Politecnico di Milano
Dipartimento di Elettronica, Informazione e Bioingegneria

Progetto di Reti Logiche 2020/21

Prof. Palermo Gianluca

Shalby Hazem Hesham Yousef (Codice Persona: 10596243, Matricola: 910871)
Perego Niccolò (Codice Persona: 10628782, Matricola: 895468)

1 Aprile 2021

POLITECNICO
MILANO 1863

Indice

1 Requisiti di progetto

1.1 Descrizione del problema

Si vuole realizzare un componente in grado di svolgere una versione semplificata del processo di equalizzazione dell'istogramma di un'immagine, ossia di ricalibrare il contrasto di quest'ultima, effettuando una ridistribuzione dei valori di intensità pixel per pixel.

Le immagini di cui è richiesta la manipolazione sono definite in scala di grigi a 256 livelli e hanno una dimensione massima di `128x128` pixel.

1.2 Interfaccia del componente

Il componente deve rispettare un'interfaccia standard così definita in liguaggio VHDL:

```
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, predisponendola alla ricezione del segnale di START. Può essere anche asincrono;
- `i_start` è il segnale di START generato dal Test Bench;
- `i_data` è il segnale (vettore) che arriva dalla memoria in seguito a 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 mandare alla memoria per poter comunicare (sia in lettura che in scrittura);
- `o_we` è il segnale di WRITE ENABLE da mandare alla memoria per comunicare quale operazione si vuole svolgere su di essa. Può assumere valori `0` e `1`, rispettivamente per lettura e scrittura;
- `o_data` è il segnale (vettore) di uscita dal componente verso la memoria.

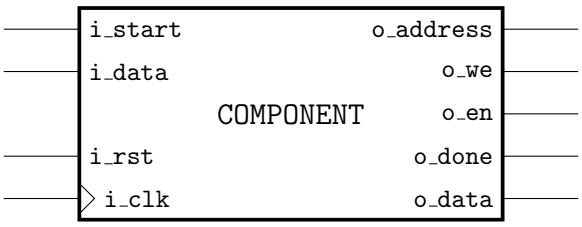


Figura 1: Schema del componente realizzato.

1.3 Descrizione della memoria e dell'interazione con il componente

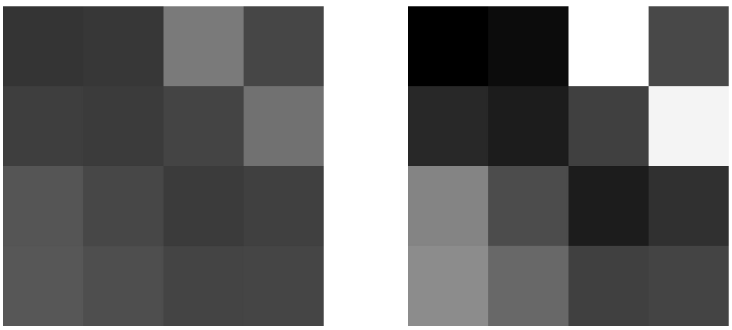
Il modulo implementato dovrà dialogare in lettura e scrittura con una RAM, indirizzata al byte.

In particolare, l'algoritmo di equalizzazione sarà applicato a immagini pre-salvate in memoria, la cui grandezza effettiva (in pixel) sarà specificata dal prodotto tra le celle a indirizzo `0` e `1` della RAM, contenenti rispettivamente il numero di colonne `n_col` e di righe `n_row` dell'immagine, entrambi di dimensione 8 bit.
Nei byte successivi, dall'indirizzo `2` all'indirizzo $\texttt{n\_col} \cdot \texttt{n\_row} + 1$, sarà contenuta, pixel per pixel, sequenzialmente e in modo contiguo, l'immagine di cui è richiesta la trasformazione.
Infine, l'immagine ottenuta dal processo di equalizzazione svolto dal componente verrà salvata in memoria dall'indirizzo $\texttt{n\_col} \cdot \texttt{n\_row} + 2$ all'indirizzo $2 \cdot \texttt{n\_col} \cdot \texttt{n\_row} + 1$.

Tabella 1: Schema generale del contenuto della memoria dopo un'elaborazione.

`n_col`	addr. 0
`n_row`	addr. 1
`Primo pixel immagine da elaborare`	addr. 2
⋮	⋮
`Ultimo pixel immagine da elaborare`	addr. $\texttt{n\_col} \cdot \texttt{n\_row} + 1$
`Primo pixel immagine elaborata`	addr. $\texttt{n\_col} \cdot \texttt{n\_row} + 2$
