Documentazione del Progetto Finale di Reti Logiche Mato e Massini

Progetto realizzato da:

Kevin Mato, Matricola: 845726 Codice persona 10499864

Luca Massini, Matricola: 844049 Codice persona: 10504929

Questo progetto è una funzione che calcola l'area in pixel, di una porzione di immagine contenuta all'interno di una memoria RAM esterna. Il progetto è stato realizzato come da specifica utilizzando il linguaggio VHDL e verificando la possibilità di sintesi, come da specifica, sulla board Xilinx xc7a200tfbg484-1.

Scelte Progettuali

Algoritmo

L'algoritmo utilizzato scritto in linguaggio Matlab.

```
function [risultato]=algo(arg)
    nr=0;%valore di default
    nc=0;%valore default per numero colonne
    s=0; %soglia

    pos=2; %reset contatore address
    col=0; %reset contatore colonna
    riga=0; %reset contatore riga
    lato1=0; %valori default ram
    lato2=0;
    riga1=0;
    riga2=0;

    nc=arg(pos);%assegnamenti richiedendo indirizzi di memoria
    pos=pos+1;%incremento contatore di address
    nr=arg(pos);
    pos=pos+1;
    s=arg(pos);
    looper=(nr*nc)+pos; %calcolo del numero di cicli

    primo=1; %flag per indicare se non è stato già trovato un pixel sopra
soglia, in vhdl è stato  indicato come isFirst
    pos=pos+1;

    while(pos<=looper)
            if(arg(pos)>=s)
                if(primo==1)
                    lato1=col;
                    lato2=col;
                    riga1=riga;
                    riga2=riga;
                    primo=0;

                elseif(primo==0)
                    if(col<lato1)
                        lato1=col;
                    elseif(col>lato2)
                        lato2=col;
                    end

                    if(riga>riga2)
                        riga2=riga;
                    end
                end
```

```
            end
            col=col+1; %aumenta I contatori della colonna
            pos=pos+1;
            if(col>nc) %verifica che la colonna non sia outOfBound
                col=1;
                riga=riga+1;
            end
     end
    if(primo==1)
        risultato=0;
    else
        el=lato2-lato1+1;
        el2=riga2-riga1+1;
        risultato=el*el2;
    end
else
    risultato=0
end
```

L'implementazione è incominciata dalla definizione di un algoritmo per la ricerca, che in ogni possibile soluzione ha complessità lineare nel caso pessimo, per la stessa natura del problema. L'algoritmo usato per la nostra versione del progetto non è stato ritenuto fondamentale per lo scopo didattico che a nostro avviso si prefiggeva il progetto, così abbiamo optato per una scansione lineare della memoria mirando all'utilizzo dei componenti imparati a lezione. L'algoritmo scorre le posizioni all'interno della memoria e verifica se il pixel a quella posizione è sopra soglia, se sì confronta le sue coordinate in righe e colonne e ne salva il valore se determina uno spostamento del perimetro della figura interessata, in base che sposti la riga più bassa o la più alta o il lato destro o il sinistro del rettangolo che circonda l'area in pixel.

Architettura

Prima di tutto abbiamo pensato alla realizzazione del progetto attraverso una macchina a stati finiti e di dividere il progetto in due sezioni, una chiamata "Datapath" dove i componenti sono controllati dalla logica della macchina a stati e dove i veri dati dell'immagine vengono raccolti ed elaborati. La seconda è la FSM vera e propria, la quale comunica con il Datapath attraverso un segnale che è il risultato di un componente logico della ALU e modifica i suoi stati secondo un percorso sul grafo determinato da quest'ultimo segnale, e impartisce degli ordini al datapath attraverso dei segnali di controllo.

Analizziamo il Datapath nel dettaglio: questa sezione si occupa di implementare gli elementi dell'algoritmo più "passivi" cioè che hanno solo il compito di salvare i dati e tenere traccia della posizione all'interno dell'immagine. Così nell'algoritmo abbiamo evidenziato l'utilizzo di tre variabili contatore. Realizzati con dei contatori non solo con segnale clock e reset ma anche un segnale di enable del clock perché potessero essere attivati solo in determinati momenti.

La posizione di un pixel è determinata attraverso tre contatori uno per la colonna ("col" è counter col), uno per la riga ("riga" è counter riga) e uno per l'indirizzo di memoria di quel pixel (counter pos). Il contatore dell'indirizzo di memoria possiede un secondo reset "rst2" che porta pos al valore 0 poiché rst che viene collegato a tutti i reset del datapath lo porterebbe a due mentre su col e riga a 0. Gli enable sono direttamente collegati alla FSM poiché ogni compito "attivo" è delegato ad essa.

