Progetto di Reti Logiche

Francesco Lorenzo, Emilio Imperiali

Sommario

[1. Componenti del gruppo 1](#_Toc508632408)

[2. Introduzione 1](#_Toc508632409)

[3. Obiettivi non funzionali 3](#_Toc508632410)

[4. Descrizione dei moduli 4](#_Toc508632411)

[4.1 Header 5](#_Toc508632412)

[4.2 Input 6](#_Toc508632413)

[4.3 Compute\_result 8](#_Toc508632414)

[4.4 Output 10](#_Toc508632415)

[4.5 Project\_reti\_logiche (Controller) 11](#_Toc508632416)

[5. Testing 13](#_Toc508632417)

# Componenti del gruppo

Il progetto è stato svolto in un gruppo da due, formato da:

* Nome: Francesco Vito  
  Cognome: Lorenzo  
  Codice Persona: 10521186  
  E-mail: [francesco1.lorenzo@mail.polimi.it](mailto:francesco1.lorenzo@mail.polimi.it)
* Nome: Emilio  
  Cognome: Imperiali  
  Codice Persona: 10526844  
  E-mail: [emilio.imperiali@mail.polimi.it](mailto:emilio.imperiali@mail.polimi.it)

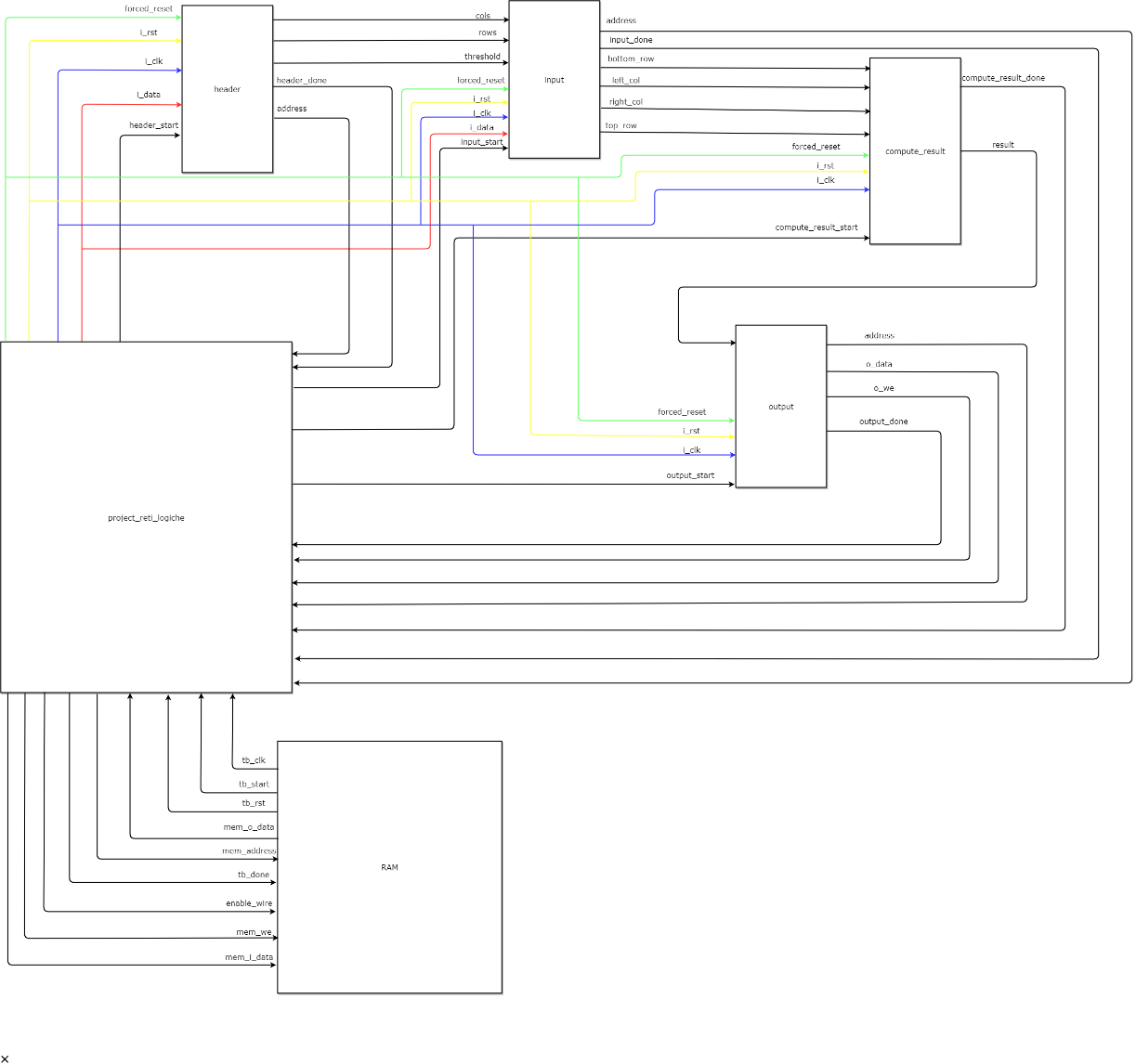
# Introduzione

Il progetto è basato su un’architettura hardware modulare con logica centralizzata, gestita da un controller implementato dall’entity project\_reti\_logiche, che coordina gli altri moduli presenti: header, input, compute\_result e output (tutti implementati da entity omonime).

