Prova finale
progetto di reti logiche

Ludovica Bindi 10619971
Andrea Borsatto 10628989

POLITECNICO
MILANO 1863

A.A. 2020/ 21

0. Indice

1. Introduzione

L'obiettivo del circuito qui elaborato è quello di aumentare il contrasto di un'immagine salvata in una memoria andando a modificare i pixel che la compongono tramite una versione semplificata del metodo dell'equalizzazione. Il metodo consiste nel cambiare la distribuzione dei pixel lungo l'intervallo d'intensità. In una immagine con poco contrasto i pixel assumono valori molto simili tra di loro: grazie a questo metodo i nuovi pixel assumeranno valori su tutto l'intervallo di intensità.

L'algoritmo da implementare è il seguente:

DELTA_VALUE = MAX_PIXEL_VALUE – MIN_PIXEL_VALUE;

SHIFT_LEVEL = 8 – FLOOR(LOG2(DELTA_VALUE + 1));

TMP_PIXEL = (CURRENT_PIXEL_VALUE – MIN_PIXEL_VALUE) << SHIFT_LEVEL;

NEW_PIXEL_VALUE = MIN(255, TMP_PIXEL);

Dove MAX_PIXEL_VALUE e MIN_PIXEL_VALUE sono rispettivamente il massimo e il minimo valore assunto dai pixel dell'immagine da modificare; il CURRENT_PIXEL_VALUE rappresenta il singolo pixel che si va a modificare. La nuova immagine sarà ottenuta guardando l'insieme dei NEW_PIXEL_VALUE.

Dall'algoritmo si evince che minore è la differenza tra il valore massimo e minimo dei pixel (quindi minore il contrasto), maggiore sarà il valore di SHIFT_LEVEL che, appunto, modifica il valore attuale del pixel decrementato di MIN_PIXEL_VALUE applicando tanti shift a sinistra dei bit quanto vale SHIFT_LEVEL; maggiore SHIFT_LEVEL più i pixel saranno discostati dal loro valore attuale. Il fatto che sia (CURRENT_PIXEL_VALUE – MIN_PIXEL_VALUE) a essere modificato piuttosto che semplicemente CURRENT_PIXEL_VALUE serve per non ottenere una mera traslazione dei valori dell'immagine, ma per poter distanziarli e quindi ricoprire maggiormente l'intero intervallo di intensità.

Infine, l'ultima riga serve per non eccedere la scala dei grigi dell'immagine (anche perché viene richiesto che la nuova immagine venga salvata nella stessa memoria di quella originaria che è composta da pixel di 8 bit, mentre valori di nuovi pixel superiori a 255 eccederebbero gli 8 bit di vincolo).

In questo metodo di elaborazione delle immagini, risulta interessante analizzare come varia la funzione di distribuzione cumulativa (FDC, in cui FDC(x) coincide con il numero di pixel dell'immagine che assumono un valore minore o uguale a x). In un'immagine a basso contrasto, questa funzione presenta un incremento brusco dove si concentra la maggior parte dei valori dei pixel, mentre nel resto dell'intervallo ha una crescita più lenta; successivamente all'equalizzazione, la FDC presenta un andamento più omogeno.

Un esempio di quanto appena detto si può osservare nelle immagini e nei grafici riportati di seguito: la figura 1 contiene un'immagine poco contrastata con il relativo istogramma e FDC, la successiva presenta invece i risultati del processo di equalizzazione. Queste figure sono state realizzate utilizzando Matlab e applicando l'algoritmo sopra riportato.

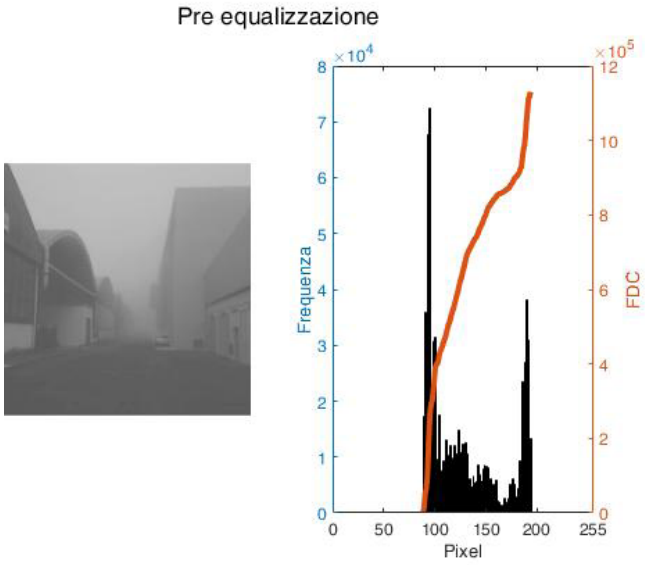

fig. 1

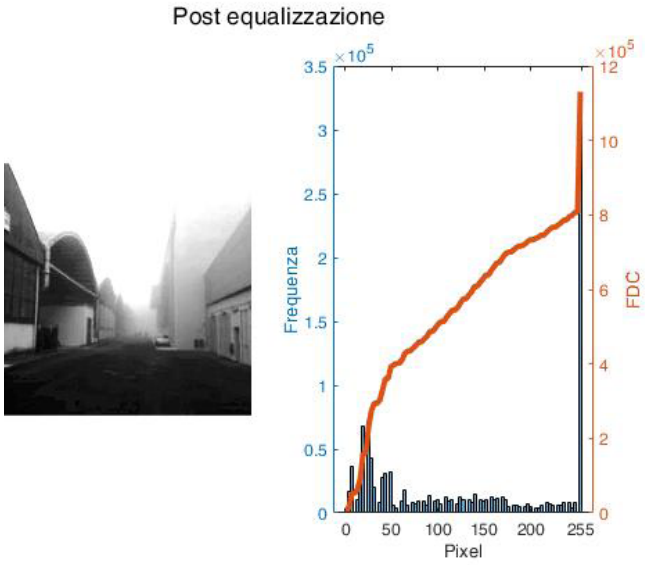

fig. 2

2. Architettura

Il nostro circuito di elaborazione è suddiviso in due parti: l'unità di elaborazione (datapath) e l'unità di controllo (finite state machine, FSM).

Abbiamo deciso di dividere ulteriormente il datapath in due parti logiche distinte: la prima si occupa di calcolare i parametri dell'immagine (MAX_PIXEL_VALUE, MIN_PIXEL_VALUE, DELTA_VALUE e SHIT_VALUE) che serviranno alla seconda parte per modificare i pixel dell'immagine e salvarli.

L'immagine è immagazzinata nella memoria sequenzialmente e ha una dimensione indicata da due byte salvati a partire dall'indirizzo zero (rispettivamente il numero di colonne, N_COL, e il numero di righe, N_RIGHE). Bisogna però tenere conto del caso in cui almeno uno dei due valori sia uguale a 0: in questo caso non c'è immagine da elaborare.

a. Finite State Machine

Per non appesantire il disegno, nelle immagini della FSM abbiamo deciso di evidenziare soltanto i segnali che sono messi o rimangono a 1, mentre tutti gli altri segnali omessi sono implicitamente a 0. Questo si riflette anche nel codice VHDL: all'interno della FSM nel process che si occupa di calcolare la funzione d'uscita, indipendentemente dallo stato, settiamo tutti i segnali a 0 e all'interno del costrutto case modifichiamo solo quelli che cambiano valore.

Il circuito evolve in questo modo: dopo una prima fase di preparazione del circuito, vengono preparati tutti i registri per l'interazione con la memoria per poi estrarre tutti i pixel dell'immagine contenuta in essa; grazie a questa prima lettura sono ottenuti i valori dei pixel massimi e minimi, vengono calcolati tutti i parametri necessari per l'attuazione dell'equalizzazione ed infine, ottenendo di nuovi i dati dalla memoria, i valori dell'immagine finale sono preparati e salvati nella memoria stessa. Il circuito viene così di nuovo preparato

per una nuova elaborazione. In figura 3 abbiamo preparato un disegno esemplificativo di come funzioni la macchina a stati: i vari "macrostati" qui rappresentati esemplificano una fase dell'esecuzione precedentemente descritta.

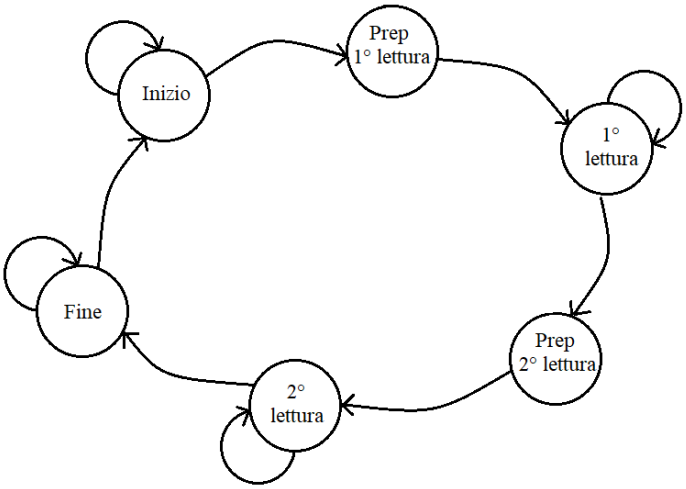

fig. 3

Nella descrizione del datapath a seguire, per ogni descrizione degli elementi descriveremo analizzeremo gli stati dell'FSM che li interessano, andando così ad approfondire i singoli "macrostati", per poi fornire una visione dettagliata e complessiva della macchina a stati. Ora qui, descriveremo come si comporta all'inizio e alla fine della elaborazione.

