RETI LOGICHE – PROVA FINALE

Leonardo Longhi
Cod. Persona **10548811**
Matricola **888626**

Mirko Li Veli
Cod. Persona **10562617**
Matricola **889700**

Corso di Reti Logiche
Anno accademico 2020-2021
Professore: Fabio Salice

INDICE

1. INTRODUZIONE

1.1 Scopo del progetto

La specifica del progetto introduceva il concetto di equalizzazione dell'istogramma di un'immagine, metodo utilizzato per ricalibrare e aumentare il contrasto di un'immagine quando l'intervallo dei valori di intensità sono molto vicini, effettuandone una distribuzione su tutto l'intervallo di intensità.

Nella versione da implementare l'algoritmo di equalizzazione è stato applicato solo ad immagini in scala di grigi a 256 livelli.

Lo scopo del progetto era, dunque, quello di implementare un componente hardware, descritto in VHDL, che, prendendo in ingresso la codifica di un'immagine in modo sequenziale e riga per riga da una memoria, ne trasformi opportunamente ogni suo pixel e lo restituisca come output.

1.2 Approfondimento sulla codifica

L'immagine da rielaborare ha una dimensione definita di 2 byte, memorizzati a partire dall'indirizzo 0 contenente il numero delle righe; l'indirizzo 1, invece, contiene il numero di colonne. A partire dall'indirizzo 2 si ha il primo pixel appartenente alla prima riga e prima colonna dell'immagine della dimensione di 1 byte.

La dimensione massima dell'immagine è 128x128 pixel.

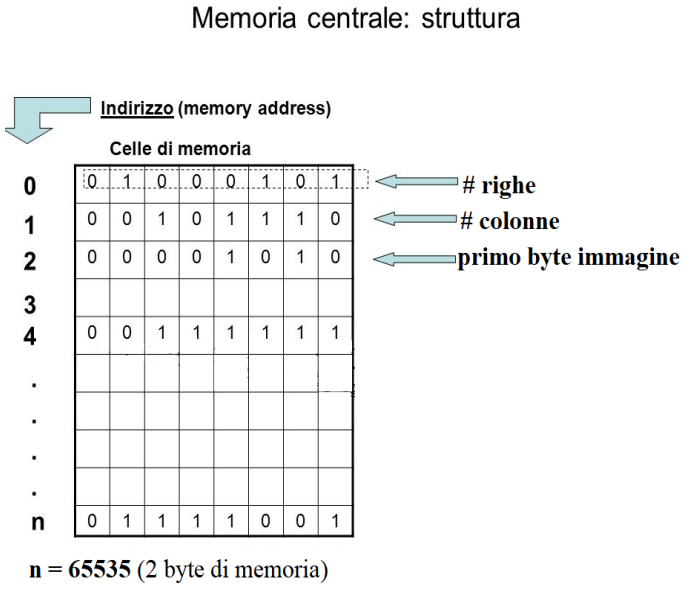

Fig. 1 - Struttura della memoria

Ogni pixel verrà rielaborato secondo le funzioni fornite della specifica e trasmesso come output al primo indirizzo di memoria disponibile, ricavabile dalla funzione:

INDIRIZZO_SCRITTURA = 2 + N-RIGHE * N-COLONNE + j

Con $0 \leq j \leq$ N-RIGHE * N-COLONNE - 1

dove 'j' indica il j-esimo pixel dell'immagine.

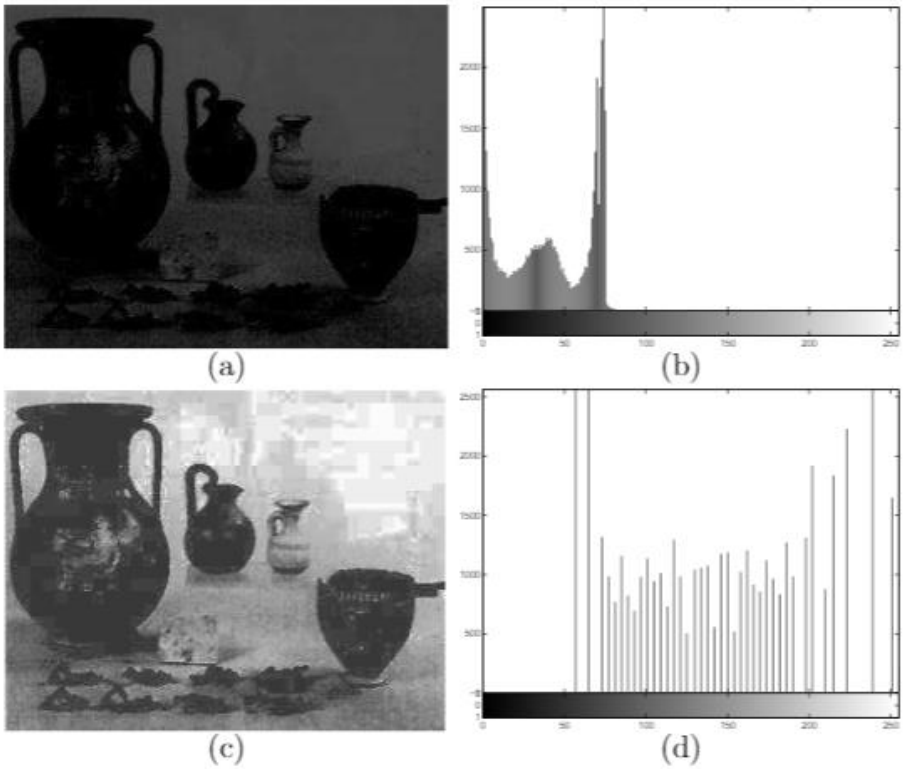

Figura 2 - In alto un'immagine non equalizzata con il suo istogramma, in basso la stessa immagine equalizzata e il suo istogramma

Il tutto partirà quando il segnale START in ingresso verrà portato a 1. Una volta finita la codifica e scritto il dato in memoria, verrà portato a 1 un segnale DONE in uscita. Questo deve rimanere alto finché il segnale START non viene riportato a 0. A questo punto si può riportare a 0 anche il segnale DONE. Quando il segnale START verrà riportato a 1, tutto il processo ripartirà.

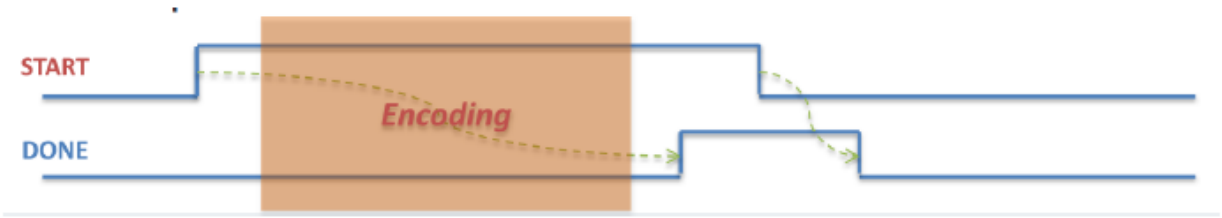

Fig. 3 - La fase di encoding parte quando si alza il segnale di START e termina alzando il segnale di DONE, che resta alto finché non viene riabbassato il segnale di START

1.3 Interfaccia del componente

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(7 downto 0)
);
end project_reti_logiche; end project_reti_logiche;
```

In particolare:

- i_clk è il segnale di CLOCK in ingresso generato dal Test Bench;
- 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;
- o_en è il segnale di ENABLE da mandare alla memoria per poter comunicare (sia in lettura che in scrittura);
- o_we è il segnale di WRITE ENABLE da 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. DESIGN

2.1 Scelte progettuali

La scelta progettuale è stata quella di creare più processi, ognuno atto a svolgere un compito differente, in modo da atomizzare il più possibile tutte le operazioni.
Affinché ciò fosse possibile, è stato implementato il component *datapath* all'interno dell'unità, di seguito definito.

