Progetto di Prova Finale del corso di Reti Logiche

Luca Conterio - Matricola: 843317 - Cod. Persona: 10498418
Andrea Donati - Matricola: 847492 - Cod. Persona: 10497694

A.A. 2017/2018 - Prof. William Fornaciari

Contents

1 Overview del Progetto

Dopo un'analisi approfondita del testo della prova abbiamo deciso di realizzare l'architettura del progetto descrivendo, in behavioural architecture, una macchina a stati (*Macchina di Moore*). Tale macchina è il *core* della soluzione, ed è divisa in 11 stati, ognuno con compiti specifici.
I primi stati sono adibiti alla lettura dalla memoria dei dati riguardanti numero di righe, numero di colonne e valore di soglia. Negli stati successivi è implementato l'algoritmo di calcolo dell'area, che verrà trattato in un'altra sezione. Gli ultimi stati della macchina hanno il compito di scrivere in memoria il risultato dell'algoritmo e di segnalare la fine della computazione.
L'evoluzione del comportamento della macchina è sincronizzata con il *falling edge* del segnale di *clock*.

2 Algoritmo Utilizzato

2.1 Descrizione dell'Algoritmo

2.1.1 Prima Soluzione

Originariamente, per il calcolo dell'area del rettangolo minore abbiamo voluto ideare un algoritmo basato sui seguenti passi:

1. Scorrere la matrice contenente l'immagine per righe, così da individuare la riga con indice **minore** in cui è presente una cella con valore sopra la soglia.
2. Scorrere la matrice contenente l'immagine per righe, partendo dall'indirizzo dell'ultima cella e procedendo "al contrario", così da individuare la riga con indice **maggiore** in cui è presente una cella con valore sopra la soglia.
3. Scorrere la matrice contenente l'immagine per colonne, così da individuare la colonna con indice **minore** in cui è presente una cella con valore sopra la soglia.
4. Scorrere la matrice contenente l'immagine per colonne, partendo dall'ultima, così da individuare la colonna con indice **maggiore** in cui è presente una cella con valore sopra la soglia.
5. Una volta ottenuti tutti gli indici, applicare la formula:

   $$[(Rmax - Rmin) + 1] \times [(Cmax - Cmin) + 1]$$

   che ha come risultato l'area cercata. Siccome gli indici di scorrimento partono da 0, è necessario aggiungere "+1" al calcolo della distanza sia in termini di righe sia in termini di colonne.

Questo metodo di procedimento è però risultato molto scomodo da implementare. Dopo un primo tentativo, dunque, abbiamo deciso di cambiare la soluzione.

2.1.2 L'algoritmo "Definitivo"

L'algoritmo che abbiamo definitivamente scelto per l'implementazione si basa semplicemente sullo scorrere tutta la matrice, per righe, ed individuare gli indici minimi e massimi di righe e colonne, contenenti un valore sopra la soglia. In questo modo individuiamo i vertici del rettangolo di area minore contenente interamente l'immagine.
Una volta individuati questi indici, procediamo con il calcolo dell'area utilizzando la stessa formula descritta nella prima soluzione, ossia:

$$[(Rmax - Rmin) + 1] \times [(Cmax - Cmin) + 1]$$

Questo metodo di procedimento è stato analizzato e ripensato più volte, la soluzione sopra descritta è risultata la più adatta all'implementazione per quanto riguarda complessità del codice e frequenza di clock massima risultante.

