POLITECNICO DI MILANO

Corso di Laurea in Ingegneria Informatica

PROVA FINALE RETI LOGICHE

2017/2018

Professore: Fabrizio Ferrandi - Pietro Fezzardi

Progetto di:

Ibrahim El Shemy
Marco Gasperini

Matricola:

847621
847650

Codice Persona:

10491265
10533178

ESECUZIONE DEL PROGRAMMA

Il programma si comporta come una FSM composta da:

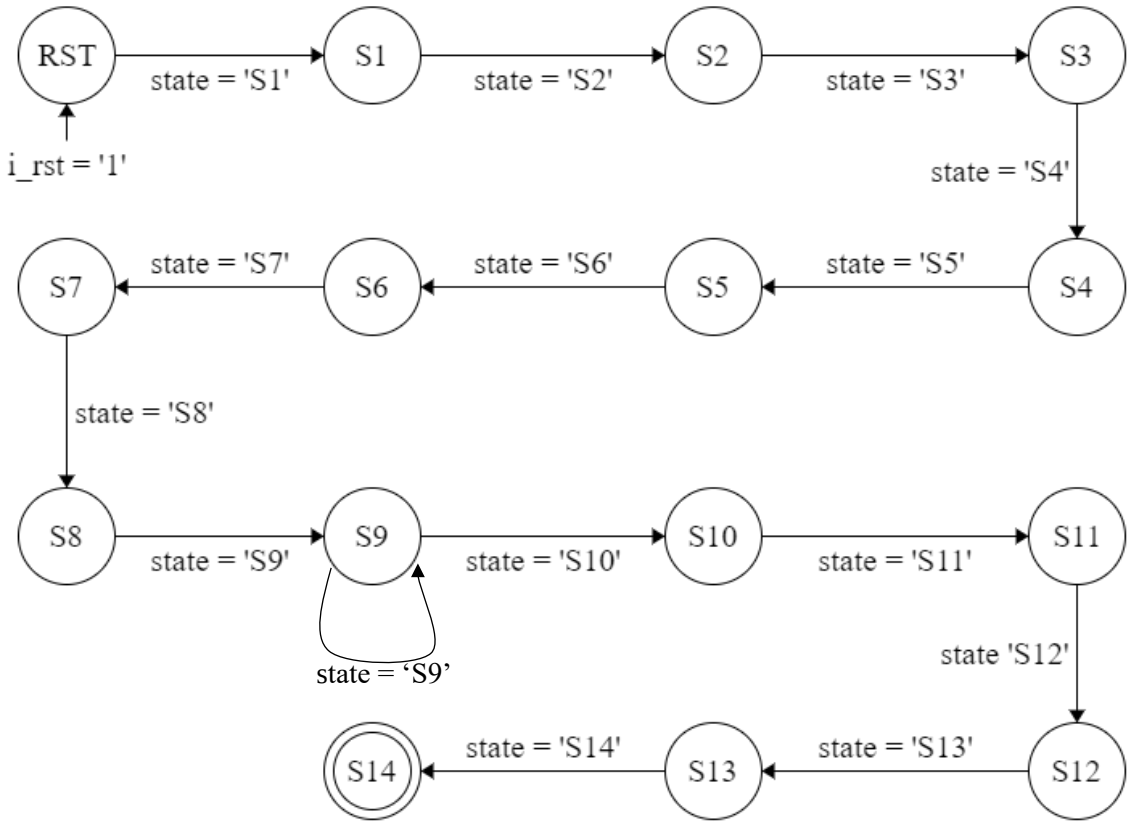
- 14 stati ai quali si accede attraverso un case regolato dal segnale “State”;
- 1 stato di reset nel quale è possibile entrare ogniqualvolta il segnale i_rst viene portato a ‘1’(a prescindere dallo stato in cui ci si trova).

A seguire ci sarà una descrizione dettagliata di tutti gli stati con uno schema rappresentativo (le parole in corsivo indicano i nomi dei *segnali).*

- **STATO RST**: la macchina si occupa di portare a ‘0’ alcuni segnali:
    - *o_address* viene portato a 2, pronto per ricevere il valore delle colonne;
    - *trovato_maxRIGA* viene portato a ‘0’. È un flag che verrà utilizzato dallo stato S7;
    - *riga* e *colonna* vengono riportati a 1 per rendersi utilizzabili in alcuni controlli nello stato S7
    - *state* viene modificato in S1 per far ripartire il programma dall’inizio
- **STATO S1**: la macchina verifica che il segnale *i_start* sia a ‘1’, dopodiché si occupa di portare *o_en* a ‘1’ e *o_we* a ‘0’ in modo da predisporsi alla lettura e non alla scrittura della memoria. Infine modifica il segnale *State* in S2.
- **STATO S2**: dopo essere stata letta la terza cella di memoria(posizione 2) e il valore in essa contenuto, viene assegnato quel valore al segnale *colonne* (quando riceviamo il segnale di reset non azzeriamo *o_address* in questo modo nello State S2 al segnale colonne viene assegnato *i_data* relativo ad *o_address* non azzerato). Successivamente portiamo il segnale *o_address* a 3, *minCOLONNA* prende il valore delle colonne da *i_data* e modifichiamo il segnale *State* in S3.
- **STATO S3**: dopo essere stata letta la quarta cella di memoria e il valore in essa contenuto, viene assegnato quel valore al segnale *righe*. Infine viene portato il segnale *o_address* a ‘4’ e il segnale *State* a S4.
- **STATO S4**: dopo essere stata letta la quinta cella di memoria e il valore in essa contenuto, viene assegnato quel valore al segnale *soglia*. Viene inoltre assegnato al segnale *controllo* il valore di righe*colonne +4 che ci servirà nello stato S7. Dopodiché viene modificato il segnale *State* in S5
- **STATO S5**: in questo stato ci limitiamo ad assegnare al segnale *temp_address* il numero binario ‘100’(4_{dec}). (il segnale *temp_address* verrà successivamente

incrementato e assegnato ad *o_address* per scorre la memoria). Per terminare porto *State* in S6.