```
COMPONENT datapath IS
    PORT (
        i_clk : IN STD_LOGIC;
        i_rst : IN STD_LOGIC;
        i_data : IN STD_LOGIC_VECTOR(7 DOWNTO 0);
        o_data : OUT STD_LOGIC_VECTOR(7 DOWNTO 0);
        o_address : OUT STD_LOGIC_VECTOR(15 DOWNTO 0);
        o_next_state : OUT STD_LOGIC;
        compute_size_dim_1 : IN STD_LOGIC;
        compute_size_dim_2 : IN STD_LOGIC;
        compute_size : IN STD_LOGIC;
        compute_max_min : IN STD_LOGIC;
        compute_shift_level : IN STD_LOGIC;
        compute_filter_r : IN STD_LOGIC;
        compute_filter_w : IN STD_LOGIC
    );
END COMPONENT;
```

In particolare:

- *i_clk :* stesso segnale descritto nell'entity principale e generato dal Test Bench;
- *i_rst :* stesso segnale descritto nell'entity principale e generato dal Test Bench;
- *i_data :* è il segnale (vettore di 8 bit) che arriva dalla memoria in seguito ad una richiesta di lettura;
- *o_data :* è il segnale (vettore di 8 bit) di uscita dal componente che trasporta i dati elaborati verso la memoria.
- *o_address :* è il segnale *(*vettore di 16 bit) di uscita dal componente contenente:
  -in fase di lettura, l'indirizzo futuro da leggere
  -in fase di scrittura, l'indirizzo in cui scrivere o_data
- *o_next_state :* segnale di uscita che permette il passaggio allo stato prossimo
- *compute_size_dim_1 :* segnale che indica il raggiungimento dello stato *sd_1* (capitolo 2.2.2)
- *compute_size_dim_2 :* segnale che indica il raggiungimento dello stato *sd_2* (capitolo 2.2.3)
- *compute_size :* segnale che indica il raggiungimento dello stato *sd* (capitolo 2.2.4)
- *compute_max_min :* segnale che indica il raggiungimento dello stato *seek* (capitolo 2.2.5)
- *compute_shift_level :* segnale che indica il raggiungimento dello stato *shift_state* (capitolo 2.2.6)
- *compute_filter_r :* segnale che indica il raggiungimento dello stato *filter_r* (capitolo 2.2.7)
- *compute_filter_w :* segnale che indica il raggiungimento dello stato *filter_w* (capitolo 2.2.8)

Il component ha al suo interno 7 processi che verranno descritti più dettagliatamente nei prossimi capitoli. Tutte le altre istruzioni vengono eseguite in maniera parallela.
L'entity principale, invece, presenta tre process:

- *assign_new_state*: è il processo che aggiorna il *curr_state*. In caso di segnale di *i_rst* alto, *curr_state* torna ad essere *reset*; mentre nel caso di segnale alto di *i_clk*, il *curr_state* prende il valore del *next_state*;
- *compute_next_state*: è il processo che in seguito a una variazione del *curr_state*, di *i_start* o di *next_state_ready*, regola il passaggio di stati della FSM;
