Prova Finale
(Progetto di Reti Logiche)

Prof. Fabio Salice - Anno 2020/2021

Annalaura Massa (Codice Persona 10616557 - Matricola 886706)
Eduardo Manfrellotti (Codice Persona 10635323 - Matricola 887705)

Sommario

1. Introduzione

1.1. Scopo del progetto e specifiche

La Prova Finale di Reti Logiche per l'anno 2020/2021 ha come obiettivo l'implementazione di un componente hardware, descritto in VHDL, che, letta un'immagine dalla memoria, applichi il metodo di equalizzazione dell'istogramma di un'immagine tramite un algoritmo semplificato, riportato in *Figura 1*.

L'equalizzazione dell'istogramma di un'immagine è un metodo usato per ricalibrare il contrasto dell'immagine stessa: i valori di intensità, quando troppo vicini, sono ridistribuiti su tutto l'intervallo di intensità e ciò comporta un aumento del contrasto.

L'algoritmo viene applicato solo a immagini in scala di grigio a 256 livelli.

Non è possibile sostituire l'immagine mentre il processo di equalizzazione è in corso; sono invece consentite più codifiche in successione. La dimensione massima consentita per un'immagine è di 128x128 pixel.

```
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)
```

Figura 1:
max_pixel_value e min_pixel_value sono il valore massimo e minimo dei pixel dell'immagine, new_pixel_value è il nuovo valore del pixel, current_pixel_value è il valore da trasformare.

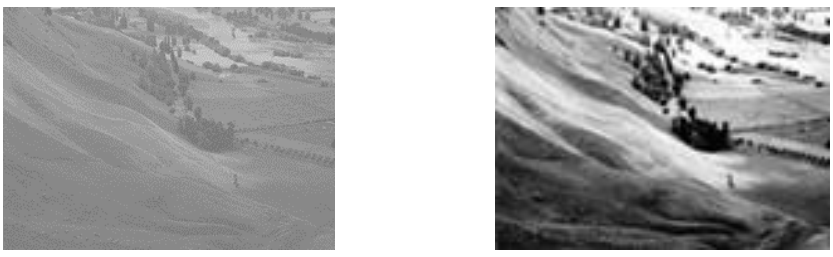

Figure 2 e 3:
esempio di applicazione del metodo di equalizzazione dell'istogramma di un'immagine.
A sinistra, l'immagine non equalizzata, a destra la medesima immagine dopo l'equalizzazione.

1.2. Interfaccia del componente

Il componente da descrivere ha un'interfaccia così definita:

```
entity project_reti_logiche is
      port (
            i_clk     : in std_logic;
            i_rst     : in std_logic;
            i_start   : in std_logic;
            i_data    : in std_logic_vector(7 downto 0);
            o_address : out std_logic_vector(15 downto 0);
            o_done    : out std_logic;
            o_en      : out std_logic;
            o_we      : out std_logic;
            o_data    : out std_logic_vector (7 downto 0)
      );
end project_reti_logiche;
```

In particolare:

- ***i_clk*** è il segnale di ***CLOCK*** in ingresso generato dal TestBench;
- ***i_rst*** è il segnale di ***RESET*** che inizializza la macchina pronta per ricevere il primo segnale di ***START***;
- ***i_start*** è il segnale di ***START*** generato dal TestBench;
- ***i_data*** è il segnale (vettore) che arriva dalla memoria in seguito ad una richiesta di lettura;
- ***o_address*** è il segnale (vettore) di uscita che manda l'indirizzo alla memoria;
- ***o_done*** è il segnale di uscita che comunica la fine dell'elaborazione e il dato di uscita scritto in memoria;
- ***o_en*** è il segnale di ***ENABLE*** da dover mandare alla memoria per poter comunicare (sia in lettura che in scrittura);
- **o_we** è il segnale di ***WRITE ENABLE*** da dover mandare alla memoria (=1) per poter scriverci. Per leggere da memoria esso deve essere 0;
- ***o_data*** è il segnale (vettore) di uscita dal componente verso la memoria.

1.3. Dati e descrizione memoria

L'immagine da equalizzare viene letta da una memoria con indirizzamento al byte in modo sequenziale e riga per riga; la sua dimensione è indicata da 2 byte e non può superare il massimo di 128x128.

