Prova Finale

Progetto Di Reti Logiche

Anno Accademico 2020/2021

Prof. William Fornaciari

Leonardo Dezi - 10588151
Roberto Donato - 10576279

POLITECNICO
MILANO 1863

INDICE

1. INTRODUZIONE

1.1. Scopo del progetto

Il progetto prevede la realizzazione di un hardware in grado di equalizzare l'istogramma di un immagine in scala di grigi a 256 bit.

1.2. Interfaccia del componente

Il componente da descrivere possiede la seguente interfaccia:

```
entity project_reti_logiche is
port (
i_clk : in std_logic;
i_rst : in std_logic;
i_start : 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_vector (7 downto 0);
end project_reti_logiche;
```

In particolare:

- il nome del modulo deve essere project_reti_logiche
- i_clk è il segnale di CLOCK in ingresso generato dal TestBench;
- 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 per poter comunicare (sia in lettura che in scrittura);
- o_we è il segnale di WRITE ENABLE da dover mandare alla memoria (=1) per poter scriverci. Per leggere da memoria esso deve essere 0;
- o_data è il segnale (vettore) di uscita dal componente verso la memoria.

1.3. Algoritmo utilizzato

Il modulo implementato riceve i byte dell'immagine dalla memoria in cui è memorizzata leggendo i dati sequenzialmente riga per riga.
Il byte all'indirizzo 0 indica il numero di pixel per riga dell'immagine, mentre il byte all' indirizzo 1 indica il numero di pixel per colonna.
I valori degli n pixel dell'immagine sono scritti sequenzialmente dall' indirizzo "2" in poi, quindi fino all'indirizzo1+n, occupando un byte per ogni pixel.
La dimensione massima dell'immagine da equalizzare è di 128pixel X 128 pixel.
L' algoritmo di equalizzazione dell'immagine, ispirato al metodo di equalizzazione dell'istogramma ma fortemente semplificato, trasforma ogni pixel dell'immagine 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)

In questo algoritmo MAX PIXEL VALUE e MIN PIXEL VALUE, rappresentano il massimo e minimo valore dei pixel dell'immagine, CURRENT PIXEL VALUE è il valore del pixel da trasformare, e NEW PIXEL VALUE si riferisce al valore del nuovo pixel.
La nuova immagine equalizzata verrà scritta in memoria immediatamente dopo l'immagine originale, con indirizzo del primo byte pari a 2+ (# colonne * # righe)

1.4. Memoria

La memoria R.A.M. (Random Access Memory) prevede l'accesso diretto ad un qualsiasi indirizzo di memoria con lo stesso tempo di accesso. In questo progetto è stata utilizzata una memoria di tal tipo indirizzata a 2^{16} byte.

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;
entity rams_sp_wf is
port(
  clk  : in  std_logic;
  we   : in  std_logic;
  en   : in  std_logic;
  addr : in  std_logic_vector(15 downto 0);
  di   : in  std_logic_vector(7 downto 0);
  do   : out std_logic_vector(7 downto 0)
);
end rams_sp_wf;