La scelta di un’architettura modulare è basata sulla volontà di separare componenti logicamente indipendenti fra loro, diminuendo la possibilità di errore, garantendo la riusabilità del codice e, a prezzo dell’implementazione della logica di controllo, la possibilità di sfruttare la semplicità dell’architettura strutturale in VHDL.

Come si potrà evincere nel seguito della trattazione, si è scelto di usare prevalentemente una descrizione Behavioral per implementare le funzionalità principali di ogni modulo, mentre l’utilizzo di un’architettura Dataflow è limitato alla descrizione dei Multiplexer.

Segue una rappresentazione ad alto livello dell’architettura del circuito.



Il controllore, tramite una macchina a stati finiti, sincronizza fra loro tutti gli altri moduli, fornendo loro il segnale di inizio computazione e ricevendo da ognuno il segnale di fine computazione.

Dopo aver alzato done, ed in corrispondenza della ricezione di un reset, il controllore si occupa di resettare ogni modulo ponendosi in uno stato in cui è pronto ad iniziare la computazione in corrispondenza di uno start.

Il modulo header si occupa di leggere dalla memoria i dati relativi al numero di colonne, di righe e alla threshold e di passarli al controller.

Il modulo input si occupa di scandire linearmente la matrice salvandosi nel frattempo le dimensioni del rettangolo minimo che racchiude l’immagine, passandole alla fine al controller.

Il modulo compute\_result calcola la base e l’altezza del rettangolo, quindi le moltiplica e passa il risultato al controller.

Il modulo output scrive nei byte 0 e 1 il risultato e notifica la fine di computazione al controller.

# Obiettivi non funzionali

Le scelte implementative fatte in fase di sviluppo, come si potrà evincere dalla descrizione dettagliata dei moduli, sono mirate ad un compromesso fra la massimizzazione della frequenza di clock che comanda il circuito e la minimizzazione dell’area dello stesso, con un focus maggiore sul primo requisito.

Un obiettivo secondario è stato quello di ottenere un corretto funzionamento anche in simulazione post-sintesi timing e, come si potrà evincere dalla sezione relativa al testing, è stato raggiunto.

Per quanto concerne la latenza totale si è raggiunta una complessità lineare nella dimensione della matrice, infatti detto T il tempo che intercorre fra il rising edge di start e il falling edge di done, si ha:

Dove n indica il numero di righe della matrice, m il numero di colonne, 7 sono i cicli di clock necessari a leggere l’header e scrivere in memoria, mentre c una costante che dipende dalle dimensioni dell’intero che rappresenta l’area. Si noti che essendo il segnale di start generalmente dato asincrono, l’inizio vero e proprio della computazione si ha in corrispondenza del successivo rising edge del clock, nella formula si è trascurata questa frazione di tempo.

La minimizzazione della latenza complessiva non è stata obiettivo principale in fase di sviluppo, si è preferito infatti inserire alcuni cicli di stallo per sincronizzare i diversi moduli in modo sicuro, oltre ad implementare un moltiplicatore asincrono che impiega generalmente più di un ciclo di clock (si vedano 4.3 per maggiori dettagli sul moltiplicatore e 5 per i risultati di latenza).

In accordo con il Timing Report, il periodo minimo di clock raggiunto è di con un WNS di:

* 0.117ns nella fase di Setup
* 0.22ns nella fase di Hold
* 1.675ns nel Pulse Width

Il collo di bottiglia nel circuito si ha in input, ed è dovuto all’overhead degli statement condizionali ed ai comparatori.

Eliminare il collo di bottiglia avrebbe richiesto la revisione di gran parte del modulo di input, portando anche ad un aumento di Flip-flop usati, si è quindi preferito fermarsi a questo risultato in fase di ottimizzazione.

I dati riguardanti l’area occupata, tratti dal Project Summary in Post-Sintesi, sono i seguenti:

* LUT: 258
* FF: 154
* IO: 38
* BUFG: 1

Le LUT sono principalmente dovute ai numerosi statement condizionali nel modulo di input, mentre i FF in eccesso sono causati dall’utilizzo di segnali temporanei da 8 e 16 bit laddove era necessario leggere il valore di un segnale in uscita, oppure dove si creavano situazione di segnali Undefined.

# Descrizione dei moduli

Segue la descrizione dettagliata di ciascun modulo, in cui verranno fornite:

* Una breve descrizione dell’interfaccia di input e di output del modulo stesso.
* L’algoritmo implementato dal modulo, descritto passo per passo.
* Un’analisi di complessità del codice.
* Le scelte progettuali fatte durante lo sviluppo, accennando ove possibile alle precedenti versioni che sono state scartate o migliorate.
* Il diagramma degli stati della FSM relativa a quel modulo.