- *assign_new_signal*: è il processo che assegna i valori dei segnali in corrispondenza della variazione del *curr_state*.

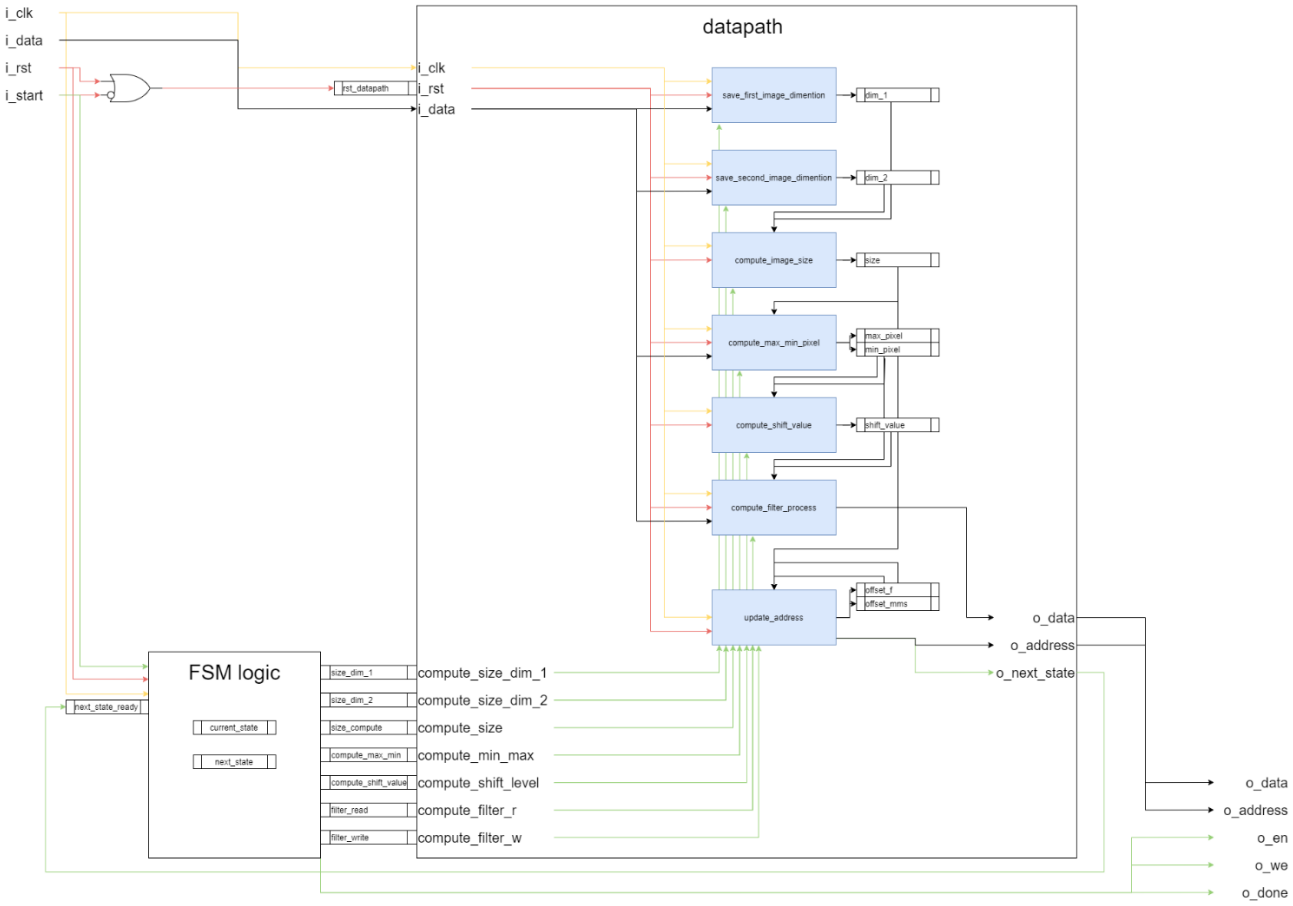

Fig. 4 - Struttura project_reti_logiche

Per favorire la comprensione di quanto appena descritto, è stata riportata qui sopra (figura 4) la struttura del progetto.
Viene inoltre descritta una breve legenda:

- Frecce rosse rappresentano segnali di reset;
- Frecce gialle rappresentano il segnale di clock;
- Frecce verdi rappresentano segnali utilizzati per specificare all'interno di quale processo si trova la macchina e se tale processo può essere terminato;
- Blocchi azzurri rappresentano i processi;
- Ogni processo ha un segnale (verde) come enable;
- Le frecce nere rappresentano segnali che trasportano dati relativi all'immagine.

2.2 Descrizione degli stati

La macchina costruita è composta da 9 stati che, di seguito, verranno descritti brevemente.

Il funzionamento della macchina è descritto in modo tale che venga utilizzato il fronte di salita del clock per aggiornare gli input ricevuti e che il fronte di discesa sia utilizzato per l'elaborazione degli input raccolti e per aggiornare gli output.

Ad ogni stato corrisponde uno specifico processo all'interno del componente *datapath* (fatta eccezione per gli stati *reset*, filter_w e *end_state*). Inoltre, in tutti gli stati viene eseguito il processo *update_address* che genera l'indirizzo di memoria utilizzato nello stato successivo.

2.2.1 RESET

Lo stato di *reset* è lo stato iniziale della macchina in cui tutti i valori dei segnali, registri, variabili e memorie vengono inizializzati.

Si è ipotizzato che un segnale di reset (*i_rst* alto) possa essere ricevuto in qualsiasi momento, in quanto si è ritenuta possibile l'eventualità di voler annullare l'operazione di equalizzazione di un'immagine. Pertanto, si è deciso di rendere lo stato di reset raggiungibile da tutti gli altri stati della macchina.

Lo stato di *reset* viene seguito dallo stato *sd1*.

2.2.2 SD1

Lo stato *sd1* è preceduto dallo stato di *reset* e viene raggiunto a seguito del cambiamento dell'input *i_start* da basso ad alto.

A questo stato corrisponde il processo *save_first_image_dimention* in cui viene letto il numero di righe dell'immagine dall'indirizzo 0000000000000000 della memoria. Tale valore viene successivamente inserito nel registro *dim_1*.

Al termine di tale operazione, la macchina passerà allo stato *sd2*.

Un segnale di reset (*i_rst* alto) può riportare la macchina allo stato di *reset*.

2.2.3 SD2

