Politecnico di Milano

Progetto di Reti Logiche 2020-2021

Juan David Liut Aymar – codice persona: 10607787

Sommario

1. INTRODUZIONE 2
1.1 OBIETTIVO 2
1.2 STRUMENTI USATI 2
1.3 INTERFACCIA DEL COMPONENTE 2
1.4 I DATI E LA MEMORIA 3
2. DESIGN 4
2.1 STATI DELLA MACCHINA 4
2.1.1 START_STATE 4
2.1.2 READ_DIM 4
2.1.3 WAIT_DIM 4
2.1.4 FIND_MAX_MIN 4
2.1.5 WAIT_STATE 4
2.1.6 CONVERSION 5
2.1.7 WRITE_STATE 5
2.1.8 WAIT_END 5
2.2 SCELTE PROGETTUALI 6
3. RISULTATI DEI TEST 7
4. CONCLUSIONE 10
4.1 RISULTATI DELLA SINTESI E DELLA “IMPLEMENTATION” 10
4.2 OTTIMIZZAZIONI 10

1. INTRODUZIONE

1.1 OBIETTIVO

La Prova Finale del corso di Reti Logiche 2020 - 2021 è ispirata al metodo di equalizzazione dell'istogramma di una immagine e chiede allo studente di sviluppare un componente hardware che sia in grado di aumentare il contrasto di una foto in bianco e nero, in cui la scala di grigi è a 256 livelli e l'intervallo dei valori di intensità dei pixel sono molto vicini.

1.2 STRUMENTI USATI

Per l'elaborazione del progetto, sono stati usati i seguenti strumenti:

- XILINX VIVADO WEBPACK: software che permette di fare *Hardware Design*, usando il linguaggio di descrizione hardware VHDL;
- FPGA: xc7a200tfbg484-1 (come suggerito nella specifica): è un dispositivo logico programmabile le cui funzionalità possono essere modificate tramite l'uso di linguaggi come il VHDL;
- TestBench (quelli forniti dal docente sul sito BEEP e quelli creati con un generatore scritto in C).

1.3 INTERFACCIA DEL COMPONENTE

Come da specifica, l'interfaccia implementata per il componente hardware è la seguente:

```
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: segnale di CLOCK in ingresso, generato dal TestBench;
- i_rst: segnale di RESET che inizializza la macchina pronta per ricevere il primo segnale di START;
- i_start: segnale di START generato dal TestBench;
- i_data: segnale (vettore) che arriva dalla memoria dopo una richiesta di lettura;
- o_address: segnale (vettore) di uscita che manda l'indirizzo alla memoria;

- o_done: segnale di uscita che comunica la fine dell'elaborazione e il dato di uscita scritto in memoria;
- o_en: segnale di ENABLE da dover mandare alla memoria per poter comunicare con essa (sia in lettura che in scrittura);
- o_we: segnale di WRITE ENABLE da dover mandare alla memoria (=1) per poterci scrivere. Per leggere da memoria esso deve essere 0;
- o_data: segnale (vettore) di uscita dal componente verso la memoria.

1.4 I DATI E LA MEMORIA

I dati, ciascuno di 8 bit, sono memorizzati in una memoria con indirizzamento al byte, dove ciascun indirizzo può contenere al massimo 16 bit.

Considerando il caso in cui si stia lavorando con una sola immagine, la disposizione dei dati all'interno di una sola unità di memoria è la seguente (Figura 1):

- all'indirizzo 0 è conservato il numero di colonne (N_COL) che compongono la foto;
- all'indirizzo 1 è salvato il numero di righe (N_RIGHE) della foto;
- nelle successive N_COL*N_RIGHE sono conservati i valori dei singoli pixel che dovranno essere convertiti;
- infine, a partire dall'indirizzo 2+(N_COL*N_RIGHE) verranno memorizzati i pixel convertiti, uno alla volta.

