# Progetto finale di Reti Logiche

## Anno Accademico 2020/2021

Matteo Crugnola - 10608406
Andrea Carotti - 10628385

## Indice

# 1 Introduzione

## 1.1 Scopo del progetto (da specifica)

Lo scopo del progetto consiste nell'implementare una versione semplificata del metodo di equalizzazione dell'istogramma di un'immagine. L'algoritmo di equalizzazione è fatto per essere applicato solo ad immagini in scala di grigi a 256 livelli.
Il metodo è pensato per ricalibrare il contrasto di un'immagine quando l'intervallo dei valori di intensità sono molto vicini effettuandone una distribuzione su tutto l'intervallo di intensità, per incrementare il contrasto.

## 1.2 Specifiche generali (da specifica)

L'algoritmo da implementare è il 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)
```

Il modulo da implementare dovrà leggere l'immagine da una memoria, sequenzialmente e riga per riga.
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.
L'immagine equalizzata deve essere scritta in memoria immediatamente dopo l'immagine originale.

## 1.3 Interfaccia del componente (da specifica)

```
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.

# 2 Architettura

Il modulo partirà nell'elaborazione quando un segnale "start" in ingresso viene portato a "1".
Il segnale di "start" rimane alto finchè il segnale di "done" non viene portato a "1".
Al termine della computazione, scritto il risultato in memoria, il modulo porta a "1" il segnale "done" che notifica la fine dell'elaborazione.
Il segnale "done" deve rimanere alto finchè il segnale "start" non è riportato a "0".
Se viene portato a "1" il segnale "start", il modulo riparte con la fase di codifica.

