Relazione Progetto Reti Logiche

AA 2020/2021

10605775 – Campana Lorenzo

10611332 – Cordioli Matteo

POLITECNICO
MILANO 1863

INDICE

1. INTRODUZIONE

La specifica del problema, che l'architettura di seguito descritta è volta a risolvere, tratta dell'equalizzazione di un'immagine in bianco e nero a 256 livelli. L'immagine ci è fornita su una memoria da 65536 byte, ogni byte rappresenta un valore corrispondente a un pixel di questa immagine. I primi 2 valori della memoria, ovvero le posizioni 0 e 1, corrispondono al numero di colonne (*col*) e righe (*rig*) che formano l'immagine da computare. La nostra architettura scrive l'immagine equalizzata a partire dal primo indirizzo di memoria disponibile dopo l'ultimo valore fornito (corrispondente all'indirizzo: *2 + (col*rig))*.
L'immagine equalizzata viene calcolata secondo i seguenti passaggi:

Calcoli preliminari:

1. si ricercano il valore massimo (*max*) e minimo (*min*) dei pixel che formano l'immagine;
2. si calcola il delta tra questi 2 valori, *delta=max-min*;
3. si calcola lo *shift_lv= 8 - floor(log2(delta+1)).*

Calcoli per ogni pixel (*curr_pixel*):

1. si calcola il *temp_pixel = (curr_pixel -min) << shit_lv* ;
2. si computa il *new_pixel = min(255, temp_pixel)*, si ottiene così il valore di questo pixel nell'immagine equalizzata.

1.1 Esempio Specifica

Per comprendere al meglio lo scopo di questo progetto, si riporta di seguito un esempio di applicazione dell'algoritmo di equalizzazione per un'immagine campione. Nella Fig.1 viene mostrato lo stato della RAM prima dell'effettivo calcolo della nuova immagine, mentre nella Fig.2 viene riportato lo stato della RAM al termine dell'esecuzione.

Fig. 1

RAM(0)	00000010 (2)	Numero Colonne *N_Col*
RAM(1)	00000010 (2)	Numero Righe *N_Rig*
RAM(2)	00101110 (46)	Primo Valore Utile
RAM(3)	10000011 (131)	Secondo Valore Utile
RAM(4)	00111110 (62)	Terzo Valore Utile
RAM(5)	01011001 (89)	Quarto Valore Utile
	….	

Immagine NON Equalizzata

Fig. 2

RAM(0)	00000010 (2)	Numero Colonne *N_Col*
RAM(1)	00000010 (2)	Numero Righe *N_Rig*
RAM(2)	00101110 (46)	Primo Valore Utile
RAM(3)	10000011 (131)	Secondo Valore Utile
RAM(4)	00111110 (62)	Terzo Valore Utile
RAM(5)	01011001 (89)	Quarto Valore Utile
RAM(6)	00000000 (0)	Primo Valore Equalizzato
RAM(7)	11111111 (255)	Secondo Valore Equalizzato
RAM(8)	01000000 (64)	Terzo Valore Equalizzato
RAM(9)	00001000 (8)	Quarto Valore Equalizzato
	….	

