Prova Finale di Reti Logiche

Sintesi di un componente per la ricerca dell'area del rettangolo minimo che circoscrive una figura di interesse.

Composizione del Gruppo

Il progetto è stato svolto dal gruppo composto dagli studenti:

- Gargano Jacopo Pio - Matricola 847989 - Codice Persona 10516854
- Gioiosa Davide - Matricola 848123 - Codice Persona 10503787

Introduzione

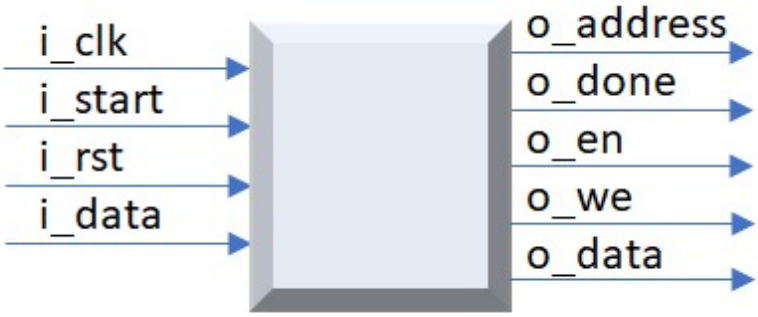
Il progetto del corso di Reti Logiche assegnatoci consiste nella creazione e sintesi di un componente hardware in grado di calcolare, partendo da un'immagine in scala di grigi che rispetta il formato assegnato nella documentazione allegata "Prova Finale di Reti Logiche", l'area del rettangolo minimo che circoscrive totalmente una figura di interesse presente nell'immagine in ingresso. In questo documento si farà riferimento a questo rettangolo come "rettangolo minimo". La figura di interesse è individuata da tutti quei pixel il cui valore è maggiore o uguale alla soglia memorizzata nella memoria RAM.
Il componente è descritto in linguaggio VHDL. È stato utilizzato Vivado (Xilinx) come ambiente di sviluppo.

Risvolti Pratici e Obiettivi

Abbiamo riflettuto sui possibili utilizzi pratici del componente e abbiamo pensato che esso potesse essere utilizzato nel campo del riconoscimento facciale, con opportune piccole modifiche. Infatti, l'area o le coordinate dei vertici del rettangolo minimo che circoscrive il volto di una persona possono essere utili per determinare la vicinanza del volto dalla fotocamera.
Per questo motivo, abbiamo scelto di creare un componente che fosse molto veloce e performante, ponendo in secondo piano l'area da esso occupata.

Approccio Iniziale

Prima di passare allo studio e all'implementazione del componente in termini di codice, abbiamo studiato l'IDE e il linguaggio di programmazione, sviluppando semplici componenti al fine di avere una comprensione più approfondita dell'ambiente di sviluppo.
Come primo passo, abbiamo disegnato il componente identificando i segnali di ingresso (*i_clk, i_start, i_rst, i_data)* e di uscita *(o_address, o_done, o_en, o_we, o_data*). Abbiamo anche disegnato la memoria RAM per individuare gli indirizzi di memoria da cui leggere (2 – *colonne*, 3 – *righe*, 4 – *soglia*, 5 – primo pixel dell'immagine...) e in cui scrivere l'area finale (0 – *LSB*, 1 – *MSB*).
Dopo un attento brainstorming delle possibili soluzioni, abbiamo deciso di realizzare il componente tramite una macchina a stati finiti (FSM) costituita da un solo processo. Abbiamo dunque disegnato un primo diagramma degli stati della macchina, tenendo a mente che il segnale di clock fosse il responsabile del cambiamento di stato.