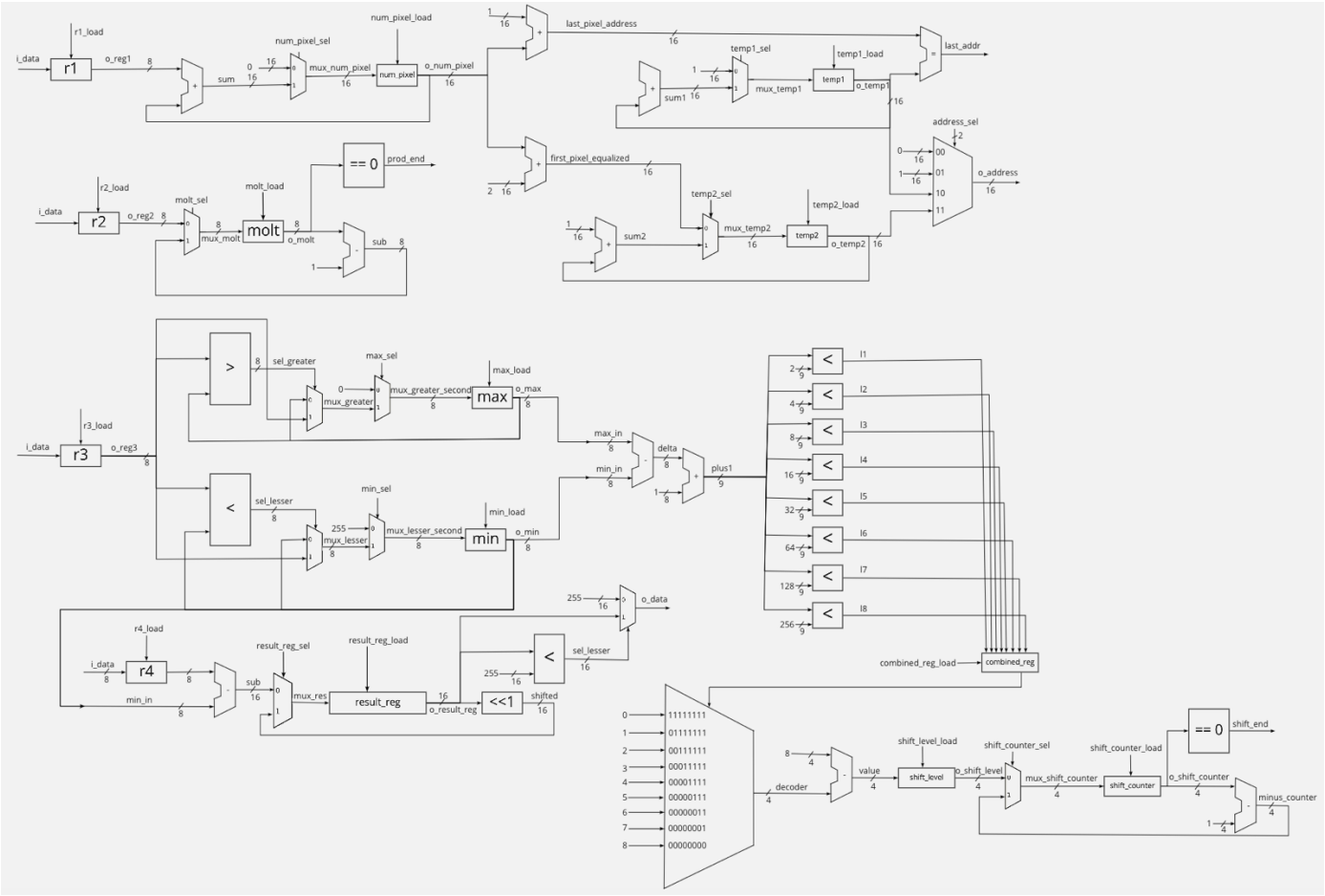
## 2.1 Scelte progettuali

Abbiamo suddiviso il datapath in sottocomponenti per semplicità di scrittura e realizzazione.

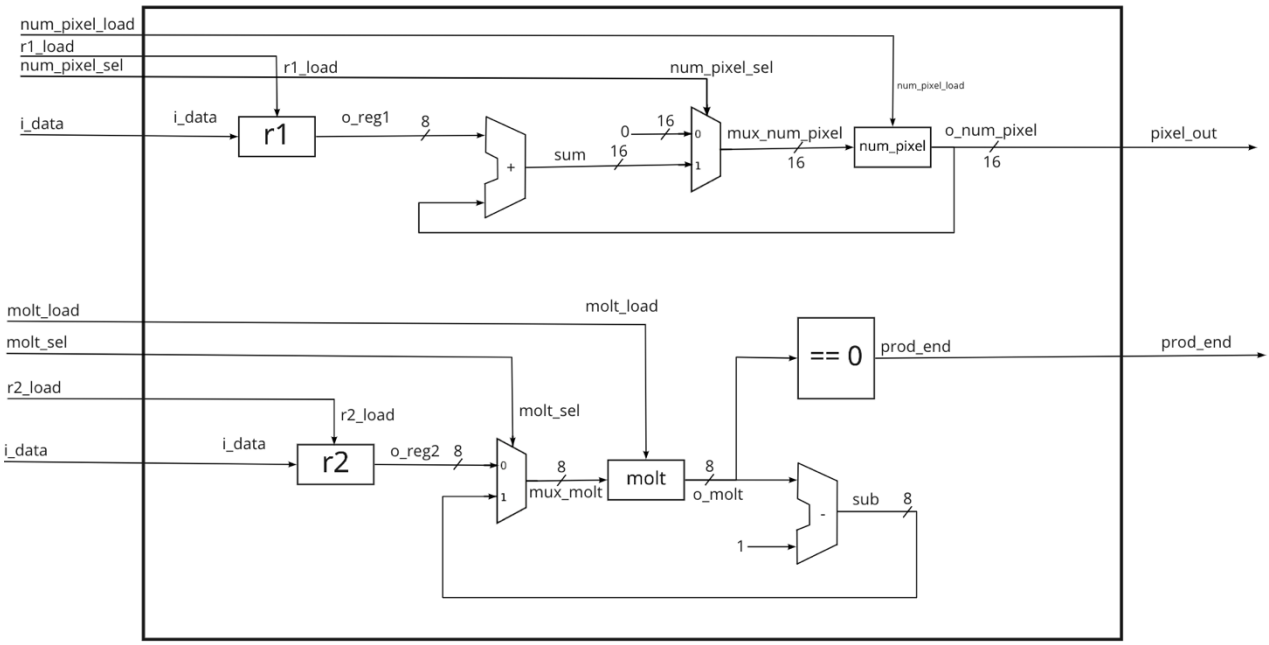
Componenti:

- calcolatore del numero di pixel dell'immagine
- calcolatore degli indirizzi di lettura e scrittura
- calcolatore di massimo e minimo
- calcolatore dello shift level
- calcolatore del risultato da scrivere in memoria

## 2.2 Datapath generale



## 2.2.1 Calcolatore del numero di pixel dell'immagine




**Parte alta – sommatore del numero di righe**
Questo componente serve per calcolare il prodotto dei primi due numeri forniti in memoria N x M. Opera in modo analogo all'elevatore a potenza, sommando M volte il numero N a sé stesso, insieme al calcolatore della terminazione del prodotto che serve per stabilire quando terminare l'operazione di somma del numero con sé stesso.

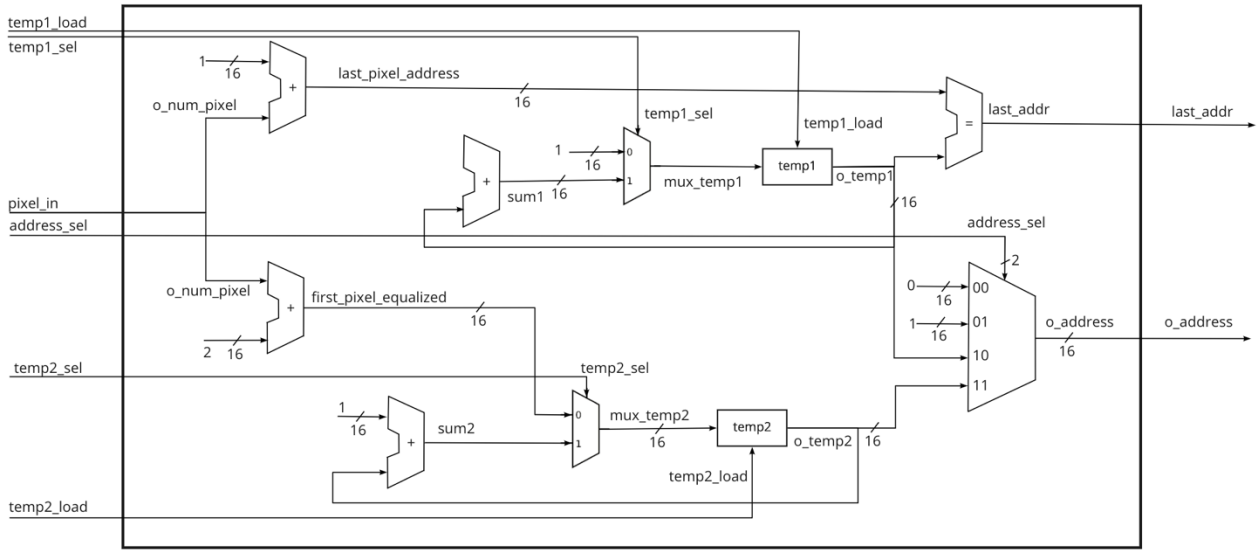
**Parte bassa – calcolatore terminazione prodotto**

Questo componente serve per far funzionare correttamente il calcolatore del numero di pixel dell'immagine, all'interno di i_data è contenuto il valore del numero di colonne, che serve per stabilire quante volte andiamo a sommare con sé stesso il numero di righe per il calcolo del numero totale di pixel.