A tal proposito, si tenga conto che:

1. Tutti i moduli ricevono come input i\_rst e forced\_reset. Quest’ultimo è un reset alternativo, alzato a fine computazione e dichiarato nel controllore, nonostante questo sia l’unico a non farne uso (maggiori dettagli in 4.5).
2. Tutti i moduli ricevono in input i\_clk per la sincronizzazione, un segnale di start per comandarne l’inizio della computazione e forniscono in output un segnale di done per segnalarne la fine (per il controllore questi ultimi due segnali sono rispettivamente i\_start e o\_done).
3. I moduli hanno dei Multiplexer descritti in Dataflow per controllare il valore di alcuni segnali in base allo stato della relativa FSM.
4. Nella rappresentazione dei diagrammi degli stati si sono adottate le seguenti convenzioni:
   1. Le uscite del modulo non sono rappresentate sugli archi.
   2. Gli ingressi che non sono presenti su nessun arco uscente da uno stato S sono da assumersi come irrilevanti per l’evoluzione della macchina da tale stato.
   3. Se in un certo stato S un ingresso I non può assumere il valore X (perché il controllore non lo permette), allora non esisterà un arco uscente da S con etichetta I=X. Un esempio è forced\_reset, che nei moduli non può mai assumere valore logico ‘1’ tranne quando la computazione è finita.
   4. Alcune etichette presentano delle condizioni logiche sui segnali in ingresso: la ‘,’ rappresenta un or logico, mentre l’and ed il not sono esplicitati a parole.

## 4.1 Header

Il modulo Header, regolato da una macchina a stati finiti (figura 4 A), si interfaccia alla RAM tramite i bus address e i\_data che si mappano rispettivamente su mem\_address e mem\_o\_data della stessa.

Lo scopo del modulo è quello di leggere i byte 2,3 e 4 che contengono il numero di colonne (cols), il numero di righe (rows) e la threshold della matrice; questi tre valori vengono quindi forniti ad input per successive elaborazioni.

*Algoritmo*

La macchina viene inizializzata dai reset allo stato S0, in cui si pone in attesa del segnale di inizio computazione (header\_start).

Dal momento in cui il modulo riceve tale segnale, vi sono due cicli di clock in cui la macchina prepara l’indirizzo da fornire alla RAM per la lettura (address), mentre a partire dal terzo ciclo (state = S2) si assegnano effettivamente i valori letti da i\_data ai segnali in uscita.

Si ha infine un ciclo (state = S4) in cui non viene assegnato un nuovo indirizzo, dato che si sta ancora aspettando l’ultimo dato in uscita dalla RAM.

Nello stato S5 la macchina alza il segnale di fine computazione (header\_done) e vi rimane fino ad un nuovo segnale di reset.

*Complessità*

La complessità del modulo è ovviamente costante, richiedendo 5 cicli di clock dal momento in cui viene alzato header\_start al momento in cui viene alzato header\_done.

*Scelte progettuali*

Nella prima versione del modulo si utilizzava un segnale temporaneo di fine computazione, che è stato però sostituito da un altro stato S5 che garantisce una maggiore chiarezza del codice, meno LUT e un FF in meno (analoghe considerazioni valgono per tutti i prototipi iniziali dei successivi moduli).

Alcuni problemi erano causati anche dalla sincronizzazione fra il momento in cui un indirizzo viene assegnato, ed il momento in cui il valore viene corrispondente effettivamente letto.

Per risolvere questo problema si è ricorsi allo stallo di un ciclo di clock in cui il modulo non assegna nessuno dei tre valori di output, ma cambia solo l’indirizzo corrente in attesa di un input significativo dalla RAM.