architecture syn of rams_sp_wf is
type ram_type is array (65535 downto 0) of std_logic_vector(7 downto 0);
signal RAM : ram_type;
begin
  process(clk)
    begin
    if clk'event and clk = '1' then
      if en = '1' then
        if we = '1' then
          RAM(conv_integer(addr)) <= di;
          do                      <= di after 2 ns;
        else
          do <= RAM(conv_integer(addr)) after 2 ns;
        end if;
      end if;
    end if;
  end process;
end syn;
```

1.5. Esempio

Qui di seguito viene riportato un esempio di contenuto della memoria al termine di una elaborazione al termine di un'elaborazione di un'immagine di dimensione 4x3. I valori che qui sono rappresentati in decimale sono memorizzati in memoria con codifica binaria su 8 bit senza segno.

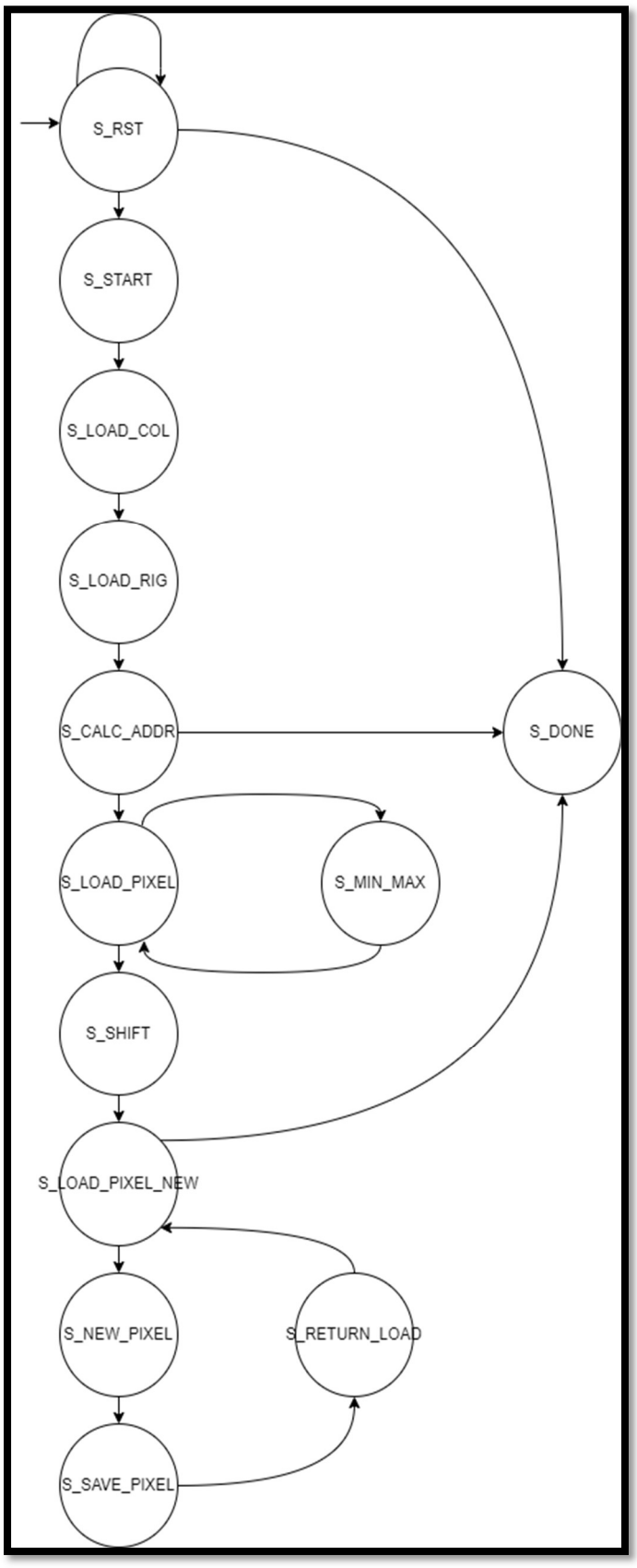
Esempio: (immagine 4 x 3 : *indirizzo* - *valore*) RANDOM

INDIRIZZO MEMORIA	VALORE	COMMENTO
0	4	\\ Byte più significativo numero colonne
1	3	\\ Byte meno significativo numero righe
2	76	\\ primo Byte immagine
3	131	
4	109	
5	89	
6	46	
7	121	
8	62	
9	59	
10	46	
11	77	
12	68	
13	94	\\ ultimo Byte immagine
14	120	\\ primo Byte immagine equalizzata (risultato)
15	255	
16	252	
17	172	
18	0	
19	255	
20	64	
21	52	
22	0	
23	124	
24	88	
25	192	

2.ARCHITETTURA

2.1. Macchina a stati

L'architettura progettata si comporta come una "Macchina Di Turing a singolo nastro" ovvero viene letta la memoria fino ad un certo punto prestabilito, che nel nostro caso è l'indirizzo di memoria dell'ultimo pixel da considerare, successivamente riparte dall'indirizzo di memoria in cui è contenuto il primo pixel, per rileggere l'input e scrivere in memoria a partire dal primo indirizzo libero successivo all'ultimo pixel da leggere, il nuovo pixel "equalizzato".



2.2. Descrizione stati

Vengono ora descritti i vari comportamenti degli stati utilizzati nel progetto

- ***S_RST***
  Il processo parte leggendo il segnale i_start, finché rimane a '0' resta in attesta in questo stato, quando il segnale passa a 1 inizia l'effettiva elaborazione.
- **S_START**
  Viene posto a '1' il segnale di lettura della memoria e viene inizializzato al primo indirizzo.
- **S_LOAD_COL e S_LOAD_RIG**
