Prova finale (Progetto di Reti Logiche)

A.A.: 2018 - 2019

Indice

1 Specifica **2**
1.1 Descrizione 2
1.2 Interfaccia 3

2 Implementazione **4**
2.1 FSM 5
2.1.1 Stato `FS` 5
2.1.2 Stato `W_CLOCK` 5
2.1.3 Stato `R_MASK` 6
2.1.4 Stato `R_X` 6
2.1.5 Stato `R_Y` 6
2.1.6 Stato `DONE` 7

3 Risultati **8**
3.1 Testbench 8
3.2 Efficenza 9

Bibliografia **10**

Capitolo 1

Specifica

Componente *HW* per la valutazione di centroidi più vicini ad un centroide di riferimento.

1.1 Descrizione

Il componente ha il compito di calcolare, dato un centroide, gli $N \leq 8$ centroidi più vicini ad esso.
Lo spazio è costituito da un quadrato 256x256 e le coordinate sono memorizzate in una memoria *RAM*, la quale non è parte del progetto. Viene inoltre fornita una maschera d'ingresso che serve al filtrare i centroidi in memoria, verranno infatti considerati solo quelli il cui bit corrispondente è settato a 1.
La maschera d'uscita evidenzierà il o i centroidi più vicini, avendo il/i bit corrispondenti ad uno.
Le maschere utilizzano una notazione *Little-Endian*.

Indirizzo	*Contenuto*
0	Maschera in ingresso
1	Coordianata x, centroide 1
2	Coordianata y, centroide 1
3	Coordianata x, centroide 2
4	Coordianata y, centroide 2
5	Coordianata x, centroide 3
6	Coordianata y, centroide 3
7	Coordianata x, centroide 4
8	Coordianata y, centroide 4
9	Coordianata x, centroide 5
10	Coordianata y, centroide 5
11	Coordianata x, centroide 6
12	Coordianata y, centroide 6
13	Coordianata x, centroide 7
14	Coordianata y, centroide 7
15	Coordianata x, centroide 8
16	Coordianata y, centroide 8
17	Coordianata x, punto da valutare
18	Coordianata y, punto da valutare
19	Maschera in uscita

Tabella 1.1: Mappa della memoria

1.2 Interfaccia

Il componente si interfaccia con la memoria tramite i_data, $o\_address$, o_en, o_we, o_data e con il *test bench* tramite i_clk, i_start, i_rst.

```
entity project_reti_logiche is
    port (
        i_clk, i_start, i_rst : in std_logic;
        i_data                : in std_logic_vector(7 downto 0);
        o_address             : out std_logic_vector(15 downto 0);
        o_done, o_en, o_we    : out std_logic;
        o_data                : out std_logic_vector(7 downto 0)
    );
end entity;
```

Capitolo 2

Implementazione

Per implementare la specifica si è scelto di usare una *macchina a stati finiti*. Si è scelto, inoltre, di sincronizzare le transizioni tra i vari stati sul *fronte di discesa* del clock in modo tale da poter eseguire tutte le operazioni in una singola occasione. Non sarebbe stato possibile farlo sul fronte di salita in quanto sono presenti dei ritardi nella memoria.

2.1 FSM

Come poi approfondito in 2.1.5 la tabella può variare in base alla maschera in ingresso. Qui è riportata quella nel caso peggiore ovvero $Mask$ = `11111111`.

S				S^*		
`state`	*step*	*i_start*	*i_rst*	`state`	*step*	*o_done*
–	–	–	1	W_CLOCK	–	0
W_CLOCK	–	1	0	R_MASK	–	0
R_MASK	–	1	0	R_X	8	0
R_X	8	1	0	R_Y	8	0
R_Y	8	1	0	R_X	0	0
R_X	0	1	0	R_Y	0	0
R_Y	0	1	0	R_X	1	0
R_X	1	1	0	R_Y	1	0
R_Y	1	1	0	R_X	2	0
R_X	2	1	0	R_Y	2	0
R_Y	2	1	0	R_X	3	0
R_X	3	1	0	R_Y	3	0
R_Y	3	1	0	R_X	4	0
R_X	4	1	0	R_Y	4	0
R_Y	4	1	0	R_X	5	0
R_X	5	1	0	R_Y	5	0
R_Y	5	1	0	R_X	6	0
R_X	6	1	0	R_Y	6	0
R_Y	6	1	0	R_X	7	0
R_X	7	1	0	R_Y	7	0
R_Y	7	1	0	DONE	–	1
DONE	–	0	0	W_CLOCK	–	0

Tabella 2.1: Tabella delle transizioni

2.1.1 Stato FS

Stato iniziale in cui si trova la macchina al primo avvio. La macchina è sensibile solo al segnale di *reset* che la porta nello stato W_CLOCK. Inoltre esso imposterà le uscite del componente in modo che sia pronto ad incominciare l'elaborazione appena il segnale di *start* commuta a 1.

