Politecnico di Milano

Report di prova finale : Progetto di Reti Logiche

Gruppo di lavoro

- Numero matricola: **845180** - Codice Persona: **10503336** - Nome e Cognome: **Matteo Belcao**
- Numero matricola: **847163** - Codice Persona: **10493758** - Nome e Cognome: **Giuseppe Bertolini**

Descrizione generale

Per l'analisi dell'immagine inizialmente si è pensato di utilizzare un algoritmo di visita delle celle a "spirale". Successivamente però, vista la complessità richiesta nel calcolo degli indici per analizzare le varie celle del vettore *RAM*, si è optato per l'implementazione di un algoritmo di efficienza equivalente e semplicità maggiore basato su quattro ispezioni lineari. Il progetto è stato strutturato basandosi su quattro moduli differenti, ognuno con una funzione logica ben distinta dagli altri:

- l'***FSM Controller*** si occupa di orchestrare la corretta sequenza di esecuzione delle varie fasi elaborazione, inviando un segnale di *FSM_EN_STATE* agli altri componenti;
- il ***LOADER*** carica in dei registri i numeri di colonne, righe e la soglia;
- l'***INSPECTOR*** esplora l'immagine alla ricerca dei confini del soggetto;
- Il ***MULTIPLIER*** calcola l'area e scrive il risultato in *RAM*.

Una volta terminata l'elaborazione ciascun modulo risponde con un segnale *end* di feedback all'*FSM*. La progettazione per componenti ha permesso di creare dei testbench specifici per ciascun modulo, rendendo più semplice il debug dei vari step di analisi dell'immagine digitale [si veda la sezione sul testing]. I moduli sono poi stati integrati in un'architecture come istanze di components, nonostante questo renda il codice notevolmente più lungo, permette un'analisi più lucida del progetto, anche grazie ad un diagramma a blocchi più pulito. Infine abbiamo effettuato una revisione del codice volta a semplificare le sequenze critiche dell'algoritmo, in particolare è stato ridotto al minimo il numero massimo di if e case annidati, e sono stati semplificati il più possibile i calcoli necessari per individuare gli indirizzi di memoria necessari.

Risultati ottenuti

L'analizzatore è riuscito a superare i testbench forniti in modalità ***Behavioural***, ***Post-Synthesis*** *Functional* e ***Post-Implementation*** *Functional*. Inoltre è stato possibile eseguire i test in *Post-Synthesis e Post-Implementation* aumentando la frequenza di clock fino a circa ***500Mhz*** (t_*clock*=2.01ns). Con frequenze superiori si sono riscontrati problemi di ritardo eccessivo nel recuperare i dati dalla *RAM* che portano ad una lettura di dati incorretti. La complessità di esecuzione dell'algoritmo ottenuta è di tipo ***lineare*** (con il numero totale di pixel dell'immagine), con l'aggiunta di un costo costante di sette cicli di clock per operazioni che non siano la lettura del valore di un pixel.

$$\Theta(celle\_visitate * CLK + 7CLK) \simeq \Theta(celle\_visitate * CLK)$$

Algoritmo di analisi

L'algoritmo si basa su quattro passaggi, ognuno dei quali, nel caso generale, termina quando si trova una cella di memoria che soddisfi il valore di soglia. Sono così individuate le due righe e le due colonne circoscriventi il "soggetto", utilizzate poi per calcolare l'area del rettangolo.
[Si assuma per righe e colonne degli indici crescenti verso il basso e verso destra]
La logica dei quattro step di analisi, più in dettaglio, è:

- ***LEFT*** : la prima fase della procedura prevede di ispezionare l'immagine a partire dal primo pixel in alto a sinistra e di proseguire scendendo di riga in riga, sulla stessa colonna, fino a raggiungere l'estremità inferiore dell'immagine, per poi ripartire dalla cima della colonna successiva (adiacente a destra). L'esecuzione termina quando si trova un punto con caratteristiche idonee memorizzandone le coordinate, o, nel caso in cui si arrivi alla fine dell'immagine senza che questo si verifichi, scrivendo il risultato (0x0000) e terminando l'algoritmo.

- ***RIGHT*** : in questa fase si procede analogamente alla prima, questa volta però si parte dalla cella in alto a destra e si prosegue con un'ispezione colonna per colonna verso sinistra. Come nel caso precedente, *RIGHT* termina nel momento in cui si individua un pixel rilevante di cui sono annotate le coordinate oppure, qualora sia stata raggiunta senza successo l'ultima cella della colonna adiacente a quella ottenuta al passaggio precedente, le coordinate scelte sono uguali a quelle di *LEFT*. A differenza di *LEFT* non è previsto il caso in cui non ci sia alcun pixel che superi o eguagli la soglia, poiché sappiamo dallo stesso che almeno uno di questi esiste.

- ***UP*** : l'ispezione procede per righe, dalla prima cella della colonna successiva a quella del pixel trovato da *LEFT*, verso la colonna precedente quella memorizzata in *RIGHT*. Nel caso in cui le due colonne siano adiacenti o coincidenti si passa direttamente al prossimo step, in caso alternativo si continua fino al rinvenimento di un pixel adeguato (di cui si memorizza l'indice di riga in luogo di quello minore fino ad allora individuato), oppure fino alla fine dell'ultima riga prima di quella più in alto tra quelle scovate da *LEFT* e *RIGHT* (l'indice di riga minore non è aggiornato).

- ***DOWN*** : muovendosi dall'ultima riga a salire, si esaminano le celle dalla colonna con l'indice minore tra le due di interesse (*LEFT*) a quella col maggiore (*RIGHT*), finché non si trova un pixel utile (che porta all'aggiornamento dell'indice di riga inferiore) oppure si raggiunge l'ultima cella della riga immediatamente sottostante la riga con indice massimo, tra quelle scovate precedentemente.

