

Progetto finale di Reti Logiche - Equalizzatore dell'istogramma di una immagine

Jasco Chen(10662235) Simone Coluccio(10633129)

Anno 2020/2021

Indice

1	Introduzione	2
1.1	Descrizione generale	2
1.2	Algoritmo	2
1.3	Interfaccia del componente	3
1.4	Dati e descrizione della Memoria	4
2	Design	5
2.1	Segnali implementati	5
2.1.1	Scelte progettuali	5
2.2	Stati della Macchina	7
3	Testing	9
3.1	Test Base	9
3.2	Test casi limite	10
3.2.1	Immagine dimensione 0xDC	11
3.2.2	Immagine dimensione 1x1	12
3.3	Test autogenerati da script python	12
4	Conclusione	13

Capitolo 1

Introduzione

1.1 Descrizione generale

Il progetto si basa sull'implementazione, in VHDL con FPGA target : xc7a200tfg484-1 e clock impostato a 100ns, di un metodo di equalizzazione dell'istogramma di una immagine. il metodo consiste nel ricalibrare il contrasto di una data immagine quando l'intervallo dei valori di intensità sono molto vicini effettuandone una distribuzione su tutto l'intervallo di intensità, al fine di incrementare il contrasto.

1.2 Algoritmo

l'algoritmo utilizzato è una versione semplificata della versione standard, è applicato su una scala di grigi a 256 livelli con intervallo tra [0 255], la trasformazione dei pixel avviene nel modo seguente:

```
DELTA VALUE = MAX PIXEL VALUE - MIN PIXEL VALUE  
SHIFT LEVEL = (8 - FLOOR(LOG2(DELTA VALUE +1)))  
TEMP PIXEL = (CURRENT - MIN PIXEL VALUE) « SHIFT LEVEL  
NEW PIXEL VALUE = MIN( 255 , TEMP PIXEL)
```

Dove **MAX PIXEL VALUE** e **MIN PIXEL VALUE**, sono il massimo e minimo valore dei pixel dell'immagine,
CURRENT PIXEL VALUE è il valore del pixel da trasformare, e
NEW PIXEL VALUE è il valore del nuovo pixel.

1.3 Interfaccia del componente

Il componente descritto ha 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:

- *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.

1.4 Dati e descrizione della Memoria

Il modulo implementato legge l'immagine da una memoria in cui è memorizzata, sequenzialmente e riga per riga, l'immagine da elaborare. Ogni byte corrisponde ad un pixel dell'immagine.

La dimensione della immagine è definita da **2 byte**, memorizzati a partire dall'**indirizzo 0**.

Il byte all'indirizzo 0 si riferisce alla dimensione di colonna; il byte nell'indirizzo 1 si riferisce alla dimensione di riga.

La **dimensione massima** dell'immagine è **128x128 pixel**.

L'immagine è memorizzata a partire dall'indirizzo 2 e in byte contigui. Quindi il byte all'indirizzo 2 è il primo pixel della prima riga dell'immagine.

I pixel dell'immagine, ciascuna di **8 bit**, sono memorizzati in memoria con indirizzamento al Byte partendo dalla **posizione 0**: il byte in posizione 0 si riferisce al numero di **colonne** (N-COL), il byte in posizione 1 si riferisce al numero di **righe** (N-RIG).

I pixel dell'immagine, ciascuno di un **8 bit**, sono memorizzati in memoria con indirizzamento al Byte partendo dalla **posizione 2**.

I pixel della immagine equalizzata, ciascuno di un **8 bit**, sono memorizzati in memoria con indirizzamento al Byte partendo dalla **posizione**: $2 + (N-COL * N-RIG) + 1$.

