
POLITECNICO
DI MILANO

# Prova Finale Reti Logiche

Prof. Terraneo Federico

Prof. Fornaciari William

Anno Accademico 2020/2021

**Membri Gruppo:**

Campo Marco Lorenzo - 886807 (ID: 10581062)

De Luca Alessandro - 908706 (ID: 10676114)

Indice

1. Introduzione al progetto
   1.1 Obiettivo del progetto
      1.1.1 Descrizione dell'algoritmo
   1.2 Descrizione memoria RAM
   1.3 Interfaccia del componente

2. Design e scelte progettuali
   2.1 Modello logico
      2.1.1 Processo di Controllo
      2.1.2 Datapath

3. Testbench e simulazioni
   3.1 Testing
      3.1.1 Valori di *SHIFT_LEVEL*
      3.1.2 Dimensioni critiche e computazioni successive
      3.1.3 Reset asincrono
   3.2 Requisiti temporali

4. Conclusioni

# 1. Introduzione al progetto

## 1.1 Obiettivo del progetto

Siano fornite da RAM una o più immagini di dimensione n x m, con n, m ∈ {0,...,128}: l'obiettivo del progetto è di sviluppare un componente *hardware* che aumenti il contrasto delle immagini in input applicando un algoritmo di *equalizzazione dell'istogramma*. Nello specifico vengono analizzate immagini i cui pixel sono in scala di grigi tra 0 e 255.

### 1.1.1 Descrizione dell'Algoritmo

Data in ingresso un'immagine, vengono identificati due valori chiave: *MIN_PIXEL_VALUE* e *MAX_PIXEL_VALUE* che contengono rispettivamente l'intensità minima e massima riscontrati nell'input. Da essi è possibile determinare il fattore di scala dell'istogramma *SHIFT_LEVEL*.

Si procede sottraendo ad ogni pixel *MIN_PIXEL_VALUE* per poi eseguire uno shift logico a sinistra con fattore *SHIFT_LEVEL*. Se il risultato della computazione rientra nel range ammissibile (massimo 255) allora viene assegnato al pixel equalizzato, altrimenti gli viene assegnato 255.

Questo algoritmo dilata lo spettro di intensità dei pixel in ingresso assegnando *MIN_PIXEL_VALUE* a 0 e mappando i restanti valori secondo il fattore di scala.

- La procedura di calcolo è la seguente: (1.1.a)

```
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)
```

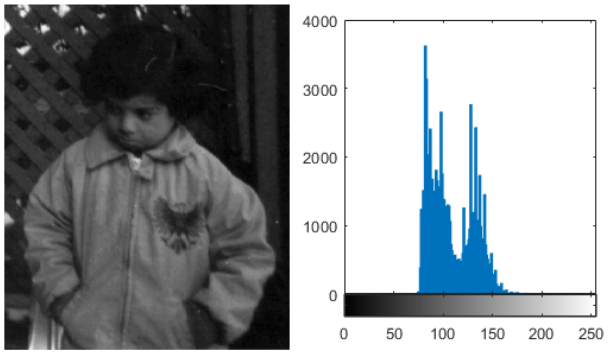



Fig.1.

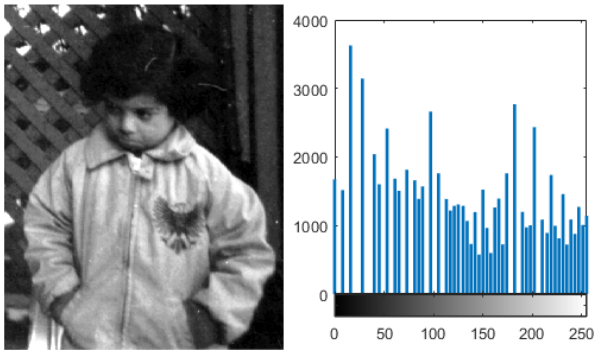



Fig.2.

Fig.1: Immagine originale, i pixel hanno intensità compresa tra 75 e 175;

Fig.2: L'immagine è stata equalizzata, i valori di intensità dei pixel sono mappati tra 0 e 255;

## 1.2 Descrizione memoria RAM

I dati, di dimensione 8 bit, sono memorizzati in RAM indirizzata a 16 bit: i primi due byte indicano le dimensioni dell'immagine (numero di *colonne* e *righe*), gli indirizzi successivi contengono i pixel da equalizzare.