Per lo storage delle informazioni abbiamo posto una RAM a due ingressi e due output, la quale è scrivibile solo sul primo degli ingressi. La RAM possiede anche due segnali di controllo in base ai quali se '1' si abilita l'ingresso, altrimenti l'uscita corrispondente riporta il valore di default zero.

Lo stack della RAM è composto da 10 celle, le prime due con valori di default con i valori neutri delle operazioni cioè 0 e 1, utili per fare somme unitarie e riportare il valore nullo in casi particolari dall'algoritmo. Nelle altre celle sono salvati tutte le variabili dell'algoritmo, come indice del primo lato, secondo lato, riga superiore e riga inferiore dell'immagine.

I valori in uscita dalla RAM vengono salvati in registri di memoria a 18 bit, così come tutta l'architettura presenta solo registri da 18 bit. La scelta dei 18 bit è stata una scelta di comodità implementativa, dovuto anche alla possibilità di riutilizzo del codice. I registri sono di tipo parallelo-parallelo in quanto il salvataggio del valore non chiede che un solo ciclo di clock.

I registri sono stati posti per tre motivi:

1. Cercare di ridurre virtualmente i percorsi critici nel datapath, mettendo così delle aree di salvataggio che ci avrebbero permesso di spezzare i tempi di commutazione del circuito che va fino alla ALU, su più cicli di clock.
2. Se si possono spezzare le commutazioni su più cicli, allora abbiamo pensato che si potessero diminuire i periodi di clock e quindi aumentare la frequenza.
3. I registri erano la base anche per una eventuale ottimizzazione della macchina a stati, poiché modificando lievemente l'architettura e i segnali di controllo si potrebbe rendere il tutto simile ad una architettura pipeline.

Attaccata ai registri si trova la ALU. Essa è il collo di bottiglia dell'intero progetto. La sua struttura multilivello è il frutto di una discussione mirata al risparmio dell'area, dato che la frequenza di base per il progetto era 15 ns e il nostro progetto non ha problemi fino a circa 9,5 ns, abbiamo puntato ad un risparmio d'area. C'è anche da dire che il progetto non si prefigge l'obiettivo di essere ottimo in risparmio di tempo e nemmeno in area, ma volevamo costruire l'architettura più scalabile in assoluto che il tempo a disposizione e l'esperienza ci permettesse, con un occhio a piccole modifiche in futuro.

All'uscita della ALU troviamo un settore di registrazione dei risultati, dove sono posti rispettivamente un registro per i risultati con un significato numerico e un flip-flop per i risultati di tipo logico/booleano derivati da operazioni di confronto.

Il primo ha un solo collegamento in retroazione sulla RAM mentre il booleano finisce direttamente a dirigere la FSM.

Lo scorrimento dei dati tra un componente non è lasciato al caso, nel datapath sono coinvolti demultiplexer e multiplexer controllati sempre dalla FSM per cambiare i percorsi dei dati in base che debbano uscire dalla entity, essere manipolati attraverso operazioni o salvati all'interno della memoria RAM.

I Demultiplexer sono indicati solitamente con una lettera D e un numero per la loro enumerazione. Hanno un ingresso e sono controllati da un segnale logico che indica quale a quale delle uscite portare l'ingresso. I multiplexer sono chiamati super e numerati a loro volta, hanno 4 ingressi chiamati per questo "super" e sono controllati da un vettore di segnali per la selezione di quello uscente. Questi ultimi due trasportano dati in vettori da 8 bit, mentre c'è un ultimo multiplexer che possiede ingressi a 8 bit chiamato mux8BIT ed è il componente che determina se alla scrittura del risultato vengano scritti i bit più significativi o meno significativi.