⋮	⋮
`Ultimo pixel immagine elaborata`	addr. $2 \cdot \texttt{n\_col} \cdot \texttt{n\_row} + 1$

1.4 Esempio di funzionamento



(a) Immagine sorgente (b) Immagine equalizzata

Figura 2: Esempio di trasformazione compiuta dal componente richiesto.

È evidente come l'immagine (b) presenti un contrasto maggiore rispetto alla (a). Questo è dato dall'ampliamento del range di valori assunti dai pixel dell'immagine (b), come evidenziato anche dal contenuto della memoria:

Tabella 2: Contenuto della memoria riferito all'elaborazione dell'immagine d'esempio (a).

addr.	data	addr.	data
0	4	17	69
1	4	18	0
2	52	19	12
3	55	20	255
4	122	21	72
5	70	22	40
6	62	23	28
7	59	24	64
8	68	25	244
9	113	26	132
10	85	27	76
11	71	28	28
12	59	29	48
13	64	30	140
14	87	31	104
15	78	32	64
16	68	33	68

- Immagine sorgente, `addr. 2 - 17`: i pixel assumono valori da `52` a `122`;
- Immagine equalizzata, `addr. 18 - 33`: i pixel assumono valori da `0` a `255`.

Non avendo ancora affrontato appieno il processo svolto dal componente, si ritiene opportuno non approfondire in questa sezione l'insieme di passaggi che permettono la trasformazione dell'immagine riportata nell'esempio, che verranno ripresi in seguito.

Si rende disponibile un *TestBench* che replica l'elaborazione dell'immagine d'esempio a questo link.

2 Design del componente

Si è scelto di descrivere un modulo *single-process* tramite architettura *behavioral* (comportamentale) in linguaggio VHDL. Questo ha determinato la necessità di definire un algoritmo adeguato allo svolgimento dell'operazione richiesta al componente, che può essere schematizzato secondo i seguenti passaggi chiave:

1. Lettura `n_col` e `n_row`;
2. Calcolo dimensione dell'immagine;
3. Ciclo sui pixel dell'immagine sorgente per individuare tra di essi i valori di massimo e minimo;
4. Calcolo dei valori necessari per l'elaborazione dell'immagine;
5. Ciclo sui pixel dell'immagine sorgente per calcolare e salvare in memoria gli effettivi valori, pixel per pixel, dell'immagine equalizzata.

Il modulo prodotto opera quindi su una macchina a stati finiti che realizza l'algoritmo sviluppato.

2.1 Macchina a stati finiti

L'FSM schematizzata è composta da 10 stati, suddivisibili in 4 gruppi principali descritti di seguito.

2.1.1 Stati ausiliari

Gruppo di stati che realizza: inizio e fine del processo, richiesta di lettura e attesa della memoria.

i. `WT_RST` **- wait reset**: stato di attesa del segnale `i_rst`;

ii. `WT_STR` **- wait start**: stato di attesa del segnale di `i_start`.
In qualsiasi momento dell'elaborazione, se il segnale `i_rst` è rilevato alto[1], anche non in corrispondenza di `i_clk`, la macchina viene riportata in questo stato, tornando in attesa di un nuovo segnale di inizio elaborazione.

Al verificarsi della condizione `i_start = 1` vengono inizializzati tutti i valori necessari al processo, prima di passare allo stato successivo. Di particolare importanza per l'algoritmo sviluppato è il segnale `count` (inizialmente `0`), che indica l'indirizzo a cui sarà effettuata l'operazione di read alla successiva richiesta di lettura del componente;

