Progetto di Reti Logiche

Prof. William Fornaciari

Arturo Caliandro 10610910
Vincenzo Converso 10625358

AA 2020-2021

Indice

Capitolo 1

Introduzione

1.1 Descrizione generale del progetto

Il progetto consiste nell'implementazione di un componente hardware, descritto in VHDL, che legga da memoria la dimensione (espressa in numero di colonne e numero di righe) e i valori dei pixel, riga per riga, e scriva in memoria la stessa immagine, dopo averne equalizzato l'istogramma[1].

Esempio

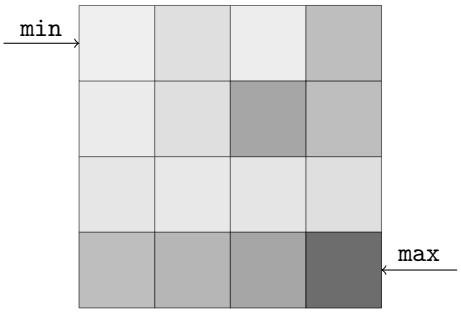


Figura 1.1: Immagine non equalizzata. `min`=16 e `max`=144

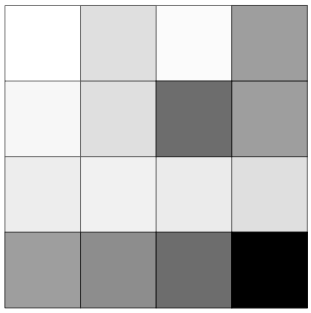


Figura 1.2: Immagine equalizzata dal componente.

Si può notare come `min` e `max` assumano, dopo l'elaborazione, rispettivamente il valore minimo (=0) e il valore massimo (=255) possibile.

1.2 Specifiche progettuali

L'algoritmo da implementare è una versione semplificata: il componente hardware opera su immagini in scala di grigi a 256 livelli, di dimensione massima 128x128 pixel.

1.2.1 Descrizione della memoria

La memoria è indirizzata al byte, ed è così strutturata:

- in posizione `0` e `1` vi sono rispettivamente il numero di colonne (`N-COL`) e di righe (`N-RIG`) dell'immagine;

[1] L'equalizzazione dell'istogramma è un'operazione che ridistribuisce i valori di intensità dei pixel di un'immagine, in modo da ricoprire tutta la gamma di valori possibile, ed aumentarne perciò il contrasto globale.

- dalla posizione 2 alla posizione 1+(N-COL*N-RIG) vi sono i valori (da 0 a 255) dei pixel che costituiscono l'immagine;
- dalla posizione 2+(N-COL*N-RIG) alla posizione 2+2*(N-COL*N-RIG), infine, vi sono i valori equalizzati dei pixel dell'immagine.

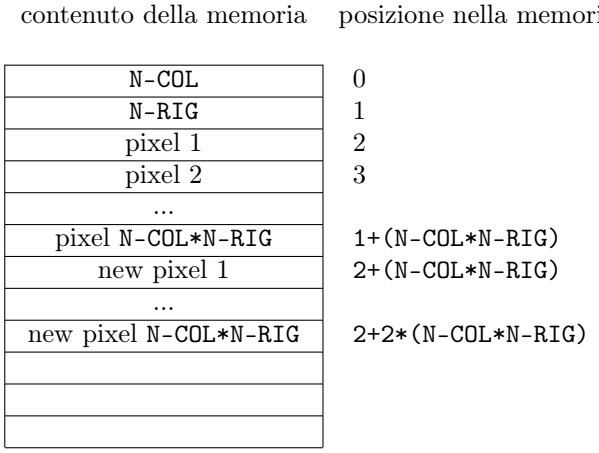


Figura 1.3: rappresentazione schematica della posizione in memoria dei dati.