Vantaggi e Svantaggi

La nostra idea per il progetto è sempre stata quella di creare qualcosa nella maniera più professionale che potessimo, con metodi più corretti e rigorosi che conoscessimo, non a caso abbiamo approfondito con parecchie letture il discorso di questo progetto, e che ci permettesse di imparare qualche cosa da riutilizzabile nel mondo del lavoro. Il nostro progetto quindi si ripropone come obiettivo assoluto la *scalabilità*. Per Scalabilità noi intendiamo dire che, dato che la nostra architettura si basa su un datapath e una FSM tutto ciò risulta di una grande scalabilità. Infatti volendo modificare l'algoritmo basterebbe modificare soltanto la FSM che da i segnali di controllo al datapath e qualora si volesse cominciare un qualsiasi progetto nuovo il datapath potrebbe essere un ottimo punto di partenza per collegarci una nuova FSM che implementi l'algoritmo del progetto proposto. Il nostro progetto potrebbe non essere il più veloce data la scelta di un algoritmo che nel caso migliore ha performance del caso peggiore ma ciò non ci interessava. Scelto un algoritmo pessimo abbiamo optato a un qualcosa che tenuta conto della scalabilità dell'architettura, potesse essere l'idea più contenuta in area che potessimo. Abbiamo rifiutato l'uso dei process finché ci è stato possibile poiché sapevamo che essi tendono a dare risultati molto ampi in area. Abbiamo cercato di scrivere un fsm che ci potesse dare libertà di ottimizzazione settando i vari segnali di comando noi stessi, uno per uno. Nel complesso siamo soddisfatti di quello che ci è stato possibile fare, soprattutto perché abbiamo anche delle performance temporali nei sottomoduli testati separatamente che ci rendono fieri, un po' meno sul sottomodulo della ALU che abbiamo ideato per le nostre computazioni e che abbiamo costruito su 8 livelli tentando di risparmiare area.

