Daniele Cicala

CODICE PERSONA: 10630561

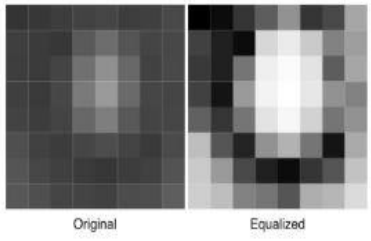
N° MATRICOLA: 910392

INDICE

1. INTRODUZIONE

1.1 PROGETTO

Il progetto si basa sull'implementazione di un componente hardware adibito all'equalizzazione di un'immagine, tramite la lettura di ogni pixel che la compone e la conseguente riscrittura del valore equalizzato.




Una immagine non equalizzata


La stessa immagine dopo l'equalizzazione dell'istogramma

1.2 ALGORITMO DI EQUALIZZAZIONE

L'algoritmo di equalizzazione si basa su una semplificazione dell'algoritmo dell'istogramma per la ricalibrazione delle immagini.

È definito nel seguente modo:

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)

MAX_PIXEL_VALUE e *MIN_PIXEL_VALUE* sono il valore massimo e minimo tra tutti i pixel che compongono l'immagine.

Dalla loro differenza calcoliamo il valore di *SHIFT_LEVEL* che verrà applicato ad ogni pixel.
Si tratta di un numero che va da 0, nel caso in cui la differenza sia esattamente 255, a 8, se invece il valore massimo e minimo coincidono.

Successivamente facciamo la sottrazione tra il valore di ogni pixel e quello minimo ed effettuiamo lo shift_level.
In ultimo luogo, se il valore calcolato è inferiore a 255, riscriviamo il valore ottenuto, altrimenti 255.

La riscrittura di ogni pixel verrà fatta a partire dal primo pixel successivo a quelli da leggere.

1.3 MEMORIA

La memoria è composta da celle di 8 bit ed è strutturata nel seguente modo:

- in posizione 0 vi è contenuto il valore relativo al numero di colonne dell'immagine, inferiore o uguale a 128.
- in posizione 1 vi è contenuto il valore relativo alle righe dell'immagine, inferiore o uguale a 128.
- dalla posizione 3 in poi saranno contenuti i valori di ogni pixel dell'immagine, fino alla posizione (n_colonne * n_righe) + 1.
- dalla posizione (n_colonne * n_righe) + 2 fino a (2 * n_colonne * n_righe) + 1 saranno presenti le celle di memoria in cui verranno riscritti i pixel equalizzati, con lo stesso ordine dei pixel originali.

0	NUMERO COLONNE
1	NUMERO RIGHE
2	PIXEL #1
3	PIXEL #2
4	PIXEL #3
5	PIXEL #4
6	PIXEL #1 EQUALIZZATO
7	PIXEL #2 EQUALIZZATO
8	PIXEL #3 EQUALIZZATO
9	PIXEL #4 EQUALIZZATO
...	