ESEMPIO di allocazione nella RAM di una immagine **2x2**:

Ind. Mem.	Valore
0	2
1	2
2	46
3	131
4	62
5	89
6	0
7	255
8	62
9	172

dove le **dimensioni** della immagine è definita negli indirizzi **0** e **1**, i **pixel** dell'immagine arrivati tramite in input dall'indirizzo **[2 5]** e i pixel equalizzati dall'indirizzo **[6 9]**

Capitolo 2

Design

2.1 Segnali implementati

Per il memorizzare i valori dei pixel abbiamo scelto di utilizzare registri **principali** e registri **ausiliari**, quest'ultimi forniscono una sorta di supporto per il corretto funzionamento della macchina a stati.

I **registri** utilizzati per il calcolo diretto dell'algoritmo hanno nomi analoghi alle variabili dell'algoritmo mentre quelle utilizzati in modo ausiliare hanno un indice numerico.

I segnali di **load** sono utilizzati per il caricamento di un valore nel registro con il nome simile a quello del segnale.

Segnali **max** e **min** sono segnali per notificare la macchina se il valore entrante è un massimo oppure un minimo assoluto.

Segnali **sel** insieme ai **mux** sono utilizzati per gestire gli input dei **multiplexer**.

Segnali **sum** e **sub** per gestire rispettivamente la somma e sottrazione.

2.1.1 Scelte progettuali

Per la moltiplicazione abbiamo scelto di procedere con un algoritmo, che dati due input **n** e **m** sommava **m**-volte **n** a se stesso.

I cicli li abbiamo gestiti tramite l'utilizzo di un **multiplexer** insieme ad un **sommatore** oppure **sottrattore** che decrementava oppure incrementava l'output del multiplexer e faceva un controllo:

E infine per il **FLOOR(LOG2)** dato che i risultato potevano variare solo in un intervallo finito e relativamente piccolo **[0 8]** abbiamo implementato dei controlli a soglia:

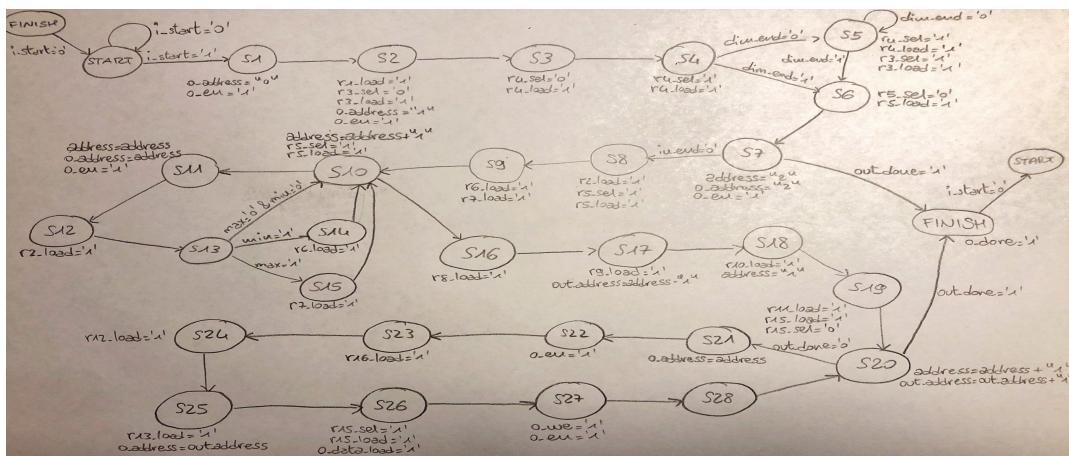
```
process(i_clk, i_rst)
begin
    if(i_rst = '1') then
        floor_log <= "00000000";
    elsif i_clk'event and i_clk = '1' then
        if(r10_load = '1') then
            if (delta_value1 <
                "0000000000000010") then
                floor_log <= "00000000";
            elsif(delta_value1 <
                "00000000000000100") then
                floor_log <= "00000001";
            elsif(delta_value1 <
                "0000000000000001000") then
                floor_log <= "00000010";
            elsif(delta_value1 <
                "00000000000000010000") then
                floor_log <= "00000011";
            elsif(delta_value1 <
                "000000000000100000") then
                floor_log <= "00000100";
            elsif(delta_value1 <
                "00000000001000000") then
                floor_log <= "00000101";
            elsif(delta_value1 <
                "00000000010000000") then
                floor_log <= "00000110";
            elsif(delta_value1 <
                "000000000100000000") then
                floor_log <= "00000111";
            elsif(delta_value1 =
                "0000000100000000") then
                floor_log <= "00001000";
            end if;
        end if;
    end if;
end process;
```

