POLITECNICO
MILANO 1863

Prova Finale di Reti Logiche

Alessio Braccini: 10621546
Fabio Berzoini: 10560200

Anno Accademico 2020/2021

Professore: Fornaciari William
Esercitatore: Terraneo Federico

Indice

1. Introduzione

1.1 Specifica

Il progetto consiste nell'implementazione in VHDL di un equalizzatore di immagine.
Data un'immagine formata da byte contigui memorizzati nella RAM viene chiesto di svolgere dei calcoli per poter ricalibrare il contrasto quando l'intervallo dei valori di intensità sono molto vicini effettuandone una distribuzione su tutto l'intervallo di intensità al fine di incrementare il contrasto.
I passaggi algebrici da fare sono di seguito elencati:

1. $\Delta V = MAX\_PIXEL\_VALUE - MIN\_PIXEL\_VALUE$
2. $SL = 8 - \lfloor log_2(\Delta V + 1) \rfloor$
3. $Tp = CURRENT\_PIXEL\_VALUE - MIN\_PIXEL\_VALUE << SL$
4. $NpV = min(255, Tp)$

Il modulo da implementare dovrà leggere l'immagine dalla RAM in cui è memorizzata, sequenzialmente e riga per riga. Ogni byte corrisponde ad un pixel dell'immagine, mentre i primi 2 byte sono rispettivamente la dimensione di riga e colonna dell'intera immagine che può variare da 0x0 a 128x128 pixel.
I nuovi valori dovranno essere inseriti immediatamente dopo l'ultimo indirizzo dell'immagine originaria, cioè a partire dall'indirizzo $2 + (\#COL * \#RIG)$.
L'implementazione deve essere sintetizzata con target FPGA xc7a200tfbg484-1

1.2 Esempio

Si riporta un esempio di equalizzazione di un'immagine: nella RAM sono contenuti tutti i valori da equalizzare e già equalizzati.

INDIRIZZO MEMORIA	VALORE	COMMENTO
0	4	\\ Byte più significativo numero colonne
1	3	\\ Byte meno significativo numero righe
2	76	\\ primo Byte immagine
3	131	
4	109	
5	89	
6	46	
7	121	
8	62	
9	59	
10	46	
11	77	
12	68	
13	94	\\ ultimo Byte immagine
14	120	\\ primo Byte immagine equalizzata (risultato)
15	255	
16	252	
17	172	
18	0	
19	255	
20	64	
21	52	
22	0	
23	124	
24	88	
25	192	

1.3 Ipotesi Progettuali

1. Il modulo deve essere progettato per poter codificare più immagini, ma l'immagine da codificare non verrà mai cambiata all'interno della stessa esecuzione, ossia prima che il modulo abbia segnalato il completamento tramite il segnale DONE.
2. Il modulo partirà nell'elaborazione quando un segnale START in ingresso verrà portato a 1. Il segnale di START rimarrà alto fino a che il segnale di DONE non verrà portato alto. Al termine della computazione ,e una volta scritto il risultato in memoria, il modulo da progettare deve alzare

(portare a 1) il segnale DONE che notifica la fine dell'elaborazione. Il segnale DONE deve rimanere alto fino a che il segnale di START non è riportato a 0. Un nuovo segnale start non può essere dato fin tanto che DONE non è stato riportato a zero. Se a questo punto viene rialzato il segnale di START, il modulo dovrà ripartire con la fase di codifica.

2. Architettura

Questa sezione è dedicata alla presentazione della macchina a stati che governa il comportamento del circuito.

2.1 Architettura ad alto livello

In un'ottica di alto livello l'implementazione esegue i seguenti passi:

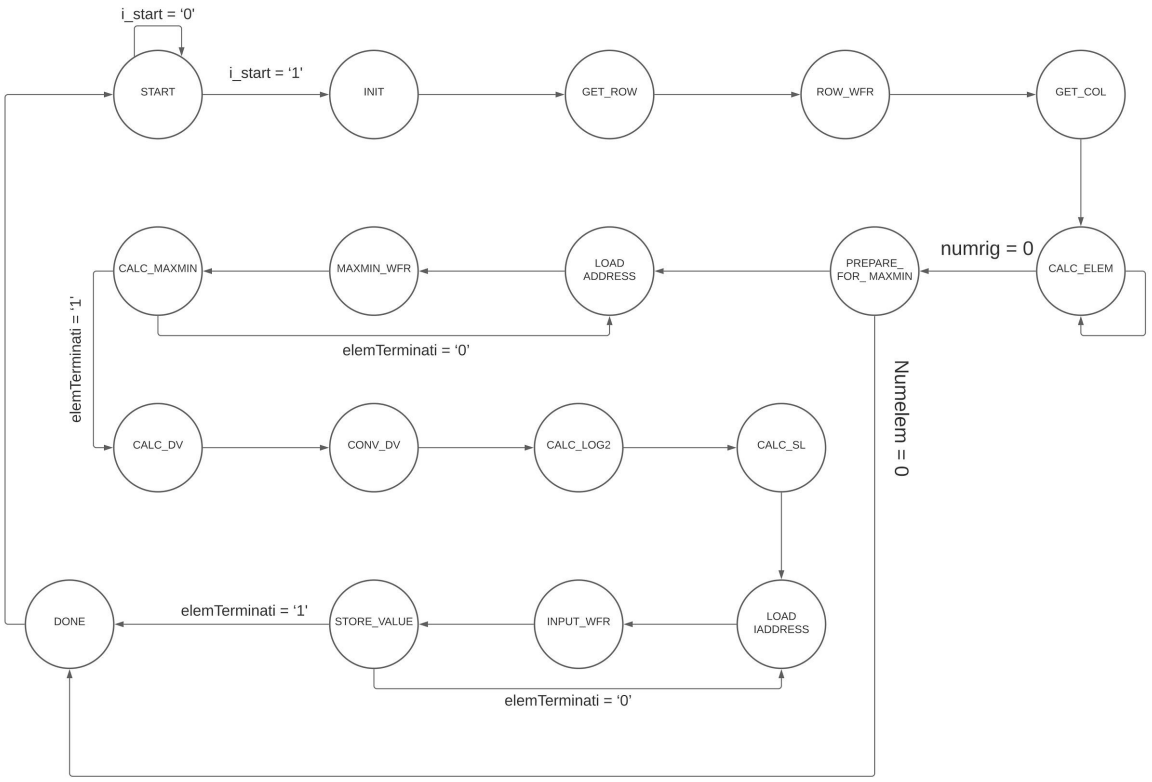
1. Acquisisce il primo e secondo byte dalla memoria corrispondente rispettivamente al numero di righe e di colonne
2. Calcola il numero di pixel presenti nell'immagine
3. Cerca il pixel con il massimo e il minimo valore
4. Calcola delta value e shift value
5. Calcola il nuovo valore che deve assumere il pixel e lo memorizza nella memoria

I passaggi 3 e 5 sono dei cicli che svolgono operazioni per ogni pixel.

2.2 Macchina a stati finiti

La **Macchina a Stati Finiti** (FSM) è il circuito che si occupa di gestire lo stato della computazione e che si interfaccia con la memoria per il trasferimento dei dati. La FSM è realizzata con specifica behavioural mediante due processi.
Il primo processo è il processo sequenziale che si occupa di cambiare lo stato interno sul fronte di salita del clock.
Il secondo è, invece, il processo combinatorio che si occupa di calcolare le uscite e lo stato validi per il prossimo fronte di salita.

Segue uno schema e una spiegazione dettagliata degli stati della FSM.



- START: stato iniziale dove si posiziona la macchina al reset della computazione, essa aspetta che venga asserito il segnale i_start.

- INIT: stato in cui si aspetta un ciclo di clock per permettere alla memoria di asserire le sue uscite.

- GET_ROW: stato in cui si legge il dato dalla posizione 0 della memoria e viene salvato il numero di righe dell'immagine.

- ROW_WFR: stato in cui si aspetta un ciclo di clock per permettere alla memoria di asserire le sue uscite.

- GET_COL: stato in cui si legge il dato dalla posizione 1 della memoria e viene salvato il numero di colonne dell'immagine.

- CALC_ELEM: stato in cui tramite un loop, formato da un autoanello, si calcola il numero dei pixel che formano l'immagine. Il funzionamento

consiste nel sommare il numero di colonne tante volte quante sono le righe, nel caso pessimo questo ciclo compirà 128 iterazioni.

