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 realizzato ha un’interfaccia così definita:

\*code snippet e immagine tikz

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.

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 del contenuto 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 n\_col\*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 n\_col\*n\_row+2 all’indirizzo 2\* n\_col\*n\_row+1.

|  |  |
| --- | --- |
| N\_col | Indirizzo 0 |
| N\_row | Indirizzo 1 |
| Primo Pixel da elaborare | Indirizzo 2 |
| ….. |  |
| Ultimo pixel da elaborare | Indirizzo n\_col\*n\_row+1 |
| Primo Pixel elaborato | Indirizzo n\_col\*n\_row+2 |
| ….. |  |
| Ultimo pixel elaborato | Indirizzo 2\*n\_col\*n\_row+1 |

1.3 esempio di funzionamento

2 Design

Si è scelto di descrivere un modulo single-process tramite architettura behavioral (comportamentale) in linguaggio VHDL.

Questo ha determinato la necessità di sviluppare 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\_rig

2) calcolo dimensione 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 si occupa nel dettaglio dell’attuazione dell’algoritmo sviluppato.

2.1 Macchina a stati finiti

L’FSM da noi prodotta è composta da 10 stati, suddivisibili in 4 gruppi principali descritti di seguito.

2.1.1 stati ausiliari

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

1. WT\_RST – wait reset: stato iniziale di attesa del segnale i\_rst

2. 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, anche non in corrispondenza di i\_clk, la macchina viene riportata in questo stato, tornando in attesa di un nuovo segnale di inizio elaborazione.

NB: 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.

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.

3. RD\_REQ – read request: stato di abilitazione della memoria in lettura. Viene predisposto su o\_address l’indirizzo della RAM che deve essere letto

4. 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

5. 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 ricominciare il processo di equalizzazione di una nuova immagine

2.1.2 calcolo dimensioni

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

6. 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

7. 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 (n\_col\*n\_row>0) o meno per il proseguimento dell’esecuzione

2.1.3 ricerca di valori di massimo e minimo dell’immagine

Gruppo di stati che permette di individuare i valori minimo e massimo, 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.

8. 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 (count <= n\_col\*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 – EL\_DATA della macchina a stati finiti.

9. 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

10. 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: count-1+n\_col\*n\_row

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 corrente (count <= n\_col\*n\_row+2) si verifica, si passa a DONE, altrimenti si procede con l’equalizzazione del pixel successivo

\*immagine FSM

2.2 approfondimento sull’elaborazione

La manipolazione del contrasto dell’immagine è fondata 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\_PIXEL\_VALUE – MIN\_PIXEL\_VALUE

delta\_value rappresenta la differenza tra il pixel più chiaro (valore maggiore – max\_pixel\_value) e il più scuro (valore minore – min\_pixel\_value) dell’immagine

· SHIFT\_LEVEL = (8 – FLOOR(LOG2(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

La funzione FLOOR(X) svolge l’ arrotondamento per difetto del valore X fornitogli come argomento.

\* tabella di esempio, senza particolari spiegazioni. si può dire qualcosa nella descrizione

· TEMP\_PIXEL = (CURRENT\_PIXEL\_VALUE - MIN\_PIXEL\_ VALUE) << 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\_VALUE = MIN(255 , TEMP\_PIXEL)

NEW\_PIXEL\_VALUE 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.

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.

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 dalla formula matematica. Di seguito si riporta il blocco IF/ELSE che si è utilizzato per il calcolo dello shift\_value relativo a un dato delta\_value:

\*code snippet

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 \* N\_ROW = 0

· Casi limite di dimensione dell’immagine 1x1 e 128x128

· 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.

\*immagine segnali significativi

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.

\*immagine esempio 2x2 post synthesis

3.2 risultati sperimentali

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

● LUT

● FF

● DSP

● IO

● BUFG

Come precedentemente accennato, sono presenti nel componente finale \* DSP, dovuti 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: \*ris

· simulazione post-synthesis functional: \*ris

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 al funzionamento di un componente dalle caratteristiche simili a quello da noi proposto.