Nello stato *sd2* viene eseguito il processo *save_second_image_dimention.* Questo processo ha lo scopo di far leggere al datapath il numero di colonne dell'immagine dall'indirizzo 0000000000000001 della memoria di input. Tale valore viene inserito nel registro *dim_2*.

Al termine dell'operazione, la macchina passerà allo stato *sd*.

Un segnale di reset (*i_rst* alto) può riportare la macchina allo stato di *reset*.

2.2.4 SD

Allo stato *sd* corrisponde il processo *compute_image_size*. Questo processo prende come input i valori inseriti nei registri *dim_1* e *dim_2* trovati negli stati precedenti e li utilizza per calcolare la dimensione dell'immagine. Il risultato di questa operazione viene inserita nel registro *size*.

Al termine del processo, la macchina passerà allo stato *seek*.

Un segnale di reset (*i_rst* alto) può riportare la macchina allo stato di *reset*.

2.2.5 SEEK

Lo stato *seek* viene raggiunto al termine del calcolo della dimensione dell'immagine.

In questo stato viene eseguito il processo *compute_max_min_pixel* che leggerà tutta la memoria tra l'indirizzo 2 e l'indirizzo 2+*size* (cioè l'indirizzo in cui è contenuto il primo pixel da equalizzare e l'indirizzo in cui il primo pixel dovrà essere scritto nella sua forma equalizzata). Tra questi indirizzi verranno cercati il massimo ed il minimo valore dei pixel dell'immagine di input.

Partendo da una condizione di inizializzazione che vede min_pixel = "11111111" e max_pixel = "00000000", ad ogni valore letto vengono aggiornati i contenuti dei registri se necessario.

Una volta terminata la lettura nei registri *max_pixel* e *min_pixel* si troveranno rispettivamente il valore massimo e il valore minimo dei pixel e *next_state_ready* verrà portato a livello logico alto segnalando la fine della ricerca permettendo così alla macchina di passare allo stato *shift_state*.

Un segnale di reset (*i_rst* alto) può riportare la macchina allo stato di *reset*.

2.2.6 SHIFT_STATE

Allo stato *shift_state* è associato il processo *compute_shift_value* che ha il compito di calcolare il valore di *shift_level* prendendo in input i valori di *max_pixel* e *min_pixel* calcolati nello stato precedente.

Lo *shift_level* viene calcolato a partire dalla formula fornita dalla specifica di progetto:

$$DELTA\_VALUE = MAX\_PIXEL\_VALUE - MIN\_PIXEL\_VALUE$$

$$SHIFT\_LEVEL = (8 - FLOOR(LOG2(DELTA\_VALUE + 1)))$$

Una volta calcolato il valore dello *shift_level* viene inserito nel registro *shift_value*. Al termine del processo, la macchina viene portata nello stato *filter_r*.

Un segnale di reset (*i_rst* alto) può riportare la macchina allo stato di *reset*.

2.2.7 FILTER_R (READ)