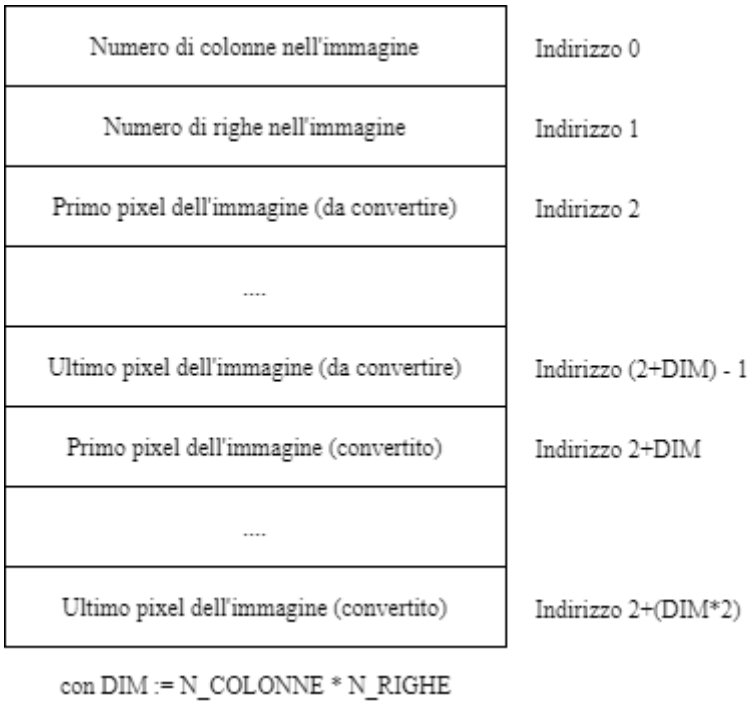

Figura 1: Rappresentazione degli indirizzi in memoria

Nel caso di più immagini conservate in più memorie RAM, la configurazione appena descritta si ripete per ogni RAM in cui è salvata un'immagine.

2. DESIGN

Il modulo è stato progettato come macchina a stati finiti (FSM) perché questa forma di descrizione permette al componente di essere sintetizzabile, oltre a fornire una visione ordinata del processo.

Quando all'ingresso dell'elemento il segnale *i_start* viene portato alto, il processo di conversione ha inizio e lo stato iniziale è *start_state*. Durante la computazione, lo stato corrente si sposta tra tutti gli stati disponibili nella FSM (che vengono elencati in seguito), ma per tutto il tempo, il segnale *o_done* viene mantenuto basso, poiché non si possono avere più conversioni in parallelo, ma solo sequenzialmente. Una volta terminata la procedura, il segnale *o_done* viene alzato e lo stato corrente passa ad uno di attesa (*wait_end*) dove ci rimane finché il segnale di *i_start* non torna basso. Infine, quando questo succede, si ritorna allo *start_state* , *o_done* scende a 0 e *i_start* può tornare alto: questo permette al componente di ricominciare il processo di codifica con l'immagine successiva.

2.1 STATI DELLA MACCHINA

La macchina è composta da 8 stati e quello che segue è una breve descrizione di ciascuno di essi.

2.1.1 START_STATE

Stato iniziale in cui si attende il segnale di *i_start* alto, mentre si inizializzano l'indirizzo da cui cominciare a leggere, i valori di massimo e minimo, *o_done* e la dimensione dell'immagine (nel codice, rappresentato dal segnale di tipo integer *dim* = 1).

2.1.2 READ_DIM

Stato in cui viene moltiplicato il contenuto dei primi due indirizzi (0 e 1) col segnale *dim*. Se quest'ultimo risulta essere maggiore di 0, si passa allo stato successivo altrimenti si termina la computazione perché l'immagine è vuota.

Si vuole far notare che le due moltiplicazioni e il passaggio di stato finale sono stati fatti passando tre volte per questo stato e, tramite il contatore *count*, si son potute distinguere le tre fasi descritte.

2.1.3 WAIT_DIM

Stato in cui si attende la lettura del contenuto di un indirizzo della memoria.