L'algoritmo descritto è chiaramente di complessità lineare con la dimensione dell'immagine, $O(n\_righe * n\_colonne)$, ma, dovendo, per poter dare una risposta esatta, controllare almeno ogni cella che sia al di fuori dell'area di interesse, sarebbe stato impossibile fare altrimenti. Ciononostante sono state messe in atto delle ottimizzazioni sia per quanto riguarda il ridurre al minimo il numero di celle visitate in media, sia per l'aver pensato queste ispezioni in modo da rendere il più semplice possibile il calcolo dell'indice di memoria di volta in volta (utilizzando esclusivamente somme).
Nelle figure successive sono mostrati tre casi di esecuzione dell'algoritmo che mostrano graficamente il comportamento appena descritto.

LEGENDA:

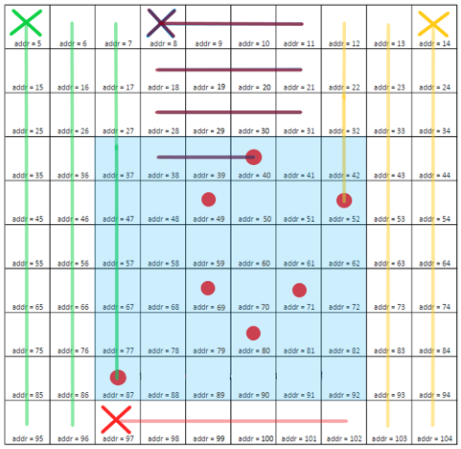


Fig.1

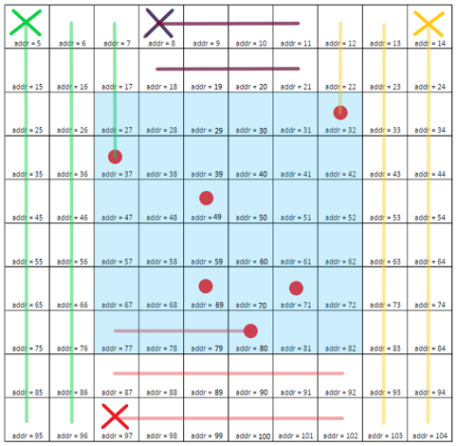


Fig.2

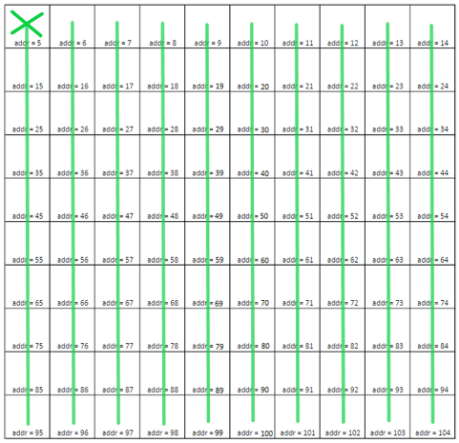
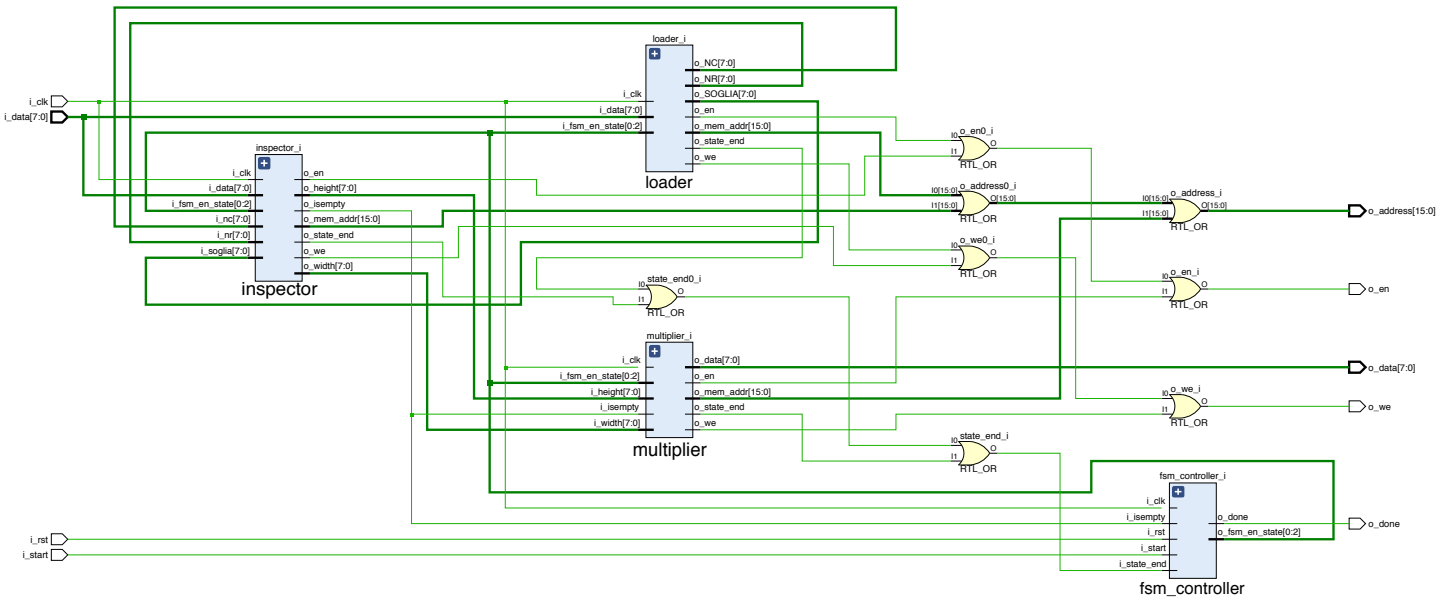


Fig.3

Schema a blocchi



Schema a blocchi risultato della RTL Analisys in Vivado

Qui sopra è riportato lo schema a blocchi del nostro progetto, generato automaticamente da *Vivado* (sono stati ingranditi i nomi dei *component* al fine di migliorarne la leggibilità), che mette in mostra come i moduli interagiscono tra loro e con gli *input/output* dell'intera *architecture.* I dettagli dell'architettura e dei moduli sono approfonditi in seguito.