- PREPARE_FOR_MAXMIN: stato in cui si preparano alcuni parametri per la ricerca del massimo e minimo oppure nel caso l'immagine sia formata da 0 elementi passa direttamente allo stato di DONE.
  I parametri inizializzati sono il massimo a '0' e il minimo a '255' questo consente di coprire tutti i casi possibili. Un altro parametro inizializzato è il primo indirizzo da leggere per far partire l'iterazione.

- LOAD_ADRESS: stato in cui si carica l'indirizzo della memoria in cui risiede il dato da leggere per la computazione successiva e in cui viene attivato il segnale "o_en".

- MAX_MIN_WFR: stato in cui si aspetta un ciclo di clock per permettere alla memoria di asserire le sue uscite.

- CALC_MAXMIN: stato in cui si calcola se un valore può essere massimo o minimo, questo viene eseguito tramite un confronto tra il massimo e il minimo memorizzati e il dato che proviene dalla memoria. Quando gli elementi in memoria sono terminati si asserisce il segnale "elemTerminati" a '1' che fa procedere la FSM allo stato "CALC_DV".

- CALC_DV: stato in cui si calcola il valore del delta value tramite l'apposita formula. Inoltre in questo stato si resetta il valore di "elemTerminati" a '0'.

- CONV_DV: stato in cui si inserisce il delta value in un vettore di 16 bit per la successiva computazione dello shift level. In questo stato si controlla anche se il delta value è pari a 255, in caso affermativo non si effettua tale conversione.

- CALC_LOG2: stato in cui si calcola il logaritmo base 2 per la successiva computazione dello shift level.

Questo conto avviene tramite un priority encoder che cerca il primo ‘1’ partendo da sinistra nel delta value appena convertito e codificando la posizione di tale 1 in numeri interi da 1 a 7, anche in questo caso se il delta value è pari a 255 non si effettua tale calcolo.

- CALC_SL: stato in cui si calcola lo shift level solo nel caso in cui il delta value valga meno di 255.
  Inoltre in questo stato si preparano i valori di memoria per il successivo blocco di iterazioni, viene anche inserito in 2 vettori std_logic il valore di massimo e minimo.

- LOAD_IADDRESS: stato in cui si carica l’indirizzo della memoria in cui risiede il dato da leggere per la computazione successiva e in cui viene attivato il segnale “o_en”.

- INPUT_WFR: stato in cui si aspetta un ciclo di clock per permettere alla memoria di asserire le sue uscite.

- STORE_VALUE: stato in cui si computa il nuovo valore che deve assumere il pixel tramite le formule sopra citate e viene memorizzato nel nuovo indirizzo della memoria, si incrementano anche gli indirizzi per preparare la macchina al successivo byte di memoria asserendo a ‘1’ sia il seganle “o_en” che “w_en”.

- DONE: stato finale in cui o_done viene portato alto e decreta il termine della computazione.

L’implementazione è anche in grado di gestire un segnale di Reset.
Si è deciso di non utilizzare l’operatore ‘*’ e di utilizzare il ciclo con autoanello nello stato CALC_ELEM in quanto il numero di pixel dell’immagine si può calcolare tramite una serie di somme successive. Così facendo si ha effettivamente l’idea di quanti cicli di clock la macchina impiega a svolgere tale operazione piuttosto che demandare il compito ad un operatore più astratto che maschera il livello di complessità di un’operazione come la moltiplicazione,

nonostante il caso pessimo compia 128 iterazioni è comunque preferibile all'uso dell'operatore '*' per i motivi sopra citati.

3. Risultati Sperimentali

3.1 Report di sintesi

In questa parte verranno presentati alcuni estratti del report di sintesi generati dal tool Vivado.
Il primo report che viene mostrato è il "report_utilization", in particolare, vengono evidenziati lo spazio occupato e le tipologie di componenti sintetizzate come la divisione logica tra la parte LUT (Look Up Table) e quella riservata ai registri.
Si è fatta particolare cautela nella scrittura del codice per evitare utilizzo di Latch.