1.2.2 Interfaccia del componente hardware

Il componente da descrivere ha un'interfaccia così definita:

```
entity project_reti_logiche is
        port (
                i_clk           : in std_logic;
                i_start         : in std_logic;
                i_rst           : 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 test bench;
- `i_start` è il segnale di `START` generato dal test bench;
- `i_rst` è il segnale di `RESET` che inizializza la macchina pronta per ricevere il primo segnale di `START`;
- `i_data` è il segnale (vettore) che arriva dalla memoria in seguito ad una richiesta dilettura;
- `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 Funzionamento

Il componente deve cercare il valore minimo (`min_pixel_value`) e il valore massimo (`max_pixel_value`), in scala di grigi, tra i pixel dell'immagine. Una volta trovati i due valori estremi, il componente deve calcolare

```
delta_value = max_pixel_value - min_pixel_value
```

e successivamente

`shift_level = 8 - floor(`$\log_2$`(delta_value + 1))`.

A questo punto, il componente entra in un ciclo iterativo in cui, dato il valore dell'i-esimo pixel `current_pixel_value`, deve:

- calcolare `temp_pixel`, che corrisponde alla differenza `current_pixel_value - min_pixel_value`, che viene shiftata a sinistra di `shift_level` bit;
- confrontare `temp_pixel` con 255, e inserire il minore nella variabile `new_pixel_value;`
- scrivere nella cella `i + (N-COL * N-RIG)` il valore di `new_pixel_value`.

Terminata la scrittura dei pixel elaborati, il componente ha concluso l'esecuzione ed è pronto per una nuova elaborazione.

Capitolo 2

Architettura

