**Lavoro di gruppo di:**

**848834 - 10523360 - Ali El Wahsh  
847218 - 10520143 - Mathyas Giudici**

DOCUMENTAZIONE ALLEGATA AL PROGETTO DI RETI LOGICHE

[I. PREFAZIONE 2](#_Toc507665290)

[II. SEGNALI GESTIONE STATI FSM 2](#_Toc507665291)

[III. SEGNALI GESTIONE MEMORIA e contatori 2](#_Toc507665292)

[IV. SEGNALI DI SUPPORTO 3](#_Toc507665293)

[V. DESCRIZIONE STATI 3](#_Toc507665294)

[A. RESET 3](#_Toc507665295)

[B. INIZIALE 3](#_Toc507665296)

[C. CONFRONTO 3](#_Toc507665297)

[D. STATO\_MOLTIPLICA 3](#_Toc507665298)

[E. SALVA 3](#_Toc507665299)

[F. ASPETTA 4](#_Toc507665300)

[VI. implementazione architettura 4](#_Toc507665301)

[VII. TESTBENCH E TESTING 7](#_Toc507665302)

[VIII. SCHEMA LOGICO 9](#_Toc507665303)

# PREFAZIONE

Al fine di poter implementare una rete logica in grado di calcolare l’area del rettangolo minimo che circoscrive la figura in memoria (dato un valore di soglia) si decide di sviluppare il progetto verso un’ottimizzazione e massimizzazione della frequenza.

Vi è da considerare che il problema presentato ha una complessità θ(n), dove n è il numero di elementi della memoria RAM in cui vi è salvata l’immagine. Infatti non vi è modo di determinare quale sia il rettangolo più grande senza aver scandito almeno una volta per intero l’immagine.

*Nell’algoritmo da noi implementato vi è una scansione sequenziale della memoria; tale algoritmo implica che bisogna arrivare fino all’ultimo indirizzo disponibile per aver effettuato tutti i confronti, di soglia, necessaria ad aver trovato il rettangolo più grande. Un altro algoritmo potrebbe essere quello di visita “a spirale”; questo algoritmo ridurrebbe sensibilmente il tempo nel caso medio (rettangolo “aderente” al bordo) dove si avrebbe una complessità logaritmica; tuttavia il caso pessimo (rettangolo non esistente) implicherebbe lo stesso una visita di tutta la memoria e complessità lineare.*

Se da un lato si ha un vantaggio nella velocità della risposta del sistema si perderà dal punto di vista di dimensionale; il circuito sintetizzato a partire dal codice VHDL occuperà una considerevole area.

# SEGNALI GESTIONE STATI FSM

In prima istanza si decide di definire un nuovo tipo “stato”.

*type state\_type is (reset,iniziale,confronto,stato\_moltiplica,salva,aspetta);*

In questo modo si possono definire i seguenti segnali:

*signal stato\_corrente, stato\_prossimo : state\_type;*

il primo rappresenta lo stato in cui si trova la macchina, mentre il secondo rappresenta lo stato in cui la macchina si troverà al successivo fronte di salita.

# SEGNALI GESTIONE MEMORIA e contatori

*signal soglia, colonne, righe: std\_logic\_vector(7 downto 0);*

Nello stato “iniziale” della macchina vengono in questi segnali vengono caricati i rispettivi valori salvati in memoria. Questi segnali non verranno modificati per tutta l’esecuzione.

*signal nord, sud, west, est: std\_logic\_vector(7 downto 0);*

Questi segnali sono usati per salvare le coordinate massime; sono di supporto per il calcolo dell’area del rettangolo.

*signal coordc, coordr: std\_logic\_vector (7 downto 0);*

Rappresentano rispettivamente la coordinata di colonna e coordinata di riga su cui si sta lavorando durante lo stato “confronto”.

*signal prossimo\_address: std\_logic\_vector (15 downto 0);*

Rappresenta il prossimo indirizzo (di memoria) che verrà caricato in *o\_address* al successivo ciclo di clock.