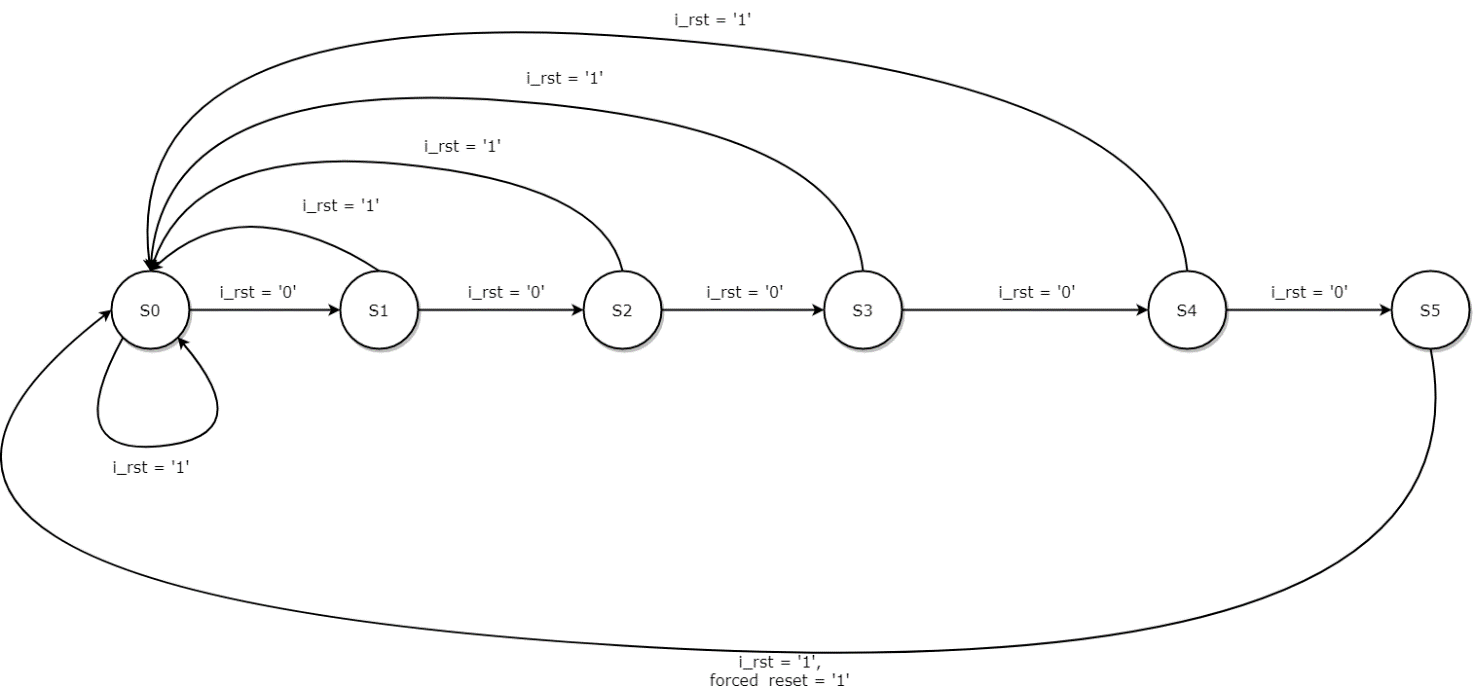


Figura 4 A

## 4.2 Input

Il modulo input, regolato da una macchina a stati finiti (figura 4 B), si interfaccia alla RAM tramite i bus address e i\_data che si mappano rispettivamente su mem\_address e mem\_o\_data della stessa.

L’output che il modulo fornisce a compute\_result consiste delle quattro dimensioni del rettangolo minimo che racchiude la matrice: la riga superiore (top\_row), la riga inferiore (bottom\_row), la colonna sinistra (left\_col) e la colonna destra (right\_col) di tale rettangolo.

L’indice di riga (prev\_row) parte da 1 per la prima riga della matrice, fino ad essere pari a rows per l’ultima riga della matrice. L’indice di colonna (prev\_col) si comporta nello stesso modo, partendo da 1 fino ad arrivare a cols per l’ultima colonna.

Si sono utilizzati quattro segnali temporanei, in quanto per fare confronti non era possibile leggere direttamente i rispettivi segnali di output: tmp\_address, tmp\_left\_col, tmp\_right\_col, tmp\_top\_row.

*Algoritmo*

Il modulo si prefigge l’obiettivo di leggere tutte le celle della matrice, da sinistra a destra e dall’alto verso il basso, con un’unica scansione lineare.

La macchina viene inizializzata dai reset allo stato S0, in cui si pone in attesa del segnale di inizio computazione (input\_start).

Quando tale segnale viene alzato si inizializzano i segnali, in particolare:

* Address e tmp\_address sono inizializzati a 5, ossia l’indirizzo del primo byte vero e proprio della matrice.
* Top\_row e tmp\_top\_row sono inizializzati a rows, tale espediente è utile in quanto qualunque indice di riga è minore o uguale a rows, quindi:
  + Se esiste almeno una cella con valore maggiore o uguale alla threshold, il suo indice di riga diventerà top\_row.
  + Se non esiste, top\_row rimane inizializzato al valore rows, permettendo poi di accorgersi che il rettangolo cercato non esiste.
* Bottom\_row, che non ha bisogno di un corrispondente segnale temporaneo, in quanto per ogni cella che supera la threshold il suo indice di riga sarà il nuovo bottom\_row, viene inizializzata a 0 per motivi speculari a quelli di top\_row.
* Left\_col e tmp\_left\_col sono inizializzati a cols, esattamente come top\_row e tmp\_top\_row.
* Right\_col e tmp\_right\_col sono inizializzati a 0, come per bottom\_row.
* Prev\_col e prev\_row sono inizializzati a 1, ad indicizzare la prima cella della matrice.

Dopo l’inizializzazione si controlla se rows o cols sono pari a 0, in tal caso la macchina va direttamente allo stato S3 di fine computazione, in quanto la matrice è degenere, altrimenti si va allo stato S1 di inizio scansione.

Lo stato S1 ha la funzione di lasciare ad 1 per un altro ciclo di clock prev\_col e prev\_row, in quanto l’indice di colonna e di riga al generico ciclo di clock T devono rappresentare le coordinate dell’elemento letto al ciclo di clock T, ma il cui indirizzo era stato settato al ciclo T-1.

In S1 si ha il controllo sul primo elemento della matrice, i dettagli sono spiegati più avanti.

Dopo un ciclo di clock in questo stato si passa alla computazione vera e propria che avviene in S2.

In S2 si leggono tutte le altre celle di memoria restanti: il modulo controlla se il valore è maggiore o uguale alla threshold, in tal caso confronta i suoi indici di riga con top\_row e bottom\_row, quelli di colonna con left\_col e right\_col. Possono presentarsi cinque casi:

* L’indice di riga attuale è minore di top\_row: allora top\_row diventa prev\_row.
* L’indice di riga attuale è maggiore di bottom\_row: allora bottom\_row diventa prev\_row (questo caso è irrilevante perché l’indice di riga attuale sarà sempre minore di bottom\_row).
* L’indice di colonna attuale è minore di left\_col: allora left\_col diventa prev\_col.
* L’indice di colonna attuale è maggiore di right\_col: allora right\_col diventa prev\_col.
* Nessuno dei precedenti: in tal caso gli output non vengono modificati (la cella attuale appartiene già al rettangolo minimo).