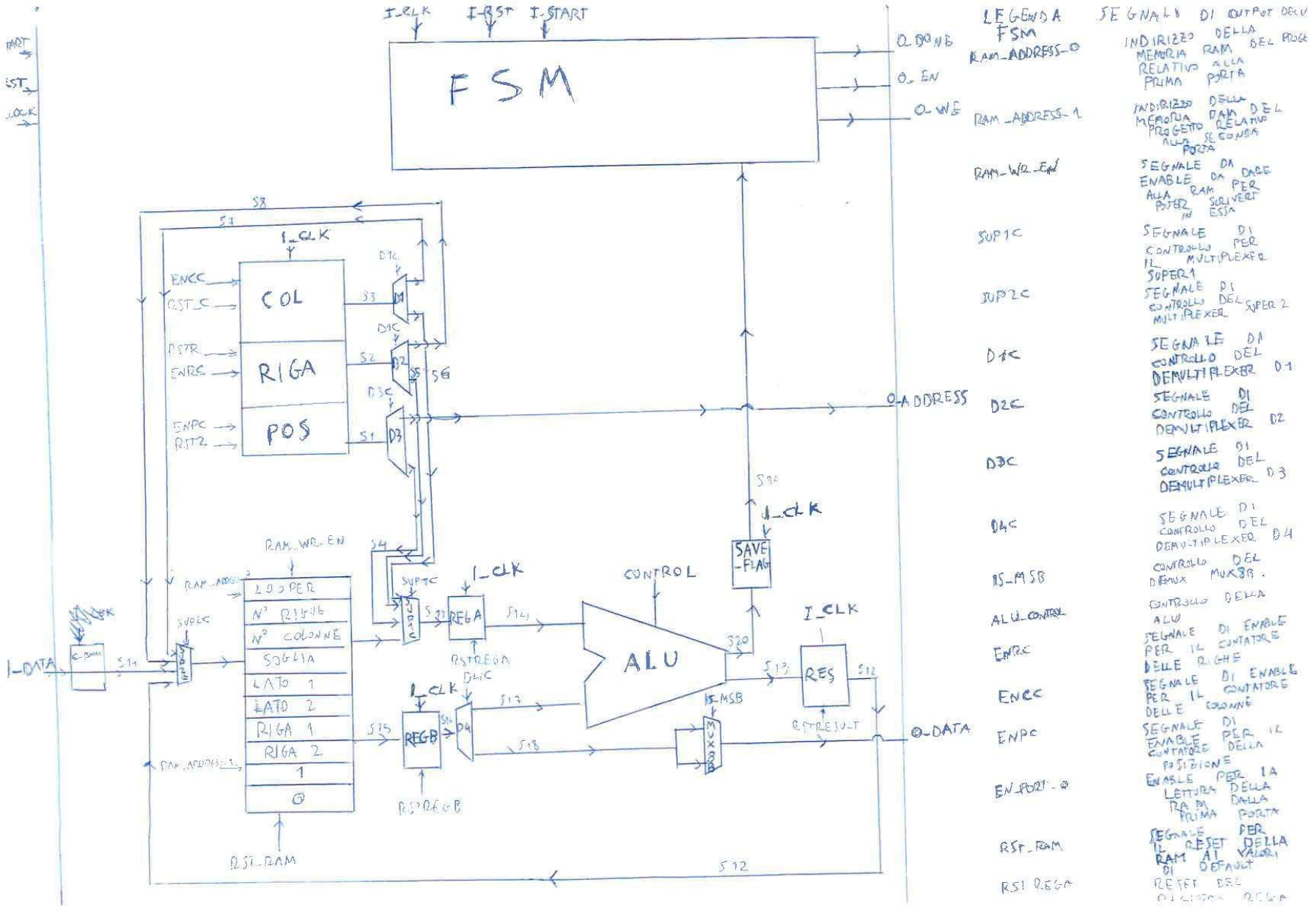
Sommario sulla progettazione

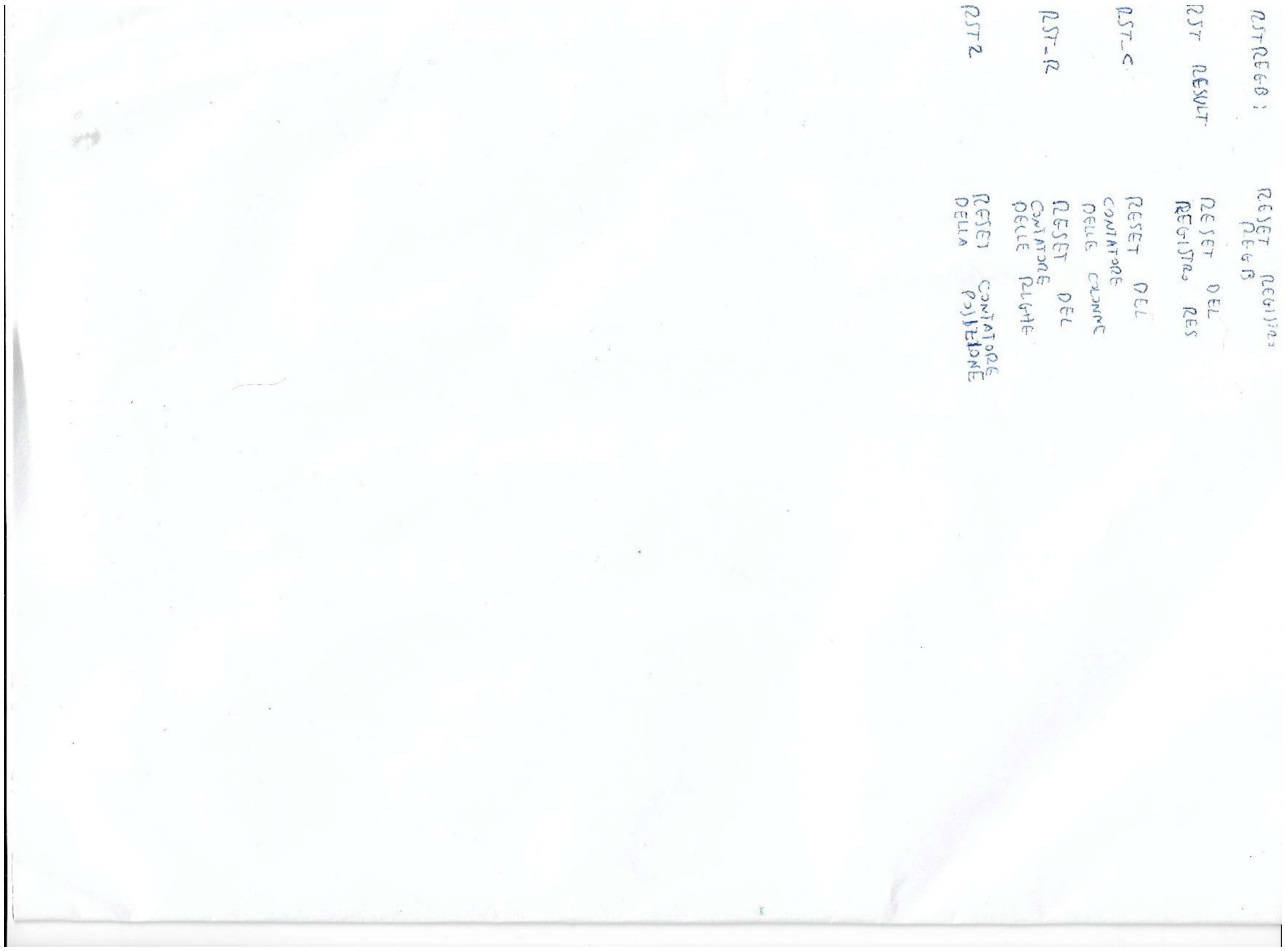
La progettazione concettuale è stata svolta seguendo una metodologia top down in cui abbiamo riscontrato in quante porzioni dividere l'architettura ricorsivamente così via fino ad arrivare ai singoli componenti. Per ogni versione del progetto abbiamo disegnato per immaginare meglio il risultato ed una volta sicuri di cosa necessitassimo, abbiamo creato ogni singolo componente separatamente, lo abbiamo testato e poi uniti in sotto-moduli in base all'utilità che avessero, per poi testare anch'essi separatamente, la fase finale ha richiesto di unirli fra di loro e di unirli con una FSM