Inizio

In S_rst, il circuito aspetta di ricevere il segnale di i_rst che lo porta ad iniziare la prima elaborazione in assoluto; una volta ricevuto questo segnale, i registri settano il loro valore interno a una serie di 0 affinché il circuito possa funzionare correttamente; si rimane in S_start finché non viene ricevuto anche il segnale di i_start. In Sx il circuito chiede il primo dato alla memoria, che poi viene salvato nello stadio successivo, S0, nel quale chiediamo il secondo dato per memorizzarlo in pp_S1. Questi due dati vengono salvati rispettivamente in reg_col e reg_row perché rappresentano le dimensioni dell'immagine.

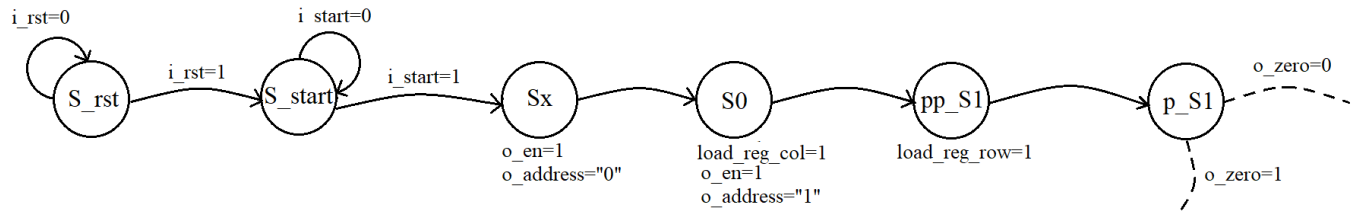

fig. 4

In p_S1 il circuito verifica che entrambe le dimensioni siano valide guardando il valore di o_zero (che è uguale a 1 se almeno una delle due dimensioni è uguale a 0) proveniente dal datapath: se o_zero è uguale a 0 si procede con la normale elaborazione dell'immagine, altrimenti si salta direttamente allo stato S10.

Fine

Una volta finita l'elaborazione (ricevuto il segnale di o_read_done, si veda b.ii) o dopo aver ricevuto la richiesta di elaborare un'immagine vuota (o_zero = 1), la FSM rimane in S10 finché non viene riportato basso i_start; in S11 viene dato il segnale di internal_reset che forza il reset di tutti i registri per una eventuale successiva elaborazione (svolge lo stesso ruolo di i_rst, soltanto che è un segnale interno al circuito) ed infine va in S_start pronto per eseguirla.

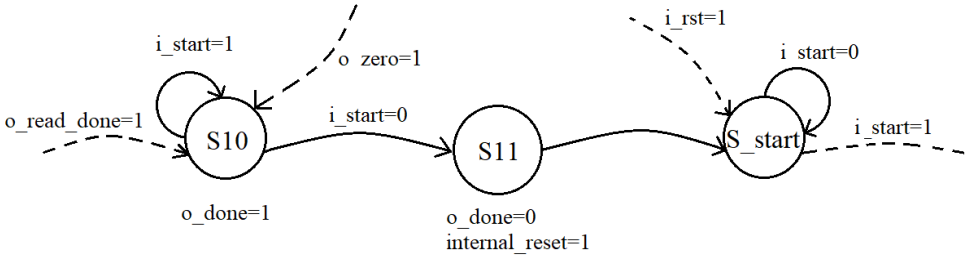

fig. 5

Come si vede, la FSM è un loop che viene eseguito tante volte quante sono le elaborazioni di immagini richieste. Lo stadio S_rst è l'unico che viene eseguito esclusivamente una volta poiché serve per preparare la prima elaborazione dopo l'attivazione del circuito stesso. Inoltre, per non appesantire i disegni, non è stata rappresentata la gestione del segnale asincrono di reset che può essere dato dall'esterno in qualunque momento dell'elaborazione (l'elaborazione viene interrotta e il circuito si riporta allo stato iniziale): il process all'interno dell'FSM, che gestisce l'assegnamento dello stato successivo, si può attivare al solo variare di i_rst (indipendentemente dal ciclo di clock) e assegnare S_start a cur_state se i_rst è alzato.

b. Datapath

i. Prima parte

La prima parte del circuito, ricevuti i segnali di reset e start, legge dalla memoria una volta tutta l'immagine, pixel per pixel, al fine di calcolare il valore massimo e il valore minimo e, successivamente, tutti gli altri parametri.
Per non appesantire i disegni del circuito, abbiamo rappresentato solo i segnali d'ingresso più significativi ai vari componenti ed elementi circuitali.
Questa parte è logicamente divisa in vari sotto-circuiti ciascuno addetto a un compito specifico.

Controllo delle dimensioni

La parte del circuito che controlla se le dimensioni sono valide (quindi che produce il segnale o_zero sopra menzionato) è composta da due comparatori che confrontano il contenuto di reg_col e di reg_row con il valore 0: le loro uscite vengono unite in una porta OR generando o_zero che esce dal datapath per finire nella FSM.

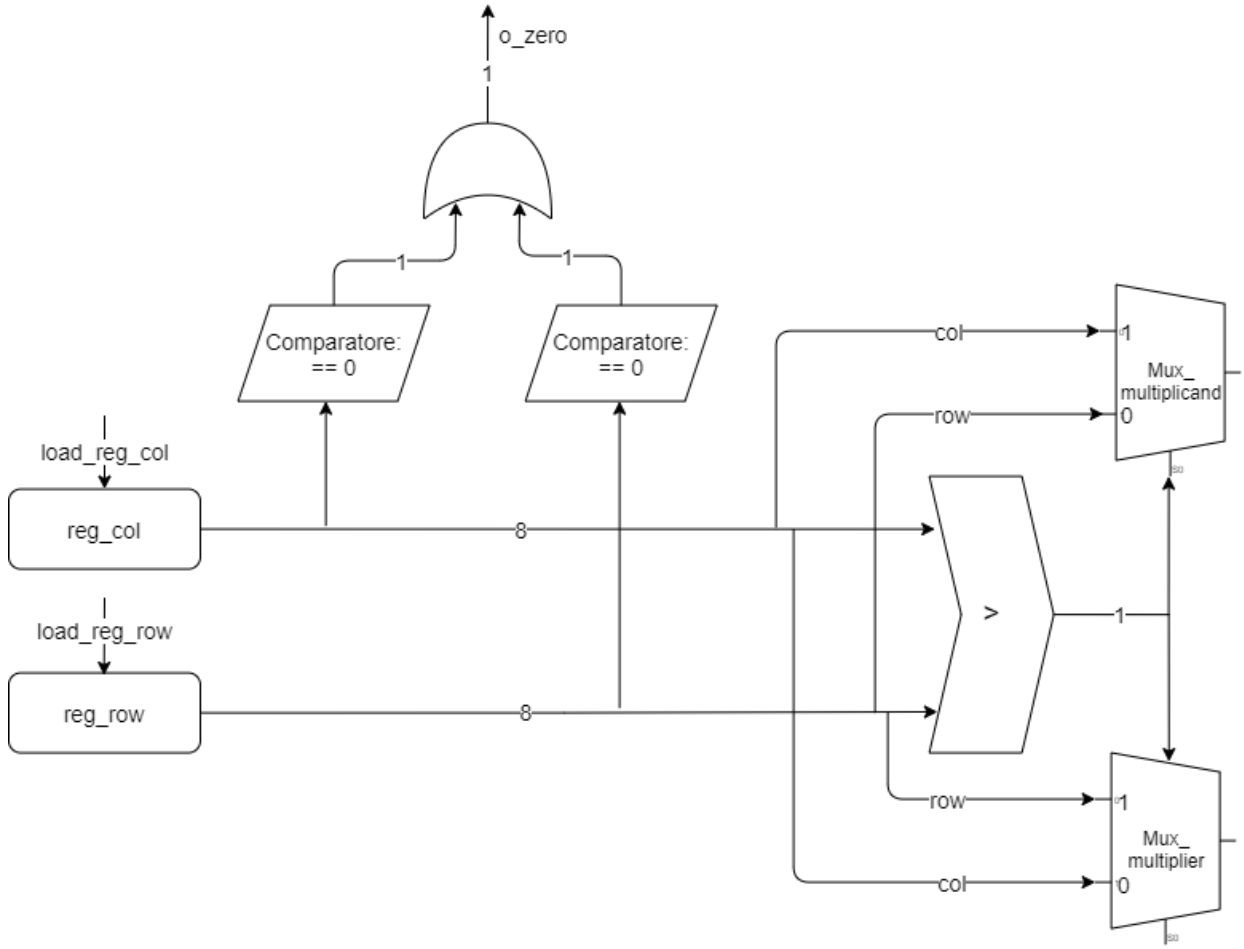

fig. 6

Moltiplicatore

Per leggere dalla memoria, abbiamo bisogno di conoscere fino a quanto l'immagine si estende. Questa informazione si può ottenere calcolando (N_COL * N_RIGHE) + 2: questo numero coincide con l'indirizzo che contiene l'ultimo pixel. L'obiettivo del componente MOLTIPLICATOR0 è proprio quello di calcolare il prodotto che poi verrà utilizzato per generare tutti gli indirizzi da richiedere alla memoria.

Abbiamo deciso di implementare questa operazione tramite delle somme ripetute, (sfruttando un moltiplicando n e un moltiplicatore m). Quindi, il nostro componente si decompone in due parti: la prima che genera le somme ripetute (somma continuamente n ai risultati intermedi), la seconda che controlla quante somme sono necessarie (m). Il tutto viene mediato dalla macchina a stati. MOLTIPLICATOR0 evolve in modo tale che a ogni ciclo di clock venga eseguita una somma.