# SEGNALI DI SUPPORTO

*signal phase: std\_logic\_vector(1 downto 0);*

Questo segnale tiene traccia del caricamento dei valori durante lo stato “iniziale”.

*signal moltiplica: std\_logic\_vector(15 downto 0);*

Dopo lo stato “moltiplica” contiene il risultato del calcolo dell'area del rettangolo.

*signal productphase: std\_logic\_vector (1 downto 0);*

Questo segnale tiene traccia della fase di caricamento in memoria del risultato della moltiplicazione.

# DESCRIZIONE STATI

L’entità project\_reti\_logiche è stata da noi implementata come una macchina a stati finiti (FSM).

La macchina è articolata in sei stati:

## RESET

Questo è lo stato iniziale della macchina. Vi si giunge attraverso il segnale di *reset* del testbench. In questo stato si attende il segnale di *start*; dopo il quale vengono inizializzati i segnali interni alla macchina e si passa allo stato successivo.

## INIZIALE

Una volta terminata l’inizializzazione dei segnali all’interno dello stato di reset si giunge in questo stato dove vengono salvati nei segnali *colonne, righe, soglia* i valori salvati in memoria che rimarranno costanti per tutta la durata dell’esecuzione.

## CONFRONTO

Questo è lo stato più importante della macchina, in questo stato avviene il confronto tra gli elementi salvati in memoria con il valore di soglia. Se il confronto da esito positivo vengono aggiornati di conseguenza i valori dei segnali di *nord, sud, est* e *west*.

Al termine della lettura in memoria si passa al successivo stato per il calcolo dell’area.

## STATO\_MOLTIPLICA

In questo stato avviene la moltiplicazione per il calcolo dell’area del rettangolo in cui vi è contenuta l’immagine secondo la logica:

*moltiplica <= (sud - nord + "00000001") \* (est - west + "00000001");*

Il più *“00000001”* viene introdotto perché il calcolo dell’area deve tenere conto degli elementi che compongono il “bordo”.

## SALVA

In questo stato vi si giunge dopo aver calcolato l’area richiesta; in questo stato si procede al salvataggio in memoria dell’area. Si innalza anche il segnale per il testbench *done*.

## ASPETTA

È l’ultimo stato della macchina. Permette di abbassare il livello *done* e permettere al testbench di eseguire la valutazione della correttezza dell’area calcolata.

# implementazione architettura

Per realizzare il componente abbiamo deciso di usare una macchina a stati finiti gestita da un singolo processo. Di seguito commenteremo le scelte più significative.

Finché non viene ricevuto il segnale di reset la macchina non si trova in uno stato particolare e quindi non verrà eseguita nessuna istruzione. Ricevuto il reset la macchina inizierà ad operare durante il fronte di salita del clock e si provvede ad aggiornare lo stato.

*if (i\_rst='1') then*

*stato\_corrente<= reset;*

*elsif (i\_clk'event and i\_clk='1') then*

*stato\_corrente <= stato\_prossimo;*

*end if;*

Dato che la memoria RAM lavora sul fronte di salita del clock, la macchina dovrà eseguire le varie operazioni durante la fase di discesa del clock.

*if (i\_clk'event and i\_clk='0') then*

Per verificare lo stato attuale della macchina usiamo il costrutto case; questo porta un vantaggio perché bisogna tener conto del fatto che la macchina al primo avvio non si trova in uno stato specifico e rendere il codice più leggibile.

*case stato\_corrente is*

*when reset =>…*

*when iniziale =>…*

*…*

*when others =>…*

All’inizializzazione i registri che tengono conto dei 4 punti estremi della figura sono inizializzati ad alta impedenza, in questo modo se l’immagine non ha pixel che superano la soglia possiamo vederlo dal fatto che i registri hanno mantenuto questo valore. Inoltre, predisponiamo la macchina per la lettura dell’header.