Site Type	Used	Fixed	Available	Util%
Slice LUTs*	286	0	134600	0.21
LUT as Logic	286	0	134600	0.21
LUT as Memory	0	0	46200	0.00
Slice Registers	177	0	269200	0.07
Register as Flip Flop	177	0	269200	0.07
Register as Latch	0	0	269200	0.00
F7 Muxes	0	0	67300	0.00
F8 Muxes	0	0	33650	0.00

Di seguito vengono mostrate le tabelle riguardo alla simulazione delle tempistiche che vengono fornite attraverso il comando "report_timing".

L'implementazione rispetta la specifica dei 100ns e si nota come il percorso critico abbia un ritardo di soli 5,136ns.

```
Timing Report

Slack (MET) :             94.251ns  (required time - arrival time)
  Source:                 dv_reg[31]/C
                            (rising edge-triggered cell FDRE clocked by clock  {rise@0.000ns fall@5.000ns period=100.000ns})
  Destination:            dv_reg[1]/R
                            (rising edge-triggered cell FDRE clocked by clock  {rise@0.000ns fall@5.000ns period=100.000ns})
  Path Group:             clock
  Path Type:              Setup (Max at Slow Process Corner)
  Requirement:            100.000ns  (clock rise@100.000ns - clock rise@0.000ns)
  Data Path Delay:        5.136ns  (logic 1.883ns (36.663%)  route 3.253ns (63.337%))
  Logic Levels:           7  (CARRY4=3 LUT2=2 LUT5=1 LUT6=1)
  Clock Path Skew:        -0.145ns (DCD - SCD + CPR)
    Destination Clock Delay (DCD):    2.100ns = ( 102.100 - 100.000 )
    Source Clock Delay      (SCD):    2.424ns
    Clock Pessimism Removal (CPR):    0.178ns
  Clock Uncertainty:      0.035ns  ((TSJ^2 + TIJ^2)^1/2 + DJ) / 2 + PE
    Total System Jitter     (TSJ):    0.071ns
    Total Input Jitter      (TIJ):    0.000ns
    Discrete Jitter          (DJ):    0.000ns
    Phase Error              (PE):    0.000ns
```

3.2 Simulazioni

Questa sezione è dedicata a tutti i casi di test a cui il codice è stato sottoposto.

Essi sono stati scelti per cercare di coprire tutti le situazioni presentabili effettuando transizioni critiche o verificando condizioni di memoria estreme. Inoltre, verranno presentati anche screenshot del superamento degli stessi.

Si è anche creato un piccolo programma in C++ che simula il funzionamento dell'implementazione in modo da non doversi calcolare a mano tutti i risultati per verificarne la correttezza, questo ci ha aiutato ad avere tempi più brevi per la fase di testing. Si allega il codice qui di seguito.

```cpp
int main(int argc, char** argv) {

    int max, min, dv, sl, nel, rig, col;
    float tmp;

    max = 0;
    min = 255;
    printf("ins rig e col");
    cin >> rig;
    cin >> col;
    nel = rig*col;
    int val[nel];
    int vale[nel];
    for (int i = 0; i<nel; i++){
        cin >> val[i];
        if (val[i] > max)
            max = val[i];
        if (val[i] < min)
            min = val[i];
    }
    tmp = floor(log2(max-min+1));
    sl = 8 - (tmp);
    for(int i = 0; i<nel; i++){
        if ((val[i]-min << sl) < 255)
            vale[i] = val[i]-min << sl;
        else
            vale[i] = 255;
        cout << vale[i] << endl;
    }
    return 0;
}
```

Si descrivono di seguito i principali casi testati.

Caso 1) Solo valori 0, 128, 255 in memoria:

Caso servito per verificare un particolare corner case che ha subito dato un problema poiché non veniva gestito correttamente il logaritmo 2 di 256, quindi il codice è stato modificato in modo da far risultare corretto tale logaritmo.