Nel nostro circuito, per non rendere il calcolo della moltiplicazione inefficiente dal punto di vista temporale, sfruttiamo un ciclo di clock per confrontare n e m e usiamo il maggiore dei due come moltiplicando, in modo da ripetere le somme il numero minimo di volte. Questo meccanismo è svolto dai componenti circuitali posti prima di MOLTIPLICATOR0 (disegnati in figura 6): il comparatore prende in ingresso l'uscita dei due registri e restituisce ai due multiplexer la valutazione dell'espressione booleana reg_col > reg_row; gli ingressi di questi due componenti sono disposti in modo tale che in i_multiplicand finisca il valore maggiore contenuto nei due registri e in i_multiplier quello minore. Questa operazione avviene nello stato p_S1. Non c'è necessità di salvare questi due segnali in appositi registri poiché reg_col e reg_row non verranno più aggiornati fino alla fine della elaborazione.

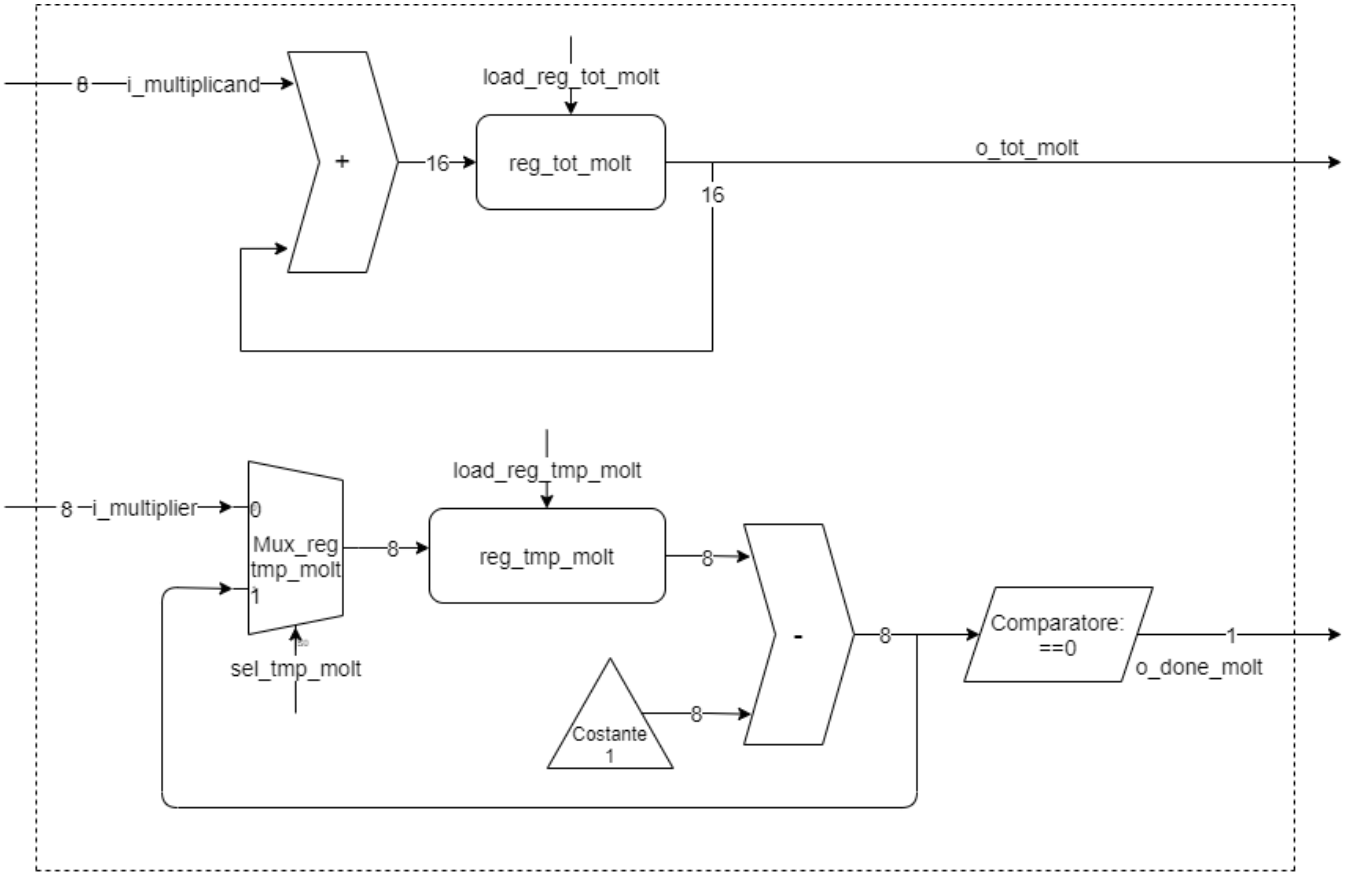

fig. 7

Somme ripetute

La parte rappresentata nella figura 7 si occupa di gestire le somme ripetute: a ogni ciclo di clock somma i_multiplier a quanto salvato in reg_tot_molt e memorizza questo risultato intermedio nello stesso registro fino quando load_tot_molt è a 1 (segnale che viene regolato dalla macchina a stati in base a come evolve l'altra parte). Per garantire che alla fine di questo calcolo nel registro in questione ci sia il risultato corretto, il valore di partenza memorizzato in reg_tot_molt è 0 grazie al segnale di i_rst / internal_reset.

Contatore

Questa parte si occupa di contare quante volte sia necessario fare la somma: a ogni ciclo di clock, viene decrementato il valore contenuto in reg_tmp_molt che all'inizio contiene il moltiplicatore; il circuito evolve fino a quando il valore nel registro non è a 1. Quando ciò avviene, o_done_molt, risultato del comparatore diventa 0 e, grazie all'intermediazione della macchina a stati, load_reg_molt passa a 0 così da bloccare le somme ripetute e, di conseguenza, tutto il processo della moltiplicazione.
Il mulitplexer mux_reg_tmp_molt è necessario perché prima di iniziare la moltiplicazione dobbiamo caricare il valore contenuto in i_multiplier, mentre durante lo svolgimento dell'operazione è necessario memorizzare nel registro il risultato proveniente dal sottrattore.

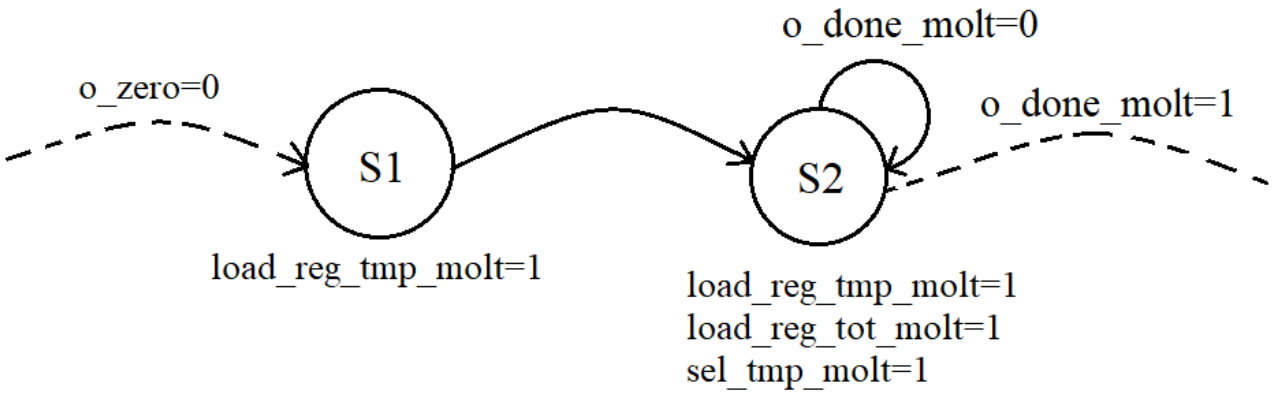

fig. 8

La macchina a stati che si occupa della preparazione e del calcolo della moltiplicazione si compone di S1 e S2: come già detto, in S1 salviamo all'interno di MOLTIPLICATOR0 il valore da cui decrementare, mentre in S2, fin tanto che o_done_molt è a 0, vengono caricati i due registri all'interno del componente perché la moltiplicazione deve procedere, quando il segnale è a 1 il risultato della moltiplicazione è pronto in reg_tot_molt. Quindi, le somme ripetute e il contatore evolvono in contemporanea.

Generazione degli indirizzi di memoria

A ogni ciclo di clock questa porzione di circuito genera su o_address un indirizzo di memoria, partendo dall'indirizzo dell'ultimo pixel dell'immagine e decrementando di volta in volta di una unità fino ad arrivare al primo pixel utile. L'indirizzo dell'ultimo pixel si ottiene sommando 1 al valore della moltiplicazione N_COL * N_RIGHE (gli indirizzi della memoria partono da 0). Il risultato di questa somma viene salvato in reg_address grazie al multiplexer mux_reg_address, necessario anche qui per differenziare il primo caricamento del registro dal caricamento a regime. Quindi, in reg_address vengono salvati i valori degli indirizzi da chiedere alla memoria: questi vanno in o_address attraversando il multiplexer mux_mem, la cui utilità verrà spiegata nella seconda parte del datapath (durante la prima parte della lettura sel_mem è settato a 0). Il decremento del valore degli indirizzi procede fino a quando in reg_address non troviamo il valore 1 perché a quell'indirizzo è presente il valore N_RIGHE; quando ciò accade, il comparatore causa la commutazione di o_read_done da 0 a 1.