Ad ogni ciclo viene decrementato di un'unità il valore contenuto in molt, fino a che non si raggiunge il valore 0, che stabilisce che dobbiamo smettere di sommare (nel componente calcolatore del numero di pixel) con sé stesso il numero di colonne.

```
entity pixel_calculator is
    port (
        i_clk : in std_logic;
        i_rst : in std_logic;
        i_data : in std_logic_vector(7 downto 0);
        r1_load : in std_logic;
        r2_load : in std_logic;
        num_pixel_load : in std_logic;
        num_pixel_sel : in std_logic;
        molt_load : in std_logic;
        molt_sel : in std_logic;
        prod_end : out std_logic;
        pixel_out : out std_logic_vector(15 downto 0)
    );
end pixel_calculator;
```

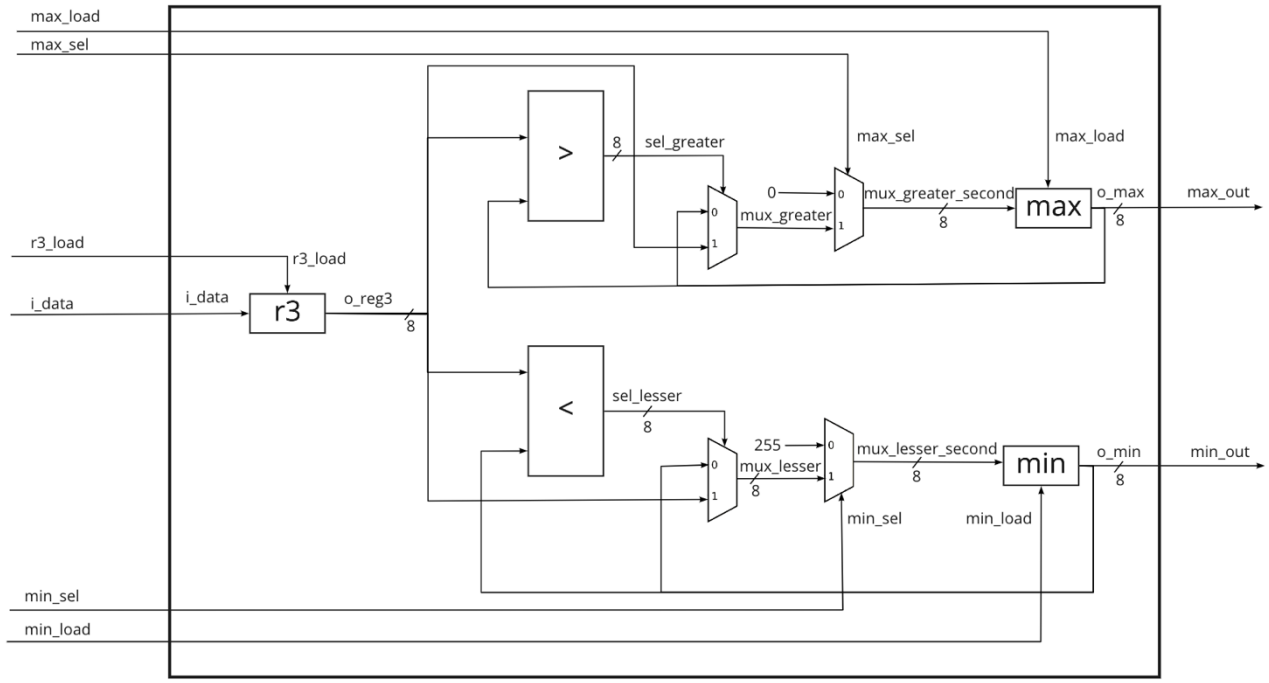
## 2.2.2 Calcolatore degli indirizzi di lettura e scrittura




Questo componente serve per decidere quale indirizzo di memoria leggere o scrivere, e per stabilire se siamo arrivati all'ultimo pixel dell'immagine.
Inizialmente leggiamo tutti i pixel dell'immagine per trovare il massimo e il minimo.
Successivamente al calcolo dello shift level ci continuiamo a spostare tra indirizzo di lettura pixel (temp1) e scrittura (temp2) per scrivere in memoria il valore del pixel modificato.

```
entity address_calculator is
    port (
        i_clk : in std_logic;
        i_rst : in std_logic;
        pixel_in : in std_logic_vector(15 downto 0);
        temp1_sel : in std_logic;
        temp1_load : in std_logic;
        temp2_sel : in std_logic;
        temp2_load : in std_logic;
        address_sel : in std_logic_vector(1 downto 0);
        o_address : out std_logic_vector(15 downto 0);
        last_addr : out std_logic
    );
end address_calculator;
```