Tabella 1 - Esempio memoria 2x2

1.4 COMPONENTE

Il componente avrà la seguente interfaccia:

```
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
- **i_rst** è il segnale di reset in ingresso
- **i_start** è il segnale che da inizio alla lettura della memoria e al processo di equalizzazione
- **i_data (7 downto 0)** è il vettore in ingresso dalla memoria a seguito di una richiesta di lettura/scrittura
- **o_address (15 downto 0)** è il vettore che manda in uscita l'indirizzo alla memoria
- **o_done** è il segnale che comunica la fine del processo di equalizzazione
- **o_en** è il segnale che abilita la lettura e scrittura della memoria
- **o_we** è il segnale che abilita la scrittura della memoria
- **o_data (7 downto o)** è il vettore in uscita da scrivere nella memoria

Nel momento in cui **i_start** = 1 comincia il processo di equalizzazione che al suo termina porterà **o_done** = 1.

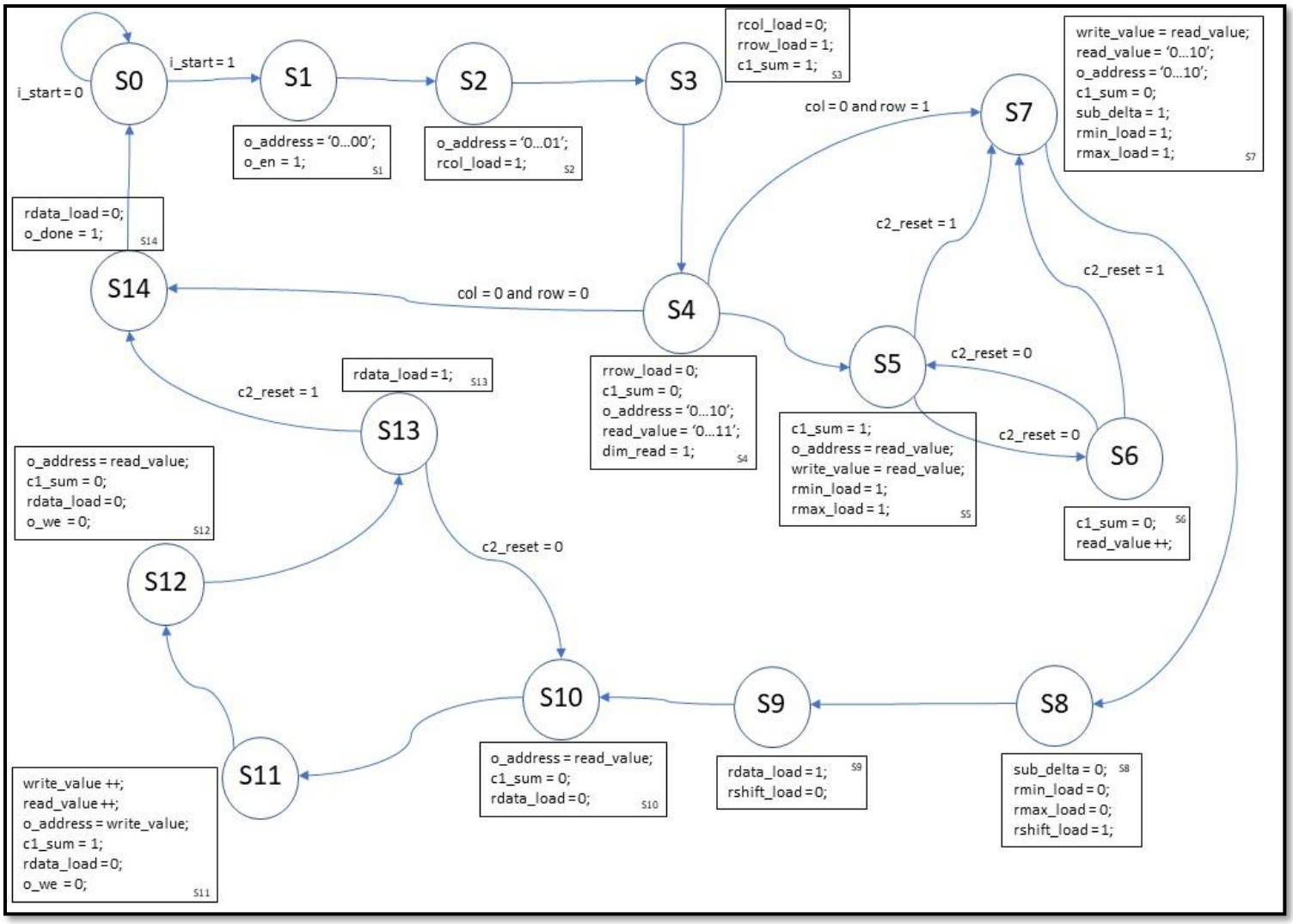
Il componente è costruito per poter permettere successive equalizzazioni, tuttavia sarà necessario aspettare che **o_done** torni a 0, per poter permettere a **i_start** di salire a 1, e quindi ricominciare il processo.

Si suppone di utilizzare il segnale **i_rst** solo per l'equalizzazione della prima immagine, pertanto nel caso di equalizzazione successive, il componente dovrà resettarsi autonomamente allo scadere del procedimento, senza il segnale **i_rst** = 1.

2. ARCHITETTURA

2.1 FSM

L'architettura è quella della seguente macchina di stati:



Sostanzialmente, verranno prima letti e salvati nei registri i valori relativi al numero di colonne e righe, e successivamente si procederà a due scansioni della memoria: nella prima si aggiornerà di volta in volta il valore minimo e massimo tra i pixel; nella seconda si ricomincerà dal primo pixel, per poi procedere in parallelo tra lettura del pixel originale dalla memoria e scrittura del pixel equalizzato in memoria.

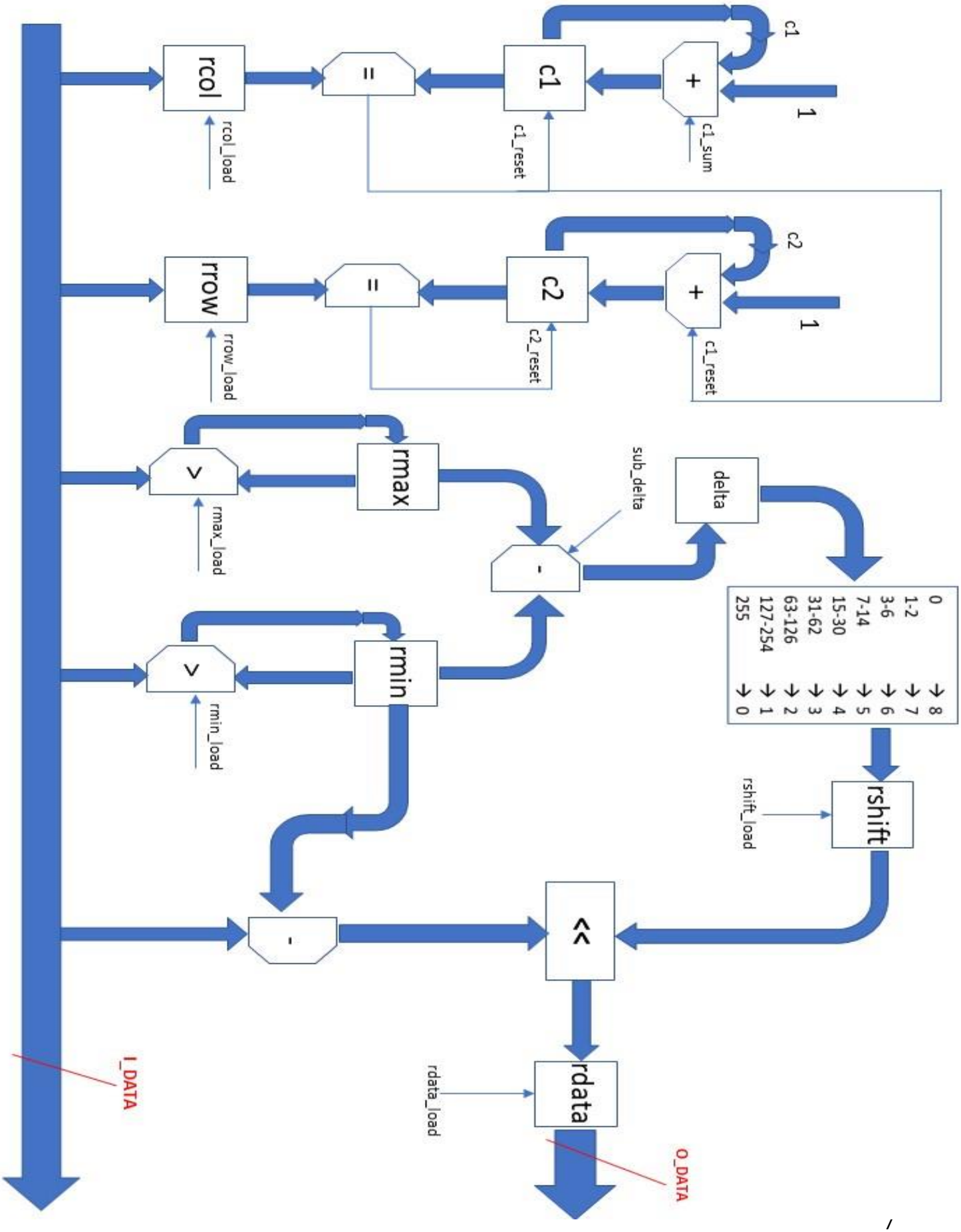
Per valutare dove si trovi l'ultimo pixel dell'immagine, si fa utilizzo di due contatori: uno per le colonne e uno per le righe.

2.2 STATI DELLA MACCHINA

Vediamo nel dettaglio tutti gli stati della macchina:

- **S0**: è lo stato iniziale e in cui si torna alla fine del processo di equalizzazione o a seguito di un segnale di reset. Rimane in attesa di **i_start** = 1.
- **S1**: è lo stato che inizia il processo di lettura dalla memoria, alzando **o_en** a 1.
- **S2**: salva il valore relativo al numero di colonne nel registro **o_rcol**.
- **S3**: salva il valore relativo al numero di righe nel registro **o_rrow**.
- **S4**: inizia la lettura della memoria a partire dal primo pixel. Contestualmente verifica se il numero di righe o colonne è uguale a 0, e nel caso va in **S14**, o se entrambi sono uguali a 1, e in tal caso va in **S7**, altrimenti va in **S5**.
- **S5**: incrementa di 1 il contatore delle colonne e confronta il pixel letto con quelli minimi e massimi, eventualmente sostituendone il valore con quello letto nei registri **o_rmin** e **o_rmax**. Nel caso in cui si sia arrivati all'ultimo pixel, va in **S7**, altrimenti in **S6**.
- **S6**: sposta l'indirizzo di memoria da cui leggere il prossimo pixel. Torna in **S5**.
- **S7**: nello stato **S7** è appena finita la prima lettura della memoria. Viene salvato il valore dell'indirizzo di memoria in cui si andrà successivamente a scrivere in **write_value**, mentre viene riportato all'indirizzo 2 ("0000000000000010") l'indirizzo di memoria da cui leggere.
- **S8**: salva nel registro **delta**, la differenza tra il valore massimo e minimo dei pixel dell'immagine.
- **S9**: salva il valore dello **shift_value** calcolato che verrà poi applicato nel processo di equalizzazione. Prepara al calcolo del pixel equalizzato con **rdata_load** = 1.
- **S10**: legge il valore in ingresso dalla memoria, lo sottrae al valore del pixel minimo, precedentemente salvato nel registro **o_rmin**, e applica la funzione **shift_level**. Salva il valore del pixel equalizzato nel registro **o_newdata**.