iii. `RD_REQ` **- read request**: stato di abilitazione della memoria in lettura. Viene predisposto su `o_address` l'indirizzo della RAM che deve essere letto;

iv. `WT_MEM` **- wait memory**: stato di attesa della memoria che permette al valore richiesto in `RD_REQ` di essere correttamente riportato sul segnale `i_data` al ciclo di clock successivo.

È un nodo decisivo per l'FSM: viene costantemente rivisitato nei cicli di lettura dei pixel dell'immagine ed è responsabile del corretto instradamento del processo, grazie a condizioni su `count` e `shift_value`. Si occupa inoltre dell'aggiornamento della variabile `count` stessa, e quindi della corretta gestione del successivo dato letto in memoria;

v. `DONE` **- done**: stato in cui `o_done` viene posto a `'1'` per segnalare la fine dell'elaborazione. A questo punto, si attende un valore di `i_start` basso per tornare in `WT_STR` e poter cominciare il processo di equalizzazione di una nuova immagine;

[1] In caso di reset si è supposto che il segnale `i_start` venga riportato basso per il periodo in cui il `i_rst` è alto.

2.1.2 Calcolo dimensioni dell'immagine

Gruppo di stati che permette il calcolo della dimensione effettiva dell'immagine da elaborare.

vi. `RD_COL` - **read column**: stato in cui il valore `n_col` relativo all'immagine, pronto su `i_data`, è salvato su una variabile temporanea per essere utilizzato in seguito;

vii. `RD_ROW` - **read row**: stato in cui `n_row`, pronto su `i_data`, viene moltiplicato con il valore `n_col` salvato precedentemente per calcolare la dimensione effettiva dell'immagine e determinare se essa è adatta ($\texttt{n\_col} \cdot \texttt{n\_row} > 0$) o meno per il proseguimento dell'esecuzione;

2.1.3 Ricerca dei valori di massimo e minimo dell'immagine

Gruppo di stati che permette di individuare i valori minimo e massimo (`min` e `max` nel codice), necessari per l'effettiva elaborazione dell'immagine, tra quelli dei pixel dell'immagine sorgente.

La ricerca è svolta tramite un ciclo sui nodi `RD_REQ`, `WT_MEM` e `CMP_DT` dell'FSM. Come già menzionato precedentemente, è `WT_MEM` a occuparsi del corretto aggiornamento della variabile `count`, e quindi della lettura sequenziale dei pixel durante il ciclo.

viii. `CMP_DT` - **compare data**: stato in cui il valore del pixel dell'immagine relativo all'iterazione corrente, pronto sul segnale `i_data`, viene confrontato con le variabili contenenti il minimo e massimo stabiliti fino a questa iterazione del ciclo di ricerca, aggiornandole se necessario.

   Se la condizione di termine della ricerca ($\texttt{count} \leq \texttt{n\_col} \cdot \texttt{n\_row} + 2$) si verifica, si riporta `count` pari all'indirizzo del primo pixel dell'immagine sorgente, ovvero `2`, per poi procedere alla fase di effettiva equalizzazione. Se invece non si è ancora scandagliata l'intera immagine in memoria, si procede nel ciclo al pixel successivo;

