Politecnico di Milano

Prova Finale di Reti Logiche

Autori:
Gregorio Galletti
Salvatore Fadda

Professore:
Fabrizio Ferrandi

Anno Accademico 2017-2018

Capitolo 1

Introduzione

Il gruppo é formato da:

- Gregorio Galletti: n° Matricola **847412**, Codice persona **10494196**
- Salvatore Fadda: n° Matricola **843745**, Codice persona **10520616**

Il gruppo, dopo una prima lettura della consegna della Prova Finale e delle spechifiche che il programma VHDL avrebbe dovuto rispettare, ha deciso per una struttura MonoComponente.
Come sará possibile osservare durante la lettura dei capitoli seguenti, questa scelta é stata effettuata prendendo subito in considerazione l'idea di gestire l'elaborazione e la scansione dell'immagine fornita in modo sequenziale, cella per cella, spostando inevitabilmente l'attenzione su una struttura "FSM-oriented".
La possibilitá di mantenere un ordine e una logica di esecuzione ben definita si é quindi concretizzata in quella che é a tutti gli effetti la versione finale del Progetto, e cioé un programma VHDL che gestisce una FSM attraverso 3 singoli process, gestendo input, output e elaborazione all'interno di essi.

Saranno quindi presenti capitoli che serviranno a una descrizione piú approfondita dell'architettura, a una descrizione dell'algoritmo utilizzato e alle varie prove e test effettuati durante lo sviluppo.
Per una piú specifica descrizione del codice VHDL, invece, si rimanda al reale codice sorgente (10494196.vhd) all'interno del quale sono presenti numerosi commenti a sostegno delle singole dichiarazioni, operazioni e scelte effettuate.

Capitolo 2

Architettura

All'interno di questo capitolo verrá descritta la struttura dell'architettura realizzata.
In particolare, verranno mostrate due figure: la prima (**2.1**) rappresenta l'entity dell'architettura, e quindi una classica *black box* chiamata project_reti_logiche (cioé il componente).
La seconda (**2.2**) rappresenta invece lo schema a blocchi funzionale dell'architettura, che mostra quindi la connessione e il rapporto tra segnali in ingresso, segnali in uscita, memoria RAM e il componente.

Si potrá notare come all'interno di project_reti_logiche siano presenti 3 blocchi, corrispondenti ai 3 process scritti in linguaggio VHDL, ognuno dei quali ha una funzione specifica:

current_state_output é il process principale, e cioé quello che si occupa di tutta la logica interna del programma: esso gestisce, a partire dallo stato corrente, quale deve essere quello successivo, e svolge le operazioni aritmetiche necessarie ai fini dell'elaborazione. Inoltre, é ovviamente sincronizzato con il fronte si salita del segnale di Clock i_clk.

next_state é il process che si occupa di eseguire effettivamente la transizione dallo stato corrente allo stato successivo (stabilito dal process sopra descritto), ad ogni ciclo di Clock.

p1 é il process che gestisce l'arrivo dei segnali di Reset e di Start, effettuando le operazioni necessarie per permettere al programma nel primo caso di ricominciare da capo, resettando i segnali interni necessari, mentre nel secondo caso di iniziare l'esecuzione vera e propria.

Successivamente verrá poi presentata la vera e propria logica del programma, seguendo la descrizione di una Finite State Machine, con opportuni schemi e descrizioni.