- **S11**: copia **o_address** da **write_value** dove verrà scritto il pixel equalizzato e lo scrive prendendone il valore da quello salvato nel registro **o_rnewdata**. Incrementa di 1 il contatore delle colonne **c1**, **read_value** e **write_value**.
- **S12**: copia **o_address** da **read_value**.
- **S13**: prepara al calcolo del pixel equalizzato con **rdata_load** = 1. Nel caso in cui l'ultimo pixel sia già stato equalizzato va in **S14**, altrimenti torna in **S10**.
- **S14**: stato finale. Pone **o_done** = 1 e torna in **S0**.

2.3 DESIGN PROGETTUALE



2.4 SEGNALI E REGISTRI

- **rcol_load (std_logic)**: quando a 1, il registro **o_rcol** salva il dato in ingresso.
- **o_rcol (std_logic_vector (7 downto 0))**: salva il numero di colonne.
- **rrow_load (std_logic)**: quando a 1, il registro **o_rrow** salva il dato in ingresso.
- **o_rrow (std_logic_vector (7 downto 0))**: salva il numero di righe.
- **dim_read (std_logic)**: viene posto a 1 dopo che vengono letti e registrati il numero di righe e colonne dell'immagine.
- **rmin_load (std_logic)**: quando a 1, il registro **o_rmin** verifica se il dato in ingresso è minore di quello precedentemente salvato, e nel caso lo sostituisce.
- **o_rmin (std_logic_vector (7 downto 0))**: salva il valore minimo tra i pixel dell'immagine.
- **rmax_load (std_logic)**: quando a 1, il registro **o_rmax** verifica se il dato in ingresso è maggiore di quello precedentemente salvato, e nel caso lo sostituisce.
- **o_rmax (std_logic_vector (7 downto 0))**: salva il valore massimo tra i pixel dell'immagine.
- **sub_delta (std_logic)**: quando a 1, salva in **delta** la differenza tra valore massimo e minimo.
- **delta (std_logic_vector (7 downto 0))**: salva la differenza tra massimo e minimo.
- **rshift_load (std_logic)**: quando a 1, il registro **o_rshift** salva lo shift_value calcolato dalla funzione **calc_shit_value** con parametro **delta**.
- **o_rshift (std_logic_vector (3 downto 0))**: salva il valore dello shift_value da applicare per l'equalizzazione.
- **c1 (std_logic_vector (7 downto 0))**: contatore del numero di colonne.
- **c2 (std_logic_vector (7 downto 0))**: contatore del numero di righe
- **c1_sum (std_logic)**: quando a 1, incrementa **c1** di 1.
- **c1_reset (std_logic)**: quando a 1, **c1** si resetta a 0 e incrementa **c2** di 1.
- **c2_reset (std_logic)**: quando a 1, **c2** si resetta a 0.
- **r_dataload (std_logic)**: quando a 1, **o_rnewdata** salva il valore del pixel equalizzato calcolato dalla funzione **shit_level** con parametri la differenza tra il dato in ingresso e il valore minore tra i pixel, e lo shift_value salvato in **o_rshift.**
- **o_rnewdata (std_logic_vector (7 downto 0))**: salva il valore da scrivere in uscita nella memoria.
- **read_value (std_logic_vector (15 downto 0))**: salva il valore dell'indirizzo di memoria da cui riprendere la lettura dei pixel.
- **write_value (std_logic_vector (15 downto 0))**: salva il valore dell'indirizzo di memoria da cui riprendere la scrittura dei pixel.
- **r_sum (std_logic)** : incrementa read_value di 1.
- **r_reset(std_logic)**: setta read_value a 2 ("0000000000000010").
- **w_sum(std_logic):** incrementa write_value di 1.
- **w_read(std_logic):** copia il valore di read_value in write_value.
- **w_sel:** quando a 1, o_address assume il valore di write_value, altrimenti assume il valore di read_value.

2.5 CONTATORI

I contatori **c1** e **c2** permettono la corretta scansione della memoria fino all'ultimo pixel disponibile.

Il funzionamento è semplice: il contatore **c1**, aumenta di 1 ogni qualvolta **c1_sum** = 1 e successivamente confronta il valore di **c1** con il valore contenuto nel registro **o_rcol**; se i due valori coincidono, allora **c1_reset** viene alzato a 1.

c1_reset = 1 comporta l'azzeramento del contatore **c1** e l'incremento di 1 del contatore **c2**.
Allo stesso modo, se **c2** coincide con **o_rrow**, allora si è arrivati alla fine della memoria e quindi sia **c1_reset** che **c2_reset** vengono alzati a 1, azzerando **c1** e **c2**.

2.6 CALC_SHIFT_VALUE

La funzione **calc_shift_value** calcola lo shift_value che verrà poi applicato durante il processo di equalizzazione.