Il byte corrispondente al numero di colonna è memorizzato all'indirizzo 0, quello corrispondente al numero di riga all'indirizzo 1; dall'indirizzo 2 in poi sono memorizzati in byte contigui i pixel dell'immagine originale (da equalizzare), ciascuno codificato su 8 bit.

L'immagine equalizzata viene scritta in memoria immediatamente dopo l'immagine originale, di conseguenza i suoi pixel vengono memorizzati a partire dall'indirizzo 2+(NCOL*NRIG), dove NCOL e NRIG sono rispettivamente il numero di colonne e di righe dell'immagine.

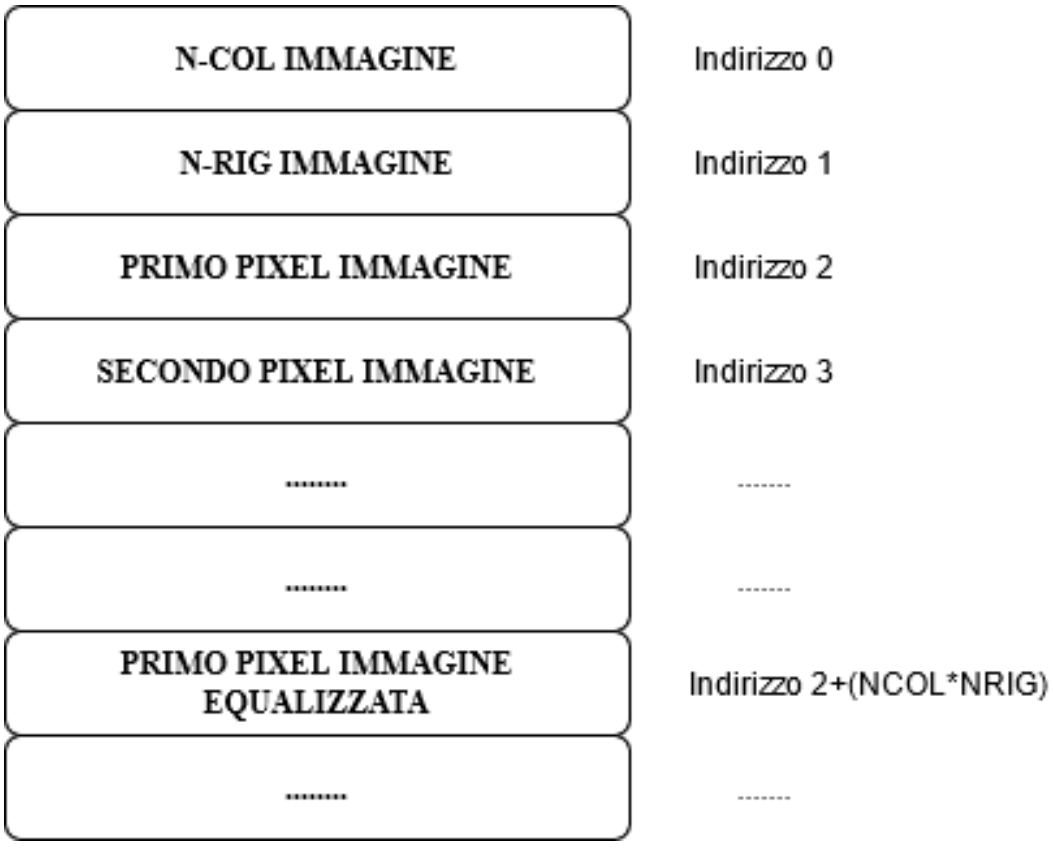

Figura 4: breve rappresentazione grafica degli indirizzi della memoria.

2. Design

Quando il segnale ***i_start*** viene portato a 1, il componente sviluppato inizia l'elaborazione spostandosi dallo stato ***INIZIO*** al primo stato della computazione. Una volta terminata la computazione, dopo aver scritto il risultato in memoria, il componente alza il segnale ***o_done***; quando la test bench risponde abbassando ***i_start***, il componente passa allo stato ***SET_DONE_DOWN*** che riporta a 0 ***o_done*** e poi ritorna allo stato ***INIZIO***, in attesa che il segnale ***i_start*** torni alto.