Lo stato *filter_r* è associato al processo *compute_filter_process*. In particolare, nello stato *filter_r* viene letto il pixel da equalizzare dalla memoria, viene calcolato il nuovo valore del pixel e portato sull'uscita *o_data* codificato su 16 bit.
Il nuovo valore del pixel viene fornito dalla specifica di progetto:

$$TEMP\_PIXEL = (CURRENT\_PIXEL\_VALUE - MIN\_PIXEL\_VALUE) << SHIFT\_LEVEL$$

$$NEW\_PIXEL\_VALUE = MIN\ (255, TEMP\_PIXEL)$$

Sul segnale di uscita *o_address* viene invece riportato l'indirizzo in cui scrivere il nuovo valore del pixel appena calcolato (indirizzo corrente + *size*).

Prima di eseguire questa operazione viene effettuato un controllo sull'indirizzo in cui scrivere:

- Se l'immagine non è stata interamente equalizzata (*o_address* <= 2+2**size*), allora viene abilitato il segnale *o_we* e la macchina viene portata allo stato *filter_w*.
- Se l'immagine è stata interamente equalizzata, allora viene settato a '1' il segnale *next_state_ready* e la macchina viene portata allo stato *end_state*.

Un segnale di reset (*i_rst* alto) può riportare la macchina allo stato di *reset*.

2.2.8 FILTER_W (WRITE)

Lo stato *filter_w* non è associato alcun processo. Si usano i cicli di clock dedicati a questo stato per attendere la corretta scrittura in memoria dei dati calcolati nello stato precedente.
Al termine della scrittura viene aggiornato l'indirizzo *o_address* inserendoci l'indirizzo da cui leggere il prossimo pixel da equalizzare. Segue poi, lo stato *filter_r* raggiunto dopo aver riportato a '0' il segnale *o_we*.
Un segnale di reset (*i_rst* alto) può riportare la macchina allo stato di *reset*.

2.2.9 END_STATE

End_state rappresenta lo stato finale della macchina. In questo stato, viene portato ad '1' il segnale associato all'uscita *o_done*. Come da specifica di progetto, si attende che il segnale *i_start* venga riportato a '0'.
o_done = '1' e *i_start* = '0' riporteranno la macchina allo stato di *reset*.

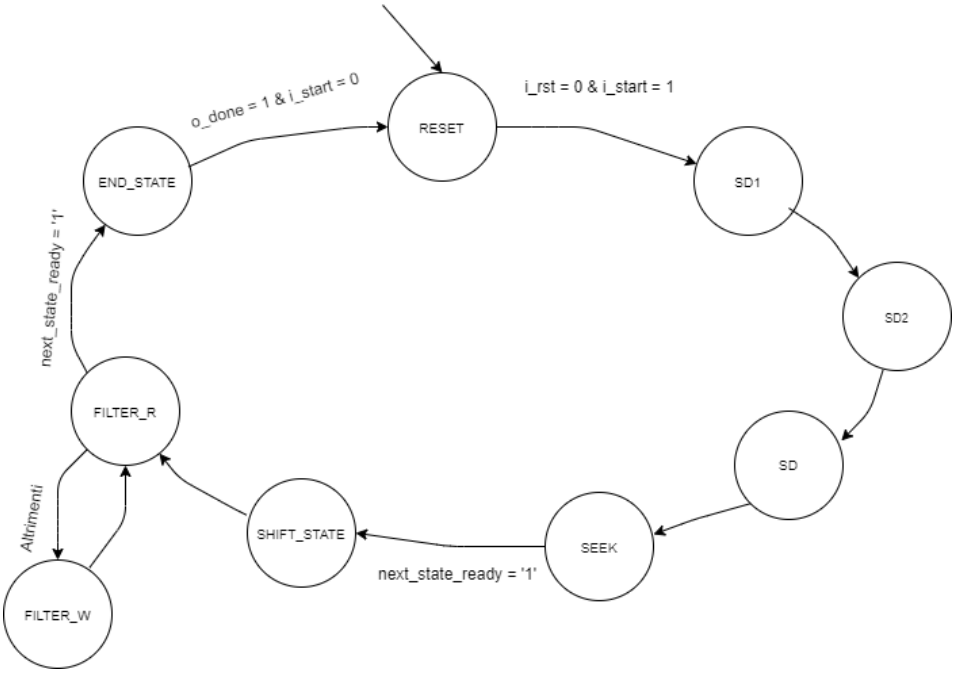

Fig. 5 - Macchina a stati riassuntiva[1]

[1] In figura 4, per facilitarne la lettura, non vengono riportate le frecce di transizione che collegano tutti gli stati allo stato di reset.

3. Casi di test

Per verificare il corretto funzionamento del componente *project_reti_logiche* è stato utilizzato il Test Bench fornito dall'insegnante. Esso consisteva in una memoria in cui era contenuta la codifica di un'immagine 2x2. Ipotizzando diversi casi limite, si è ritenuto necessario testare diversi comportamenti della macchina. Tutti i test eseguiti e i loro risultati verranno dettagliatamente presentati nei prossimi paragrafi.

3.1 Test_bench 2x2

Questo è il primo test in cui tutti i componenti implementati lavorano collaborando.

