Progetto di Reti Logiche

Barnabò Alberto (10560076 – 887951)

20/04/2021

INTRODUZIONE E SPECIFICA

Il progetto consiste nell'implementazione di un componente hardware mediante il linguaggio VHDL che realizzi l'equalizzazione di un'immagine in bianco e nero che gli viene data in ingresso. L'algoritmo di equalizzazione, fornito in una sua versione semplificata rispetto a quella standard, procede nel seguente modo:

- Calcola la dimensione dell'immagine, della quale vengono forniti numero di colonne e di righe in ingresso;
- Passa in rassegna tutti i pixel (ciascuno di dimensione un byte) e ne trova il valore massimo e minimo;
- Calcola il valore delle due variabili come segue:

  DELTA_VALUE = MAX_PIXEL_VALUE – MIN_PIXEL_VALUE

  SHIFT_LEVEL = (8 – FLOOR(LOG2(DELTA_VALUE + 1)))
- Rilegge da capo il valore di ognuno dei pixel in ingresso, e per ciascuno di essi calcola il suo nuovo valore, che sarà quello dell'immagine equalizzata, in questo modo:

  TEMP_PIXEL=(CURRENT_PIXEL_VALUE – MIN_PIXEL_VALUE)<<SHIFT_LEVEL

  NEW_PIXEL_VALUE = MIN (255, TEMP_PIXEL)

Il modulo leggerà dalla memoria l'immagine in modo sequenziale. In particolare, il byte in posizione 0 si riferisce alla dimensione di colonna, mentre il byte in posizione 1 a quella di riga. Conseguentemente, i byte successivi rappresenteranno il valore dei pixel dell'immagine da elaborare. I pixel risultanti dell'equalizzazione saranno scritti a partire dal byte successivo all'ultimo in ingresso. Riportato a seguito c'è un esempio di funzionamento, tratto dalla specifica di progetto, che ho riscritto in forma "tabellare", in modo da rendere chiari i passi che segue il componente per arrivare al risultato, inizialmente con una rappresentazione "ad alto livello". Successivamente, illustrerò le scelte progettuali e mostrerò più nel dettaglio le sue caratteristiche.

IN:

76	131	109	89
46	121	62	59
46	77	68	94

OUT:

120	255	252	172
0	255	64	52
0	124	88	192

MAX_PIXEL_VALUE = 131

MIN_PIXEL_VALUE = 46

DELTA_VALUE = MAX_PIXEL_VALUE - MIN_PIXEL_VALUE

= 131 - 46 = 85

SHIFT_LEVEL = (8 - FLOOR (LOG_2(DELTA_VALUE+1)))

= 8 - 6 = 2

Per ogni pixel:

