Prova Finale (Progetto di Reti Logiche)

Palazzoli Matteo
Codice Persona: 10614119
Matricola: 907397

a.a. 2020/2021

POLITECNICO
MILANO 1863

Indice

1 Introduzione

1.1 Obbiettivo generale

Lo scopo di questo progetto era di descrivere, sintetizzare e implementare su un FPGA (*Field Programmable Gate Array* - un circuito integrato programmabile) una versione semplificata dell'algoritmo di equalizzazione dell'istogramma di un'immagine. L'algoritmo in questione ridistribuisce i vari valori dei pixel su tutto l'intervallo ammissibile, per fare in modo che se ne aumenti il contrasto. Il modulo opera su immagini in scala di grigi a 256 livelli, con dimensioni massime di 128x128 pixel. La descrizione dell'hardware e del suo comportamento è stata scritta nel linguaggio VHDL, mentre la sintesi e l'implementazione sono state a carico del software *Vivado Design Suite* di Xilinx. Il processo di trasformazione dei pixel è il seguente per ognuno di essi. Supponiamo di conoscere il massimo e il minimo pixel p_M e p_m. Calcoliamo la loro differenza D e il valore di shift S così:

$$D = p_M - p_m$$

$$S = 8 - \lfloor log_2(D+1) \rfloor$$

Il valore equalizzato di un pixel p_e in funzione del pixel p si calcola quindi come:

$$p_e(p) = Min(255, (p - p_m) \ll S)$$

nella quale il simbolo $\ll$ indica l'operazione di shift a sinistra. Nelle Tabelle 1 e 2 si può vedere un esempio di come l'algoritmo elabora un'immagine 3x3: la prima ha valori di grigio tutti molto vicini fra loro, mentre ha seconda ha valori distribuiti dal minimo al massimo.

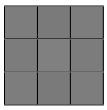

Tabella 1: Immagine originale

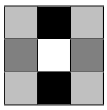

Tabella 2: Immagine equalizzata

1.2 Descrizione del componente

Il componente si chiama "`project_reti_logiche`" e la sua interfaccia è illustrata in Figura 1. I segnali di input hanno questo significato:

- `i_clk` è il segnale di clock del componente. Il componente deve poter funzionare con un periodo di clock inferiore a 100ns.
- `i_rst` è il segnale di reset della macchina. Esso porta il componente ad uno stato iniziale pronto a elaborare un'immagine a partire dal successivo ciclo di clock.

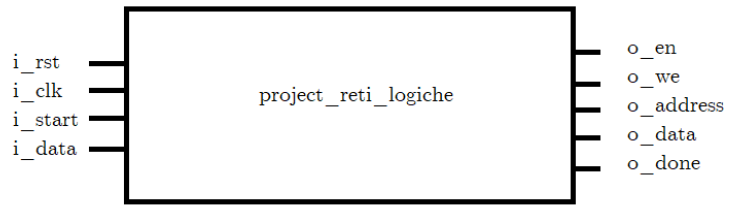

Figura 1: Entity di project_reti_logiche

- `i_start` è il segnale di start che viene portato a 1 dalla RAM per iniziare un'elaborazione. Esso rimane a 1 per tutto il tempo di lavoro, e non verrà portato a 0 prima dell'imposizione del segnale `o_done` a 1 da parte del componente. Solo da allora potrà tonare a 0 e, nel caso, tornare poi a 1 per una nuova elaborazione.
- `i_data` è un segnale a 8 bit che comunica il contenuto di una cella di RAM al componente. L'indirizzo della cella è quello precedentemente specificato in `o_address`.
- `o_en` è un segnale a un bit che, se posto a 1, abilita la comunicazione con la RAM, che non verrebbe abilitata altrimenti.
- `o_we` è un segnale a un bit che viene preso in considerazione dalla RAM solo se `o_en` è posto a 1. Se anche `o_we` posto a 1 vuol dire che il contenuto di `o_data` deve essere scritto nella cella di indirizzo `o_address`.
- `o_address` è un segnale a 16 bit che indica l'indirizzo da cui leggere o scrivere un dato.
- `o_data` è un segnale a 8 bit che contiene il dato da scrivere in RAM. Viene preso in considerazione da essa solo se `o_we` è posto a 1.
- `o_done` è un segnale a un bit che viene posto a 1 dopo che l'elaborazione ha terminato, per segnalarlo alla RAM. Può essere posto a 0 solo dopo che `i_start` è anch'esso riabbassato a 0.