2.1.4 Elaborazione dell'immagine

Gruppo di stati che svolge l'effettiva elaborazione, pixel per pixel, dell'immagine da trasformare, tramite specifici valori calcolati in `PREP_EL` e un ciclo sui nodi `RD_REQ`, `WT_MEM` e `EL_DATA` della macchina a stati finiti.

ix. `PREP_EL` - **prepare elaboration**: stato in cui vengono stabiliti, per mezzo dei dati ottenuti negli stati precedenti, il `delta_value` e lo `shift_level` relativi all'immagine da elaborare, necessari per il proseguimento del processo;

x. `EL_DATA` - **elaborate data**: stato in cui si svolge l'elaborazione effettiva del pixel dell'immagine relativo all'iterazione corrente. In particolare:

   a. Si abilita in scrittura la memoria, ponendo in `o_address` il valore dell'indirizzo di destinazione per la scrittura: $\texttt{count} - 1 + \texttt{n\_col} \cdot \texttt{n\_row}$[2];

   b. Il valore del pixel dell'immagine originale, disponibile su `i_data`, è utilizzato per calcolare quello del rispettivo pixel nell'immagine trasformata, il quale viene posto in `o_data` per essere scritto in memoria;

   c. Se la condizione di termine dell'elaborazione dell'immagine ($\texttt{count} \leq \texttt{n\_col} \cdot \texttt{n\_row} + 2$) si verifica, si passa a `DONE`, altrimenti si procede con l'equalizzazione del pixel successivo.

[2] Essendo `count` già stato incrementato nello scorso passaggio dallo stato `WT_MEM`, il pixel elaborato ad ogni iterazione è contenuto nell'indirizzo $\texttt{count} - 1$ della memoria.

2.1.5 Diagramma della macchina a stati finiti

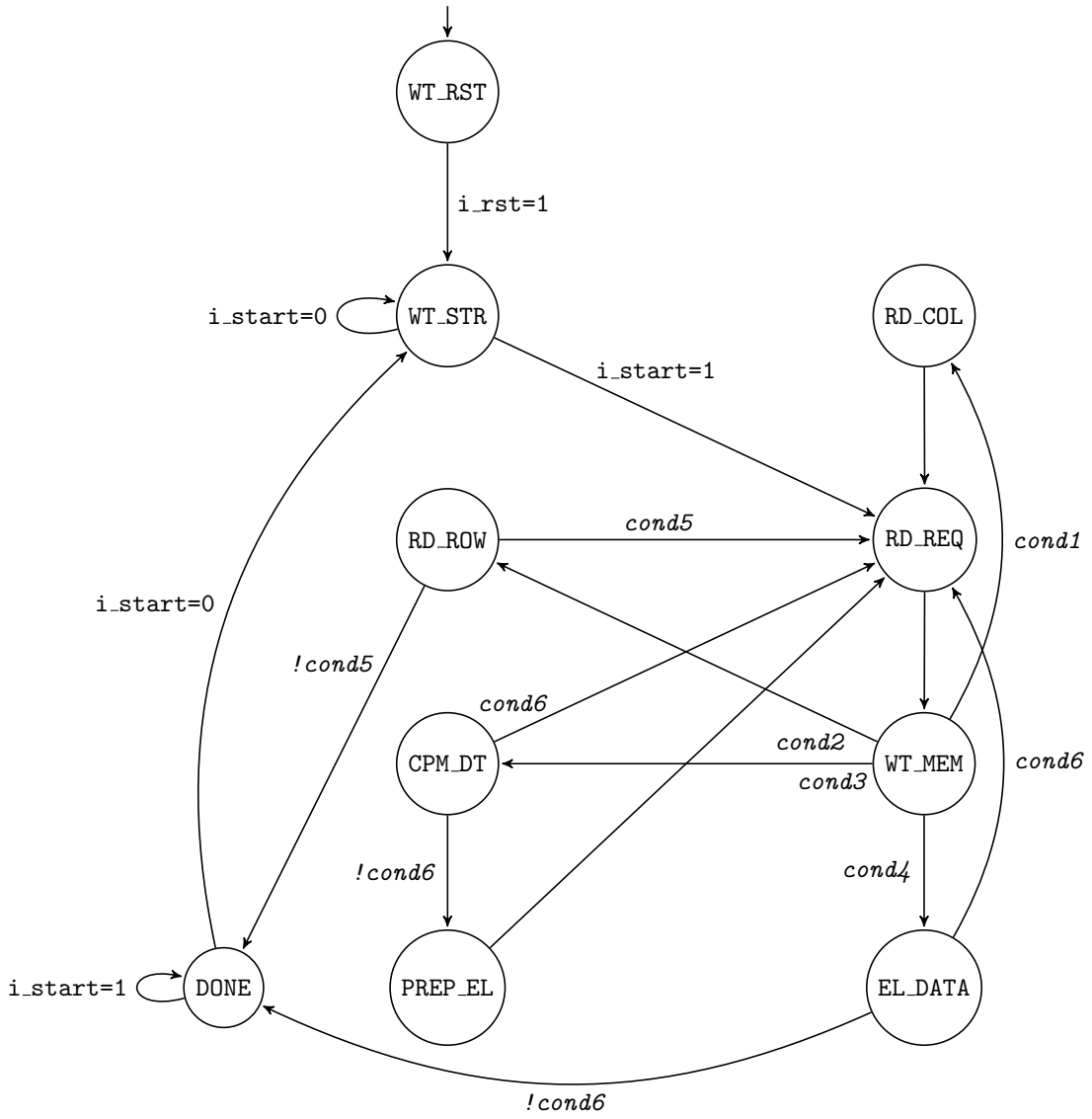


Figura 3: Diagramma della macchina a stati finiti.

Si noti che in Figura 3 si sono utilizzate le seguenti abbreviazioni:

cond1	count = 0
cond2	count = 1
cond3	shift_level = 9 (non ancora calcolato)
cond4	*!cond1* $\wedge$ *!cond2* $\wedge$ *!cond3*
cond5	n_col $\cdot$ n_row $>$ 0
cond6	count $\leq$ n_col $\cdot$ n_row + 2

Da ultimo, si ricorda che per ogni stato dell'FSM è presente un arco uscente implicito diretto verso lo stato `WT_STR`, che simboleggia la possibilità di interrompere in qualsiasi momento l'elaborazione dell'immagine corrente, tramite un segnale `i_rst = 1`.

2.2 Approfondimento sull'equalizzazione dell'immagine

La manipolazione del contrasto dell'immagine si fonda su 4 espressioni fondamentali. Si noti che le prime due sono valutate una sola volta per ogni immagine nello stato `PREP_EL`, mentre le restanti sono determinate ad ogni iterazione del ciclo di elaborazione nello stato `EL_DATA`.

- `delta_value` = `max` − `min`

  `delta_value` rappresenta la differenza tra il pixel più chiaro (valore maggiore, `max`) e il più scuro (valore minore, `min`) dell'immagine;

- `shift_value` = (8 − `floor`[3](log$_2$(`delta_value` + 1)))

  `shift_value` determina il numero di shift a sinistra da applicare al risultato della differenza tra il pixel considerato nell'iterazione corrente e il pixel di valore minore dell'immagine;