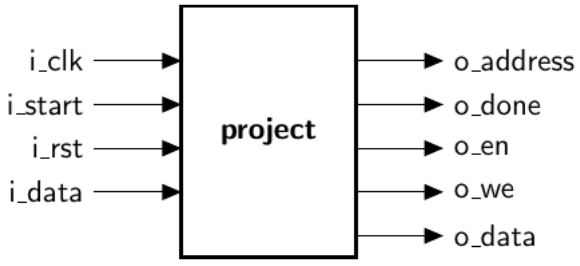


Figura 2.1: Entity di project_reti_logiche

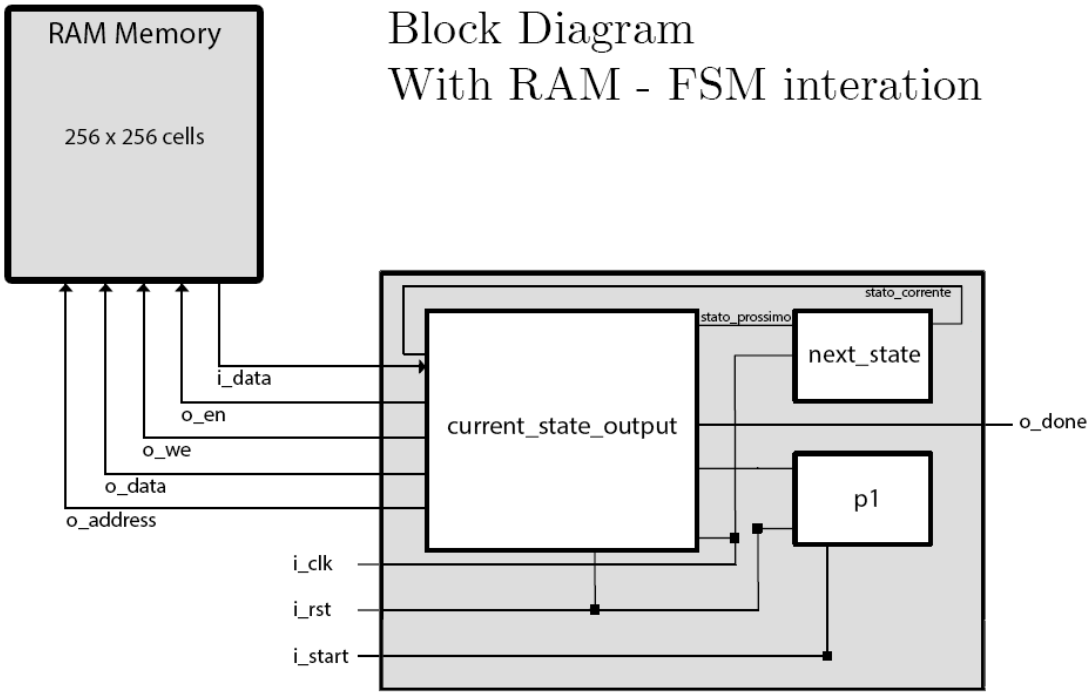
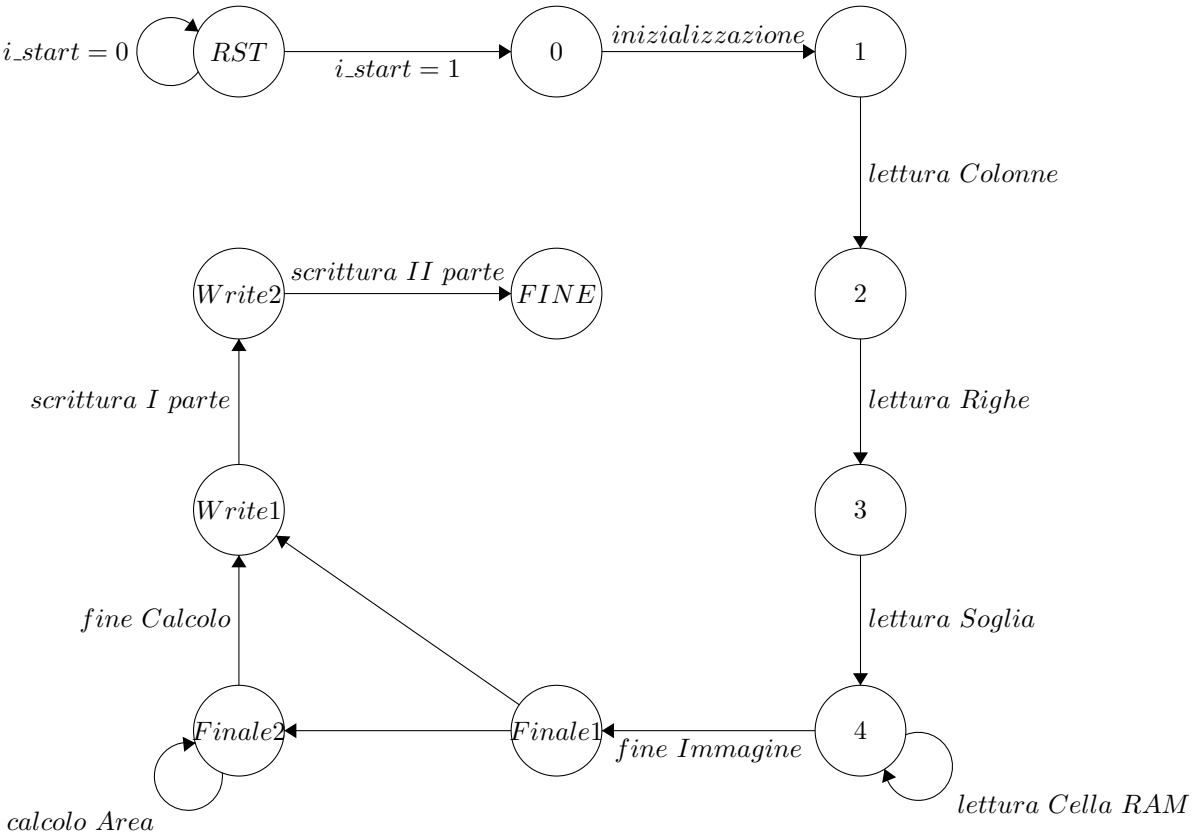


Figura 2.2: Schema a blocchi dell'architettura

Di seguito é riportato uno schema della FSM pensata per la progettazione, in cui ogni stato ha una transazione dopo un ciclo di clock, quindi aspettando *rising_edge(i_clk)*. Non é stata riportata, per pura comoditá ed organizzazione, la transazione verso lo statoRST (di Reset) che si verifica da ogni stato quando *i_rst=1*.