- **STATO S6**: incremento *temp_address* e lo assegno ad *o_address*. Mi trovo così a leggere la memoria nella quinta cella, ovvero dove si trova il primo valore valido per la figura da analizzare. Concludo portando *State* in S7.
- **STATO S7**: questo stato contiene l'algoritmo principale che ci permette di leggere tutta la memoria e di salvare le coordinate (dei pixel nella figura in memoria) che ci serviranno per il calcolo dell'area. L'algoritmo funziona in questo modo: viene letta linearmente tutta la memoria, ai segnali *minRIGA, maxRIGA, minCOLONNA, maxCOLONNA* vengono assegnate rispettivamente le coordinate dei pixel più in basso, più in alto, più a sinistra e più a destra. L'algoritmo termina quando viene letta l'ultima cella di memoria contenente valori validi per la figura di riferimento (infatti l'algoritmo viene iterato finchè *temp_address* è uguale a *controllo*). Quando l'ultimo valore è stato letto *State* viene modificato in S8.
- **STATE S8**: in questo stato, nel caso in cui il seganle *trovato_maxRIGA* sia uguale a zero, l'area viene automaticamente impostata a 0 e *state* viene portato in S10. Altrimenti si procede a calcolare la base e l'altezza dell'area del rettangolo e successivamente si porta *state* in S9.
- **STATE S9**: in questo stato viene calcolata l'area nel caso in cui la nostra figura contenga dei valori superiori alla soglia e il suo valore viene assegnato al segnale *area (*l'algoritmo è iterativo ed evita volutamente la moltiplicazione per essere più efficiente). Al termine del calcolo dell'area portiamo *State* in S10
- **STATE S10**: in questo stato riportiamo il segnale *o_address* a '0' (per scrivere i valori dell'area nella corretta cella di memoria) e abilitiamo la scrittura portando a '1' il segnale *o_we*. Per concludere portiamo *State* in S11.
- **STATE S11**: in questo stato assegniamo ad *o_data*, che si occuperà di mandare alla memoria il dato da scrivere, la parte meno significativa dell'area che andrà inserita nella cella 0 della memoria. Per terminare portiamo *State* in S12.
- **STATE S12**: in questo stato incremento il segnale *o_address* per spostarmi sulla cella 1 della memoria nella quale andrò a salvare la parte più significativa dell'area. Infine porto *State* in S13.
- **STATE S13**: in questo stato assegniamo a *o_data* la parte più significativa dell'area e portiamo a '1' il segnale *o_done* (per un ciclo di clock) per avvertire la macchina che il corretto valore dell'area è stato salvato in memoria. Portiamo *State* in S14.
- **STATE S14**: in questo stato portiamo i segnali *o_done* e *o_en* a '0' e portiamo la macchina nello stato di reset assegnando a *State* RST.



TESTING

Oltre ai test predefiniti caricati dal tutor abbiamo testato alcuni casi limite che avrebbero potuto creare problemi o conflitti. Qui di seguito i principali test con la loro descrizione, in fondo al documento è possibile vedere alcuni esempi col codice (il codice non è completo ma riporta solo le parti da sostituire ai testbench originali).

- testValoriInizioFine: questo test consiste nel controllare se il programma funziona in casi in cui gli unici valori sopra alla soglia siano nella prima cella o nell’ultima cella di memoria;
- testReset: mando al programma diversi segnali di reset, sia durante che al termine della computazione;
- testOneRow: test con una sola riga;
- TestOneCols: test con una sola colonna;
- TestRowColsMax: test con righe e colonne al massimo valore;

e altri test casuali con aree più grandi e più piccole rispetto a quelle proposte dai testbench del tutor.

CONCLUSIONI

Dopo aver terminato il programma abbiamo apportato alcune modifiche al fine di renderlo più efficiente sia da un punto di vista prestazionale che da quello del consumo di risorse. Abbiamo cercato il giusto compromesso tra una buona frequenza (145 MHz) e un numero ridotto di componenti logiche.
Di seguito riportiamo la tabella relativa all'utilizzo delle risorse presa direttamente dal progetto su Vivado.

Resource	Utilization	Available	Utilization %
LUT	216	133800	0.16
FF	170	267600	0.06
IO	38	285	13.33
BUFG	1	32	3.13

TEST VALORE NELL'ULTIMA CELLA

NUOVA MEMORIA:

```
type ram_type is array (65535 downto 0) of std_logic_vector(7 downto 0);
signal RAM: ram_type := (2 => "00011000", 3 =>"00000111",4 => "00000111", 172 => "00000111", others =>
(others =>'0'));
```

NUOVO ASSERT:

```
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;
```

TEST VALORE NELLA PRIMA CELLA

NUOVA MEMORIA:

```
type ram_type is array (65535 downto 0) of std_logic_vector(7 downto 0);
signal RAM: ram_type := (2 => "00011000", 3 =>"00000111",4 => "00000111", 5 => "00000111", others =>
(others =>'0'));
```

NUOVO ASSERT:

```
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;
```

TEST RESET

NUOVO PROCESSO:

```
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);
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);
end process test;
```

NUOVO PROCESSO:

```
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 for 300 ns;
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);
end process test;
```