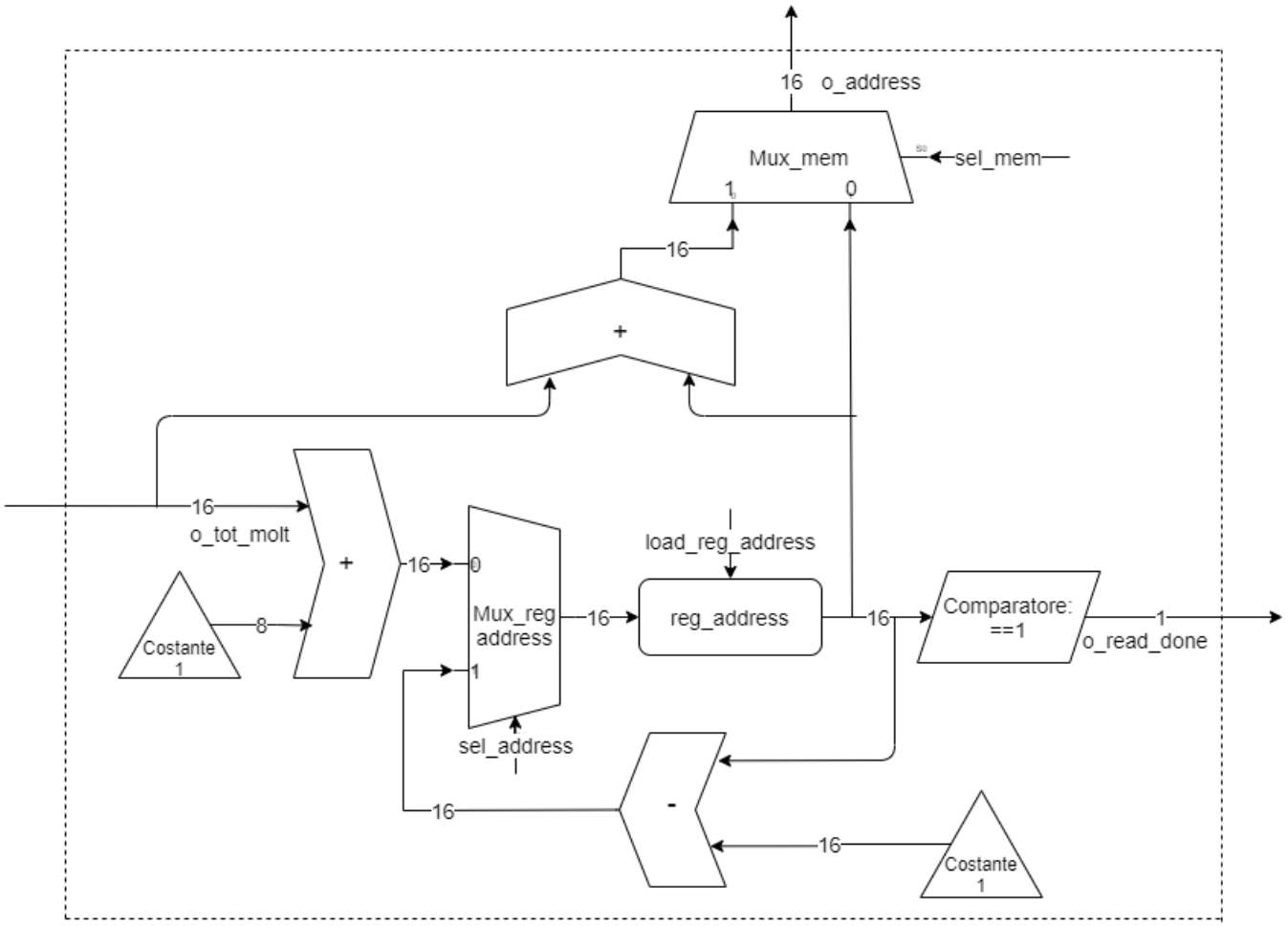

fig. 9

Poiché questa parte è strettamente connessa ai due successivi componenti, presenteremo la comune FSM dopo averli descritti.

Calcolo MAX_PIXEL_VALUE e calcolo MIN_PIXEL_VALUE

MAXVAL0 e MINVAL0 hanno entrambi la stessa struttura, differiscono solo per l'operatore di comparazione: maggiore e minore, rispettivamente. Il comportamento che realizzano è questo: a ogni ciclo di clock, confrontano il valore proveniente dalla memoria (i_data, che si mappa in i_value) con il valore salvato nel registro, e se questo rende vera l'uscita del comparatore (se i_value è maggiore del massimo finora trovato, deve diventare il nuovo massimo; se i_value è minore del minimo finora trovato, deve diventare il nuovo minimo) allora il segnale di caricamento del registro di load diventa 1 affinché si possa memorizzare questo dato.

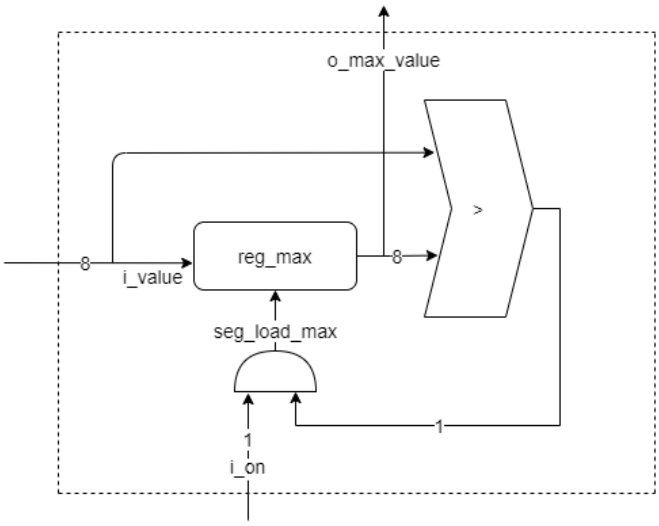

fig. 10

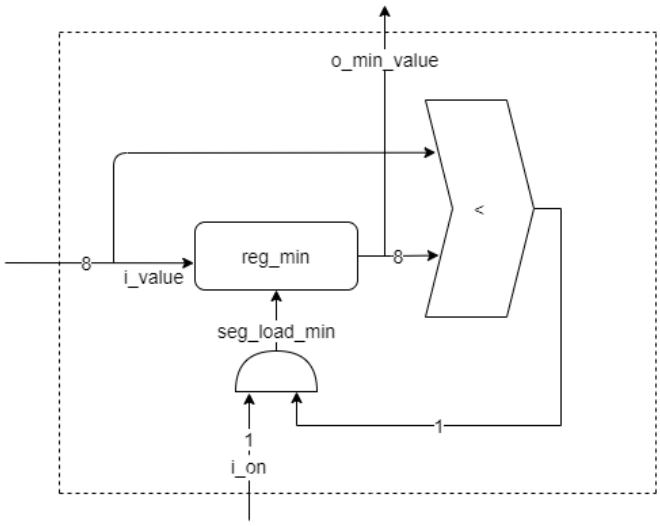

fig. 11

Per realizzare ciò è necessario operare due accorgimenti:

- È necessario precaricare nei registri dei valori e, poiché i dati sono compresi tra due estremi finiti (0 – 255), allora nel registro in MAXVAL0 carichiamo 0 e in quello di MINVAL0 255 garantendo così un corretto funzionamento del circuito;
- È necessario un segnale, i_on, proveniente dalla macchina a stati (lì si chiama sig_on_max_min), che "attivi" il circuito (diventando uguale a 1) rendendo il valore dei segnali di load pari all'uscita dei comparatori quando si è sicuri che in uscita dalla memoria ci siano effettivamente i pixel dell'immagine. Questo è necessario poiché su i_data compaiono anche i valori di righe e colonne che, altrimenti, verrebbero salvati come massimo o minimo, compromettendo in alcuni casi il comportamento del circuito. Terminata la lettura della memoria il booleano torna a zero, bloccando l'aggiornamento dei valori.

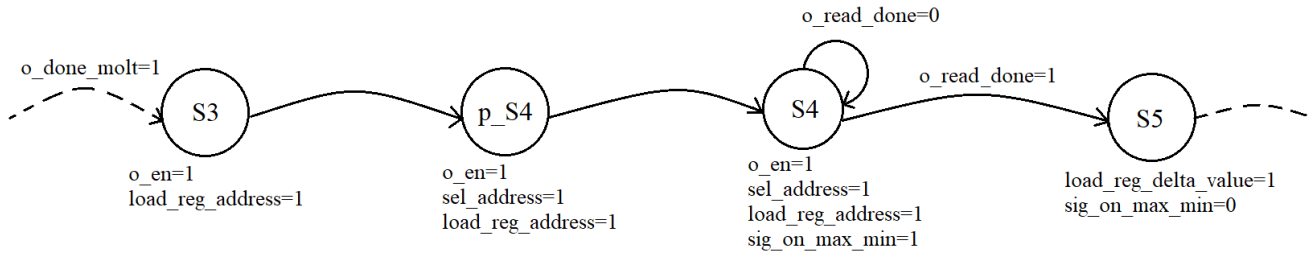

fig. 12

Nello stato S3 viene caricato in reg_address l'ultimo indirizzo dell'immagine (per come abbiamo scritto il codice, il segnale di selezione del multiplexer posto prima del registro è implicitamente a 0); in p_S4 il circuito chiede il dato contenuto in quella posizione grazie a o_en uguale a 1 (il pixel sarà disponibile al ciclo di clock successivo, S4) e viene caricato nel registro l'indirizzo precedente decrementato (load_reg_address è alto); in S4, fintanto che la lettura dell'immagine non è terminata, attiviamo i circuiti predisposti al calcolo di MAX_PIXEL e MIN_PIXEL (sig_on_max_min alto) e lasciamo che si aggiorni il valore in reg_address (anche load_reg_address è uguale a 1). È importante sottolineare l'ordine delle operazioni: in un ciclo di clock viene richiesto il k-esimo indirizzo alla memoria e, nel mentre, viene generato il (k-1)-esimo; al ciclo di clock successivo, il contenuto posto nella k-esima cella della memoria è disponibile (e viene processato da MAXVAL0 e MINVAL0) e su o_address è ora presente l'indirizzo (k-1)-esimo. Quando su reg_address è contenuto l'indirizzo 1 il comparatore genera su o_read_done 1 e il circuito ha così finito la lettura: nello stadio successivo a S4, S5, sig_on_max_min verrà posto uguale a 0 perché i valori di massimo e minimo sono stati calcolati.

