POLITECNICO
MILANO 1863

Politecnico di Milano

051228 - Progetto di Reti Logiche

Prova Finale

Documentazione

Autori:	*Matricole:*
Stefano Formicola	847762
Leonardo Guerra	844624

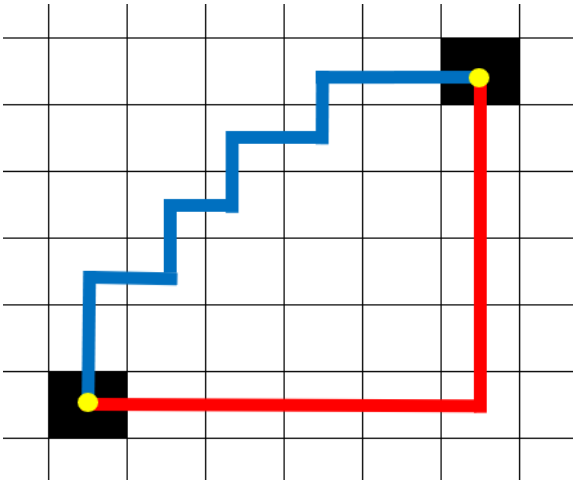
Indice

1 Introduzione e specifiche di progetto

La seguente documentazione è frutto del lavoro di Stefano Formicola e Leonardo Guerra, nell'ambito dell'implementazione del progetto per la *Prova Finale di Reti Logiche* del Corso di Laurea in *Ingegneria Informatica*.

Lo scopo del progetto è la realizzazione di un componente hardware scritto nel linguaggio VHDL, con l'ausilio del tool di sintesi e analisi Xilinx Vivado Webpack.

Dati uno spazio bidimensionale (di dimensioni 256x256), le coordinate di un punto di riferimento e 8 *centroidi* appartenenti a tale spazio, il componente da descrivere valuta quale (o quali) dei centroidi risulta più vicino al punto secondo la *Manhattan Distance* (di seguito una figura esemplificativa di come i due percorsi tra i due punti siano equivalenti per la Manhattan Distance).



Inoltre, una maschera in ingresso specifica quali dei centroidi devono essere considerati per la valutazione della distanza (elencati dal bit meno significativo al più significativo, cioè nella posizione più a destra il centroide 1, nella posizione più a sinistra il centroide 8).

Il risultato della valutazione della distanza è riportato in una maschera di uscita che ha un "1" in corrispondenza dei centroidi a minima distanza dal punto (i centroidi seguono lo stesso ordinamento della maschera d'ingresso).

Tutte le simulazioni, *Behavioural*, *Post-Synthesis Functional*, *Post-Synthesis Timing* e *Post-Implementation*, sono state effettuate sulla FPGA `xc7a200tfbg484-1`.

L'interfaccia del componente implementato è la seguente:

```
entity 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);
                o_address : out std_logic_vector(15 downto 0);
                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 project_reti_logiche;
```

2 Scelte progettuali

Abbiamo scelto di implementare una FSM con 6 stati principali (*IDLE*, *START*, *RESET*, *PROCESSING*, *ENDING*, *DONE*), dei quali 3 si compongono a loro volta di stati secondari:

- *START* ha *readInMask* (per leggere dalla RAM la maschera in ingresso), *readX* e *readY* (che prendono le coordinate del punto di riferimento);
- *ENDING* ha *writeMask* (che scrive in memoria la maschera in uscita);
- *PROCESSING* ha *checkMask*, lo stato principale per la gestione della FSM, che valuta se il centroide corrente ha il bit relativo nella maschera in ingresso a "1" (continuando l'esecuzione sullo stesso centroide, o passando alla valutazione del centroide successivo in caso contrario) e controlla se i centroidi siano già stati valutati tutti, spostandosi in tale eventualità nello stato di *ENDING*. *getCentroidX*, *getCentroidY* sono gli stati che prelevano dalla RAM le coordinate del centroide corrente, *evaluateDist* confronta la distanza del centroide corrente dal punto di riferimento con quella trovata fino a quel momento, *modifyOutMask*, infine, modifica la maschera d'uscita ogni qualvolta sia trovata una distanza minore o uguale alla minima.