2.1.4 FIND_MAX_MIN

Stato su cui si passa tante volte quanti sono i pixel dell'immagine. Qui si cerca il valore massimo e minimo tra tutti i pixel. Una volta terminata la ricerca, si torna al primo pixel, vengono settati i diversi segnali e il flag *M_m* viene impostato come "*true*", per segnalare che i valori di massimo e minimo sono stati trovati.

2.1.5 WAIT_STATE

Stato in cui si attende la lettura di un indirizzo da memoria. Il passaggio di stato dipende dal flag *M_m* e dal valore di count:

- se il flag è uguale a "*true*" e il contatore è uguale a *dim*, il nuovo stato successivo diventa *wait_end.*
- se il contatore non è uguale a *dim* , allora la macchina passa a *conversion.*
- Infine, se il flag è uguale a "*false*", il processo continua con la lettura dei valori (*read*).

2.1.6 CONVERSION

Stato in cui viene letto e convertito ogni singolo pixel, in base alla versione semplificata del metodo dell'equalizzazione dell'istogramma. Seppur scritte in modo diverso nel codice VHDL per questione di ordine, le equazioni usate sono le seguenti:

```
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)
```

Dove `MAX_PIXEL_VALUE` e `MIN_PIXEL_VALUE`, sono il massimo e minimo valore dei pixel dell'immagine, `CURRENT_PIXEL_VALUE` è il valore del pixel da trasformare, e `NEW_PIXEL_VALUE` è il valore del nuovo pixel.

2.1.7 WRITE_STATE

Stato in cui, una volta terminata la conversione del pixel, questo viene scritto in memoria. Viene poi richiesta la lettura di un indirizzo, i cui dati saranno disponibili nel prossimo stato, ovvero in *wait_state.*

2.1.8 WAIT_END

Stato finale in cui il segnale *o_done* viene tenuto alto finché il modulo non riceve un segnale di *i_start* basso. Allora lo stato successivo tornerà ad essere *start_state* e il processo ricomincerà.

