POLITECNICO
MILANO 1863

Prova Finale di Reti Logiche

AA 2020-2021

Prof. Gianluca Palermo

Gabriele Passoni, Michelangelo Rosa

INDICE DEI CONTENUTI

INTRODUZIONE

Il progetto si pone lo scopo di sviluppare un componente hardware, descritto attraverso specifica *VHDL*, in grado di effettuare un'operazione di equalizzazione di un'immagine in scala di grigi a 256 livelli.
Le immagini in ingresso sono dimensionalmente limitate a 128 pixel per lato; queste vengono quindi lette da una memoria *RAM* composta da 2^{16} celle da 8 bit ciascuna. Questa configurazione è in grado di registrare il numero esatto di informazioni necessarie per rappresentare tutti i possibili 256 livelli di grigio di una immagine avente dimensione massima.
Ad ogni pixel viene dedicata una cella di memoria nell'intervallo *[2 , (riga * colonna + 1)]*, gli indirizzi 0 e 1 sono invece riservati per contenere larghezza e altezza dell'immagine in pixel, valori sempre contenuti nell'intervallo *[0 , 128]*.
Di seguito l'interfaccia del componente:

```
  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;
```

Per poter iniziare il processo di elaborazione il componente rimane in attesa di un segnale di **reset**, *i_rst=1*, questo porta la macchina a uno stato di ready, specificato in questa implementazione attraverso lo stato "***IDLE***" di una Macchina a Stati Finiti.
A seguito del segnale di reset, per poter iniziare un'elaborazione, la macchina rimane in attesa che il segnale di **start** venga alzato, quindi *i_start=1*.
Il termine dell'elaborazione viene comunicato dal componente alzando il segnale di **done**, ovvero *o_done=1*.
Per poter trasformare una nuova immagine non sarà necessario un ulteriore segnale di **reset**, il quale si limiterebbe a impostare la macchina nuovamente nello stato di ready, ma sarà sufficiente alzare nuovamente il segnale di **start**.
L'equalizzazione della immagine viene svolta dalla macchina elaborando un pixel alla volta a seguito di una prima lettura integrale dei pixel, necessaria per l'identificazione del massimo e del minimo dei valori all'interno dei dati memorizzati in memoria.
Il valore ottenuto a seguito del processo di equalizzazione verrà salvato nel seguente indirizzo di memoria:

$$1. \quad ind_{dest} = ind_{pixel} + larghezza * altezza - 1$$

Dove ind_{pixel} è l'indirizzo del pixel da equalizzare, *larghezza* è la larghezza dell'immagine, sita all'inirizzo 0 e *altezza* è l'altezza dell'immagine, sita all'indirizzo 1.
Questo ordinamento consente di salvare l'immagine equalizzata negli indirizzi di memoria adiacenti a quelli della immagine originale.

All'interno del processo viene seguita la seguente procedura: per prima cosa vengono lette larghezza e altezza dell'immagine per conoscere il numero complessivo di pixel da leggere, successivamente vengono letti tutti i pixel per identificarne il massimo e il minimo, necessari per calcolare il valore del pixel modificato, infine viene eseguita un'ultima lettura che si occuperà di equalizzare il pixel scrivendolo sul primo banco di memoria libero dal basso.
Le formule utilizzate per l'equalizzazione sono le seguenti:

$$2. \quad shift = 8 - \text{floor}(\log_2(max - min + 1))$$

$$3. \quad pixel_{new} = \min(255, (pixel_{old} - min) \ll shift)$$

Dove *floor* è la funzione che ritorna la parte intera del valore, *max* e *min* sono il massimo e il minimo dei pixel nella immagine, $pixel_{old}$ è il valore del pixel originale e $pixel_{new}$ quello del pixel equalizzato.