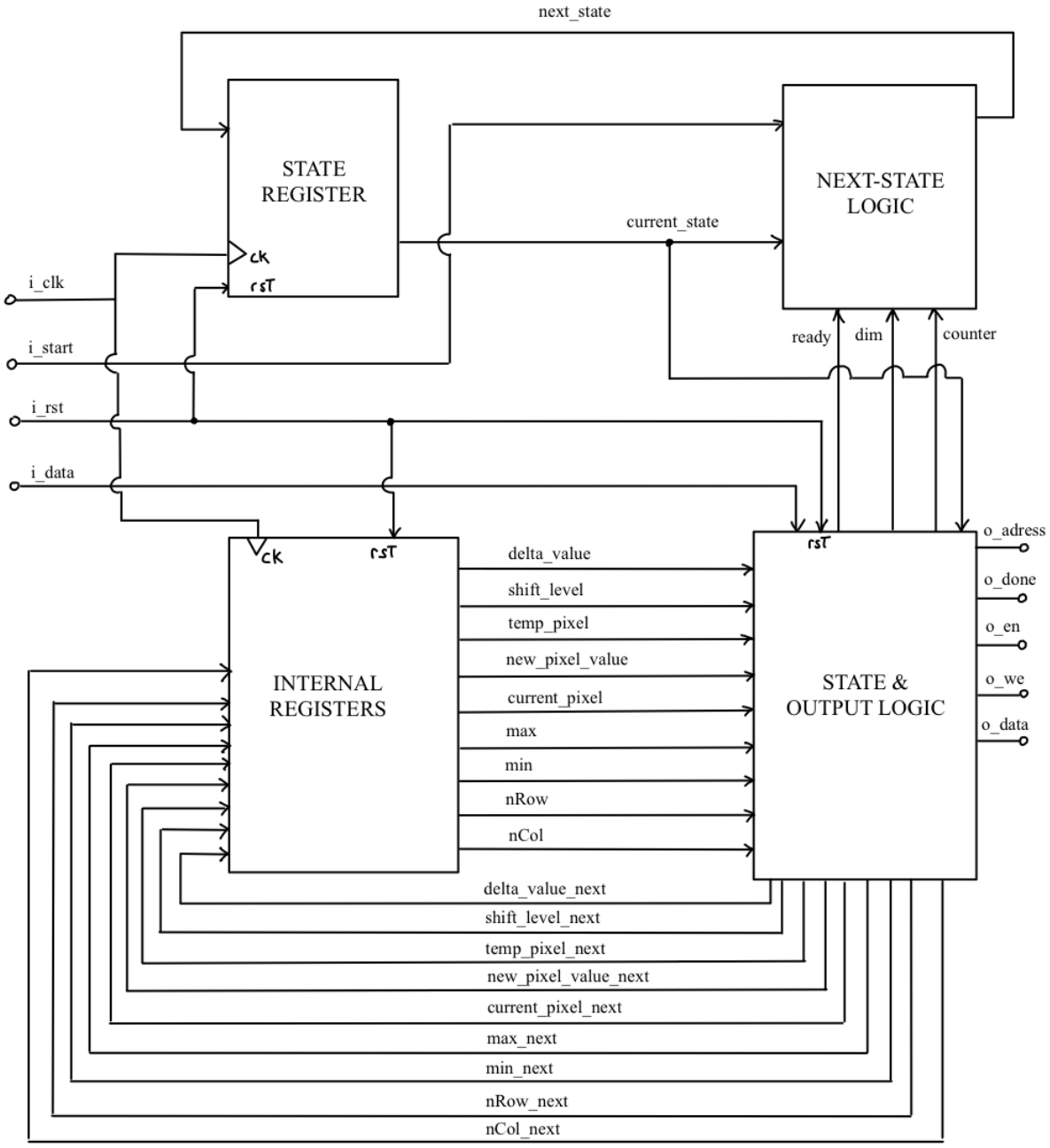
TEMP_PIXEL = (CURRENT_PIXEL_VALUE - MIN_PIXEL_VALUE)<< SHIFT_LEVEL

Per il primo pixel : TMP_PIXEL = $(76 - 46) \cdot 2^2 = 120$

NEW_PIXEL_VALUE = MIN (255, TMP_PIXEL_VALUE)

ARCHITETTURA

Vediamo ora le scelte progettuali effettuate in maniera più approfondita. Ho deciso di descrivere il componente con tre processi, secondo il seguente schema:



Il primo ha la funzione di gestire il fronte di salita del clock e l'arrivo del segnale di reset. La prima funzionalità viene realizzata assegnando ad ogni segnale necessario per la computazione il suo corrispettivo _next nell'istante in cui il clock ha un fronte di salita (rising_edge). Questa scelta mi ha permesso di poter aggiornare liberamente i segnali durante il ciclo di clock evitando eventuali oscillazioni che potrebbero causare malfunzionamenti. La seconda è un semplice assegnamento dei valori di default a tutti i segnali nel momento in cui i_rst salga a '1'.

- Il secondo si occupa di realizzare la funzione "stato prossimo". Il suo scopo cioè è quello di gestire l'evoluzione della macchina e dei suoi stati, che dipendono dallo stato corrente e da alcuni segnali ausiliari definiti da me, come *ready*, *counter*, e *dim*.

- L'ultimo, infine, è il processo che si occupa di svolgere tutti i calcoli necessari per ogni stato, settare i segnali di uscita del componente e scrivere in memoria il risultato.