Il componente dispone inoltre di un segnale ***i_rst*** che, quando viene alzato, fa tornare la computazione allo stato iniziale.

La macchina a stati finiti da noi progettata si compone di una parte combinatoria che permette lo spostamento tra i vari stati e di una parte sequenziale che consente la gestione dei registri utilizzati.

2.1. Stati della macchina

La macchina è composta da 15 stati. Nei sottoparagrafi successivi ne viene fornita una breve descrizione.

2.1.1. Stato 1: INIZIO

Stato **iniziale** in cui si attende il segnale di ***i_start***. In caso venga alzato il segnale ***i_rst*** si torna in questo stato.

2.1.2. Stato 2: RICHIEDI_COL_RIGA

Stato in cui viene richiesto alla memoria il numero di colonne dell'immagine (***N-COL***) e successivamente il numero di righe (***N-RIG***). Il tutto viene fatto tornando due volte in questo stato in quanto non è possibile effettuare queste operazioni in simultanea.

2.1.3. Stato 3: ATTESA_RAM

Stato in cui si attende la risposta dalla memoria in seguito alla richiesta di un dato.

2.1.4. Stato 4: RICEVI_COL_RIGA

Stato in cui vengono memorizzati, in due rispettivi registri, il numero di colonne e il numero di righe dell'immagine.

2.1.5. Stato 5: CALCOLA_DIM

Stato in cui viene calcolata la dimensione dell'immagine da equalizzare e successivamente viene impostato l'indirizzo base in cui verrà memorizzato il primo pixel equalizzato. Se la dimensione dell'immagine è 0, non viene effettuato il calcolo dell'indirizzo e si passa direttamente allo stato di **FINE**.

2.1.6. Stato 6: RICHIEDI_PIXEL

Stato in cui viene richiesto l'indirizzo del pixel da equalizzare. Si torna in questo stato per un numero di volte pari al doppio del numero di pixel dell'immagine, in quanto non è possibile analizzare più di un pixel contemporaneamente e in quanto è necessario scandire l'immagine due volte, una per la ricerca del massimo e del minimo e una per la computazione del nuovo valore del pixel. La seconda lettura è segnalata da un boolean *secondRead* posto a true.

2.1.7. Stato 7: RICEVI_PIXEL

Stato in cui viene memorizzato il pixel ricevuto in un registro e viene incrementato il contatore dei pixel letti. A seconda del valore assunto da *secondRead* fa proseguire la computazione nello stato **MIN_MAX** (*secondRead=false*) o nello stato **NUOVO_PIXEL.**

2.1.8. Stato 8: MIN_MAX

Stato in cui, analizzando i pixel ricevuti, vengono trovati e successivamente salvati in due registri differenti i pixel dell'immagine con valore massimo e minimo.

2.1.9. Stato 9: DELTA_VALUE

Stato in cui, una volta terminata la prima lettura dell'immagine, viene calcolato il ***DELTA_VALUE*** utilizzando i valori del pixel massimo e minimo, come riportato in *Figura 1*.

2.1.10. Stato 10: SHIFT

Stato in cui viene calcolato, in base al ***DELTA_VALUE***, lo ***SHIFT*** per i pixel dell'immagine, come riportato in *Figura 1*. Viene poi azzerato il contatore dei pixel letti e impostato a true *secondRead* per segnalare l'inizio della seconda lettura dell'immagine.

2.1.11. Stato 11: NUOVO_PIXEL

Stato in cui viene calcolato, per ciascun pixel dell'immagine, il corrispettivo pixel equalizzato secondo la formula che compare in *Figura 1*.

2.1.12. Stato 12: CHECK_PIXEL

Stato in cui viene effettivamente assegnato il nuovo valore del pixel letto: se il valore trovato allo stato precedente (**NUOVO_PIXEL**) è minore di 255, esso diventerà il valore del pixel dopo l'equalizzazione, altrimenti il nuovo valore è fissato a 255. Questa operazione è necessaria in quanto le specifiche prevedono immagini con valori in scala di grigio a 256 bit (da 0 a 255), dunque non è possibile avere un pixel con valore maggiore di 255.