Altra osservazione. La FSM che regola l'evoluzione dei componenti MAXVAL0 e MINVAL0 coincide con lo stato S4 in cui sig_on_max_min viene mantenuto alto per tutto il tempo in cui viene letta la memoria; perciò è come se, una volta che i due circuiti sono "attivi", questi evolvessero da soli, regolando autonomamente i valori dei load dei loro rispettivi registri. Abbiamo fatto questa scelta rispetto a un'altra possibilità in cui la FSM gioca un ruolo maggiore: in base al valore dell'uscita dei comparatori che vede nel ciclo di clock corrente, la macchina a stati setta il valore del load nel ciclo di clock successivo. La nostra soluzione, oltre a richiedere meno tempo, riduce la complessità della FSM e del circuito in generale (in particolare, diminuiscono i collegamenti tra FSM e

datapath). Affinché tutto funzioni correttamente, il valore di uscita del comparatore deve diventare stabile prima della fine del ciclo di clock: dai nostri test in post-sintesi, vediamo che questo accade (il circuito ha tempo sufficiente per evolvere: si veda nella parte c.a nel report timing il worst slack time).

Calcolo DELTA_VALUE e SHIFT_VALUE

Il circuito sottrae i valori di MAX_PIXEL e MIN_PIXEL rispettivamente salvati in reg_max e reg_min e li salva in nuovo registro, reg_delta_value. Questo entra nel componente SHIFT_VALUE0 che genera l'ultimo parametro necessario per l'equalizzazione, shift_value.

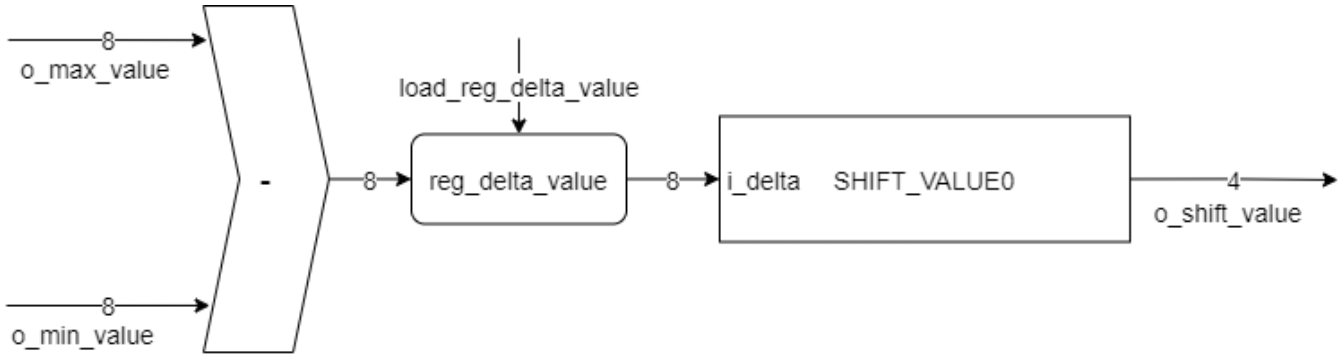

fig. 13

Abbiamo deciso di creare questo elemento circuitale come puramente combinatorio poiché i possibili valori in ingresso sono limitati: dall'algoritmo, l'argomento del logaritmo è DELTA_VALUE + 1 che può essere un numero compreso tra 1 e 256 (per questo è stata necessaria un'estensione di un bit a sinistra nella somma). Poiché bisogna calcolare l'approssimazione all'intero inferiore del logaritmo in base 2, più valori di DELTA_VALUE + 1 daranno il medesimo valore in uscita (per valori appartenenti alle medesime soglie, il risultato è lo stesso). In particolare, in tabella in figura 14 viene rappresentata questa situazione.

DELTA_VALUE	FLOOR(LOG2(DELTA_VALUE + 1))	SHIFT_VALUE
000 000 001	0	8
000 000 01-	1	7
000 000 1--	2	6
000 001 ---	3	5
000 01- ---	4	4
000 1-- ---	5	3
001 --- ---	6	2
01- --- ---	7	1
1-- --- ---	8	0

fig. 14

Pertanto, il valore dell'espressione dipende dalla posizione dell'bit più significativo posto a 1. Da qui la descrizione in codice di questa parte risulta molto simile a quella di un priority encoder (abbiamo deciso di implementarla con il costrutto condizionale if usando opportunamente l'operatore di slicing dei vettori). Come si vede dalla tabella, il valore dello shift è compreso tra 0 e 8: sono necessari solo 4 bit per rappresentarlo.
L'operazione di somma e il calcolo dello shift_value sono concorrenziali ma il clock dato è risultato più che sufficiente nei nostri test affinché tutti i segnali andassero a regime nel singolo periodo. Inoltre, poiché gli ingressi di questo componente non cambiano più per il resto dell'elaborazione, l'uscita o_shift_value rimarrà sempre la stessa, e per questo abbiamo deciso che non fosse necessario salvare questo valore in un registro. Di conseguenza la FSM che gestisce questo circuito corrisponde allo stato S5 (si veda fig. 12) in cui viene salvato DELTA_VALUE in reg_delta_value: nel giro di clock successivo (S6) shift_value si assesterà al suo valore definitivo.

ii. Seconda parte

Questa parte si occupa del calcolo del nuovo valore del pixel da salvare in memoria: ottenendo i dati dalla memoria come nella prima parte, li modifichiamo come l'algoritmo richiede e li salviamo nella loro posizione finale. Poiché questa sezione del circuito è molto interconnessa, la FSM corrispondente verrà presentata dopo la descrizione di tutti gli elementi fisici. Anche questa parte è logicamente suddivisa in parti, ciascuna delle quali con un compito specifico.

Attivazione circuito che genera gli indirizzi da ottenere e gestione futuro indirizzo

Avendo già un sotto-circuito che si occupa di generare e richiedere gli indirizzi della memoria (si veda b.i per la descrizione), abbiamo deciso di riutilizzarlo (quindi, anche in questa parte gli indirizzi vengono richiesti a partire dell'ultimo fino al primo). Ma, poiché non c'è solo la necessità di leggere dalla memoria ma anche di scrivere, in fase di progettazione sono state necessarie delle aggiunte. Dato che per ogni pixel salvato all'indirizzo i, il nuovo valore deve essere scritto all'indirizzo i + N_COL * N_RIGHE, il circuito modulerà i segnali di sel_mem e o_en per poter generare su o_address il giusto valore: quando viene richiesta la lettura della memoria, sel_mem sarà uguale a 0 (affinché o_address assuma il valore contenuto in reg_address, come nella prima parte) e o_en a 1 e o_we a 0 (in modo da leggere il dato dalla memoria); quando il circuito presenterà su o_data il nuovo dato da salvare in memoria, i segnali saranno sel_mem pari a 1, o_en e o_we pari a 1 in modo da scrivere il nuovo valore all'indirizzo o_address pari alla somma tra l'ultimo indirizzo letto e la dimensione dell'immagine (N_COL * N_RIGHE).

Calcolo TMP_PIXEL

Per calcolare TMP_PIXEL associato al valore corrente proveniente dalla memoria, abbiamo costruito il componente TMP_VAL0: prima il circuito calcola la differenza tra i_value e il contenuto di reg_min (che viene qui mappato in i_min), poi il risultato della differenza, sub_tmp, subisce lo shift di SHIFT_VALUE posizioni.

