

01/04/2021

# Prova Finale di Reti Logiche

Eleonora Giudici

Luca Gerin



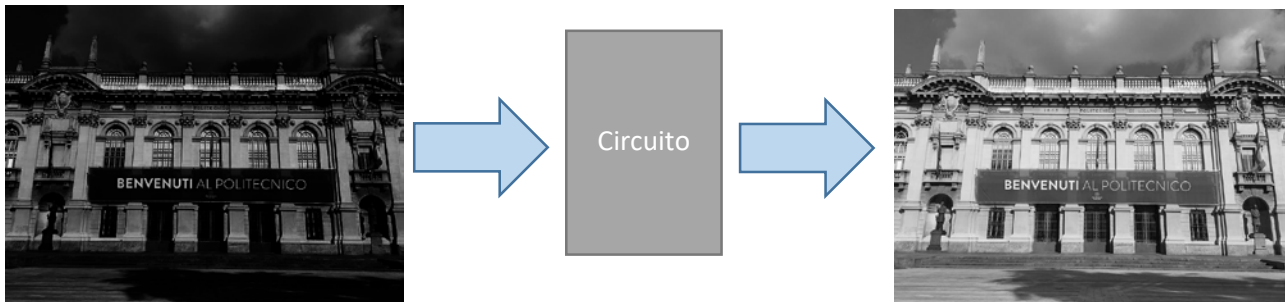
Politecnico di Milano

# INDICE

1. Introduzione	3
2. Architettura	5
➤ Descrizione della macchina a stati finiti	5
➤ Descrizione dei segnali utilizzati	8
3. Risultati sperimentali	9
4. Simulazioni	10
5. Conclusioni	13
➤ Ottimizzazioni possibili	13

## Introduzione

Si vuole realizzare un circuito che implementi la funzione di equalizzazione dell'istogramma di un'immagine, cioè che si occupi di modificare il valore di ogni suo pixel ridistribuendolo lungo la gamma totale, in modo da produrre un'immagine che abbia contrasto migliorato.



L'immagine è letta dal circuito da una memoria sequenziale dove è salvata, elaborata e poi riscritta sulla stessa. L'algoritmo fornito e utilizzato per l'elaborazione del singolo pixel è il seguente, riportato in pseudocodice:

```
DeltaValue = MaxPixelValue - MinPixelValue  
ShiftLevel = ( 8 - floor(log2 ( DeltaValue + 1 )) )  
TempPixel = ( CurrentPixelValue - MinPixelValue ) << ShiftLevel  
NewPixelValue = min ( 255, TempPixel )
```

Il significato delle variabili è intuitivo: *MaxPixelValue* e *MinPixelValue* rappresentano rispettivamente il valore numerico massimo e minimo dei pixel che si trovano nell'immagine prima dell'elaborazione, mentre *CurrentPixelValue* e *NewPixelValue* rappresentano rispettivamente il valore che lo stesso pixel dell'immagine assume prima e dopo l'elaborazione. Le altre variabili sono utili allo svolgimento dell'algoritmo e assumono significato solo al suo interno.

L'immagine da elaborare è in scala di grigi a 256 livelli e ogni pixel è salvato in un byte di memoria.

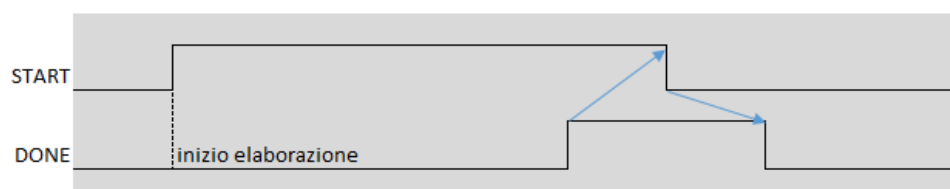
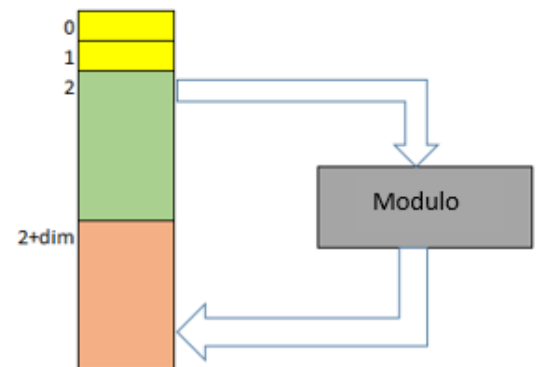
Nei primi due byte della memoria si trovano le dimensioni dell'immagine: nella prima cella il numero di righe e nella seconda cella il numero di colonne, entrambi valori compresi tra 0 e 128. Il bit all'indirizzo seguente, l'indirizzo 2, contiene il primo pixel dell'immagine, seguito dagli altri. L'immagine elaborata è riscritta sulla memoria a partire dal primo indirizzo esattamente dopo la versione originale.