Quando `i_start` viene portato a 1, il componente viene avviato e passa dallo stato di `reset` alla fase di elaborazione; dopo aver elaborato l'immagine in memoria, pone il segnale `o_done` a 1 e rimane in attesa. Il test bench, una volta osservato il valore alto di `o_done`, risponde ponendo a valore basso `i_start`, e a quel punto il componente abbassa a sua volta `o_done` e torna allo stato di `reset`, attendendo nuovamente un valore alto di `i_start`.

2.1 Diagramma degli stati

2.1.1 Descrizione degli stati

`reset` è lo stato iniziale, che attende il valore alto di `i_start` per entrare nel flusso di elaborazione.

`wait` è lo stato in cui viene atteso il test bench, per la lettura in memoria o per attendere l'abbassamento di `i_start`.

`read` è lo stato in cui il valore letto durante la permanenza in `wait` viene trasmesso ad un altro segnale, per renderlo usufruibile.

`get_column` **e** `get_line` sono gli stati in cui viene interrogata la memoria rispettivamente per numero di colonne e di righe dell'immagine in memoria.

`get_last_p` è lo stato in cui viene calcolato l'indirizzo dell'ultimo pixel in memoria.

`check_curr_p_address` è lo stato in cui viene confrontato l'indirizzo del prossimo pixel da controllare, portando la macchina al calcolo di `shift_level` quando ha letto tutti i pixel, o qualora fossero stati trovati due pixel di valore 0 e 255.

`check_p_value` è lo stato che si occupa di verificare se il pixel in analisi abbia un valore minore/maggiore del pixel minore/maggiore trovato fino a quel punto, ed eventualmente lo sovrascrive. Successivamente incrementa l'indirizzo del pixel da analizzare.

`get_shift`[1] è lo stato che si occupa di calcolare lo `shift_level`.