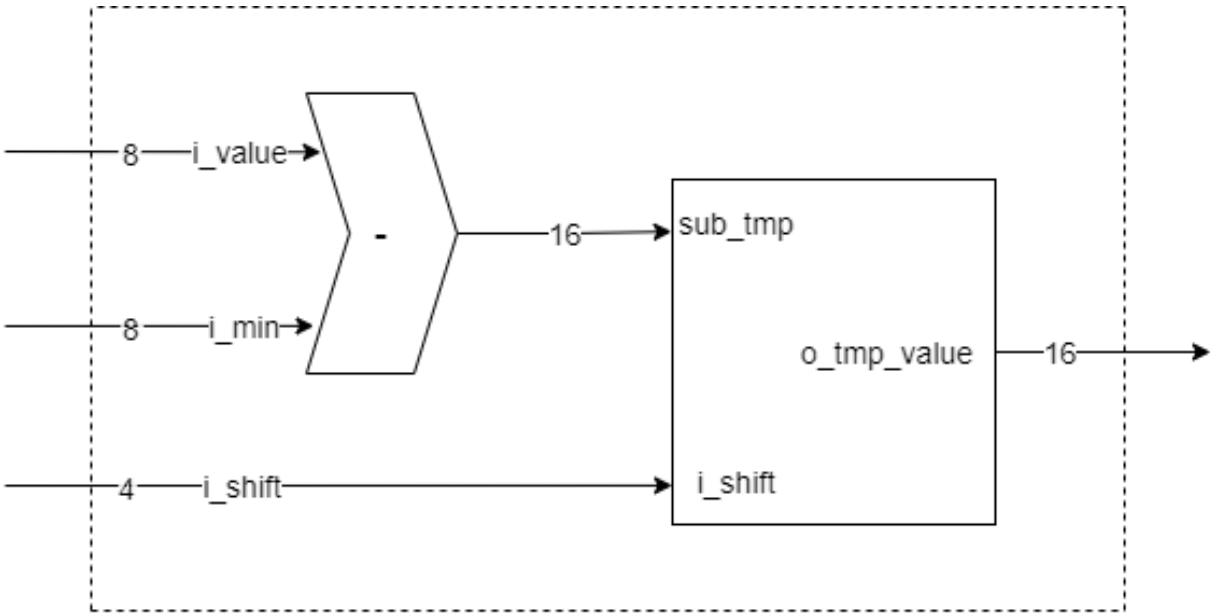

fig. 15

Questo avviene tramite un circuito puramente combinatorio poiché, anche qui, tutti i dati in ingresso hanno una dimensione finita (sub_tmp è su 16 bit e i_shift, che contiene SHIFT_VALUE, su 4): in un assegnamento condizionale, a seconda del valore di shift_value, assegniamo a o_tmp_value il valore di sub_tmp opportunamente troncato ed esteso a sinistra con gli zero necessari (come si vede in figura 15, da notare che noi abbiamo prima esteso la differenza su 16 bit).

Selezione del NEW_PIXEL_VALUE

Per la scelta del nuovo valore del pixel, abbiamo creato un componente NEW_VALUE0: l'uscita del componente TMP_VALUE (quindi TMP_PIXEL associato all'attuale dato proveniente dalla memoria) viene mappata qui dentro come i_tmp; il risultato della comparazione tra i_tmp e 255 (se i_tmp > 255, viene restituito 1, 0 altrimenti) diventa il segnale di selezione del multiplexer, sel_mux_new; gli altri ingressi sono posti coerentemente con i valori di quest'ultimo segnale. Le dimensioni di questi segnali sono interessanti: 255 viene scritto su 16 bit per poterlo confrontare con i_tmp; i due ingressi vettoriali di mux_new_value sono entrambi su 8 bit poiché l'uscita di questo multiplexer è su 8 bit (da notare che nel caso in cui risulti che i_tmp sia minore di 255, i suoi 8 bit più significativi sono tutti uguali a 0: quindi, troncando non si perde informazione).

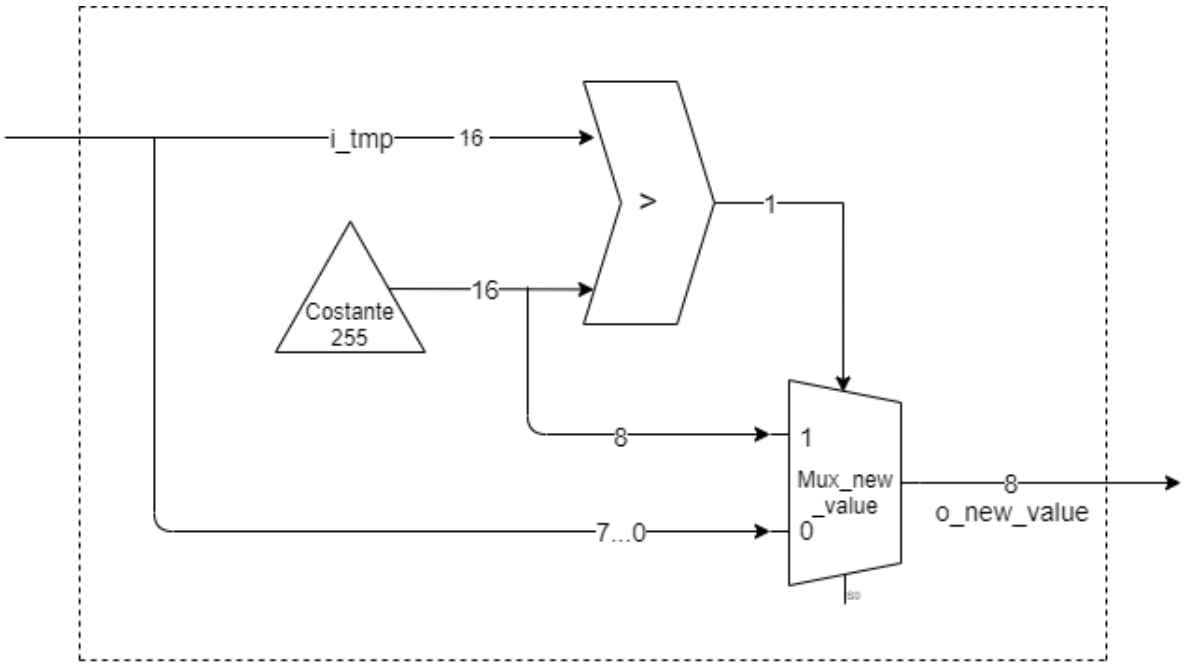

fig. 16

Anche qui, come nel caso di MAX_VAL0 e MIN_VAL0, abbiamo lasciato che il circuito evolvesse da solo, ossia settasse i suoi segnali interni di selezione senza l'intervento della macchina a stati: questa scelta è stata guidata da un desiderio di semplicità nella FSM e di risparmio di tempo.

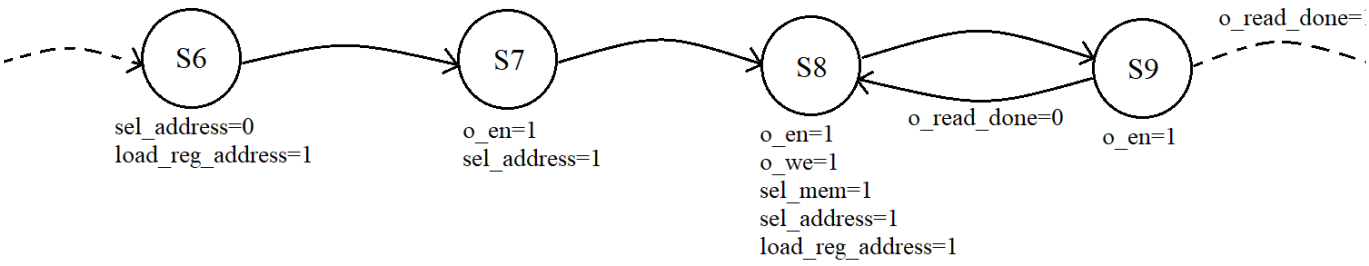

fig. 17

La FSM che gestisce l'evoluzione di questa seconda parte è composta dai seguenti stati:

- S6: SHIFT_VALUE si assesta al suo valore definitivo (ha un comportamento asincrono rispetto al clock) e tramite il suo segnale di load e sel_address, viene caricato in o_reg_address l'indirizzo dell'ultimo pixel (come in S3);
- S7: è uno stato d'appoggio che ci serve per chiedere il primo dato alla memoria, il quale arriverà nello stato successivo;

- S8: ora che il dato è presente su i_data, il circuito calcola TMP_PIXEL e NEW_PIXEL, genera l'indirizzo in cui salvare il dato (sel_mem = 1 e o_we = 1 per comunicare alla memoria la necessità di salvarlo), prepara l'indirizzo successivo;
- S9: il circuito chiede il dato successivo da modificare (sel_mem è implicitamente a 0).

Da quanto si vede, si alternano due stati: uno di scrittura della memoria (S8) e uno di lettura della memoria(S9), fino all'esaurimento dei pixel dell'immagine. Inoltre, l'elaborazione finisce quando o_read_done è uguale a 1: tutti i dati in memoria sono stati modificati e l'immagine equalizzata è stata opportunamente salvata.

c. Disegni complessivi

Il circuito descritto finora è qui rappresentato aggregando i singoli componenti.