Il modulo è in grado di codificare più immagini consecutivamente.

La macchina ha il seguente protocollo di re-start:

- Quando un segnale START in ingresso viene portato a 1, il modulo progettato inizia l'elaborazione (il segnale START rimane alto)
- Quando la macchina ha concluso il processo, il modulo porta un segnale DONE a 1, in modo da segnalare la fine dell'elaborazione
- Il modulo mantiene alto il segnale DONE fino a quando il segnale START commuta a 0, a questo punto riporta DONE a 0 e può ricevere un altro segnale di START per ricominciare l'elaborazione.

Si assume che precedentemente alla prima codifica venga sempre dato un segnale di RESET al modulo.



La memoria è una RAM che opera in modalità *write first* ed è situata all'interno dei test bench, se ne conoscono interfaccia e architettura:

```
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;
```

L'interfaccia del componente progettato è la seguente:

```
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;
```

Si riporta un breve esempio del contenuto della memoria dopo l'elaborazione di un'immagine dalle dimensioni ridotte:

0	2
1	2
2	68
	131
	76
	46
2+dim	88
	255
	94
	0

## Architettura

L'architettura implementata è composta da un unico modulo, il quale viene connesso tramite opportuni segnali alla memoria RAM fornita dal Test Bench.

Tale modulo permette di realizzare una macchina a stati finiti e il suo funzionamento si basa sulla presenza di due processi sincronizzati:

- **state\_update** è il processo che si occupa di gestire la condizione di avvio, in cui il Test Bench genera un segnale di reset, e successivamente ha il compito di aggiornare i segnali utili alla macchina con il loro nuovo valore ogni ciclo di clock. Tra i segnali aggiornati c'è anche quello che contiene il valore dello stato presente. Questo processo è sincronizzato sul segnale di clock e di reset in modo che, quando il circuito riceve un reset, tutti i segnali vengano inizializzati (si forza inoltre il ritorno nello stato iniziale) e, sul fronte di salita del clock, vengano aggiornati. Il processo **main** utilizzerà i segnali aggiornati e, a seconda dello stato presente, deciderà se e come questi dovranno essere successivamente modificati.

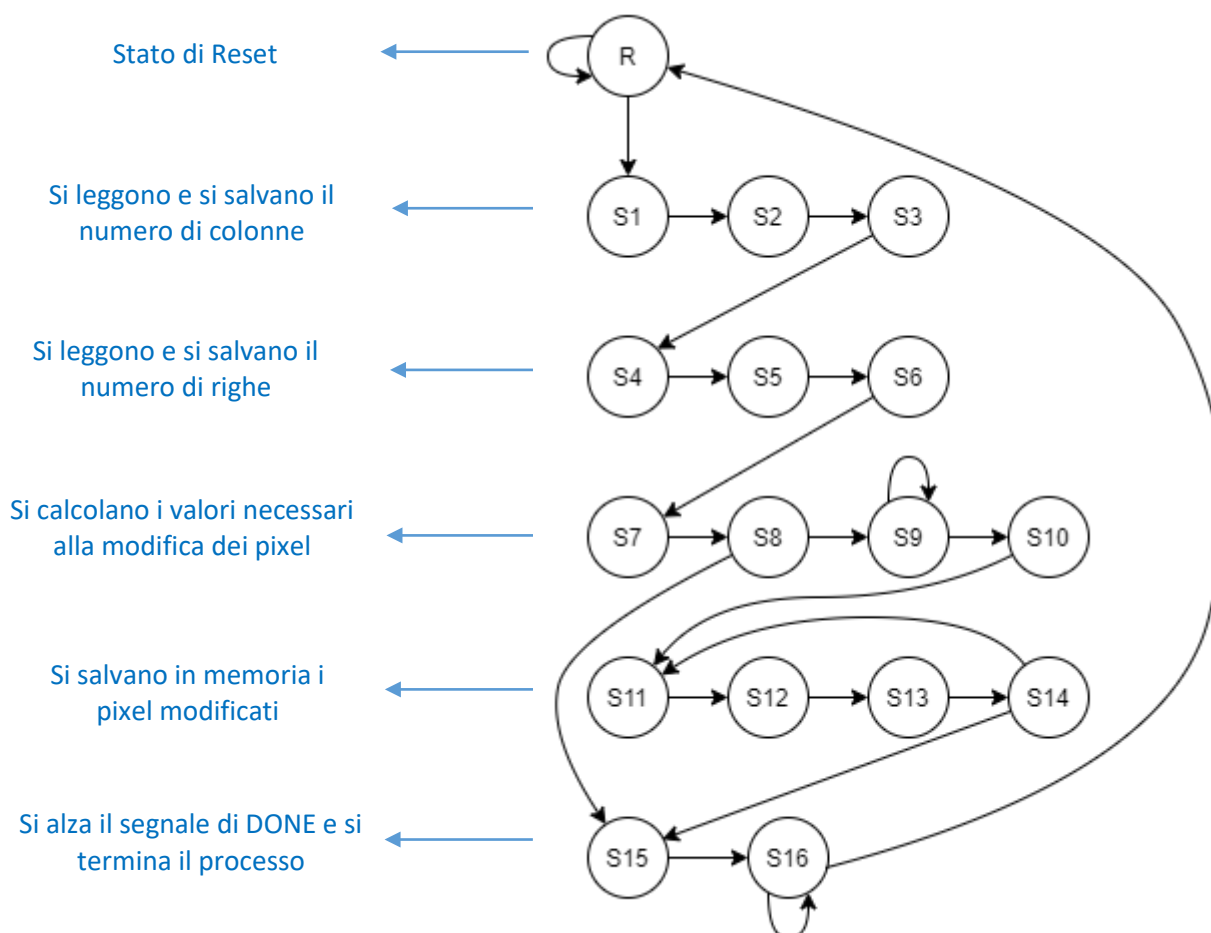
```
-- process to handle the reset and the update of the signals throughout clock cycles
state_update: process (i_clk,i_rst)
```