```
function calc_shift_level(delta: std_logic_vector(7 downto 0)) return std_logic_vector is
    begin
    if(delta = "00000000") then
       return "1000";
    elsif(delta < "000000011") then
       return "0111";
    elsif(delta < "00000111") then
       return "0110";
    elsif(delta < "00001111") then
       return "0101";
    elsif(delta < "00011111") then
       return "0100";
    elsif(delta < "00111111") then
       return "0011";
    elsif(delta < "01111111") then
       return "0010";
    elsif(delta < "11111111") then
       return "0001";
    elsif(delta = "11111111") then
       return "0000";
    end if;
end function;
```

Questo valore va da 0 a 8 e viene calcolato a partire da **delta**, la differenza tra il massimo e minimo pixel dell'immagine.

Riassumendo:

- **delta** = 0 → shift_value = 8 – log(0+1) = 8 – log(1) = 8 – 0 = 8
- **delta** = 1 o 2 → shift_value = 8 – log(1+1) = 8 – log(2) = 8 – 1 = 7
- **delta** < 7 → shift_value = 6
- **delta** < 15 → shift_value = 5
- **delta** < 31 → shift_value = 4
- **delta** < 63 → shift_value = 3
- **delta** < 127 → shift_value = 2
- **delta** < 255 → shift_value = 1
- **delta** = 255 → shift_value = 8 – log(255+1) = 8 -log(256) = 8 – 8 = 0

Il valore di ritorno della funzione verrà poi assegnato al registro **o_rshiftvalue** quando

r_shiftload = 1.

2.7 SHIFT_LEVEL

La funzione **shift_level** calcola il pixel equalizzato che verrà poi scritto in memoria, a partire dalla differenza tra il pixel originale e quello minimo e il valore dello shift_value.