A completamento della descrizione viene riportato un esempio di elaborazione in una immagine 2x1.
In tabella vengono riportati i valori in memoria della immagine avente un pixel di valore 200 e un altro di valore 50:

- Viene calcolato il numero di pixel dell'immagine moltiplicando tra loro il valore di larghezza e altezza agli indirizzi 0 e 1.
- In seguito vengono letti tutti i pixel individuando con un confronto il valore massimo e il minimo, in questo caso il massimo risulta essere 200 e il minimo 50.
- Applicando la *2.* viene calcolato il valore di shift da applicare nella *3.* per poter equalizzare il pixel.
  $shift = 8 - floor(\log2(200 - 50 + 1)) = 1$
- Si effettua dunque un'ultima lettura dei pixel per ottenere con la *3.* il valore di $pixel_{new}$ da posizionare nella cella ottenuta con la *1.*
  $pixel_1 = \min(255,(200 - 50) \ll 1) = \min(255,300) = 255$
  $pixel_2 = \min(255,(50 - 50) \ll 1) = \min(255,0) = 0$

Indirizzo	Valore
0	2
1	1
2	200
3	50
4	-
5	-
6	-

Stato della memoria prima della elaborazione.

Indirizzo	Valore
0	2
1	1
2	200
3	50
4	255
5	0
6	-

Stato della memoria al termine della elaborazione.

ARCHITETTURA

L'architettura realizzata è composta da un modulo contenente una Macchina a Stati Finiti realizzata tramite un unico processo avente nella sua lista di sensitività il segnale di clock, *i_clk*, e il segnale di reset, *i_rst*.
La macchina è composta da 9 stati e altrettanti segnali descritti di seguito:

index : `std_logic_vector(15 downto 0)`
Segnale a 16 bit che memorizza il prossimo indirizzo di memoria da leggere.

max : `integer RANGE 255 downto 0`
Contiene il massimo valore del pixel, viene iniziallizzato a 0 nello stato di IDLE e aggiornato durante la prima lettura dei pixel.

min : `integer RANGE 255 downto 0`
Contiene il valore del pixel più piccolo, viene iniziallizzato a 255 nello stato di IDLE e aggiornato durante la prima lettura dei pixel.

shift : `integer RANGE 8 downto 0`
Contiene il numero di bit da shiftare per equalizzare il pixel con la equazione *3.*

shifted_value : `std_logic_vector(15 downto 0)`
Contiene il valore codificato in binario del pixel dopo lo shift. Dal momento che lo shift massimo è di 8 bit e il valore del pixel è contenuto in 8 bit i dati del pixel temporaneo vengono memorizzati in questo vettore da 16 bit e sottoposti a una successiva verifica.