Immagine NON Equalizzata

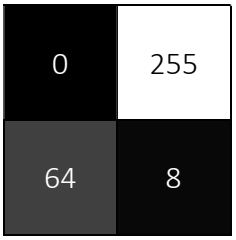


Immagine Equalizzata

2. ARCHITETTURA

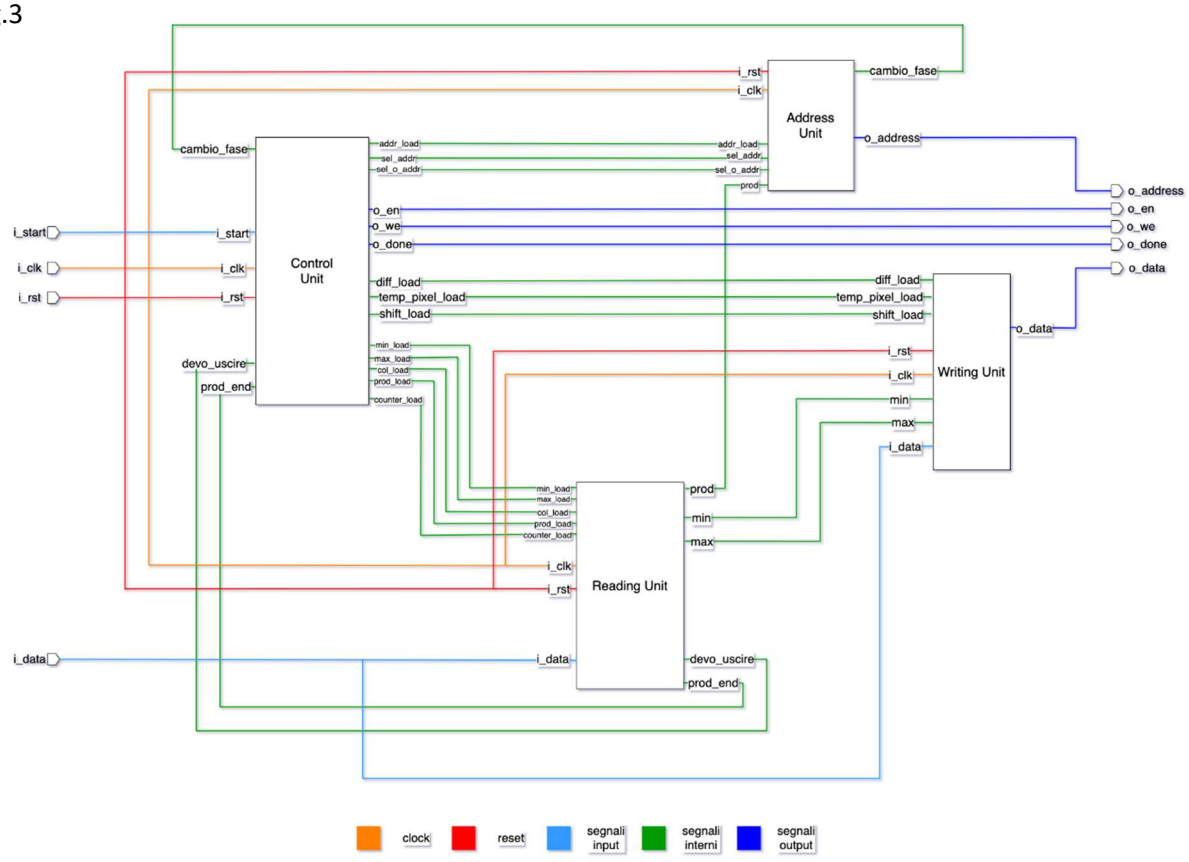
Per poter separare le funzionalità all'interno del nostro progetto e per ottimizzare al meglio i cicli di clock che l'elaborato compie per terminare l'esecuzione, l'architettura è stata progettata in maniera modulare, realizzando 4 moduli sequenziali: ControlUnit, ReadingUnit, WritingUnit, AddressUnit. La gestione dell'architettura è affidata a una Macchina a Stati Finiti (FSM), descritta successivamente.

2.1 Moduli

Per comprendere al meglio la nostra implementazione, riportiamo di seguito una breve descrizione dei moduli utilizzati e in Fig.3 le loro connessioni:

- **ControlUnit:** controlla e coordina tutti i segnali necessari per il corretto funzionamento del circuito e si assicura che la FSM proceda come da progetto.
- **ReadingUnit:** calcola il prodotto *(prod)* e si occupa di leggere tutti i valori utili presenti nella memoria e di restituire il valore massimo e minimo.
- **WritingUnit:** restituisce il valore equalizzato del pixel pronto per essere scritto in memoria, effettuando il calcolo della differenza tra il valore corrente e il minimo valore trovato dalla ReadingUnit per poi calcolare il valore definitivo. Necessita di alcuni segnali computati dalla ReadingUnit.
- **AddresssUnit:** calcola l'indirizzo di memoria necessario al prossimo ciclo di clock. In input sono forniti dei segnali che indicano se l'indirizzo (*o_address*) sarà utilizzato in scrittura o in lettura e se l'indirizzo (*o_address*) dovrà essere impostato al primo valore utile della memoria ("0000000000000010" => 2) durante il passaggio da ReadingUnit a WritingUnit.