Interfaccia

Il componente richiesto, ai morsetti, si presenta così:

```
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:

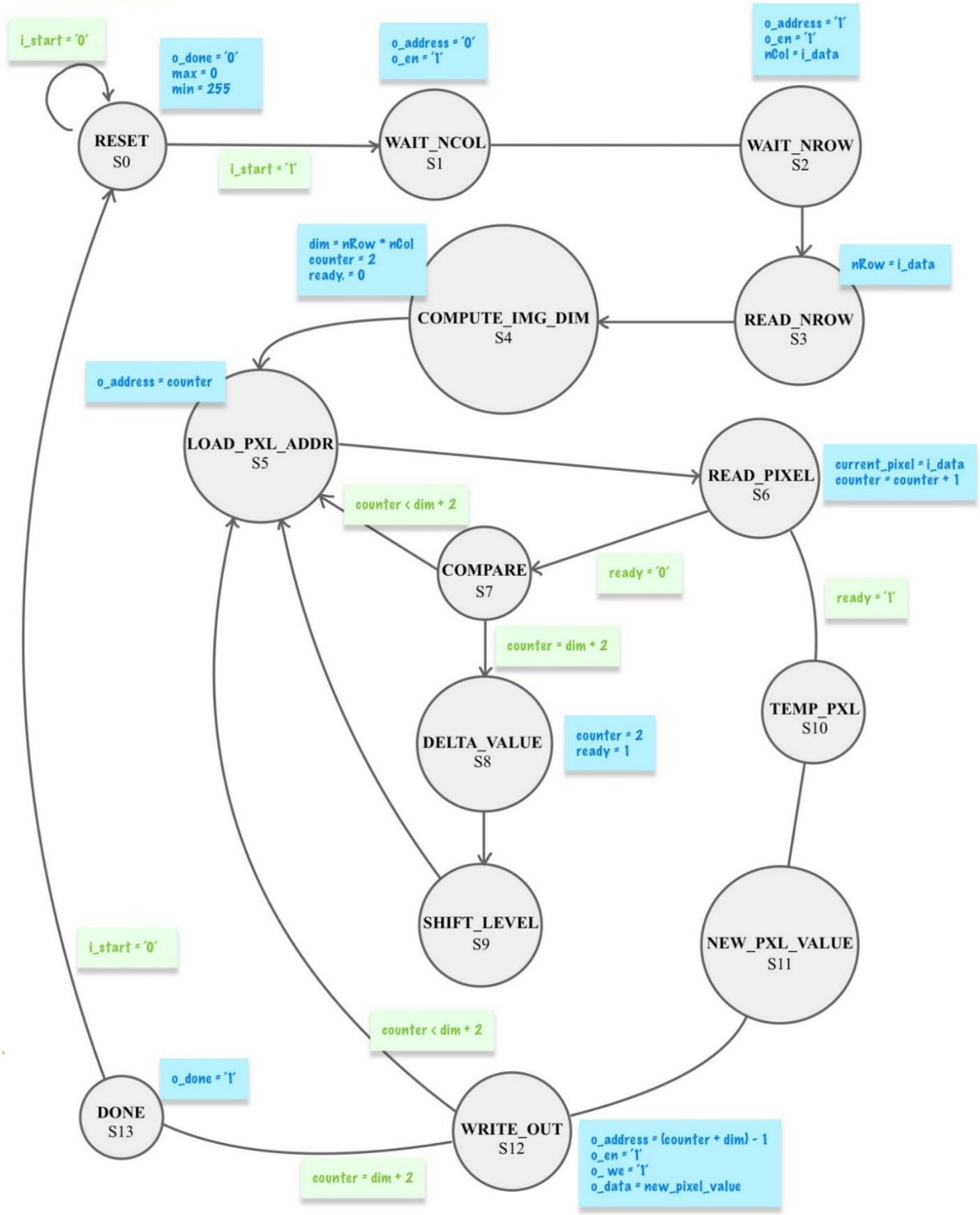
- i_clk è il segnale di CLOCK in ingresso generato dal TestBench;
- i_rst è il segnale di RESET che inizializza la macchina pronta per ricevere il primo segnale di START;
- i_start è il segnale di START generato dal Test Bench;
- i_data è il segnale (vettore) che arriva dalla memoria in seguito ad una 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 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.

Design

La computazione comincia quando il segnale di ingresso di i_start viene portato a 1. Il componente si sposta dallo stato di RESET (S0) al suo successivo, e comincia così l'elaborazione. Al termine, porta il segnale o_done a 1 e il componente può ricomciare solo in seguito ad un nuovo segnale di start.

Ho disegnato, per realizzare il componente, una FSM che conta 14 stati, e poi ho trasposto il suo funzionamento su Vivado, dove ho potuto testare il suo funzionamento. Riporto a seguito il nome di ogni stato e la sua funzione, con annesso uno schema per rendere più chiara la trattazione:

Macchina a stati



- **RESET** (S0)

  Stato iniziale in cui si rimane finchè *i_start* è basso. Si occupa di inizializzare i valori max e min rispettivamente a 0 e 255. Quando *i_rst* viene alzato, si torna in questo stato.

- **WAIT_NCOL** (S1)

  In questo stato si legge dalla memoria il numero di colonne dell'immagine, che si trova all'indirizzo 0 della memoria. Si prosegue poi verso S2 senza condizioni.

- **WAIT_NROW** (S2)

  Solo una volta arrivati in questo stato il valore del numero di colonna è disponibile su *i_data*. Quindi, per prima cosa lo salviamo con un segnale chiamato nCol. All'interno dello stato, possiamo già chiedere il numero di righe alla memoria (impostando quindi *o_address* all'indirizzo 1), che sarà disponbile nel prossimo stato. Da questo stato si salta ad S3 senza condizioni.