Quando prev\_col raggiunge cols, se la matrice non è ancora stata letta completamente, si incrementa di 1 prev\_row e si riporta ad 1 prev\_col. Quando anche prev\_row è uguale a rows si passa allo stato S3 alzando il segnale di fine computazione (input\_done).

*Complessità*

Nel caso non degenere, input presenta una complessità lineare nella dimensione della matrice, richiedendo un numero di cicli di clock, dal momento in cui viene alzato input\_start al momento in cui viene alzato input\_done, pari a , con n=rows e m=cols.

Nel caso degenere la complessità è ovviamente costante, richiedendo solo due cicli di clock.

*Scelte progettuali*

In una versione precedente del codice erano presenti due segnali chiamati curr\_col e curr\_row che erano l’indice di colonna e di riga corrispondenti all’indirizzo attuale di memoria, quindi prev\_col e prev\_row con un ciclo di anticipo. Tramite l’utilizzo di questi segnali era possibile eliminare lo stato S1 e partire sin da subito con la computazione in S2, ovviamente l’area occupata era maggiore, quindi si è deciso di adottare l’attuale soluzione risparmiando in totale 16FF.

Sempre in una precedente versione non era stato utilizzato l’espediente di inizializzare top\_row a rows e left\_col a cols, era presente invece uno stato aggiuntivo di stallo in cui si cercava il primo elemento che superava la threshold, a quel punto si inizializzavano le dimensioni del rettangolo e si passava a quello che ora è lo stato S2, in cui avviene la lettura della restante parte della matrice. Ovviamente nello stato attuale il modulo occupa meno area ed è più chiaro.

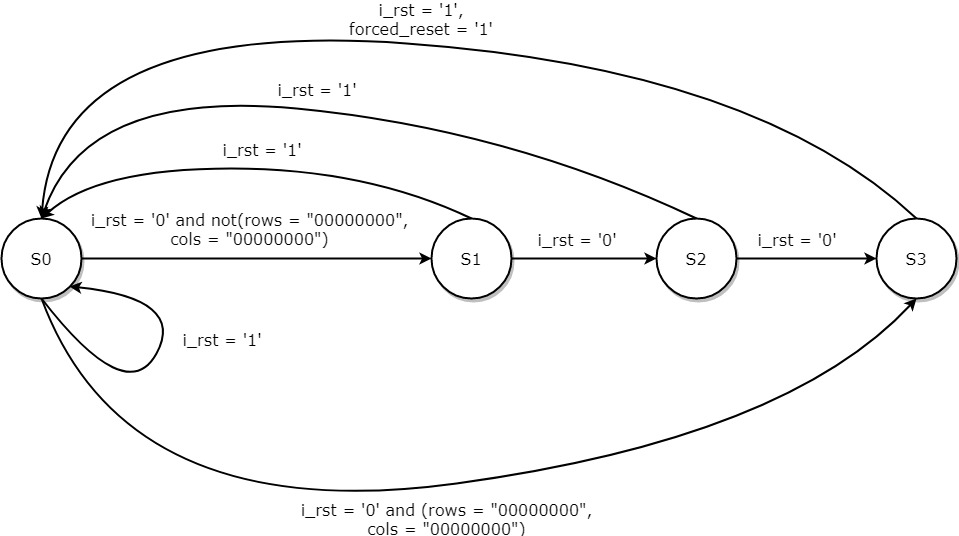


Figura 4 B

## 4.3 Compute\_result

Il modulo compute\_result, regolato da una macchina a stati finiti (figura 4 C), non si interfaccia alla RAM. Si interfaccia invece al modulo input tramite i bus top\_row, bottom\_row, left\_col e right\_col che si mappano sui segnali omonimi.

Il modulo calcola la base (width) e l’altezza (height) del rettangolo, le moltiplica e fornisce l’area (result) al modulo di output per essere scritta in memoria.

*Algoritmo*

Con un segnale di reset la macchina viene inizializzata allo stato S0, la computazione vera e propria inizia quando compute\_result\_start viene alzato ad 1.

Il modulo controlla se le dimensioni corrispondono a quelle di un rettangolo degenere, quindi se left\_col o bottom\_row sono rimasti invariati rispetto al valore di inizializzazione (andrebbero controllati anche right\_col e top\_row, ma se un indice non è stato modificato allora anche l’altro è rimasto invariato), in caso negativo si passa in S1, altrimenti si assegna zero al risultato e si va in S3 dove finisce la computazione.

In S1 il modulo calcola base ed altezza in un ciclo di clock, a quel punto va in S2. A fronte di questa commutazione un secondo process, asincrono, si attiva e comincia il calcolo dell’area, sfruttando il moltiplicatore di libreria. Una volta finita la moltiplicazione, che generalmente impiega più di un ciclo di clock (con ), viene alzato un segnale di fine moltiplicazione che permette al modulo di concludere la sua computazione.

Si noti che l’utilizzo di tmp\_result è dovuto al fatto che il secondo process non può direttamente assegnare un valore a result stesso, o si creerebbe una situazione di Undefined.

*Complessità*

Nel caso di area nulla il modulo impiega un solo ciclo di clock dal momento in cui compute\_result\_start viene alzato al momento in cui compute\_result\_done viene alzato. In caso contrario, i cicli di clock si possono approssimare con il valore , dove Tm è il numero di cicli richiesti dal moltiplicatore, ed è funzione dell’attuale periodo di clock. La complessità è in questo caso costante.