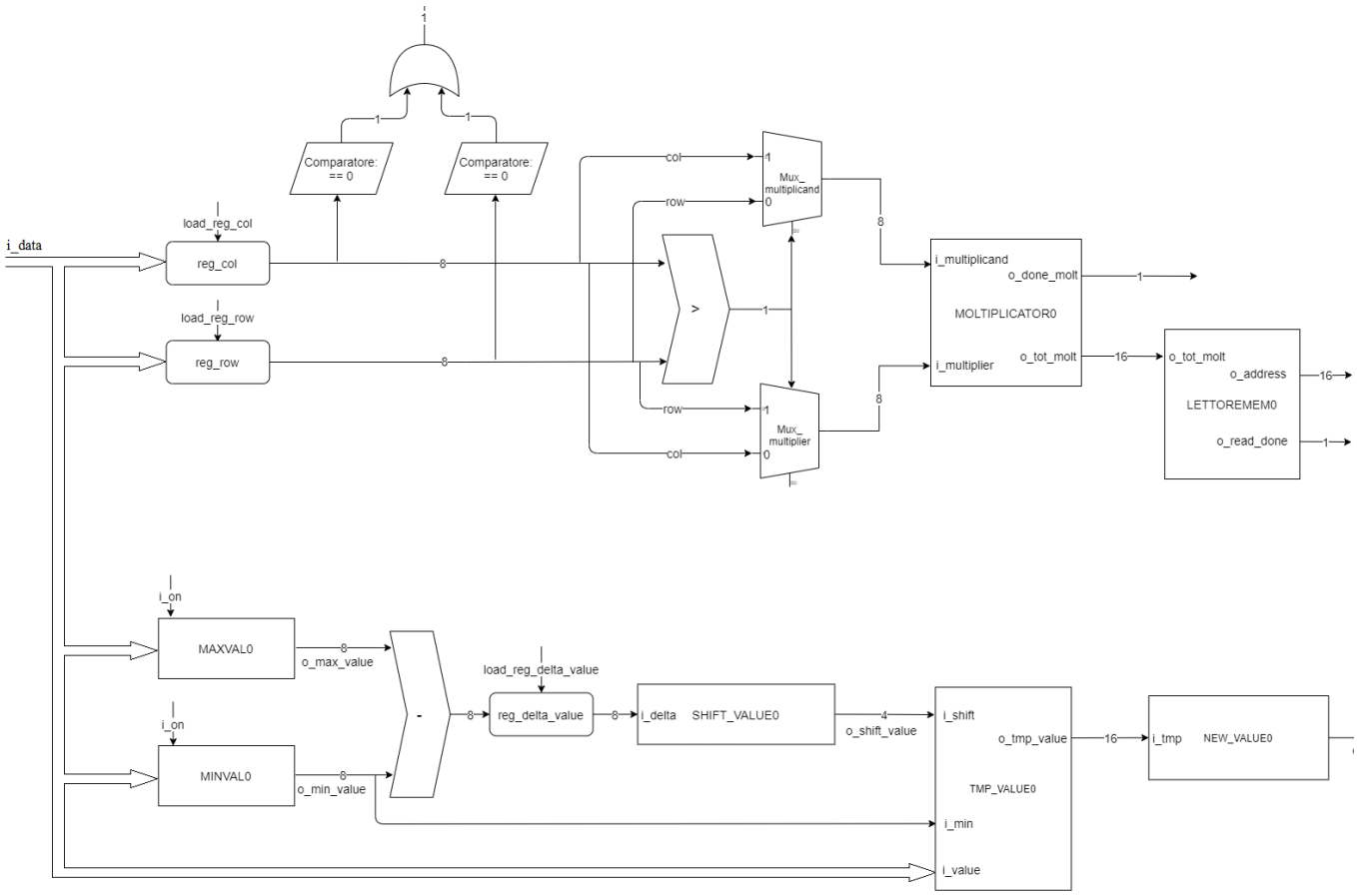

fig. 18

La macchina a stati complessiva è la seguente:

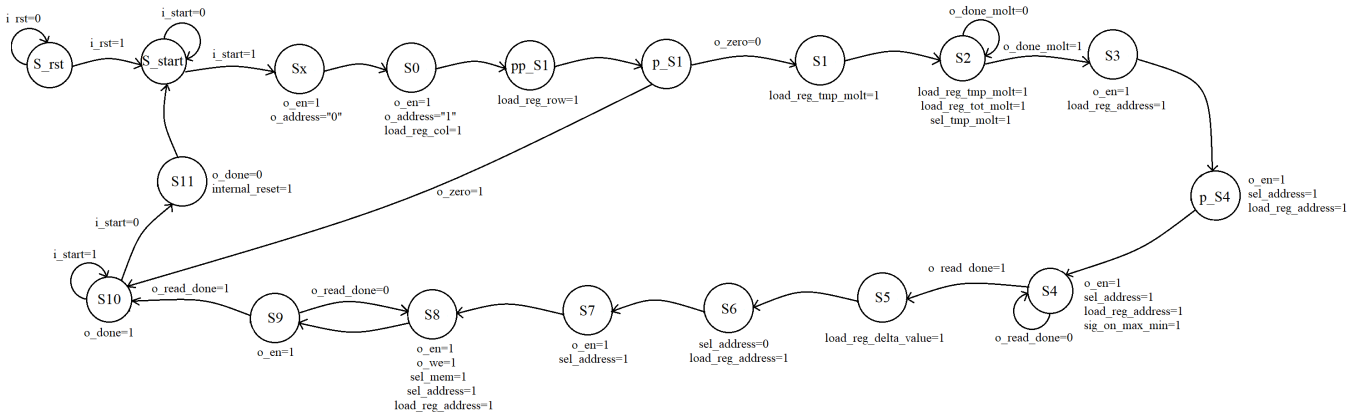

fig. 19

3. Risultati sperimentali

a. Report di sintesi

Abbiamo sintetizzato il nostro circuito ottenendo questi principali risultati di sintesi:

Site Type	Used	Fixed	Available	Util%
Slice LUTs*	171	0	134600	0.13
LUT as Logic	171	0	134600	0.13
LUT as Memory	0	0	46200	0.00
Slice Registers	97	0	269200	0.04
Register as Flip Flop	97	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

fig. 20

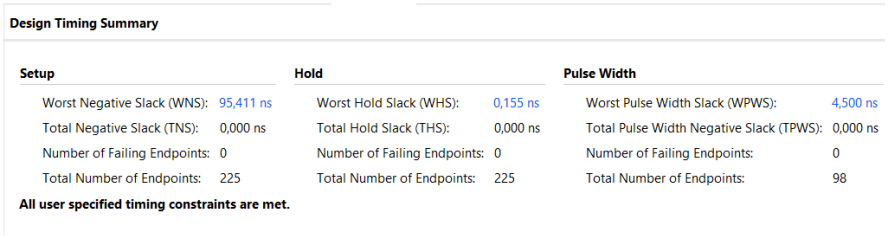

fig. 21

Da quanto si può evincere da queste figure, Vivado non ha inferito alcun latch garantendoci così una sintesi coerente con le nostre aspettative. Inoltre, il Worst Negative Slack è risultato essere di circa 95ns: solo nel 5% del periodo di clock il circuito effettivamente lavora per portare i vari segnali a regime (quindi, il circuito potrebbe lavorare a frequenze di clock superiori a quella richiesta).

b. Testbench

Abbiamo testato il nostro circuito con vari testbench al fine di testare tutte le funzionalità (abbiamo testato che il circuito percorresse tutti gli archi possibili della nostra FSM). I test più significativi vengono qui descritti con i vari corner case e le rispettive parti sollecitate.

Testbench forniti

Questi testbench sono stati significativi per controllare che il comportamento dei due componenti MAXVAL0 e MINVAL0 fosse corretto, soprattutto nel caso in cui i valori definitivi di MAX_PIXEL_VALUE e MIN_PIXEL_VALUE corrispondessero ai primi e ultimi valori provenienti dalla memoria.

In particolare,

- Testbench 1: il massimo e minimo si aggiornano almeno una volta;
- Testbench 2 e 3: aggiornano il registro del valore massimo alla prima iterazione (il nostro circuito è stato progettato in modo tale che il primo dato proveniente dalla memoria è quello corrispondente all'ultima cella utile) e l'ultimo dato su i_data diventa il MIN_PIXEL_VALUE. In particolare, il testbench 3 aggiorna continuamente il valore del minimo (si veda figura 22) e, inserendo i dati in memoria in ordine opposto, abbiamo testato anche il caso in cui il massimo viene aggiornato continuamente. Per la particolare distribuzione dei dati, questo testbench risulta quello in cui si ha una più significativa applicazione dell'algoritmo di equalizzazione dell'istogramma;
- Testbench 4 e 5: sono simili ai precedenti con la differenza che l'aggiornamento non avviene in maniera continuativa (il testbench 5 è stato testato anche posizionando i dati in memoria in ordine opposto).

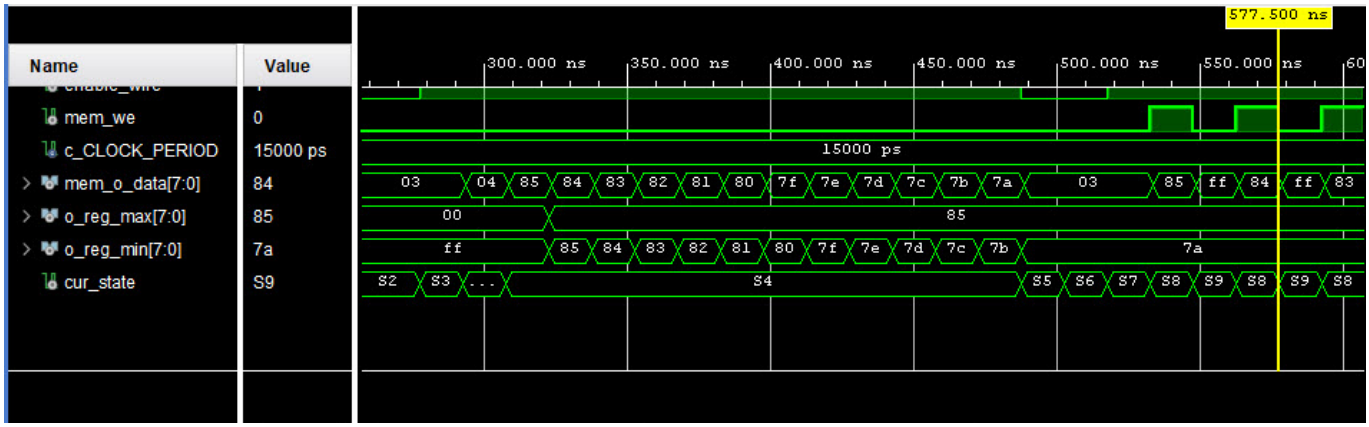

fig. 22

Casi di test per la moltiplicazione

Il componente MOLTIPLICATOR0 riveste un ruolo fondamentale per il nostro circuito in quanto se fornisse un valore sbagliato tutta l'elaborazione successiva sarebbe compromessa. Abbiamo fatto le seguenti verifiche:

- Settaggio corretto di moltiplicando e moltiplicatore: abbiamo verificato che nei tre modi in cui le dimensioni delle immagini si potessero trovare (reg_col > reg_row, reg_col < reg_row, reg_col = reg_row) il circuito non presentasse errori e garantisse la più veloce moltiplicazione possibile. Il funzionamento del circuito nel caso in cui NUM_COL sia minore di NUM_RIGHE è mostrato nella figura seguente.