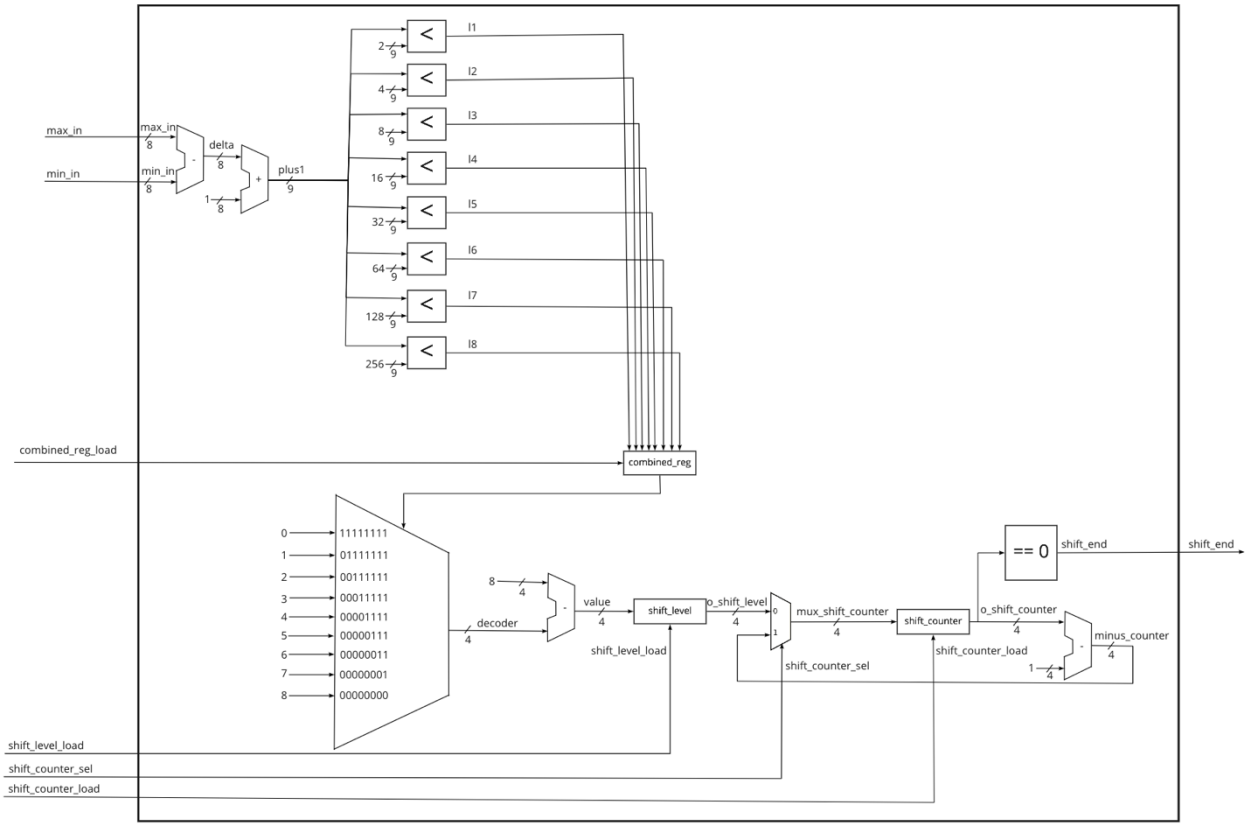
## 2.2.3 Calcolatore massimo e minimo




Questo componente serve per calcolare i valori del massimo e del minimo pixel in memoria, fornito un intervallo di valori compresi tra 0 e 255 l'algoritmo dovrà verificare uno per uno i valori incontrati in memoria, e se il valore incontrato è maggiore del maggiore elemento precedente viene salvato nel registro altrimenti continua a leggere, in modo analogo esegue per trovare il minimo.

```
entity maxmin_calculator is
    port (
        i_clk : in std_logic;
        i_rst : in std_logic;
        i_data : in std_logic_vector(7 downto 0);
        r3_load : in std_logic;
        max_sel : in std_logic;
        min_sel : in std_logic;
        max_load : in std_logic;
        min_load : in std_logic;
        max_out : out std_logic_vector(7 downto 0);
        min_out : out std_logic_vector(7 downto 0)
    );
end maxmin_calculator;
```