inizialmente poco matura. Una volta allineati i segnali di controllo della FSM con il suo datapath e abbiamo verificato la correttezza del progetto.

Le difficoltà riscontrate nel progetto sono state soprattutto nell'uso dei tool e nella ricerca della forma migliore per definire in un linguaggio come VHDL un progetto nuovo per noi.

Rappresentazione


I_CLK
I_RST
I_START
FSM
O_DONE
O_EN
O_WE
LEGENDA FSM
RAM_ADDRESS_0
RAM_ADDRESS_1
RAM_WR_EN
SUP1C
SUP2C
D1C
D2C
D3C
D4C
IS_MSB
ALU_CONTROL
ENRC
ENCC
ENPC
EN_PORT_0
RST_RAM
RST REGA
SEGNALI DI OUTPUT DELL
INDIRIZZO DELLA MEMORIA RAM DEL PROG RELATIVO ALLA PRIMA PORTA
INDIRIZZO DELLA MEMORIA RAM DEL PROGETTO RELATIVO ALLA SECONDA PORTA
SEGNALE DA ENABLE DA DARE ALLA RAM PER SCRIVERE IN ESSA
SEGNALE DI CONTROLLO PER IL MULTIPLEXER SUPER1
SEGNALE DI CONTROLLO DEL MULTIPLEXER SUPER 2
SEGNALE DA CONTROLLO DEL DEMULTIPLEXER D1
SEGNALE DI CONTROLLO DEL DEMULTIPLEXER D2
SEGNALE DI CONTROLLO DEL DEMULTIPLEXER D3
SEGNALE DI CONTROLLO DEL DEMULTIPLEXER D4
CONTROLLO DEL DEMUX MUX8B.
CONTROLLO DELLA ALU
SEGNALE DI ENABLE PER IL CONTATORE DELLE RIGHE
SEGNALE DI ENABLE PER IL CONTATORE DELLE COLONNE
SEGNALE DI ENABLE CONTATORE PER IL DELLA POSIZIONE
ENABLE PER LA LETTURA DELLA RAM DALLA PRIMA PORTA
SEGNALE PER IL RESET DELLA RAM AI VALORI DI DEFAULT
RESET DEL
ENC
RST_C
INSTR
ENRS
ENPC
RST2
COL
RIGA
POS
D1C
D2C
D3C
M
D2
D3
S1
S2
S3
S5
S6
O_ADDRESS
I_CLK
SAVE -FLAG
S31
S32
RAM_WR_EN
S4
SUP1C
SUP2C
MUX8B
REGA
RSTREGA
D4C
S33
S14
CONTROL
ALU
S20
S19
RES
S12
RSTRESULT
O_DATA
I_DATA
S11
RAM_ADDRESS_1
LOOPER
N° RIGHE
N° COLONNE
SOGLIA
LATO 1
LATO 2
RIGA 1
RIGA 2
1
0
S15
REGB
S16
D4
S17
S18
IS_MSB
RSTREGB
RST_RAM
S12



TEST

Riportiamo i test eseguiti sul componente VHDL per la relativa validazione:

NB: i test qui presentati sono in post sintesi.

1.

Questo è il quarto test datoci come test di esempio:

Valori dati alla ram del testbench:

```
signal RAM: ram_type := (2 => "00011000", 3 =>"00000111",4 => "00010101", 30 => "00000011",
31 => "00000011", 32 => "00000011", 33 => "00000011", 36 => "00000111", 37 => "00000111",
38 => "00000111", 39 => "00000111", 42 => "00001011", 43 => "00001011", 44 => "00001011", 45 => "00001011",
48 => "00001111", 54 => "00000011", 60 => "00000111", 66 => "00011111", 72 => "00011001", 78 => "00000011",
79 => "00000011", 80 => "00000011", 84 => "00000111", 85 => "00000111", 86 => "00000111", 90 => "00011111",
91 => "00011111", 92 => "00011111", 96 => "00011001", 102 => "00000011", 108 => "00000111", 114 => "00011111",
120 => "00011001", 126 => "00000011", 132 => "00000111", 133 => "00000111", 134 => "00000111", 135 => "00000111",
138 => "00001011", 139 => "00001011", 140 => "00001011", 141 => "00001011", 144 => "00001111", 145 => "00001111",
146 => "00001111", 147 => "00001111", others => (others =>'0'));
```