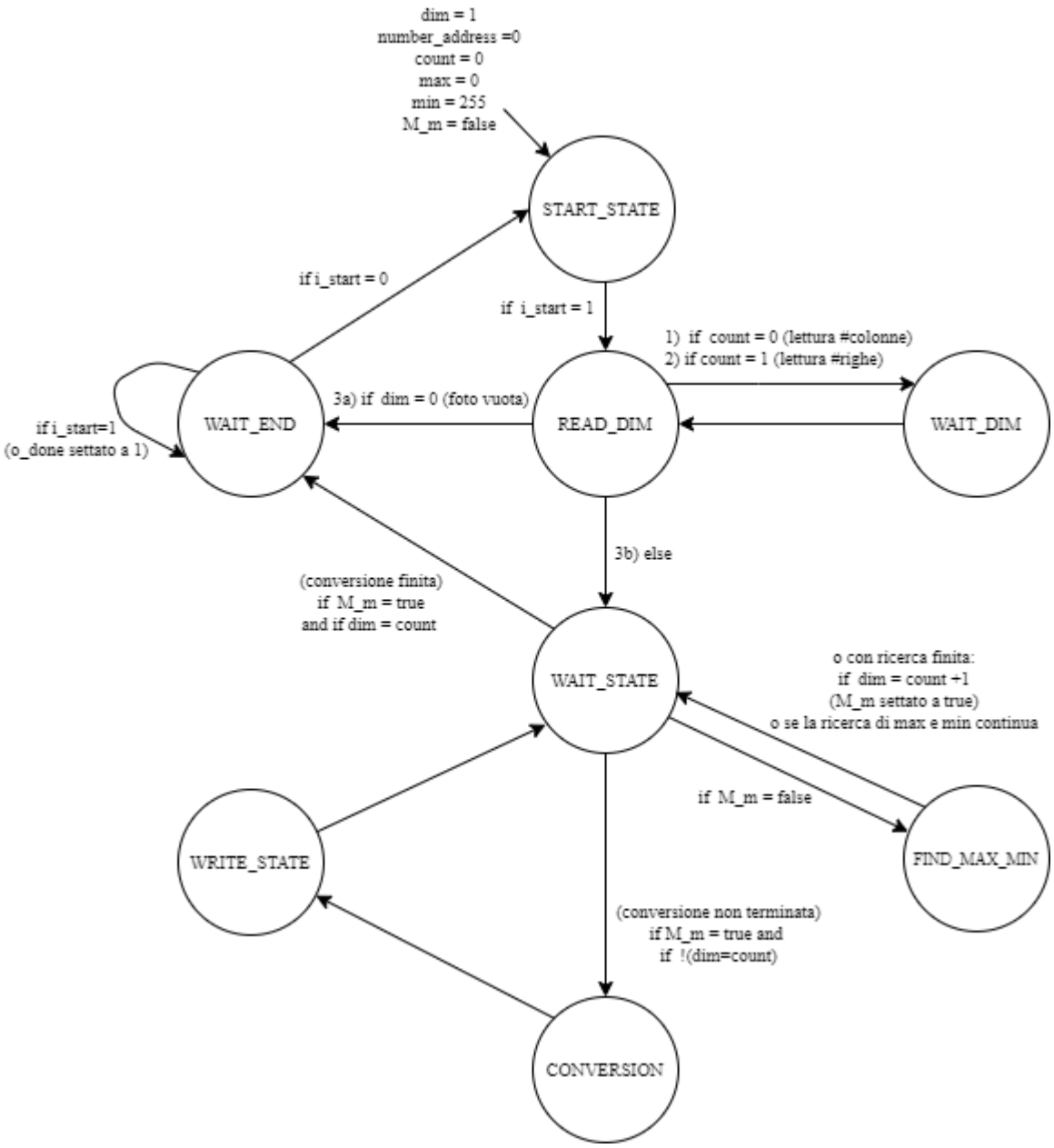

Figura 2: macchina a stati FSM

2.2 SCELTE PROGETTUALI

All'inizio si era optato per descrivere il componente come una sequenza di comandi, come si fa con i comuni linguaggi di programmazione come il C. Il problema che si è poi verificato non era legato al linguaggio VHDL, ma al processo di sintesi: infatti i cicli for non sono ideali se si vuole sintetizzare l'elemento, soprattutto se il loro range dipende da una variabile fissa e non da un numero intero predeterminato.

Allora si è scelto di ideare il componente come una macchina a stati finiti: oltre alla segmentazione dei tre processi principali (lettura dimensione immagine, lettura dei pixel, conversione), resa ancora più semplice e ordinata dalla presenza degli stati, il codice ora risulta sintetizzabile e permette di usare i cicli, ottenuti passando tra le varie configurazioni del componente.

Una volta scelto il tipo di struttura, è stato pensato anche come il componente avrebbe dovuto gestire le risorse durante il processo. A questo proposito sono stati implementati questi due punti importanti:

- per quanto riguarda l'accesso alla memoria, si è optato per ridurlo il più possibile, cosicché i tempi di elaborazione si riducessero ed eventuali problemi di sovra-scrizione venissero evitati: per questo motivo, ad ogni ciclo di clock, il segnale *o_en* viene settato a 0;
- per quanto riguarda l'accesso ai vari stati, sono stati usati dei flag e dei contatori che permettono di distinguere al meglio le diverse casistiche e agire in modi diversi di conseguenza. Un esempio è il flag *M_m* che in base al suo valore, nello stato *wait_state*, permette di gestire con facilità tre casi e quindi tre stati (*read*, *conversion*, *wait_end*).

