscita il byte maggiore tra quelli ricevuti in ingresso, mentre | <? | il minore.

Dati gli estremi tonali, i restanti calcoli da fare per ogni pixel dell'immagine sono puramente combinatori.

Il componente $|<<|$, presi in ingresso $< curr\_px\_val - min\_px\_val >$ e $< \Delta val + 1 >$, produce lo shift restituendo *temp_px*.
Il calcolo di *shift_level* viene infatti eseguito *solo implicitamente* sulla base del numero di 0 più significativi presenti in $< \Delta val + 1 >$, secondo il seguente schema:

$\Delta val + 1$	$shift\_level$
100000000	0
01 - - - - - - -	1
001 - - - - - -	2
0001 - - - - -	3
00001 - - - -	4
000001 - - -	5
0000001 - -	6
00000001 -	7
000000001	8

Non è infatti necessario approfondire lo schema, in quanto si occuperà Vivado, durante la sintesi, di ottimizzare il risultante circuito combinatorio.

Address producer
Altro compito del datapath è quello di produrre gli indirizzi di memoria a cui accedere, in lettura e scrittura.

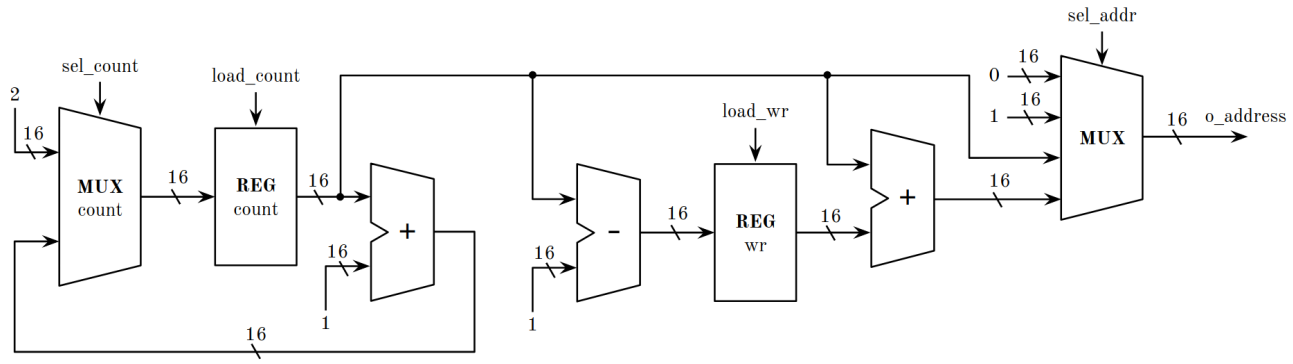

Figura 4: Schema circuitale del modulo *address producer*

Il multiplexer finale seleziona l'indirizzo di accesso. I primi due ingressi corrispondono agli indirizzi 0 e 1, riservati al numero di righe e colonne. Il terzo ingresso riceve l'indirizzo del pixel da leggere, generato da un semplice contatore in progressione inizializzato al valore 2 decimale. Eseguirà due cicli di lettura: il primo interessa il calcolo di *min_px_value* e *max_px_value*, il secondo serve per completare la restante parte dell'algoritmo.
Al termine del primo ciclo, *reg_wr* memorizzerà il penultimo byte letto. Esso sarà utile nel secondo ciclo, in quanto rappresenta l'offset tra l'indirizzo di ogni pixel letto e l'indirizzo in cui dovrà essere riscritto, selezionato dal quarto ingresso del multiplexer.

Counter
Ovviamente è necessario disporre di un meccanismo di controllo dell'*address producer*, per sapere quando arrestare la lettura, sulla base del numero di righe e colonne.

Tale controllo *non* è fattibile attraverso l'operatore aritmetico VHDL di moltiplicazione, in quanto verrebbe sintetizzato in una gigantesca rete combinatoria, introducendo significativi ritardi nel circuito.