*Scelte progettuali*

La scelta del moltiplicatore asincrono è avvenuta dopo varie prove con moltiplicatori diversi.

Inizialmente in un solo ciclo di clock si calcolavano base, altezza e poi la moltiplicazione delle due, in quel caso il tempo richiesto da queste operazioni era approssimativamente di 5.8ns.

Si è poi scelto di separare il calcolo di base e altezza da quello della moltiplicazione, raggiungendo 5.4ns come periodo di clock minimo.

A questo punto sono state provate due alternative:

* La prima è stata quella di implementare un moltiplicatore di Dadda sincrono, raggiungendo un periodo di clock di 4.8ns.
* La seconda, che è poi quella attuale, è stata quella di rendere il moltiplicatore di libreria asincrono, rimuovendo così completamente il collo di bottiglia al costo di più di un ciclo di clock di calcolo. Il periodo minimo raggiunto è quindi di 4.35ns.

Si noti che nella situazione attuale i warning relativi ai latch sui segnali di mult\_end e tmp\_result sono voluti: i componenti che si stanno descrivendo, non dipendendo da un segnale di sincronismo, sono dei latch, non dei FF.

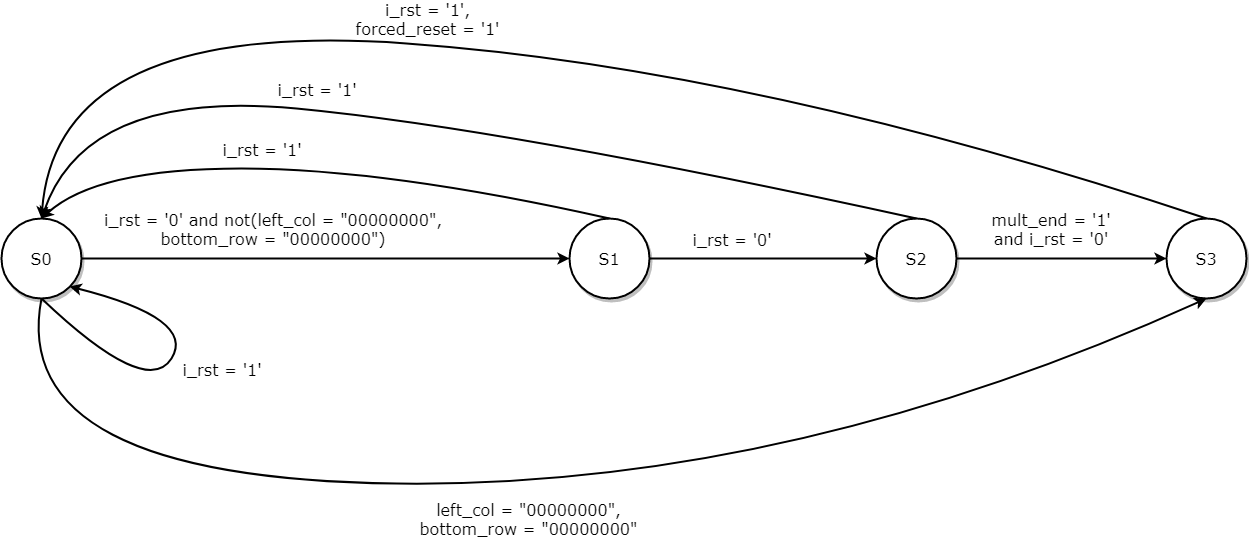


Figura 4 C

## 4.4 Output

Il modulo output, regolato da una macchina a stati finiti (figura 4 D), si interfaccia alla RAM tramite i bus address, o\_data e o\_we, che si mappano rispettivamente su mem\_address, mem\_i\_data e mem\_we della stessa.

*Algoritmo*

Output ha come obiettivo quello di scrivere nel byte 0 della RAM gli 8 bit meno significativi di result, e nel byte 1 gli 8 bit più significativi.

Dopo l’inizializzazione a S0 da parte dei segnali di reset, il modulo attende che il controllore alzi il segnale output\_start. Dopo aver assegnato i valori di address e o\_data corretti, la macchina commuta in S1, dove viene alzato o\_we per poter scrivere in memoria. I valori vengono aggiornati e la macchina va in S2, dove finisce di scrivere e si sposta in S3 per terminare la computazione.

*Complessità*

Il modulo richiede sempre due cicli di clock dal momento in cui viene alzato output\_start al momento in cui viene alzato output\_done, la complessità è quindi costante.

*Scelte progettuali*

Output non è variato dal suo iniziale concepimento, in quanto le ottimizzazioni possibili erano state fatte subito.

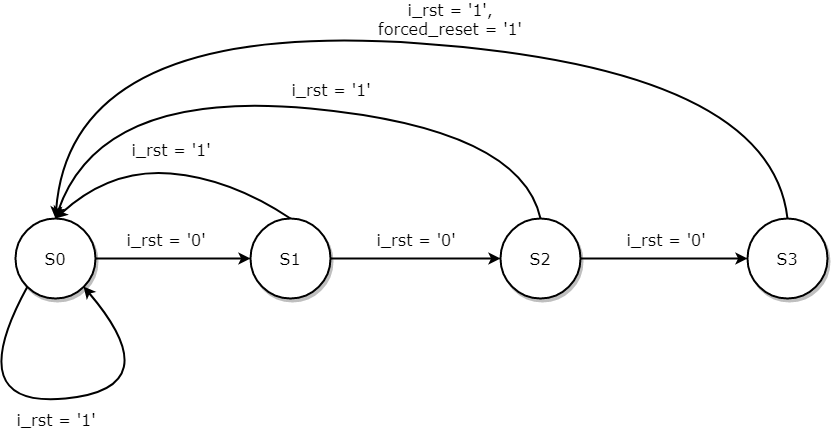


Figura 4 D

## 4.5 Project\_reti\_logiche (Controller)

Il controllore, regolato da una macchina a stati finiti (figura 4E), è legato alla RAM dall’interfaccia standard richiesta dalla specifica del progetto.