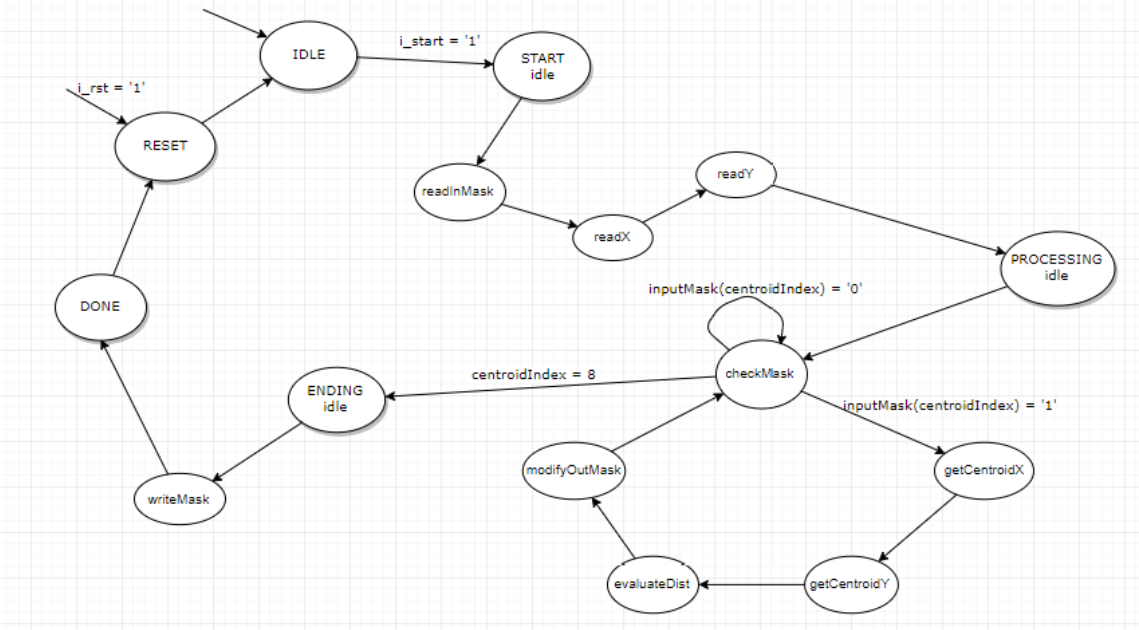
Inoltre, uno stato di *idle* è utilizzato da *START*, *PROCESSING* e *ENDING* per supportare le transizioni di stato. Il segnale di *reset* è asincrono, mentre la commutazione degli stati avviene sul fronte di salita del *clock*.

Per la gestione dell'indice del centroide corrente abbiamo definito un segnale *centroidIndex*, utile anche per definire l'indirizzo di memoria dal quale prendere le coordinate del centoide corrente.

Se il centroide corrente ha distanza minore o uguale alla distanza minima trovata fino a quel momento, utilizziamo un segnale denominato *oneHotIndex*, vettore di 8 bit che rappresenta la codifica `One-hot` del centroide in analisi.

Se a distanza pari alla minima trovata fino a quel passo, la maschera d'uscita è posta in *OR* con la codifica one-hot del centroide; se a distanza minore, invece, la maschera d'uscita viene azzerata e posta uguale alla variabile *oneHotIndex*.

Di seguito uno diagramma degli stati della FSM implementata.



3 Test

Per testare il funzionamento della FSM implementata, in aggiunta al testbench di esempio fornitoci con le specifiche di progetto, abbiamo elaborato una batteria di 10 test[2], dei quali i primi 7 riguardano ognuno uno specifico caso limite, mentre gli ultimi 3 sono stati generati casualmente da un programma[3] scritto in C, per originare il contenuto della RAM del testbench in questione.

Di seguito sono riportate le casistiche esplorate da ogni testbench.

3.1 TB_1

Testa il caso in cui la maschera in ingresso abbia tutti i bit posti a 0. Quindi, a prescindere dal contenuto della RAM dall'indirizzo 1 a 18 (cioè indifferentemente da dove siano posizionati i centroidi ed il punto di riferimento), la maschera in uscita dovrà risultare con tutti i bit posti a 0.

3.2 TB_2

Testa il caso in cui tutti i centroidi siano alla stessa distanza dal punto: non essendoci tutti 1 nella maschera in ingresso la maschera in uscita non sarà un vettore interamente di 1, ma sarà semplicemente uguale alla maschera in ingresso.

3.3 TB_3

Testa il caso in cui i centroidi siano posti a distanze decrescenti dal punto (il centroide 1 più lontano, il centroide 8 più vicino). La maschera in uscita avrà, quindi, un 1 in corrispondenza del solo centroide 8.