Risultato ottenuto:

```
assert RAM(1) = "00000000" report "FAIL high bits" severity failure;
assert RAM(0) = "00010101" report "FAIL low bits" severity failure;


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

2.

Questo è il terzo test datoci come test di esempio:

Valori dati alla ram del testbench:

```
signal RAM: ram_type := (2 => "00011000", 3 =>"00000111",4 => "00000010",
30 => "00000011", 31 => "00000011", 32 => "00000011", 33 => "00000011",
36 => "00000111", 37 => "00000111", 38 => "00000111", 39 => "00000111",
42 => "00001011", 43 => "00001011", 44 => "00001011", 45 => "00001011",
48 => "00001111", 54 => "00000011", 60 => "00000111", 66 => "00011111",
72 => "00011001", 78 => "00000011", 79 => "00000011", 80 => "00000011",
84 => "00000111", 85 => "00000111", 86 => "00000111", 90 => "00011111",
91 => "00011111", 92 => "00011111", 96 => "00011001", 102 => "00000011",
108 => "00000111", 114 => "00011111", 120 => "00011001", 126 => "00000011",
132 => "00000111", 133 => "00000111", 134 => "00000111", 135 => "00000111",
138 => "00001011", 139 => "00001011", 140 => "00001011", 141 => "00001011",
144 => "00001111", 145 => "00001111", 146 => "00001111", 147 => "00001111",
others => (others =>'0'));
```

Risultato ottenuto:

```
assert RAM(1) = "00000000" report "FAIL high bits" severity failure;
assert RAM(0) = "01101110" report "FAIL low bits" severity failure;


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

3.

Questo è il secondo test datoci come test di esempio:

Valori dati alla ram del testbench:

```
signal RAM: ram_type := (2 => "00011000", 3 =>"00000111",4 => "00000000",
30 => "00000011", 31 => "00000011", 32 => "00000011", 33 => "00000011",
36 => "00000111",
42 => "00001011",
48 => "00001111", 54 => "00000011", 60 => "00000111", 66 => "00011111",
72 => "00011001", 78 => "00000011", 79 => "00000011", 80 => "00000011",
84 => "00000111", 85 => "00000111", 86 => "00000111", 90 => "00011111",
91 => "00011111", 92 => "00011111", 96 => "00011001", 102 => "00000011",
108 => "00000111", 114 => "00011111", 120 => "00011001", 126 => "00000011",
132 => "00000111", 133 => "00000111", 134 => "00000111", 135 => "00000111",
138 => "00001011", 139 => "00001011", 140 => "00001011", 141 => "00001011",
144 => "00001111", 145 => "00001111", 146 => "00001111", 147 => "00001111",
others => (others =>'0'));
```

array (65535 downto 0) of std_logic_vector(7 downto 0)

Risultato ottenuto:

```
assert RAM(1) = "00000000" report "FAIL high bits" severity failure;
assert RAM(0) = "10101000" report "FAIL low bits" severity failure;


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

4.

Questo è il test primo test datoci come test di esempio:

Valori dati alla ram del testbench:

```
signal RAM: ram_type := (2 => "00011000", 3 =>"00000111",4 => "11111111",
30 => "00000011", 31 => "00000011", 32 => "00000011", 33 => "00000011",
36 => "00000111", 37 => "00000111", 38 => "00000111", 39 => "00000111",
42 => "00001011", 43 => "00001011", 44 => "00001011", 45 => "00001011",
48 => "00001111", 54 => "00000011", 60 => "00000111", 66 => "00011111",
72 => "00011001", 78 => "00000011", 79 => "00000011", 80 => "00000011",
84 => "00000111", 85 => "00000111", 86 => "00000111", 90 => "00011111",
91 => "00011111", 92 => "00011111", 96 => "00011001", 102 => "00000011",
108 => "00000111", 114 => "00011111", 120 => "00011001", 126 => "00000011",
132 => "00000111", 133 => "00000111", 134 => "00000111", 135 => "00000111",
138 => "00001011", 139 => "00001011", 140 => "00001011", 141 => "00001011",
144 => "00001111", 145 => "00001111", 146 => "00001111", 147 => "00001111",
others => (others =>'0'));
```

Risultato ottenuto:

```
assert RAM(1) = "00000000" report "FAIL high bits" severity failure;
assert RAM(0) = "00000000" report "FAIL low bits" severity failure;


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

5.