Il test fornitoci presenta la memoria inizializzata nel seguente modo:

Indirizzo memoria	0	1	2	3	4	5
Dec	2	2	46	131	62	89
Hex	2	2	2E	83	3E	59

Qui di seguito viene illustrata la memoria attesa a fine processo:

Indirizzo memoria	0	1	2	3	4	5	6	7	8	9
Dec	2	2	46	131	62	89	0	255	64	172
Hex	2	2	2E	83	3E	59	0	FF	40	AC

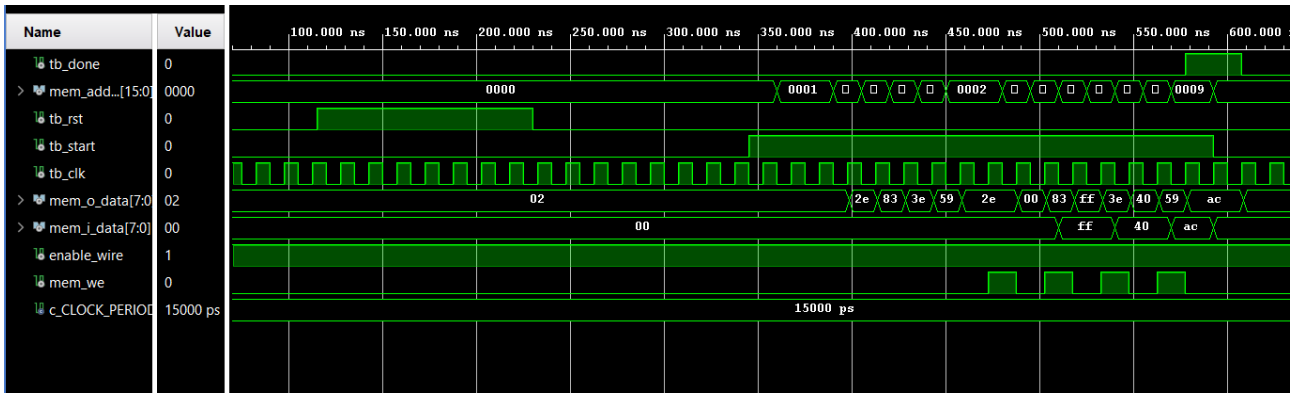

Fig. 6 - Risultato test Behavioral Simulation del 1

3.2 Test_bench 2x2 con reset

Viene ripetuto il test precedente e durante l'equalizzazione della foto viene inviato un segnale di reset. Per semplicità viene risottoposta la stessa immagine da equalizzare.

Il sistema risponde correttamente, ripartendo da capo e riiniziando subito la codifica (*i_start*, infatti, è rimasto =1).

Nell'immagine è evidenziato il momento in cui si alza *i_rst*.

Fig. 7 - Risultato Behavioral Simulation del test 2

3.3 Test con memoria vuota

Questo test vuole simulare il comportamento della macchina quando le viene richiesto di lavorare avendo come ingresso una memoria vuota. La memoria finale attesa è anch'essa una memoria vuota.

In foto viene mostrato il ragionevole comportamento del componente:

Appena lette le dimensioni dell'immagine, viene alzata l'uscita *o_done* collegato al segnale *tb_done*.

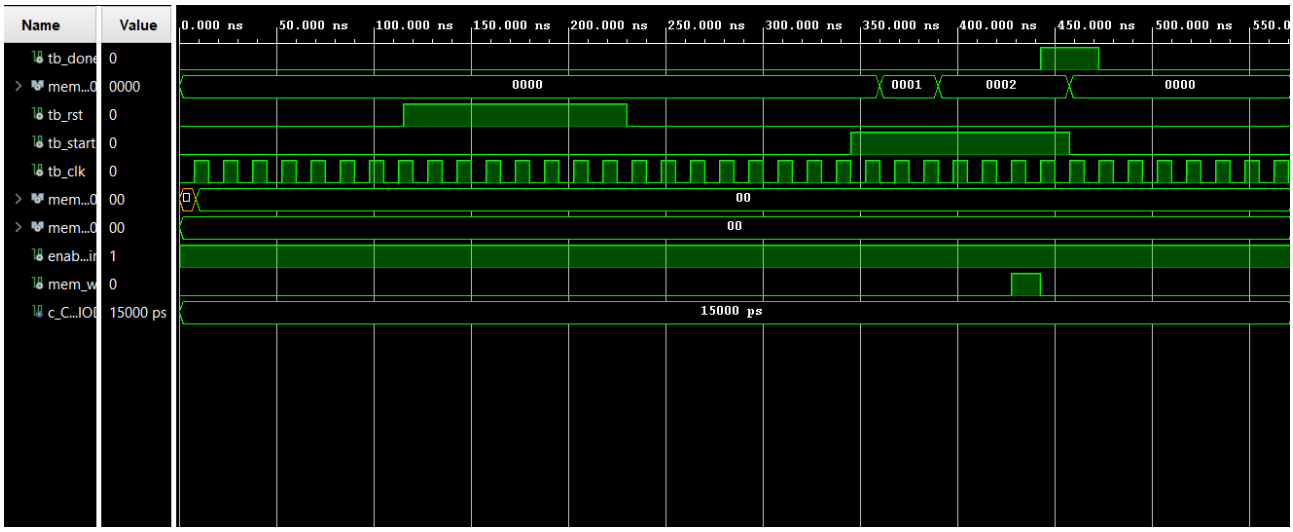

Fig. 8 - Risultato Behavioral Simulation del test 3

3.4 Test con tutti i pixel uguali

Questo test simula il comportamento della macchina nel caso in cui l'immagine sia composta da pixel tutti uguali.

L'output atteso è composto da valori uguali a '0'.

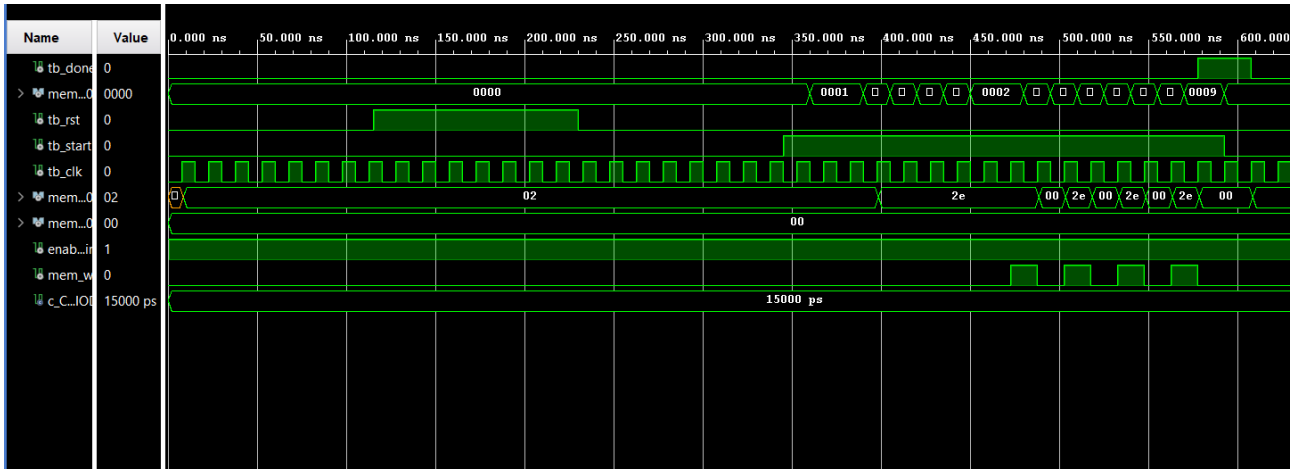

Fig. 9 - Risultato Behavioral Simulation del test 4

3.5 Test con immagine rettangolare

Il test vuole verificare il corretto funzionamento anche nel caso in cui il rapporto righe-colonne non sia 1:1. In questo particolare caso è stato testato il funzionamento dando in ingresso un'immagine 2x4.

La memoria viene inizializzata nel seguente modo:

Indirizzo memoria	0	1	2	3	4	5	6	7	8	9
Dec	2	4	141	241	56	250	227	96	80	124
Hex	2	4	8D	F1	38	FA	E3	60	50	7C

La memoria attesa dopo il processo di equalizzazione è:

Indirizzo memoria	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
Dec	2	4	141	241	56	250	227	96	80	124	170	255	0	255	255	80	48	136
Hex	2	4	8D	F1	38	FA	E3	60	50	7C	AA	FF	0	FF	FF	50	30	88

Si noti in foto, il corretto output del processo di equalizzazione.

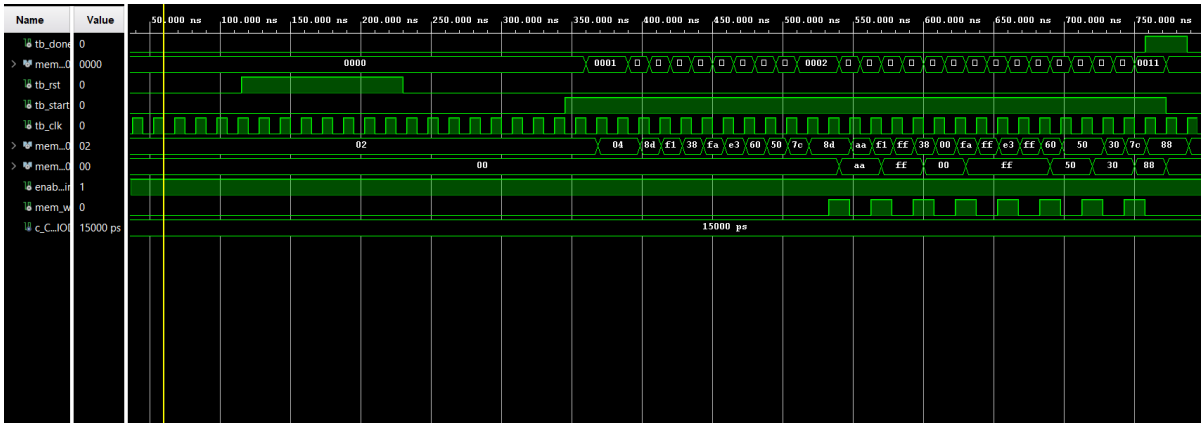

Fig. 10 - Risultato Behavioral Simulation del test 5

3.6 Test con più immagini

Il test 6 è atto a verificare il corretto comportamento del componente quando viene richiesto di equalizzare più immagini sequenzialmente.