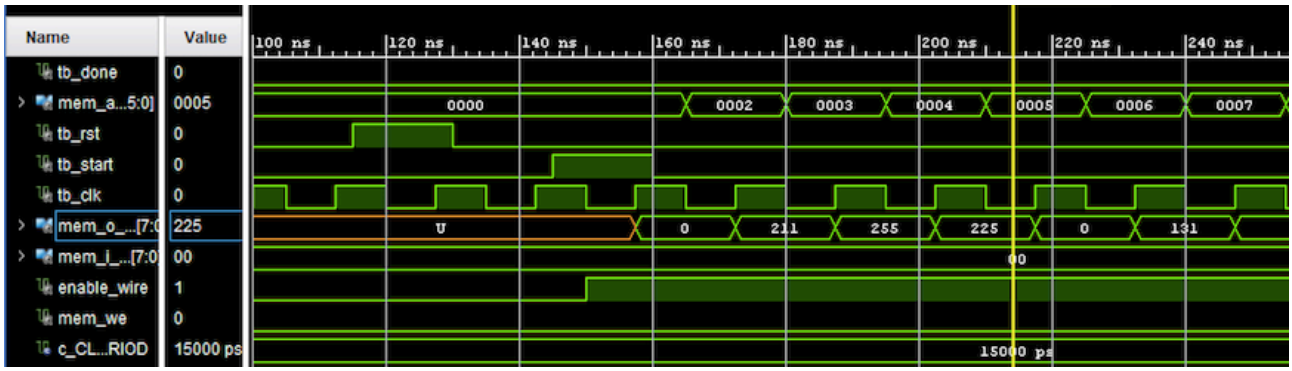


Disegno del componente con ingressi e uscite.

Per implementare la FSM abbiamo scelto di avere un solo processo la cui lista di sensitività fosse costituita da *i_clock*, i_*rst* e *i_start* in modo tale da eseguirlo ogni volta che questi segnali cambiassero livello logico. Abbiamo utilizzato un costrutto *switch* per implementare il comportamento del componente a seconda dello stato della macchina, realizzato tramite il segnale *state*.

Analisi delle Operazioni di Assegnamento, Lettura e Scrittura in Memoria

Utilizzando il grafo della FSM e il Waveform Viewer di Vivado, abbiamo analizzato le procedure e il numero di cicli di clock necessari per effettuare le operazioni di lettura e scrittura in memoria e di assegnamento di un valore a un segnale.



Esempio di Debugging tramite Waveform Viewer.

Abbiamo quindi modificato il diagramma degli stati della FSM in modo tale che rispecchiasse il comportamento del componente realizzabile in VHDL. Infatti, analizzando il componente in pre-sintesi, se lo *switch* è implementato con il clock sul fronte di salita, allora sono necessari due cicli di clock per leggere un dato, corrispondenti a due transizioni tra tre stati diversi. Nel primo stato si assegna al segnale *o_data* l'indirizzo di memoria che si vuole leggere. Nel secondo stato non si esegue nessuna operazione. Nel terzo si assegna il valore in ingresso contenuto nel segnale *i_data* al rispettivo segnale. Per la scrittura di un dato in memoria, abbiamo osservato che è necessario invece un solo stato in cui si implementa l'operazione: lo stato intermedio di no-op non è necessario e il risultato dell'operazione è effettivo dopo un ciclo di clock, come per le operazioni di assegnamento.

Principali Procedure della FSM

La macchina è descritta dalle seguenti procedure principali:

- Inizialmente si aspetta un segnale di reset e di start per avviare la macchina e la funzione da essa eseguita.
- Una volta ricevuto un segnale di start, si acquisisce il numero di colonne, di righe e la soglia. Questi valori vengono salvati nei rispettivi segnali *colonne*, *righe*, *soglia*.

```
when S4 =>
   o_address <= soglia_ram;
   if(i_data = zero_eightbits)
   then state <= S9;
   else
       righe <= i_data;
       updated_righe <= i_data;
       state <= S5;
   end if;
```

Inizio procedura di lettura da memoria RAM per il valore della soglia.
Termine della procedura di lettura per il numero di righe.

- Viene quindi eseguito un algoritmo per trovare le coordinate dei vertici del rettangolo minimo. Questi valori sono aggiornati e salvati nei segnali *colonne_left*, *colonne_right*, *righe_up* e *righe_down*. Si rimanda alla sezione '*Algoritmi di Ricerca dei Vertici*' per una descrizione dettagliata dell'algoritmo utilizzato e di quelli considerati.
- Una volta trovati i valori della procedura del punto precedente, si procede calcolando base e altezza del rettangolo minimo e successivamente la sua area.
- Infine, l'area viene scritta in memoria, agli indirizzi 0 e 1.

```
-- S8: stato in cui si calcola l'area del rettangolo minimo
when S8 =>
   area <= base * height;
   o_we <= '1';
   state <= S9;

-- S9: stato in cui si scrivono i LSB dell'area del rettangolo minimo
when S9 =>
   o_address <= LSB_ram;
   o_data <= area(7 downto 0);
   state <= S10;
```

Calcolo dell'area del rettangolo minimo e scrittura di essa in memoria RAM.