```
function shift_level(value: std_logic_vector(7 downto 0) ; shift_value: std_logic_vector(3 downto 0)) return std_logic_vector is
    begin
    if(value = "00000000") then
        return value;
    end if;
    case shift_value is
    when "0000" =>
       return value;
    when "0001" =>
       if(value(7) = '1') then
           return "11111111";
       end if;
       return (value(6 downto 0) & "0");
    when "0010" =>
       if(value(7 downto 6) >= "01") then
           return "11111111";
       end if;
       return (value(5 downto 0) & "00");
    when "0011" =>
       if(value(7 downto 5) >= "001") then
           return "11111111";
       end if;
       return (value(4 downto 0) & "000");
    when "0100" =>
       if(value(7 downto 4) >= "0001") then
           return "11111111";
       end if;
       return (value(3 downto 0) & "0000");
    when "0101" =>
       if(value(7 downto 3) >= "00001") then
           return "11111111";
       end if;
       return (value(2 downto 0) & "00000");
    when "0110" =>
       if(value(7 downto 2) >= "000001") then
           return "11111111";
       end if;
       return (value(1 downto 0) & "000000");
    when "0111" =>
       if(value(7 downto 1) >= "0000001") then
           return "11111111";
       end if;
       return (value(0) & "0000000");
    when "1000" =>
       return "11111111";
    when others =>
       return "00000000";
    end case;
end function;
```

La funzione inoltre limita autonomamente il valore di ritorno a 255, ossia calcola se eventualmente il valore calcolato sia superiore a 255 e nel caso ritorna direttamente 255, evitando un successivo controllo a posteriori.

Lo shift_level si tratta di uno spostamento a sinistra di tanti bit quanti sono quelli inidicati dallo shift_value, ossia una moltiplicazione per 1 (shift_value = 0), 2 (shift_value = 1), 4 (shift_value = 2), 8, 16, 32, 64, 128, o 256 (shift_value = 8).
Per prima cosa se il valore in ingresso è 0, ritorna semplicemente 0, indipendentemente dallo shift_value.
È facile poi notare due casi limite: se shift_value = 0, allora non viene effettuata alcuna moltiplicazione, perciò ritorna semplicemente il valore in ingresso; se invece shift_value = 8 qualsiasi sia il valore in ingresso sarà sempre maggiore di 255: perciò, eccezion fatta se il valore in ingresso è 0, ritorna direttamente 255.
In tutti gli altri casi viene fatto un primo controllo se il valore moltiplicato sarebbe superiore a 255, e nel caso ritorna 255, altrimenti ritorna solo una porzione del valore in ingresso seguita da tanti zeri quando vale shift_value.
Il valore di ritorno verrà poi assegnato al registro **o_rnewdata** quando **r_dataload** = 1, per poi essere scritto in memoria.

3. RISULTATI

3.1 SINTESI

Risultati derivanti dalla sintesi:

Site Type	Used	Fixed	Prohibited	Available	Util%
Registers	115	0	0	1450000	<0.01
Register as Flip Flop	115	0	0	1450000	<0.01
Register as Latch	0	0	0	1450000	0.00
CLB LUTs*	158	0	0	725000	0.02
LUT as Logic	158	0	0	725000	0.02
LUT as Memory	0	0	0	449920	0.00
LOOKAHEAD8	3	0	0	112480	<0.01

Tcl Console | Messages | Log | Reports | **Design Runs** | Power | DRC | Methodology | Timing

Name	Constraints	Status	WNS	TNS	WHS	THS	TPWS	Total Power	Failed Routes	LUT	FF	BRAM	URAM	DSP	Start	Elapsed
synth_1	constrs_1	synth_design Complete!								158	115	0.0	0	0	12/1/21, 9:37 PM	00:02:3
impl_1	constrs_1	route_design Complete!	96.082	0.000	0.083	0.000	0.000	11.026	0	154	115	0.0	0	0	12/1/21, 9:40 PM	00:04:2