- **main** è il processo che realizza la funzionalità principale della macchina, definendo il comportamento degli stati della macchina a stati finiti, modificando i valori di opportuni segnali e variabili secondo necessità, col fine di effettuare calcoli e di leggere o scrivere in memoria. Il processo è attivato dal segnale di clock e dal cambiamento del segnale *current\_state*, in modo che un cambiamento dello stato lo risvegli, ma anche un ciclo sullo stesso stato lo porti a essere eseguito di nuovo al successivo ciclo di clock.

```
-- main process to implement the FSM with main function
main: process(i_clk,current_state)
```

### DESCRIZIONE DELLA MACCHINA A STATI FINITI

Si riporta di seguito il diagramma degli stati con successiva descrizione di ognuno di essi.



## **RESET (R)**

Inizializza tutti i segnali di output del componente e tiene la macchina in attesa del segnale di *start*. Alla commutazione di *start* a 1 avviene un passaggio allo stato seguente, se invece essa non è riscontrata avviene un ritorno allo stato di RESET per ripetere nuovamente il controllo al ciclo di clock successivo.

## **AKS\_COLUMNS (S1)**

Si occupa di inizializzare i segnali necessari per la lettura della prima cella di memoria, la quale contiene il numero di colonne occupate dai pixel dell'immagine. Si passa poi allo stato successivo WAIT\_COLUMNS.

## **WAIT\_COLUMNS (S2)**

Al fine di assicurarsi che non ci siano errori nell'output della memoria, legati alla sincronizzazione o a eventuali ritardi, in questo stato si riconfermano i segnali che erano stati precedentemente impostati per la lettura, per poi passare allo stato successivo SAVE\_COLUMNS in cui l'output corretto verrà salvato.

## **SAVE\_COLUMNS (S3)**

Salva il valore di uscita dalla memoria, per poi passare allo stato successivo ASK\_LINES e ripetere lo stesso procedimento con la lettura delle righe.

## **AKS\_LINES (S4)**

Ha la stessa funzione dello stato ASK\_COLUMNS, ma questa volta l'indirizzo da cui leggere sarà quello corrispondente alla seconda cella della memoria, contenente il valore del numero di righe.

Si passa poi allo stato successivo WAIT\_LINES.

## **WAIT\_LINES (S5)**

La sua funzione è la stessa di WAIT\_COLUMNS. I segnali vengono quindi riconfermati per poi passare allo stato successivo SAVE\_LINES.

## **SAVE\_LINES (S6)**

Salva il valore di uscita dalla memoria. A questo punto sono stati salvati sia il numero di righe che di colonne, quindi si passa allo stato CALCULATE\_DIM.

## **CALCULATE\_DIM (S7)**

Si occupa di calcolare e salvare la dimensione dell'immagine usando il numero di righe e di colonne letti precedentemente.

Avviene poi il passaggio allo stato INIT\_MAX\_MIN.

## **INIT\_MAX\_MIN (S8)**

Se la foto ha dimensione nulla si passa allo stato SET\_DONE perché non c'è nulla in memoria su cui poter lavorare. Se invece la dimensione non è nulla, vengono inizializzati i segnali riguardanti i pixel di valore massimo e minimo e memorizzati gli indirizzi per leggere e scrivere in memoria il primo pixel.

Avviene poi il passaggio allo stato FIND\_MAX\_MIN.

### **FIND\_MAX\_MIN (S9)**

Confronta i valori del pixel in ingresso con il valore massimo e minimo trovati fino a quel momento, aggiornandoli se necessario con il nuovo valore letto. Aumenta allora l'indirizzo di lettura da mandare alla RAM e ritorna su sé stesso, in modo da ripetere lo stesso processo per il pixel successivo dell'immagine. Al termine, esauriti i pixel dell'immagine, si passa allo stato CALCULATE\_SHIFT.