`get_pixel` è analogo a `check_curr_p_address`, ma esegue l'incremento dell'indirizzo del prossimo pixel da elaborare e, una volta finita la fase di elaborazione, alza il valore `o_done` e porta la macchina a `wait`.

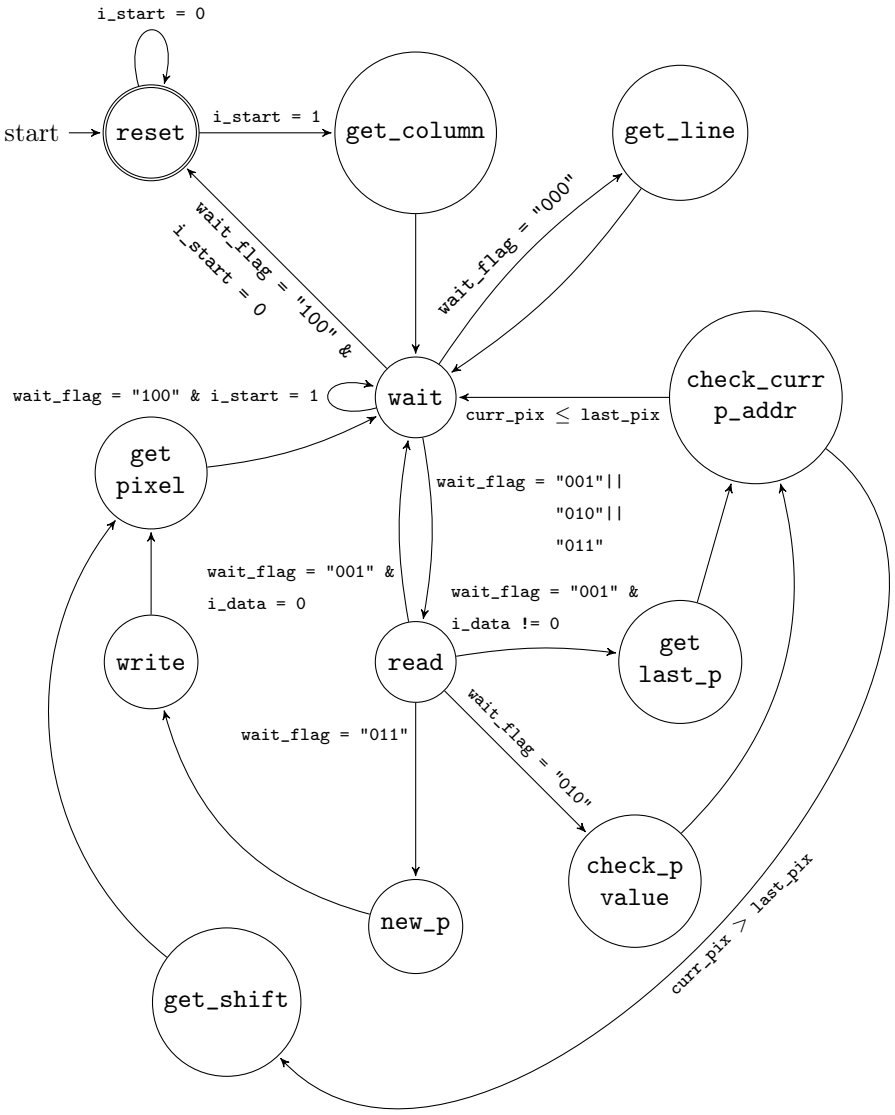
[1] Il calcolo del `delta_value` è bypassato eseguendo dei confronti a soglia.

`new_p` è lo stato in cui viene effettuato il calcolo del nuovo valore del pixel in ingresso.

`write` è lo stato in cui la macchina accede alla memoria per scrivere il nuovo valore del pixel. In seguito, incrementa l'indirizzo della prossima cella in cui inserire il pixel elaborato.

2.1.2 Pallogramma

La figura mostra il diagramma degli stati della macchina.



2.2 Scelte di progettazione

La principale scelta da effettuare era nel calcolo dello `shift_level`: per ridurre il numero di stati complessivo, abbiamo scelto di confrontare $\Delta + 1$ con intervalli costituiti da potenze di 2 successive. In questo modo, non è necessario implementare un registro che tenga traccia di `delta_value`.

$\Delta + 1$	`shift_level`
$= 1$	8
$\geq 2 \wedge < 4$	7
$\geq 4 \wedge < 8$	6
$\geq 8 \wedge < 16$	5
$\geq 16 \wedge < 32$	4
$\geq 32 \wedge < 64$	3
$\geq 64 \wedge < 128$	2
$\geq 128 \wedge < 256$	1
$= 256$	0

Tabella 2.1: valori assunti da `shift_level` a seconda del risultato dell'operazione.

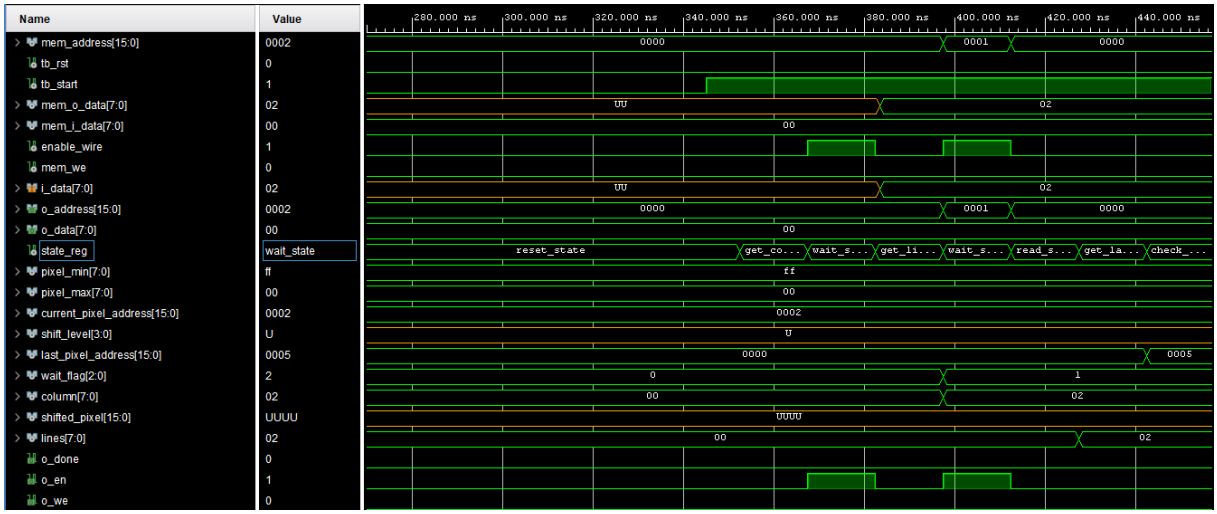
Successivamente, abbiamo optato per usare `shift_level` come flag per identificare la potenza di 2 da moltiplicare a `current_p_value` - `pixel_min`.

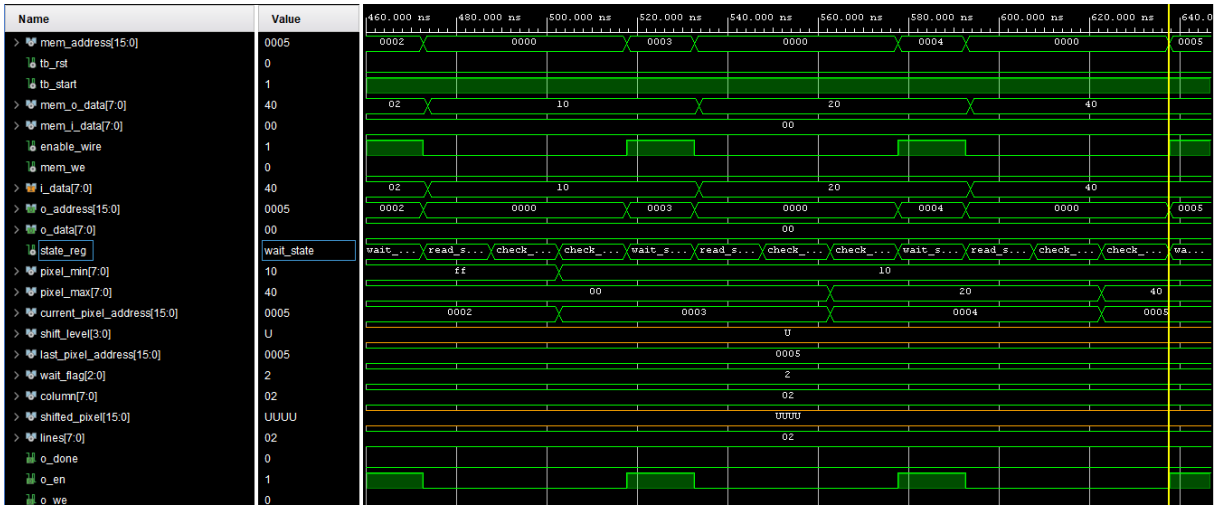
Un'altra scellta effettuata è quella di ridurre il numero di stati funzionali all'attesa della memoria, inglobando in `wait` e `read` tutte le attese di risposta della memoria/copia dei dati richiesti dai vari stati: il segnale `wait_flag` serve ad indicare al componente in quale stato dovrà recarsi nel prossimo fronte di salita del clock.