Descrizione dei blocchi

- **LOADER**

Svolge la funzione di caricamento iniziale dei dati riguardanti le dimensioni dell'immagine e il valore di soglia nei registri *r_nc* , *r_nr* e *r_soglia.* Per fare ciò ci si avvale di un counter da 0x02 a 0x04 che impone il suo valore all'uscita *o_address* ad ogni fronte di discesa del clock, mentre allo stesso tempo legge il dato proveniente da *i_data* e lo memorizza nel registro opportuno. I dati in lettura si riferiscono all'indirizzo imposto prima della commutazione, di conseguenza sono necessari 4 cicli per lettura e memorizzazione.

- **INSPECTOR**

Si divide in quattro "sottoprocessi" tramite un *case* sull' *fsm_en_state* (concatenato con *isp_work*) che permette di distinguere l'ispezione nei quattro passi (e rispettive fasi di preset) già visti nella sezione "Algoritmo di analisi". L'inspector lavora sul fronte di discesa del clock, in opposizione al cambio di stato dell'*FSM* e sul fronte opposto a quello in cui agisce la RAM, da cui legge secondo l'algoritmo descritto i vari byte rappresentanti i pixel via *i_data.* I vari sottoprocessi si servono di una variable *isp_work*, un bit che permette loro di distinguere tra una prima fase (un ciclo di clock) (*isp_work = 0*) di preset di variabili e registri necessari e una seconda (*isp_work = 1*) in cui eseguono effettivamente l'ispezione.

I. ***LEFT*** (*fsm_en_state = "010"*) parte presettando (*isp_work = 0*) l'indirizzo della prima lettura da memoria per la prima cella della RAM valida per l'immagine, ovvero RAM (5), e *r_r1* e *r_c1* a 1 entrambi (indicano così la prima riga e la prima colonna). Procede poi (*isp_work = 1*) incrementando *r_r1* e l'indirizzo di lettura del numero di colonne ($isp\_mem\_addr = isp\_mem\_addr + r\_nc$), poiché la matrice rappresentante l'immagine è salvata come vettore ordinato composto dalle sue righe concatenate. Alla fine della colonna (*r_r1 == r_nr*) *r_r1* è reimpostato a 1, mentre *r_c1* viene incrementato dell'unità e utilizzato per settare il corretto indirizzo $isp\_mem\_address = r\_c1 + 5$. Se non si trova niente fino alla fine, viene messo a 1 *isempty* che bypassa gli stati successivi e porta alla scrittura di 0x0000 in RAM nel *MULTIPLIER,* altrimenti si passa semplicemente allo stato successivo mediante il segnale *isp_end* che fa scattare l'*FSM*. A questo punto *r_c1* rappresenta già l'estremo sinistro del soggetto e non sarà più modificato, mentre r_r1, poiché l'ispezione procedeva dall'alto verso il basso, figura come un papabile estremo superiore dello stesso, ma potrà subire variazioni.

II. ***RIGHT*** (*fsm_en_state = "011"*) lavora invece, almeno inizialmente, su *r_r2* e *r_c2* e, nella fase di preset (*isp_work = 0*), li inizializza rispettivamente a *1* e *r_nc*, e impone $isp\_mem\_addr = r\_nc + 4$. L'ispezione procede (*isp_work = 1*) in maniera analoga a quella di *LEFT*, con l'unica differenza che si visitano le colonne da destra a sinistra. Il procedimento termina nel caso si trovi un valore uguale o superiore la soglia oppure nel caso in cui la colonna sia terminata e $r\_c2 = r\_c1 + 1$ (la colonna successiva è già stata controllata da LEFT). Al termine viene eseguito un confronto tra *r_r2* e *r_r1* che vengono poi eventualmente scambiati con lo scopo di avere in *r_r1* il valore minore e in *r_r2* il maggiore. In *r_c2* sta invece il valore definitivo della colonna estrema di destra.

III. ***UP*** (*fsm_en_state = "100"*) visiona le celle soprastanti l'area d'interesse partendo dal pixel nella prima riga e nella colonna successiva quella salvata in *r_c1 (* $isp\_mem\_addr = r\_c1 + 5$ *)* (*isp_work = 0*). Di cella in cella visita le righe (*isp_work = 1*) fino alla colonna precedente quella indicata da *r_c2* semplicemente incrementando di 1 *isp_mem_addr* ad ogni passo, una volta arrivata a tale colonna *isp_mem_addr* è impostato in modo da passare alla riga successiva con la seguente formula: $isp\_mem\_addr = isp\_mem\_addr + (r\_nc - r\_c2 + r\_c1 + 2)$. Il fatto di aver visitato dall'alto verso il basso le colonne *r_c1* e *r_c2* in *LEFT* e *RIGHT* ci permette di dire che in *r_r1,* nel caso in cui non si sia trovato un punto valido prima della riga *r_r1*, stia il limite superiore dell'area soggetto, e ci consente di non dover effettuare moltiplicazioni per inizializzare, ad esempio, *isp_mem_addr* all'ultima

cella dell'intera matrice.