In questo caso sono state sottoposte al componente tre diverse immagini.

Per semplicità di scrittura dei test, ogni immagine è stata inserita in una memoria diversa.

Le 3 memorie sono state poi poste come input in modo sequenziale ogni volta che veniva alzato il segnale di uscita *o_done*.

Di seguito vengono riportate le 3 memorie inizializzate e le rispettive memorie attese a fine processo.

RAM 1:

Indirizzo memoria	0	1	2	3	4	5
Dec	2	2	141	241	56	250
Hex	2	2	8D	F1	38	FA

Indirizzo memoria	0	1	2	3	4	5	6	7	8	9
Dec	2	2	141	241	56	250	170	255	0	255
Hex	2	2	8D	F1	38	FA	AA	FF	0	FF

RAM 2:

Indirizzo memoria	0	1	2	3	4	5	6	7	8	9
Dec	2	4	141	241	56	250	227	96	80	124
Hex	2	4	8D	F1	38	FA	E3	60	50	7C

Indirizzo memoria	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
Dec	2	4	141	241	56	250	227	96	80	124	170	255	0	255	255	80	48	136
Hex	2	4	8D	F1	38	FA	E3	60	50	7C	AA	FF	0	FF	FF	50	30	88

RAM 3:

Indirizzo memoria	0	1	2	3	4	5
Dec	2	2	48	241	17	54
Hex	2	2	30	F1	11	36

Indirizzo memoria	0	1	2	3	4	5	6	7	8	9
Dec	2	2	48	241	17	54	62	255	0	74
Hex	2	2	30	F1	11	36	3E	FF	0	4A

Qui sotto viene invece riportato il risultato del test 9.

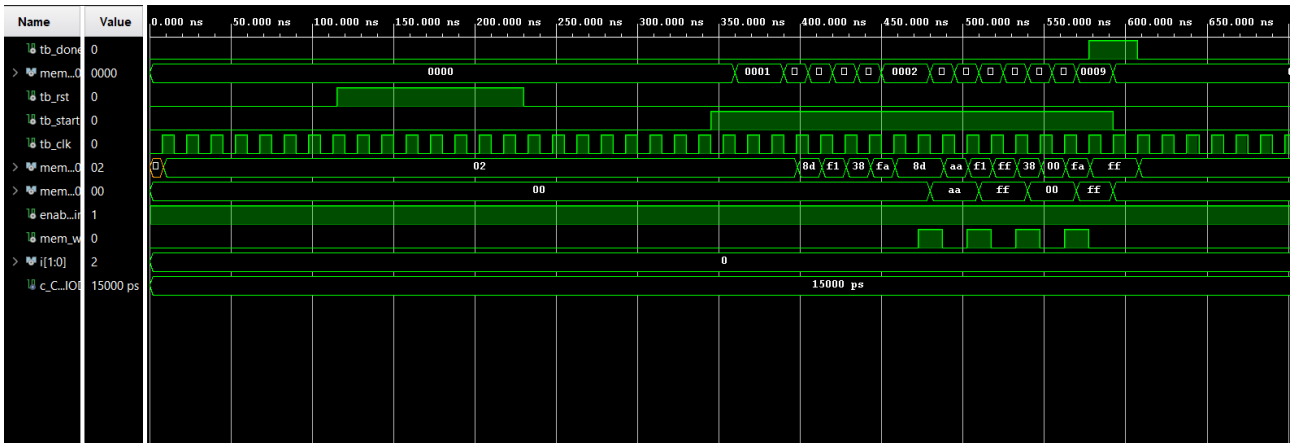

Fig. 11 – Risultato Behavioral Simulation del test 6 pt.1

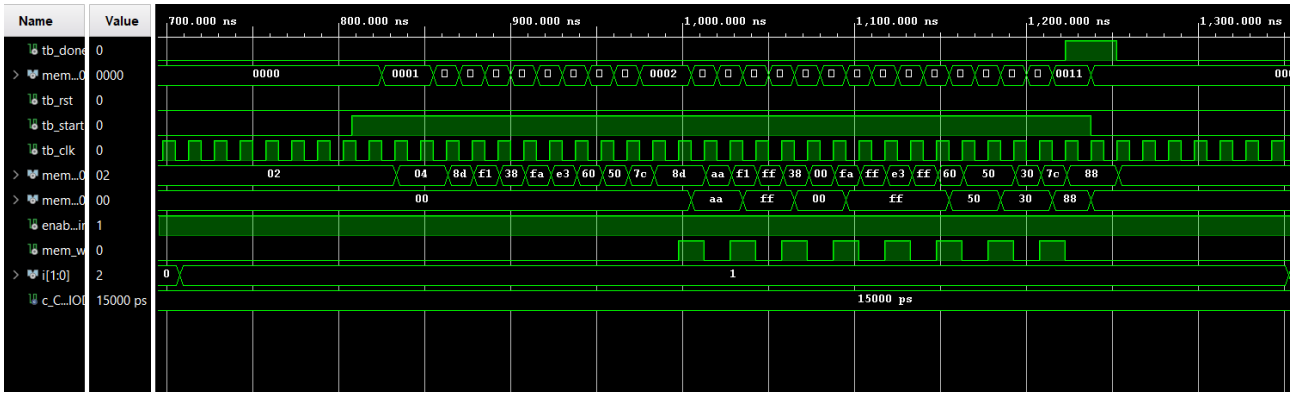

Fig. 12 – Risultato Behavioral Simulation del test 6 pt.2

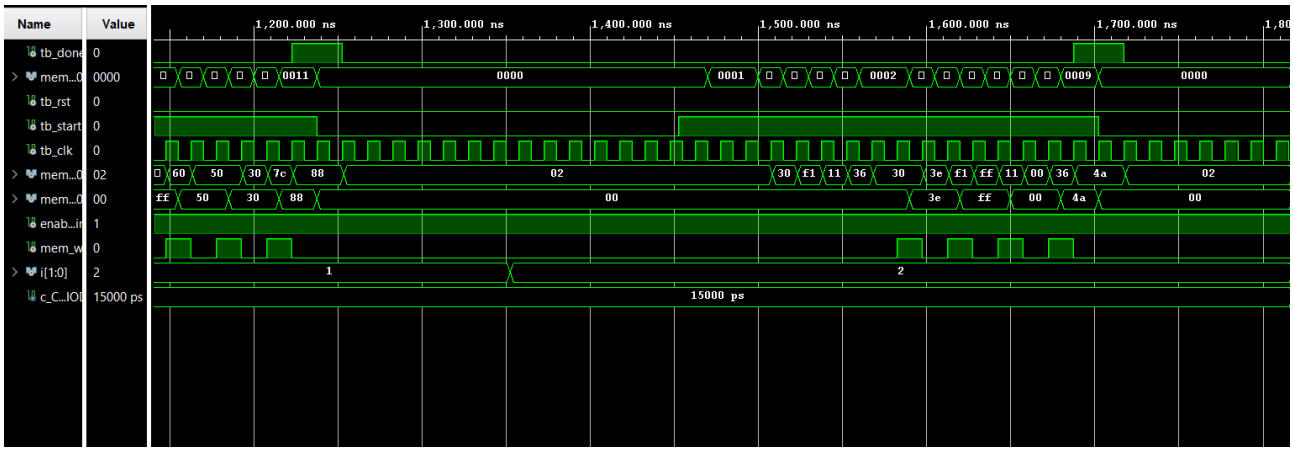

Fig. 13 – Risultato Behavioral Simulation del test 6 pt.3

Si noti in figura l'inizio di equalizzazione delle tre immagini caratterizzate dai tre segnali alti di start e la fine dei tre processi caratterizzate dai tre segnali di *o_done* connessi al segnale *tb_done*.

4. RISULTATI DELLA SINTESI

Vengono adesso riportati report utili legati alla sintesi del componente. In particolare, un importante parametro nella valutazione è il *Worst Negative Slack*, ovvero il parametro che quantifica rispetto al tempo totale quale sia il tempo impiegato dal path peggiore. Il Design Timing Summary è stato calcolato in seguito all'aggiunta di un constraint sul clock di questo tipo:
create_clock -name clk -period 100.000 -waveform {0.000 50.000} [**get_ports** i_clk]

Design Timing Summary

Setup		Hold		Pulse Width	
Worst Negative Slack (WNS):	47,247 ns	Worst Hold Slack (WHS):	0,182 ns	Worst Pulse Width Slack (WPWS):	49,600 ns
Total Negative Slack (TNS):	0,000 ns	Total Hold Slack (THS):	0,000 ns	Total Pulse Width Negative Slack (TPWS):	0,000 ns
Number of Failing Endpoints:	0	Number of Failing Endpoints:	0	Number of Failing Endpoints:	0
Total Number of Endpoints:	190	Total Number of Endpoints:	190	Total Number of Endpoints:	117

All user specified timing constraints are met.