Tabella 3: Esempi di valori di `delta_value`, `floor(x)` e `shift_level` possibili per la codifica delle immagini in scala di grigi a 256 livelli.

`delta_value`	`floor(x)`[4]	`shift_level`
0	0	8
1	1	7
2	1	7
3	2	6
⋮	⋮	⋮
6	2	6
7	3	5
8	3	5
⋮	⋮	⋮
14	3	5
15	4	4
16	4	4
⋮	⋮	⋮
29	4	4

`delta_value`	`floor(x)`[4]	`shift_level`
30	4	4
31	5	3
32	5	3
⋮	⋮	⋮
62	5	3
63	6	2
64	6	2
⋮	⋮	⋮
126	6	2
127	7	1
128	7	1
⋮	⋮	⋮
254	7	1
255	8	0

- `temp_pixel` = (`current_pixel` − `min`) ≪ `shift_level`

  `temp_pixel` rappresenta il possibile valore da attribuire nell'immagine finale al pixel valutato in questa iterazione del ciclo di elaborazione. Si noti che il suo valore potrebbe superare il limite massimo di `255` imposto dalla codifica in scala di grigi a 256 livelli. Per questo motivo non può essere utilizzato *"as-is"*;

- `new_pixel` = `min(255, temp_pixel)`

  `new_pixel` è pari al minimo tra `255` e `temp_pixel` e rappresenta l'effettivo valore che verrà scritto sulla RAM per il pixel considerato in questa iterazione.

Riprendendo quindi l'esempio proposto nella Sezione 1.4:

- `delta_value` = `max` − `min` = 122 − 52 = 70
- `shift_value` = (8 − `floor`(log$_2$(`delta_value` + 1))) = 8 − `floor(6.149)` = 2

[3] La funzione `floor(x)` svolge l'arrotondamento per difetto del valore x fornitogli come argomento.
[4] Dove: `x` = log$_2$(`delta_value` + 1).

Iterando quindi sui pixel dell'immagine sorgente:

current_pixel	temp_pixel	new_pixel
52	0	0
55	12	12
122	280	255
70	72	72
⋮	⋮	⋮

2.3 Scelte progettuali e ottimizzazioni

Si è scelto di progettare un componente sensibile al clock su *rising_edge*.

Nell'implementazione dell'algoritmo si sottolinea la scelta di mantenere l'operazione di moltiplicazione per il calcolo della dimensione effettiva dell'immagine da processare. Nonostante il prodotto sia un operatore pesante rispetto alla semplice somma o differenza, lavorando su segnali a 8 bit la sintesi è capace di gestirlo in modo efficace tramite l'inserimento di alcuni DSP[5]. Si noti inoltre che l'operazione di moltiplicazione ricorre una sola volta nell'intera elaborazione di ogni singola immagine, e non risulta quindi particolarmente rilevante rispetto all'intero processo. Questo è stato verificato attraverso lo sviluppo simultaneo rispetto alla versione proposta di una ulteriore implementazione che non facesse uso di nessun prodotto all'interno del processo. In quest'ultima non si sono osservati vantaggi significativi in termini di tempo e area di sintesi, a scapito della leggibilità del codice stesso.

Da ultimo, si noti che `shift_level`, assumendo valore intero compreso tra `0` e `8`, è facilmente ricavabile tramite controlli di soglia. Questo permette di evitare nel codice l'effettivo sviluppo del logaritmo indicato nella relativa formula matematica, enunciata alla Sezione 2.2.
Di seguito, si riporta il blocco IF/ELSE utilizzato per il calcolo dello `shift_value` corrispondente a un dato `delta_value`:

```
-- pongo delta_value in temp_integer
temp_integer := TO_INTEGER(unsigned (max)) - TO_INTEGER(unsigned (min));
IF temp_integer = 0 THEN -- switch case per determinare shift_level
    shift_level <= 8;
ELSIF temp_integer > 0 AND temp_integer < 3 THEN
    shift_level <= 7;
ELSIF temp_integer > 2 AND temp_integer < 7 THEN
    shift_level <= 6;
ELSIF temp_integer > 6 AND temp_integer < 15 THEN
    shift_level <= 5;
ELSIF temp_integer > 14 AND temp_integer < 31 THEN
    shift_level <= 4;
ELSIF temp_integer > 30 AND temp_integer < 63 THEN
    shift_level <= 3;
ELSIF temp_integer > 62 AND temp_integer < 127 THEN
    shift_level <= 2;
ELSIF temp_integer > 126 AND temp_integer < 255 THEN
    shift_level <= 1;
ELSE
    shift_level <= 0;
END IF;
```

[5] DSP (digital signal processor) è un processore dedicato e ottimizzato per eseguire in maniera estremamente efficiente sequenze di istruzioni ricorrenti (come ad esempio somme, moltiplicazioni e traslazioni) nell'elaborazione di segnali digitali.

3 Testing e risultati sperimentali

3.1 Casi di test

Il corretto funzionamento del componente sviluppato è stato verificato tramite numerosi *TestBench*.

In particolare, si è scelto di concentrare l'attenzione su diversi casi critici possibili durante l'esecuzione e sul corretto calcolo di tutti i valori utilizzati. Di seguito una breve lista di condizioni e test più significativi:

- Corretto calcolo e utilizzo di tutti i possibili `shift_value`;
- Condizione particolare: `n_col` $\cdot$ `n_row` $= 0$[6];
- Casi limite di dimensione dell'immagine: `1x1` e `128x128` pixel;
- Caso di reset dell'elaborazione;
- Caso di reset dell'elaborazione seguito da un cambio di immagine in memoria;
- Corretto rapporto tra i segnali `i_rst`, `i_start` e `o_done` durante l'esecuzione.

Al fine di verificare la correttezza degli ultimi 3 punti in elenco, si è rivelata particolarmente utile l'analisi grafica dei segnali di input/output del modulo.

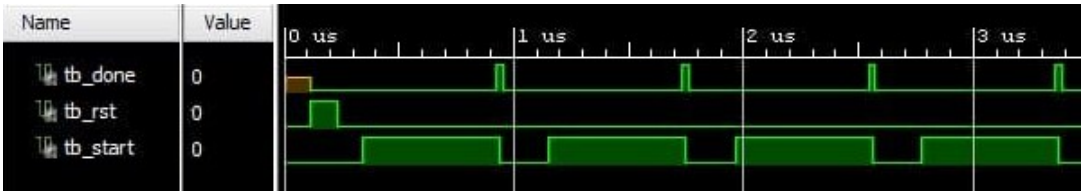


Figura 4: Analisi dei segnali `i_start`, `i_rst` e `o_done` nell'elaborazione di 4 immagini consecutive.