IV. ***DOWN*** (*fsm_en_state = "101"*) parte dall' ultima cella della colonna memorizzata in *r_c1* (*isp_work = 0*) (indicata da *down_start_index*) e procede (*isp_work = 1*) con una scansione per righe verso l'alto incrementando *isp_mem_addr* di 1 ad ogni passo, finchè non si raggiunge la cella della colonna *r_c2*. Per salire di una riga l'indirizzo di lettura è aggiornato nella maniera seguente:

$$isp\_mem\_addr = isp\_mem\_addr - (r\_nc + r\_c2 - r\_c1)$$

Lo stadio può terminare se si raggiunge un pixel utile, oppure, con un ottimizzazione, se si completa l'ispezione della riga sottostante quella memorizzata in *r_r2* (dove è sicuramente presente un punto con valore pari o superiore alla soglia).

- **MULTIPLIER**

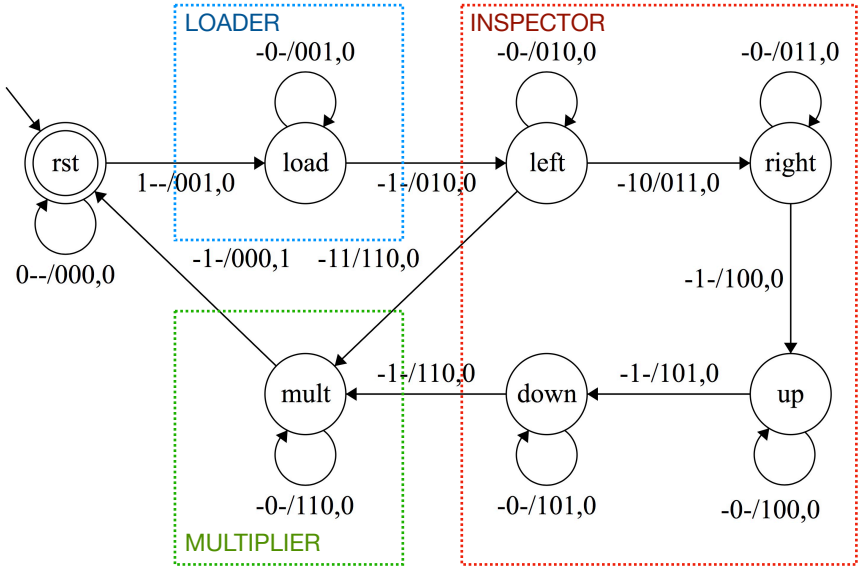
Viene utilizzato per calcolare l'area finale di interesse, ottenuta con un moltiplicatore a 16 bit e memorizzata nella variable *ris*. I due lati del rettangolo sono calcolati facendo le differenze tra gli indici dei pixel di interesse ottenuti dal componente INSPECTOR, quindi:

$$ris = i\_width * i\_height = (r\_c2 - r\_c1 + 1) * (r\_r2 - r\_r1 + 1)$$

Nel caso il flag *isempty* sia uguale a 1 (l'immagine non ha pixel con valore superiore o uguale alla soglia) allora *ris = 0x0000*.
Una volta ottenuto il risultato *ris* si procede ad inviarlo in due blocchi da 8 bit alla memoria attraverso il segnale *o_data* utilizzando due cicli di clock distinti.

Analisi funzionale (macchina a stati finiti)

L'ultimo modulo, l'*FSM_controller,* implementa invece una macchina a stati di Mealy che ha il compito di "governare" gli altri moduli e coordinarli al fine di ottenere il risultato desiderato. Si occupa in oltre di gestire i segnali di input (*i_start* e *i_reset*) e output (*o_done*) che sanciscono inizio e fine delle operazioni da compiere. Le transizioni avvengono sul fronte di salita di *i_clk*, in modo da ridurre al minimo lo spreco di cicli per il passaggio di stato (gli altri moduli lavorano sul fronte opposto).



Transizioni: i_start i_state_end i_isempty / o_fsm_en_state[3] , o_done

Partendo da uno stato di *RESET*, in cui vengono ignorati tutti i segnali di input finchè non sopraggiunge un segnale *i_start=1*, che avvia l'esecuzione vera e propria dell'algoritmo di analisi.
Il passaggio da uno stato all' altro è determinato dalla ricezione in input di *i_state_end=1* dagli altri moduli che porta all'aggiornamento di *o_fsm_en_state* (sul fronte di salita), se invece *i_state_end=0* si rimane nello stesso. La FSM è stata inoltre ottimizzata inserendo un terzo segnale di input *i_isempty*, che può assumere valore 1 solamente nello stato di *LEFT*, se ispezionando tutta la matrice non si trova alcun pixel utile, permettendo di passare alla scrittura del risultato 0x0000 direttamente nello step di *MULTIPLIER*, saltando gli altri passaggi di analisi dell'immagine. Una volta terminata l'esecuzione dell'algoritmo (il

MULTIPLIER ha calcolato e scritto il risultato in memoria e dato l' "end di stato"), la FSM provvede inoltre a trasmettere per un ciclo di clock il segnale *o_done=1* per segnalare la terminazione del processo di analisi, per poi tornare allo stato INIZIALE/RESET, in attesa di una nuova esecuzione.

Ottimizzazioni e accorgimenti adottati

Durante tutto lo sviluppo del progetto dal principio alla consegna, sono state effettuate scelte nell'ottica di ridurre al minimo lo spreco di risorse, il tempo su tutte. Prima ancora di provare a scrivere linee di codice VHDL è stato svolto un intenso lavoro di brainstorming per cercare una soluzione algoritmica ottimale, che ci ha visti simulare su carta procedure di vario genere applicate a casi di vario genere col fine di ottenerne una che permettesse di visitare il minor numero di celle possibile e con la maggior frequenza. Così siamo giunti all'algoritmo che abbiamo scelto di portare avanti sia per la sua efficienza sia per la semplicità, nonostante siano stati prese in considerazione molte sue varianti con lo scopo di escludere che ci fosse una soluzione che sembrasse migliore o che lo fosse davvero. Di seguito sono riportate alcune delle altre ottimizzazioni apportate al design.