[9][7:0]	255
[8][7:0]	128
[7][7:0]	128
[6][7:0]	0
[5][7:0]	255
[4][7:0]	128
[3][7:0]	128
[2][7:0]	0
[1][7:0]	2
[0][7:0]	2

```
signal RAM: ram_type := (0 => std_logic_vector(to_unsigned(  2  , 8)),
                         1 => std_logic_vector(to_unsigned(  2  , 8)),
                         2 => std_logic_vector(to_unsigned(  0  , 8)),
                         3 => std_logic_vector(to_unsigned(  128  , 8)),
                         4 => std_logic_vector(to_unsigned(  128  , 8)),
                         5 => std_logic_vector(to_unsigned(  255 , 8)),
```

```
    assert RAM(6) = std_logic_vector(to_unsigned( 0 , 8)) report "TEST FALLITO (WORKING ZONE). Expected  0  found " & integer'image(to_integer(unsigned(RAM(6))))  severity failure;
    assert RAM(7) = std_logic_vector(to_unsigned( 128 , 8)) report "TEST FALLITO (WORKING ZONE). Expected  255  found " & integer'image(to_integer(unsigned(RAM(7))))  severity failure;
    assert RAM(8) = std_logic_vector(to_unsigned( 128 , 8)) report "TEST FALLITO (WORKING ZONE). Expected  64  found " & integer'image(to_integer(unsigned(RAM(8))))  severity failure;
    assert RAM(9) = std_logic_vector(to_unsigned( 255 , 8)) report "TEST FALLITO (WORKING ZONE). Expected  172  found " & integer'image(to_integer(unsigned(RAM(9))))  severity failure;

    assert false report "Simulation Ended! TEST PASSATO" severity failure;
```

Caso 2) Funzionamento normale:

In questo caso la macchina non ha avuto nessun tipo di problema a calcolare i nuovi valori dei pixel.

[9][7:0]	172
[8][7:0]	64
[7][7:0]	255
[6][7:0]	0
[5][7:0]	89
[4][7:0]	62
[3][7:0]	131
[2][7:0]	46
[1][7:0]	2
[0][7:0]	2

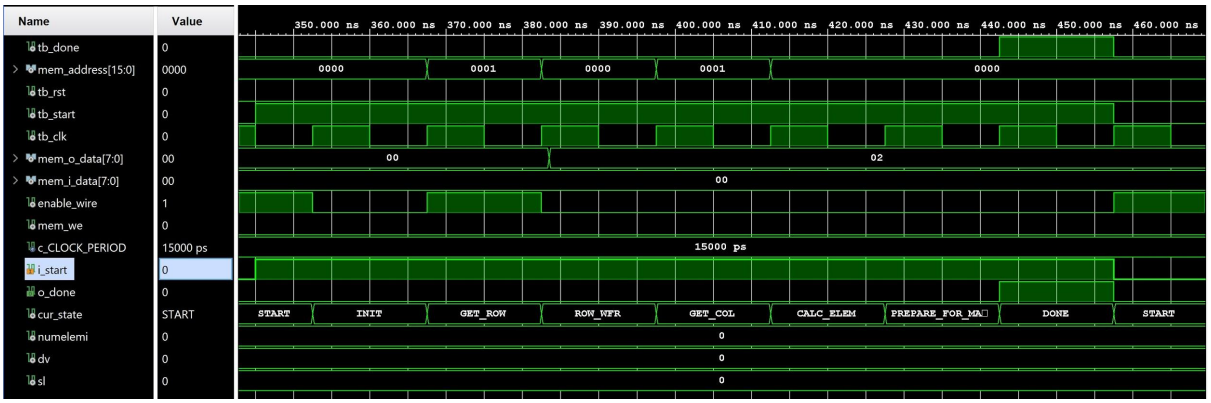
```
signal RAM: ram_type := (0 => std_logic_vector(to_unsigned(  2  , 8)),
                         1 => std_logic_vector(to_unsigned(  2  , 8)),
                         2 => std_logic_vector(to_unsigned(  46  , 8)),
                         3 => std_logic_vector(to_unsigned(  131  , 8)),
                         4 => std_logic_vector(to_unsigned(  62  , 8)),
                         5 => std_logic_vector(to_unsigned(  89  , 8)),
                        others => (others =>'0'));
            -- Expected Output  6 -> 0
            -- Expected Output  7 -> 255
            -- Expected Output  8 -> 64
            -- Expected Output  9 -> 172
```

```
    -- Immagine originale =  [46, 131, 62, 89]
    -- Immagine di output =  [0, 255, 64, 172]

    assert RAM(6) = std_logic_vector(to_unsigned( 0 , 8)) report "TEST FALLITO (WORKING ZONE). Expected  0  found " & integer'image(to_integer(unsigned(RAM(6))))  severity failure;
    assert RAM(7) = std_logic_vector(to_unsigned( 255 , 8)) report "TEST FALLITO (WORKING ZONE). Expected  255  found " & integer'image(to_integer(unsigned(RAM(7))))  severity failure;
    assert RAM(8) = std_logic_vector(to_unsigned( 64 , 8)) report "TEST FALLITO (WORKING ZONE). Expected  64  found " & integer'image(to_integer(unsigned(RAM(8))))  severity failure;
    assert RAM(9) = std_logic_vector(to_unsigned( 172 , 8)) report "TEST FALLITO (WORKING ZONE). Expected  172  found " & integer'image(to_integer(unsigned(RAM(9))))  severity failure;

    assert false report "Simulation Ended! TEST PASSATO" severity failure;
end process test;
```