## 2.2.4 Calcolatore dello shift level

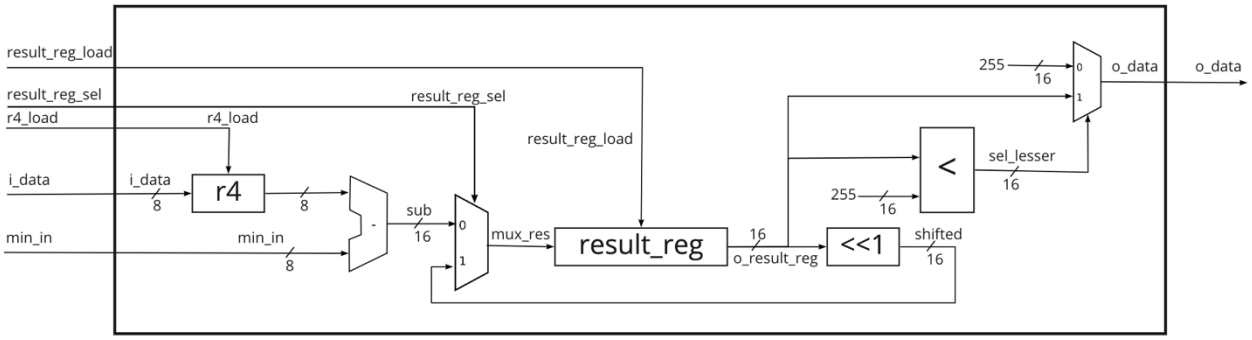



La prima parte di questo componente calcola lo shift level una sola volta dopo che i pixel di massimo e minimo sono stati calcolati.

Successivamente utilizziamo il shift counter per eseguire più volte (esattamente tante quante il valore di shift level) lo shift di 1 bit del valore dei pixel dell'immagine prima scrivere il valore finale in memoria.
Questo procedimento viene ripetuto per ogni singolo pixel dell'immagine.

```
entity shift_level_calculator is
    port (
        i_clk : in std_logic;
        i_rst : in std_logic;
        max_in : in std_logic_vector (7 downto 0);
        min_in : in std_logic_vector (7 downto 0);
        combined_reg_load : in std_logic;
        shift_level_load : in std_logic;
        shift_counter_sel : in std_logic;
        shift_counter_load : in std_logic;
        shift_end : out std_logic
    );
end shift_level_calculator;
```

## 2.2.5 Calcolatore del risultato da scrivere in memoria




Questo componente esegue, a partire dalla differenza tra il pixel da equalizzare e il pixel minimo, lo shift di 1 bit più volte.