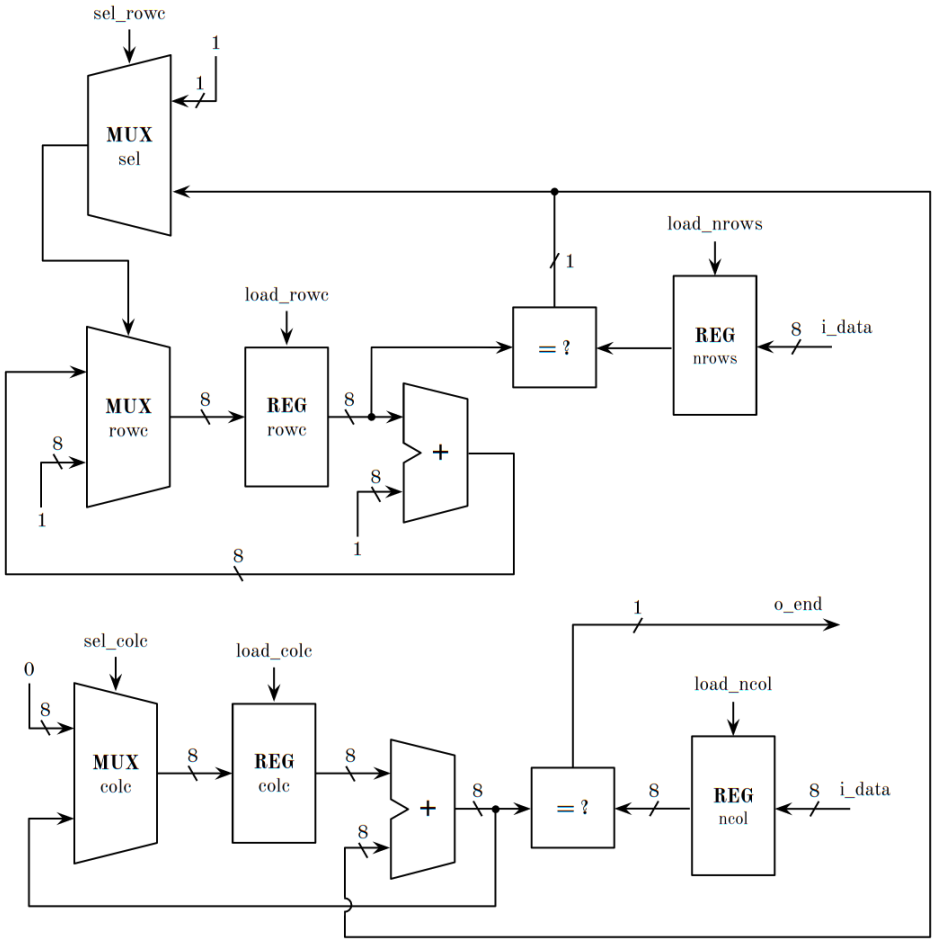

Figura 5: Schema circuitale del modulo *counter*

È stato utilizzato un contatore modulo m in cascata ad un contatore modulo n, dove n e m sono rispettivamente il numero di righe e colonne. Ogni qualvolta il contatore di righe termina il conteggio, fa incrementare il contatore di colonne e si auto-resetta.
Una volta terminato il conteggio, verrà portato a 1 il segnale *o_end*, informando l'unità di controllo che l'immagine è terminata.

2.3 Unità di controllo

L'unità di controllo è rappresentata da una macchina a stati finiti deterministica (FSM). In particolar modo, si è optato per una *macchina di Moore*, dunque l'uscita risponde ad una variazione degli ingressi solo al ciclo di clock successivo.

Come osservabile dai circuiti precedentemente illustrati, l'insieme di segnali prodotti dall'unità di controllo per regolare il funzionamento del datapath è:
ctrl signals = {sel_max, load_max, sel_min, load_min, sel_count, load_count, load_wr, sel_addr, sel_rowc, load_rowc, load_nrows, sel_colc, load_colc, load_ncol}

Per ragioni di comprensibilità, nel disegno sono state indicate, per ogni stato, solo le uscite che sono state modificate rispetto allo stato precedente.
Nello stato iniziale, ogni uscita vale 0.