Il seguente test è relativo al caso in tutti i valori dell'immagine siano sopra la soglia. La soglia in questo caso è posta a zero e il numero di righe ed il numero di colonne sono poste rispettivamente a 4 e a 3. Nella ram ogni cella di memoria viene assegnato al valore convenzionale superiore al valore di soglia ,ovvero 214 (11010110 in binario). Il risultato ottenuto è infatti pari a 12 ovvero al prodotto del numero di righe e del numero delle colonne.

Valori dati alla ram del testbench:

```
signal RAM: ram_type := (2 => "00000011", 3 =>"00000100",4 => "00000000",
others => ("11010110"));
```

Risultato ottenuto:

```
assert RAM(1) = "00000000" report "FAIL high bits" severity failure;
assert RAM(0) = "00001100" report "FAIL low bits" severity failure;


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

6.

Il seguente test è relativo al caso in cui il numero delle colonne della porzione d'immagine da considerarsi sia uguale a zero.

Valori dati alla ram del testbench:

```
signal RAM: ram_type := (2 => "00000000", 3 =>"01001100",4 => "00001100",
others => ("11010110"));
```

Risultato ottenuto:

```
assert RAM(1) = "00000000" report "FAIL high bits" severity failure;
assert RAM(0) = "00000000" report "FAIL low bits" severity failure;


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

7.

Questo test è relativo al caso in cui il numero di righe sia zero. Ovviamente ci si aspetta di conseguenza, come riscontrato positivamente poi nel risultato del test, che l'area minima della figura d'interesse sia zero anch'essa.

Valori dati alla ram del testbench:

```
signal RAM: ram_type := (2 => "00001110", 3 =>"00000000",4 => "00001100",
others => ("11010110"));
```

Risultato ottenuto:

```
assert RAM(1) = "00000000" report "FAIL high bits" severity failure;
assert RAM(0) = "00000000" report "FAIL low bits" severity failure;


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

8.

Questo è il test in cui sia numero di colonne, righe, soglia e valori memorizzati nella ram siano pari a zero. Ci si aspetta che l'area della figura d'interesse sia zero anch'essa di conseguenza, come poi riscontrato positivamente nel risultato del testbench.

Valori dati alla ram del testbench:

```
signal RAM: ram_type := (2 => "00000000", 3 =>"00000000",4 => "00000000",
others => (others =>'0'));
```

Risultato ottenuto:

```
assert RAM(1) = "00000000" report "FAIL high bits" severity failure;
assert RAM(0) = "00000000" report "FAIL low bits" severity failure;


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

9.

Questo test rappresenta il caso in cui tutti i valori siano sopra soglia. Il numero di righe è colonne vale 31 in entrambi i casi, la soglia vale 4. Ci si aspetta come risultato quindi il prodotto del numero di righe e del numero di colonne ovvero 961 (cioè 31 x 31). La differenza, non da poco, tra questo caso, e l'analogo test in cui si avevano sempre tutti i pixel sopra soglia, è che in questo caso si va a testare anche gli otto bit più significativi del risultato ram (1) e non solo gli 8 bit meno significativi ram(0) come si faceva negli altri test compresi quelli datoci come test di esempio.

Valori dati alla ram del testbench:

```
signal RAM: ram_type := (2 => "00011111", 3 =>"00011111",4 => "00000100",
others=>("00000111"));
```

Risultato ottenuto:

```
assert RAM(1) = "00000011" report "FAIL high bits" severity failure;
assert RAM(0) = "11000001" report "FAIL low bits" severity failure;


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

10.

Il seguente test è relativo al caso in cui ci sia un solo pixel sopra soglia e in cui il numero di righe e colonne sia quello massimo ovvero 11111111 in binario cioè 255 in decimale. Ci si aspetta, come poi riscontrato positivamente nel test che il valore dell'area della figura d'interesse sia 1 in ram(0).

Valori dati alla ram del testbench:

```
signal RAM: ram_type := (2 => "11111111", 3 =>"11111111",4 => "00000100",
7=>conv std logic vector(18,8), others=>(others=>'0'));
```

Risultato ottenuto:

```
assert RAM(1) = "00000000" report "FAIL high bits" severity failure;
assert RAM(0) = "00000001" report "FAIL low bits" severity failure;


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

11.