- Nella fase di implementazione una prima semplice ma corposa ottimizzazione è stata quella di avere scelto, in seguito ad un'analisi circa il funzionamento della RAM attraverso il codice fornitoci, di effettuare calcoli e aggiornamenti di output quali l'enable per la lettura da memoria e l'address sul fronte di discesa del clock, in opposizione a quello utilizzato dalla RAM (e dall'FSM controller, che però non legge mai dalla RAM). Questo dettaglio ci ha consentito di ridurre sia i possibili problemi di conflitti, lettura di dati scorretti sia i tempi di esecuzione di un fattore due.
- L'algoritmo è stato scelto e implementato in modo da potere e dovere effettuare i calcoli più semplici possibile, si tratta quasi esclusivamente di semplici somme, che sono state scritte in modo espanso, evitando annidamenti che rendessero più difficile la vita a Vivado nella sua fase di ottimizzazione ed effettuando i padding (ad esempio per signal 8 -> 16 bit) solo una volta effettuate le somme (a 8bit). Uniche eccezioni sono l'indice di partenza di *DOWN* che è stato necessario calcolare tramite moltiplicazione, così come il risultato finale dell'esecuzione.
- Sono stati utilizzati *case* su più signal concatenati in luogo di *if* annidati, ed è stato ulteriormente abbattuto il numero massimo di livelli di annidamento grazie ad un'operazione di "appiattimento" che ha portato alla riduzione a uno (tolti *if(falling_edge(i_clk))* e il *case* sull'fsm_state_en) della profondità dei controlli. Le nuove espressioni degli if sono state scritte e commentate in maniera chiara e lineare ed è stato verificato logicamente come non vi fosse overlapping tra loro e che non ci fosse alcun caso non che non avesse un branch in cui entrare. Insomma è stato fatto in modo che l'insieme di tutte le possibili combinazioni valide di dati fossero un unione disgiunta di delle classi definite da tali condizioni.
- I quattro passi dell'algoritmo sono stati accorpati in un unico modulo (*INSPECTOR*) in modo da non rendere inutilmente complicato l'utilizzo di registri comuni ed evitare di dovere duplicarli.

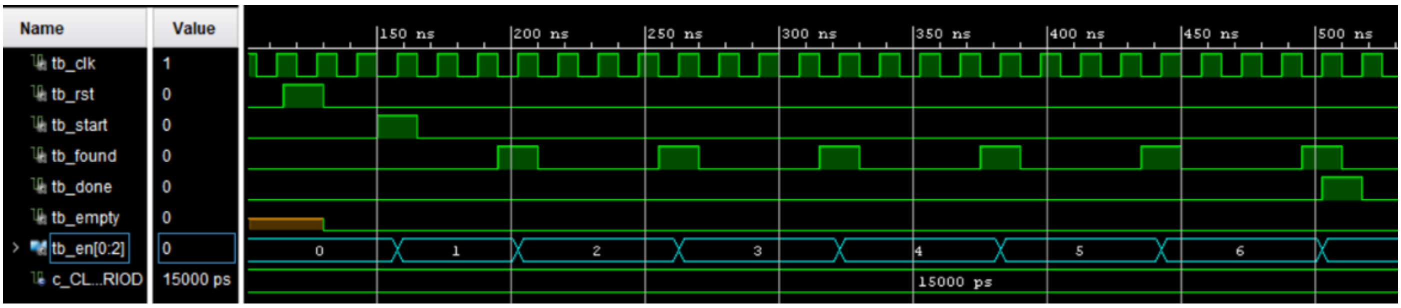
Testing

Come detto ogni modulo è stato testato separatamente con dei testbench creati appositamente, il cui codice è riportato a seguire con la relativa immagine della simulazione effettuata.
Particolare attenzione è stata prestata all' *INSPECTOR* in quanto è di gran lunga il componente più complesso, numerosi casi di test, con griglie di dimensioni diverse e pixel validi in posizioni più o meno atipiche, gli sono stati dedicati. Riportiamo in seguito i testbench di ogni componente e il risultato della sua simulazione.