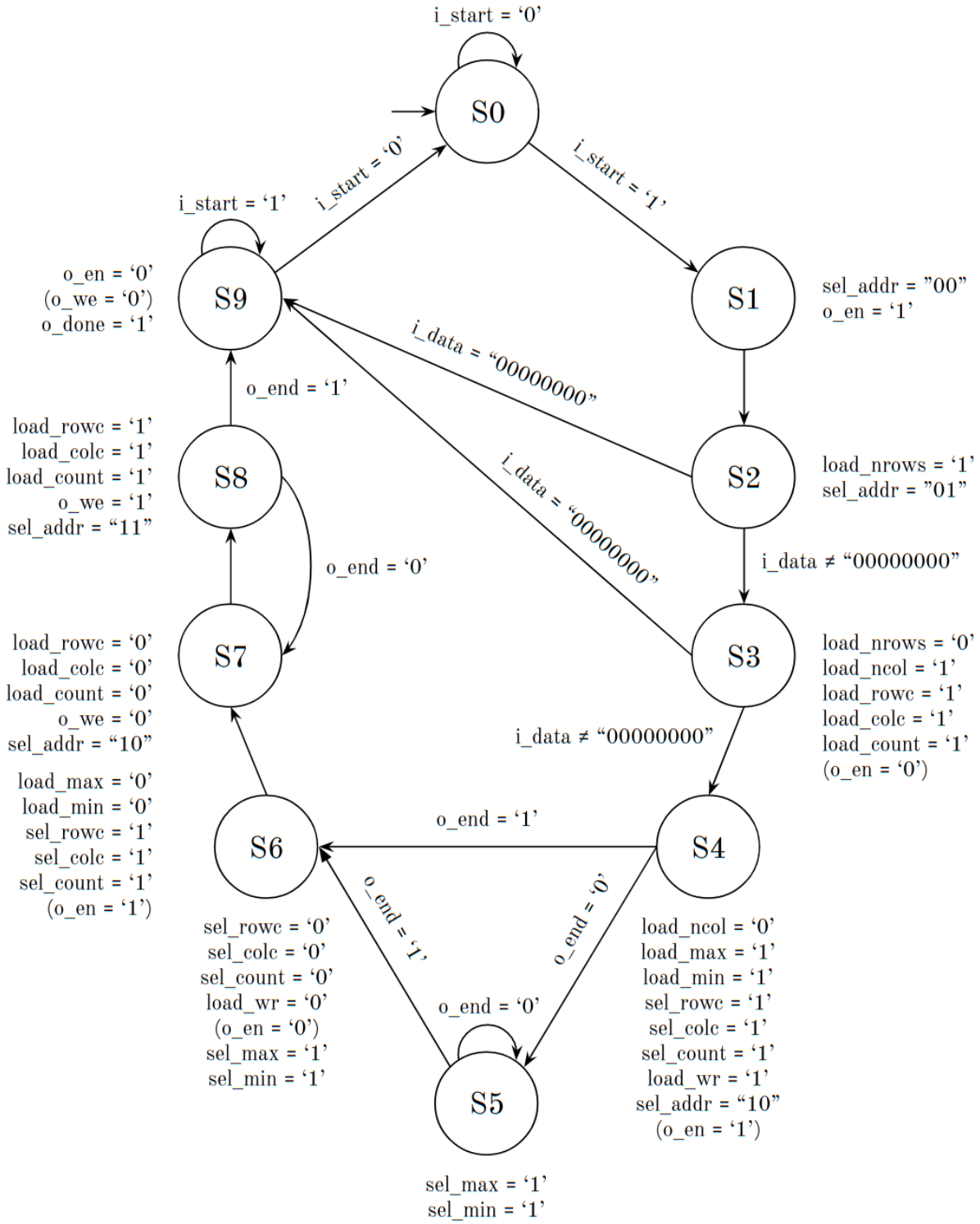

Figura 6: Diagramma degli stati della macchina a stati finiti

2.3.1 Descrizione FSM

S0 - Reset
Nessuna computazione in corso. Tutte le uscite della FSM sono basse: i registri non memorizzano e il modulo non interagisce con la memoria.

Ogni volta che il segnale *i_rst* va alto, si torna in questo stato. Tale transizione è asincrona ed è quindi stata omessa dal diagramma.

S1 - Selezione numero righe
Il segnale di start è stato portato alto e la computazione è iniziata.
Viene fatto accesso in lettura al numero di righe.

S2 - Selezione numero colonne
Viene ricevuto il numero di righe.
Viene acceduto il numero di colonne.

S3 - Predisposizione lettura
Viene ricevuto il numero di colonne.
I moduli *counter* e *address producer* vengono predisposti per iniziare il ciclo di lettura dell'immagine.

S4 - Inizio lettura
Viene fatto accesso al primo pixel.
Il modulo *algorithm* viene predisposto per iniziare il calcolo.

S5 - Calcolo estremi tonali
Viene ricevuto un pixel dell'immagine che il modulo *algorithm* usa per calcolare *min_px_val* e *max_px_val*, mentre *address producer* fa acceso al pixel successivo.
Si resta in questo stato fino al termine dell'immagine.