```
Timing Report

Slack (MET) :             97.733ns  (required time - arrival time)
  Source:                 o_rmin_reg[3]/C
                            (rising edge-triggered cell FDPE clocked by clock  {rise@0.000ns fall@5.000ns period=100.000ns})
  Destination:            o_rnewdata_reg[4]/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:        1.949ns  (logic 0.759ns (38.943%)  route 1.190ns (61.057%))
  Logic Levels:           5  (LUT4=1 LUT5=1 LUT6=3)
  Clock Path Skew:        -0.145ns (DCD - SCD + CPR)
    Destination Clock Delay (DCD):    3.742ns = ( 103.742 - 100.000 )
    Source Clock Delay      (SCD):    3.932ns
    Clock Pessimism Removal (CPR):    0.045ns
  Clock Uncertainty:      0.171ns  (CJ)/2 + PE + PJ
    Clock Jitter            (CJ):    0.342ns
    Phase Error             (PE):    0.000ns
    Phase Jitter            (PJ):    0.000ns
```

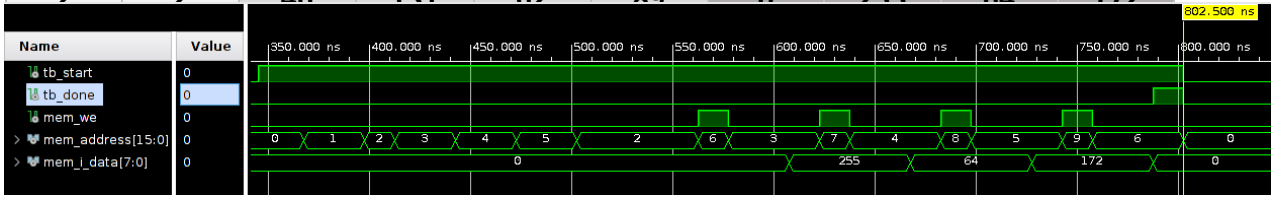
Lo slack generato è di 1.949 ns su un clock di 100 ns.

3.2 SIMULAZIONI

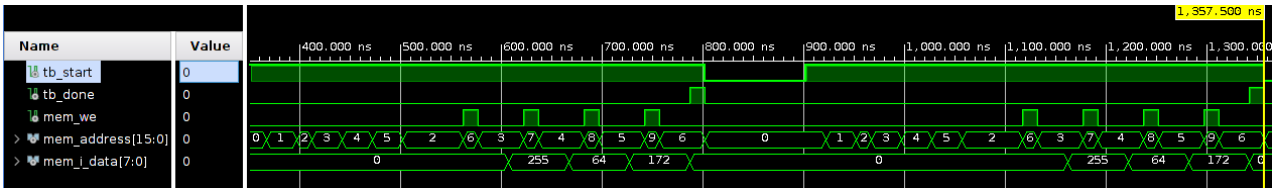
Oltre al testbench fornito sono stati effettuati altri 17 test:

1. testbench iniziale fornito dal docente.

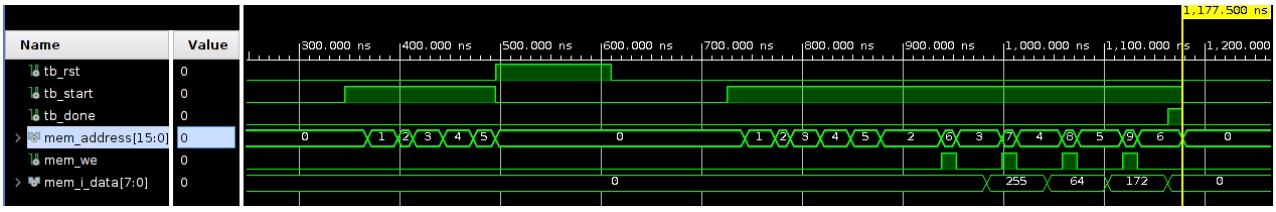
0	1	2	3	4	5	6	7	8	9
2	2	46	131	62	89	0	255	64	172



1bis. testbench iniziale eseguito di volte di seguito per verificare lo stato alla fine della prima computazione.

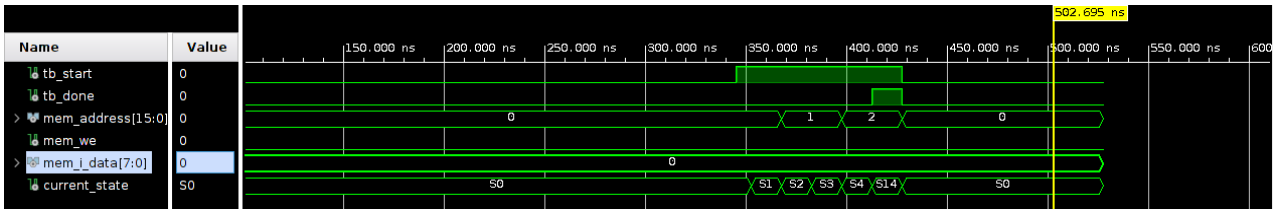


1reset. testbench iniziale resettato asincronicamente e ricominciato successivamente.



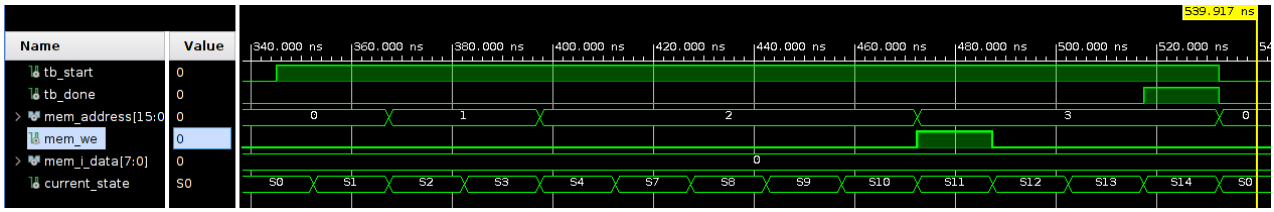
2. test con 0 colonne e 0 righe. Si nota il passaggio da S4 a S14 diretto.

0	**1**
0	**0**



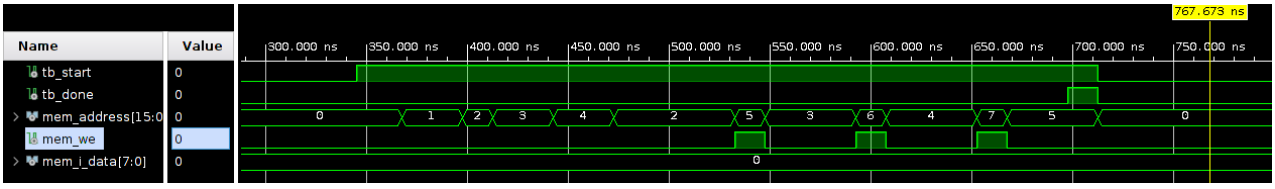
3. test con 1 colonna e 1 riga. Si nota il passaggio da S4 a S7 direttamente.