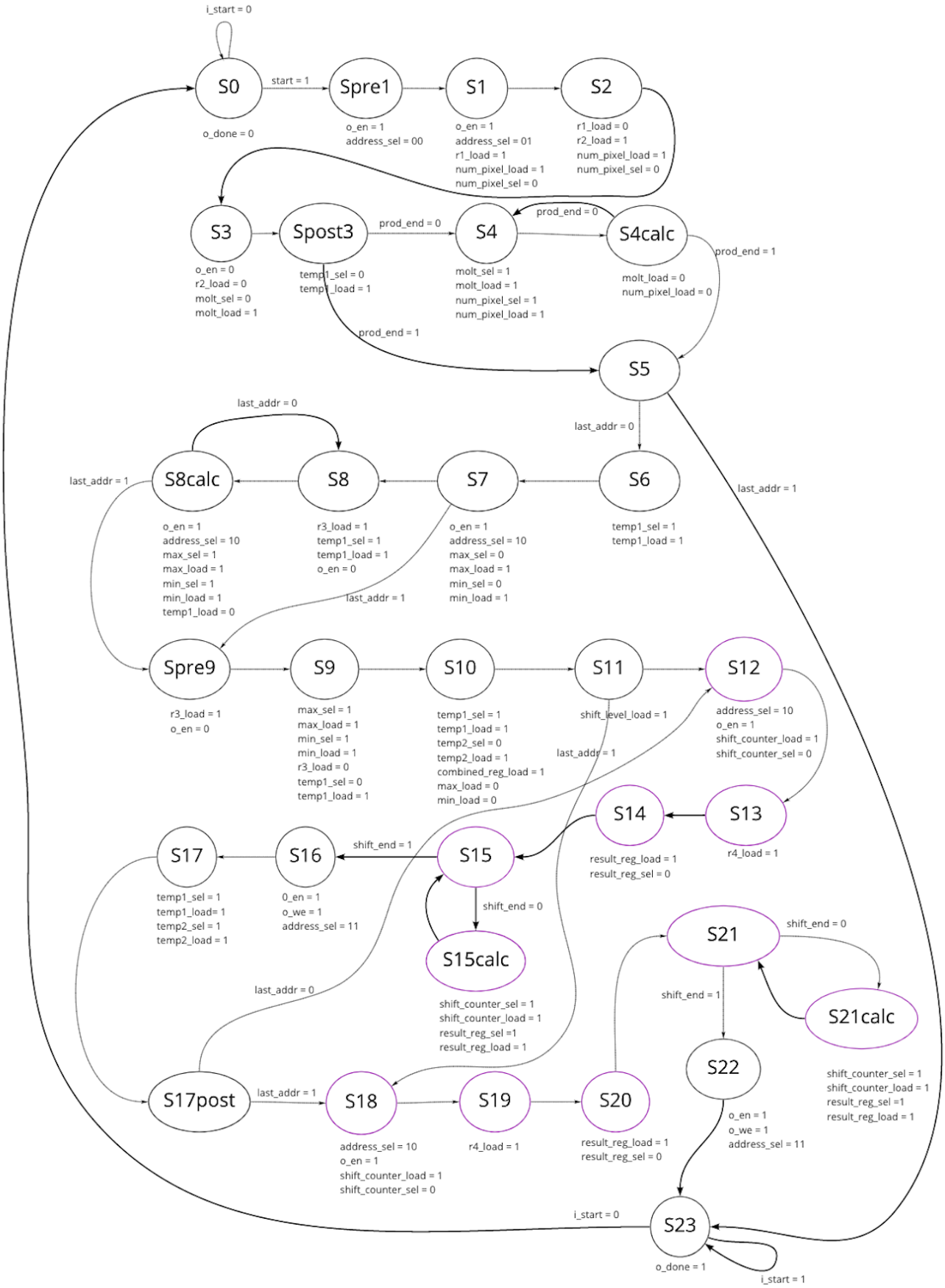
Il numero di volte è controllato dal componente dello shift level.

Infine, scrive in memoria il numero minimo tra 255 e il valore che è stato shiftato.

```
entity pixel_writer is
    port (
        i_clk : in std_logic;
        i_rst : in std_logic;
        i_data : in std_logic_vector(7 downto 0);
        min_in : in std_logic_vector(7 downto 0);
        r4_load : in std_logic;
        result_reg_sel : in std_logic;
        result_reg_load : in std_logic;

        o_data : out std_logic_vector(7 downto 0)
    );
end pixel_writer;
```

## 2.2 Finite State Machine



Gli stati della FSM sono così organizzati:

- da Spre1 a S5 viene calcolato il numero di pixel dell'immagine
- da S6 a S9 vengono calcolati il massimo e il minimo pixel dell'immagine
- da S10 a S11 viene calcolato lo shift level
- da S12 a S22 vengono letti, computati e riscritti in memoria i valori dei pixel equalizzati
- S23 è lo stato finale in cui rimaniamo finché i_start rimane a 1

Abbiamo aggiunto un salto da S5 a S23 nel caso limite in cui l'immagine da equalizzare sia di 0 pixel.