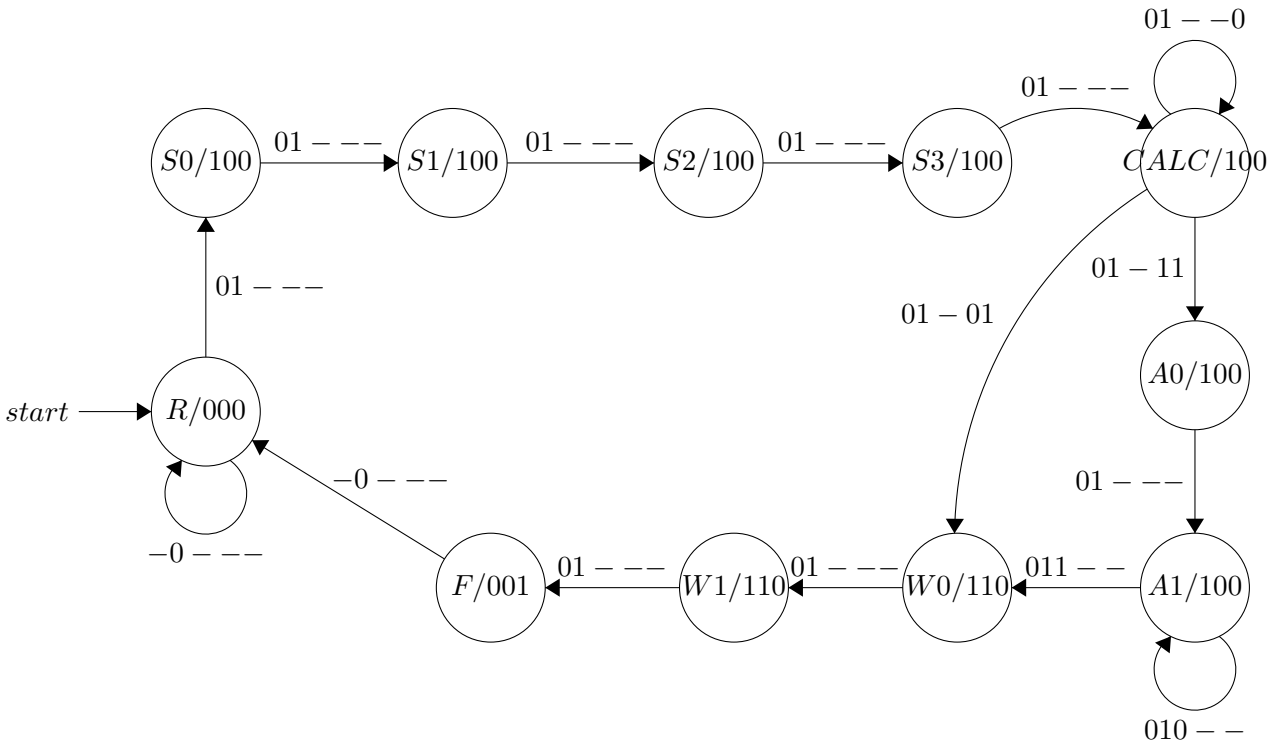
2.2 Complessità

La complessità della soluzione iniziale permetteva, a caso ottimo, di non scorrere tutta l'immagine. Tuttavia, abbiamo preferito un numero maggiore di cicli che però permettono di avere una frequenza maggiore alla quale i cicli sono effettuati.
La complessità dell'algoritmo definitivamente implementato è linearmente dipendente dalla dimensione (in termini di celle) della matrice in analisi. Si ha quindi, con una lunghezza dell'input pari ad n, una complessità che è sempre fissa a $\Theta(n)$.

2.3 Implementazione

La strategia di implementazione che abbiamo usato in definitiva è stata quella di scomporre il processo principale, parallelizzando il calcolo, in diversi altri processi secondari, ognuno con la sua mansione *"atomica"*.
Questi processi sono, ad esempio: *address_counter* (contatore degli indirizzi), *aggiornatori* (processi che a ogni colpo di clock aggiornano gli indici di massima e minima riga e colonna, indici di riga e colonna corrente), *calcolatori* (processi come quelli adibiti al calcolo della moltiplicazione "finale").
La moltiplicazione che calcola l'area è stata da noi implementata per via di somme successive.

3 Diagramma della Macchina



Ci sembra importante specificare inoltre che ogni stato ha implicitamente un arco verso lo stato di *RESET* che non abbiamo riportato per semplicità di lettura. Questo arco ha come vettore di input $1-----$, siccome c'è un 1 in corrispondenza del segnale *"i_rst"*.

3.1 Descrizione degli Input

Il vettore di bit di input riportato nel diagramma della macchina è un vettore composto da 5 segnali, quali:

- *i_rst*: è il segnale dato in input dal *testbench* , nel caso in cui si voglia fare ricominciare il calcolo.
- *start*: è un segnale definito da noi, che assume valore alto sul fronte di salita del segnale *"i_start"* dato in input e permane a 1 fino alla fine del calcolo.
- *fine_mult*: definito da noi, è un segnale il cui valore alto indica la fine della moltiplicazione adibita al calcolo dell'area.
- *trov*: è un segnale definito da noi, il cui valore alto indica che è stato trovato almeno un elemento al di sopra della soglia.
- *fine_imm*: sempre definito da noi, è un segnale il cui valore alto indica che si è finito di scorrere ogni cella dell'immagine data.