1.3 Descrizione della memoria

La memoria RAM è modellizzata con un'array di 65536 celle da 8 bit ognuna dotata di un indirizzo. La prima cella ha indirizzo 0. Al fronte di salita del clock, la RAM controlla se il segnale `o_en` è 1. Se lo è, fa un controllo analogo per `o_we`. Nel caso in cui quest'ultimo sia 0, si occuperà di mandare in uscita (ovvero in `i_data`) il dato specificato da `o_address`. Altrimenti provvederà a

scrivere nella cella di quell'indirizzo il dato proveniente da `i_data` e lo stesso dato verrà anche rimandato in uscita. L'immagine da analizzare viene salvata nella RAM con la seguente convenzione: nel primo indirizzo (indirizzo 0) viene salvato il numero di righe dell'immagine, nell'indirizzo 1 il numero di colonne, mentre a seguire vengono salvati tutti i pixel uno dopo l'altro. Il componente dovrà scrivere in RAM i pixel dell'immagine equalizzata a partire dalla prima cella libera, mantenendo l'ordine. Ad esempio, poniamo che ci sia un'immagine da equalizzare di 2x2 pixel. L'indirizzo 0 e 1 contengono rispettivamente la quantità di righe e colonne dell'immagine. Gli indirizzi 2, 3, 4, e 5 contengono i pixel. I pixel del'immagine risultante verranno scritti nelle celle di indirizzo 6, 7, 8, e 9. Nelle Figure 2 e 3 è descritto lo stato della RAM rispettivamente prima e dopo l'elaborazione.

INDRIZZO	DATO
0	Numero di righe R
1	Numero di colonne C
2	Pixel numero 1
3	Pixel numero 2
...	...
...	...
R*C+1	Pixel numero R*C
R*C+2	0
...	0
...	...
...	...
65535	0

Figura 2: Stato della RAM prima dell'elaborazione

INDRIZZO	DATO
0	Numero di righe R
1	Numero di colonne C
2	Pixel numero 1
3	Pixel numero 2
...	...
R*C+1	Pixel numero R*C
R*C+2	Pixel elaborato 1
R*C+3	Pixel elaborato 2
...	...
2*R*C+1	Pixel elaborato R*C
2*R*C+2	0
...	...
65535	0

Figura 3: Stato della RAM dopo l'elaborazione

2 Architettura

Il componente è formato da un singolo modulo che, tramite due processi, implementa la parte di registri e la macchina a stati di Figura 4.

2.1 Registri

I registri sono descritti dal *process* `registers` e sono 15: 9 per segnali interni (righe R, colonne C, indirizzo R*C+2, massimo, minimo, delta, shift, indirizzo di lettura e di lavoro), 5 per le uscite e uno per lo stato.

2.2 Macchina a stati

La macchina a stati di Figura 4, invece, è descritta dal *process* `lambda_delta` che si occupa sia di elaborare i segnali interni, sia di stabilire le uscite prossime. Ci sono 10 stati. Essi sono:

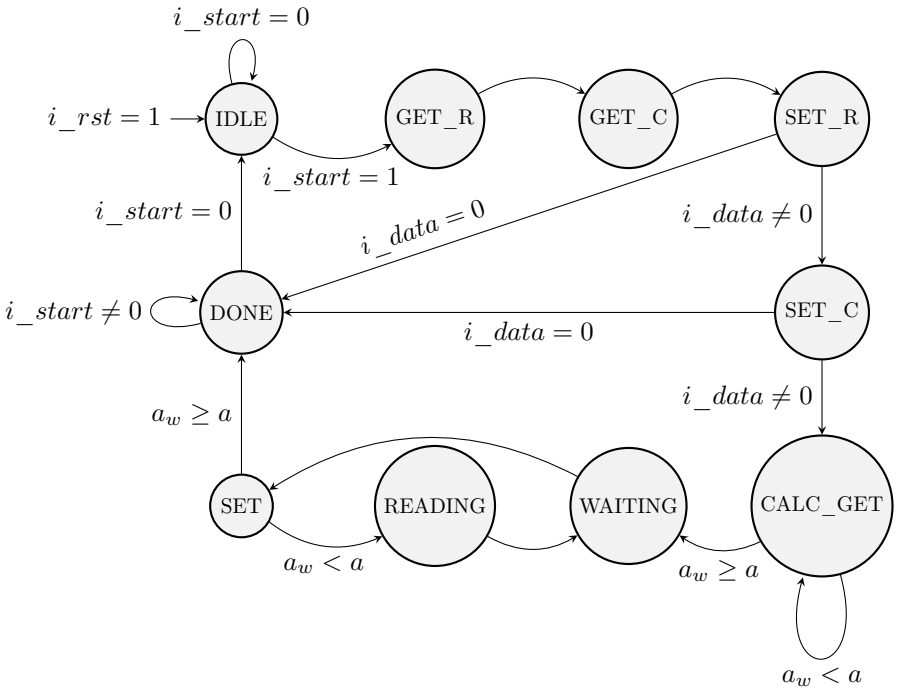

Figura 4: La macchina a stati implementata. Le abbreviazioni a_w e a si riferiscono rispettivamente all'indirizzo del dato elaborato nel ciclo corrente e all'indirizzo $R * C + 2$. (prima cella libera nello stato iniziale della RAM.)

1. IDLE: la macchina entra in questo stato quando `i_rst` è posto a 1. Qui aspetta fino a quando `i_start` viene posto a 1. Se ciò si verifica, la macchina passa nello stato GET_R.

2. GET_R: in questo stato la macchina imposta l'uscita in modo tale da leggere il numero di righe. Dopodiché passa allo stato GET_C.

3. GET_C: analogamente, qui viene posta l'uscita in modo da leggere il numero di colonne dell'immagine. La macchina passa poi allo stato SET_R.

4. SET_R: a questo punto la macchina è pronta per salvare il valore resituito dalla RAM corrispondente al numero di righe. Se esso è 0 l'elaborazione finisce e si passa in DONE. Altrimenti Lo salva e, a prescindere dal numero totale di pixel presenti, imposta l'uscita per leggere l'indirizzo del primo di essi (indirizzo 2). Poi passa a SET_C.

5. SET_C: analogamente viene ricevuto il numero di colonne. Se è 0 si va in DONE, altrimenti viene salvato e viene calcolato il valore corrispondente al primo indirizzo libero (ovvero $R * C + 2$). Imposta l'uscita per leggere il secondo pixel (anche se non esiste) e passa allo stato CALC_GET.

6. CALC_GET: in questo stato la macchina riceve un valore corrispondente ad un pixel (la prima volta sarà il primo e così via), calcola i valori parziali del massimo, del minimo, di delta e del valore di shift, e si prepara a leggere il pixel dopo. Una volta processati tutti i valori chiede in lettura il primo pixel e va in WAITING.

7. WAITING: la macchina ora aspetta un ciclo di clock in cui si fa restituire un valore non significativo per l'algoritmo, e nel frattempo imposta l'uscita per leggere il secondo pixel. Lo stato successivo è SET.

8. SET: scopo di questo stato è ricevere un pixel (la prima volta il primo e così via) e di calcolarne il pixel risultante usando i valori del minimo e dello shift salvati. L'uscita è impostata per scrivere questo valore nell'indirizzo corrispondente della RAM. Se rimangono pixel da elaborare torna in READING, altrimenti va in DONE.