- La macchina torna in uno stato in cui è pronta per rieseguire il calcolo.

Le diverse procedure della FSM sono state implementate utilizzando costrutti *if-else* e vari segnali *std_logic* a un bit per dei *flag*.

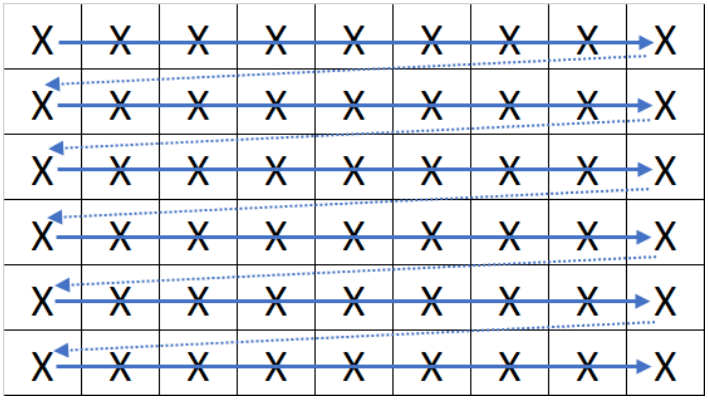
Algoritmi di Ricerca dei Vertici

Analisi di Complessità

Analizzando il problema della ricerca del rettangolo minimo che circoscrive totalmente una figura di interesse siamo giunti alla conclusione che la complessità minima necessaria è *O(n)*, con n pari al numero di pixel dell'immagine. Tuttavia, algoritmi differenti, pur avendo la stessa complessità temporale, porterebbero ad avere differenze sostanziali in merito al tempo di esecuzione. Infatti, statisticamente la posizione dei pixel con valore maggiore o uguale alla soglia nelle immagini di ingresso può risultare vantaggiosa. Abbiamo analizzato tre possibilità: Scansione Lineare, Ricerca Spirale e Ricerca Random.

Scansione Lineare

L'algoritmo si basa sulla scansione di tutte le celle di memoria che fanno parte dell'immagine in ingresso, aggiornando ogni volta le coordinate dei vertici del rettangolo minimo in base alla soglia.

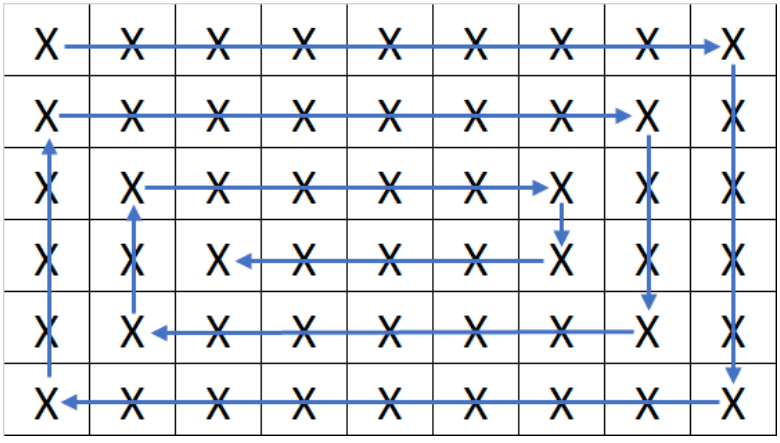


Scansione Lineare

La complessità di questo algoritmo è lineare: il numero di operazioni è sempre pari al numero di pixel che costituiscono l'immagine.

Ricerca Spirale

L'algoritmo esegue una ricerca spirale partendo dal primo pixel in alto a sinistra nell'immagine. Il primo passo, rappresentato dal percorso azzurro in figura, consiste nella scansione del rettangolo più esterno. Tramite l'utilizzo dei segnali *left*, *right*, *up* e *down*, oltre i già esistenti *colonne_left*, *colonne_right*, *righe_up* e *righe_down*, si procede all'aggiornamento dei vertici del rettangolo minimo, se necessario. In particolare, i valori di questi segnali aggiuntivi indicano i lati orizzontali e verticali che non possiedono pixel con valori maggiori o uguali alla soglia. Ogni volta che sono aggiornati, riducono l'immagine in ingresso fino ad arrivare al rettangolo minimo cercato.