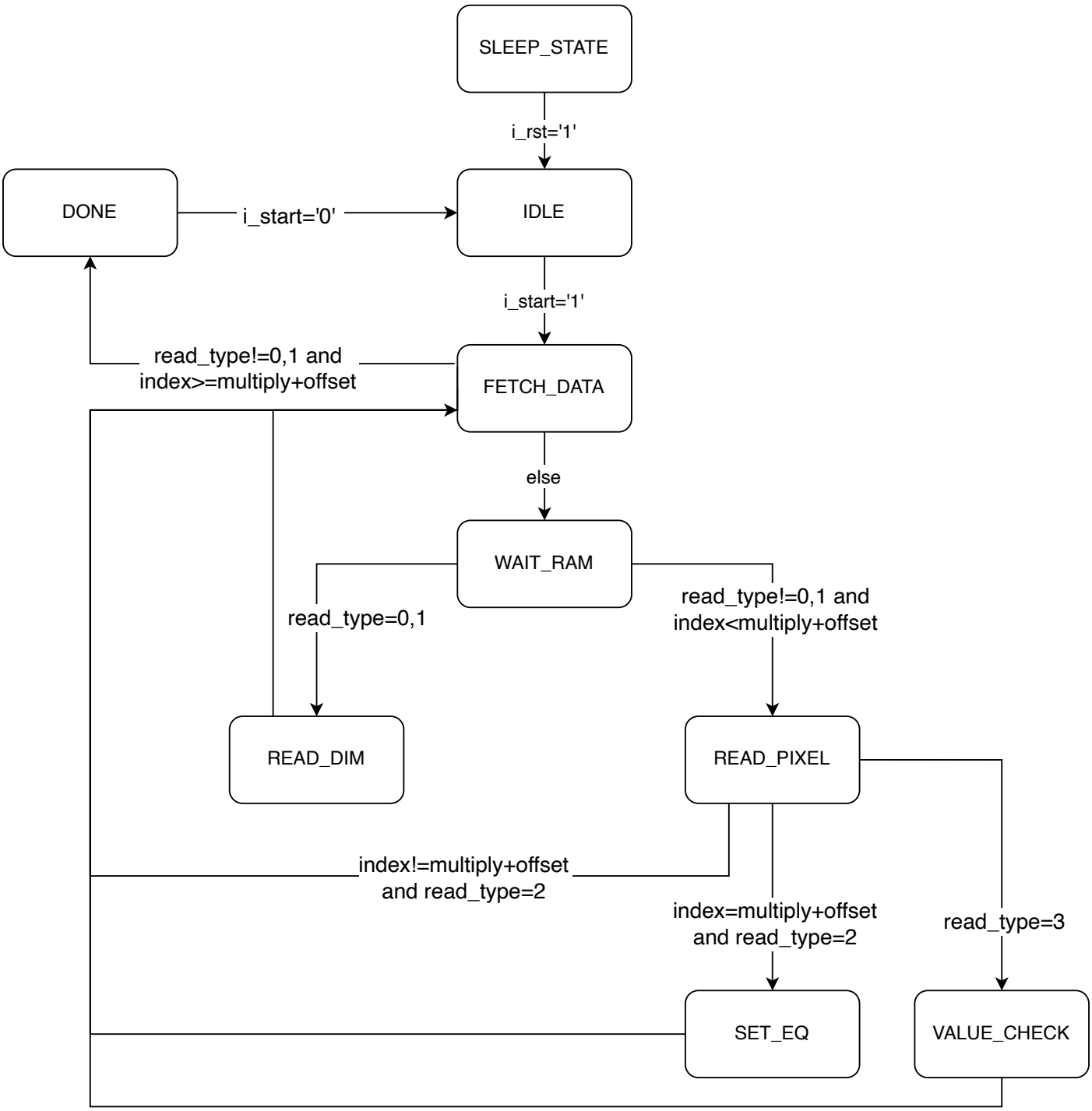
curr_state : `state_type`
Contiene lo stato attuale della macchina. Il segnale è di tipo *state_type*, un tipo di dato contenente tutti i 9 possibili stati della macchina. Viene aggiornato negli stati per consentire la gestione ordinata del controllo di flusso degli stati.

next_state : `state_type`
Contiene lo stato successivo della macchina, necessario quando per accedere alla memoria si deve attendere un ciclo di clock passando dallo stato di ***WAIT RAM***.

reading_type : `integer RANGE 3 downto 0`
Valore utilizzato durante il ciclo di lettura per determinare il tipo di lettura che deve essere effettuata dalla memoria, in particolare vengono distinti i seguenti tipi: colonna, riga, pixel durante la ricerca degli estremi e pixel per la equalizzazione.

multiply : `integer RANGE 16384 downto 0`
Contiene il valore dei pixel dell'immagine, ottenuto moltiplicando i primi due byte della memoria.

Di seguito il diagramma degli stati sviluppato per la macchina:



Descrizione degli stati:

1. **SLEEP STATE**: Stato di attesa del primo segnale di ***RESET*** necessario per l'inizio delle elaborazioni.

2. **IDLE**: Stato in cui la macchina è pronta ed è in attesa dell'alzata del segnale di start per poter passare agli stati successivi. In questo vengono iniziallizzati i valori di *max* a 0, *min* a 255, *index* a 2 per la lettura del primo pixel e *reading_type* a 0 per la lettura della larghezza della immagine.

