Tesina di Architettura dei Sistemi di Elaborazione
Gruppo 4

Milo Saverio - Mat. M63/XXXX Pommella Michele - Mat. M63/XXXX
Trimaldi Davide - Mat. M63/XXXX

16 gennaio 2018

Indice

Capitolo 1

Minimizzazione reti combinatorie

1.1 Traccia

Minimizzare le funzioni descritte dai seguenti ON-SET e DC-SET.

1. ON-SET={0, 2, 4, 8, 10, 11, 15} ; DC-SET={7, 14}
2. ON-SET={0, 1, 2, 7, 8, 10, 15}; DC-SET={5, 9, 11}
3. ON-SET={1, 3, 5, 7, 8, 9, 11, 13, 14, 15}; DC-SET={2, 12}
4. Fuzione a più uscite F1, F2, F3, descritte rispettivamente da:
   (a) ON-SET1={0, 4, 8, 9, 14, 15}; DC-SET1={1, 2, 5, 7}
   (b) ON-SET2={0, 1, 7, 9, 11, 13}; DC-SET2={4, 12}
   (c) ON-SET3={7, 9, 10, 11, 15}; DC-SET3={5, 12}

1.2 Soluzione

La minimizzazione delle funzioni sarà effettuata attraverso i differenti metodi indicati.

1.2.1 Minimizzazione 1

La prima funzione è stata minimizzata attraverso il metodo delle mappe di Karnaugh.

Si può generare la seguente tabella di verità a partire dalla definizione insiemistica:

X	Y	Z	V	F
0	0	0	0	1
0	0	0	1	0
0	0	1	0	1
0	0	1	1	0
0	1	0	0	1
0	1	0	1	0
0	1	1	0	0
0	1	1	1	-
1	0	0	0	1
1	0	0	1	0
1	0	1	0	1
1	0	1	1	1
1	1	0	0	0
1	1	0	1	0
1	1	1	0	-
1	1	1	1	1

ab \ cd	00	01	11	10
00	1			1
01	1		-	
11			1	-
10	1		1	1

Individuando i sottocubi di area massima e gli implicanti primi essenziali, perveniamo alla funzione minimizzata:

$$F = \neg x \neg z \neg v + \neg y \neg v + xz$$

1.2.2 Minimizzazione 2

Al fine di minimizzare questa funzione è stato utilizzato il metodo di Quine Mc Cluskey.

Si può generare la seguente tabella di verità a partire dalla definizione insiemistica:

X	Y	Z	V	F
0	0	0	0	1
0	0	0	1	1
0	0	1	0	1
0	0	1	1	0
0	1	0	0	0
0	1	0	1	-
0	1	1	0	0
0	1	1	1	1
1	0	0	0	1
1	0	0	1	-
1	0	1	0	1
1	0	1	1	-
1	1	0	0	0
1	1	0	1	0
1	1	1	0	0
1	1	1	1	1

Seguiamo brevemente le fasi di espansione e copertura del metodo di minimizzazione applicato.

implicante	x	y	z	v	check
0	0	0	0	0	✓
1	0	0	0	1	✓
2	0	0	1	0	✓
8	1	0	0	0	✓
5	0	1	0	1	✓
9	1	0	0	1	✓
10	1	0	1	0	✓
7	0	1	1	1	✓
11	1	0	1	1	✓
15	1	1	1	1	✓

implicante	x	y	z	v	check
0,1	0	0	0	-	✓
0,2	0	0	-	0	✓
0,8	-	0	0	0	✓
1,5	0	-	0	1	a
1,9	-	0	0	1	✓
2,10	-	0	1	0	✓
8,9	1	0	0	-	✓
8,10	1	0	-	0	✓
5,7	0	1	-	1	b
9,11	1	0	-	1	✓
10,11	1	0	1	-	✓
7,15	-	1	1	1	c
11,15	1	-	1	1	d

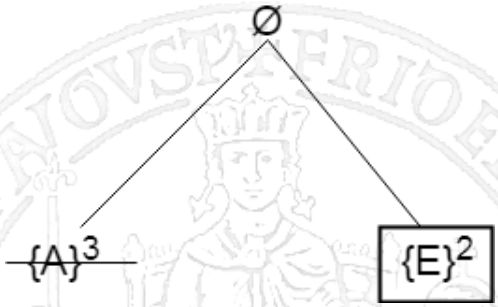
implicante	x	y	z	v	check
0,1,8,9	-	0	0	-	e
0,2,8,10	-	0	-	0	f
8,9,10,11	1	0	-	-	g

Individuati gli implicanti primi al termine della fase di espansione, si prosegue con la fase di copertura.