- **READ_NROW** (S3)

  A questo punto la memoria ci avrà restituito il valore del numero di righe. Qui lo salviamo in nRow e andiamo allo stato successivo (S4) senza condizioni.

- **COMPUTE_IMG_DIM** (S4)

Qui ci occupiamo per prima cosa del calcolo della dimensione dell'immagine. Inoltre, inizializziamo alcuni segnali che servono per scandire i cicli che faremo successivamente. In particolare, utilizzeremo *counter* e *ready*. Il primo si occupa di tenere il conto dell'indirizzo di memoria dal quale vogliamo scrivere o leggere. A questo punto del calcolo, dobbiamo prima fare un ciclo che legge tutti i byte in ingresso e trova massimo e minimo. Settiamo quindi *counter* a 2 perchè è l'indirizzo nel quale si trova il primo pixel, mentre *ready* a 0 sta ad indicare che non abbiamo ancora trovato massimi e minimo, e non siamo quindi pronti per la fase successiva.

- **LOAD_PXL_ADDR** (S5)

  S5 si occupa di chiedere alla memoria il valore di un pixel in ingresso. Verrà usato sia quando leggeremo i byte per trovare massimo e minimo, sia quando dovremo rileggerli tutti per poterne calcolare il nuovo valore.

- **READ_PIXEL** (S6)

  Salva il valore del pixel richiesto in S5 in un segnale ausiliario chiamato *current_pixel*, e incrementa *counter*. Se *ready* è ancora a 0, vuol dire che massimo e minimo non sono pronti, e quindi lascia il controllo ad S7. Altrimenti, siamo pronti a calcolare per ogni pixel il suo nuovo valore. In questo secondo caso il controllo passa ad S10.

- **COMPARE** (S7)

  Calcola massimo e minimo con un semplice algoritmo. Per ogni pixel che riceve, controlla se è maggiore del massimo attuale e/o minore del minimo. In caso affermativo, aggiorna il valore corretto. Al momento del passaggio di stato, prende una decisione: se non ci sono più altri pixel da

leggere (e quindi *counter = dim + 2*), può andare verso S8 e calcolare il delta_value. In caso contrario, torna ad S5 per leggere un nuovo pixel.

- **DELTA_VALUE** (S8)