3. **FETCH DATA**: Stato di preparazione alla lettura, il tipo di lettura effettuato dipende dal valore di *reading_type* e di *index*; a seconda di questi lo stato successivo viene impostato a ***READ DIM*** se è una lettura delle dimensioni dell'immagine, ***READ PIXEL*** se è una lettura di un pixel generico. Nel caso in cui *reading_type* non sia dedicato alla lettura delle dimensioni della immagine e *index* sia minore del numero dei pixel dell'immagine + 2 la macchina procederà alla lettura di questi attraverso lo stato di ***READ PIXEL***, in caso contrario la computazione sarà considerata terminata e lo stato successivo verrà impostato a ***DONE***. In tutti casi ad eccezione dell'ultimo vengono inoltre impostati i segnali necessari per la lettura in memoria passando dallo stato di ***WAIT RAM*** in attesa di ottenere il dato in lettura e impostando lo stato prossimo al tipo di operazione da svolgere una volta ottenuto il dato.

4. **WAIT RAM**: Stato di attesa del dato richiesto dalla RAM, assegna allo stato corrente il valore dello stato prossimo impostato dal comando di ***FETCH DATA*** precedente.

5. **READ DIM**: Stato di lettura delle dimensioni della immagine, in caso di lettura di colonna, con *reading type = 0* questo verrà impostato a 1 e il valore della colonna sarà salvato in *multiply*; in caso di lettura di riga questa verrà moltiplicata per il valore della colonna e il risultato verrà assegnato a *multiply*; *reading type* verrà quindi impostato a 2 per procedere alla lettura del massimo e del minimo dei pixel in memoria.

6. **READ PIXEL**: Stato di lettura del pixel memorizzato all'indirizzo indicato da *o_address*. Con *reading type* pari a 2 si procede alla ricerca degli estremi per confronto con i segnali di min e di max ritornando a ***FETCH DATA***. Durante la prima lettura dell'ultimo pixel lo stato corrente verrà impostato su ***SET EQUALIZATION***. Durante il secondo ciclo di lettura dei pixel, con *reading type* pari a 3, lo stato successivo verrà impostato a ***VALUE CHECK***.

7. **SET EQUALIZATION**: Stato in preparazione all'equalizzazione del pixel, *index* viene nuovamente portato al primo pixel valido dell'immagine, *reding type* viene portato a 3 e viene calcolato il valore dello shift. Lo stato successivo viene infine portato a ***FETCH DATA***.

8. **VALUE CHECK**: Stato dedicato al controllo del risultato, in particolare verifica che il valore a seguito dello shift non sia superiore a 255 e in tal caso viene impostato a 255. Successivamente i segnali di *write enable* e *read enable* vengono impostati a 1 per la scrittura in memoria nella cella calcolata con la formula *1.*, rimandando infine a ***FETCH DATA***.

9. **DONE**: Stato di termine elaborazione, la macchina rimane in questo finché non vengono impostati gli stati per la successiva elaborazione. Una volta che *i_start* viene abbassato anche *o_done* viene impostato a 0, rimandando allo stato di ***IDLE***.

RISULTATI SPERIMENTALI

Il componente ha risposto positivamente ad ogni tipologia di test sia tramite simulazioni di tipo Behavioural sia di tipo Post Synthesis Functional. Tra le test bench utilizzate sono state impiegate diverse immagini randomizzate e alcuni casi limite significativi, in particolare:

- Immagine con una dimenzione pari a 0: l'elaborazione termina correttamente senza necessità che vengano inseriti controlli particolari nel componente.
- Immagine 0x0: caso particolare del precedente che costituisce inoltre il limite temporale minimo di elaborazione della simulazione che è stata completata in 6,32 μs in Post Synthesis Functional.
- Immagine 128x128: questa casistica testa la gestione del massimo valore di multiply ed è stata completata in circa 1720 μs in Post Synthesis Functional.
- Gestione di più immagini con reset asincrono: il componente reagisce opportunatamente; nel ciclo di clock in cui viene letto un segnale alto su *i_rst* lo stato viene riportato ad ***IDLE*** e l'elaborazione prosegue regolarmente.
- Gestione degli estremi del dominio dei pixel: di fronte a una immagine monocromatica di 255 pixel la risultante immagine equalizzata è composta da soli 0, come per la *3.*, viceversa ponendo un massimo a 255 e un minimo a 0 il valore del segnale di shift viene correttamente calcolato a 0 e l'immagine equalizzata risulta uguale alla originale, come si prevedeva sempre dalla *3.*

Al termine della prima stesura funzionante della macchina si è proceduto a compiere una fase di ottimizzazione concentrata principalmente nella riduzione dei segnali superflui ai fini della corretta elaborazione della immagine e dalla riduzione degli stati attraverso una più attenta gestione del segnale di *reading type*.

A seguito delle suddette operazioni il numero di FlipFlop necessari per sintetizzare la board riportati dal comando "*report_utilization*" di Vivado è diminuito circa del 10% rispetto alla prima versione del programma.

Di seguito alcuni dettagli significativi sull'utilizzo di memoria e sul ritardo di segnale ricavati dal programma di sintesi attraverso i comandi di *report timing* e di *report utilization*.

```
Data Path Delay:        4.618ns
logic 3.451ns (74.727%)
route 1.167ns (25.273%)
+-------------------------+------+
|        Site Type        | Used |
+-------------------------+------+
| Slice LUTs*             |  172 |
|   LUT as Logic          |  172 |
|   LUT as Memory         |    0 |
| Slice Registers         |  107 |
|   Register as Flip Flop |  107 |
|   Register as Latch     |    0 |
+-------------------------+------+
```

CONCLUSIONI

Il progetto è stato sviluppato in maniera modulare a partire dal disegno di una bozza della macchina a stati finiti. L'approccio top-down è stato in grado di evidenziare le maggiori sfide da superare nella trasposizione di pseudo codice in codice VHDL funzionante; questo aspetto è stato presente ad ogni ciclo di produzione della macchina, da quando sono stati scritti semplici moduli di codice funzionante fino alla versione ottimizzata in Post Sintesi.
Per raggiungere una versione sintetizzabile, una volta creata una versione funzionante in Behavioural, è stata necessaria una analisi più critica del codice che ha portato a una sua semplificazione che conseguentemente ha permesso di semplificare anche la fase di debugging necessaria per produrre un programma sintetizzabile.
Questa parte del progetto si è rilevata essere la più impegnativa in quanto si è reso necessario abbandonare una logica ad alto livello derivata dalla conoscenza di linguaggi di natura procedurale come *C* e ad oggetti come *Java* per doversi confrontare con le caratteristiche tipiche di un progetto a basso livello che richieda un attento esame anche a livello di segnali.
A tal proposito, per una buona parte della fase di debugging, è stato necessario un confronto visualizzando i sengali con il prezioso aiuto del Waveform Viewer del tool di sintesi; questo ha consenstito di affrontare il problema non più dal semplice punto di vista della risoluzione di un algoritmo con del codice ma dal punto di vista della progettazione logica di un sistema digitale, permettendo di comprendere i vantaggi e gli svantaggi che comporta questo tipo di processo di sviluppo.
Sempre attraverso l'uso del tool di sintesi si è stati inoltre in grado di avere una tangibile testimonianza degli effetti ottenuti dal processo di ottimizzazione della macchina che hanno portato a una riduzione del criterio di costo a livello di elementi di memoria.

Nel complesso gli autori si ritengono soddisfatti del lavoro prodotto per lo svolgimento del seguente progetto dal momento che la sua creazione ha consentito di applicare diversi concetti trasmessi dal corso di Reti Logiche e da altri insegnamenti caratterizzanti acquisiti durante il corso di questa laurea triennale.