	0	1	2	7	8	10	15
a		X					
b				X			
c				X			X
d							X
e	X	X			X		
f	X		X		X	X	
g					X	X	

	1
a	X
e	X

Discriminando gli implicanti primi essenziali primari (f) e secondari (c), questi ultimi mediante le regole di dominanza, si ottiene una tabella cliclica non ulteriormente riducibile. Applicando il metodo Branch&Bound in questo caso banale, si prosegue nella scelta dell'implicante e per il completamento della copertura. Questa scelta si basa sul costo in termini di letterali dei due implicanti, che individua e come l'implicante che consente la copertura a costo minimo.



Analogamente, tramite il metodo di Petrick, otteniamo la somma degli implicanti che coprono il mintermine in questione:

$$A + E = 1$$

Tra essi si effettua la medesima scelta dettata dal costo dei letterali.
La funzione di uscita, dunque, è pari a:

$$F = c + e + f = yzv + \neg y \neg z + \neg y \neg v$$

1.2.3 Minimizzazione 3

Si prosegue nuovamente con metodo Quine Mc Cluskey.

Si può generare la seguente tabella di verità a partire dalla definizione insiemistica:

X	Y	Z	V	F
0	0	0	0	0
0	0	0	1	1
0	0	1	0	-
0	0	1	1	1
0	1	0	0	0
0	1	0	1	1
0	1	1	0	0
0	1	1	1	1
1	0	0	0	1
1	0	0	1	1
1	0	1	0	0
1	0	1	1	1
1	1	0	0	-
1	1	0	1	1
1	1	1	0	1
1	1	1	1	1

Attraverso la minimizzazione riusciamo ad ottenere un insieme di implicanti primi essenziali che coprono totalmente la funzione:

$$F = v + x\neg z + xy$$

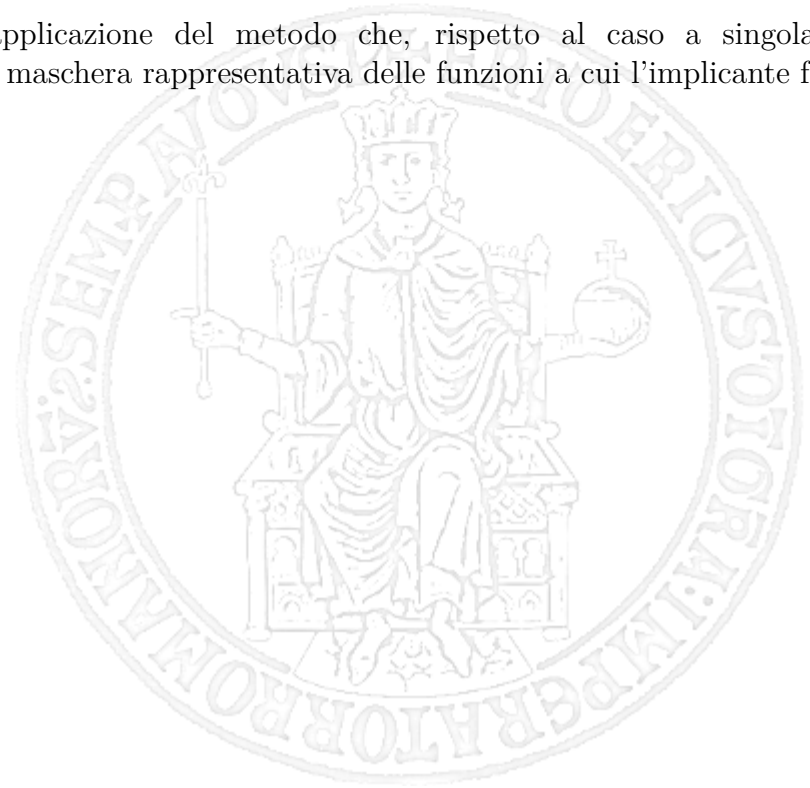
1.2.4 Minimizzazione 4

Poiché non si ricade più nel caso a singola uscita, si è adottato il metodo di Quine Mc Cluskey per funzioni a più uscite. Esso consente il riuso di implicanti comuni a più funzioni, che non si avrebbe per semplice minimizzazione individuale.