Il salvataggio dei dati modificati avviene consecutivamente all'immagine in input, a partire dall'indirizzo *'colonne*righe + 2'* e occuperà la stessa dimensione.

*Esempio*: Un'immagine 2x2 sarà fornita dalla RAM in questo modo:

| 0 | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|
| 2 | 2 | 3 | 13 | 133 | 67 |

L'immagine viene equalizzata applicando la (1.1.a):

- *MIN_PIXEL_VALUE* = 3;
- *MAX_PIXEL_VALUE* = 133;
- *DELTA_VALUE=133-3=130;*
- *SHIFT_LEVEL* = 8 – FLOOR(LOG2(131)) = 1;

I pixel equalizzati sono salvati a partire dall'indirizzo 6:

| 6 | 7 | 8 | 9 | 10 | 11 |
|---|---|---|---|---|---|
| 0 | 20 | 255 | 128 | Ø | Ø |

## 1.3 Interfaccia del componente

Il componente da descrivere ha la seguente interfaccia:

```
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;
```

Nello specifico i segnali in ingresso forniti da Testbench sono:

- **i_clk:** Segnale di clock;
- **i_rst:** Segnale di reset del componente;
- **i_start:** Segnale di inizio computazione;
- **i_data:** Segnale di 8 bit contenente il pixel in ingresso;

I segnali di output sono prodotti dal componente:

- **o_address:** Indirizzo di lettura o scrittura in memoria;
- **o_done:** Segnale asserito al termine dell'equalizzazione;
- **o_en:** Segnale asserito per attivare la RAM;
- **o_we:** Segnale per l'abilitazione della scrittura in memoria;
- **o_data**: Segnale di 8 bit che contiene il pixel equalizzato;

# 2. Design e scelte progettuali

## 2.1 Modello logico

Il componente è stato implementato secondo un approccio *multi-processo* in cui i due fondamentali sono il *Processo di Controllo (FSM)* e il *Datapath* che comunicano attraverso segnali[fig.3].

La FSM è una *macchina di Moore* composta da 11 stati e il passaggio tra essi è definito a run-time da una funzione di transizione δ(S,I) che analizza lo stato attuale S, i segnali dati in input alla FSM (da testbench o da componente) e determina lo stato futuro S*.

Il Datapath è controllato dalla FSM e implementa l'unità di calcolo e di gestione RAM utilizzando quattro classi di componenti: *Registri*, *Contatori*, *Multiplexer* e *Encoder*.

Quando viene alzato il segnale **`i_start`**, si ha una transizione dallo stato IDLE a START che determina l'inizio della computazione. Quando quest'ultima termina, l'immagine equalizzata è salvata in memoria e il segnale **`o_done`** è posto a '1'. Il testbench risponde ponendo **`i_start`** a 0, il componente abbassa **`o_done`** e la macchina torna nello stato di IDLE in attesa di una nuova computazione.