FSM_CONTROLLER

Codice del testbench e risultato della simulazione

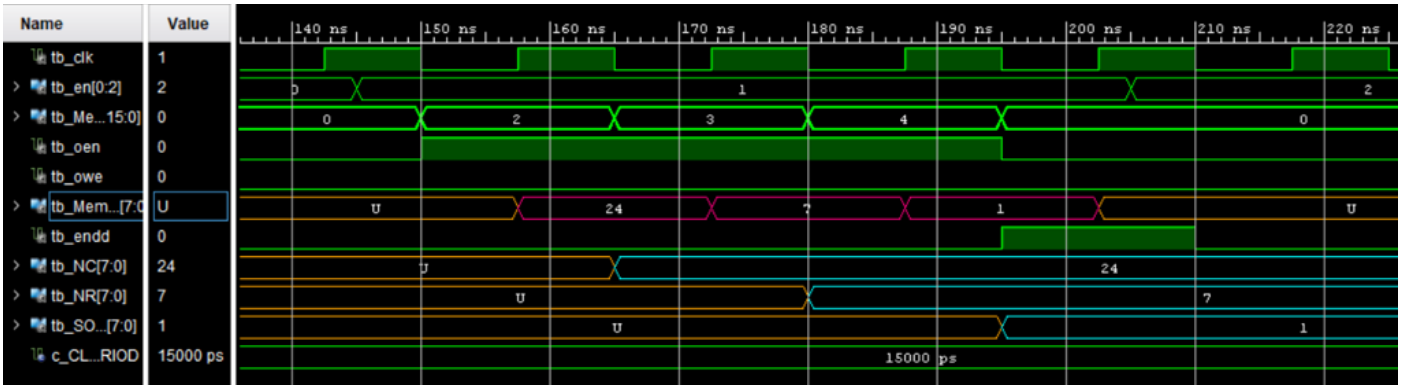
```vhdl
entity project_tb is
end project_tb;

architecture projecttb of project_tb is
    constant c_CLOCK_PERIOD : time := 15 ns;
    signal   tb_clk         : std_logic := '0';
    signal   tb_rst         : std_logic := '0';
    signal   tb_start       : std_logic := '0';
    signal   tb_found       : std_logic := '0';
    signal   tb_done        : std_logic;
    signal   tb_empty       : std_logic;
    signal   tb_en          : std_logic_vector (0 to 2);

component fsm_controller is
    port (
        i_clk          : in  std_logic;
        i_start        : in  std_logic;
        i_rst          : in  std_logic;
        i_state_end    : in  std_logic;
        i_isempty      : in  std_logic;
        o_fsm_en_state : out std_logic_vector ( 0 to 2 );
        o_done         : out std_logic
    );
end component;

begin
    UUT: FSM_CONTROLLER
        port map (
            i_clk           => tb_clk,
            i_start         => tb_start,
            i_rst           => tb_rst,
            i_state_end     => tb_found,
            i_isempty       => tb_empty,
            o_fsm_en_state  => tb_en,
            o_done          => tb_done
        );

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

    test : process is
    begin

        wait for 100 ns;
        wait for c_CLOCK_PERIOD;
        tb_rst   <= '1';
        wait for c_CLOCK_PERIOD;
        tb_rst   <= '0';
        tb_empty <= '0';
        wait for c_CLOCK_PERIOD;
        wait until falling_edge(tb_clk);
        tb_start <= '1';
        wait until falling_edge(tb_clk);
        tb_start <= '0';
        wait for c_CLOCK_PERIOD;
        wait for c_CLOCK_PERIOD;
        wait until falling_edge(tb_clk);
        tb_found <= '1';
        wait until falling_edge(tb_clk);
        tb_found <= '0';
        wait for c_CLOCK_PERIOD;
        wait for c_CLOCK_PERIOD;
        wait for c_CLOCK_PERIOD;
        wait until falling_edge(tb_clk);
        tb_found <= '1';
        wait until falling_edge(tb_clk);
        tb_found <= '0';
        wait for c_CLOCK_PERIOD;
        wait for c_CLOCK_PERIOD;
        wait for c_CLOCK_PERIOD;
        wait until falling_edge(tb_clk);
        tb_found <= '1';
        wait until falling_edge(tb_clk);
        tb_found <= '0';
        wait for c_CLOCK_PERIOD;
        wait for c_CLOCK_PERIOD;
        wait for c_CLOCK_PERIOD;
        wait until falling_edge(tb_clk);
        tb_found <= '1';
        wait until falling_edge(tb_clk);
        tb_found <= '0';
        wait for c_CLOCK_PERIOD;
        wait for c_CLOCK_PERIOD;
        wait for c_CLOCK_PERIOD;
        wait until falling_edge(tb_clk);
        tb_found <= '1';
        wait until falling_edge(tb_clk);
        tb_found <= '0';
        wait for c_CLOCK_PERIOD;
        wait for c_CLOCK_PERIOD;
        wait for c_CLOCK_PERIOD;
        wait until falling_edge(tb_clk);
        tb_found <= '1';
        wait until falling_edge(tb_clk);
        tb_found <= '0';
        wait for c_CLOCK_PERIOD;
        wait for c_CLOCK_PERIOD;
        wait for c_CLOCK_PERIOD;
        wait until falling_edge(tb_clk);
        tb_found <= '1';
        wait until falling_edge(tb_clk);
        tb_found <= '0';
        wait until rising_edge(tb_clk);

    end process test;

end projecttb;
```



LOADER

Codice del testbench e risultato della simulazione

```
entity project_tb is
end project_tb;

architecture projecttb of project_tb is
    constant c_CLOCK_PERIOD : time := 15 ns;
    signal   tb_clk         : std_logic := '0';
    signal   tb_en          : std_logic_vector (0 to 2);
    signal   tb_MemAddr     : std_logic_vector ( 15 downto 0);
    signal   tb_oen         : std_logic;
    signal   tb_owe         : std_logic;
    signal   tb_MemData     : std_logic_vector (7 downto 0);
    signal   tb_endd        : std_logic;

    signal   tb_NC, tb_NR, tb_SOGLIA : std_logic_vector ( 7 downto 0);

    type     ram_type is array (65535 downto 0) of std_logic_vector(7 downto 0);
    signal   RAM: ram_type := (
                  2 => "00011000",
                  3 =>"00000111",
                  4 => "00000001",
                  others => (others =>'0')
                  );
    component loader is
        port (
            i_clk           : in  std_logic;
            i_fsm_en_state  : in  std_logic_vector ( 0 to 2 );
            i_data          : in  std_logic_vector ( 7 downto 0 );
            o_state_end     : out std_logic;
            o_en            : out std_logic;
            o_we            : out std_logic;
            o_mem_addr      : out std_logic_vector ( 15 downto 0 );
            o_NC            : out std_logic_vector ( 7 downto 0 );
            o_NR            : out std_logic_vector ( 7 downto 0 );
            o_SOGLIA        : out std_logic_vector ( 7 downto 0 )
        );
    end component;

begin
    UUT: LOADER
        port map (
            i_clk           => tb_clk,
            i_fsm_en_state  => tb_en,
            i_data          => tb_MemData,
            o_state_end     => tb_endd,
            o_en            => tb_oen,
            o_we            => tb_owe,
            o_mem_addr      => tb_MemAddr,
            o_nc            => tb_nc,
            o_nr            => tb_nr,
            o_soglia        => tb_soglia
        );

    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) is
        begin
            if tb_clk'event and tb_clk = '1' then
                if tb_oen = '1' then
                    tb_MemData <= RAM(conv_integer(tb_MemAddr));
                else
                    tb_MemData <= "UUUUUUUU";
                end if;
            end if;
        end process;

    test : process is
    begin

        wait for 100 ns;
        wait for c_CLOCK_PERIOD;
        tb_en <= "000";
        wait for c_CLOCK_PERIOD;
        wait for c_CLOCK_PERIOD;
        ---INIZIO LOADING
        tb_en <= "001";
        wait for c_CLOCK_PERIOD;
        wait for c_CLOCK_PERIOD;
        wait for c_CLOCK_PERIOD;
        wait for c_CLOCK_PERIOD;
        ---FINE LOADING
        tb_en <= "010";
        wait until rising_edge(tb_clk);
    end process test;
end projecttb;
```