  Lo scopo di questi due stati è stabilire quanto è grande l'immagine considerata, con un massimo di 128 x 128 pixel, questi due valori sono memorizzati nei primi due indirizzi di memoria, che sono riservati esclusivamente a loro.
- **S_CALC_ADDR**
  Viene effettuata la moltiplicazione fra riga e colonna per calcolare l'offset della memoria, questo valore sommato all'indirizzo di base del primo pixel fornisce come risultato l'indirizzo dove verrà caricato il primo pixel elaborato(final_address), se però l'offset risulta essere '0' (considerato su 16 bit) l'immagine è considerata nulla e si va nello stato "S_DONE"; se invece l'offset è maggiore di '0' calcola il new_address che rappresenta l'indirizzo di memoria del primo pixel da caricare e si va nello state successivo.
- **S_LOAD_PIXEL**
  Viene caricato il pixel in memoria, viene aggiornato il current_address e se quest'ultimo è uguale a final_address si passa a S_SHIFT altrimenti si entra in un ciclo con lo stato S_MIN_MAX.
- **S_MIN_MAX**
  Ogni pixel caricato viene confrontato con il massimo e il minimo valore trovato fino a quel momento per trovare appunto il maggiore e il minore.
- **S_SHIFT**
  Viene calcolato il "DELTA" e il "FLOOR" per eseguire il calcolo dello "SHIFT".

- **S_LOAD_PIXEL_NEW**
  Riiniziando dal primo, vengono letti i pixel in memoria e come verificato in precedenza, quando il final_address è uguale al current_address si esce dal ciclo e si termina in S_DONE, altrimenti si passa allo state successivo.
- **S_NEW_PIXEL**
  Il pixel ricaricato viene trasformato attraverso lo shift_level e viene aggiornato.
- **S_SAVE_PIXEL**
  Viene salvato il pixel elaborato ed equalizzato in memoria, creando la nuova immagine.
- **S_RETURN_LOAD**
  Vengono resettati i valori di scrittura, è uno stato di passaggio che serve per ritornare in S_LOAD_PIXEL_NEW.
- **S_DONE**
  È lo stato finale del programma, da qui si ritorna ad S_RST

2.3. Scelte progettuali

È stato scelto di creare un progetto basato su un unico modulo con due process:

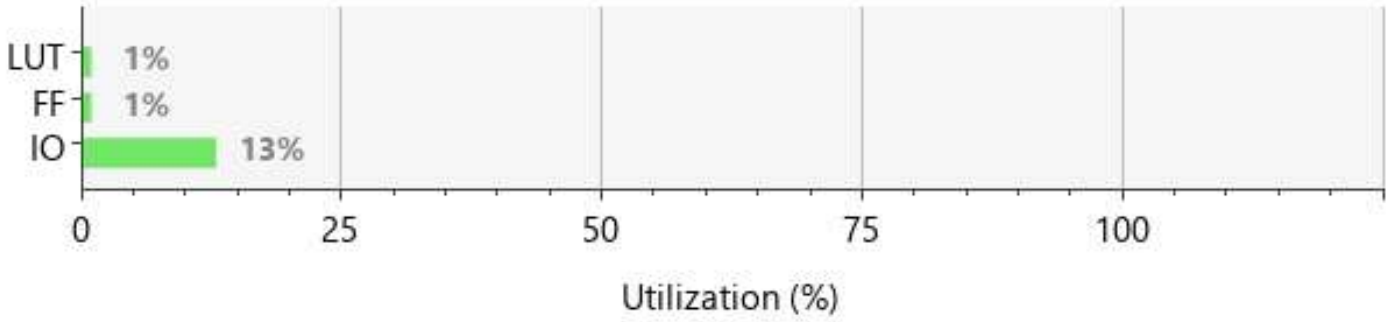
1. Il primo process si occupa esclusivamente del clock e del segnale di reset, ovvero di come viene comandata la macchina a stati sul fronte di salita del clock e della gestione del segnale di reset per far ripartire il programma