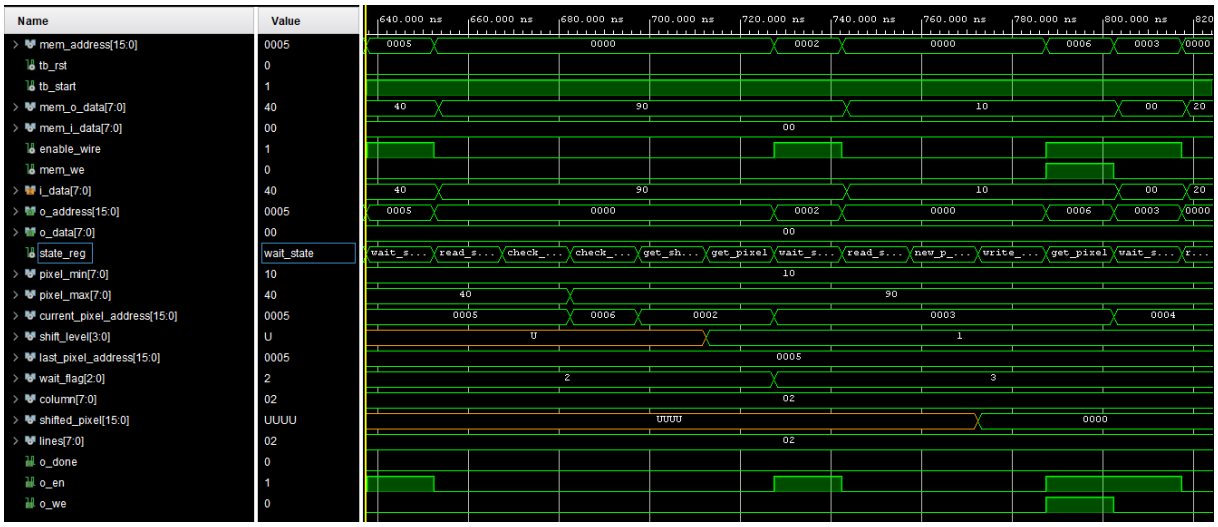
Capitolo 3

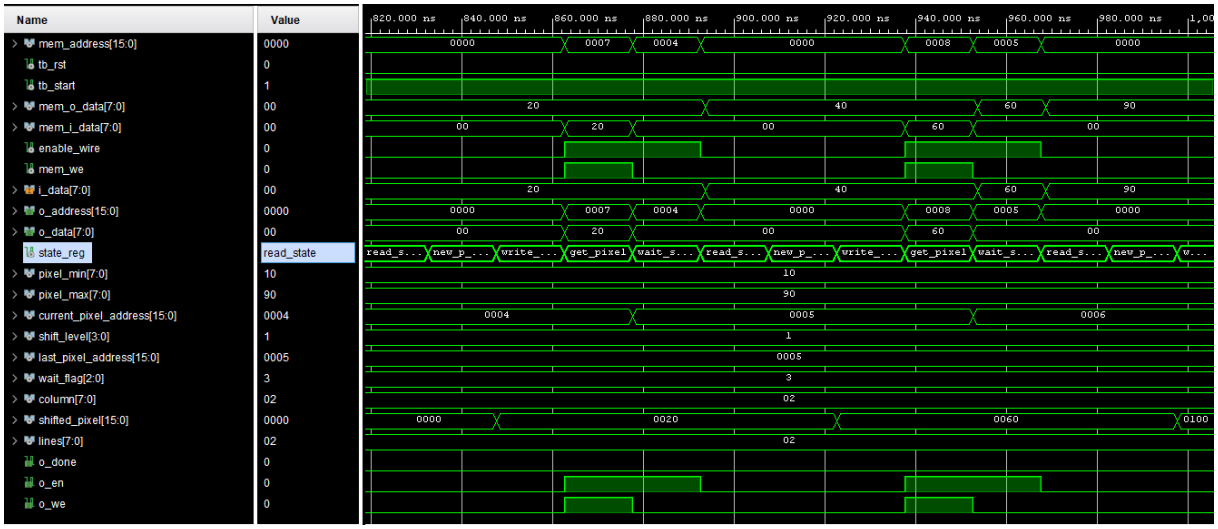
Risultati sperimentali

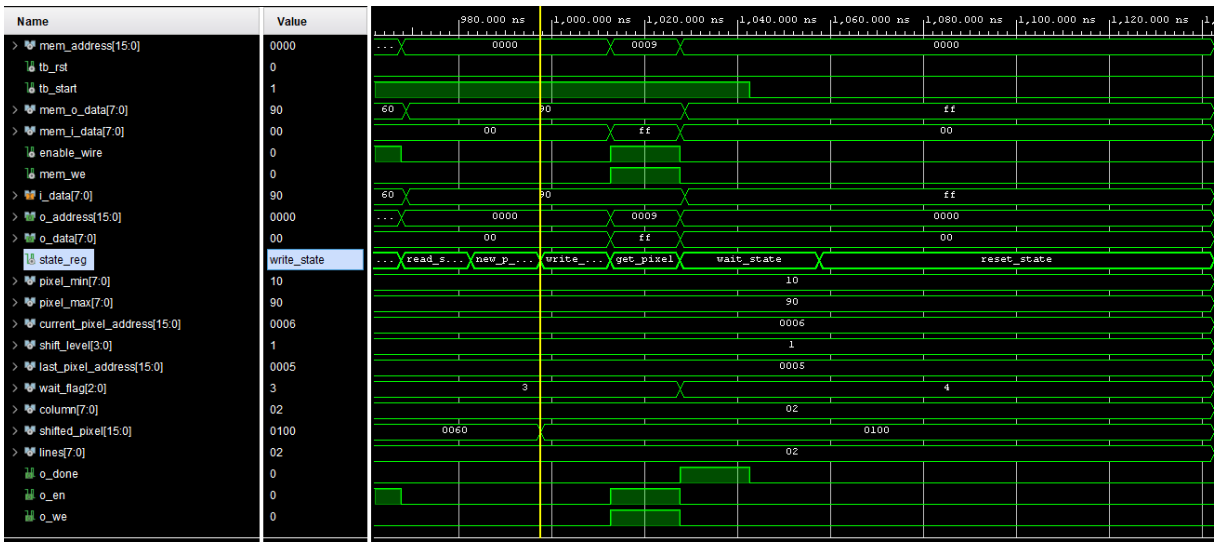
Il primo test effettuato ha come input un quadrato di 4 pixel, di cui nessun estremo. Per questo test abbiamo deciso di riprendere l'intero diagramma d'onda per screenshot:






Name
Value
mem_address[15:0] 0005
tb_rst 0
tb_start 1
mem_o_data[7:0] 40
mem_i_data[7:0] 00
enable_wire 1
mem_we 0
i_data[7:0] 40
o_address[15:0] 0005
o_data[7:0] 00
state_reg wait_state
pixel_min[7:0] 10
pixel_max[7:0] 40
current_pixel_address[15:0] 0005
shift_level[3:0] U
last_pixel_address[15:0] 0005
wait_flag[2:0] 2
column[7:0] 02
shifted_pixel[15:0] UUUU
lines[7:0] 02
o_done 0
o_en 1
o_we 0


Name
Value
mem_address[15:0] 0000
tb_rst 0
tb_start 1
mem_o_data[7:0] 00
mem_i_data[7:0] 00
enable_wire 0
mem_we 0
i_data[7:0] 00
o_address[15:0] 0000
o_data[7:0] 00
state_reg read_state
pixel_min[7:0] 10
pixel_max[7:0] 90
current_pixel_address[15:0] 0004
shift_level[3:0] 1
last_pixel_address[15:0] 0005
wait_flag[2:0] 3
column[7:0] 02
shifted_pixel[15:0] 0000
lines[7:0] 02
o_done 0
o_en 0
o_we 0


Name
Value
mem_address[15:0] 0000
tb_rst 0
tb_start 1
mem_o_data[7:0] 90
mem_i_data[7:0] 00
enable_wire 0
mem_we 0
i_data[7:0] 90
o_address[15:0] 0000
o_data[7:0] 00
state_reg write_state
pixel_min[7:0] 10
pixel_max[7:0] 90
current_pixel_address[15:0] 0006
shift_level[3:0] 1
last_pixel_address[15:0] 0005
wait_flag[2:0] 3
column[7:0] 02
shifted_pixel[15:0] 0100
lines[7:0] 02
o_done 0
o_en 0
o_we 0