S6 - Predisposizione equalizzazione
Il modulo *counter* ha decretato la fine della lettura.
Viene ricevuto l'ultimo pixel, con il quale si completa definitivamente il calcolo degli estremi tonali.
Vengono resettati tutti i contatori.

S7 - Lettura pixels
Secondo ciclo di lettura.
Viene letto un pixel dell'immagine, e i moduli *counter* e *address producer* arrestano il conteggio.

S8 - Scrittura pixels
Viene ricevuto il pixel letto.
Vengono quindi eseguiti su di esso gli appositi calcoli e viene scritto in memoria. Sarà effettivamente disponibile in RAM a partire dal ciclo successivo (stato S7 o S9).
I moduli *counter* e *address producer* vengono abilitati per riprendere il conteggio.

S9 - Termine computazione
Il contrasto dell'immagine è stato aumentato (oppure l'immagine aveva dimensione di riga e/o colonna nulla).
L'ingresso *o_done* viene tenuto alto fino a che *i_start* non viene portato basso.

2.3.1 Implementazione

Il diagramma degli stati è sufficiente per implementare la macchina, in quanto codifiche e ottimizzazioni sono delegate a Vivado nella fase di sintesi.

Gli stati sono stati rappresentati attraverso la definizione di un nuovo tipo enumerazione.

La macchina di Moore è stata realizzata riflettendo la sua struttura classica, ossia dividendo idealmente il modulo in 3 parti:

- Una parte combinatoria, che riceve gli input del datapath e lo stato presente, restituendo lo stato futuro
- Una parte sequenziale, sensibile a *clock* e *reset* e avente come terzo input lo stato futuro. Restituisce lo stato presente, che è rappresentato da S0 e viene dato istantaneamente se reset=1, viceversa viene prodotto in uscita ad ogni fronte di salita del clock.
  Tale blocco sarà poi sintetizzato in un banco di flip flop.
- Un'ulteriore parte combinatoria, che preso in ingresso lo stato presente produce le uscite dell'unità di controllo

Per semplicità, anche la prima e l'ultima parte sono scritte mediante processi, prestando particolare attenzione all'assegnazione di tutti i segnali affinché non fossero inferiti latch.

3 Risultati sperimentali

3.1 Test benches

3.1.1 Tipologie

La verifica comportamentale del modulo è stata eseguita con un approccio bottom-up, attraverso *test benches di unità*, per stimolare le entità atomiche, e *test benches di integrazione*, per la verifica delle macro-entità.
In questa metodologia i moduli a più basso livello vengono testati per primi e più a lungo, e ciò offre maggiori garanzie di funzionamento e una più semplice localizzazione degli errori.

3.1.2 Contenuti

Il modulo è stato in prima battuta testato con il test bench pubblico, e poi su altre semplici immagini, allo scopo di osservare più agevolmente che il suo comportamento rispettasse i requisiti.

Le successive verifiche sono state fatte attraverso stress test che comprendevano immagini input di ogni tipo.
Per facilitare il processo di collaudo, ho realizzato uno script Python per la generazione automatica di test benches contenenti pixels dai valori pseudo-casuali.
Nel generatore, per garantire una completa copertura del dominio di *shift_level*, l'immagine viene creata al contrario: viene prima scelto *shift_level*, e conseguentemente i valori dei pixels a seconda del corrispondente range[3].

Ciò ha permesso innanzitutto di testare il progetto su ogni possibile valore di *shift_level*.
Sì è inoltre prestata particolare attenzione alle *boundary conditions* (casi limite), dettate dal dominio degli ingressi:

- **Immagine minima**: Caso in cui le dimensione di riga e/o colonna siano pari a 0, oppure siano pari a 1. Sebbene il caso in cui sia nulla solo una delle due dimensioni sia tecnicamente privo di senso, ci si è protetti anche per questo corner case.

- **Immagine massima**: Caso in cui le dimensioni di riga e/o colonna siano 128;

- **Immagine a contrasto minimo**: Caso in cui la differenza tra i pixels con tono massimo e minimo sia 0;

- **Immagine a contrasto massimo**: Caso in cui la differenza tra i pixels con tono massimo e minimo sia 255;

- **Immagini consecutive**: Al termine di una prima computazione, viene dato un nuovo segnale di *start* per lavorare su un'altra immagine;

- **Reset**: Il modulo viene resettato in istanti casuali della computazione. Si è verificato che il modulo abbia effettivamente lavorato sull'immagine solo prima di essere stato resettato.