Tutti gli altri moduli sono qui mappati strutturalmente, vengono infatti dichiarati segnali appositi che rappresentano gli ingressi e le uscite dei moduli stessi.

*Algoritmo*

Il controllore è posto nello stato S0 dal segnale di start della RAM, da quel momento la macchina commuterà progressivamente da S1 fino a S4, dando ogni volta il segnale di inizio al modulo corrente (header > input > compute\_result > output) ed aspettando che esso termini la sua computazione.

In S4 il modulo azzera i segnali di inizio degli altri moduli, li resetta alzando forced\_reset ad 1 e segnala la fine di computazione portando o\_done ad 1.

Dal ciclo di clock successivo, in cui o\_done viene riportato a 0, il controllore è pronto a ricevere immediatamente un nuovo segnale di start, a tal proposito si noti che l’assegnamento all’inizio del process di o\_done e forced\_reset a 0 è necessario a garantire questa proprietà, altrimenti non ci si sarebbe potuti accorgere del segnale di start alto se nello stesso ciclo si entrava in uno statement condizionale per abbassare done.

La selezione del corretto indirizzo da fornire alla RAM è fatta tramite un multiplexer descritto in architettura Dataflow, che, a seconda dello stato corrente, seleziona l’indirizzo dal modulo attivo in tale stato.

*Complessità*

Per l’analisi di complessità del modulo si consulti la sezione “Obiettivi non funzionali”, essendo questa la complessità dell’intero codice.

*Scelte progettuali*

Il modulo project\_reti\_logiche è, insieme ad input, quello che ha subito più modifiche durante la progettazione.

La primissima iterazione del codice prevedeva un unico process, interno a project\_reti\_logiche, che implementava tutte le funzionalità dei moduli precedentemente descritti. La principale difficoltà incontrata era quella di non riuscire a selezionare l’address da utilizzare ogni volta per poter modularizzare il codice, poi con l’utilizzo del multiplexer attualmente presente il problema è stato risolto.

Il controllore ha quindi subito assunto la configurazione di una FSM per poter sincronizzare i moduli fra loro, anche se inizialmente non prevedeva la possibilità di ricevere un rising edge di start contemporaneo al falling edge di done, in quanto quest’ultimo veniva abbassato da forced\_reset, e quindi per resettare veniva ignorato il ramo if che controllava un nuovo start.

Come sottolineato in precedenza, si è deciso di rimuovere il compito di forced\_reset nel controllore, delegando l’utilizzo di tale segnale al solo reset degli altri moduli. Questo ha permesso di garantire il corretto funzionamento con start contemporaneo all’abbassarsi di done.