```
    -- Immagine originale =  [16 , 32, 64, 144]
    -- Immagine di output =  [0, 32, 96, 255]

    assert RAM(6) = std_logic_vector(to_unsigned( 0 , 8)) report "TEST FALLITO (WORKING ZONE). Expected  0  found " & integer'image(to_in
    assert RAM(7) = std_logic_vector(to_unsigned( 32 , 8)) report "TEST FALLITO (WORKING ZONE). Expected  32  found " & integer'image(to_
    assert RAM(8) = std_logic_vector(to_unsigned( 96 , 8)) report "TEST FALLITO (WORKING ZONE). Expected  96  found " & integer'image(to_
    assert RAM(9) = std_logic_vector(to_unsigned( 255 , 8)) report "TEST FALLITO (WORKING ZONE). Expected  255  found " & integer'image(t

    assert false report "Simulation Ended! TEST PASSATO" severity failure;
end process test;
```

Di seguito, invece, sono riportati i test di controllo dei casi particolari e dei casi limite, effettuati sul componente.

3.1 Casi particolari

3.1.1 Immagine monocromatica

Un'immagine monocromatica ha `min_pixel_value` come valore di ogni pixel, quindi l'immagine equalizzata sarà composta di soli pixel di valore `current_pixel_value - min_pixel_value` = `min_pixel_value - min_pixel_value` = 0.