Si può generare la seguente tabella di verità a partire dalla definizione insiemistica:

X	Y	Z	V	F1	F2	F3
0	0	0	0	1	1	0
0	0	0	1	-	1	0
0	0	1	0	-	0	0
0	0	1	1	0	0	0
0	1	0	0	1	-	0
0	1	0	1	-	0	-
0	1	1	0	0	0	0
0	1	1	1	-	1	1
1	0	0	0	1	0	0
1	0	0	1	1	1	1
1	0	1	0	0	0	1
1	0	1	1	0	1	1
1	1	0	0	0	-	-
1	1	0	1	0	1	0
1	1	1	0	1	0	0
1	1	1	1	1	0	1

Percorriamo l'applicazione del metodo che, rispetto al caso a singola uscita, presenta l'introduzione della maschera rappresentativa delle funzioni a cui l'implicante fa riferimento.



implicante	x	y	z	v	F1	F2	F3	check
0	0	0	0	0	1	1	0	✓
1	0	0	0	1	-	1	0	✓
2	0	0	1	0	-	0	0	✓
4	0	1	0	0	1	-	0	✓
8	1	0	0	0	1	0	0	✓
5	0	1	0	1	-	0	-	✓
9	1	0	0	1	1	1	1	a
10	1	0	1	0	0	0	1	✓
12	1	1	0	0	0	-	-	✓
7	0	1	1	1	-	1	1	b
11	1	0	1	1	0	1	1	✓
13	1	1	0	1	0	1	0	✓
14	1	1	1	0	1	0	0	✓
15	1	1	1	1	1	0	1	✓

implicante	x	y	z	v	F1	F2	F3	check
0, 1	0	0	0	-	1	1	0	c
0, 2	0	0	-	0	1	0	0	d
0, 4	0	-	0	0	1	1	0	e
0, 8	-	0	0	0	1	0	0	✓
1, 5	0	-	0	1	-	0	0	✓
1, 9	-	0	0	1	1	1	0	f
4, 5	0	1	0	-	1	0	0	✓
4, 12	-	1	0	0	0	-	0	✓
8, 9	1	0	0	-	1	0	0	✓
5, 7	0	1	-	1	-	0	1	g
9, 11	1	0	-	1	0	1	1	h
9, 13	1	-	0	1	0	1	0	i
10, 11	1	0	1	-	0	0	1	l
12, 13	1	1	0	-	0	1	0	m
7, 15	-	1	1	1	1	0	1	n
11, 15	1	-	1	1	0	0	1	o
14, 15	1	1	1	-	1	0	0	p

implicante	x	y	z	v	F1	F2	F3	check
0, 1, 4, 5	0	-	0	-	1	0	0	q
0, 1, 8, 9	-	0	0	-	1	0	0	r

Segue la fase di copertura con gli implicanti primi trovati. Essa viene effettuata con l'obiettivo di minimizzare il costo dei letterali.

	F1						F2						F3					costo
	0	4	8	9	14	15	0	1	7	9	11	13	7	9	10	11	15	
a				X						X				X				4
b									X				X					4
c	X						X	X										3
d	X																	3
e	X	X					X											3
f				X				X		X								3
g													X					3
h										X	X			X		X		3
i										X		X						3
l															X	X		3
m												X						3
n						X							X				X	3
o																X	X	3
p					X	X												3
q	X	X																2
r	X		X	X														2

	F1	F2			F3			costo
	4	0	1	13	7	9	15	
a						X		4
b					X			1
c		X	X					3
e	X	X						3
f			X					3
g					X			3
h						X		1
i				X				3
m				X				3
n					X		X	3
o							X	3
q	X							2

	F1	F2			F3			costo
	4	0	1	13	7	9	15	
b					X			1
c		X	X					3
e	X	X						3
h						X		1
i				X				3
m				X				3
n					X		X	3
q	X							2