Fig.3



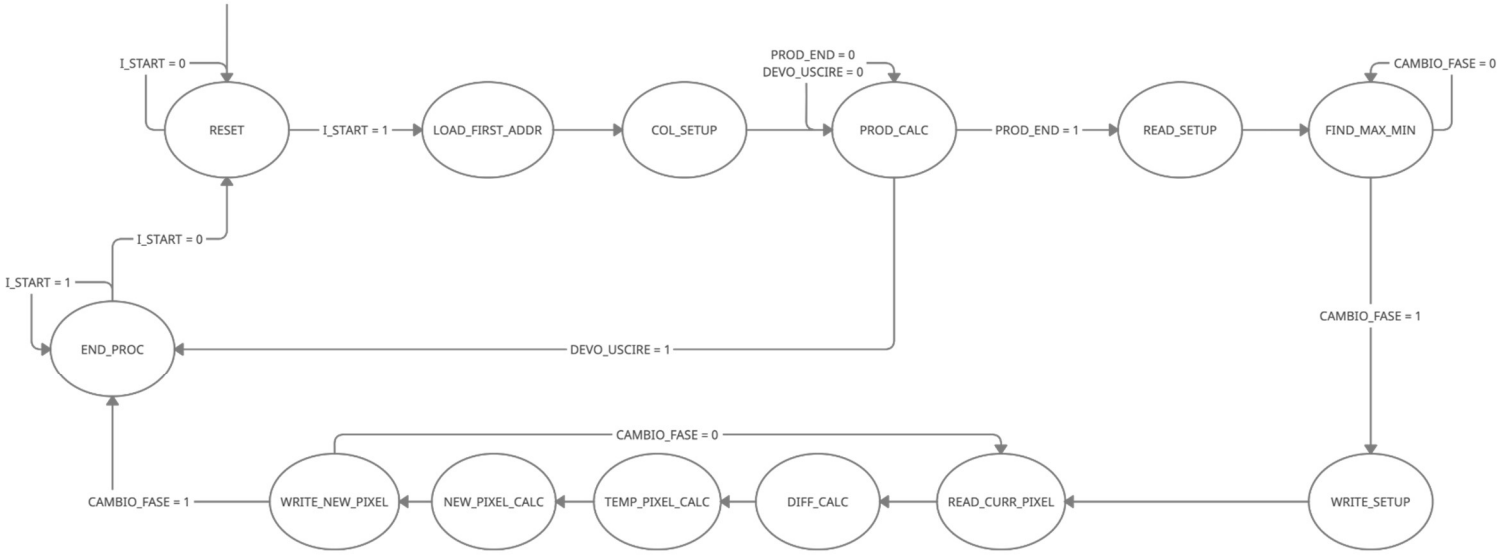
2.2 Macchina a Stati Finiti (FSM)

Diamo ora una descrizione più analitica e dettagliata della macchina a stati finiti (FSM) che è stata implementata per la risoluzione della specifica. Nella Fig.4 è inoltre riportato il diagramma della FSM.

- RESET: lo scopo di questo stato è duplice: all'inizio, attende il segnale alto di *i_start* per poter cominciare la computazione, inoltre, ogni qualvolta venga alzato il segnale di *i_rst* indipendentemente dallo stato corrente, si torna a questo stato, detto appunto "di reset" (tutti i segnali e registri vengono infatti settati a un valore di default).
- LOAD_FIRST_ADDR: si inizia col caricare il primo indirizzo che sappiamo essere valido, *RAM(0)*, in modo da poter estrarre informazioni dalla RAM dal ciclo di clock successivo. Questo stato permette quindi di caricare l'indirizzo iniziale e di settare la lettura dalla RAM.
- COL_SETUP: viene salvato il primo valore che si legge dalla RAM in un registro *(col)* e si continua a far avanzare l'indirizzo di una posizione in modo da poter leggere il prossimo valore al ciclo seguente.
- PROD_CALC: in questo stato ci si occupa del calcolo del prodotto *rig * col*: *rig* sarà contenuto in *i_data,* mentre *col* è stato salvato in un registro precedentemente.
  Si individua il valore minimo tra *rig* e *col* N=min(*rig*, *col*) e si itera sommando N volte l'altro valore. In questo modo si assicura che il prodotto venga eseguito per somme successive e allo stesso tempo che l'esecuzione dell'elaborato sia ottimizzata dal punto di vista temporale.
  Durante questo stato viene inoltre disabilitato l'accesso in memoria e l'avanzamento progressivo dell'indirizzo *(o_address)*, poichè sono disponibili già tutte le informazioni necessarie per calcolare in maniera esatta il prodotto *(prod)*. Se *rig* o *col* hanno valore "00000000" (0), il che implicherebbe una immagine con 0 pixel, si torna allo stato RESET in attesa di una nuova immagine.
- READ_SETUP: dopo aver calcolato il prodotto ci si assicura di ri-attivare l'avanzamento progressivo di *o_address*, che era momentaneamente bloccato all'indirizzo 2, in modo da leggere il primo valore utile già dal ciclo di clock successivo.
- FIND_MAX_MIN: si scorrono tutti i valori utili dell'immagine per identificare il valore massimo e il minimo che si salvano nei rispettivi registri *max* e *min* inizializzati durante il RESET a *max*="00000000" (0) e *min*="11111111" (255).
  Quando saranno terminati i valori utili da leggere e confrontare, si alzerà il segnale *cambio_fase* che permetterà l'avanzamento al prossimo stato.
- WRITE_SETUP: in questo stato ci si prepara per la rielaborazione dei valori dei pixel appena letti.
  Si riporta l'indirizzo della memoria al primo valore utile dell'immagine (in modo da poterlo leggere al ciclo di clock successivo), si calcola il *delta_value* e lo *shift_level,* che viene salvato in un registro *(shift)* come da specifica. Questa operazione viene effettuata una volta sola, perché si tratta di valori costanti, che serviranno per l'elaborazione di tutti i pixel.
- READ_CURR_PIXEL: viene letto il valore all'indirizzo di memoria corrente.
- DIFF_CALC: viene calcolata la differenza *(diff)* tra il valore corrente il valore minimo. Questa viene quindi salvata in un registro a 16 bit.
- TEMP_PIXEL_CALC: si calcola *temp_pixel* come da specifica, effettuando lo shift su *diff,* selezionando solo i bit necessari e concatenando gli zeri sufficienti ad arrivare a 16 bit.
- NEW_PIXEL_CALC: si calcola *new_pixel* come da specifica, ovvero selezionando il minimo tra 255 e *temp_pixel* su 16 bit e successivamente consideriamo solo gli 8 bit meno significativi e li si salva in *o_data* (disponibile al ciclo successivo).
- WRITE_NEW_PIXEL: viene abilitata la modalità di scrittura sulla RAM, si scrive all'indirizzo del valore letto + prodotto e si incrementa di uno l'indirizzo, per poter leggere il valore seguente da elaborare al ciclo di clock successivo.
  Dopo aver processato l'ultimo pixel dell'immagine, *cambio_fase* sarà ad un valore alto, il che condurrà nello stato END, altrimenti si proseguirà a leggere e calcolare i nuovi valori dell'immagine equalizzata e si tornerà in READ_CURR_PIXEL.