3.2 Descrizione degli Output

Il vettore di output è invece un vettore di 3 segnali, che sono:

- *o_en*: è il segnale che abilita la comunicazione con la memoria.
- *o_we*: "write enable", abilita la scrittura in memoria quando assume valore alto.
- *o_done*: segnala la fine della computazione.

Per una maggiore chiarezza nella spiegazione di questi segnali si rimanda alla documentazione fornita dai tutor.

3.3 Descrizione degli Stati

Di seguito una descrizione del comportamento di ogni stato:

- R**:** è lo stato di "reset", in cui la macchina si trova dall'inizio dell'elaborazione all'arrivo del segnale di *start*. In questo stato si reinizializzano eventualmente tutti i valori che durante l'evoluzione della macchina sono stati modificati, preparando l'architettura per un nuovo turno di calcolo. In questo stato è possibile arrivare in qualsiasi momento della computazione, nel caso in cui il segnale di *i_rst*, appunto, risulti alto.

- $S0$**:** In questo stato viene impostato l'indirizzo con il quale accediamo alla memoria ("*o_address*") al valore inziale: 2, leggendo il valore del numero di colonne.

- $S1$**:** Anche in questo stato viene impostato il valore di *o_address*, per ricevere gli ulteriori dati di inizializzazione (numero di righe dell'immagine, all'indirizzo 3).

- $S2$**:** in modo identico ai precedenti, in questo stato si riceve il valore di soglia.

- $S3$**:** : all'indirizzo viene assegnato il valore necessario a iniziare a scorrere l'immagine.

- $CALC$**:** la sigla sta per "Calcolo", è lo stato "centrale" dell'architettura, nel quale scorriamo l'immagine per righe e effettiuamo gli aggiornamenti per il calcolo degli indici di minima e massima riga e colonna.
  Quando lo scorrimento dell'immagine termina, se è stato trovato almeno un valore sopra la soglia si passa allo stato $A0$ che procede a calcolare l'area, mentre in caso contrario, si passa direttamente allo stato $W0$, che scrive 0 come risultato.

- $A0$**:** la sigla sta per "Area". In questo stato calcoliamo la dimensione in pixel dei lati del rettangolo contenente l'immagine.

- $A1$**:** Usando i risultati calcolati al punto precedente, si procede ad effettuare la moltiplicazione per somme successive.

- $W0/1$**:** la sigla sta per "Write", cioè gli stati in cui andiamo a scrivere in memoria il il risultato calcolato nello stato precedente.

- $Fine$**:** è alquanto autoesplicativo, alza il segnale "*o_done* " per notificare alla memoria la fine della computazione.

4 Testbench Personalizzati

Analizzando i testbench pubblici, abbiamo controllato come i valori della RAM fossero definiti e, prendendo esempio, abbiamo voluto creare i nostri propri testbench personalizzati, per controllare passo-passo la corretta evoluzione del progetto. Calcolare gli indirizzi e scrivere pedissequamente a mano i valori nel file ".vhd" non ci è sembrata un'opzione conveniente, perciò abbiamo scritto alcuni programmi in linguaggio Python per rendere il processo automatico.
Impostando un valore di soglia di 128 e facendo corrispondere agli indirizzi che volevamo essere facenti parte dell'immagine lo stesso valore, siamo riusciti ad ottenere dei testbench *copia-incollando* il risultato dei programmi all'interno della descrizione della RAM e impostando comme valore di *assertion* l'area da noi prevista.
Il procedimento descritto è stato utilizzato sia per realizzare una grande mole di testbench per la simulazione di tipo *behavioural*, sia per la simulazione in *post-sintesi*, sostituendo il risultato dei programmi in alcuni tra i file di test pubblici.
Allegare in calce tutti i file che abbiamo utilizzato sembrava essere una soluzione sconveniente per chiarezza e leggibilità, quindi di seguito descriviamo la struttura e il funzionamento dei programmi che abbiamo utilizzato.