Qui il tool utilizzato per la creazione del diagramma della FSM precedente.

Le azioni dettagliate che vengono effettuate in ogni stato sono specificate in questo modo:

statoRST é lo stato in cui inizia l'esecuzione del programma, ed é anche lo stato di *reset*. La macchina rimane in questo stato fino a quando arriva il segnale di start, momento in cui viene posto lo stato successivo a stato0.

stato0 é lo stato in cui vengono resettati i segnali utilizzati all'interno del programma, in particolare:

- indirizzo viene posto a 2
- controllo (variable) viene posto a 0
- rCorrente viene posto a 0
- deltaR viene posto a 0

Gli altri segnali (cCorrente, righe, colonne, soglia,...) non sono inizializzati in quanto subiranno un assegnamento diretto durante l'esecuzione del programma, mente non subiranno modifiche del tipo $segnale <= segnale +' 1'$. Viene infine posto lo stato successivo a stato1.

stato1 é lo stato in cui viene effettivamente letta la prima cella di memoria, salvato il suo contenuto in una variabile chiamata *colonne* e posto l'indirizzo a 3 (*'0000000000000011'*) per poi passare allo stato successivo (stato2).

stato2 é lo stato in cui viene letta la seconda cella di memoria, salvato il suo contenuto in una variabile chiamata *righe* e posto l'indirizzo a 4 (*'0000000000000100'*) per poi passare allo stato successivo (stato3).

stato3 é lo stato in cui viene letta la terza cella di memoria, salvato il suo contenuto in una variabile chiamata *soglia* e posto l'indirizzo a 5 (*'0000000000000101'*) per poi passare allo stato successivo.

stato4 é lo stato in cui viene letta la quarta cella di memoria, e cioé la prima cella che rappresenta l'immagine, e salvato il suo contenuto in una variabile chiamata *valore*, che servirá poi per l'analisi effettiva dell'immagine e della figura contenuta.

statoFinale1 é lo stato in cui viene posta a 0 l'area, ed effettuato il controllo per stabilire se deve essere calcolata l'area: se questo é necessario, lo stato prossimo viene impostato a statoFinale2, altrimenti l'area rimane 0 e lo stato prossimo é impostato a statoWrite1.

statoFinale2 é lo stato in cui viene calcolata l'area del piú piccolo rettangolo che circoscrive la figura rappresentata, iterando su questo stato e sommando all'area la quantitá $cMax - cMin + '1'$, che rappresenta la "base" del rettangolo.

statoWrite1 é lo stato in cui viene scritta la parte meno significativa (i bit da 7 a 0) dell'area nella cella di memoria con indirizzo 0, e posto lo stato successivo a statoWrite2.

statoWrite2 é lo stato in cui viene scritta la parte piú significativa (i bit da 15 a 8) dell'area nella cella di memoria con indirizzo 1, posto o_done a 1 e impostato infine lo stato successivo a statoFINE.

statoFINE é lo stato conclusivo, in cui o_done viene semplicemente riportato a 0 dopo un ciclo di clock. Lo stato prossimo viene impostato a statoRST per poter essere pronto a iniziare una nuova esecuzione.

Capitolo 3

Algoritmo

Si procede alla descrizione dell'algoritmo utilizzato; molti nomi di segnali e variabili sono autoesplicativi, per altri invece verrá fornita una breve descrizione.

1. L'algoritmo parte da una situazione in cui i segnali sono tutti inizializzati a 0, esclusi quelli che assumeranno valori significativi durante l'esecuzione: ad esempio, il segnale *colonne* non viene posto a 0, in quanto durante la lettura della memoria esso assumerá il valore di *i_data* letto all'indirizzo corrispondente.

2. Viene letto il numero di colonne, il numero di righe e la soglia dell'immagine, rispettivamente in 3 signals chiamati *colonne*, *righe*, *soglia*, e vengono inizializzati i segnali *cMin* (= colonne - '1') e *rMin* (= righe - '1') che serviranno successivamente.

3. A questo punto inizia la parte iterativa dell'algoritmo, che procede a una semplice scansione della memoria RAM (e quindi della figura) in modo lineare.
   Viene quindi letta la cella corrispondente all'*indirizzo* e assegnato il relativo valore al segnale *valore*, appunto, e incrementato subito l'*indirizzo* di 1. Qui il primo controllo, cioé se il segnale *rCorrente* (inizialmente posta a 0) equivale a *righe* significa che le righe sono state "sforate" e quindi che l'iterazione deve terminare e passare alla fase 5; altrimenti, viene effettuato un ulteriore controllo che serve per identificare la colonna e la riga dell'immagine corrispondenti alla cella di memoria appena letta.
   La variabile *controllo* parte da 0 e corrisponde alla colonna corrente *cCorrente*, viene incrementata a ogni interazione e resettata ogni volta che *controllo* = *colonne* (significa che ho "sforato" le colonne e che quindi devo incrementare di 1 *rCorrente*).