Ricerca Spirale

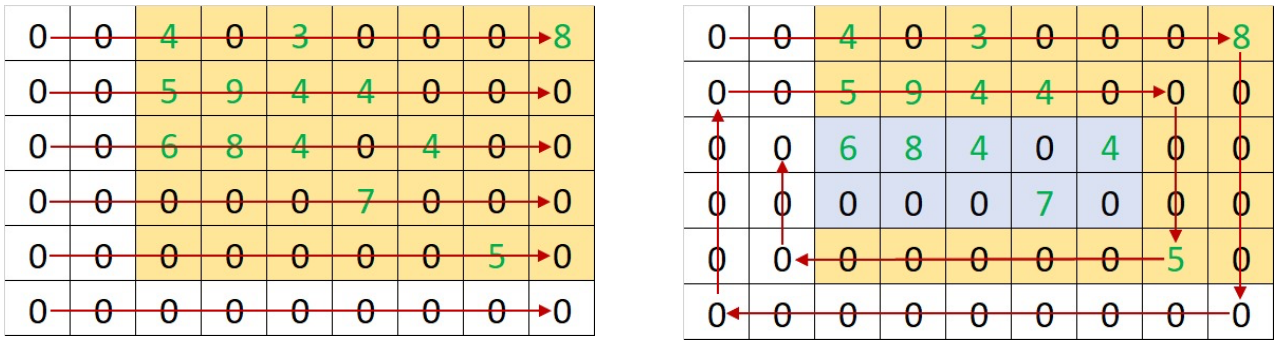
La complessità di questo algoritmo è lineare: il numero delle operazioni varia a seconda della disposizione dei pixel che costituiscono la figura di interesse. Infatti, più sono vicini al contorno dell'immagine in ingresso e meno operazioni verranno effettuate dalla FSM. Il numero di operazioni effettuate è quasi sempre molto minore rispetto alla scansione lineare e alla ricerca random. Il caso in cui le tempistiche si avvicinano senza però superare quelle degli altri due algoritmi è quando la figura di interesse è al centro dell'immagine in ingresso ed è molto piccola rispetto a tutta l'immagine.

Ricerca Random

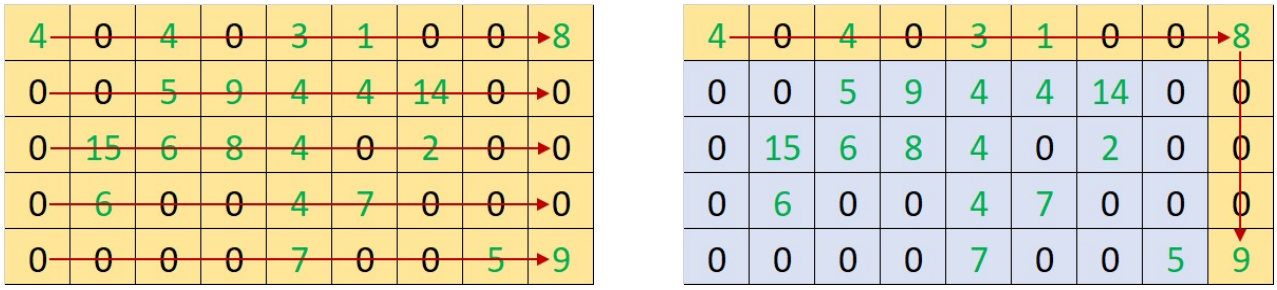
L'algoritmo esegue una ricerca casuale nelle celle di memoria che costituiscono l'immagine. Per ogni pixel analizzato, aggiorna le coordinate dei vertici del rettangolo minimo fino ad ottenere i dati necessari per comporre un rettangolo. Una volta trovato, si prosegue la ricerca casuale per i soli elementi al di fuori del rettangolo che viene aggiornato se ne esiste uno di area maggiore.
La complessità di questo algoritmo è lineare: il numero delle operazioni varia a seconda della disposizione dei pixel che costituiscono la figura di interesse e della ricerca casuale stessa.
Abbiamo considerato che le tempistiche sarebbero statisticamente inferiori rispetto all'algoritmo di scansione lineare, tuttavia non presenterebbero un miglioramento notevole. Il numero di Look Up Tables e di Flip Flops aumenterebbe considerevolmente poiché bisognerebbe salvare l'indirizzo dei pixel già analizzati. Abbiamo dunque scartato questo algoritmo senza implementarlo.