	F1	F2	costo
	4	13	
e	X		3
i		X	3
m		X	3
q	X		2

	F1	F2	costo
	4	13	
i		X	3
m		X	3
q	X		2

	F2	costo
	13	
i	X	3
m	X	3

Si arriva ad una tabella ciclica, i cui implicanti presentano anche lo stesso costo. Per la minimizzazione si è scelto i, ottenendo:

$$F1 = r + p + q = \neg y \neg z + xyz + \neg x \neg z$$

$$F2 = b + h + c + i = \neg xyzv + x \neg yv + \neg x \neg y \neg z + x \neg zv$$

$$F3 = l + n + h = x \neg yz + yzv + x \neg yv$$

Capitolo 2

Reti combinatorie con l'ausilio di SIS e Mapping Tecnologico

2.1 Traccia

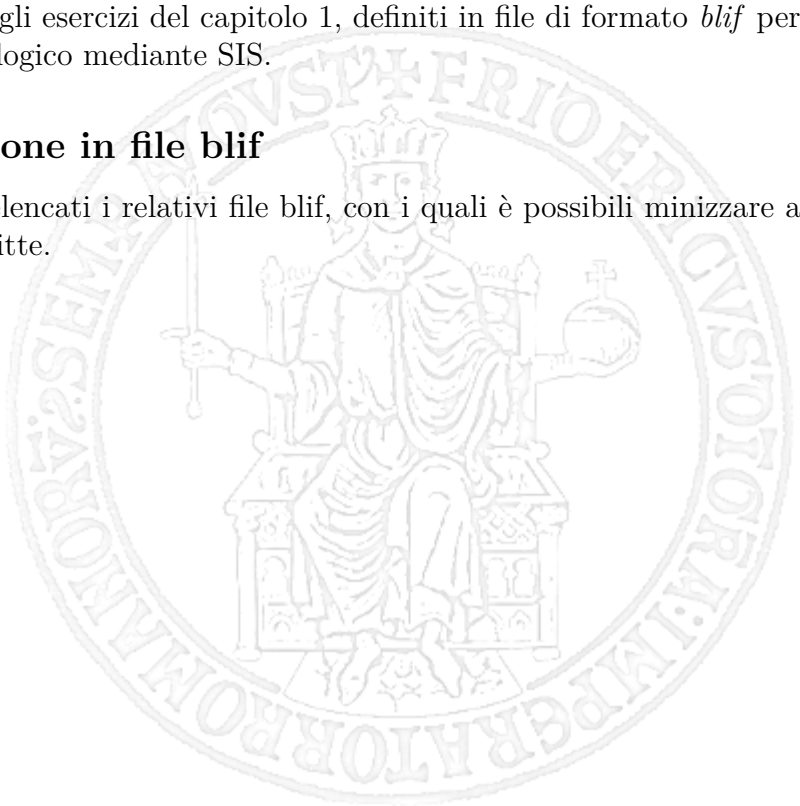
Utilizzare lo strumento automatico SIS per la minimizzazione di circuiti combinatori e il relativo mapping tecnologico.

2.2 Soluzione

Si farà riferimento agli esercizi del capitolo 1, definiti in file di formato *blif* per la minimizzazione ed il mapping tecnologico mediante SIS.

2.2.1 Descrizione in file blif

Di seguito verrano elencati i relativi file blif, con i quali è possibili minizzare automaticamente le funzioni sopra descritte.



In riferimento a 1.2.1

```
.model esercizio1
.inputs a b c d
.outputs f

.names a b c d f
0000 1
0010 1
0100 1
1000 1
1010 1
1011 1
1111 1

.exdc
.inputs a b c d
.outputs f

.names a b c d f
0111 1
1110 1

.end
```

In riferimento a 1.2.2

```
1  .model esercizio2
2  .inputs a b c d
3  .outputs f
4
5  .names a b c d f
6  0000 1
7  0001 1
8  0010 1
9  0111 1
10 1000 1
11 1010 1
12 1111 1
13
14 .exdc
15 .inputs a b c d
16 .outputs f
17
18 .names a b c d f
19 0101 1
20 1001 1
21 1011 1
22
23 .end
```

In riferimento a 1.2.3

```
.model esercizio3
.inputs a b c d
.outputs f

.names a b c d f
0001 1
0011 1
0101 1
0111 1
1000 1
1001 1
1011 1
1101 1
1110 1
1111 1

.exdc
.inputs a b c d
.outputs f

.names a b c d f
0010 1
1100 1

.end
```