```
    -- Immagine originale =  [4 , 4, 4, 4]
    -- Immagine di output =  [0, 0, 0, 0]

    assert RAM(6) = std_logic_vector(to_unsigned( 0 , 8)) report "TEST FALLITO (WORKING ZONE). Expected  0  found " & integer'image(to_intege
    assert RAM(7) = std_logic_vector(to_unsigned( 0 , 8)) report "TEST FALLITO (WORKING ZONE). Expected  0  found " & integer'image(to_intege
    assert RAM(8) = std_logic_vector(to_unsigned( 0 , 8)) report "TEST FALLITO (WORKING ZONE). Expected  0  found " & integer'image(to_intege
    assert RAM(9) = std_logic_vector(to_unsigned( 0 , 8)) report "TEST FALLITO (WORKING ZONE). Expected  0  found " & integer'image(to_intege

    assert false report "Simulation Ended! TEST PASSATO" severity failure;
l process test;
```

3.1.2 Immagine con entrambi i valori estremi della scala di grigi

In tal caso, $\Delta + 1 = 256$ e lo shift è 0. In più, `min_pixel_value` = 0, quindi l'immagine viene riscritta inalterata.

```
    -- Immagine originale =  [0 , 255, 60, 120]
    -- Immagine di output =  [0, 255, 60, 120]

    assert RAM(6) = std_logic_vector(to_unsigned( 0 , 8)) report "TEST FALLITO (WORKING ZONE). Expected  0  found " & integer'image(to_int
    assert RAM(7) = std_logic_vector(to_unsigned( 255 , 8)) report "TEST FALLITO (WORKING ZONE). Expected  255  found " & integer'image(to
    assert RAM(8) = std_logic_vector(to_unsigned( 60 , 8)) report "TEST FALLITO (WORKING ZONE). Expected  60  found " & integer'image(to_i
    assert RAM(9) = std_logic_vector(to_unsigned( 120 , 8)) report "TEST FALLITO (WORKING ZONE). Expected  120  found " & integer'image(to