0	**1**	**2**	**3**
1	**1**	**46**	**0**



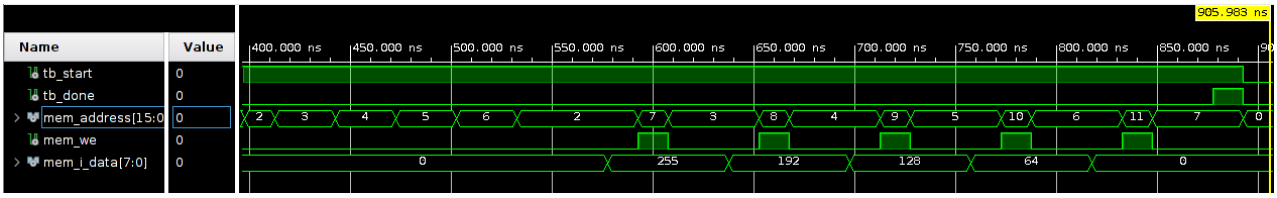
4. test con 1 riga. Inoltre test con valore massimo e minimo coincidenti.

0	**1**	**2**	**3**	**4**	**5**	**6**	**7**
3	**1**	**12**	**12**	**12**	**0**	**0**	**0**



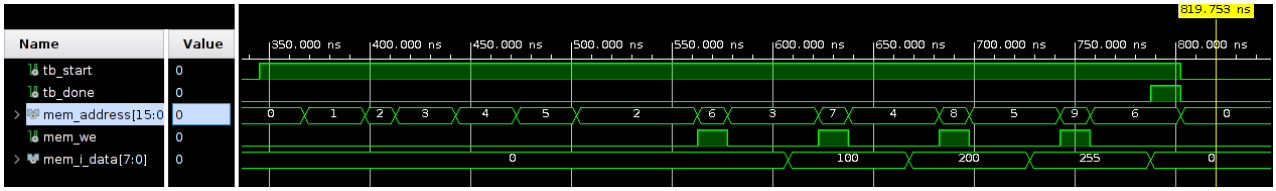
5. test con 1 colonna.

0	1	2	3	4	5	6	7	8	9	10	11
1	5	46	45	44	43	42	255	192	128	64	0



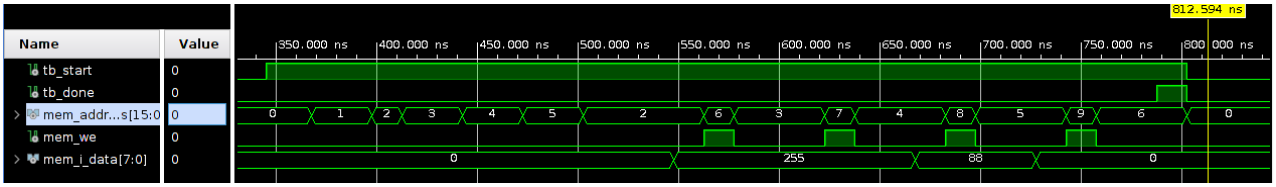
6. test 2x2 con valori incrementali ascendenti.

0	1	2	3	4	5	6	7	8	9
2	2	0	100	200	255	0	100	200	255



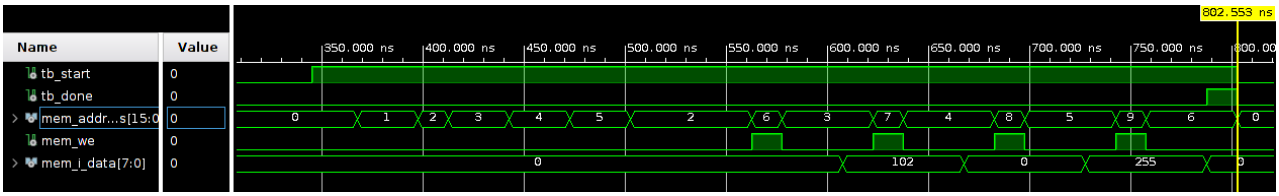
7. test 2x2 con valori incrementali discendenti.

0	1	2	3	4	5	6	7	8	9
2	2	131	86	34	12	255	255	88	0



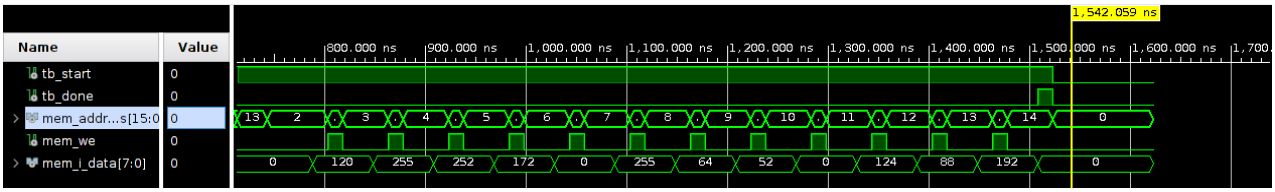
8. test 2x2 con valori casuali

0	1	2	3	4	5	6	7	8	9
2	2	21	72	21	205	0	102	0	255



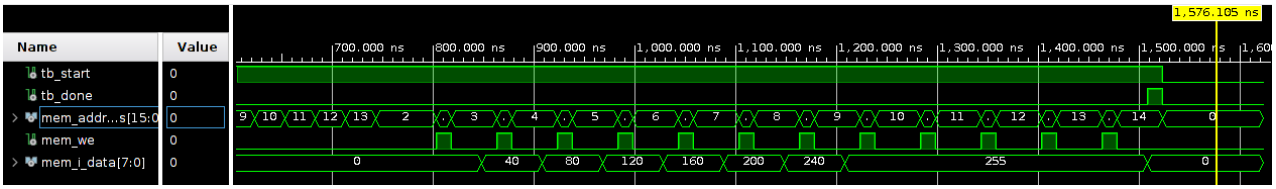
9. esempio 1 specifica 2020/21 fornita dal docente

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25
2	2	76	131	109	46	121	62	59	46	77	68	94	94	120	255	252	172	0	255	64	52	0	124	88	192



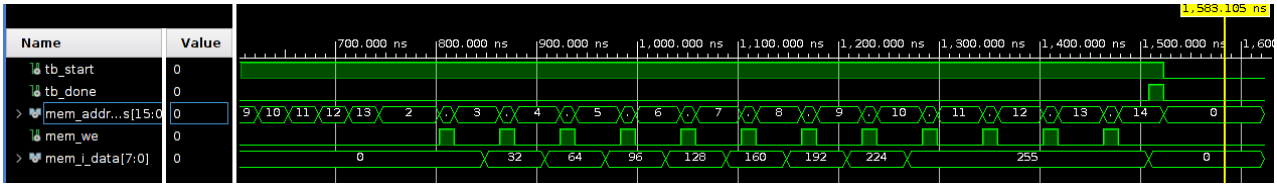
10. esempio 2 specifica 2020/21 fornita dal docente

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25
2	2	0	10	20	30	40	50	60	70	80	90	100	120	0	40	80	120	160	200	240	255	255	255	255	255



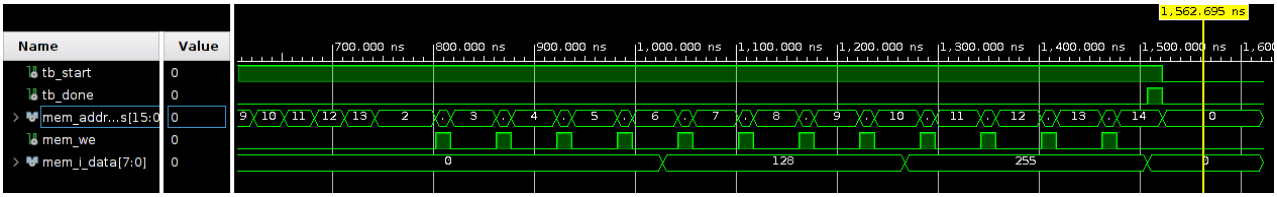
11. esempio 3 specifica 2020/21 fornita dal docente

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25
2	2	122	123	124	125	126	127	128	129	130	131	132	133	0	32	64	96	128	160	192	224	255	255	255	255



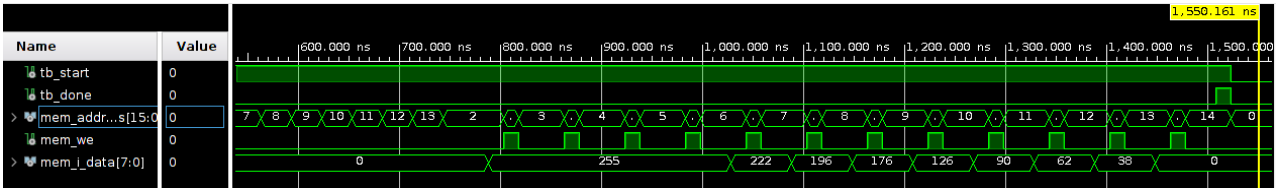
12. esempio 4 specifica 2020/21 fornita dal docente

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



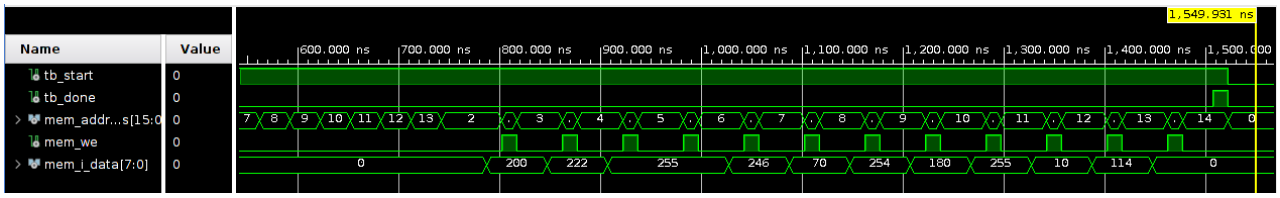