### **CALCULATE\_SHIFT (S10)**

Si occupa del calcolo del *delta\_value* e dello *shift\_level* e prepara i segnali per la successiva lettura dell'indirizzo di memoria in cui risiede il valore del primo pixel da modificare.

### **WAIT\_PIXEL (S11)**

Riconferma i segnali dello stato precedente riguardanti la lettura in memoria per assicurarsi che non ci siano eventuali errori nell'output della memoria stessa. La sua funzione è quindi la stessa dei due stati di WAIT precedentemente descritti.

Avviene poi il passaggio allo stato successivo MODIFY\_PIXEL.

### **MODIFY\_PIXEL (S12)**

Il valore del pixel letto in uscita dalla memoria viene opportunamente modificato utilizzando il valore dello *shift\_level* calcolato precedentemente. Viene poi preparata la memoria, aggiornando gli opportuni segnali, per la scrittura di tale valore.

Si passa poi allo stato WAIT\_WRITING che si occuperà di assicurarsi che la memoria sia effettivamente pronta per essere scritta.

### **WAIT\_WRITING (S13)**

Ha la stessa funzione degli altri stati di WAIT descritti in precedenza ma questa volta l'operazione è di scrittura, quindi viene riconfermato anche il segnale contenente il valore da scrivere in memoria.

Avviene poi il passaggio allo stato successivo MOVE\_ON.

### **MOVE\_ON (S14)**

Questo stato, dopo un opportuno controllo, si occupa di porre la memoria in condizioni tali da poter leggere il pixel dell'immagine successivo a quello appena elaborato, per poi tornare allo stato WAIT\_PIXEL, oppure, se sono stati modificati tutti i pixel dell'immagine, di far avvenire il passaggio allo stato SET\_DONE.

### **SET\_DONE (S15)**

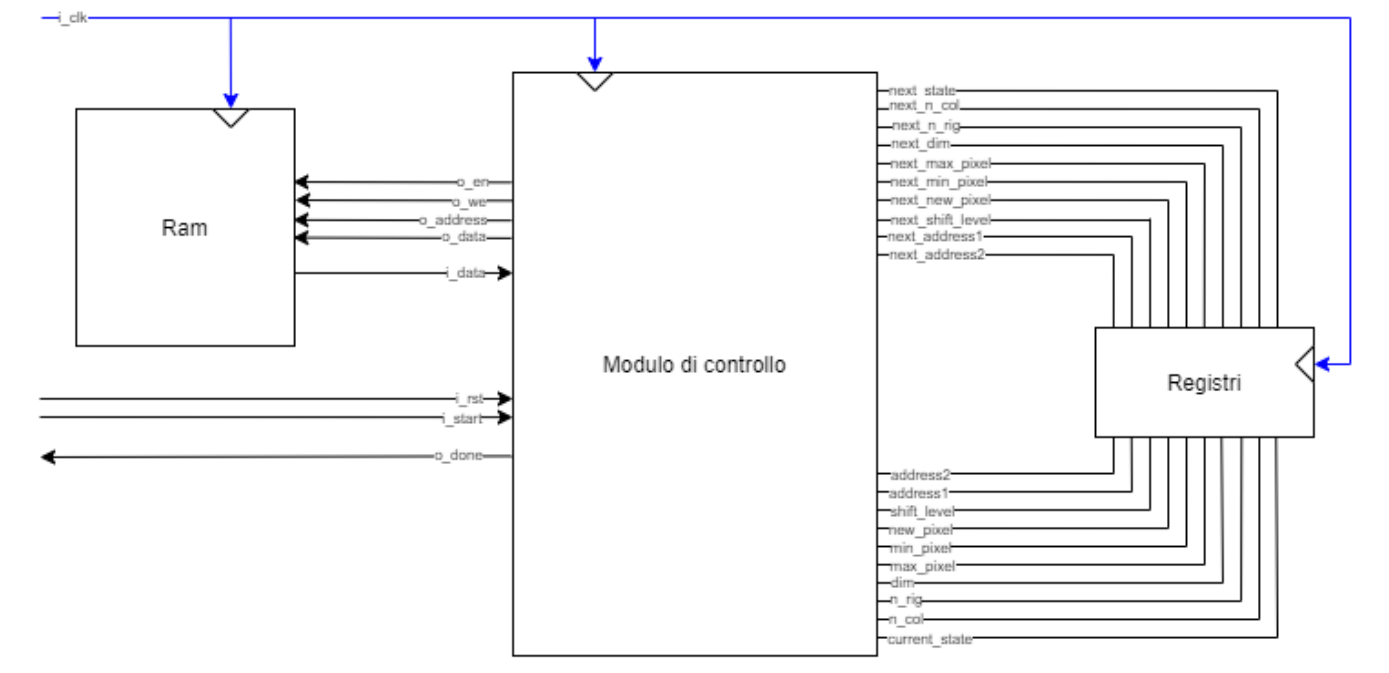
Viene portato a 1 il segnale di *done* per segnalare che l'operazione di modifica dell'immagine è stata terminata e si passa allo stato FINALIZE.