9. READING: qui la macchina imposta l'uscita per leggere il pixel successivo e poi va in WAITING.

10. DONE: in quest'ultimo stato la macchina alza il valore di `o_done` a 1 fino a quando `i_start` torna a 0. Dopodiché abbassa `o_done` e torna in IDLE.

In qualsiasi stato si trovi la macchina, alla ricezione di `i_rst`=1 essa torna nello stato IDLE in modo asincrono.

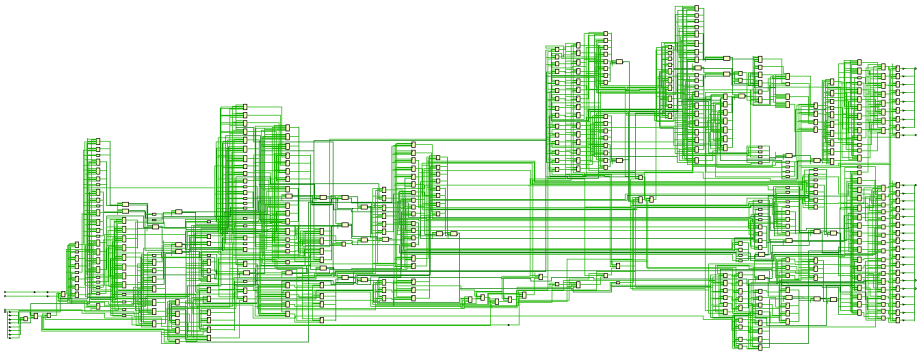

Figura 5: Schematic del modulo sintetizzato

3 Risultati sperimentali

3.1 Sintesi

La sintesi e l'implementazione del modulo sono state operate dal software *Vivado 2020.2* su FPGA `xc7k70tfbg484-1`. Nella Tabella 3 si possono vedere le quantità dei componenti base utilizzati (*Lookup table*, *flip-flop*, *input/output*) e la loro percentuale rispetto al totale presente. Come da specifica, il modulo può funzionare con un periodo di clock minore di 100ns. Il *Timing Report* del tool utilizzato, infatti, riporta il valore del **Worst Negative Slack** (WNS), ovvero il valore di tempo che intercorre dalla stabilizzazione dei segnali fino alla fine del ciclo di clock, di 94.02ns. Possiamo quindi dedurre che il tempo di stabilizzazione dei segnali è di $100 - WNS = 100 - 94.02 = 5.98ns$. Nella Figura 5 si può vedere lo *Schematic* realizzato dal tool di sintesi.

Risorse	Quantità	Percentuale utilizzata
LUT	248	0.60%
FF	107	0.13%
IO	38	13.33%

Tabella 3: Tabella dei componenti utilizzati.

3.2 Simulazioni

Per verificare il corretto funzionamento del modulo sono stati condotti alcuni test, simulando gli input e la RAM attraverso dei testbench. Tutti i test riportati sono stati superati con successo dal modulo.

3.2.1 Testbench base

Il testbench base serviva per verificare il corretto funzionamento del modulo a fronte di una normale immagine senza particolari condizioni limite. Il risultato è visibile in Figura 6.

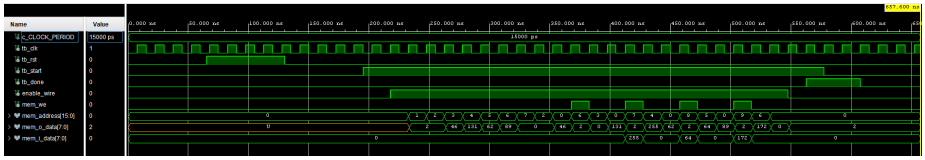

Figura 6: Forme d'onda del testbench base

3.2.2 Zero pixel

Nel caso di un'immagine da zero pixel è stato supposto che la RAM non debba subire modifiche in quanto non ci sono pixel da elaborare. Questo caso è gestito dal modulo anche in condizioni impossibili, come nel caso di immagine con solamente una delle due dimensioni pari a zero (ad esempio 23 righe e 0 colonne o viceversa). Il risultato del testbench è visibile in Figura 7.