Un'altra idea fondamentale per il processo è stata applicata allo stato *conversion*: qui il cambiamento è sequenziale, cioè avviene un pixel alla volta e si optato di farlo usando una funzione *shift_lvl*, che tramite il segnale *delta_value* e i controlli a soglia, permette di convertire i pixel, usando il metodo semplificato di equalizzazione dell'istogramma. La scelta di usare una funzione come questa ha i benefici di mantenere ordinato il codice, di ridurlo e di semplificarlo.

Infine, un'altra scelta progettuale che ha aiutato a migliorare la coordinazione tra i singoli sotto-processi è stata la creazione:

- dello stato di attesa *write_state* che permette la scrittura in memoria del nuovo pixel;
- dello stato di attesa *wait_end*, affinché il tempo dato a disposizione per il processo di conversione venisse usato al meglio prima di eventuali segnali come *i_rst=1* o *i_start=0*.

Tutte queste scelte permettono al componente in questione di poter lavorare in diverse situazioni, ma soprattutto con diverse immagini in sequenza, senza incorrere in problemi come la sovra-scrizione.

3. RISULTATI DEI TEST

Per verificare il corretto funzionamento del componente progettato, sono stati usati dei test, alcuni forniti dai docenti ed altri creati con un generatore, in modo da trovare tutti i possibili cammini che l'elemento può dover percorrere durante l'elaborazione di una o più immagini.

Per testare il corretto funzionamento generale, si sono scelti i seguenti casi particolari

1. **Immagine vuota**
   Come dice il titolo del paragrafo, la foto in questione era vuota; quindi, in memoria aveva solo le dimensioni, di cui una era pari a 0. Il waveform dei segnali dopo il test è il seguente:

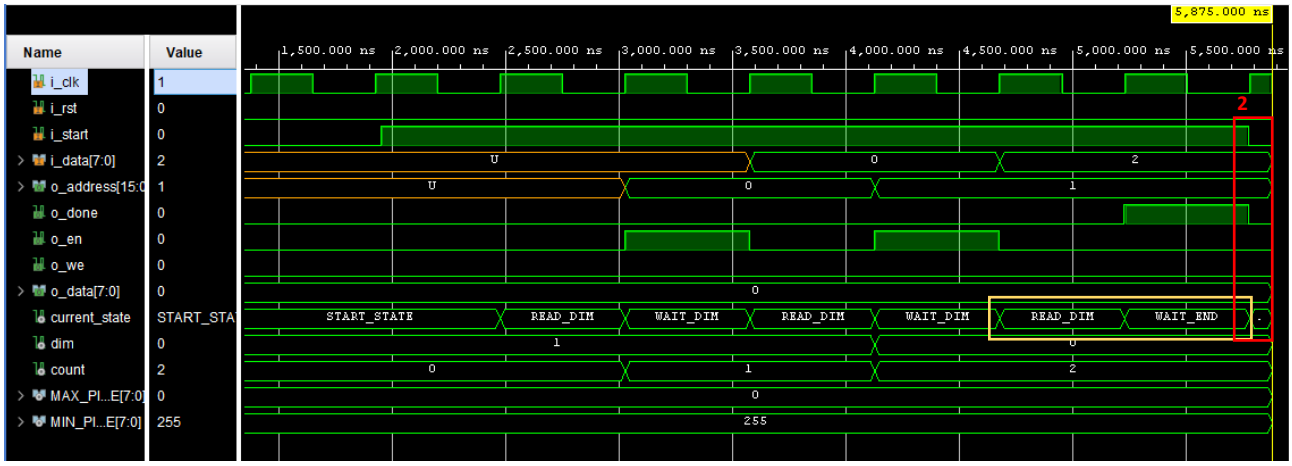

Figura 3: simulazione in pre-sintesi con una foto vuota

Guardando gli stati, si può notare come la condizione *if dim = 0* nello stato *read_dim* sia entrata in funzione e abbia permesso di terminare la computazione, passando direttamente allo stato *wait_end* (1). Infine, si può osservare che una volta portato in alto *o_done*, il segnale *i_start* viene tenuto basso e il nuovo stato diventa *start_state* (2)

2. **Un solo pixel**

   Lo scopo di questo test era quello di controllare il corretto funzionamento delle attività basilari dell'elemento durante la conversione di un solo pixel.