Caso 3) Caso con 0 pixel:

Questo caso è servito per stressare una particolare condizione limite, cioè l’assenza di dati memorizzati in memoria. Esso ha fatto sorgere alcuni problemi

che sono stati risolti tramite un percorso alternativo nello stato PREPARE_FOR_MAXMIN che arriva direttamente nello stato di DONE.



Caso 4) Immagine 128*128:

Questo test è stato eseguito in modo da controllare che anche con la massima dimensione di un'immagine l'implementazione funzionasse normalmente, il test non ha sollevato nessuna criticità.

Caso 5) Reset:

Qui si riassumono tutte le casistiche dei test che hanno coinvolto dei reset, si è testata l'implementazione con reset sparsi in vari punti e si è testato anche il caso di reset asincrono non evidenziando in alcun modo complicazioni dovute al segnale di reset.

Caso 6) 1 pixel:

Si è testato il codice con un solo pixel, sia come riga (con colonna pari a 0) che come colonna (con riga pari a 0), anche in questo caso l'implementazione si è è comportata correttamente non facendo emergere alcuna criticità.

Caso 7) Immagini consecutive:

Si è voluto testare il codice con più immagini consecutive per verificare che il funzionamento fosse corretto anche in presenza di più immagini di seguito, nessuna criticità è stata riportata anche in questo caso.

Sono stati eseguiti altri numerosi test, di cui i più importanti visti qui sopra, alcuni anche con centinaia o migliaia di immagini con lo scopo di far emergere criticità nel codice. Gli altri test eseguiti sono stati ad esempio un'immagine con dei pixel o tutti a '1' o tutti a '255' oppure con massimo pari a '255' e

minimo pari a ‘0’ proprio per stressare casi limite e sempre facendo particolare attenzione al segnale di reset.

Nessuno di questi test ha generato criticità confermando la robustezza del codice scritto sia in simulazione Behavioral che in simulazione Post-Synthesis Functional.

4. Conclusioni

Come conclusione di questa prova finale possiamo affermare che questo progetto ci ha fatto entrare nel vivo del lavoro di programmazione hardware e ci ha permesso di andare oltre lo studio astratto.

La realizzazione del progetto è avvenuta nei tempi previsti grazie al lavoro di squadra, la sintonia e la serietà creatasi nel gruppo. Essi hanno fatto sì che il lavoro finale risultasse completo e curato, condividendo idee ma soprattutto aiutandoci a vicenda per raggiungere un obiettivo comune e superare le difficoltà iniziali.

Si ritiene che l’architettura rispetti le specifiche, fatto verificato dalle esaustive sessioni di testing a cui l’implementazione è stata sottoposta. Le migliorie apportate durante il progetto hanno permesso di incrementare le prestazioni dell’implementazione e di farci conoscere meglio il linguaggio VHDL.

Siamo soddisfatti del risultato finale in quanto ci ha consentito di approfondire una tematica come quella del VHDL permettendoci di crescere anche in ulteriori ambiti, come la gestione di un lavoro in team o quella di dover scrivere una relazione di progetto, sfida mai affrontata fino a questo punto ma che ci ha permesso di curare una delle parti fondamentali di un progetto.