3.4 TB_4

Testa il caso in cui il punto abbia coordinate (0,0) e, nonostante il centroide 8 sia alla stessa distanza del centroide 1, avendo il bit corrispondente nella maschera di ingresso a 0, avrà lo stesso valore del bit (a 0) nella maschera in uscita.

3.5 TB_5

Testa il caso in cui il punto di riferimento abbia coordinate (255, 255).

3.6 TB_6

Testa il caso in cui uno dei centroidi abbia coordinate corrispondenti a quelle del punto diriferimento.

3.7 TB_7

Testa il caso in cui i centroidi da 1 a 6 siano tutti alla stessa distanza dal punto di riferimento, ma il centroide 7 sostituisca la maschera in uscita, essendo alla distanza minima dal punto.

3.8 TB_8, TB_9, TB_10

Testbench generati casualmente attraverso il programma sopra citato.
Nelle due pagine seguenti è riportato uno schema dei vari testbench eseguiti.

TB_0		
inMask	**0**	1 0 1 1 1 0 0 1
C1	**1**	75
	2	32
C2	**3**	111
	4	213
C3	**5**	79
	6	33
C4	**7**	1
	8	33
C5	**9**	80
	10	35
C6	**11**	12
	12	254
C7	**13**	215
	14	78
C8	**15**	211
	16	121
P	**17**	78
	18	33
outMask	**19**	0 0 0 1 0 0 0 1

TB_1		
inMask	**0**	0 0 0 0 0 0 0 0
C1	**1**	75
	2	32
C2	**3**	111
	4	213
C3	**5**	79
	6	33
C4	**7**	1
	8	33
C5	**9**	80
	10	35
C6	**11**	12
	12	254
C7	**13**	215
	14	78
C8	**15**	211
	16	121
P	**17**	78
	18	33
outMask	**19**	0 0 0 0 0 0 0 0

TB_2		
inMask	**0**	1 0 0 1 1 1 1 0
C1	**1**	110
	2	105
C2	**3**	107
	4	108
C3	**5**	90
	6	95
C4	**7**	93
	8	92
C5	**9**	115
	10	100
C6	**11**	100
	12	85
C7	**13**	89
	14	96
C8	**15**	96
	16	89
P	**17**	100
	18	100
outMask	**19**	1 0 0 1 1 1 1 0

TB_3		
inMask	**0**	1 1 1 1 1 1 1 1
C1	**1**	55
	2	55
C2	**3**	55
	4	54
C3	**5**	54
	6	54
C4	**7**	54
	8	53
C5	**9**	53
	10	53
C6	**11**	53
	12	52
C7	**13**	52
	14	52
C8	**15**	52
	16	51
P	**17**	50
	18	50
outMask	**19**	1 0 0 0 0 0 0 0

TB_4		
inMask	**0**	0 0 0 0 1 1 1 1
C1	**1**	1
	2	1
C2	**3**	2
	4	2
C3	**5**	3
	6	3
C4	**7**	4
	8	4
C5	**9**	5
	10	5
C6	**11**	6
	12	6
C7	**13**	7
	14	7
C8	**15**	1
	16	1
P	**17**	0
	18	0
outMask	**19**	0 0 0 0 0 0 0 1

TB_5		
inMask	**0**	1 1 1 1 0 0 0 0
C1	**1**	1
	2	1
C2	**3**	2
	4	2
C3	**5**	3
	6	3
C4	**7**	4
	8	4
C5	**9**	5
	10	5
C6	**11**	6
	12	6
C7	**13**	7
	14	7
C8	**15**	254
	16	254
P	**17**	255
	18	255
outMask	**19**	1 0 0 0 0 0 0 0

TB_6		
inMask	0	0 0 1 1 1 1 0 0
C1	1	101
	2	101
C2	3	102
	4	102
C3	5	103
	6	103
C4	7	104
	8	104
C5	9	99
	10	99
C6	11	100
	12	100
C7	13	100
	14	99
C8	15	0
	16	0
P	17	100
	18	100
outMask	19	0 0 1 0 0 0 0 0

TB_7		
inMask	0	0 1 0 1 1 1 1 1
C1	1	80
	2	70
C2	3	70
	4	100
C3	5	65
	6	95
C4	7	55
	8	75
C5	9	71
	10	99
C6	11	70
	12	80
C7	13	72
	14	82
C8	15	71
	16	81
P	17	70
	18	80
outMask	19	0 1 0 0 0 0 0 0