4. Per il valore letto nella fase precedente, vengono effettuati i relativi controlli: prima di tutto, se esso é minore della soglia non vengono effettuate operazioni, trascurando quindi questa cella di memoria. Altrimenti, vengono effettuati 4 controlli successivi:

   - se $cCorrente < cMin$ significa che devo aggiornare la colonna minima della figura, e quindi $cMin = cCorrente$;
   - se $cCorrente > cMax$ significa che devo aggiornare la colonna minima della figura, e quindi $cMax = cCorrente$;
   - se $rCorrente < rMin$ significa che devo aggiornare la colonna minima della figura, e quindi $rMin = rCorrente$;
   - se $rCorrente > rMax$ significa che devo aggiornare la colonna minima della figura, e quindi $rMax = rCorrente$;

   Ovviamente, tutti questi controlli possono essere veri cosí come falsi, quindi posso sia aggiornare tutti e 4 i valori cosí come possono restare invariati.
   L'algoritmo riprende dalla fase 3.

5. Una volta finita la scansione della memoria, l'algoritmo controlla se é necessario il calcolo dell'area oppure no, tramite il controllo $if(cMin > cMax \ or \ rMin > rMax)$: in questo caso significa che cMax, cMin, ... non sono stati modificati e che quindi non é stato trovato un valore dell'immagine maggiore del valore di soglia, quindi il calcolo dell'area non va effettuato.
   Altrimenti, si procede al calcolo tramite somme successive, sommando deltaR ($rMax - rcMin +' 1'$) volte il valore $cMax - cMin +' 1'$ all'area.

6. Ora é possibile scrivere in memoria, negli opportuni indirizzi, l'area appena trovata. Vengono quindi scritti gli 8 bit meno significativi e gli 8 bit piú significativi, e infine viene posto il segnale di o_done a 1.

7. Il segnale di o_done viene rimesso a 0, ció segnala quindi che la macchina ha finito l'esecuzione ed é pronta per effettuarne una nuova.

Capitolo 4

Test

Oltre ai 4 Test Bench forniti, é stato deciso di scrivere un programma in linguaggio C che, in modo completamente casuale, crea un vettore rappresentante una memoria RAM con un numero di colonne, un numero di righe e una soglia compresi tra 0 e 255.
Il programma assegna quindi a ogni cella della memoria un valore, anch'esso casuale e compreso tra 0 e 255. Infine, il programma "risolve" il problema modificando gli assert del TestBench con lo stesso procedimento utilizzato per il progetto in VHDL, in modo da poter verificare la correttezza del risultato ottenuto.
Sono stati quindi utilizzati 5 test bench generati in questo modo, per coprire una vasta gamma di **casi intermedi** (cioé con un numero medio di righe, colonne e soglia) e 5 test bench strutturati in modo da coprire i **casi limite** (cioé con un numero di righe, colonne e soglia ben definito).
Nel dettaglio:

- **test_pers_MAX** é il caso in cui le righe e le colonne sono 255, mentre la soglia é casuale
- **test_pers_ONE_ROW** é il caso in cui c'é solo una riga e un numero casuale di colonne (e di soglia)
- **test_pers_ONE_COL** é il caso in cui c'é solo una colonna e un numero casuale di righe (e di soglia)
- **test_pers_SOGLIA_MIN** é il caso in cui le righe e le colonne sono casuali, mentre la soglia é 0
- **test_pers_SOGLIA_MAX** é il caso in cui le righe e le colonne sono casuali, mentre la soglia é 255

Per completezza si rimanda alla Sezione 6 (**Appendice**) dove é presente il codice C utilizzato per la creazione di questi TestBench personalizzati, mentre vengono omessi i codici di tutti i test in quanto renderebbero esageratamente lungo il file e non risulterebbero significativi.
A titolo di esempio, viene presentato un solo test, gli altri seguono ovviamente la stessa impostazione.

Capitolo 5

Risultati

Lo sviluppo e l'ottimizzazione del programma che verrá descritta in questo capitolo é sempre stata affiancata a un rigido controllo della correttezza, effettuato tramite i casi di test descritti nel capitolo precedente.
Ogni versione presentata, infatti, mostrava sempre una correttezza di esecuzione (con i 4 testbench forntiti) per quanto riguarda sia la Pre che la Post Synthesis.

5.1 Prima versione

Inizialmente non era stato pensato nessun metodo che potesse aumentare la frequenza di CLOCK, in quanto non di principale interesse (si mirava prima una correttezza formale del programma in Pre e Post Synthesis, e poi a una eventuale ottimizzazione).
Il primo prototipo del programma presentava un calcolo della riga e della colonna attuale con all'interno delle moltiplicazioni; queste causavano un rallentamento e pertanto é stato rivoluzionato l'algoritmo fino ad arrivare alla versione finale (giá descritta sopra).
La **frequenza maggiore** raggiunta da questa versione del programma dopo la Implementation e fissando un Constraint con periodo di CLOCK di $7,50ns$ era di circa: $f =\sim 133MHz$.