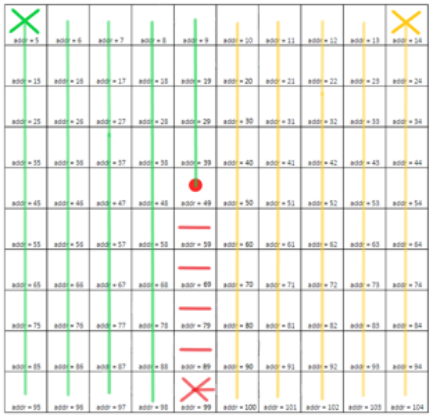
INSPECTOR

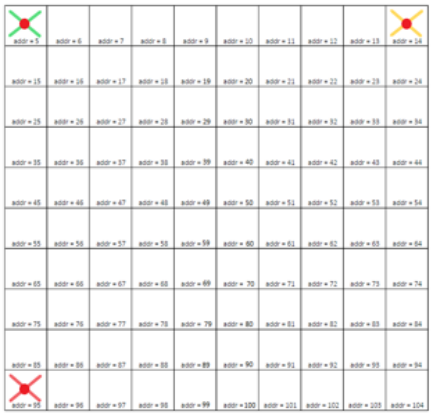
Codice del testbench (il risultato della simulazione ha dimensioni eccessive) e altri esempi.

```
entity project_tb is
end project_tb;

architecture projecttb of project_tb is
    constant c_CLOCK_PERIOD          : time := 15 ns;
    signal   tb_clk                  : std_logic := '0';
    signal   tb_en                   : std_logic_vector (0 to 2);
    signal   tb_height, tb_width     : std_logic_vector (7 downto 0);
    signal   tb_MemAddr              : std_logic_vector ( 15 downto 0);
    signal   tb_oen                  : std_logic;
    signal   tb_owe                  : std_logic;
    signal   tb_empty                : std_logic;
    signal   tb_MemData              : std_logic_vector (7 downto 0);
    signal   tb_endd                 : std_logic;
    signal   tb_NC, tb_NR, tb_SOGLIA : std_logic_vector ( 7 downto 0) := "00000000";

    type ram_type is array (65535 downto 0) of std_logic_vector(7 downto 0);
    signal RAM: ram_type := (2 => "00011000",
        3   => "00000111", 4   => "00000001",
        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')
    );

    component inspector
        port (
            i_clk          : in  std_logic;
            i_fsm_en_state : in  std_logic_vector ( 0 to 2 );
            i_data         : in  std_logic_vector ( 7 downto 0 );
            i_nr           : in  std_logic_vector ( 7 downto 0 );
            i_nc           : in  std_logic_vector ( 7 downto 0 );
            i_soglia       : in  std_logic_vector ( 7 downto 0 );
            o_height       : out std_logic_vector ( 7 downto 0);
            o_width        : out std_logic_vector ( 7 downto 0);
            o_isempty      : out std_logic;
            o_state_end    : out std_logic;
            o_en           : out std_logic;
            o_we           : out std_logic;
            o_mem_addr     : out std_logic_vector ( 15 downto 0 )
        );
    end component;


begin
    UUT: INSPECTOR
        port map (
            i_clk          => tb_clk,
            i_fsm_en_state => tb_en,
            i_data         => tb_MemData,
            I_NC           => tb_nc,
            I_NR           => tb_nr,
            I_SOGLIA       => tb_soglia,
            o_height       => tb_height,
            o_width        => tb_width,
            o_isempty      => tb_empty,
            o_state_end    => tb_endd,
            o_en           => tb_oen,
            o_we           => tb_owe,
            o_mem_addr     => tb_MemAddr
        );

    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) is
    begin
        if tb_clk'event and tb_clk = '1' then
            if tb_oen = '1' then
                tb_MemData <= RAM(conv_integer(tb_MemAddr));
            else
                tb_MemData <= "UUUUUUUU";
            end if;
        end if;
    end process;

    test : process is
    begin
        wait for 100 ns;
        wait for c_CLOCK_PERIOD;
        tb_en <= "000";
        wait for c_CLOCK_PERIOD;
        wait for c_CLOCK_PERIOD;
        wait for c_CLOCK_PERIOD;
        wait for c_CLOCK_PERIOD;
        tb_en <= "001";
        TB_NC <= "00011000";
        TB_NR <= "00000111";
        TB_SOGLIA <= "00000001";
        wait for c_CLOCK_PERIOD;
        wait for c_CLOCK_PERIOD;
        wait for c_CLOCK_PERIOD;
        ---INIZIO ISPEZIONE
        tb_en <= "010";
        wait until tb_endd = '1';
        wait for c_CLOCK_PERIOD;
        tb_en <= "011";
        wait until tb_endd = '1';
        wait for c_CLOCK_PERIOD;
        tb_en <= "100";
        wait until tb_endd = '1';
        wait for c_CLOCK_PERIOD;
        tb_en <= "101";
        wait until tb_endd = '1';
        ---FINE ISPEZIONE

        wait for c_CLOCK_PERIOD;
        tb_en <= "110";
        wait for c_CLOCK_PERIOD;
        wait for c_CLOCK_PERIOD;
        wait for c_CLOCK_PERIOD;
        wait for c_CLOCK_PERIOD;
        wait for c_CLOCK_PERIOD;
        wait until rising_edge(tb_clk);
    end process test;
end projecttb;
```

Nel corso dello sviluppo del progetto sono stati eseguiti diversi testbench di casi particolari volti a vericare la bontà dell' algoritmo di ispezione su griglie di varie dimensioni. Per brevità sono riportati solo i frammenti riguardanti la RAM, e le illustrazioni delle celle visitate dall' *INSPECTOR [si veda la legenda a pag 3]*. Per eseguirne la simulazione è semplicemente necessario sostituire il codice all'interno di uno dei testbench fornitici.