### **FINALIZE (S16)**

Questo stato attende che il segnale di *start* venga riportato a 0: se ciò non avviene allora si ritornerà sullo stesso stato FINALIZE, altrimenti lo stato si occupa di portare a 0 il segnale di *done* e il prossimo stato sarà quello di RESET, in modo che la macchina si prepari ad elaborare una nuova immagine.

## DESCRIZIONE DEI SEGNALI UTILIZZATI

Si riporta uno schema del modulo, dettagliato con i segnali:



Ogni segnale è presente, oltre che nella sua versione base *signal*, anche nella sua versione *next\_signal*. Il processo **main** utilizza per le valutazioni dei valori utili la versione base dei segnali e, qualora volesse aggiornarli, salva il loro nuovo valore nella corrispondente versione *next\_signal*, che è utilizzata dal processo **state\_update** per effettuare l'aggiornamento concreto del valore della versione base *signal*.

```
elsif rising_edge(i_clk) then
    -- aggiornamento dei segnali: current_x <- next_x
    current_state <= next_state;
    address1 <= next_address1;
    address2 <= next_address2;
    n_col <= next_n_col;
    n_rig <= next_n_rig;
    dim <= next_dim;
    max_pixel <= next_max_pixel;
    min_pixel <= next_min_pixel;
    shift_level <= next_shift_level;
    new_pixel <= next_new_pixel;

end if;
```

*current\_state*: Questo segnale contiene il valore dello stato attuale della macchina.

*next\_state*: Contiene il valore dello stato futuro della macchina.

*n\_col* e *n\_rig*: Questa coppia di segnali è atta alla memorizzazione rispettivamente del numero di colonne e del numero di righe.

*dim*: Contiene la dimensione dell'immagine, calcolata a partire dai due segnali precedenti, ed è utilizzato per il calcolo degli indirizzi usati per la lettura e scrittura della memoria.

*max\_pixel* e *min\_pixel*: Questa coppia di segnali è atta alla memorizzazione rispettivamente del valore massimo e minimo dei pixel trovati nell'immagine da elaborare.

*new\_pixel*: Segnale che contiene di volta in volta il valore calcolato del pixel nell'immagine elaborata.

*shift\_level*: Questo segnale è utile alla memorizzazione del valore dello *shift\_level* che serve per realizzare l'algoritmo da applicare ai pixel.


*address1* e *address2*: Segnali che sono utilizzati per tener conto dell'indirizzo di lettura e di scrittura dello stesso pixel durante il processo di elaborazione dell'immagine. Un pixel viene letto in memoria all'indirizzo *address1*, il suo valore è modificato e viene riscritto in memoria all'indirizzo *address2*.

Oltre ai segnali, nel codice *vhdl*, sono utilizzate delle variabili per ottimizzare lo svolgimento dei calcoli. Il loro valore è utile solo all'interno di un ciclo di clock, quindi non è necessario venga memorizzato mediante l'impiego di un segnale.



## Risultati Sperimentali

Il componente progettato è sintetizzabile in modo corretto.

Name	Constraints	Status	WNS	TNS	WHS	THS	TPWS	Total Power	Failed Routes	LUT	FF
▼  synth_1	constrs_1	Synthesis Out-of-date								234	97
✓ impl_1	constrs_1	route_design Complete!	NA	NA	NA	NA	NA	5.736	0	234	97

Si riporta la tabella di utilizzo delle risorse in fase di post-Sintesi, corredata da parti ritenute significative del report di sintesi:

Resource	Estimation	Available	Utilization %
LUT	234	134600	0.17
FF	97	269200	0.04
IO	38	285	13.33
BUFG	1	32	3.13

### Detailed RTL Component Info :

#### +---Adders :

```

      3 Input   17 Bit      Adders := 1
      2 Input   16 Bit      Adders := 4
      3 Input    8 Bit      Adders := 2

```

#### +---Registers :

```

      16 Bit      Registers := 3
      8 Bit       Registers := 6
      5 Bit       Registers := 1

```

#### +---Muxes :

```

      2 Input   16 Bit      Muxes := 3
     18 Input   16 Bit      Muxes := 1
     18 Input    8 Bit      Muxes := 2
      2 Input    8 Bit      Muxes := 1
      2 Input    4 Bit      Muxes := 4
     18 Input    3 Bit      Muxes := 1
      2 Input    3 Bit      Muxes := 4
      2 Input    2 Bit      Muxes := 3
     18 Input    1 Bit      Muxes := 8
      6 Input    1 Bit      Muxes := 1