In riferimento a 1.2.4

```
.model esercizio4
.inputs a b c d
.outputs f1 f2 f3

.names a b c d f1
0000 1
0100 1
1000 1
1001 1
1110 1
1111 1

.names a b c d f2
0000 1
0001 1
0111 1
1001 1
1011 1
1101 1

.names a b c d f3
0111 1
1001 1
1010 1
1011 1
1111 1

.exdc
.inputs a b c d
.outputs f1 f2 f3

.names a b c d f1
0001 1
0010 1
0101 1
0111 1

.names a b c d f2
0100 1
1100 1

.names a b c d f3
0101 1
1100 1

.end
```

2.2.2 Esercizio 1

La minimizzazione automatica restituisce lo stesso risultato di quella eseguita manualmente attraverso il comando *full_simplify*. Effettuando due diversi tipi di mapping, i valori di area e ritardo variano secondo le esigenze esposte.

```
sis> read_library "C:\Users\Saverio\Desktop\ASE\sis\mcnc.genlib"
sis> read_blif C:\Users\Saverio\Desktop\ASE\sis\esercizio1.blif
sis> map -W -n 1 -s
# of outputs:          1
total gate area:       18.00
maximum arrival time: (6.60,6.60)
maximum po slack:     (-6.60,-6.60)
minimum po slack:     (-6.60,-6.60)
total neg slack:      (-6.60,-6.60)
# of failing outputs:  1
sis> map -W -n 0 -s
>>> before removing serial inverters <<<
# of outputs:          1
total gate area:       12.00
maximum arrival time: (9.00,9.00)
maximum po slack:     (-9.00,-9.00)
minimum po slack:     (-9.00,-9.00)
total neg slack:      (-9.00,-9.00)
# of failing outputs:  1
>>> before removing parallel inverters <<<
# of outputs:          1
total gate area:       12.00
maximum arrival time: (9.00,9.00)
maximum po slack:     (-9.00,-9.00)
minimum po slack:     (-9.00,-9.00)
total neg slack:      (-9.00,-9.00)
# of failing outputs:  1
# of outputs:          1
total gate area:       12.00
maximum arrival time: (9.00,9.00)
maximum po slack:     (-9.00,-9.00)
minimum po slack:     (-9.00,-9.00)
total neg slack:      (-9.00,-9.00)
# of failing outputs:  1
```

Ciò è possibile perchè la libreria è fornita di diversi componenti, ma se la riduciamo ad un set funzionalmente completo, **And** e **Not** ad esempio, i risultati non variano.

```
sis> read_library "C:\Users\Saverio\Desktop\ASE\sis\mcnc - Copia.genlib"
sis> read_blif C:\Users\Saverio\Desktop\ASE\sis\esercizio1.blif
sis> map -W -n 1 -s
# of outputs:          1
total gate area:       29.00
maximum arrival time: (17.60,17.60)
maximum po slack:     (-17.60,-17.60)
minimum po slack:     (-17.60,-17.60)
total neg slack:      (-17.60,-17.60)
# of failing outputs:  1
sis> map -W -n 0 -s
>>> before removing serial inverters <<<
# of outputs:          1
total gate area:       29.00
maximum arrival time: (17.60,17.60)
maximum po slack:     (-17.60,-17.60)
minimum po slack:     (-17.60,-17.60)
total neg slack:      (-17.60,-17.60)
# of failing outputs:  1
>>> before removing parallel inverters <<<
# of outputs:          1
total gate area:       29.00
maximum arrival time: (17.60,17.60)
maximum po slack:     (-17.60,-17.60)
minimum po slack:     (-17.60,-17.60)
total neg slack:      (-17.60,-17.60)
# of failing outputs:  1
# of outputs:          1
total gate area:       29.00
maximum arrival time: (17.60,17.60)
maximum po slack:     (-17.60,-17.60)
minimum po slack:     (-17.60,-17.60)
total neg slack:      (-17.60,-17.60)
# of failing outputs:  1
```

2.2.3 Esercizio 2

Vogliamo mettere in evidenza in questo esempio come la tecnologia impatta sulle prestazioni di una rete combinatoria.

```
sis> read_blif C:\Users\Saverio\Desktop\ASE\sis\esercizio2.blif
sis> espresso
sis> write_eqn
INORDER = a b c d;
OUTORDER = f;
[920] = !b*!c;
[921] = b*c*d;
[922] = !b*!d;
[923] = ![920]*![921]*![922];
f = ![923];
```