2.2 Stati della Macchina

Abbiamo progettato una macchina a stati di basso livello, spezzando ogni operazione in uno stato a parte per una maggiore chiarezza:

- **START**: stato iniziale dove attendiamo un segnale di `i_start = 1`;
- **S1**: leggiamo il valore all'indirizzo di memoria = 0;
- **S2**: carichiamo il primo valore letto nel registro `col`, poi carichiamo 0 nel registro `img_dim` e in fine leggiamo il valore all'indirizzo di memoria = 1;
- **S3**: carichiamo il valore letto nello stato precedente nel registro `row`;
- **S4**: cominciamo a decrementare `row`;
- **S5**: cominciamo il ciclo di decremento di `row` e incremento del suo `col` del suo stesso valore per fare la moltiplicazione;
- **S6**: il risultato della moltiplicazione viene salvato nel registro `o_r5`;
- **S7**: comincia il ciclo per la lettura di ogni singolo pixel dell'immagine;
- **S8**: si salva il valore appena letto in `o_r2` e decremento il contatore della dimensione dell'immagine
- **S9**: assegnamo il primo pixel ai registri `max` e `min`;
- **S10**: incrementiamo l'indirizzo per leggere il valore successivo;
- **S11**: assegniamo a `o_address` l'indirizzo incrementato e leggiamo il valore;
- **S12**: si salva il nuovo pixel in `o_r2`;
- **S13**: attendiamo il caricamento del pixel per una corretta valutazione del max e min;
- **S14**: stato in cui aggiorniamo `min` nel caso in cui non avessimo letto tutti i pixel si ritorna allo stato **S10**;
- **S15**: stato in cui aggiorniamo `max` nel caso in cui non avessimo letto tutti i pixel si ritorna allo stato **S10**;

- **S16:** calcoliamo il **delta_value**;
- **S17:** calcoliamo il **delta_value + 1** e si imposta l'indirizzo corretto di **scrittura**;
- **S18:** calcoliamo il **log(delta_value + 1)** e si imposta l'indirizzo corretto di **lettura**;
- **S19:** calcoliamo lo **shift_lvl** e carichiamo **o_r5** in **o_r15**;
- **S20:** cominciamo il ciclo di **scrittura e lettura**;
- **S21:** assegniamo l'indirizzo di **lettura** a **o_address**;
- **S22:** effettuiamo la richiesta di **lettura**;
- **S23:** salviamo il valore letto in **o_r16**;
- **S24:** calcoliamo il valore del pixel corrente meno il **min**;
- **S25:** shiftiamo il valore ottenuto e assegniamo a **o_address** l'indirizzo di **scrittura**;
- **S26:** incrementiamo il contatore del ciclo e carichiamo il valore ottenuto dall'algoritmo in **o_data** per la **scrittura**;
- **S27:** effettuiamo la scrittura in memoria;
- **S28:** attendiamo l'incremento del contatore per una corretta valutazione del segnale di fine **scrittura**;
- **FINISH:** settiamo **o_done = 1** e ritorniamo allo stato **START**.