```

### Part Resources:

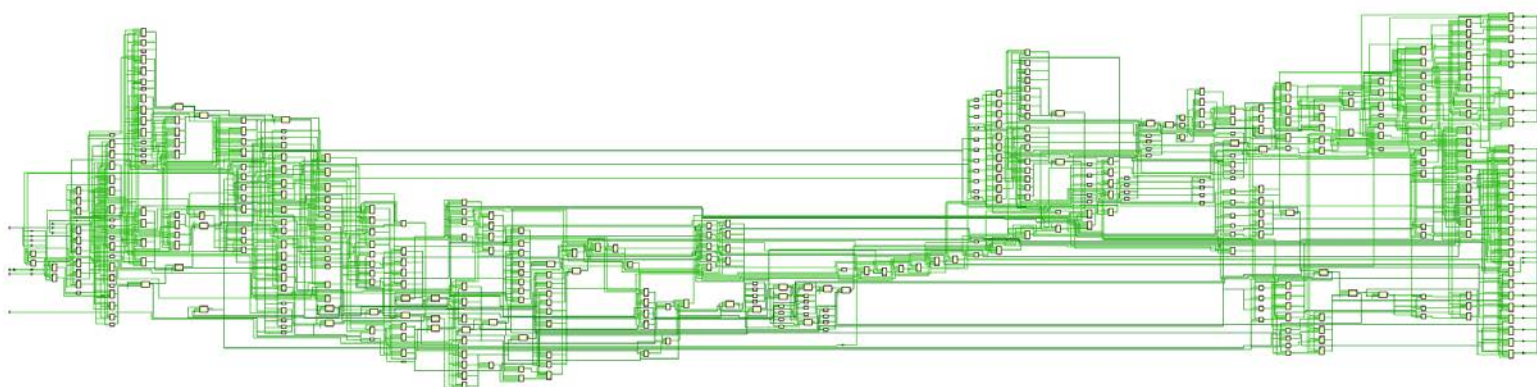
DSPs: 740 (col length:100)

BRAMs: 730 (col length: RAMB18 100 RAMB36 50)

### 7. Primitives

Ref Name	Used	Functional Category
FDCE	97	Flop & Latch
LUT6	87	LUT
LUT4	86	LUT
LUT2	75	LUT
CARRY4	38	CarryLogic
LUT5	31	LUT
OBUF	27	IO
LUT3	26	LUT
IBUF	11	IO
LUT1	1	LUT
BUFG	1	Clock

Lo schema logico risultante è il seguente:

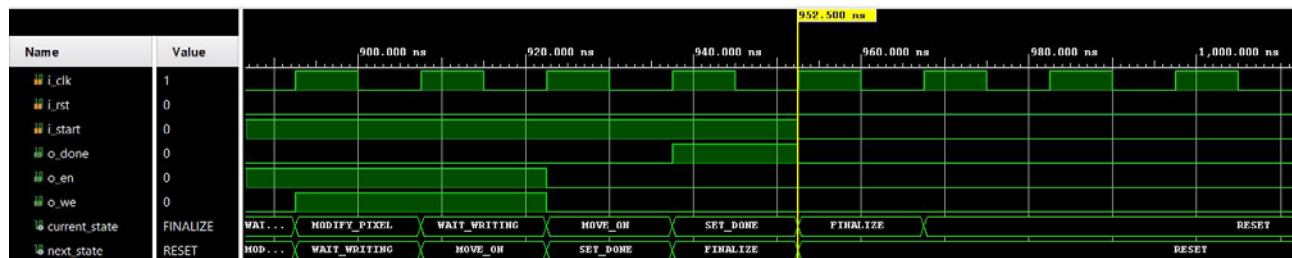


## Simulazioni

Qui di seguito si riportano tutti i Test Bench significativi usati per testare il funzionamento del componente in diversi casi di esecuzione.

### Elaborazione di immagini multiple

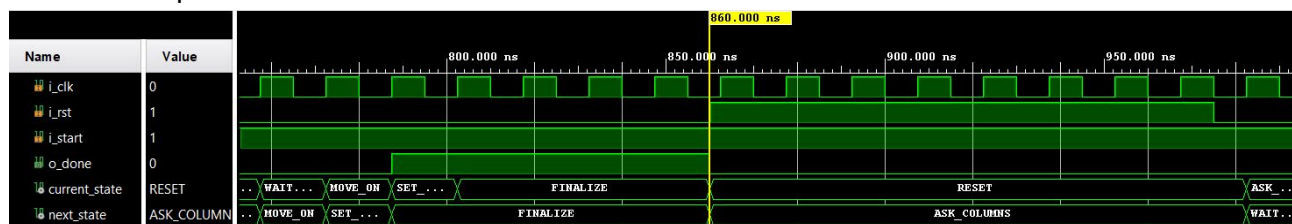
Si testa il componente nel caso particolare in cui è richiesta l'elaborazione di più immagini in modo consecutivo. Al termine della modifica di ogni immagine, viene portato a 1 il segnale di *o\_done* e, in caso di successiva ricezione del segnale di *i\_start*, si inizia l'elaborazione dell'immagine seguente.