   L'andamento dei segnali e gli stati assunti durante il processo, tra cui c'è lo stato *conversion*, hanno confermato questo aspetto.

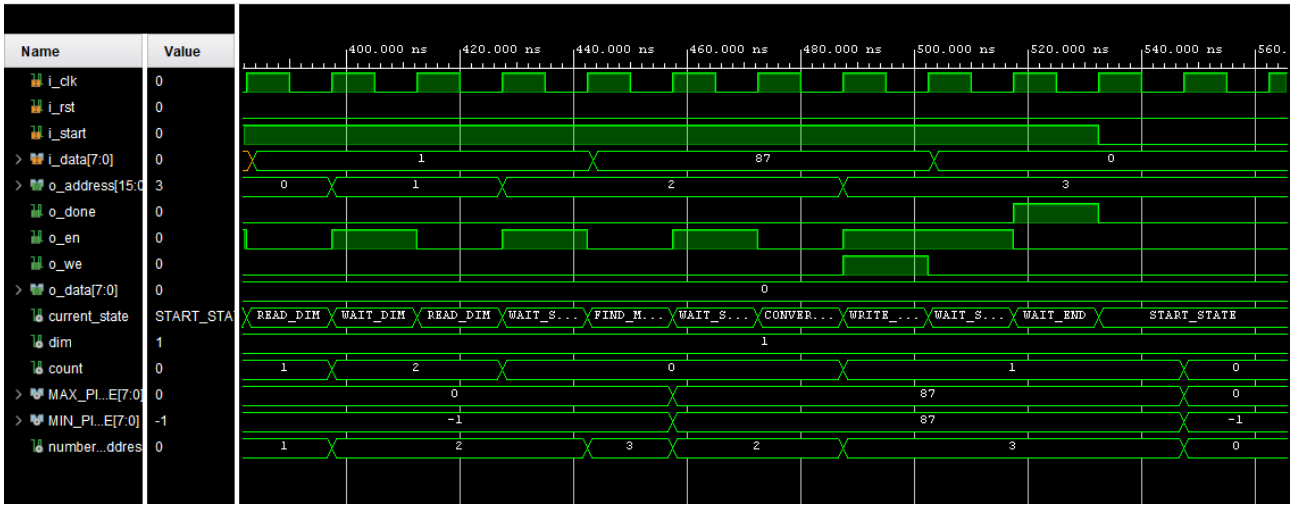

Figura 4: segnali durante il test con una foto di dimensione 1

3. **Tutti i pixel hanno come valore 255**

   Con questo caso particolare si è voluto testare anche un altro simile: quello in cui il valore dei pixel fosse unico e pari a 0.