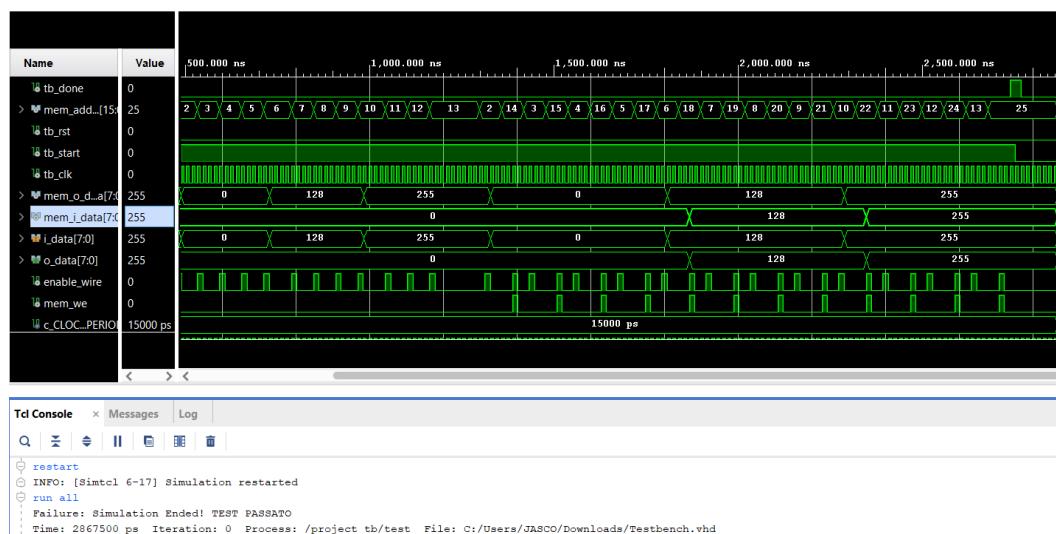
Capitolo 3

Testing

3.1 Test Base

Abbiamo usato i test base forniti dalla consegna come primo step per verificare il corretto funzionamento del progetto.

Sotto è riportato il risultato del **Behavioural** con input presi di una immagine 4x3:



valori utilizzati:

Ind. Mem.	Valore
0	4
1	3
2	0
3	0
4	0
5	0
6	128
7	128
8	128
9	128
10	255
11	255
12	255
13	255
14	0
15	0
16	0
17	0
18	128
19	128
20	128
21	128
22	255
23	255
24	255
25	255

3.2 Test casi limite

Abbiamo identificato due principali casi limite:

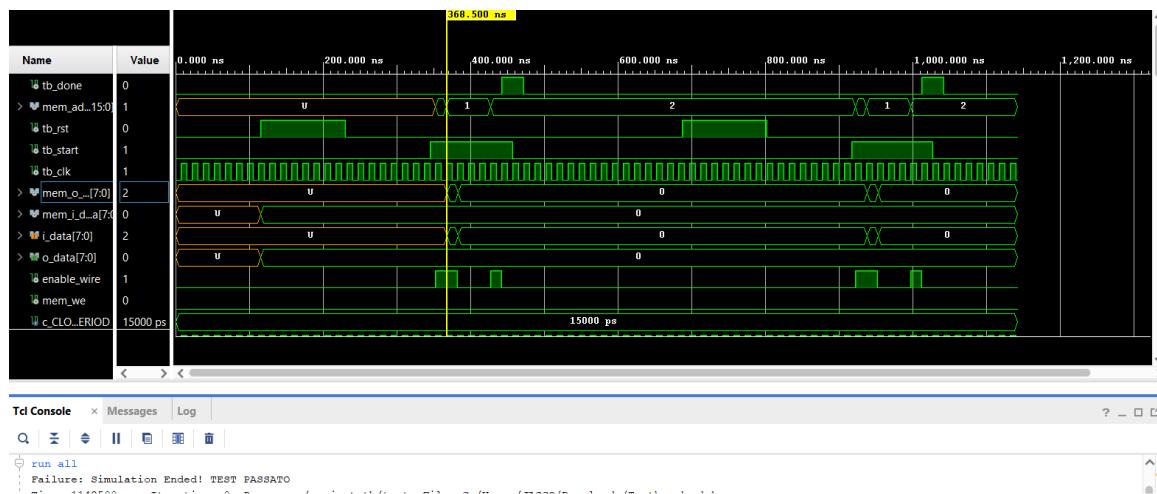
1. caso immagine con **dimensione 0**, ovvero se uno dei valori presenti nei primi due indirizzi di memoria, che corrispondono alla dimensione, fosse 0;

- caso immagine con **dimensione 1**, ovvero i valori presenti nei primi due indirizzi di memoria, che corrispondono alla sua dimensione, fossero entrambi 1.