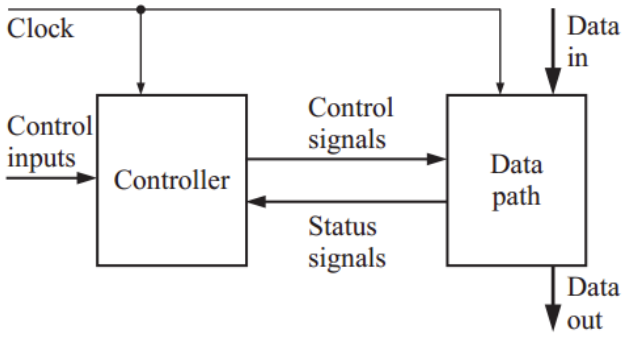
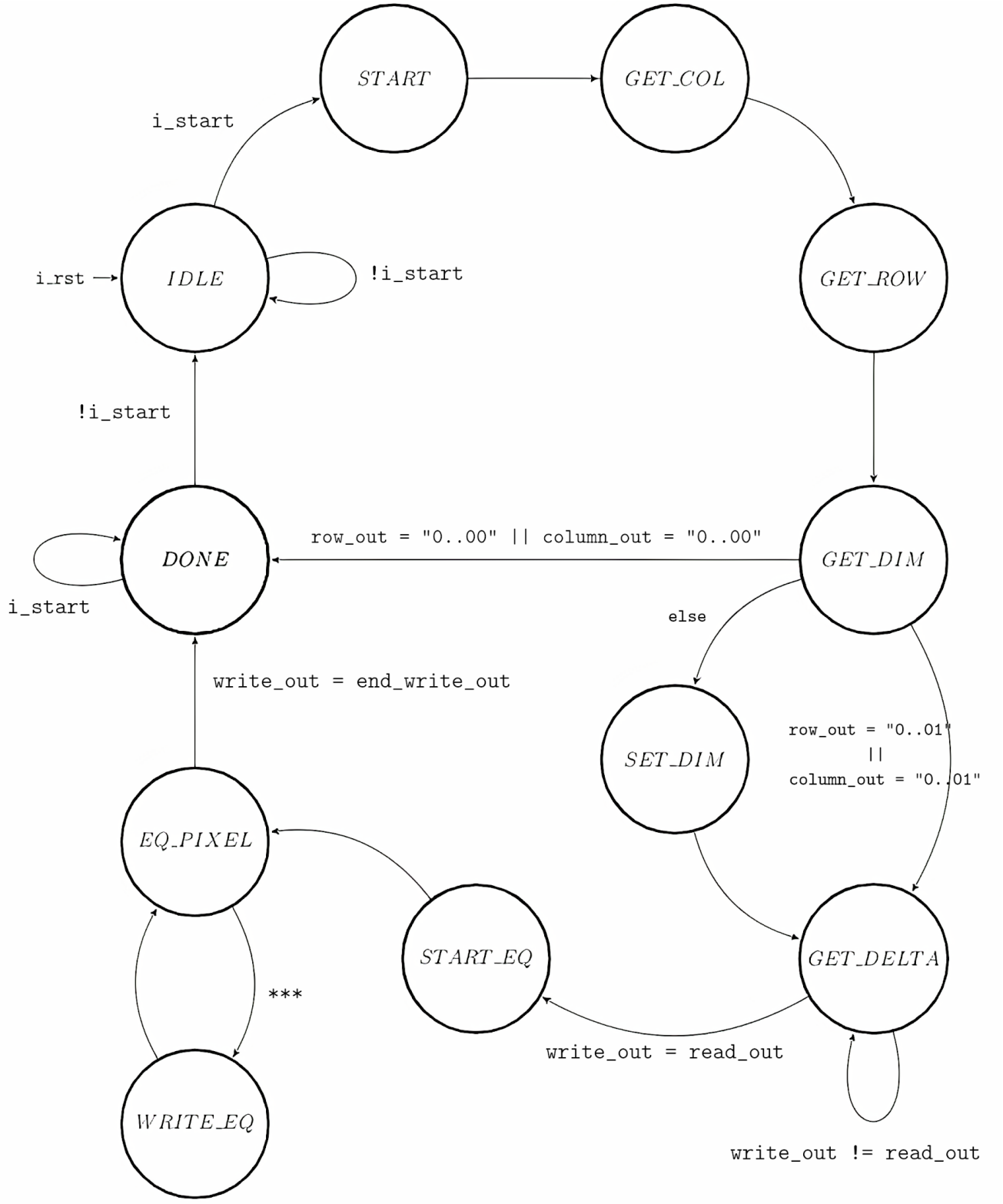



Fig.3 : Schema logico per l'implementazione di Datapath e Controllore come processi in comunicazione.