    assert false report "Simulation Ended! TEST PASSATO" severity failure;
end process test;
```

3.2 Casi limite

3.2.1 Immagine degenere

Il test verifica che, data un'immagine di cui venga dato un parametro tra `N-COL` o `N-RIG` pari a 0, la macchina vada in stato di `wait` subito dopo tale constatazione, alzando `o_done` a 1 in quanto non vi è nessun pixel da elaborare. Se, in particolare, `N-COL` = 0, il caso è **ottimo**.

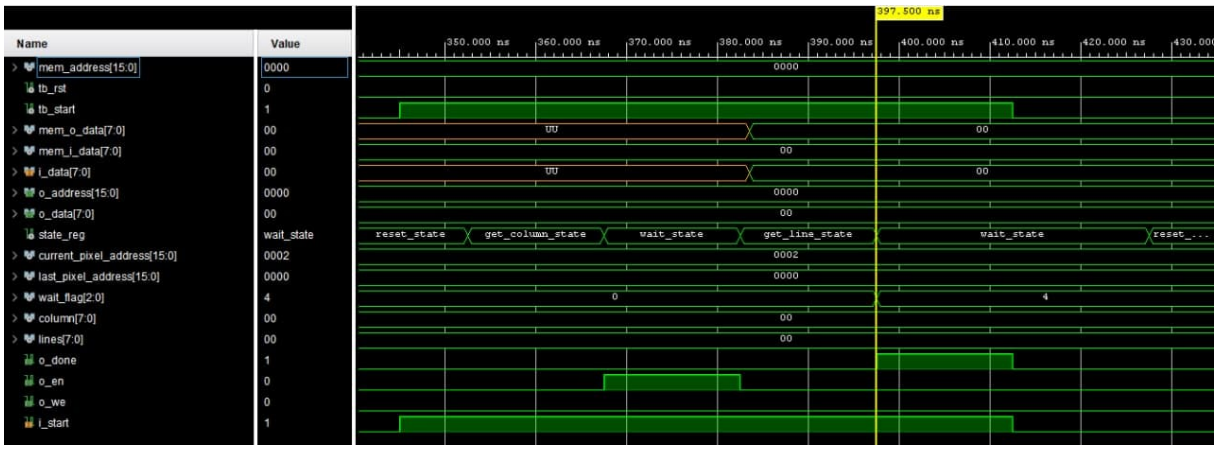


Figura 3.1: `N-COL` = 0: il controllo avviene in `get_line`.

3.2.2 Caso pessimo

Il caso pessimo, in termini di elaborazione, è dovuto a un'immagine di dimensione massima $(128 * 128)$, che non presenti entrambi i due valori estremi, o che li presenti entrambi ma uno di essi corrisponda all'ultimo pixel dell'immagine: in queste condizioni, il componente deve necessariamente controllare tutti i pixel prima di passare alla loro elaborazione.

Capitolo 4

Conclusioni

4.1 Risultati di sintesi

Il componente sintetizzato ha passato tutte le simulazioni disponibili: *behavioral, post-synthesis functional e post-synthesis timing.*

Le prestazioni del componente sono state valutate misurando la distanza temporale tra il fronte di salita di `i_start` e il fronte di salita di `o_done`:

Caso	Descrizione	T(S)
ottimo	Immagine degenere (`N-COL` = 0)	52.5 nS
pessimo	Immagine di dimensione massima	2212 µS

Tabella 4.1: prestazioni del componente in simulazione *behavioral*.

Per completezza, riportiamo la tabella Design Timing Summary, fornita da Vivado al termine della simulazione.

Setup

Worst Negative Slack (WNS):	6,013 ns
Total Negative Slack (TNS):	0,000 ns
Number of Failing Endpoints:	0
Total Number of Endpoints:	240

Hold

Worst Hold Slack (WHS):	0,150 ns
Total Hold Slack (THS):	0,000 ns
Number of Failing Endpoints:	0
Total Number of Endpoints:	240

Pulse Width

Worst Pulse Width Slack (WPWS):	7,000 ns
Total Pulse Width Negative Slack (TPWS):	0,000 ns
Number of Failing Endpoints:	0
Total Number of Endpoints:	142