Appena viene alzato il segnale di *o\_done* e *i\_start* viene portato basso, il componente torna infatti nello stato di RESET, nel quale si mette in attesa di un successivo segnale di *start*, alla cui ricezione fa partire l'elaborazione di una nuova immagine.

### Ricezione di un reset asincrono

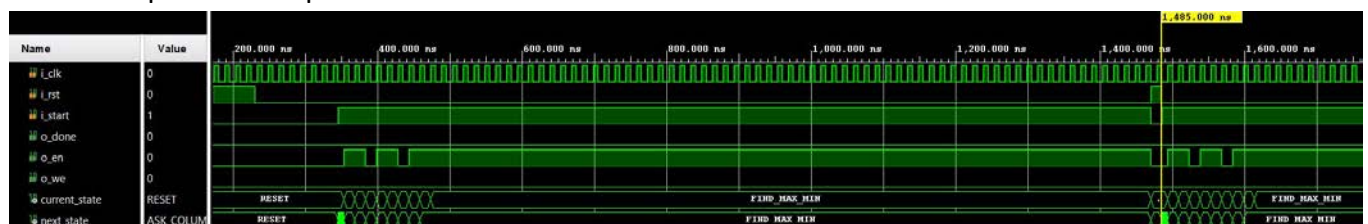
Il componente viene testato nel caso particolare in cui, durante l'elaborazione di un'immagine, riceve un segnale di *reset*. In questa situazione, appena ricevuto il segnale, deve interrompere il processo e ritornare in uno stato iniziale e, se richiesto, ricominciare l'elaborazione dell'immagine presente in memoria in quel momento.



Il segnale di *reset* viene portato alto prima che il segnale di *start* venga riportato basso, quindi si testa la capacità del componente di saper gestire la situazione in cui è necessario tornare allo stato di RESET e ripetere tutti i passaggi di lettura e modifica della stessa immagine.

### Elaborazione di immagini multiple con reset asincrono

Questo tipo di test unisce i due precedentemente descritti, e serve per capire se il componente è in grado di gestire in contemporanea queste due situazioni. Durante l'elaborazione di una delle immagini, il componente riceve un segnale di *reset* che lo porta a dover ricominciare l'elaborazione dell'immagine su cui stava operando in quell'istante.

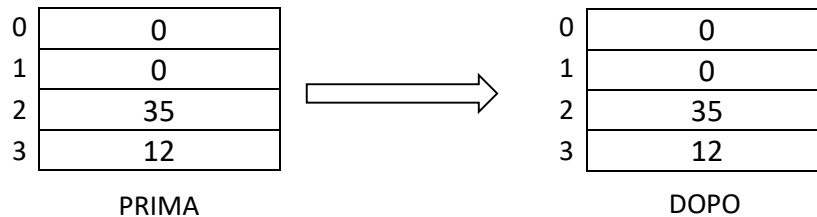


Quando poi l'elaborazione dell'immagine è terminata, il componente attende un successivo segnale di *start* per l'elaborazione dell'immagine successiva.

### Immagine di dimensione nulla

Si vuole osservare il comportamento del componente nel caso particolare in cui l'immagine inserita in memoria sia composta da 0 pixel e abbia quindi una dimensione nulla. Almeno una delle prime due celle di memoria deve quindi contenere il valore 0 che indica l'assenza di righe o di colonne (oppure di entrambe).

Per controllare il corretto funzionamento, vengono riempite la terza e la quarta cella della memoria con pixel di valore qualunque, in modo da poter controllare a fine elaborazione che questi valori non siano stati modificati.



### Pixel tutti del medesimo valore

Si eseguono 3 test differenti, al fine di verificare il funzionamento del componente prima nel caso generale in cui i pixel sono tutti uguali e poi nei due casi limite, in cui contengono tutti o il valore massimo o il valore minimo.

Con un primo Test Bench viene inserita in memoria un'immagine con pixel tutti di valore 1 per testare il caso in cui abbiano tutti uguale valore.

```
signal RAM: ram_type := (0 => std_logic_vector(to_unsigned( 2 , 8)),
                          1 => std_logic_vector(to_unsigned( 2 , 8)),
                          2 => std_logic_vector(to_unsigned( 1 , 8)),
                          3 => std_logic_vector(to_unsigned( 1 , 8)),
                          4 => std_logic_vector(to_unsigned( 1 , 8)),
                          5 => std_logic_vector(to_unsigned( 1 , 8)),
                          others => (others => '0'));
```