2.1.13. Stato 13: SCRITTURA

Stato in cui i nuovi pixel equalizzati vengono scritti in memoria uno ad uno a partire dall'indirizzo calcolato nello stato ***CALCOLA_DIM***.

2.1.14. Stato 14: FINE

Stato in cui si attende che la memoria abbassi ***i_start*** per passare allo stato ***SET_DONE_DOWN.***

2.1.15. Stato 15: SET_DONE_DOWN

Stato in cui viene settato ***o_done*** a 0 e la macchina torna allo stato ***INIZIO***.

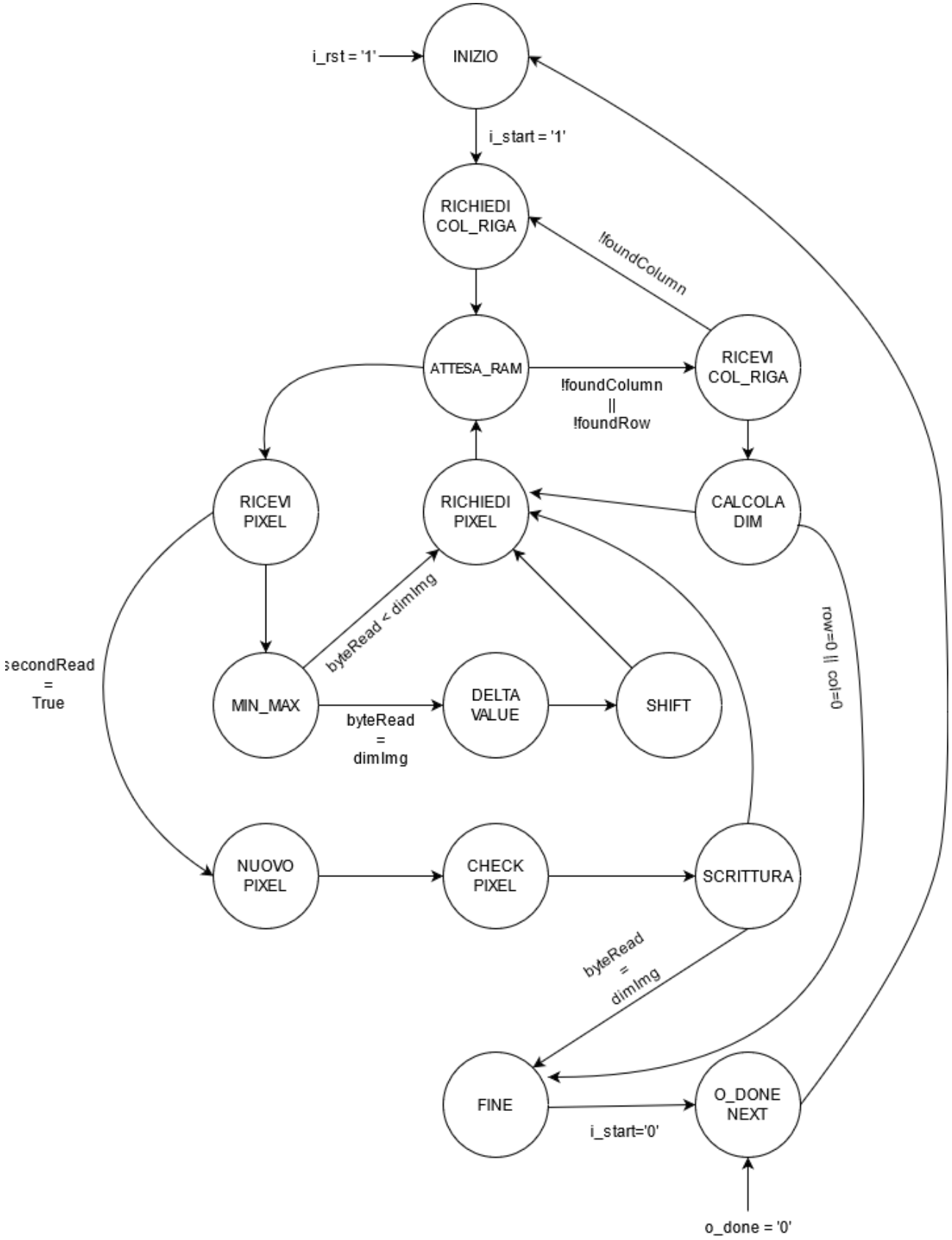

Figura 5: rappresentazione degli stati della macchina con le principali condizioni di transizione.