5.2 Prima (minima) Ottimizzazione

Dopo la consultazione di manuali, dispense e documentazioni ufficiali Xilinx (in particolare questa sul Timing), si é pensato di utilizzare la codifica One-Hot per gli stati della macchina, dato che durante la sintesi l'ambiente di sviluppo utilizzava la codifica classica Binary. Questo ha velocizzato il programma permettendo di diminuire il periodo di CLOCK a $7,30ns$ e portando quindi la **frequenza massima** a $f =\sim 137MHz$.

5.3 Seconda Ottimizzazione

Nonostante i giá buoni risultati ottenuti, la seconda versione del programma conteneva ancora due operazioni di moltiplicazione che (nonostante l'operazione in sé venga eseguita in poco tempo) rallentavano l'esecuzione generale del programma.
Queste erano, nel dettaglio, il calcolo della dimensione della figura con $dimensione = righe * colonne$ e il calcolo dell'area con $Area = (C_{max} - C_{min} + 1) * (R_{max} - R_{min} + 1)$.

5.3.1 Calcolo dell'area

Per il calcolo dell'area si é deciso di aggiungere uno stato non presente nelle versioni precedenti, statoFinale2, sfruttato appunto per il calcolo dell'area mediante delle somme successive (giá descritto precedentemente). Questa modifica ha quindi permesso di raggiungere frequenze molto piú elevate,

aumentando in quantitá minime e quindi trascurabili il tempo di Simulation. Periodo e **frequenza** di CLOCK risultanti sono, rispettivamente: $5,60ns$ e $f =\sim 179MHz$.

5.3.2 Calcolo della dimensione

Il calcolo della dimensione della figura é fondamentale per capire quando fermarsi e quindi passare dallo stato4 allo statoWrite1. É stata quindi modificata l'implementazione di questo calcolo inizialmente presente in stato3, scomponendo la moltiplicazione in un successione di somme presente nel (nuovo) stato chiamato statoDim. La versione modificata prevede quindi iterazioni su statoDim per *righe* volte, effettuando l'operazione $dimensione = dimensione + colonne$.
Il risultato di questa modifica ha diminuito il periodo di CLOCK a $5,12ns$, e quindi a una **frequenza massima** di: $f =\sim 195MHz$.

5.3.3 Ciclo Clock nello statoFinale2

In statoFinale2 era presente un controllo non ottimizzato, che richiedeva una iterazione in piú durante il calcolo dell'area. Ció portava a uno spreco di tempo, e riscrivendo il controllo nel modo giusto il periodo di CLOCK ha raggiunto i $4,85ns$ e la **frequenza massima** i $f =\sim 206MHz$.

5.4 Terza Ottimizzazione

Calcolo della dimensione integrato

Si é notato come la creazione di uno stato apposito per il calcolo della dimensione aumentasse la frequenza massima, ma non quanto un calcolo piú rapido e intelligente come quello presente in questa versione.
Dovendo giá iterare sullo stato4 per la lettura della memoria RAM, é risultato efficiente integrare la serie di somme successive all'interno di questo stato (eseguendo i controlli necessari) in modo da riuscire a eliminare lo statoDim. Questo ragionamento si é rivelato corretto, abbassando il periodo di CLOCK a $4,41ns$ e aumentando quindi la **frequenza massima** a: $f =\sim 227MHz$.

5.5 Quarta Ottimizzazione

A questo punto si é scoperto che la dichiarazione esplicita della codifica One-Hot descritto precedentemente (**5.2**) portasse a un rallentamento, probabilmente dovuto al numero totale di stati, aumentato a causa delle modifiche appena descritte. Eliminandola, il periodo di CLOCK é diminuito fino a raggiungere $4,06ns$ e la **frequenza massima** é di conseguenza aumentata fino a $f =\sim 246MHz$.

5.6 Gestione Segnale di Reset

Dopo tutte queste modifiche descritte, é stato notato come la gestione del segnale di Reset fino ad allora implementata non riuscisse a cambiare effettivamente lo stato corrente e lo stato prossimo (questo é stato visto modificando il testbench3_delay.vhd "ripetendo" l'arrivo del segnale RST e successivamente START nel mezzo dell'esecuzione del programma).
Per rendere effettivamente corretta la funzione é stato necessario aggiungere un nuovo stato, statoRST, in cui il programma rimane e attende il segnale di inizio.
Ovviamente, l'aggiunta di un nuovo stato e dei controlli necessari ha alzato il periodo di CLOCK a $4,277ns$ e abbassato la **frequenza massima** a $f =\sim 234MHz$.

5.7 Rimozione del calcolo della dimensione

Per compensare l'aumento del periodo di clock dovuto alla modifica precedente, si é pensato di ridurre al minimo i calcoli algebrici all'interno del programma.

É stato quindi eliminato il calcolo della dimensione descritto prima e di modificare quindi il controllo per l'iterazione in stato4, associandola al segnale rCorrente e verificando le condizioni necessarie; tutto questo ha portato all'eliminazione di 2 segnali e di 2 operazioni algebriche (contatore, utilizzato per contare le celle dell'immagine analizzate, e dimensione).