Esempi di Confronto tra Scansione Lineare e Ricerca Spirale

Di seguito due esempi sulle differenze di funzionamento dei due algoritmi implementati, evidenziando in giallo il rettangolo minimo e in azzurro le caselle appartenenti ad esso che non sono visitate dall'algoritmo di ricerca spirale.



Esempio 1



Esempio 2

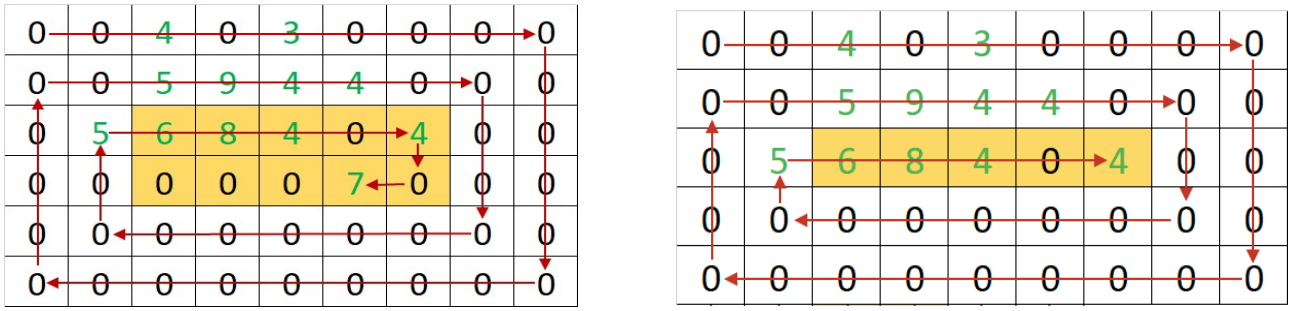
Post-sintesi

Una volta ottenuti esiti positivi nella pre-sintesi, abbiamo sintetizzato il componente e avviato la simulazione in post-sintesi. Tuttavia, anche se il componente veniva sintetizzato, abbiamo avuto risultati negativi con il programma scritto fino a quel momento.
Tramite l'utilizzo del Debugging e del Waveform Viewer sono stati riscontrati problemi che non emergevano nell'analisi fatta finora in pre-sintesi. Infatti, abbiamo osservato la mancata sintesi di determinati flag e un differente numero di cicli di clock necessari a leggere o scrivere un dato rispetto al componente in pre-sintesi.

È stato inizialmente necessario modificare il codice eliminando i flag che generavano errori sostituendo la loro funzione con nuovi stati o con costrutti *if-else*.
Successivamente abbiamo introdotto nel diagramma degli stati della FSM degli stati di no-op al fine di ottenere una sincronizzazione sulle operazioni di assegnamento e di lettura e scrittura in memoria.
In questo modo abbiamo ottenuto un risultato uniforme e corretto in entrambe le analisi di pre-sintesi e post-sintesi.