2.2. Scelte Progettuali

Il componente è stato implementato con due process: il primo, sensibile a clock e reset, si occupa della parte sequenziale e della gestione e manipolazione dei registri, mentre il secondo gestisce la parte combinatoria, ovvero il passaggio da uno stato all'altro.

Abbiamo inoltre scelto di mantenere separati gli stati di richiesta e ricezione di riga/colonna e dei pixel per una maggiore chiarezza nel codice scritto.

3. Risultati della sintesi

Il componente descritto al paragrafo precedente, realizzato in VHDL, è stato sintetizzato da Vivado con una rete composta da 257 LUT e 172 Flip-Flop, che servono per memorizzare la gran quantità di dati necessari all'equalizzazione dell'immagine.

Inoltre, dopo la sintesi del componente, ci siamo accertati che non fossero presenti latch inferiti che potessero comprometterne il corretto funzionamento.

Site Type	Used	Fixed	Available	Util%
Slice LUTs*	257	0	134600	0.19
LUT as Logic	257	0	134600	0.19
LUT as Memory	0	0	46200	0.00
Slice Registers	172	0	269200	0.06
Register as Flip Flop	172	0	269200	0.06
Register as Latch	0	0	269200	0.00
F7 Muxes	0	0	67300	0.00
F8 Muxes	0	0	33650	0.00

Figura 6: Stralcio del report di sintesi.

Inoltre, abbiamo realizzato la sintesi anche con un clock di 10 ns, inferiore a quello richiesto dalle specifiche (100 ns), per essere maggiormente sicuri che il componente rispetti il costraint di clock richiesto dalle specifiche; come anticipato, il componente sintetizzato è in grado di lavorare anche a frequenze maggiori, rispettando il costraint di 10 ns: in questo caso il report mostra un datapath delay di 8.843 ns e uno slack di 1.006 ns.

4. Risultati dei test

Abbiamo verificato il corretto funzionamento del componente da noi realizzato mediante numerosi test, oltre a quello fornito come esempio, per coprire il maggior numero di cammini possibili effettuati dalla macchina e testare eventuali casi limite. Di seguito riportiamo alcuni test svolti, inserendo anche l'andamento dei segnali per i più importanti.

4.1. Testing dei casi limite

Tutti i test riportati in questo paragrafo sono stati effettuati al fine di testare se la macchina opera correttamente in condizioni di un comportamento non standard.

- **Immagine vuota:** questo test si pone l'obiettivo di verificare il corretto funzionamento della macchina nel caso in cui le righe e le colonne dell'immagine siano 0. In questo caso, dopo il calcolo delle dimensione dell'immagine, si passa direttamente allo stato di fine, terminando l'elaborazione.

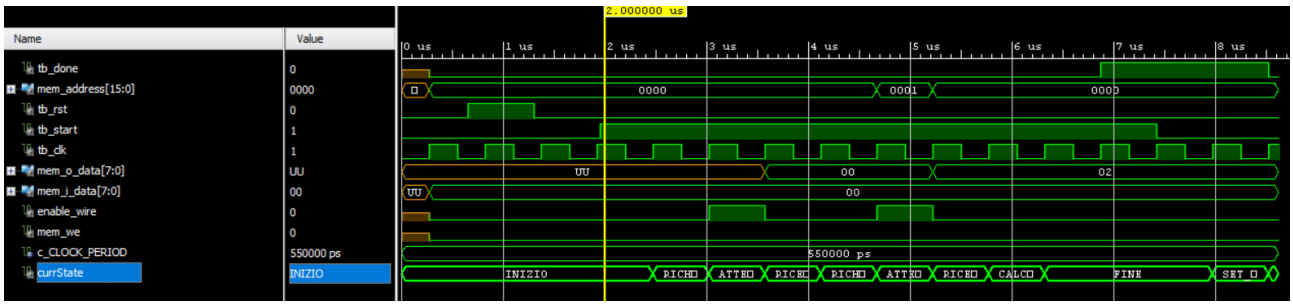

Figura 7: andamento dei segnali nel test di immagine vuota. Si noti come curr_state cambia a FINE dopo il calcolo dell'immagine, segnalando il corretto funzionamento del componente.