Si sono utilizzati diversi *TestBench* con caratteristiche differenti e dimensioni variabili dalla singola alle `10000` immagini (*`TB10K`*), redatti manualmente (da colleghi e da noi) o auto-generati tramite uno script python appositamente creato.

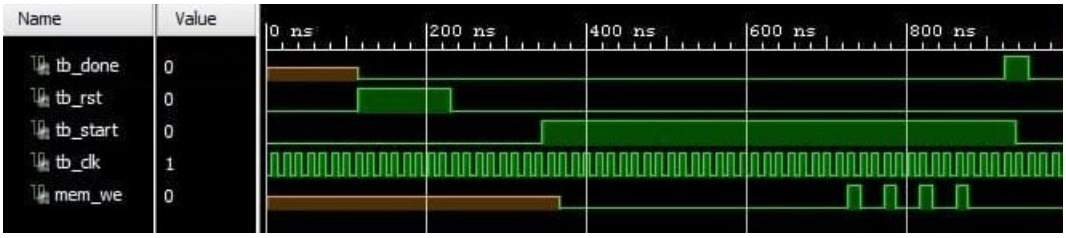


Figura 5: Analisi dei segnali di simulazione del *TestBench* *`TB2x2`* fornito con la specifica di progetto.

[6] `n_col` $= 0 \vee$ `n_row` $= 0$.

3.2 Risultati sperimentali

Il report di sintesi ha evidenziato l'utilizzo nell'area del modulo sintetizzato dei seguenti componenti:

Tabella 4: Risultati della tabella *"utilization"* generata dalla simulazione di *post-synthesis*.

Risorsa	Stima	Utilizzo %[7]
`LUT`	180	0.44
`FF`	81	0.10
`DSP`	1	0.42
`IO`	38	12.67
`BUFG`	1	3.13

Come precedentemente accennato, è presente nel componente finale 1 `DSP`, dovuto alla moltiplicazione volutamente mantenuta nel processo. Si noti che il numero dei `DSP` utilizzati è trascurabile rispetto al numero di `DSP` generalmente disponibili in una scheda `FPGA`.

3.3 Risultati di simulazione

Per tutti i casi di test e *TestBench* utilizzati, sono state svolte con successo le simulazioni richieste dalle specifiche di progetto, di cui si riportano come riferimento di tempi di esecuzione relativi al *TestBench* `TB10K`:

- Simulazione *behavioral*: `459410050000ps`
- Simulazione *post-synthesis functional*: `461409950100ps`

Si è inoltre verificato che il componente progettato supera le seguenti simulazioni non richieste:

- Simulazione *post-synthesis timing*;
- Simulazione *post-implementation functional*;
- Simulazione *post-implementation timing*.

4 Conclusioni

Si ritiene che l'architettura rispecchi a pieno le specifiche di progetto assegnateci. Inoltre, si ritiene di aver ampliato le nostre competenze riguardo il processo di progettazione e il funzionamento di un componente dalle caratteristiche simili a quello da noi proposto.

[7] Percentuali riferite alla scheda `FPGA` utilizzata: `xc7k70tfbv676-1`.