4.1 Testbench "Triangolo"

Il programma seguente è studiato per generare una sequenza di indirizzi e i corrispondenti valori binari che rappresentano un triangolo rettangolo di altezza *altezza* e base *base*, dato a priori il numero di colonne e righe che compogono l'immagine.

```
F = open("test.txt", "w")
pos = 5          #valore assegnabile
cols = 24        #valore assegnabile
altezza = 5      #valore assegnabile
n = 0
for x in range (0,altezza):
    n = n + 1
    for a in range (0, n-1):
        F.write (str(pos)," => "\"10000000\"")
    pos += cols
```

4.2 Testbench "Random"

Questo programma, differentemente dal primo, non ha una figura precisa e preimpostata da "disegnare", ma si limita a inserire il valore binario 128 in indirizzi casuali della RAM.

```
import random

randArray = [];
num = input("Numero elementi da inserire: ")
# matrice 7x24 con valore di soglia pari a 128
cols = "2 => \"00011000\", "
rows = "3 => \"00000111\", "
soglia = "4 => \"10000000\", "
addrmin = 5     # restringo l'intervallo fra addrmin e addrmax
addrmax = 172   # per avere rettangoli di dimensioni differenti

F = open("test.txt", "w")

for i in range(0, num):
  R = random.randint(addrmin, addrmax) # estraggo numero casuale
  while R in randArray: # se il numero e' gia' stato estratto, riprovo (
    evita duplicati)
    R = random.randint(addrmin, addrmax)
  randArray.append(R)   # aggiungo il numero estratto ad un array

randArray.sort() # riordino l'array

F.write(cols)    # scrivo su file l'array generato
F.write(rows)
F.write(soglia)
for element in randArray:
  F.write(str(element))
  F.write(" => \"10000000\", ")
F.close()
```

4.3 Testbench di "Casi Limite"

I tetbench precedentemente descritti coprono molti tra i casi che necessitano analisi e hanno aiutato a scovare alcune parti fallacee. Tuttavia, una volta scoperta la presenza di un difetto, per accertarci del corretto funzionamento del nostro progetto abbiamo voluto scrivere a mano alcuni test che coprivano dei casi "limite" in cui, data la struttura che abbiamo elaborato, pensavamo potessero sorgere i problemi riscontrati.
I casi analizzati sono:

- Immagine composta unicamente da valori che si trovano **sotto** la soglia.
- Immagine composta unicamente da valori che si trovano **sopra** la soglia.
- Immagine contenente un unico valore sopra la soglia, nei casi di prima cella, ultima cella, cella centrale.
- Immagine di dimensioni massime consentite, cioè 255×255.
- Immagine composta da una sola riga od una sola colonna.
- Immagine di dimensioni accettabili in cui i valori sopra la soglia sono:
    1. Esattamente nella prima cella della matrice;
    2. Esattamente nell'ultima cella della matrice;
    3. Nelle posizioni di confine tra una riga e la successiva;
    4. Nelle posizioni di confine tra una colonna e la successiva.

Questa maniera di procedere ci ha permesso di ottenere, in conclusione, un'elaborazione corretta e coerente.

4.4 Testbench di Reset

Abbiamo modificato alcuni dei testbench precedentemente descritti, unitamente a quelli pubblici, in modo tale da avere l'arrivo di un segnale di *reset* dopo un certo lasso di tempo. Ci è servito particolarmente nel caso un cui il segnale di *reset* venga ricevuto dopo la fine della computazione (dopo avere alzato *o_done*).
Quindi in totale abbiamo realizzato 4 test che coprono i seguenti casi:

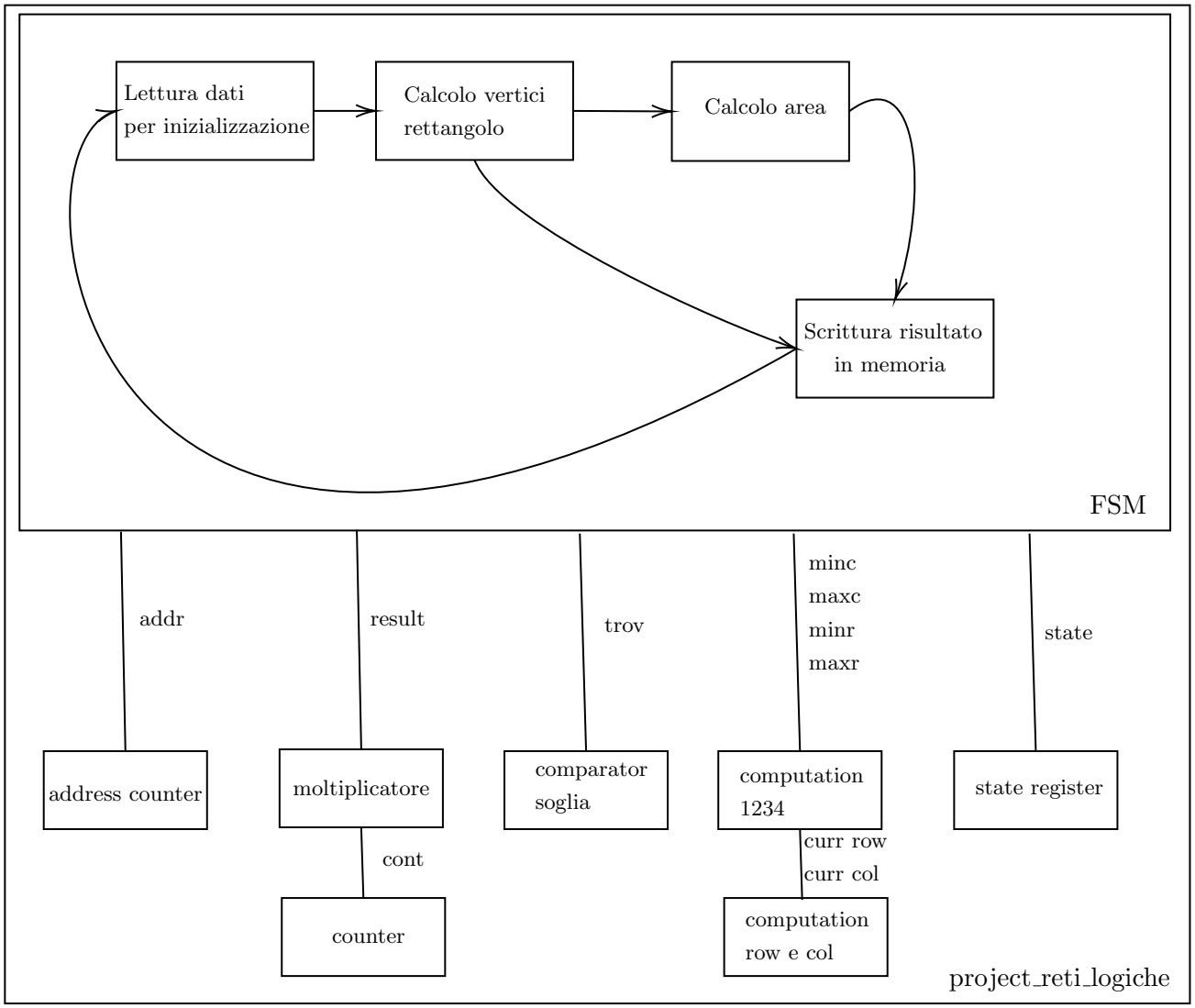
- I segnali di reset e di start arrivano distanziati di più di un ciclo di clock.
- I segnali di reset e di start che permangono in valore alto per più di un ciclo di clock.
- Il segnale di reset che viene alzato dopo la fine della computaizone, cioè dopo aver alzato il segnale *o_done*, obbligando la UUT (Unit Under Test) ad effettuare nuovamente il calcolo.
- Il segnale di reset alzato durante la computazione, obbligando anche in questo caso la UUT a ricominciare la computazione.

5 Frequenza Massima Raggiunta

Abbiamo più e più volte testato la nostra aschitettura in *post-sintesi* e *post-implementazione* ottenendo come frequenza massima raggiunta:

- *post-sintesi*: 192 MHz (5.2 ns)
- *post-implementazione*: 232 MHz (4.3 ns)

6 Schema a Blocchi



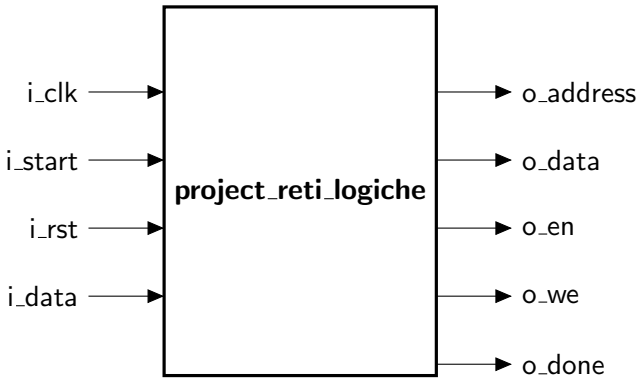


Figure 1: Entity of project_reti_logiche