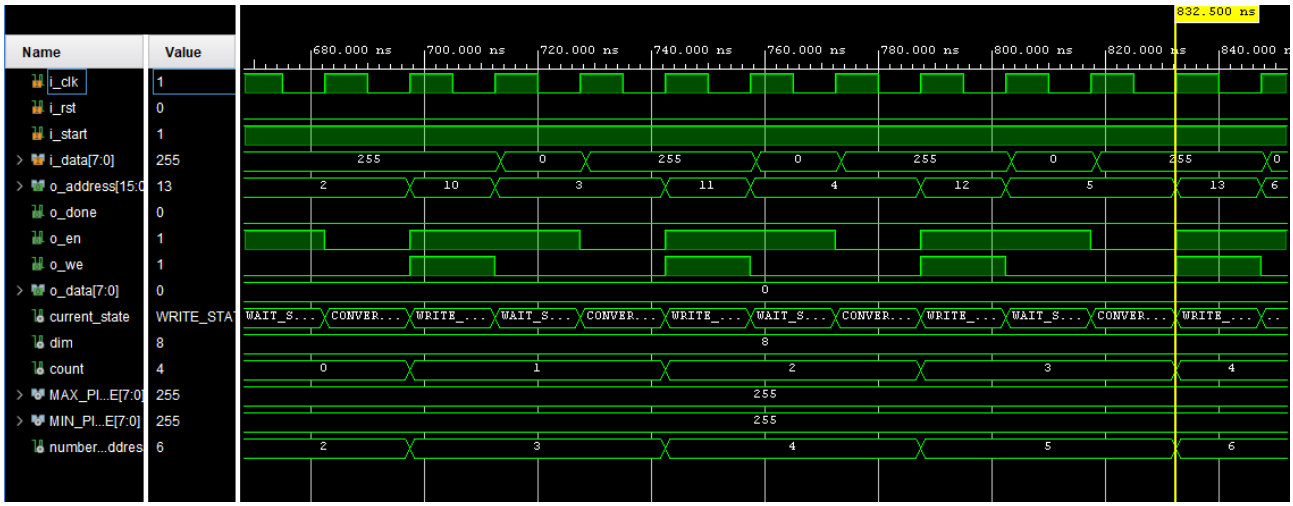

Figura 5: schema di una immagine processata che ha come unico valore 255

Lo schema ottenuto dimostra come l'immagine sia stata elaborata normalmente senza che venisse privilegiata per la sua semplicità in termini di tempo di lettura e scrittura agli altri casi: infatti ogni pixel viene elaborato e scritto, uno alla volta.
Questo risultato evidenzia come il componente sia preciso nella codifica, ma pagando un certo prezzo in termini di tempo.
Infine, sempre con questo TestBench, oltre alle due casistiche sopracitate, si è analizzato contemporaneamente anche la situazione generale, dove la foto è una sola e i valori dei pixel variano tra 0 e 255: come viene mostrato nel waveform, i pixel vengono elaborati singolarmente.

4. Tre immagini consecutive con la presenza di un segnale di reset

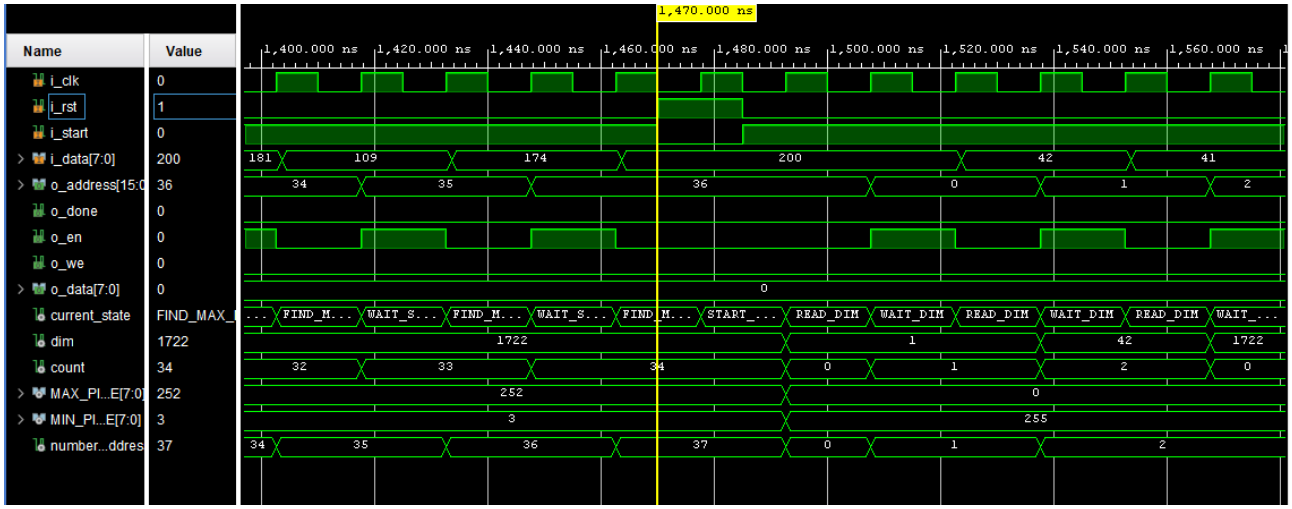

Figura 6: il segnale di reset viene alzato durante la simulazione

In questa situazione, il segnale di reset è stato alzato mentre era in corso la codifica di una immagine. Questo ha interrotto correttamente tutto e ha riportato la FSM allo stato iniziale, ma ha fatto ripartire la conversione da zero: quindi da una parte, la macchina continua a leggere e a scrivere un pixel alla volta, dall'altra, in caso di interruzione, il processo ricomincia e questo comporta maggior spreco di tempo in cambio di una maggior precisione nel calcolo.