2. Il secondo process rappresenta la descrizione della FSM analizzando i segnali in ingresso e determinando in quale stato la macchina evolverà, questo rappresenta il lato combinatorio del programma.

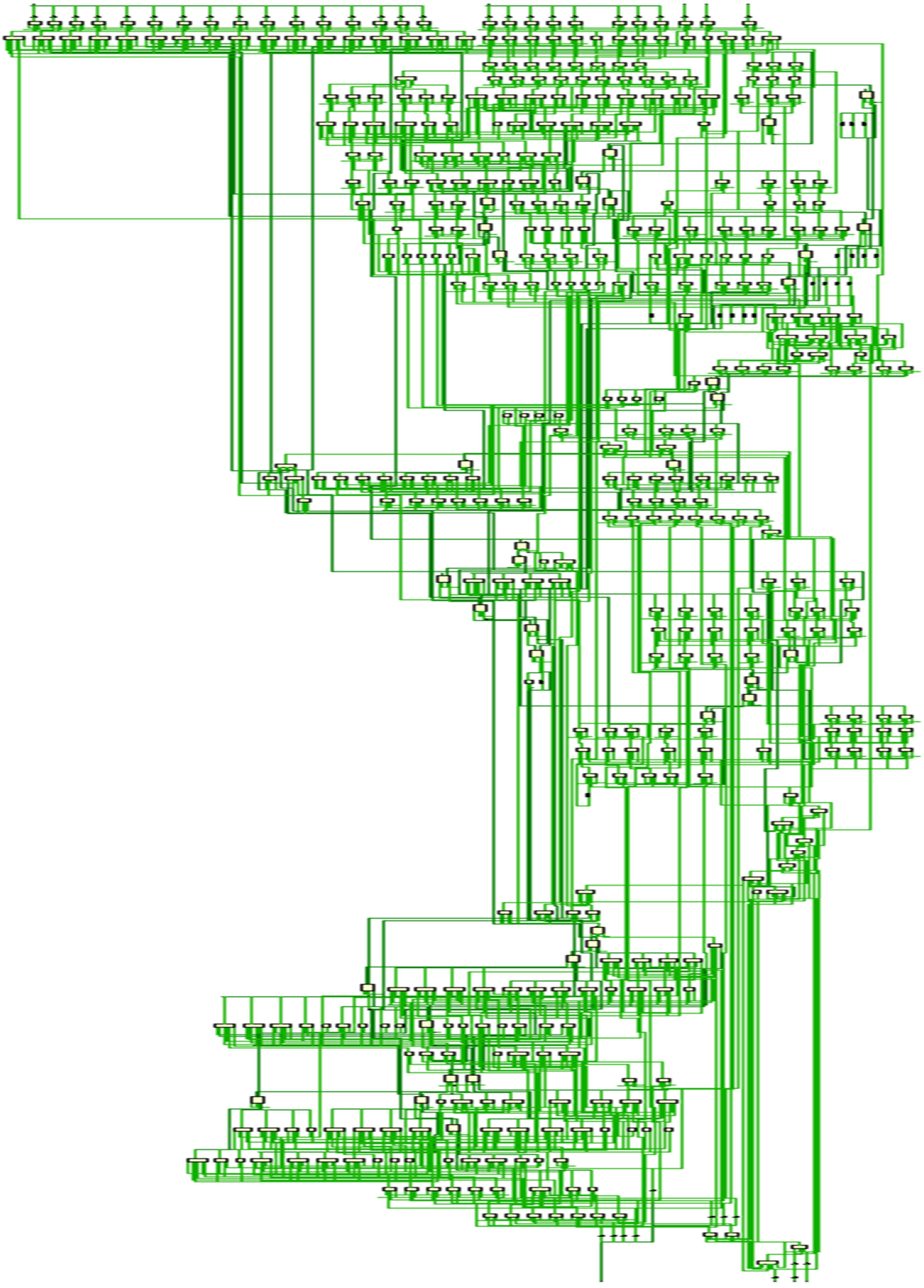
3. Risultati sperimentali

3.1. Report di sintesi

Resource	Utilization	Available	Utilization %
LUT	215	134600	0.16
FF	181	269200	0.07
IO	38	285	13.33



3.2. Schema



3.3. Simulazioni

Il programma è stato testato più volte passando con successo diversi test, tra cui il test di default fornito durante le lezioni.
Tra i vari test riportiamo ad esempio la descrizione dei più significativi:

- ✓ *Esecuzioni Multiple*

Questo test possiede 3 diverse memoria RAM, su ognuna di esse viene caricata un'immagine, la quale alla fine da come risultato l'immagine elaborata correttamente equalizzata, la peculiarità di tale test riede nell'utilizzo di molteplici memorie e quindi nel ricominciare ogni volta il processo dall'inizio.

- ✓ *Caso Limite 1: massima grandezza dell'immagine*

È stato verificato il corretto funzionamento del programma dando in input un'immagine grande al limite accettato, ovvero 128 pixel x 128 pixel, nonostante la grandezza molto elevata il test risulta passato in un tempo pari a:

Time (s): cpu = 00:00:16 ; elapsed = 00:02:37 . Memory (MB): peak = 1934.504

- ✓ *Caso Limite 2: immagine da "0 pixel"*

In questo test è stata data in input un'immagine grande '0' pixel, per verificare se il programma accettasse l'eccezione come previsto, passando la simulazione con successo.

4. Conclusioni

Il progetto è stato sviluppato rispettando le specifiche e coerentemente alle scelte progettuali che ci siamo imposti. Il componente è risultato funzionante su tutti i test effettuati sia in pre-sintesi che in post-sintesi. Per sintetizzarlo sono stati utilizzati: 215 LUT e 181 FF.
In conclusione, possiamo affermare che il componente progettato è in grado di equalizzare correttamente un'immagine di grandezza massima 128x128 pixel in scala di grigi portando a compimento il proprio compito anche con più immagini consecutivamente date in input.