Un secondo Test Bench, invece, prevede l'inserimento in memoria di un'immagine con pixel tutti di valore nullo.

```
signal RAM: ram_type := (0 => std_logic_vector(to_unsigned( 2 , 8)),
                          1 => std_logic_vector(to_unsigned( 2 , 8)),
                          2 => std_logic_vector(to_unsigned( 0 , 8)),
                          3 => std_logic_vector(to_unsigned( 0 , 8)),
                          4 => std_logic_vector(to_unsigned( 0 , 8)),
                          5 => std_logic_vector(to_unsigned( 0 , 8)),
                          others => (others => '0'));
```

Infine, con un terzo Test Bench, si considera un ulteriore caso limite, in cui l'immagine presente in memoria è composta da pixel tutti di valore massimo.

```
signal RAM: ram_type := (0 => std_logic_vector(to_unsigned( 2 , 8)),
                          1 => std_logic_vector(to_unsigned( 2 , 8)),
                          2 => std_logic_vector(to_unsigned( 255 , 8)),
                          3 => std_logic_vector(to_unsigned( 255 , 8)),
                          4 => std_logic_vector(to_unsigned( 255 , 8)),
                          5 => std_logic_vector(to_unsigned( 255 , 8)),
                          others => (others => '0'));
```

È necessario, per una buona copertura, eseguire tutti e tre i test, perché il primo considera un valore generico, mentre gli altri due i valori limite. In tutti i casi, si devono modificare i pixel affinché assumano il valore 0.

### Immagine di dimensione minima e di dimensione massima

Per eseguire il controllo sull'elaborazione di immagini di dimensione massima e minima, sono necessari due Test Bench: uno per ognuno dei due casi limite.

Il primo testa la capacità del componente di modificare un'immagine di dimensione unitaria, producendone una di un unico pixel di valore nullo come output.

Il secondo, invece, fa sì che il componente elabori una foto con dimensioni 128x128.

### Pixel di valore massimo e minimo situati agli estremi dell'immagine

All'inizio dell'immagine si pone il pixel di valore minimo e alla fine quello di valore massimo, con lo scopo di testare il corretto funzionamento del componente in questo caso limite.

Il Test Bench ha il fine di verificare che si prendano in considerazione i valori dei pixel situati agli estremi dell'immagine e in particolare che essi non vengano ignorati.

L'immagine riscritta in memoria dovrà avere il primo pixel a 0, mentre l'ultimo al valore massimo 255.

### Pixel di valore minimo 0 e valore massimo 255

Questo Test Bench verifica il caso in cui i valori minimo e massimo assunti dai pixel sono rispettivamente 0 e 255.

In questa situazione particolare nessun pixel verrà modificato, dato che il valore dello *shift\_level* sarà nullo. Dunque l'immagine verrà riscritta con gli stessi valori dei pixel iniziali.

2	0
2	1
35	2
255	3
146	4
0	5
35	6
255	7
146	8
0	9

## Conclusioni

Il modulo progettato secondo l'approccio scelto, in cui i valori utili sono salvati in segnali ciclicamente aggiornati, risulta funzionante, in quanto rispetta la specifica. Infatti, sono stati passati sia test specifici che test generati in modo casuale. Inoltre, in aggiunta al testing black-box è stata effettuata una analisi del codice per garantire la copertura delle istruzioni.

L'utilizzo di una macchina a stati finiti per realizzare le funzionalità principali rende il componente facilmente espandibile e modificabile, attraverso l'aggiunta o rimozione di stati.

Tra le scelte effettuate si riporta: la scelta di salvare il valore di due indirizzi diversi, per la lettura e scrittura in memoria dei pixel da modificare, ha evitato che si dovesse ogni volta calcolare l'indirizzo di scrittura a partire da quello di lettura sommando la dimensione dell'immagine, che sarebbe stata un'operazione più onerosa rispetto al semplice incremento di un'unità effettuato.

Il fatto di avere degli stati di attesa, che non fanno altro che occupare un ciclo di clock rimandando al successivo la lettura di segnali di input provenienti dalla memoria, rende il componente maggiormente compatibile con memorie meno performanti.

### OTTIMIZZAZIONI POSSIBILI

Una ottimizzazione possibile consiste nel ridurre il numero di stati. Ciò è possibile, ad esempio, realizzando un unico stato che si occupi della fase di attesa e sostituisca tutti gli stati del tipo WAIT\_X utilizzati. Sarebbe in questo caso necessario tenere traccia dello stato da raggiungere successivamente, per esempio mediante un segnale apposito.

Un'altra possibilità risiede nell'unire stati che svolgono funzioni che non richiedono di essere eseguite in cicli di clock distinti e successivi: le funzionalità dello stato SAVE\_COLUMNS e di ASK\_LINES possono essere svolte da uno stesso stato apposito, e lo stesso discorso vale per la coppia di stati SAVE\_LINES e CALCULATE\_DIM, che si potrebbero unire introducendo opportune variabili per lo svolgimento dei calcoli, in modo da non rendere necessario aspettare l'aggiornamento dei segnali, che avviene nel ciclo successivo.