*nord <= "ZZZZZZZZ";*

*sud <= "ZZZZZZZZ";*

*west <= "ZZZZZZZZ";*

*est <= "ZZZZZZZZ";*

*o\_address <="0000000000000010";*

*coordr <="00000001";*

*coordc <="00000001";*

*phase <= "00";*

*o\_en <= '1';*

*o\_we <= '0';*

*o\_done <= '0';*

*stato\_prossimo <= iniziale;*

Lo stato iniziale è in realtà un gruppo di tre stati (uno per ogni elemento dell’header) nei quali si esegue la lettura da memoria e il salvataggio di questi valori.

*case phase is*

*when "00" =>*

*colonne <= i\_data;*

*o\_address <="0000000000000011";*

*o\_en <= '1';*

*o\_we <= '0';*

*phase <= "01";*

*when "01" =>*

*righe <= i\_data;*

*o\_address <="0000000000000100";*

*o\_en <= '1';*

*o\_we <= '0';*

*phase <= "10";*

*when "10" =>*

*soglia <= i\_data;*

*phase <= "00";*

Per evitare un inutile passaggio allo stato d confronto abbiamo sempre messo nell’ultima fase dello stato iniziale il controllo di valori degeneri del numero di righe o di colonne.

In caso di valori non degeneri ci predisponiamo per iniziare il confronto.

*if (righe="00000000" or colonne="00000000") then*

*stato\_prossimo <= stato\_moltiplica;*

*o\_address <="0000000000000000";*

*prossimo\_address <="0000000000000001";*

*else*

*stato\_prossimo <= confronto;*

*o\_address <="0000000000000101";*

*prossimo\_address <="0000000000000110";*

*o\_en <= '1';*

*o\_we <= '0';*

*end if;*

Durante la fase di confronto si valutano le coordinate del pixel con valore che supera la soglia con i valori estremi che abbiamo trovato finora.

Se il valore dei registri è alta impedenza allora vengono aggiornati col valore delle coordinate del punto. Per semplicità valutiamo solo il valore di nord per controllare se sono in alta impedenza dato che non posso trovarsi in situazioni “miste”.

*if(i\_data >= soglia) then*

*if (nord = "ZZZZZZZZ") then*

*nord <= coordr;*

*sud <= coordr;*

*est <= coordc;*

*west <= coordc;*

*else*

*sud <= coordr;*

*if(coordc < west) then*

*west <= coordc;*

*elsif(coordc > est) then*

*est <= coordc;*

*end if;*

*end if; -- fine if dove setto*

*end if; -- fine controllo soglia*

Inoltre, aggiorniamo l’indirizzo sempre nello stato di confronto in modo da risparmiare sul numero di cicli di clock

*if(coordr = righe and coordc = colonne) then*

*o\_address <= “00000000”;*

*prossimo\_address <= “00000001”;*

*o\_en <= ‘0’;*

*o\_we <= ‘0’;*

*stato\_prossimo <= moltiplica;*

*else*

*o\_address <= prossimo\_address;*

*prossimo\_address <= prossimo\_address + "0000000000000001";*

*o\_en <= '1';*

*o\_we <= '0';*

*if(coordc = colonne) then*

*coordc <= "00000001";*

*coordr <= coordr + "00000001";*

*else*

*coordc <= coordc + "00000001";*

*end if;*

*end if;*

Lo stato di salvataggio è diviso in quattro sottostati per poter eseguire tutte le operazioni necessarie al salvataggio su più cicli di clock (4 per l’esattezza). Inoltre, per evitare problemi di scrittura cambiamo l’indirizzo per la seconda metà del prodotto solo nel ciclo di clock successivo.

*case productphase is*

*when "00" =>*

*o\_en <= '1';*

*o\_we <= '1';*

*o\_data <= moltiplica (7 downto 0);*

*productphase <= "01";*

*when "01" =>*

*o\_en <= '0';*

*o\_we <= '0';*

*o\_address <= prossimo\_address;*

*productphase <= "10";*

*when "10" =>*

*o\_en <= '1';*

*o\_we <= '1';*

*o\_done <= '1';*

*o\_data <= moltiplica (15 downto 8);*

*productphase <= "11";*

*when "11" =>*

*o\_en <= '1';*

*o\_we <= '1';*

*o\_done <= '0';*

*stato\_prossimo <= aspetta;*

*when others => stato\_prossimo <= aspetta;*

# development E TESTING

Dopo la prima stesura della macchina a stati finiti abbiamo iniziato a valutarne, usando come appoggio i testbench forniti dal professore, il funzionamento in pre-sintesi. Apparentemente l’algoritmo da noi implementato inizialmente funzionava senza problemi e quindi siamo passati alla post sintesi.