Ci sono inoltre due salti per gestire il caso di immagine con 1 solo pixel:

- Da S7 a Spre9
- Da S11 a S18

# 3 Sintesi

## 3.1 Report utilization

Dal Report Utilization notiamo che il progetto utilizza 141 FF e 190 LUT senza inferire dei latch.

```
1. Slice Logic
--------------

+-------------------------+------+-------+-----------+-------+
|        Site Type        | Used | Fixed | Available | Util% |
+-------------------------+------+-------+-----------+-------+
| Slice LUTs*             |  190 |     0 |    134600 |  0.14 |
|   LUT as Logic          |  190 |     0 |    134600 |  0.14 |
|   LUT as Memory         |    0 |     0 |     46200 |  0.00 |
| Slice Registers         |  141 |     0 |    269200 |  0.05 |
|   Register as Flip Flop |  141 |     0 |    269200 |  0.05 |
|   Register as Latch     |    0 |     0 |    269200 |  0.00 |
| F7 Muxes                |    0 |     0 |     67300 |  0.00 |
| F8 Muxes                |    0 |     0 |     33650 |  0.00 |
+-------------------------+------+-------+-----------+-------+
```

## 3.2 Report timing

Analizzando il timing report, si può vedere quanto del periodo di clock è effettivamente utile per svolgere il lavoro. Si è ottenuto con il periodo di clock del testbench di 100ns uno Slack pari a 94,398ns.

```
Timing Report

Slack (MET) :             94.398ns  (required time - arrival time)
  Source:                 PIXEL_CAL/o_num_pixel_reg[4]/C
                            (rising edge-triggered cell FDCE clocked by clock  {rise@0.000ns fall@5.000ns period=100.000ns})
  Destination:            FSM_sequential_cur_state_reg[1]/D
                            (rising edge-triggered cell FDCE clocked by clock  {rise@0.000ns fall@5.000ns period=100.000ns})
  Path Group:             clock
  Path Type:              Setup (Max at Slow Process Corner)
  Requirement:            100.000ns  (clock rise@100.000ns - clock rise@0.000ns)
  Data Path Delay:        5.451ns  (logic 2.742ns (50.303%)  route 2.709ns (49.697%))
  Logic Levels:           7  (CARRY4=4 LUT5=1 LUT6=2)
  Clock Path Skew:        -0.145ns (DCD - SCD + CPR)
    Destination Clock Delay (DCD):    2.100ns = ( 102.100 - 100.000 )
    Source Clock Delay      (SCD):    2.424ns
    Clock Pessimism Removal (CPR):    0.178ns
  Clock Uncertainty:      0.035ns  ((TSJ^2 + TIJ^2)^1/2 + DJ) / 2 + PE
    Total System Jitter     (TSJ):    0.071ns
    Total Input Jitter      (TIJ):    0.000ns
    Discrete Jitter          (DJ):    0.000ns
    Phase Error              (PE):    0.000ns
```

# 4 Simulazioni

Per poter testare il componente, dopo aver verificato il funzionamento generale, abbiamo utilizzato oltre al test bench fornito, anche gli esempi forniti da specifica: tutte immagini da 4x3 pixel.
Abbiamo verificando anche i casi limite(con immagini da 0x4 o da 4x0 pixel e immagini da 1x1).

# 5 Conclusione

La realizzazione finale del progetto soddisfa le specifiche richieste ed impiega:
- Data path delay pari a 5,451ns
- Utilizzo di 190 LUT
- Utilizzo di 141 FF

Dopo aver realizzato il datapath e la macchina a stati, la scrittura del codice ha impiegato poco tempo, dopo dei primi problemi dovuti alla memoria, che inizialmente pensavamo asincrona (quindi durante i cicli di lettura il dato viene fornito nel ciclo di clock successivo a quando viene fornito l'indirizzo), abbiamo introdotto degli stati intermedi risolvendo così i problemi incontrati.
Non abbiamo inserito immagini dei test perché le scritture in memoria dei valori dei singoli pixel si trovano a molti cicli di clock di distanza tra loro e sarebbe risultato insignificante vedere un singola foto del test.