- **END:** viene alzato il segnale di *o_done* che, come da specifica, indica il termine dell'equalizzazione. Se il segnale *i_start* viene portato alto significa che è presente una nuova immagine da processare in RAM e quindi ci si riporta allo stato di RESET, per processare la nuova immagine.

Fig.4



3. RISULTATI SPERIMENTALI

In questa sezione vengono riportati alcuni risultati del nostro elaborato, dando particolare importanza all'aspetto di testing, che ha avuto sicuramente un ruolo chiave nel nostro percorso.

3.1 Report di Sintesi (Post Functional)

La sintesi secondo strategia standard effettuata con Vivado2020.2 con FPG xc7a200tfbg484-1 vede l'impiego di 175 LUT (0.13%) e 97 FF (0.04%).

3.2 Test Benches (TB)

Sono state testate circa 100K immagini in diverse configurazioni. I primi test sono stati fatti per verificare i casi limite più critici e successivamente ne sono stati sottomessi altri per verificare la correttezza dell'operato.

In particolare, sono stati analizzati i seguenti casi:

- **Caso Limite 1:** immagine vuota, ovvero o la riga *(rig)* o la colonna *(col)* erano state assegnate uguali a zero.
- **Caso Limite 2:** immagine con dimensione massime, ovvero 128x128 (16384 valori utili).
- **Caso Limite 3:** immagine con tutti valori uguali, ed in particolare sono stati testati i casi con tutti 0, tutti 255 e tutti 120 (numero casuale all'interno dell'intervallo).
- **Immagini in Sequenza:** sono state fornite più immagini, una di seguito all'altra, per verificare che l'elaborato fosse effettivamente in grado di gestire più immagini in successione.
- **Reset Asincrono:** sono stati creati dei TB con dei segnali di reset all'interno dell'esecuzione dell'algoritmo risolutivo, in modo da verificare se la FSM fosse in grado di reagire ad un reset asincrono.
- **Generatore di Test:** è stato implementato in C un programma che, dati in input il numero di casi da voler generare e il numero massimo di righe e colonne, può generare un file di testo con tutti i valori di input e output attesi per le esecuzioni. In questo modo, con un TB che leggeva da file di testo *(.txt),*

abbiamo potuto testare circa 100k immagini in blocchi da 5/10k con dimensione variabile tra 0 e 128 sia per colonne che per le righe.

4. CONCLUSIONI

I test, ed in particolare i casi limite, hanno permesso di raffinare la nostra implementazione sempre più, fino allo stato finale in cui la stiamo presentando. Tutti i test citati nella sezione 3.2 sono stati eseguiti e superati sia in modalità Behavioural che in Post Functional. Possiamo quindi affermare che l'architettura presentata rispetti anzitutto la specifica data e risulti inoltre ottimizzata dal punto di vista temporale, ovvero che ogni ciclo di clock sia utilizzato nel modo più efficiente possibile e che i cicli necessari allo svolgimento della specifica siano stati minimizzati. Per esempio: durante la lettura nel ciclo corrente, si prepara l'indirizzo a cui accedere al prossimo ciclo, inoltre si calcola il minimo tra *rig* e *col* prima di fare il prodotto per somme successive.