Oltre a questi casi particolari, è stato usato anche un generatore di test scritto in C che permetteva di creare diverse situazioni e di simularle, caricando il test in formato VHDL su VIVADO come Design Source. Inoltre, in caso di errori (messaggi di *assertions*), riportava un resoconto di questi all'interno di un file di testo nominato dentro il codice del test.

4. CONCLUSIONE

4.1 RISULTATI DELLA SINTESI E DELLA "IMPLEMENTATION"

Il componente è stato scritto in linguaggio VHDL, totalmente sintetizzabile, ed è stato testato nei 3 tipi di simulazioni disponibili all'interno di VIVADO: *Behavioral, Post-Synthesis Functional* e *Post-Synthesis Timing.*

Il minor tempo di simulazione in *Behavioral* è stato ottenuto analizzando il caso in cui veniva data un'immagine vuota: 5875 ns.

Il maggior tempo di simulazione invece non si può definire perché dipende dal numero di pixel e di immagini che vengono date al componente.

Infine, dalla fase di "*implementation*" si può vedere che la quantità di energia che l'elemento consuma è di 12.115W , distribuiti tra i segnali, le porte logiche e I/O come segue:

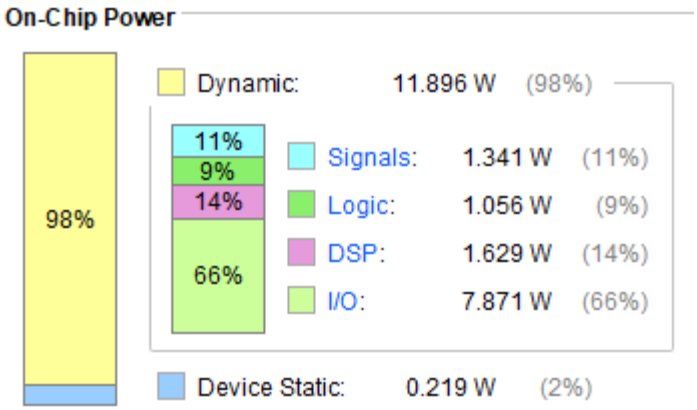

Figura 7: l'energia consumata dal chip

4.2 OTTIMIZZAZIONI

Le ottimizzazioni accennate nella relazione sono le seguenti:

- L'idea di un processo descritto con il VHDL, che riprende la programmazione con linguaggio C, è stata sostituita con una concezione a stati della macchina (FSM) perché i cicli for usati non erano sintetizzabili. Così facendo, l'elemento è risultato sintetizzabile e il suo codice compatto, ordinato e funzionante dal punto di vista dei cicli.
- Le equazioni utili alla conversione sono state spezzate in modo tale da facilitare la lettura del codice. Inoltre, con lo stesso scopo, oltre a quello di rendere comprensibile i passaggi fatti nel calcolo della variabile *shift_level*, è stata aggiunta la funzione *shift_lvl* che lavora con controlli a soglia.
- Infine, sono stati inseriti dei contatori e dei flag che permettono di distinguere determinate situazioni e quindi diversi comportamenti che il componente può assumere durante il

processo. Un esempio importante è il flag *M_m* che a seconda del valore assunto, nello stato *wait_state*, permette di distinguere tre percorsi diversi.