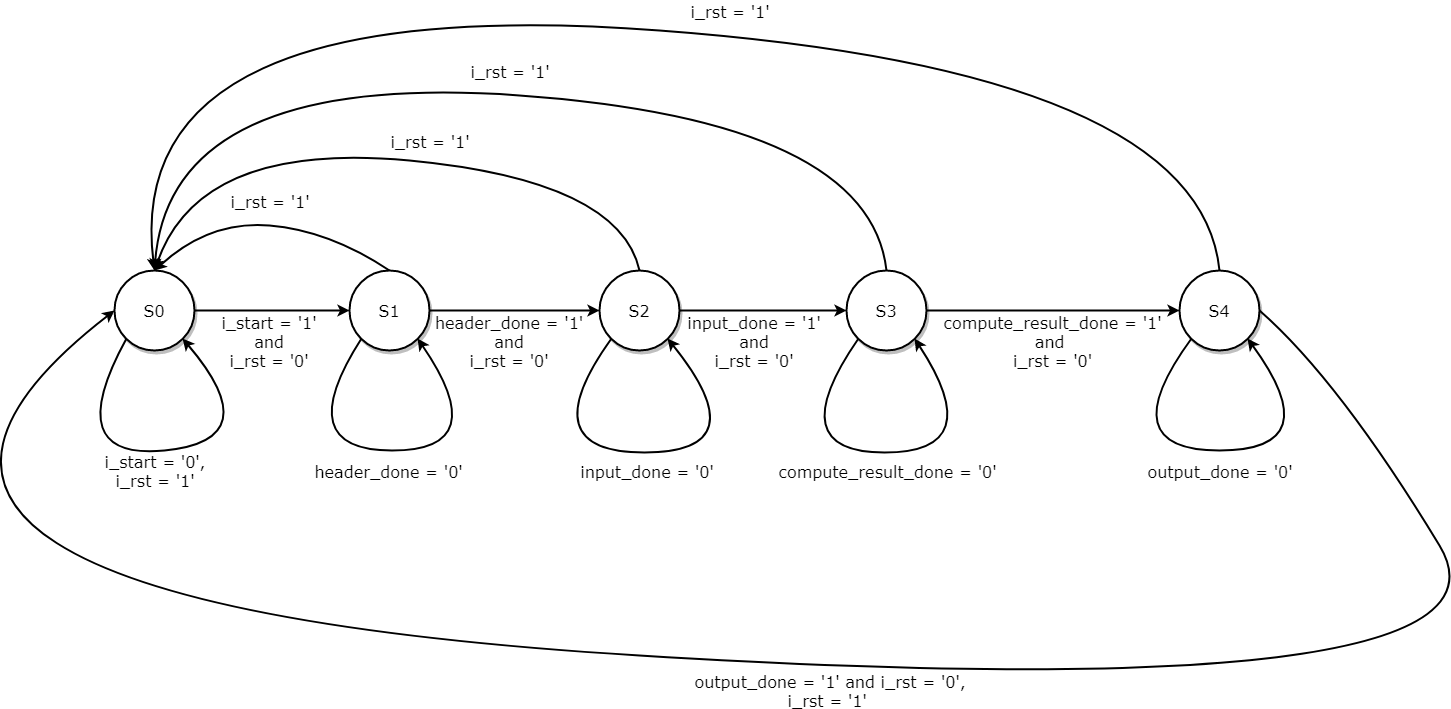


Figura 4 E

# Testing[[1]](#endnote-2)

Il testing che va a validare il progetto è stato eseguito con le seguenti modalità: in primo luogo, sono stati testati i singoli moduli utilizzando dei testbench appositamente creati da noi. La logica che si è voluta seguire è quella del testing black box, mentre il white box è stato evitato in quanto si è notato che i casi testati nel primo erano già sufficienti a coprire il codice al 100%. In secondo luogo, dopo aver collegato insieme i vari moduli, si è effettuato un test di sistema, utilizzando dei testbench basati su quegli stessi casi di test che sono stati usati per il test di unità. Si vuole sottolineare che tutti i test effettuati sono stati superati:

* In Behavioral Simulation;
* In Post-Synthesis Functional Simulation;
* In Post-Synthesis Timing Simulation.

È bene ricordare infine che il testing è stato interamente effettuato basandosi sull’indicazione di assegnare alla memoria un ritardo di I/O di 1 ns, e come periodo della forma d’onda che definisce il clock è stato sempre lasciato 15 ns.

Di seguito vengono riportati dunque i casi di test di unità e di sistema (sono stati raggruppati in un unico punto quei casi di test che si sono ritenuti concettualmente simili, sia per semplificare il nostro lavoro sia per rendere meno pesante la trattazione):

1. Si sono usati per ogni modulo dei testbench basati sui 4 che sono stati forniti su Beep, per poi invece usare i 4 originali sul progetto nella sua interezza;
2. Si è definito un testbench che controlli i casi degeneri della matrice, aventi tutti risultato nullo:
   * Rows=0, Cols=0, Threshold=255;
   * Rows=0, Cols=0, Threshold=0;
   * Rows=0, Cols=255, Threshold=0;
   * Rows=0, Cols=255, Threshold=255;
   * Rows=255, Cols=0, Threshold = 0;
   * Rows=255, Cols=0, Threshold = 255;
3. Si è definito un testbench che controlli i casi di matrice piena di dimensioni massime, aventi tutti risultato 65025:
   * Rows=255, Cols=255, Threshold=0;
   * Rows=255, Cols=255, Threshold=255;
4. Si sono definiti due testbench che controllino elementi negli angoli di una generica matrice 16x45, avente area 720:
   * Uno con un elemento che supera la threshold in ogni angolo della matrice, mentre tutti gli altri elementi sono inferiori;
   * Un altro con un elemento che supera la threshold soltanto in due angoli opposti della matrice, mentre tutti gli altri elementi sono inferiori;
5. Due testbench in cui:
   * Una sola riga ha valori maggiori della threshold (7x5, risultato 5);
   * Una sola colonna ha valori maggiori della threshold (4x10, risultato 4)
6. Una successione di reset-start durante l’esecuzione, seguito poi da un ulteriore start in contemporanea al falling edge di o\_done;
7. Casi particolari di una matrice con area 7, in cui:
   * La matrice è composta da una sola riga (1x7);
   * La matrice è composta da una sola colonna (7x1);
8. Un testbench in cui un solo elemento supera la threshold, in una matrice 7x3.

Per concludere, in quanto si ritiene utile al fine di comprendere le prestazioni del codice progettato, si vuole riportare, per ognuno dei 4 testbench forniti su Beep, il periodo di tempo intercorso fra il rising edge del segnale di start e il falling edge del segnale di done (quindi in pratica quanto tempo ci mette il codice a effettuare tutte le operazioni da quando effettivamente inizia a eseguire a quando effettivamente finisce). Questi dati sono stati raccolti in Post-Synthesis Functional Simulation:

* testbench\_delay: 2772.6 ns (la differenza rispetto agli altri tre è dovuta al fatto che non viene eseguita alcuna moltiplicazione);
* testbench2\_delay: 2802.6 ns;
* testbench3\_delay: 2802.6 ns;
* testbench4\_delay: 2802.6 ns.

1. Tutti i codici dei testbench sono stati caricati su https://github.com/iPhra/reti\_testbench/tree/master [↑](#endnote-ref-2)