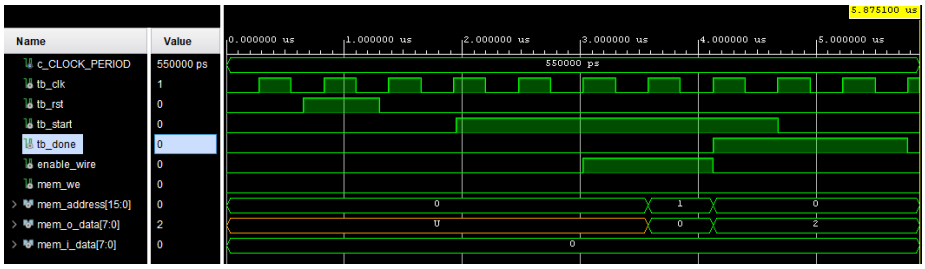

Figura 7: Forme d'onda del testbench Zero Pixel

3.2.3 Singolo pixel

Il caso di un'immagine da singolo pixel è stato testato perché considerato un caso limite, infatti appena inizia l'elaborazione deve finire subito dopo. I risultati sono mostrati in Figura 8.

3.2.4 Pixel tutti uguali

In questo test i pixel dell'immagine fornita sono tutti uguali (per la precisione tutti "1"). Anche questo è considerato un caso limite perché il *delta* è 0. L'immagine equalizzata secondo l'algoritmo implementato deve avere tutti "0". Il risultato è in Figura 9.

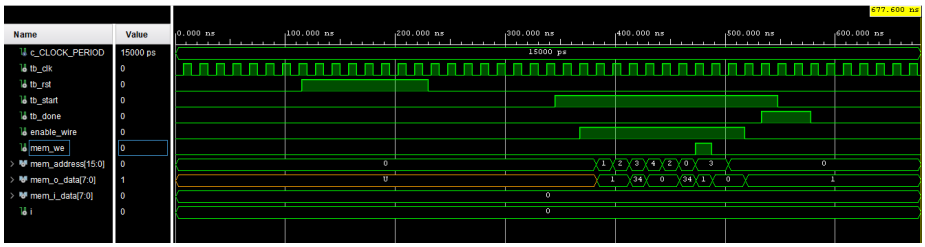

Figura 8: Forme d'onda del testbench Singolo Pixel

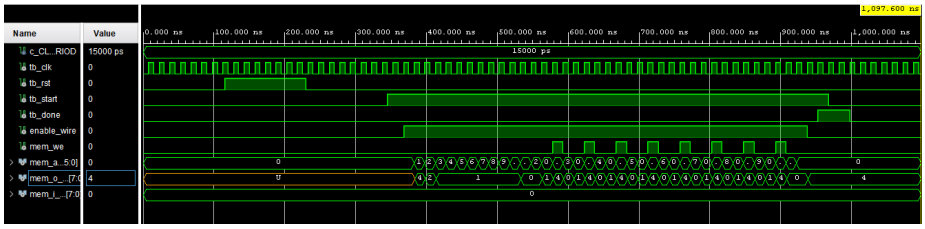

Figura 9: Forme d'onda del testbench Tutti 1

3.2.5 Molteplici immagini

In questo test si verifica il funzionamento del modulo per più immagini consecutive senza reset: appena finisce una codifica, `o_done` si alza, `i_start` si abbassa, i valori della RAM vengono modificati e `i_start` si alza nuovamente. Questo test, per la precisione, sottopone al modulo 3 immagini ed il risultato è visibile in Figura 10.