Il seguente test è relativo ad un input di ram generale in cui però all'interno della figura (numero righe = 4 e numero colonne = 5 e soglia uguale a 4) i pixel sopra soglia formano un trapezio rettangolo posizionato in maniera tale che l'altezza (di valore uguale a 3) sia orizzontale e che la base maggiore (di valore uguale a 4) si trovi alla sinistra di quella minore. L'area minima del rettangolo che circoscrive questa figura in questo caso è pari a 12, pari al prodotto della base maggiore per l'altezza del trapezio, come riscontrato positivamente dal risultato del test.

Valori dati alla ram del testbench:

```
signal RAM: ram_type := (2 => "00000101", 3 =>"00000100",4 => "00000100",
7=>conv_std_logic_vector(5,8), 8=>conv_std_logic_vector(9,8),
9=>conv_std_logic_vector(5,8), 12=>conv_std_logic_vector(12,8),
13=>conv_std_logic_vector(17,8),14=>conv_std_logic_vector(5,8),
17=>conv_std_logic_vector(8,8),22=>conv_std_logic_vector(5,8),
18=>conv std logic vector(8,8), others=>(others=>'0'));
```

Risultato ottenuto

```
assert RAM(1) = "00000000" report "FAIL high bits" severity failure;
assert RAM(0) = "00001100" report "FAIL low bits" severity failure;


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

12.

Il seguente test è relativo al caso in cui tutti i valori della figura di interesse siano uguali alla soglia. Il numero delle righe e delle colonne e della soglia è posto uguale a 15. Il valore di tutti gli elementi all'interno di questa figura sono uguali alla soglia

Valori dati alla ram del testbench:

```
signal RAM: ram_type := (2 => "00001111", 3 =>"00001111",4 => "00001111",
others => ("00001111"));
```

Risultato ottenuto:

```
assert RAM(1) = "00000000" report "FAIL high bits" severity failure;
assert RAM(0) = "11100001" report "FAIL low bits" severity failure;


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

13.

In questo test si è nel caso in cui il numero di colonne sia uguale a 5 e il numero di righe sia uguale a 4 e la soglia sia posta uguale a 3. I bit sopra soglia all'interno della figura sono posti come un triangolo rettangolo con cateto maggiore pari a 5 e cateto minore pari a 4. Il rettangolo minimo che circoscrive l'area ha area uguale a 20 come poi confermato dal test.

Valori dati alla ram del testbench:

```
signal RAM: ram_type := (2 => "00000101", 3 =>"00000100",4 => "00000011",
5 => conv_std_logic_vector(4,8), 9 => conv_std_logic_vector(5,8),
24 => conv_std_logic_vector(6,8),others => (others =>'0'));
```

Risultato ottenuto:

```
assert RAM(1) = "00000000" report "FAIL high bits" severity failure;
assert RAM(0) = "00010100" report "FAIL low bits" severity failure;


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

14.

Nel seguente test si è nel caso in cui gli unici elementi sopra soglia siano i vertici della figura di interesse. In particolare, il primo vertice è in alto a sinistra ed il secondo in basso a destra. Il numero di righe ed il numero di colonne sono posti rispettivamente uguali a 4 e 5. La soglia è posta uguale a 3. Il valore dell'area rettangolo minimo che ci si aspetta è di 20 come poi confermato nel risultato del test.

Valori dati alla ram del testbench:

```
signal RAM: ram_type := (2 => "00000101", 3 =>"00000100",4 => "00000011",
5 => conv_std_logic_vector(4,8),  24 => conv_std_logic_vector(6,8),
others => (others =>'0'));
```

Risultato ottenuto:

```
assert RAM(1) = "00000000" report "FAIL high bits" severity failure;
assert RAM(0) = "00010100" report "FAIL low bits" severity failure;


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

Frequenza Limite

```
create_clock -period 9.6 -name i_clk [get_ports i_clk]
```

Design Timing Summary

Setup		Hold		Pulse Width	
Worst Negative Slack (WNS):	0,039 ns	Worst Hold Slack (WHS):	0,264 ns	Worst Pulse Width Slack (WPWS):	4,300 ns
Total Negative Slack (TNS):	0,000 ns	Total Hold Slack (THS):	0,000 ns	Total Pulse Width Negative Slack (TPWS):	0,000 ns
Number of Failing Endpoints:	0	Number of Failing Endpoints:	0	Number of Failing Endpoints:	0
Total Number of Endpoints:	869	Total Number of Endpoints:	869	Total Number of Endpoints:	316

All user specified timing constraints are met.