### 2.1.1 Processo di Controllo

Il controllore è rappresentabile mediante una FSM[fig.4]. Essa analizza i segnali forniti da testbench e lo stato attuale S per determinare la transizione a $S^* = \delta(S,I)$, stato futuro. Gli stati della FSM sono:

- **IDLE**: Stato iniziale della macchina in cui si attende il segnale di inizio computazione `i_start`. L'asserzione di `i_rst` riporta la macchina in questo stato.
- **START**: Stato buffer che avvia la computazione chiedendo il byte in posizione 0 alla RAM.
- **GET_COL**: Stato in cui il numero di colonne viene salvato in `column_out.`
- **GET_ROW**: Stato in cui il numero di righe viene salvato in `row_out.`
- **GET_DIM**: Stato in cui viene inizializzato il contatore per la moltiplicazione e vengono effettuati controlli su `row_out` e `column_out` per decidere lo stato successivo S*.
- **SET_DIM**: esegue la moltiplicazione `row_out*column_out`, e vengono calcolati `write_out` (primo indirizzo di scrittura) e `end_write_out` (ultimo indirizzo di scrittura).
- **GET_DELTA**: stato in cui vengono salvati i valori di intensità massima e minima in ingresso e calcolati *DELTA_VALUE* e *SHIFT_LEVEL*.
- **START_EQ**: stato buffer per riportare la RAM al primo pixel ed iniziare l'equalizzazione.
- **EQ_PIXEL**: stato in cui viene richiesto alla RAM il nuovo pixel da equalizzare
- **WRITE_EQ**: stato di calcolo e salvataggio di *NEW_PIXEL_VALUE*.
- **DONE**: stato in cui viene posto `o_done = '1'` e termina la computazione. Il passaggio ad IDLE avviene quando `i_start` torna a 0.




***: write_out != end_write_out

Fig.4: Rappresentazione della FSM.

### 2.1.2 Datapath

- **Registri**:
    1. Memorizzano informazioni utili a livello computazionale. **`max_out`** contiene *MAX_PIXEL_VALUE*, **`min_out`** contiene *MIN_PIXEL_VALUE*, **`column_out`** e **`row_out`** contengono rispettivamente RAM[0] e RAM[1];
    2. Gestiscono la RAM: il registro **`read_out`** salva l'indirizzo di lettura; **`write_out`** salva l'indirizzo di scrittura; **`end_write_out`** marca l'ultimo indirizzo utile per memorizzare l'immagine equalizzata.

- **Multiplexer:** Il loro impiego è necessario per la gestione degli indirizzi di memoria e per l'assegnamento di *NEW_PIXEL_VALUE*.

- **Contatore:** La moltiplicazione è implementata come somma ripetuta. Il numero di iterazioni viene determinato dal minore tra **`row_out`** e **`column_out`**:

    - column_out > row_out:

$$row\_out * column\_out = \overbrace{column\_out + \ldots + column\_out}^{row\_out}$$

    - column_out <= row_out :

$$row\_out * column\_out = \overbrace{row\_out + \ldots + row\_out}^{column\_out}$$

  Il contatore è decrementato ad ogni iterazione, fino ad ottenere "**`row_out`***`column_out`" usato per il calcolo di **`write_out`** ed **`end_write_out`**.

- **Encoder**: Il componente esegue un controllo di soglia su $\Delta = DELTA\_VALUE + 1$ per ricavare la parte intera del logaritmo, successivamente manda in output 8-FLOOR($LOG_2(\Delta)$). Ciò è possibile perché *SHIFT_LEVEL* è un valore compreso tra 0 e 8 facilmente determinabile dall'uno in posizione più significativa di Δ

  Vale quindi la relazione:

  $\forall\, \Delta \in \{1,\cdots,256\}$, SHIFT_LEVEL = $(8 - FLOOR(LOG_2(\Delta))) = n$, $n \in \{0,\cdots,8\}$

| Δ | Δ[8] | Δ[7] | Δ[6] | Δ[5] | Δ[4] | Δ[3] | Δ[2] | Δ[1] | Δ[0] | S[3] | S[2] | S[1] | S[0] | SHIFT_ LEVEL |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 256 | **1** | X | X | X | X | X | X | X | X | 0 | 0 | 0 | 0 | **0** |
| 128-255 | 0 | **1** | X | X | X | X | X | X | X | 0 | 0 | 0 | **1** | **1** |
| 64-127 | 0 | 0 | **1** | X | X | X | X | X | X | 0 | 0 | **1** | 0 | **2** |
| 32-63 | 0 | 0 | 0 | **1** | X | X | X | X | X | 0 | 0 | **1** | **1** | **3** |
| 16-31 | 0 | 0 | 0 | 0 | **1** | X | X | X | X | 0 | **1** | 0 | 0 | **4** |
| 8-15 | 0 | 0 | 0 | 0 | 0 | **1** | X | X | X | 0 | **1** | 0 | **1** | **5** |
| 4-7 | 0 | 0 | 0 | 0 | 0 | 0 | **1** | X | X | 0 | **1** | **1** | 0 | **6** |
| 2-3 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | **1** | X | 0 | **1** | **1** | **1** | **7** |
| 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | **1** | **1** | 0 | 0 | 0 | **8** |