5.8 Versione Finale

Le ottimizzazioni precedentemente illustrate hanno quindi reso possibile la riduzione del periodo di CLOCK a $4,033ns$, e di conseguenza a una ***frequenza massima finale*** di $f =\sim 248MHz$.

*Tutti i risultati illustrati in questa sezione sono da considerarsi indicativi, in quanto é possibile che si verifichino delle lievi differenze durante le varie implementazioni. A titolo di esempio, si consideri che rifacendo l'implementazione della stessa versione descritta in **5.8** in vista della consegna del progetto, i risultati osservabili erano di $T = 3,969$, $f =\sim 252MHz$.*

Capitolo 6

Appendice

6.0.1 Programma C

Codice sorgente in linguaggio C del programma di creazione di TestBench:

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#define max_dim 65536

/*
Il programma crea un vettore della dimensione massima della memoria RAM data. Questo vettore,
     inizialmente formato dai soli valori 0, viene riempito casualmente con dei valori anch'essi
     casuali (chiaramente compresi tra 0 e 255). Tutti i numeri vengono convertiti in binario e
     scritti all'interno del testbench; anche l'area del rettangolo risultante viene calcolata,
     codificata e scritta su file negli assert separando bit pi significativi e bit meno
     significativi*/

void inizializza();       //le funzioni verranno specificate dopo
void genera();
void scrivi();
void toBinary(int);
void toBinary16(int);
void calcola();

int numeri[max_dim];
int conv[8];
int conv16[16];
FILE *scrivere, *leggere;
int col, row, soglia, dim, daGenerare, posizione, odd, temp, cMin, rMin, cCorrente, rCorrente;
int cMax=0, rMax=0;

int main()
{
  char buf[200];  char *res;  //necessari per la lettura da file

    //apro un file contenente il primo pezzo del testbench fino alla dichiarazione della memoria
     RAM, e lo aggiungo al MIO testbench
    scrivere= fopen("test_pers_1.vhd", "w");
    leggere=fopen("daScrivere1.txt", "r");

  while(1) {
      res=fgets(buf, 200, leggere);
    if( res==NULL )
      break;
    fprintf(scrivere, "%s", buf);   //"ricopio" la prima parte nel mio file
    }
  fclose(scrivere);

    inizializza();            //chiamo le funzioni necessarie per scrivere tutta la dichiarazione
     delle celle della memoria RAM
    genera();
    scrivi();

    //apro un file contenente il resto del testbench fino agli assert, e lo aggiungo al MIO
     testbench
    scrivere= fopen("test_pers_1.vhd", "a");
    leggere=fopen("daScrivere2.txt", "r");

  while(1) {
```

```
48        res=fgets(buf, 200, leggere);
49      if( res==NULL )
50        break;
51      fprintf(scrivere, "%s", buf);   //"ricopio" la seconda parte nel mio file
52      }
53    fclose(scrivere);
54
55      calcola();      //scrivo infine l'area che il programma VHDL deve trovare negli assert, e
       scrivo anche la conclusione del testbench
56  }
57
58
59  void inizializza() {
60    //genero i numeri di righe, colonne e soglia e inizializzo a 0 tutti gli elementi del vettore
61    srand(time(NULL));
62
63    col=rand() % 256;   //genero il numero di colonne
64    cMin=col-1;       //serve per il calcolo finale dell'area
65
66    row=rand() % 256;   //genero il numero di righe
67    rMin=row-1;       //serve per il calcolo finale dell'area
68
69    soglia=rand() % 256;  //genero la soglia
70
71    dim=row*col;      //calcolo la grandezza del vettore, cioe il numero di celle della RAM
72
73    for(i=0;i<dim;i++)    //inizializzo
74      numeri[i]=0;
75    return;
76  }
77
78  void genera(){
79    //riempio il vettore con alcuni numeri casuali
80    srand(time(NULL));
81    daGenerare=rand() % dim;              //ho un numero massimo di numeri da generare
82
83    int i=0;
84    while (i<daGenerare){
85      posizione=rand() % dim;             //scelgo la posizione a caso tra tutte
86      numeri[posizione]=rand() % 256;         //in quella posizione metto un numero a caso, in
       questo modo me ne sbatto se ho duplicati perch+ li sovrascrivo
87      i++;                      //aumento il contatore
88    }
89    return;
90  }
91
92  void scrivi(){
93    //scrivo sul file solo i numeri del vettore diversi da 0
94    scrivere= fopen("test_pers_1.vhd", "a");    //parte scritta dal programma scritta casualmente,
       la prima parte c'e gia
95
96    //scrivo le prime 3 celle "speciali" della RAM
97    fprintf(scrivere, "(2 => \"");    toBinary(col);
98    fprintf(scrivere, ", 3 => \"");   toBinary(row);
99    fprintf(scrivere, ", 4 => \"");   toBinary(soglia);
100
101   int i;
102   for(i=5;i<dim;i++){
103     if(numeri[i]!=0){
104       fprintf(scrivere, ", %d => \"", i);       //necessario per la formattazione
105       toBinary(numeri[i]);              //converto in binario il numero
106     }
107   }
108   fprintf(scrivere, ", others => (others =>'0'));");    //metto a 0 tutto il resto
109   fclose(scrivere);
110   return;
111 }
112
113 void toBinary(int n){
114   //converto un numero intero in un numero binario a 8 bit, e lo scrivo su file
115   if(n==1){
116     fprintf(scrivere, "00000001\"");
117     return;
118   }
119
120   temp=n;
121   int i=0;
122   while (i<8){
123     odd=temp%2;       //classico algoritmo per la conversione in binario
124     temp=temp/2;
125     conv[i]=odd;
126     i++;
```