  Si arriva in questo stato solo una volta trovati il massimo e il minimo tra tutti i pixel. Arrivati qui si può procedere al calcolo di *delta_value* con la formula indicata della specifica (è una semplice sottrazione tra *max* e *min*). E' qui che inoltre resettiamo i valori di *counter* e *ready* per prepararci al successivo ciclo di lettura di tutti valori e quindi al calcolo di *new_pixel_value.*

- **SHIFT_LEVEL** (S9)

  E' l'ultimo passo prima del secondo ciclo. La formula per il calcolo di *shift_level* comprende un logaritmo, ma il suo calcolo effettivo si può evitare effettuando dei semplici controlli a soglia che sono facilmente leggibili nel codice fornito.

- **TEMP_PXL** (S10)

  Calcola *temp_pixel* dopo aver letto e memorizzato (negli stati S5 ed S6) *current_pixel_value*. Ho realizzato l'operazione "a mano", cioè concatenando gli zeri necessari a destra e sinistra.

- **NEW_PXL_VALUE** (S11)

  Una volta pronto *temp_pixel*, dobbiamo solo trovare il minimo tra quest'ultimo e il valore 255. Il risultato sarà proprio *new_pixel_value*, che andremo a scrivere in memoria nello stato successivo.

- **WRITE_OUT** (S12)

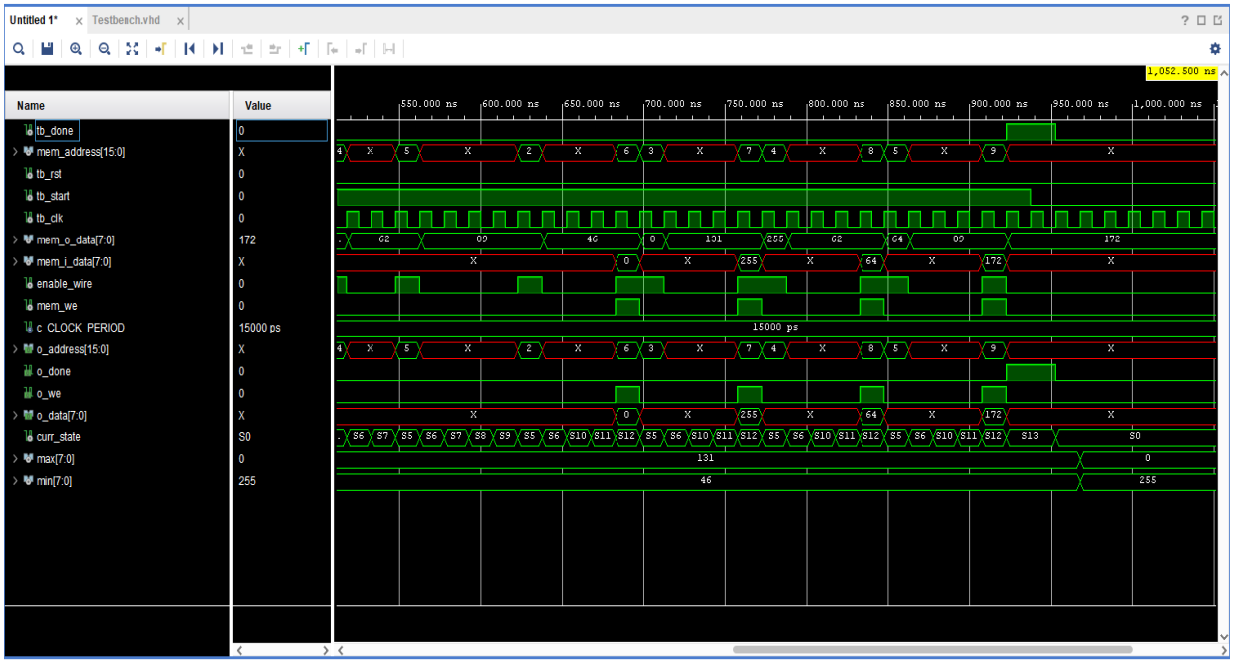
  Stato che si occupa di scrivere in memoria (all'indirizzo *counter + dim - 1*) il risultato ottenuto al punto precedente. Inoltre, proprio come in S7, ci chiediamo se abbiamo già letto tutti i pixel (e quindi *counter = dim + 2*) oppure dobbiamo leggerne ancora. Nel primo caso la computazione finisce e andiamo nello stato di DONE (S13), altrimenti dobbiamo tornare ad S5 e chiedere un nuovo valore in ingresso.

- **DONE** (S13)

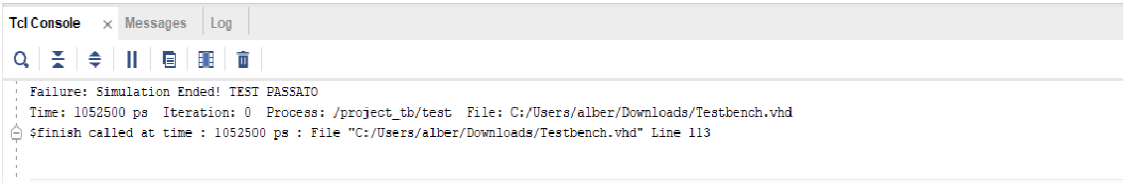
  Stato che si occupa di portare *o_done* a uno, in modo da indicare la fine della computazione. Solo quando il segnale *i_start* tornerà a zero si potrà andare nello stato di RESET e far partire l'elaborazione di una nuova immagine.

RISULTATI SPERIMENTALI

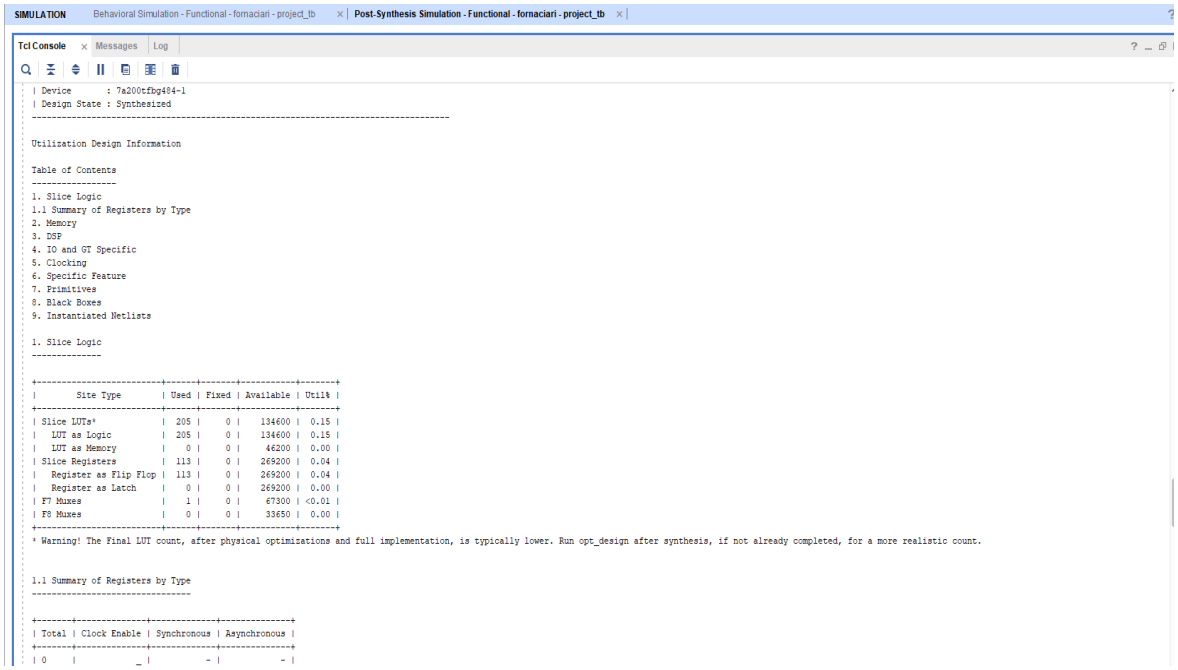