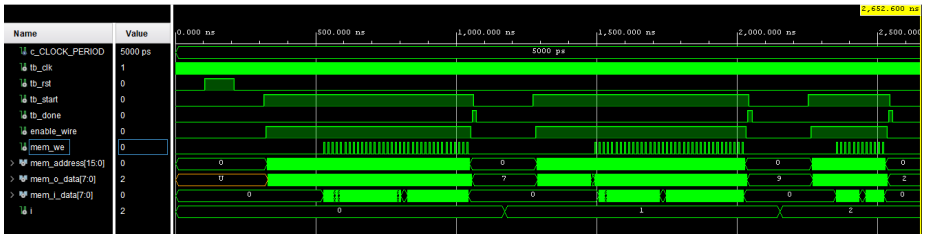

Figura 10: Forme d'onda del testbench 3 Immagini

3.2.6 Reset Asincrono

In questo test, mentre il modulo elabora l'immagine, viene alzato il segnale di reset in modo asincrono. Il comportamento aspettato consiste nel far ripartire l'equalizzazione da zero, come se fosse appena partita. Le forme d'onda sono visibili in Figura 11.

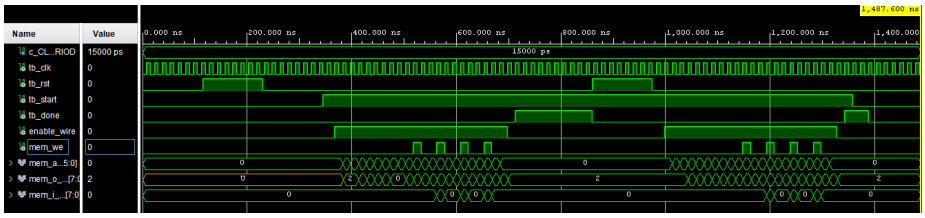

Figura 11: Forme d'onda del testbench Reset Asincrono

3.2.7 Resistenza

Questo test metteva alla prova il componene con 50 immagini consecutive di dimensione variabile fino a 128x128, con possibilità di reset asincrono. Lo scopo è di testare l'affidabilità a fronte di un carico pesante di elaborazioni. Le forme d'onda dall'inizio alla fine del test sono riportate in Figura 12.

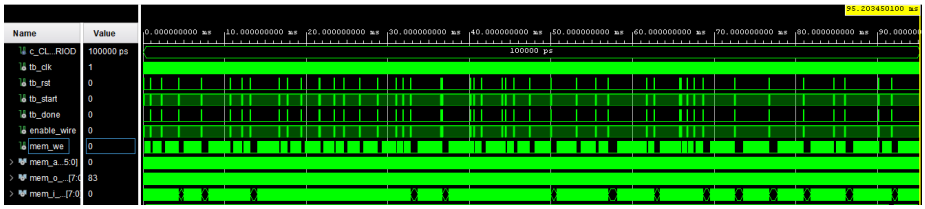

Figura 12: Forme d'onda del testbench Resistenza

4 Conclusioni

Il componente supera tutti i test nelle simulazioni *Behavioral*, *Post-Synthesis Functional*, e *Post-Synthesis Timing* rispettando quindi la specifica.

4.1 Ottimizzazioni

Alcune ottimizzazioni effettuate risiedono nel modo in cui il modulo legge i dati dalla RAM. Inizialmente veniva richiesto il dato e nel ciclo di clock successivo veniva raccolto. I dati, però, venivano raccolti non ancora inizializzati, probabilmente a causa di tempi di propagazione, e quindi era necessario un ciclo di clock di attesa. Si rendeva così necessaria la presenza di uno stato aggiuntivo in cui non si compiva alcun task. L'ottimizzazione sta quindi nel richiedere nei cicli di GET_C, SET_R, SET_C e CALC_GET già l'indirizzo successivo, eliminando così la necessità di avere uno stato di pura attesa e accorciando di molto il tempo complessivo di elaborazione. Il modulo inoltre supporta l'elaborazione di immagini anche più grandi di 128x128; più precisamente può equalizzare immagini fino a 255x255 senza subire alcuna modifica, a patto che ci sia spazio sufficiente sulla RAM.