- **Riga o colonna 0:** simile al precedente, ma o la riga o la colonna hanno dimensione diversa da 0. Il test verifica che, anche in questo caso, non vengano richiesti i pixel da elaborare e che dopo il calcolo della dimensione dell'immagine si passi nello stato di fine.

- **Pixel con lo stesso valore:** questo test si pone l'obiettivo di verificare il corretto funzionamento della macchina nel caso in cui lo *shift level* sia massimo (cioè pari a 8): in questo caso tutti i pixel dell'immagine equalizzata hanno valore 0.

4.2. Testing del corretto funzionamento dei segnali

I test riportati di seguito sono stati effettuati per verificare il corretto funzionamento dei segnali in casi particolari.

- **Reset asincrono:** test che verifica che, ricevuto il segnale di reset in maniera asincrona durante un'elaborazione, essa si interrompa e la macchina torni allo stato iniziale.

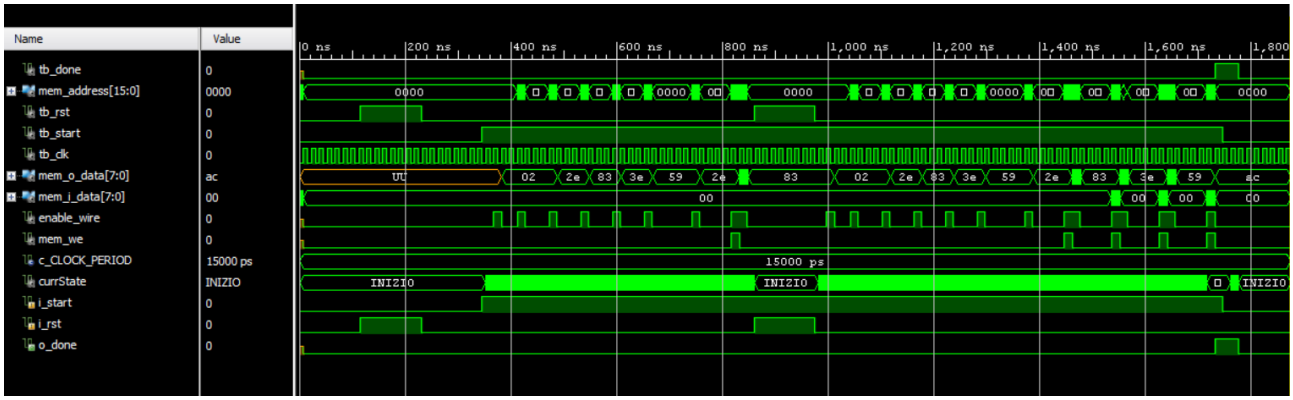

Figura 8: andamento dei segnali nel test di reset asincrono. Si noti il corretto avvicendamento degli stati in base al segnale di reset.

- **Elaborazione multipla:** questo test verifica la corretta sincronizzazione dei segnali *i_start*, *i_rst* e *o_done* nel caso in cui si vogliano elaborare più immagini diverse in successione o la stessa immagine per più volte di fila. In particolare, si osservi come il segnale di reset venga alzato solo prima della prima elaborazione. Le immagine successive vengono correttamente elaborate senza che sia nuovamente necessario portare alto *i_rst*.