Utilizzando *espresso* ritroviamo la medesima minimizzazione attuata manualmente col metodo di Quine Mc Cluskey. I costi della rete, in termini di numero di litterali Cl, numero di ingressi Ci, numero di porte Cp , sono $Cl = 7$; $Ci = 10$; $Cp = 4$. Vediamo come si comporta invece la rete con l' utilizzo della libreria *mcnc*. Ottimizzando l'area occupata:

```
sis> read_blif C:\Users\Saverio\Desktop\ASE\sis\esercizio2.blif
sis> map -W -n 0 -s
>>> before removing serial inverters <<<
# of outputs:          1
total gate area:       14.00
maximum arrival time: (9.30,9.30)
maximum po slack:     (-9.30,-9.30)
minimum po slack:     (-9.30,-9.30)
total neg slack:      (-9.30,-9.30)
# of failing outputs:  1
>>> before removing parallel inverters <<<
# of outputs:          1
total gate area:       14.00
maximum arrival time: (9.30,9.30)
maximum po slack:     (-9.30,-9.30)
minimum po slack:     (-9.30,-9.30)
total neg slack:      (-9.30,-9.30)
# of failing outputs:  1
# of outputs:          1
total gate area:       14.00
maximum arrival time: (9.30,9.30)
maximum po slack:     (-9.30,-9.30)
minimum po slack:     (-9.30,-9.30)
total neg slack:      (-9.30,-9.30)
# of failing outputs:  1
sis> write_eqn
INORDER = a b c d;
OUTORDER = f;
[975] = !d;
[950] = !a*!b*!c*![975];
[1000] = !c + !b + !d;
[979] = !b*!d + ![1000];
f = [979] + [950];

Don't care:
INORDER = a b c d;
OUTORDER = f;
f = a*!b*!c*d + !a*b*!c*d + a*!b*c*d;
```

In questo caso i costi sono $Cl = 9 - Ci = 13 - Cp = 5$. I costi aumentano perchè SIS cerca di usare porte che occupano meno spazio, non curandosi dei tempi di commutazione e di propagazione dell'informazione lungo il path critico.

Ottimizzando il ritardo:

```
sis> map -W -n 1 -s
# of outputs:          1
total gate area:       20.00
maximum arrival time: (5.00,5.00)
maximum po slack:     (-5.00,-5.00)
minimum po slack:     (-5.00,-5.00)
total neg slack:      (-5.00,-5.00)
# of failing outputs:  1
sis> write_eqn
INORDER = a b c d;
OUTORDER = f;
[1063] = !a;
[1056] = !b;
[1058] = !c;
[1077] = !d + ![1058] + ![1056] + ![1063];
[1079] = !d + !b + !c;
[1059] = !d;
[1083] = ![1059] + ![1056];
f = ![1083] + ![1079] + ![1077];

Don't care:
INORDER = a b c d;
OUTORDER = f;
f = a*!b*!c*d + !a*b*!c*d + a*!b*c*d;
```

Rilassando il vincolo di area ottieniamo prestazioni vicine a quelle di *espresso:* $Cl = 9$; $Ci = 12$; $Cp = 4$. Ciò è possibile perchè la libreria ha a disposizione un adeguato numero di componenti da utilizzare, situazione non sempre veritiera.

2.2.4 Esercizio 4

Proviamo ad applicare il rugged-script ad un rete multi-uscita:

```
sis> read_blif "C:\Users\Saverio\Desktop\ASE\Template Elaborato\esercizio01\listing\simulazione\esercizi
o4.blif"
sis> source -x script.rugged
sweep; eliminate -1; simplify -m nocomp; eliminate -1
sweep; eliminate 5; simplify -m nocomp
resub -a;fx; resub -a
sweep; eliminate -1
sweep; full_simplify -m nocomp

sis> write_eqn
INORDER = a b c d;
OUTORDER = f1 f2 f3;
f1 = a*b*c + !a*![321] + !b*!c;
f2 = d*!f1*f3 + a*!c*d + !a*!b*!c;
f3 = a*!b*[321] + b*c*d;
[321] = d + c;

Don't care:
INORDER = a b c d;
OUTORDER = f1 f2 f3;
f1 = !a*!b*c*!d + !a*!b*!c*d + !a*b*!c*d + !a*b*c*d;
f2 = !a*b*!c*!d + a*b*!c*!d;
f3 = a*b*!c*!d + !a*b*!c*d;

sis> print_stats
esercizio4  pi= 4  po= 3  node=  4  latch= 0  lits(sop)=  24  lits(ff)=  23
```