Fig. 14 - Design Timing Summary

Utilization **Post-Synthesis** | Post-Implementation

Graph | **Table**

Resource	Estimation	Available	Utilization...
LUT	246	41000	0.60
FF	116	82000	0.14
IO	38	300	12.67
BUFG	1	32	3.13

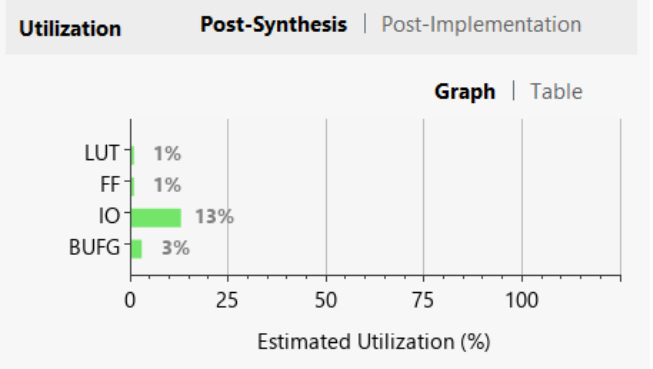

Fig. 15 - Estimated utilization results

Vengono adesso riportati i risultati della simulazione post sintesi dei test introdotti in precedenza.

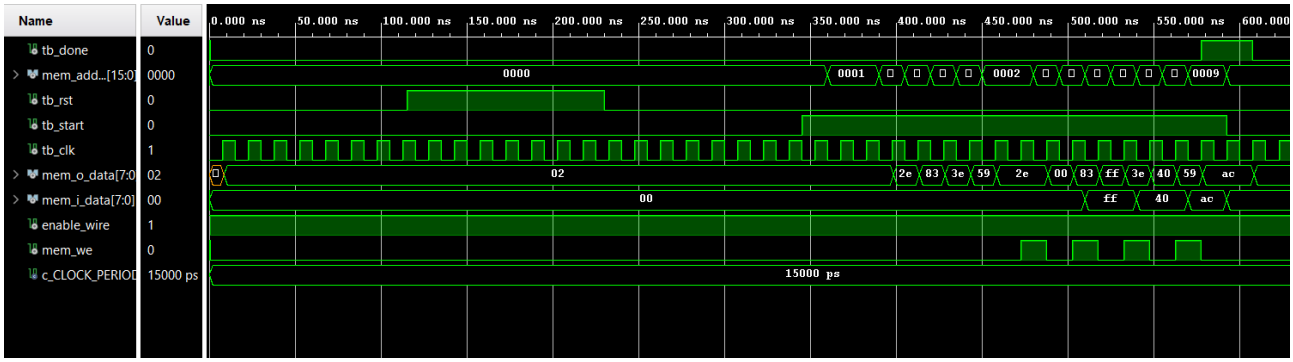

Fig. 16 - Risultato della Post-Synthesis Functional Simulation del test 1

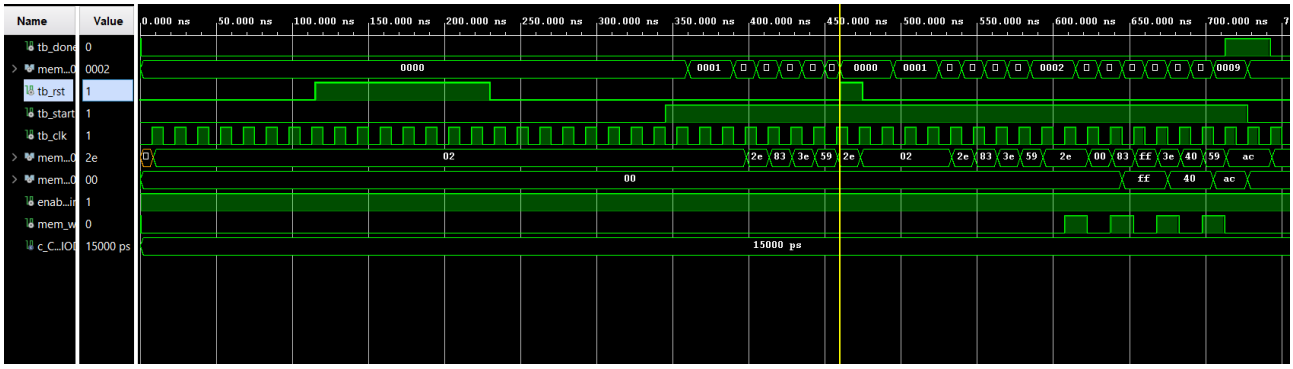

Fig. 17 - Risultato della Post-Synthesis Functional Simulation del test 2

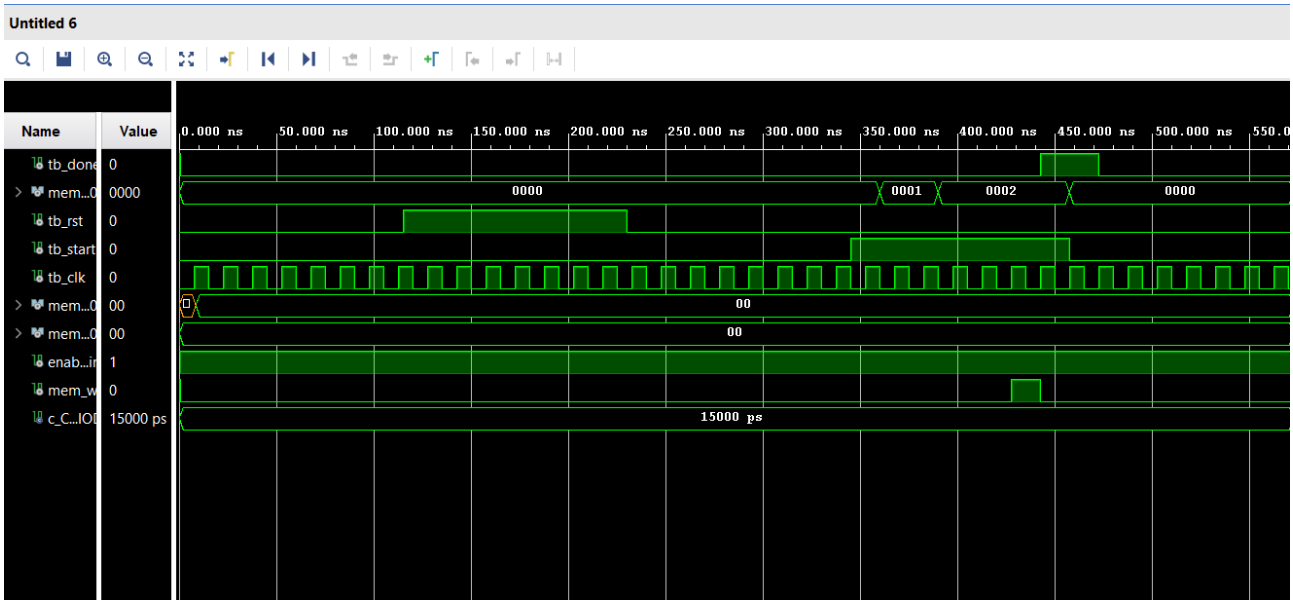

Fig. 18 - Risultato della Post-Synthesis Functional Simulation del test 3

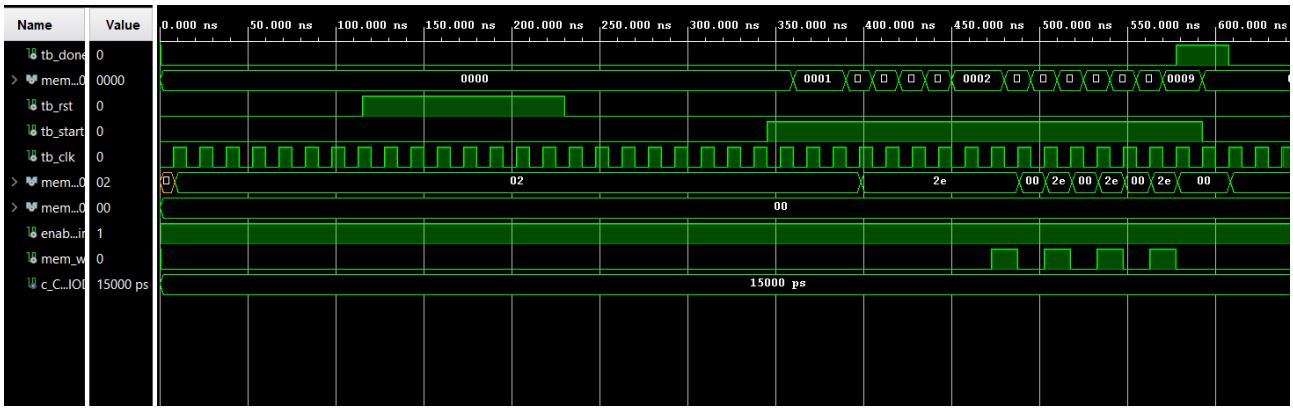

Fig. 19 - Risultato della Post-Synthesis Functional Simulation del test 4

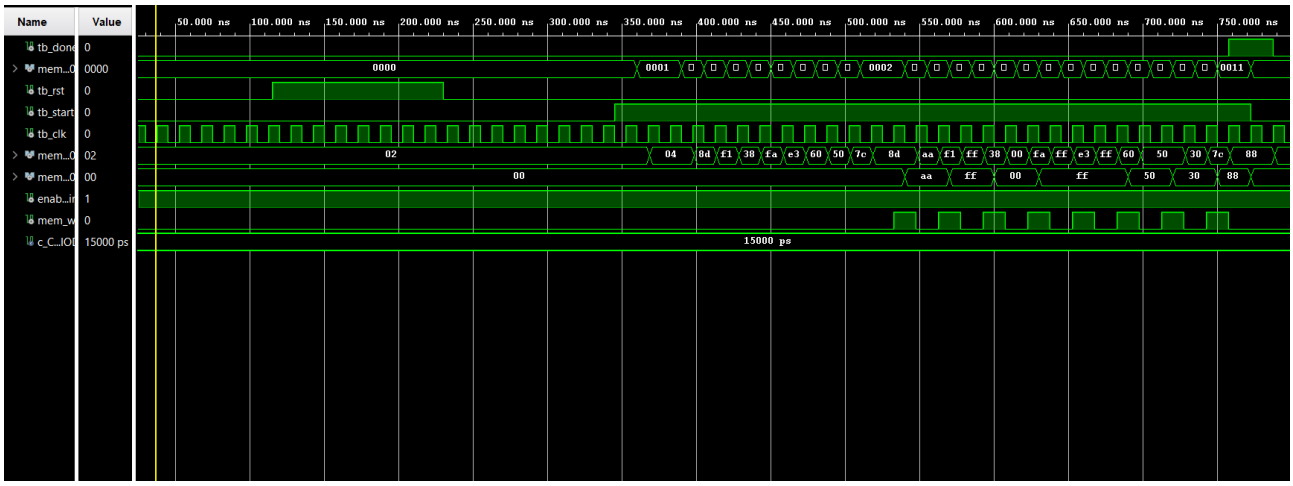

Fig. 20 - Risultato della Post-Synthesis Functional Simulation del test 5

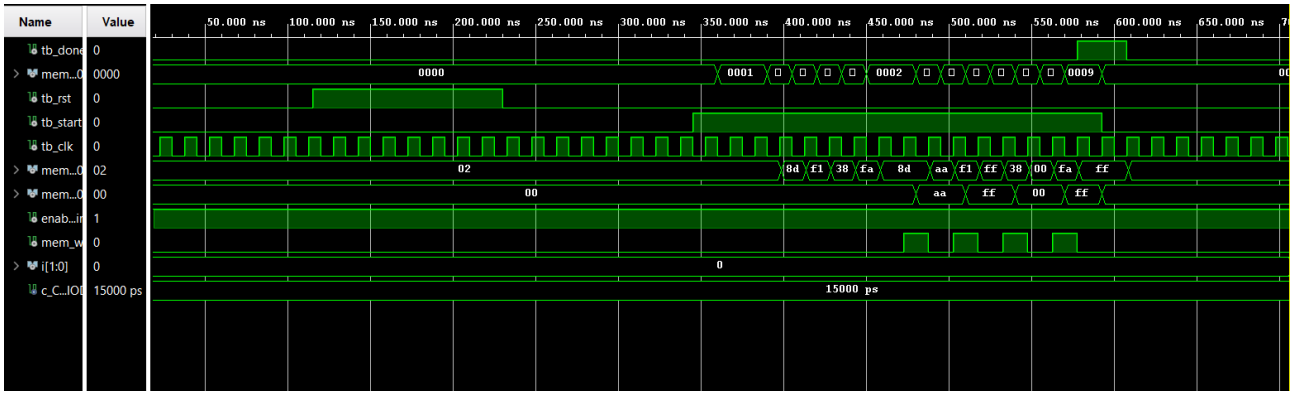

Fig. 21 - Risultato della Post-Synthesis Functional Simulation del test 6 pt.1

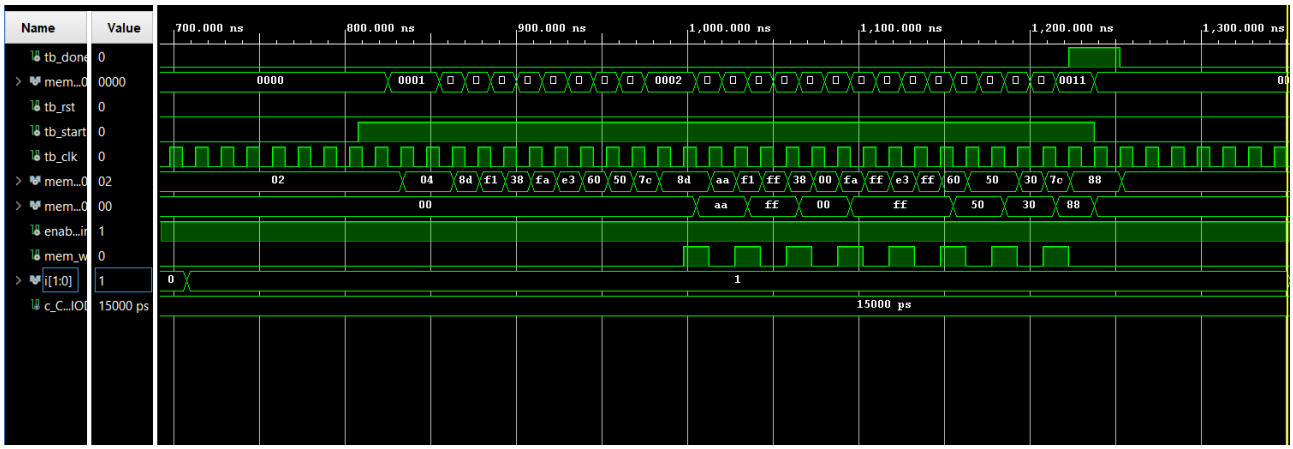

Fig. 22 - Risultato della Post-Syntheysis Functional Simulation del test 6 pt.2

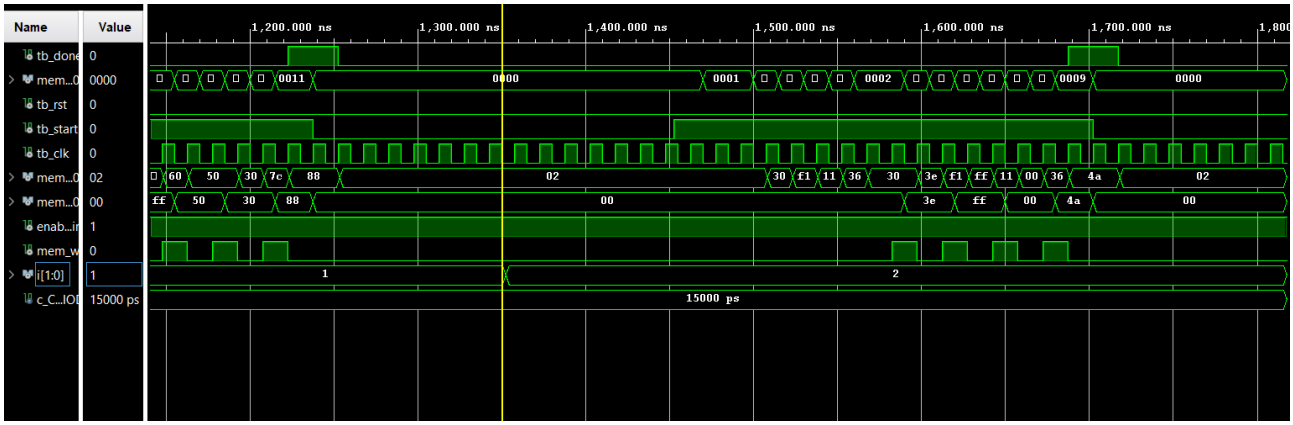

Fig. 23 - Risultato della Post-Syntheysis Functional Simulation del test 6 pt.3

5. CONCLUSIONI

L'idea alla base della realizzazione del progetto è sempre stata quella di atomizzare il più possibile tutti i processi in modo da renderne più facile la scrittura e la lettura.
Dopo la fase di progettazione si è passati alla fase di implementazione in cui si è tradotto in codice VHDL quanto pensato in precedenza.
L'idea inziale ci ha, quindi, indotto alla definizione di piccoli processi, ognuno associato ad uno stato, con il proprio semplice compito. Ciò ci ha dato la possibilità di scrivere il codice con maggiore semplicità, così da rilevare con accuratezza la presenza di eventuali errori.
Durante la fase di debugging si è pensato di sfruttare entrambi i fronti del clock per eseguire operazioni differenti (quello di salita per l'aggiornamento dello stato e dei segnali di ingresso; quello di discesa per l'elaborazione dei dati e l'aggiornamento delle uscite). Questo ha portato a ottimizzare il codice scritto, permettendo al componente di comportarsi correttamente rispetto ai differenti casi di test.
Infatti, il componente ha restituito i giusti risultati ad ogni test a cui è stato sottoposto, sia in fase Behavioral Simulation che in Post-Syntheysis Functional Simulation, il tutto rispettando i vincoli imposti sul clock dalla specifica.