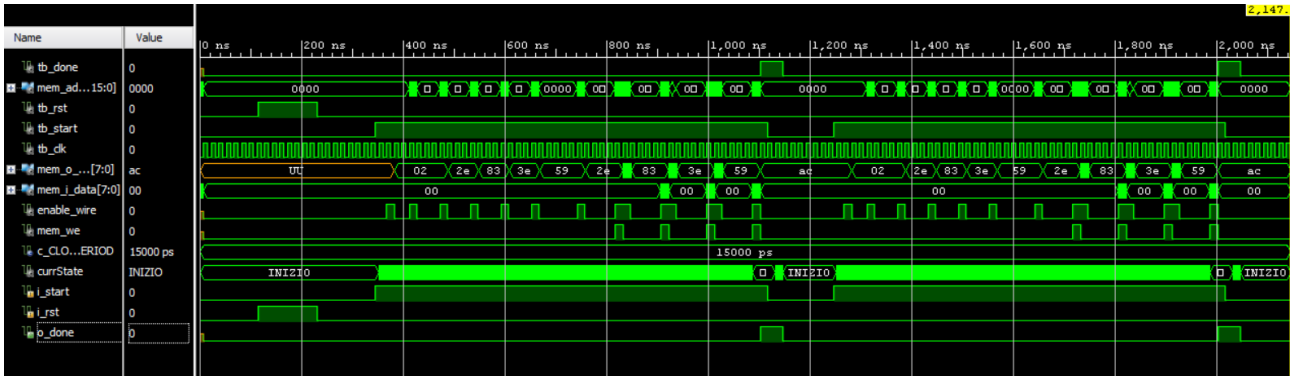

Figura 9: andamento dei segnali nel test di elaborazione multipla.Nello specifico questo screenshot raccoglie le informazioni sui segnali relative all'equalizzazione di due immagini in sequenza.

4.3. Altri test

Sono poi stati effettuati altri test per verificare il corretto funzionamento della macchina con un comportamento standard. Tra questi si segnalano:

- **Immagine di dimensione massima:** questo test verifica il corretto funzionamento del componente nel caso in cui l'immagine abbia dimensione massima, ovvero 128x128 pixel.
- **Immagine di dimensione minima:** questo test verifica il corretto funzionamento del componente nel caso in cui l'immagine abbia dimensione minima, ovvero 1 pixel.
- **Massimo e minimo come ultimi pixel dell'immagine:** questo test verifica il corretto funzionamento del componente nel caso in cui il massimo e il minimo siano gli ultimi pixel dell'immagine; questo comporta uno stress sul componente in quanto deve sempre aggiornare i valori di massimo e minimo.

Inoltre, abbiamo testato il componente con numerosi test randomici ottenuti tramite il generatore messo a disposizione.

5. Conclusioni e possibili ottimizzazioni

Il componente sintetizzato ha superato correttamente tutti i test a cui è stato sottoposto nelle simulazioni *Behavioral* e *Post-Synthesis*. Ci sentiamo quindi abbastanza sicuri di poter affermare che il componente presenta un corretto funzionamento, anche con un costraint di clock più restrittivo di quello richiesto, come spiegato nel paragrafo 3.

Inoltre, il componente realizzato è riutilizzabile: se volessimo equalizzare immagini di dimensione maggiore a quella massima, basterebbe solo cambiare il range delle variabili al momento della loro dichiarazione all'inizio del codice. Se invece volessimo elaborare immagini con un diverso numero di livelli, oltre a cambiare il range delle variabili, bisognerebbe anche modificare il controllo effettuato nello stato **CHECK_PIXEL**, impostando il massimo valore dei pixel consentito.

Infine, una possibile ottimizzazione può essere effettuata sulla ricerca del massimo e del minimo: prendendo ad esempio un immagine in scala di grigi a 256 livelli, sappiamo che il minimo valore di un pixel è 0, mentre il massimo 255. Alla luce di ciò, è possibile interrompere la ricerca del massimo quando viene letto un pixel con valore 255 e quella del minimo quando viene letto un pixel con valore 0, in quanto sappiamo con assoluta certezza che non ci saranno valori superiori a 255 o inferiori a 0. Il risvolto positivo di questa ottimizzazione è che si risparmia tempo sulla prima lettura dell'immagine, non essendo sempre necessario scorrere tutta l'immagine per trovare il massimo e il minimo valore; dall'altro lato questa ottimizzazione penalizza la scalabilità nel caso di immagini con livelli diversi da 256, poiché bisognerebbe modificare anche il codice di ricerca del massimo e del minimo, aggiornandolo con i valori desiderati. Per quest'ultimo motivo e per non appesantire le transizioni degli stati, abbiamo deciso di non implementare questa – seppur valida – ottimizzazione.