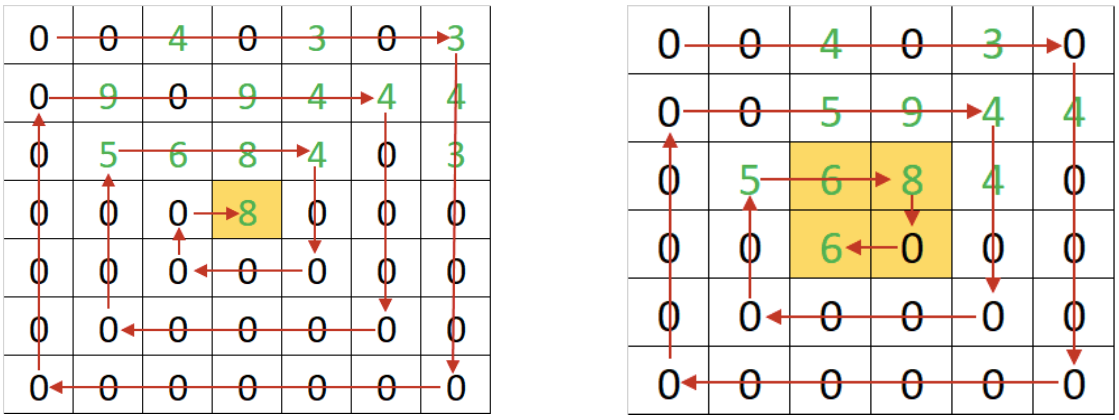
Testbench

Ci sono stati forniti quattro testbench per verificare il comportamento del nostro componente. Simulando la FSM in pre-sintesi abbiamo appurato il suo comportamento, ottenendo risultati positivi.
Al fine di garantire la correttezza e completezza del programma ottenuto, abbiamo scritto nuovi testbench. Ci siamo concentrati soprattutto sullo studio dei casi limite, come ad esempio immagini con area pari a 1, con il numero massimo di righe e di colonne, con soglia pari a 0, con lo stesso numero di righe e colonne.
Abbiamo studiato tutti i casi possibili, che ci sono venuti in mente mentre implementavamo l'algoritmo di ricerca spirale, ragionando sulla parità o disparità di righe e colonne, ottenendo quindi i casi minimi a cui si riducono le immagini di ogni forma e dimensione.
In particolare, se in ingresso si ha un'immagine rettangolare, essa si ridurrà a un rettangolo con una dimensione qualsiasi e l'altra che può essere pari a 1 o pari a 2.



Alcuni casi minimi a cui si riduce un'immagine rettangolare.

Se invece in ingresso si ha un quadrato, allora esso può ridursi a un singolo pixel oppure a un quadrato di lato pari a 2.



Casi minimi a cui si riduce un'immagine quadrata.

Descrizione di Testbench Personali

Abbiamo raccolto quattro dei numerosi testbench che abbiamo scritto per verificare la correttezza del componente.

Testbench Personal 1: 3 colonne, 255 righe, soglia pari a 1. Abbiamo disposto i pixel con valore superiore alla soglia sui bordi della figura in ingresso. In questo modo, con l'algoritmo di ricerca spirale, la procedura impiega circa la metà dei passi per terminare, rispetto al caso di scansione lineare.

Testbench Personal 2: 255 colonne, 255 righe, soglia pari a 1. Abbiamo disposto i pixel con valore superiore alla soglia nell'angolo in alto a sinistra e in quello in basso a destra della figura in ingresso. In questo modo, il rettangolo minimo è vincolato a questi due pixel. Ne abbiamo disposti altri in maniera casuale all'interno di esso. Quindi, con l'algoritmo di ricerca spirale, la procedura termina una volta letti questi due valori, invece di continuare a leggere anche gli altri come nel caso di scansione lineare.

Testbench Personal 3: 211 colonne, 255 righe, soglia pari a 225. La soglia è molto alta, e soltanto il valore di alcuni pixel la supera.

Testbench Personal 4: 255 colonne, 255 righe, soglia pari a 1. L'unico pixel che supera la soglia è posto al centro. Dunque, entrambi gli algoritmi impiegheranno lo stesso tempo per trovare il risultato.

Miglioramenti

Casi Immediati

Grazie ai testbench per garantire la completezza della FSM, ci siamo accorti di alcuni casi in cui la macchina potesse terminare direttamente la procedura senza dover eseguire altre operazioni, scrivendo quindi direttamente il risultato in memoria, velocizzando così la risposta del componente. Di seguito alcuni casi osservati:

- Se la soglia è pari a 0, allora l'area del rettangolo minimo corrisponderà all'area di tutta l'immagine in ingresso. Sono necessari in questo caso soltanto il numero di colonne e di righe.
- Se il numero di colonne o di righe è pari a 0, allora l'area del rettangolo minimo è pari a 0.