13. test 4x3 con valori incrementali discendenti

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25
2	2	196	182	174	173	112	99	89	64	46	32	30	1	255	255	255	255	222	196	176	126	90	62	38	0



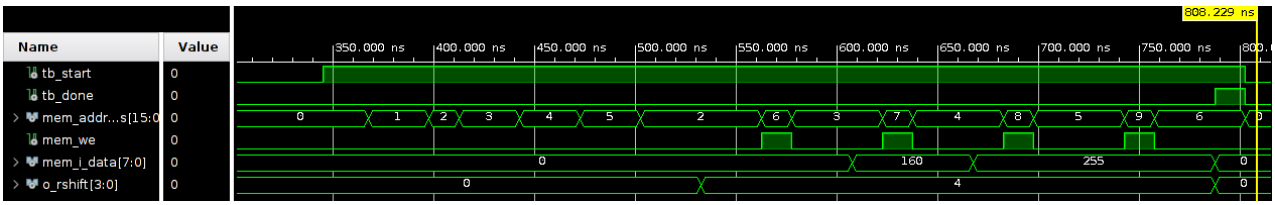
14. test 4x3 con valori casuali

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25
2	2	116	127	185	214	139	51	143	106	168	21	73	16	200	222	255	255	246	70	254	180	255	10	114	0



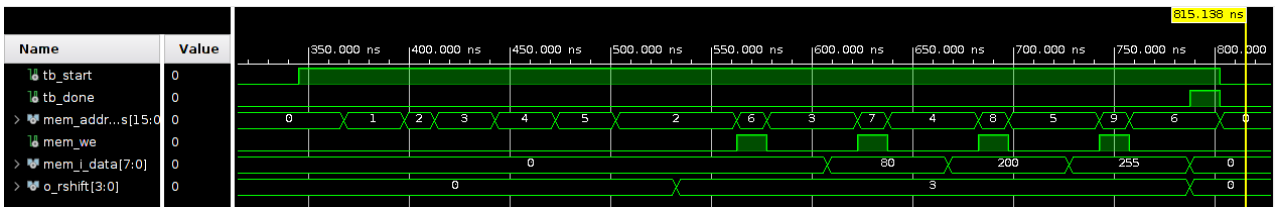
15. test 2x2 con shift_value = 3 (precedentemente non testato)

0	1	2	3	4	5	6	7	8	9
2	2	10	20	30	40	0	160	255	255



16. test 2x2 con shift_value = 4 (precedentemente non testato)

0	1	2	3	4	5	6	7	8	9
2	2	100	110	125	140	0	80	200	255



4. CONCLUSIONI

Il progetto di reti logiche mi ha permesso di interfacciarmi per la prima volta con la costruzione e realizzazione vera e propria di una macchina a stati a partire solo da una specifica.

Portando a scelte durante la fase di design progettuale, su ciò che meglio si addice ed è più funzionale, piuttosto che ciò che è più semplice e immediato, ma al contempo poco funzionale.

Ad esempio ho evitato in ogni modo l'utilizzo di moltiplicatori, optando invece per sommatori e contatori, così come la creazione "fai da te" dello shift_level.

Inoltre, ho cercato di trovare tutti i possibili punti deboli della macchina attraverso un notevole quantitativo di test che andassero a toccare tutte le situazioni limite, in cui poteva esserci un problema.

In conclusione, il progetto mi ha particolarmente stimolato nel ricercare quella che fosse una soluzione ovviamente corretta, ma al contempo senza usufruire di stati della macchina aggiuntivi o scorciatoie, rendendola quindi semplice ma al contempo sintetica.