```
127   }
128
129   for(i=7;i>=0;i--)
130     fprintf(scrivere, "%d",conv[i]);      //scrivo su file il risultato
131   fprintf(scrivere, "\"");
132   return;
133 }
134
135 void calcola(){
136   //calcolo l'area del rettangolo per scriverla negli assert
137   int i;
138   for(i=5;i<dim;i++){
139     if(numeri[i]>=soglia){
140       cCorrente=(i-5)%col;
141       rCorrente=(i-5)/col;
142       if(cCorrente>cMax)  cMax=cCorrente;
143
144       if(cCorrente<cMin)  cMin=cCorrente;
145
146       if(rCorrente>rMax)  rMax=rCorrente;
147
148       if(rCorrente<rMin)  rMin=rCorrente;
149     }
150   }
151
152   if(cMax>cMin && rMax>rMin)
153     toBinary16((cMax-cMin+1)*(rMax-rMin+1));    //se l'area e' accettabile la converto e la scrivo
        sul file
154   else
155     toBinary16(0);                  //altrimenti converto e scrivo 0 su file
156   return;
157 }
158
159 void toBinary16(int n){
160   //converto un numero intero in un numero binario a 16 bit, e lo scrivo su file con la parte
        restante del testbench
161   scrivere= fopen("test_pers_1.vhd", "a");
162   fprintf(scrivere, "\"");
163
164   if(n==0){
165     fprintf(scrivere, "00000000\" report \"FAIL high bits\" severity failure;\nassert RAM(0) =
        \"00000000\" report \"FAIL low bits\" severity failure;\nassert false report \"Simulation
        Ended!, test passed\" severity failure;\nend process test;\nend projecttb;");
166     return;
167   }
168
169   if(n==1){
170     fprintf(scrivere, "00000000\" report \"FAIL high bits\" severity failure;\nassert RAM(0) =
        \"00000001\" report \"FAIL low bits\" severity failure;\nassert false report \"Simulation
        Ended!, test passed\" severity failure;\nend process test;\nend projecttb;");
171     return;
172   }
173
174   temp=n;
175   int i=0;
176   while (i<16){
177     odd=temp%2;          //classico algoritmo utilizzato anche prima
178     temp=temp/2;
179     conv16[i]=odd;
180     i++;
181   }
182
183   //scrivo ora i primi 8 bit (meno significativi)
184   for(i=15;i>=8;i--)
185     fprintf(scrivere, "%d",conv16[i]);
186   fprintf(scrivere, "\" report \"FAIL high bits\" severity failure;\nassert RAM(0) = ");
187
188   fprintf(scrivere, "\"");
189   //scrivo ora gli ultimi 8 bit (piu significativi)
190   for(i=7;i>=0;i--)
191     fprintf(scrivere, "%d",conv16[i]);
192   fprintf(scrivere, "\" report \"FAIL low bits\" severity failure;\nassert false report \"
        Simulation Ended!, test passed\" severity failure;\nend process test;\nend projecttb;");
193
194   fclose(scrivere);
195   return;
196 }
```

test1.c

6.0.2 TestBench esempio

Codice sorgente in linguaggio VHDL del primo TestBench generato: in questo caso si tratta di un'immagine di *'10000110'*=**134 colonne**, *'11101110'*=**238 righe**, con **soglia** pari a *'10001011'*=**139** e con 99 celle che hanno un valore significativo ai fini dell'elaborazione (diverso da 0):

```
----------------------------------------------------------------------------------
-- Company:
-- Engineer:
--
-- Create Date: 12.12.2017 17:48:44
-- Design Name:
-- Module Name: FSM_testbench - Behavioral
-- Project Name:
-- Target Devices:
-- Tool Versions:
-- Description:
--
-- Dependencies:
--
-- Revision:
-- Revision 0.01 - File Created
-- Additional Comments:
--
----------------------------------------------------------------------------------
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
use ieee.std_logic_unsigned.all;

entity project_tb is
end project_tb;


architecture projecttb of project_tb is
constant c_CLOCK_PERIOD    : time := 15 ns;
signal   tb_done         : std_logic;
signal   mem_address       : std_logic_vector (15 downto 0) := (others => '0');
signal   tb_rst            : std_logic := '0';
signal   tb_start      : std_logic := '0';
signal   tb_clk            : std_logic := '0';
signal   mem_o_data,mem_i_data  : std_logic_vector (7 downto 0);
signal   enable_wire       : std_logic;
signal   mem_we        : std_logic;