Fronte di Discesa del Clock

Siamo riusciti ad ottenere un ottimo risultato quando abbiamo provato ad eseguire le transizioni di stato e le relative operazioni durante il fronte di discesa del clock, invece del fronte di salita. In questo modo abbiamo potuto eliminare gli stati di no-op ed effettuare tutte le operazioni relative alla memoria con lo stesso numero di cicli di clock sia per la simulazione in pre-sintesi che in post-sintesi. È quindi necessario un solo ciclo di clock e due stati per leggere un dato e assegnarlo a un segnale, mentre un ciclo di clock e uno stato per scriverlo in memoria o per eseguire operazioni di assegnamento.

```
-- S2: stato in cui si inizia la lettura del numero di colonne
when S2 =>
   o_address <= colonne_ram;
   state <= S3;

-- S3: stato in cui si inizia la lettura del numero di righe e si assegna i_data alle colonne
when S3 =>
   o_address <= righe_ram;
   if(i_data = zero_eightbits)
   then state <= S9;
   else
       colonne <= i_data;
       updated_colonne <= i_data;
       state <= S4;
   end if;
```

Stati S2 e S3: lettura del numero di colonne, utilizzando il fronte di discesa del clock.

Tramite l'utilizzo del fronte di discesa del clock abbiamo circa dimezzato i tempi di esecuzione della procedura poiché abbiamo eliminato uno stato di no-op per ogni operazione di lettura o scrittura.

	Fronte di Salita	Fronte di Discesa
Testbench 3	5,422,500	2,827,500
Testbench Personal 1	23,332,500	11,782,500

Confronto tra l'utilizzo del fronte di salita e il fronte di discesa del clock.

Commenti e Costanti

Al fine di aumentare la leggibilità del codice, abbiamo aggiunto delle costanti con un nome che ricorda la loro funzione. Un esempio è dato dalle costanti che assumono il valore dell'indirizzo di memoria al quale è salvato il rispettivo valore: *colonne_ram*, *righe_ram* e *soglia_ram* hanno il valore degli indirizzi 2, 3 e 4. In questo modo, se si vuole cambiare la posizione dei tre valori in memoria basterà modificare soltanto queste costanti e niente altro all'interno dell'implementazione della FSM.

Abbiamo aggiunto dei commenti nel codice per rendere più facile la sua comprensione e continuità. Infatti, se in futuro dovranno essere apportate modifiche al funzionamento e all'implementazione della FSM, la sua comprensione risulterà più agevole grazie alle spiegazioni date per ogni stato, segnale e per tutte le altre operazioni effettuate.

Prestazioni Temporali

	Scansione Lineare	Ricerca Spirale
Testbench 1	2,827,500	2,827,500
Testbench 2	2,827,500	307,500
Testbench 3	2,827,500	1,582,500
Testbench 4	2,827,500	2,827,500
Testbench Personal 1	11,782,500	6,862,500
Testbench Personal 2	975,682,500	7,957,500
Testbench Personal 3	807,382,500	807,382,500
Testbench Personal 4	975,682,500	975,682,500

Prestazioni Spaziali e Massima Frequenza di Funzionamento

	Scansione Lineare	Ricerca Spirale
Look Up Tables	192	517
Flip-Flops	152	213
TOTALE	344	730
Max Frequenza	172 MHz	158 MHz

Conclusioni

La scelta sull'utilizzo dell'algoritmo di ricerca spirale rispetto a quello di scansione lineare è stata dettata dall'obiettivo iniziale di ottenere tempistiche ottime a scapito di un utilizzo maggiore di area. Risultano infatti 730 tra LUT e FF contro 344. L'aumento non è significativo, considerando che in termini temporali si ottiene un notevole guadagno. Abbiamo quindi preferito l'algoritmo veloce di ricerca spirale. L'area risulta implementabile in uno scenario simile a quello descritto nella sezione "*Risvolti Pratici e Obiettivi*".

L'algoritmo scelto ha prestazioni ottimali nel caso in cui la figura di interesse è vicina al bordo dell'immagine in ingresso. Altrimenti, nel caso pessimo, impiega lo stesso tempo dell'algoritmo di scansione lineare.

Abbiamo testato il componente e osservato che esso può funzionare con un clock di 6,3 ns che corrisponde a una frequenza di 158 MHz. La frequenza è poco più bassa rispetto all'algoritmo di scansione lineare, in quanto l'algoritmo di ricerca spirale è più complesso.

Infine, il componente risulta funzionante per ogni tipo di immagine in ingresso che rispetta le specifiche, e la correttezza della funzione svolta è stata verificata.