Fig.5: Δ[i]: bit i-esimo di *DELTA_VALUE* + 1; S[i] = bit i-esimo di *SHIFT_LEVEL*;

# 3. Testbench e simulazioni

## 3.1 Testing

I fattori che hanno influenzato la scelta dei Testbench sono essenzialmente quattro:

- I possibili valori di *SHIFT_LEVEL*;
- La *dimensione* dell'immagine;
- Il *numero* di immagini da equalizzare;
- L'eventuale presenza di *reset asincroni*;

### 3.1.1 Valori di *SHIFT_LEVEL*

Il funzionamento del componente è stato valutato sia con i dati forniti dalla specifica, sia con test *ad hoc* in in grado di simulare tutti i possibili fattori di scala.

A fronte dell'implementazione scelta per il calcolo di *SHIFT_LEVEL*, il componente funziona correttamente indipendentemente dai valori di intensità dei pixel nell'immagine.

### 3.1.2 Dimensioni critiche e computazioni successive

Sono stati individuati tre *corner cases* relativi alla dimensione delle immagini: massima 128x128, degenere nx0/0xn/0x0 e minima 1x1. Dopo aver valutato il corretto funzionamento con le singole immagini, il componente è stato sottoposto ad un test contenente i tre *corner cases* consecutivi. Nello specifico sono state scelte tre immagini: una 4x0, una 1x1 e una 128x128.

Anche in questo caso il risultato è stato conforme alla specifica: l'indirizzo di lettura è riportato a 0 in corrispondenza di una nuova computazione ed i controlli implementati riportano la macchina nello stato IDLE nel caso di immagini degeneri.

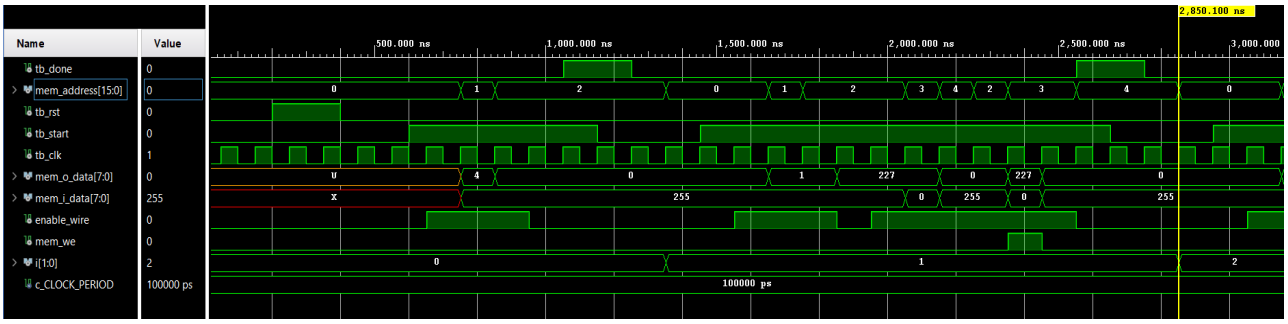



Fig.6: Output con immagini 0x4 e 1x1.