Ho testato il componente in un primo momento utilizzando il test bench fornito dai docenti, disponibile sulla piattaforma BeeP. I risultati sono stati quelli attesi sia in simulazione "Behavioral" sia "Post Synthesis".



Vediamo da questa immagine che i risultati (che per chiarezza ho impostato in notazione decimale) sono quelli attesi. Non sorprendentemente, la console mostra l'esito positivo del test e il tempo impiegato in esecuzione.



Ho poi testato il componente in fase post synthesis, ottenendo sempre l'esito positivo. Inoltre, con il comando *report_utilization* della console, ho controllato che Vivado non avesse inserito latch "parassiti" in fase di sintesi, il che è indice di una buona progettazione del componente. La voce "Registered as latch" è infatti correttamente sul valore 0.



Oltre al test bench fornito dai docenti, ho avuto di modo di testare su ampia scala il componente grazie ad un generatore di test. Questo mi ha permesso di controllare il corretto funzionamento del modulo non solo stressandolo con input di dimensioni considerevoli, ma anche di separare il caso in cui il segnale di reset fosse fornito in ingresso da quello in cui il segnale di reset non veniva mai fornito e il componente doveva gestire esecuzioni multiple successive. I risulati sono sempre stati gli stessi riportati precedentemente per diverse centinaia di test, in entrambe le modalità Behavioral e Post Synthesis.

CONCLUSIONE

Per risolvere un problema di programmazione a livello hardware sono partito dalla progettazione della macchina a stati. Ho pensato a quante operazioni potessi fare in ogni stato, coerentemente con la gestione del clock. Sono poi passato al codice, dove, dopo aver diviso i vari compiti della FSM in processi, ho scritto in VHDL tutte le funzionalità della macchina, prestando molta attenzione alla gestione del fronte di salita del clock e del segnale di reset. Dopo aver testato il componente e aver ottenuto i risultati positivi desiderati, ho dichiarato concluso il lavoro.