TB_8		
inMask	0	0 1 0 0 1 0 1 0
C1	1	22
	2	254
C2	3	233
	4	240
C3	5	97
	6	15
C4	7	37
	8	59
C5	9	169
	10	148
C6	11	39
	12	125
C7	13	136
	14	230
C8	15	86
	16	50
P	17	3
	18	156
outMask	19	0 0 0 0 1 0 0 0

TB_9		
inMask	0	1 0 0 1 0 0 0 1
C1	1	70
	2	168
C2	3	98
	4	17
C3	5	13
	6	118
C4	7	167
	8	112
C5	9	91
	10	18
C6	11	208
	12	118
C7	13	166
	14	9
C8	15	27
	16	7
P	17	188
	18	39
outMask	19	0 0 0 1 0 0 0 0

TB_10		
inMask	0	1 0 0 1 1 0 1 0
C1	1	20
	2	68
C2	3	216
	4	40
C3	5	229
	6	177
C4	7	172
	8	121
C5	9	134
	10	225
C6	11	36
	12	224
C7	13	252
	14	27
C8	15	90
	16	184
P	17	25
	18	27
outMask	19	0 0 0 0 0 0 1 0

4 Risultati della sintesi

Da una prima stesura del codice dell'implementazione, dopo aver verificato il funzionamento in *Pre-Synthesis*, abbiamo proceduto ad avviare la sintesi del componente che, in seguito, presentava un design di `305 cells, 38 I/O Ports, 366 Nets` ed un funzionamento corretto in simulazione.

Successivamente la nostra attenzione si è rivolta all'ottimizzazione della FSM, prima riducendo il numero degli stati, poi utilizzando le `variables`, ove possibile, per sostituire i `signals`. Questa serie di operazioni ha ridotto il design post-synthesis a `264 cells, 325 Nets`, dopodichè abbiamo reso il segnale di reset asincrono, arrivando così a `258 cells, 319 Nets`, fino ad un minimo di 184 cells nell'analisi RTL.

Terminata l'ottimizzazione abbiamo condotto tutti i test, eseguiti correttamente in simulazione *Behavioural*, *Post-Synthesis Functional* e *Timing*.

In fase *Post-Synthesis*, oltre ai testbenches effettuati, ci siamo serviti della possibilità di specificare delle `Timing Constraints` per verificare il vincolo di funzionamento del componente a un periodo di clock di `100ns`.

A tale scopo il codice utilizzato è stato:

```
create_clock -period 100.0 -name i_clk [get_ports i_clk]
```

La conferma del funzionamento al periodo di clock richiesto è riscontrabile nei risultati della sezione `Design Timing Summary`.

Design Timing Summary

Setup		Hold		Pulse Width	
Worst Negative Slack (WNS):	90,670 ns	Worst Hold Slack (WHS):	0,153 ns	Worst Pulse Width Slack (WPWS):	49,500 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:	166	Total Number of Endpoints:	166	Total Number of Endpoints:	79

All user specified timing constraints are met.

Dai valori appare evidente come il design implementato rispetti il vincolo sul periodo di clock.

Inoltre sono stati effettuati ulteriori test del componente anche in *Post-Implementation*, che hanno confermato la frequenza di funzionamento a 10MHz.

Pur avendo soddisfatto il vincolo richiesto dalle specifiche del progetto[1], abbiamo ritenuto interessante testare il design a frequenze maggiori di 10MHz, arrivando fino ad una frequenza limite del clock di 100MHz in *Post-Synthesis* e *Post-Implementation*, come di seguito riportato.

```
create_clock -period 10.0 -name i_clk [get_ports i_clk]
```

Design Timing Summary

Setup		Hold		Pulse Width	
Worst Negative Slack (WNS):	0,670 ns	Worst Hold Slack (WHS):	0,153 ns	Worst Pulse Width Slack (WPWS):	4,500 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:	166	Total Number of Endpoints:	166	Total Number of Endpoints:	79

All user specified timing constraints are met.

5 Riferimenti

[1] Specifica del progetto di reti logiche. `https://github.com/ste7en/progetto-reti-logiche/raw/master/doc/Specifica_progetto_AA_2018_2019.pdf`, 2018.

[2] Testbench del progetto di reti logiche. `https://github.com/ste7en/progetto-reti-logiche/tree/master/test`, 2019.

[3] Testbench generator per la prova finale di reti logiche. `https://github.com/ste7en/Project-Reti-Logiche-Testbench-Generator`, 2019.

[4] K.Skahill. *VHDL for Programmable Logic*. Addison Wesley, 1996.