Tali condizioni, oltre ad essere valutate singolarmente, sono state combinate negli stessi casi di test.

Sì è inoltre prestata particolare attenzione al fatto che *o_done* fosse effettivamente portato a 0 solo dopo aver abbassato *i_start*.

[3] Scegliendo infatti i pixels casualmente all'interno del loro intero dominio (0,255), al crescere dell'immagine la probabilità di trovare 2 pixels con valori molto distanti è esageratamente grande, e ciò porterebbe a testare il modulo solo su immagini con *shift_level* basso.

3.1.3 Risultati

Pre-sintesi e post-sintesi: Ogni banco di test, valutato "ai morsetti" del modulo, ha prodotto gli stessi risultati nella simulazione pre-sintesi e post-sintesi funzionale.

Esempio di comportamento: Ecco riportato, a scopo dimostrativo, il diagramma temporale[4] che rappresenta l'intera computazione su una semplice immagine 2x1 di pixels *[18,40]*. L'immagine risultato sarà *[0,255]*, come per ogni altra immagine di 2 pixels:

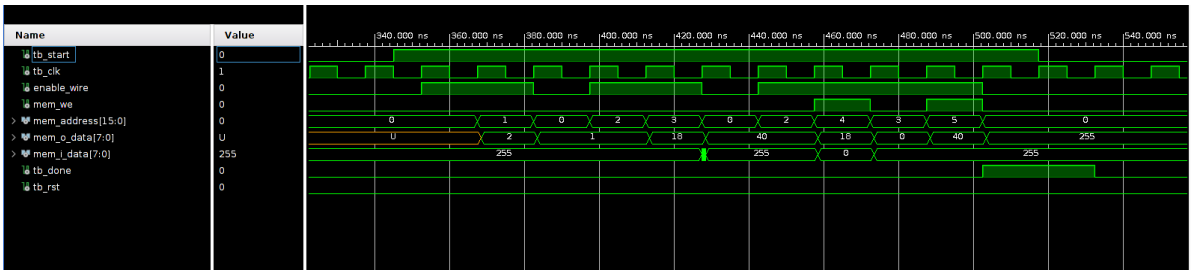

Figura 7 - Esempio di funzionamento
Il glitch (del tutto innocuo) di *mem_i_data* all'istante 427.5 ns è dovuto al ritardo con cui la RAM produce il dato in uscita

3.2 Report di sintesi

Alcuni interessanti risultati del report di sintesi sono i seguenti:

Timing
```
Slack: 94.634ns
 Data Path Delay: 5.215 ns
```

Utilization
```
LTU as Logic: 198
LTU as Memory: 0
Register as Flip Flop: 90
Register as Latch: 0
```

Il modulo è quindi veloce in relazione ai requisiti progettuali, e ciò consente di poter eventualmente aumentare di molto la frequenza di funzionamento.

Il numero di flip flop è effettivamente quello aspettato, infatti:

- Nel datapath sono stati utilizzati 8 registri, 6 dei quali a 8 bit, e 2 a 16 bit;
- Per quanto riguarda l'unità di controllo, Vivado utilizza di default una codifica One-hot, sintetizzando dunque 10 flip flop per memorizzare 10 differenti stati.

[4] Periodo di clock di 15 ns

Dato che il design non prevedeva latch, il fatto che non siano stati inferiti rassicura sul fatto che le parti combinatorie scritte attraverso *process* siano effettivamente risultati tali.

4 Conclusioni

La realizzazione del progetto ha avuto innanzitutto come obiettivo la costruzione di un componente che rispettasse le specifiche.
Le scelte progettuali si sono però mosse in direzione di altri importanti requisiti, tra i quali:

- In termini temporali, è stata applicata il più possibile parallelizzazione tra le operazioni svolte, consentendo di eseguire l'algoritmo restando nella sua complessità computazionale $O(n)$.
- In termini di componenti fisici, si è optato ove possibile per una descrizione dell'hardware a basso livello, al fine di garantire ordine e modularizzazione. Tale approccio, oltre a migliorare come già accennato verificabilità e comprensibilità, sarà di grande vantaggio per eventuali successive manutenzioni ed estensioni.