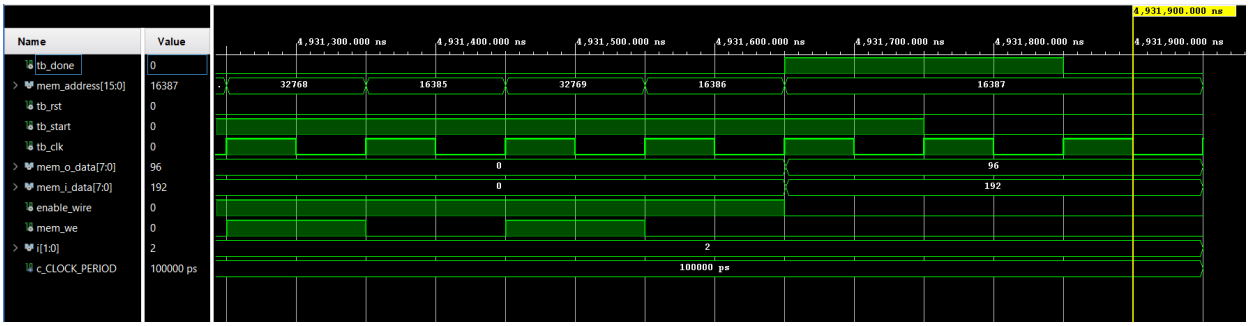



Fig.7: Output con immagine 128x128.

### 3.1.3 Reset asincrono:

In questo test il segnale **`i_rst`** è stato asserito in modo asincrono; il componente segue la specifica ritornando nello stato di IDLE e attende una nuova computazione.

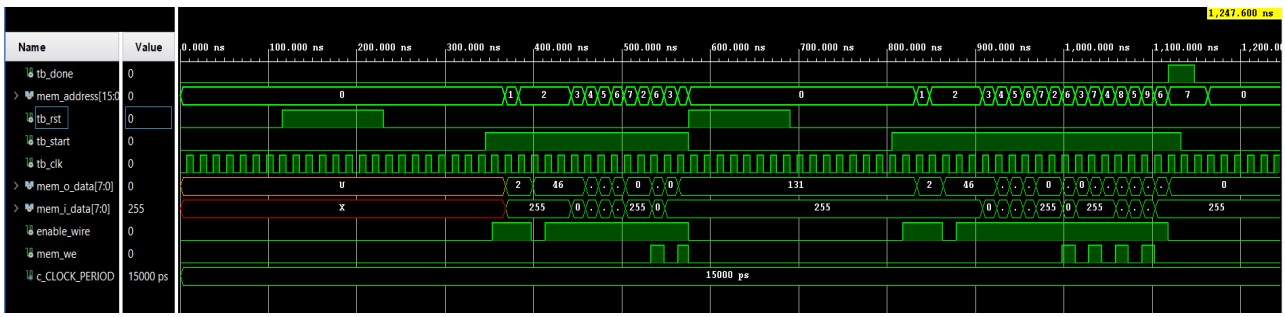



Fig.8: Output nel caso di reset asincrono.

## 3.2 Requisiti temporali

La specifica impone un periodo di clock massimo di 100 ns che risulta più che sufficiente come mostrato dal Timing Report:

```
Data Path Delay:        4.965ns  (logic 2.727ns (54.924%)  route 2.238ns
(45.076%))
Slack (MET) :           94.884ns  (required time - arrival time)
```

Si può quindi assumere un corretto funzionamento del componente fino ad una frequenza di 201MHz.

# 4. Conclusioni

Il componente hardware sviluppato è in grado di soddisfare le specifiche richieste, sia a livello di timing che di correttezza dell'output. Risulta simulabile in Behavioral Pre-Synthesis e sintetizzabile in Functional Post-Synthesis senza introduzione di latch da parte di VIVADO[fig.9].

Le scelte progettuali attuate hanno come finalità un utilizzo minimo degli elementi di memoria: tutti i segnali non salvati in registri sono ininfluenti sul comportamento globale del componente e vengono controllati solo quando richiesti dalla FSM.

| Site Type | Used | Fixed | Available | Util% |
|---|---|---|---|---|
| Slice LUTs* | 174 | 0 | 134600 | 0.13 |
| LUT as Logic | 174 | 0 | 134600 | 0.13 |
| LUT as Memory | 0 | 0 | 46200 | 0.00 |
| Slice Registers | 99 | 0 | 269200 | 0.04 |
| Register as Flip Flop | 99 | 0 | 269200 | 0.04 |
| Register as Latch | 0 | 0 | 269200 | 0.00 |
| F7 Muxes | 0 | 0 | 67300 | 0.00 |
| F8 Muxes | 0 | 0 | 33650 | 0.00 |

Fig.9: Componente sintetizzabile senza inferenza di latch.