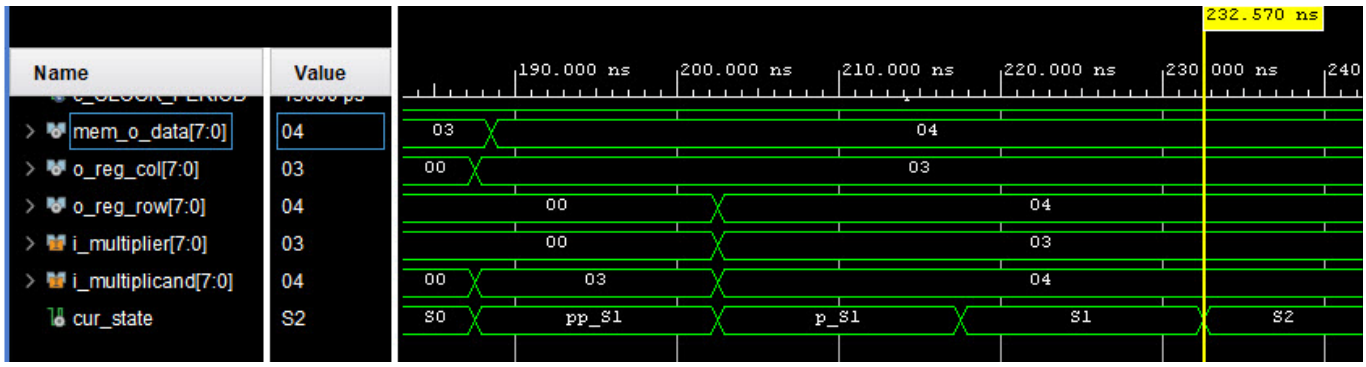

fig. 23

- Immagini di piccole dimensioni: abbiamo testato casi in cui come input si avessero immagini dalle dimensioni [1 1] e [1 2] perché in questi casi il componente MOLTIPLICATOR0 non deve iterare (lo stato S2 viene visitato solo una volta, come si può vedere nell'immagine successiva).

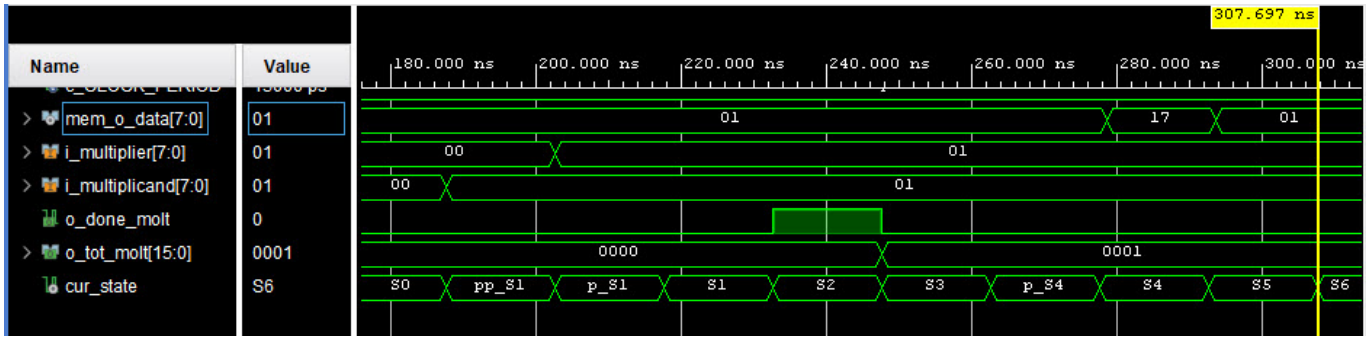

fig. 24

- Immagini di grandi dimensioni: in input abbiamo utilizzato un'immagine con la massima dimensione possibile, [128 128] (generando automaticamente il codice del testbench con Matlab), e il circuito ha elaborato correttamente tutti i pixel dell'immagine.

Immagine degenere

Abbiamo testato il comportamento in presenza di immagini degeneri (ossia che il circuito evolvesse dallo stato p_S1 allo stato S10) del tipo [0 X], [X 0] o [0 0]. Un esempio di quest'ultimo caso è riportato qua sotto.

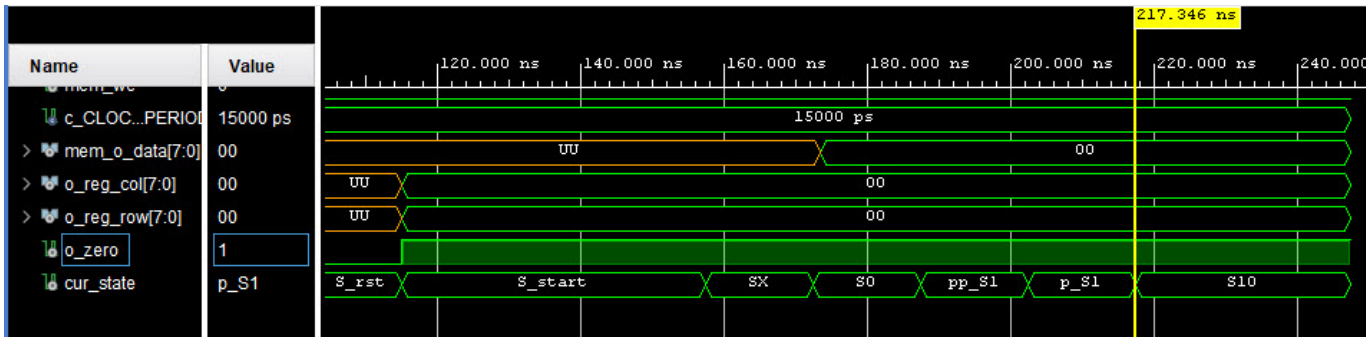

fig. 25

Si può notare che, per come abbiamo disegnato il circuito, il segnale di o_zero va a 1 quando diamo il segnale di reset e rimane a tale valore perché le dimensioni non sono valide.

l_rst

Abbiamo testato il caso in cui il circuito ricevesse il segnale di i_rst in un momento qualsiasi della computazione e il circuito ha mostrato un comportamento corretto.

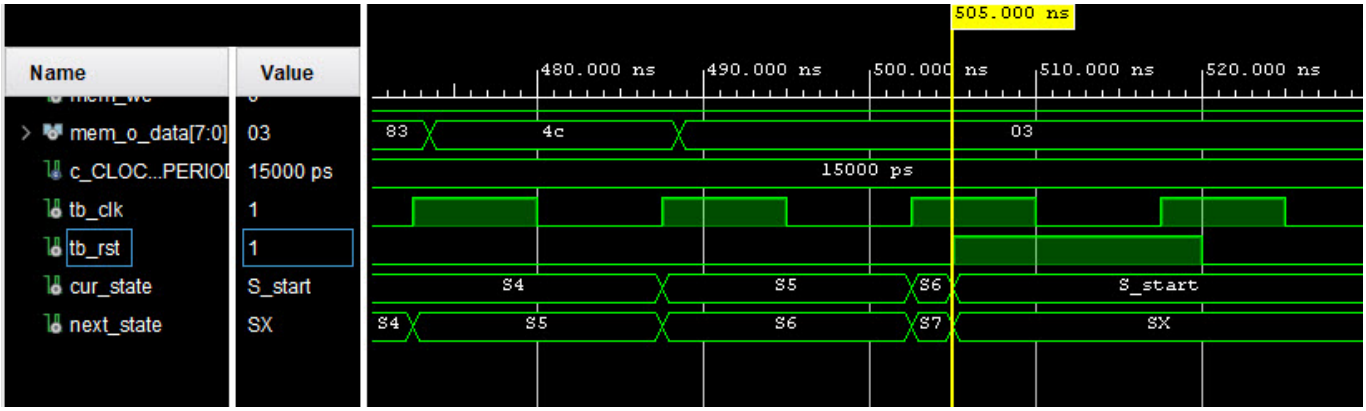

fig. 26

Il circuito ha superato tutti i test in Behavioral, Post-Synthesis e Post-Implementation Simulation.

4. Conclusioni

Nella fase di progettazione abbiamo optato per una elevata modularizzazione che ci ha permesso di ottenere un circuito più facilmente comprensibile dove i vari componenti logici svolgono una specifica funzione, dandoci anche la possibilità di riutilizzare opportunamente la parte di generazione degli indirizzi in fasi successive. Per fare questo, abbiamo diviso il flusso di lavoro richiesto al circuito come ci veniva più naturale dalla lettura della richiesta utilizzando un approccio top-down: diminuendo a iterazioni successive la granularità delle operazioni abbiamo ottenuto i nostri moduli fondamentali.
Questa modularità si è tradotta in una FSM lineare, fatta anch'essa a moduli ben distinti rispecchianti le varie funzionalità del datapath. Questo approccio ci ha profondamente semplificato la progettazione, la scrittura in codice VHDL e nella successiva fase di debugging.

23