2.1.2 Stato W_CLOCK

Essendo la transizione tra stati sincronizzata con il fronte di discesa del clock e non avendo la certezza che tra il segnale di *reset* e quello di *start* sia passato almeno un ciclo di clock, questo stato serve ad assicurare un output della memoria corretto.

2.1.3 Stato `R_MASK`

In questo stato viene letta la maschera in ingresso, sulla quale è però necessario fare delle considerazioni.
Per ottimizzare la *FSM* (in termini di tempo di esecuzione), è già possibile portare a termine l'elaborazione se

$$Mask \in \{0 \vee 2^n, n \in [0, 7]\}$$

ovvero se la maschera contiene non più di un bit a 1.[1]

```
if unsigned(i_data and std_logic_vector(signed(i_data) - 1)) = 0 then
    --la maschera contiene zero o un bit a 1.
else
    --la maschera contiene almeno due bit a 1.
end if;
```

in questo caso la maschera d'uscita sarà uguale a quella in ingresso.

Nel caso, invece, avessimo due o più centroidi da verificare bisognerà procedere con il calcolo della *Manhattan distance* di ognuno.

2.1.4 Stato `R_X`

In questo stato viene letta la coordinata x del centroide, in base allo step attuale.

$$Addr_x = step * 2 + 1$$

Lo step 8 si riferisce al centroide di riferimento.

2.1.5 Stato `R_Y`

In questo stato viene letta la coordinata y del centroide, in base allo step attuale.

$$Addr_y = step * 2 + 2$$

Ad ogni *step*, ad esclusione di 8, si calcola la *Manhattan distance* con il riferimento

```
abs(to_integer(unsigned(x)) - to_integer(unsigned(c_x))) +
abs(to_integer(unsigned(i_data)) - to_integer(unsigned(c_y)))

-- x: coordiata x del centroide attuale
-- c_x: coordiata x del centroide da valutare
-- i_data: coordiata y del centroide attuale (non è necessario
-- salvarne il valore)
-- c_y: coordiata y del centroide da valutare
```

e se la distanza è minore o uguale a quella più piccola calcolata finora la maschera d'uscita viene aggiornata di conseguenza.

Successivamente si incrementa *step* in modo da saltare i centroidi esclusi.

```
for i in 0 to 6 loop
    if step < 8 and i_mask(step) = '0' then step := step + 1;
    else exit;
    end if;
end loop;
```

2.1.6 Stato DONE

La computazione è stata completata e in memoria è stata scritta la maschera d'uscita, la *FSM* viene quindi riportata in W_CLOCK pronta per incominciare un'altra elaborazione.

Capitolo 3

Risultati

Il componente supera correttamente la simulazione in *Behavioral* e *Post-Syntesis*, inoltre non presenta *warning*.

3.1 Testbench

Per testare il componente, oltre a il test fornito come esempio, ho realizzato un programma in C che realizzasse dei test per i casi limite, oltre che a una serie di test casuali.
I casi limite includono le seguenti maschere in ingresso: `00000000`, `10000000`, `00000001`, `11111111`, `00010000`.

```
int mem[20];
int min = 511;
srand(time(0));
for (int i = 0; i < 19; i++) mem[i] = rand() % 256;
for (int i = 0; i < 8; i++) if (mem[0] & 1 << i == 1 << i) {
    int dist = abs(mem[17] - mem[i * 2 + 1]) +
               abs(mem[18] - mem[i * 2 + 2]);
    if (dist < min) {
        min = dist;
        mem[19] = 1 << i;
    } else if (dist == min) mem[19] += 1 << i;
}

//successivamente il risultato viene scritto su file vhd, pronto
//per essere importato in vidado
```

3.2 Efficenza

Il componente ha la massima efficenza in termini di cicli di clock richiesti all'elaborazione del risultato, infatti esso termina l'elaborazione in

$$Cicli(n) = \begin{cases} 2 * n + 4 & : n \in [2, 7] \\ 2 & : n \in [0, 1] \end{cases}$$

dove n è il numero di centroidi da considerare.

Componente	*Utilizzo*
LUT	125
FF	73
BUFG	1

Tabella 3.1: Utilizzo componenti FPGA

Bibliografia

[1] S. E. Anderson, *Bit twiddling hacks*. Available at: `https://graphics.stanford.edu/~seander/bithacks.html#DetermineIfPowerOf2`, 2005.