Dopo la sintesi sono emersi numerosi errori durante l’esecuzione delle simulazioni tra cui:

1. Il contatore non avanzava come previsto ad ogni ciclo di clock;
2. Il valore del segnale “nord” era superiore di una unità a quello effettivo;
3. I segnali in uscita “we” e “en” non venivano aggiornati correttamente.

Dopo un’attenta analisi del codice (e una lettura di un manuale sul VHDL) abbiamo apportato delle modifiche al codice onde evitare possibili errori di sintesi. Abbiamo raggruppato il controllo del clock prima del controllo dello stato presente, evitando inoltre le ripetizioni di if clause ridondanti.

L’inizializzazione dei contatori di riga e colonna da 0 è stata imposta ad 1 per semplificare il conteggio e seguire l’andamento dei numeri naturali. Inoltre, per controllare meglio i valori che certi segnali possono assumere si è scelta la forma case invece di quella if/elsif, potendo così tener conto di possibili stati di errore nella macchina.

Superati anche i test in post sintesi abbiamo voluto valutare i casi estremi che la macchina poteva affrontare in esecuzione. Mentre i test su matrici 255\*255 e valori particolari delle moltiplicazioni si sono dimostrati positivi, il test sul caso di matrice 0\*0 con poca sorpresa ha creato non pochi problemi in sintesi. Infatti, la macchina era parzialmente hardcodata per la lettura su matrici non degeneri. Abbiamo quindi optato per un controllo del valore di righe e colonne prima della fase di confronto in modo da evitare sprechi di tempo leggendo parti di memoria che non andavano controllate. Le modifiche effettuate alla macchina l’hanno resa anche più flessibile dato che abbiamo reso più elastiche le parti di codice “fisse” che avevamo introdotto in precedenza.

La RAM da noi utilizzata per i casi limite è la seguente:

*signal RAM: ram\_type := (2 => "00000000", 3 =>"00000000",4 => "00000000", others => (others =>'0'));*

*assert RAM(1) = "00000000" report "FAIL high bits" severity failure;*

*assert RAM(0) = "00000000" report "FAIL high bits" severity failure;*

*assert false report "Simulation Ended!, test passed" severity failure;*

*\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_*

*signal RAM: ram\_type := (2 => "11111111", 3 =>"11111111",4 => "00000000", others => (others =>'0'));*

*assert RAM(1) = "00000001" report "FAIL high bits" severity failure;*

*assert RAM(0) = "11111110" report "FAIL high bits" severity failure;*

*assert false report "Simulation Ended!, test passed" severity failure;*

Al termine delle modifiche abbiamo rieseguito tutti i casi di test (sia quelli scaricabili, sia quelli creati da noi) ottenendo un risultato positivo e un mantenimento delle prestazioni sulla frequenza di lavoro.

Data la natura del nostro algoritmo di attraversamento della memoria, non esistevano effettivi casi “pessimi” visto che la memoria veniva attraversata sempre per intero. L’unica problematica legata all’attraversamento è infatti la complessità θ(n) che, per valori grandi di righe e di colonne, si traduce in lunghi tempi di esecuzione; un sacrificio necessario, per ottenere una frequenza di lavoro ottimale.

***I risultati di Timing dell’implementation hanno prodotto come risultato: frequenza massima 131.320 MHz ( ovvero un periodo di clock di 7.615 ns).***

# VirtualBox_Windows%2010_27_02_2018_11_53_07.pngVirtualBox_Windows%2010_27_02_2018_11_52_18.pngVirtualBox_Windows%2010_27_02_2018_11_52_02.pngSCHEMA LOGICO