3.2.1 Immagine dimensione 0xDC

In questo caso, dato che nel nostro **design** della macchina a stati abbiamo gestito tale caso nello stato **S7** dove il risultato della moltiplicazione risulta **0** e porta la macchina direttamente allo stato **FINISH** con **o_we** sempre basso, il test risulta corretto.

Sotto è riportato il risultato del **Behavioural** con input presi di una immagine **2x0**:

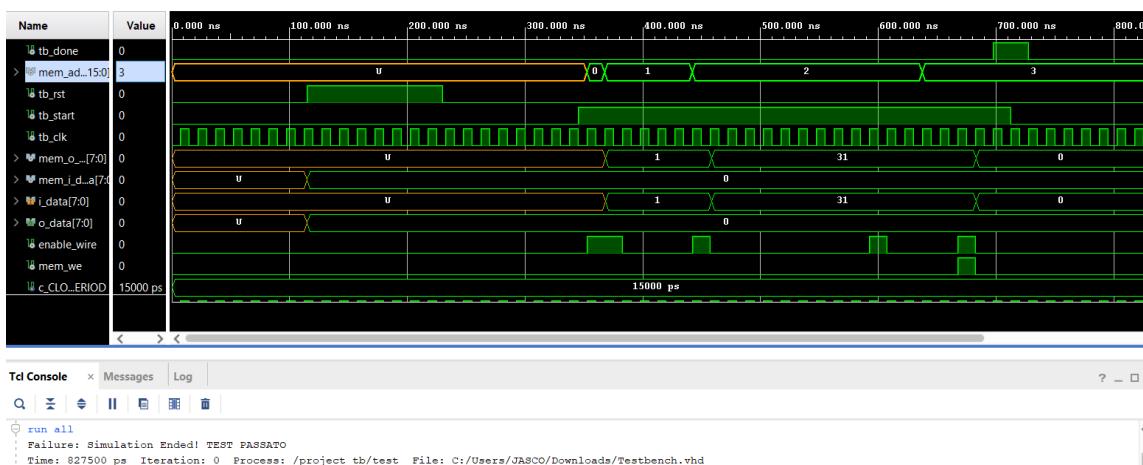


valori utilizzati:

Ind. Mem.	Valore
0	2
1	0

3.2.2 Immagine dimensione 1x1

Trattato come caso limite ma non tanto da dover scalare il design, è dunque possibile processarlo come una qualsiasi immagine di dimensioni maggiori di 0. Sotto è riportato il risultato del **Behavioural** con input presi di una immagine **1x1**:



valori utilizzati:

Ind. Mem.	Valore
0	1
1	1
2	31
3	0

3.3 Test autogenerati da script python

Abbiamo in fine usuffruito di un generatore di casi di test(git repository) a singole e multiple immagini con esisito positivo.

Capitolo 4

Conclusione

Il componente non funziona in **post sintesi** dato che ci siamo accorti della presenza di **latch** quando assegniamo un valore direttamente negli stati : durante la descrizione del **current state** ai registri che manipolano gli indirizzi (3 **latch** ognuno di 16: uno per il registro **address**, uno per **out_address** e uno per **o_address**).

Il problema è risolvibile tramite l'assegnamento dei valori tramite **segnali** che verranno gestiti nel **datapath**(tramite i **process**) evitando l'assegnamento diretto nella descrizione del **current state**.

Abbiamo implementato una bozza della soluzione sopracitata ed effettivamente il numero di **latch** scompare e il componente si sintetizza correttamente. Tuttavia tale implementazione fa variare il risultato dell'algoritmo e lo porta a dei valori non corretti, abbiamo dunque capito che fosse necessario una riprogettazione abbastanza sostanziosa del componente. Per problemi di tempistica dunque abbiamo optato di lasciare il componente funzionante solo in **pre sintesi** e riprogettarlo in futuro.