type ram_type is array (65535 downto 0) of std_logic_vector(7 downto 0);
signal RAM: ram_type := (2 => "10000110", 3 => "11101110", 4 => "10001011", 8 => "10000110", 66 =>
      "01000100", 350 => "00010011", 635 => "11111101", 652 => "10101001", 750 => "11000001", 916
     => "00110101", 1006 => "00111101", 1394 => "10100111", 1428 => "10010111", 1640 => "10100010",
      1802 => "00010001", 1830 => "10111101", 1876 => "00101000", 1904 => "11011111", 2445 => "
     00010010", 2542 => "01110101", 4152 => "01011001", 4354 => "01011011", 4494 => "01000101",
     6623 => "11001011", 7824 => "00011101", 8082 => "11000111", 8271 => "11101100", 8641 => "
     01010100", 8677 => "10000101", 8830 => "10001101", 8888 => "01001011", 9165 => "01100001",
     9377 => "10111100", 10104 => "11111100", 10292 => "01100111", 10506 => "10110010", 10619 => "
     00110010", 10669 => "10010011", 10843 => "00110110", 10859 => "01011010", 11069 => "01100110",
      11863 => "00000111", 12944 => "11110011", 13256 => "11010110", 13317 => "11101100", 13336 =>
     "11110111", 13430 => "11010011", 13908 => "00001010", 14465 => "01100110", 14587 => "11100011"
     , 14593 => "11010001", 14661 => "00101001", 15114 => "00100100", 15190 => "00111111", 15213 =>
      "01001111", 15249 => "01111001", 15315 => "00010011", 15373 => "01010011", 16478 => "01001110
     ", 16519 => "10001111", 16969 => "01100000", 17271 => "00101100", 17308 => "10100110", 17892
     => "01101111", 17970 => "11100111", 18797 => "11000001", 18822 => "11101110", 19260 => "
     11011101", 19595 => "00100110", 19772 => "10001011", 20434 => "00001100", 20583 => "11110110",
      21055 => "01000011", 21247 => "00011110", 22050 => "01001001", 22091 => "01100110", 22183 =>
     "01000100", 22683 => "01010100", 22759 => "01101000", 22761 => "01110001", 23030 => "11011101"
     , 23242 => "00100110", 23384 => "00001001", 23576 => "11110101", 23991 => "00010100", 25273 =>
      "01111110", 25854 => "10000101", 26465 => "01010111", 26641 => "11100011", 27016 => "10111001
     ", 27211 => "01001001", 27509 => "10010110", 27626 => "01100110", 27736 => "01011010", 27854
     => "11100100", 27863 => "01100111", 28499 => "10011000", 28901 => "11011111", 29524 => "
     10100111", 30287 => "10001001", 30293 => "10000000", 31130 => "10100111", others => (others
     =>'0'));
component project_reti_logiche is
    port (
            i_clk         : in  std_logic;
            i_start       : in  std_logic;
            i_rst         : in  std_logic;
            i_data        : in  std_logic_vector(7 downto 0); --1 byte
            o_address     : out std_logic_vector(15 downto 0); --16 bit addr: max size is 255*255
    + 3 more for max x and y and thresh.
            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 component project_reti_logiche;


begin
  UUT: project_reti_logiche
  port map (
      i_clk       => tb_clk,
          i_start       => tb_start,
          i_rst       => tb_rst,
          i_data      => mem_o_data,
          o_address   => mem_address,
          o_done        => tb_done,
          o_en    => enable_wire,
      o_we  => mem_we,
          o_data    => mem_i_data
);

p_CLK_GEN : process is
  begin
    wait for c_CLOCK_PERIOD/2;
    tb_clk <= not tb_clk;
  end process p_CLK_GEN;


MEM : process(tb_clk)
   begin
    if tb_clk'event and tb_clk = '1' then
     if enable_wire = '1' then
      if mem_we = '1' then
       RAM(conv_integer(mem_address))              <= mem_i_data;
       mem_o_data                      <= mem_i_data after 1 ns;
      else
       mem_o_data <= RAM(conv_integer(mem_address)) after 1 ns;
      end if;
     end if;
    end if;
   end process;


test : process is
begin
wait for 100 ns;
wait for c_CLOCK_PERIOD;
tb_rst <= '1';
wait for c_CLOCK_PERIOD;
tb_rst <= '0';
wait for c_CLOCK_PERIOD;
tb_start <= '1';
wait for c_CLOCK_PERIOD;
tb_start <= '0';
wait until tb_done = '1';
wait until tb_done = '0';
wait until rising_edge(tb_clk);

assert RAM(1) = "01101001" report "FAIL high bits" severity failure;
assert RAM(0) = "10001110" report "FAIL low bits" severity failure;

assert false report "Simulation Ended!, test passed" severity failure;
end process test;
end projecttb;
```

test_pers_1.vhd