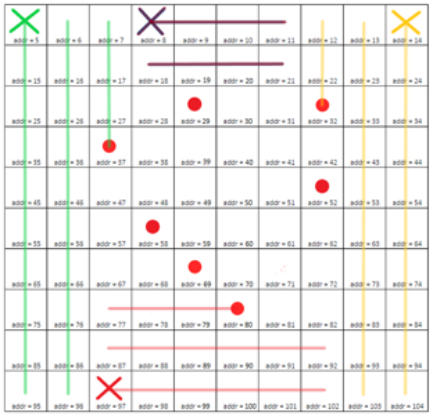
```
-- Test 1 solo pixel >= la soglia
type ram_type is array (65535 downto 0) of std_logic_vector(7 downto 0);
signal RAM: ram_type := (2 => "00001010",
    3 => "00001010", 4  => "00000010",
    7 => "00000001", 49 => "00000010",
    others => (others =>'0')
    );
.....................................................................

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





```
-- Test area di interesse coincidente con la dimensione della matrice
type ram_type is array (65535 downto 0) of std_logic_vector(7 downto 0);
signal RAM: ram_type := (2 => "00001010",
    3  => "00001010", 4  => "00000010",
    5  => "00000010", 14 => "00000011",
    95 => "00000111", others => (others =>'0')
    );
.....................................................................

assert RAM(1) = "00000000" report "FAIL high bits" severity failure;
assert RAM(0) = "01100100" report "FAIL low bits"  severity failure;
```

```
-- Test generico
type ram_type is array (65535 downto 0) of std_logic_vector(7 downto 0);
signal RAM: ram_type := (2 => "00001010",
    3  => "00001010", 4  => "00000101",
    29 => "00100101", 32 => "00100100",
    36 => "00000010", 37 => "00000101",
    52 => "00000111", 58 => "00001111",
    59 => "00000011", 69 => "00000101",
    80 => "00010001", others => (others =>'0')
    );
.....................................................................

assert RAM(1) = "00000000" report "FAIL high bits" severity failure;
assert RAM(0) = "00100100" report "FAIL low bits"  severity failure;
```



```
-- Test per verificare la corretta scrittura in high e low bits
type ram_type is array (65535 downto 0) of std_logic_vector(7 downto 0);
signal RAM: ram_type := (2 => "00011000",
    3   => "00011011", 4   => "00000001",
    62  => "00000001", 638 => "00000001",
    118 => "00000001", 550 => "00000001",
    502 => "00000001", 487 => "00000001",
    others => (others =>'0')
    );
.....................................................................

assert RAM(1) = "00000001" report "FAIL high bits" severity failure;
assert RAM(0) = "10010000" report "FAIL low bits"  severity failure;
```

MULTIPLIER

Codice del testbench e risultato della simulazione

```
entity project_tb is
end project_tb;

architecture projecttb of project_tb is
    constant c_CLOCK_PERIOD     : time := 15 ns;
    signal  tb_clk              : std_logic := '0';
    signal  tb_en               : std_logic_vector (0 to 2);
    signal  tb_empty            : std_logic;
    signal  tb_height, tb_width : std_logic_vector (7 downto 0);
    signal  tb_MemAddr          : std_logic_vector ( 15 downto 0);
    signal  tb_oen              : std_logic;
    signal  tb_owe              : std_logic;
    signal  tb_MemData          : std_logic_vector (7 downto 0);
    signal  tb_endd             : std_logic;

    component multiplier is
        port (
            i_clk           : in  std_logic;
            i_fsm_en_state  : in  std_logic_vector ( 0 to 2 );
            i_isempty       : in  std_logic;
            i_height        : in  std_logic_vector ( 7 downto 0);
            i_width         : in  std_logic_vector ( 7 downto 0);
            o_state_end     : out std_logic;
            o_en            : out std_logic;
            o_we            : out std_logic;
            o_data          : out std_logic_vector ( 7 downto 0);
            o_mem_addr      : out std_logic_vector ( 15 downto 0 )
        );
    end component;

begin
    UUT: MULTIPLIER
        port map (
            i_clk           => tb_clk,
            i_fsm_en_state  => tb_en,
            i_isempty       => tb_empty,
            i_height        => tb_height,
            i_width         => tb_width,
            o_state_end     => tb_endd,
            o_en            => tb_oen,
            o_we            => tb_owe,
            o_data          => tb_MemData,
            o_mem_addr      => tb_MemAddr
    );

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

    test : process is
    begin
        wait for 100 ns;
        wait for c_CLOCK_PERIOD;
        wait for c_CLOCK_PERIOD;
        tb_en       <= "000";
        tb_empty    <= '0';
        tb_height   <= "00001000";
        tb_width    <= "00000010";
        wait for c_CLOCK_PERIOD;
        --INIZIO MULTIPLIER
        tb_en       <= "110";
        wait until rising_edge(tb_endd);
        wait for c_CLOCK_PERIOD;
        --FINE MULTIPLIER
        tb_en       <= "000";
        wait until rising_edge(tb_clk);
    end process test;
end projecttb;
```