La minimizzazione sub-ottima ottenuta con questo metodo euristico è simile a quella ricavata da Mc Cluskey multi-funzione, a conferma del fatto che il rugged-script si rivela una buona sequenza per le trasformazioni da applicare alla rete. Rimappiamo la nostra rete su un fpga avente componenti elementari a quattro ingressi.

```
sis> xl_imp -n 4
sis> xl_cover -n 4
sis> write_eqn
INORDER = a b c d;
OUTORDER = f1 f2 f3;
f1 = !a*!c*!d + a*b*c + !b*!c;
f2 = a*!b*c*d + !a*b*c*d + a*!c*d + !a*!b*!c;
f3 = b*c*d + a*!b*d + a*!b*c;

Don't care:
INORDER = a b c d;
OUTORDER = f1 f2 f3;
f1 = !a*!b*c*!d + !a*!b*!c*d + !a*b*!c*d + !a*b*c*d;
f2 = !a*b*!c*!d + a*b*!c*!d;
f3 = a*b*!c*!d + !a*b*!c*d;

sis> print_stats
esercizio4  pi= 4  po= 3  node=  3  latch= 0  lits(sop)=  31  lits(ff)=  26
```

Il mapping è stato possibile perchè il fan-in dei nodi della rete non è maggiore del numero di input di una cella. Al seguito di questa verifica, è stata effettuata la minimizzazione del numero di nodi della rete e l'associazione di questi alle celle Xilinx. Il numero di nodi utilizzati per sintetizzare la rete è diminuito e il numero di letterali è aumentato di 7 unità: un nodo è stato eliminato sostituendo la sua espressione nei nodi a valle. Ciò ha consentito un risparmio di area, in termini

di celle Xilinx, ed eventualmente anche di tempo, grazie ad una dipendenza più diretta dei nodi a valle dagli input primari.

Capitolo 3

Latch/Flip Flop

3.1 Traccia

Sviluppare i circuiti illustrati nel documento sui flip-flop. Eseguire per ciascun esercizio una simulazione comportamentale e post-sintesi, illustrando i passaggi salienti.

3.2 Latch RS

3.2.1 Schematico

Si è scelto di implementare il latch RS in logica 1-attiva con due porte NOR, questo implica che, quando S=1 e R=0, l'uscita Q è alta, che lo stato neutro degli ingressi è S=R=0 e quello non ammesso è S=R=1, il quale, seguito dal passaggio allo stato neutro, produce un'uscita non stabile e soggetta ad un transitorio, che può portare ad alee e quindi a corse. Dualmente, grazie alla simmetria del circuito, è possibile cambiare la logica in 0-attiva utilizzando due porte NAND.

3.2.2 Codice

3.2.2.1 RS_Latch

Questo componente è stato realizzato con un'architettura di tipo Structural, connettendo due componenti *nor_gate*, che non fanno altro che la NOR dei due ingressi. Le uscite di ogni porta nor viene poi retroazionata all'ingresso dell'altra porta.
Il componente in questione è osservabile a questo link: RS_Latch

3.2.3 Simulazione

3.2.3.1 Behavioral

In questo tipo di simulazione, viene modellato solo il comportamento funzionale del sistema, infatti le porte sono considerate ideali e quindi senza ritardo; per questo motivo quando provo ad utilizzare la configurazione degli ingressi non ammessa R=S=1 (che viola il vincolo logico $R \cdot S = 0$ e quindi quello $Q = not(\neg Q)$) e poi nello stesso istante (80 ns) le abbasso entrambe , questo genera degli eventi oscillatori che producono cicli di delta cycle. Questi però non riusciamo ad osservarli in

simulazione poichè il tempo di simulazione non avanza e viene generato un errore che indica che si è raggiunto il limite di iterazioni, dovuto al fatto che il sistema non riesce a raggiungere uno stato stabile. Queste oscillazioni invece sono visibili nella simulazione Post-Map, in cui vengono introdotti i ritardi dei componenti di libreria ma non quelli relativi alle loro connessioni.

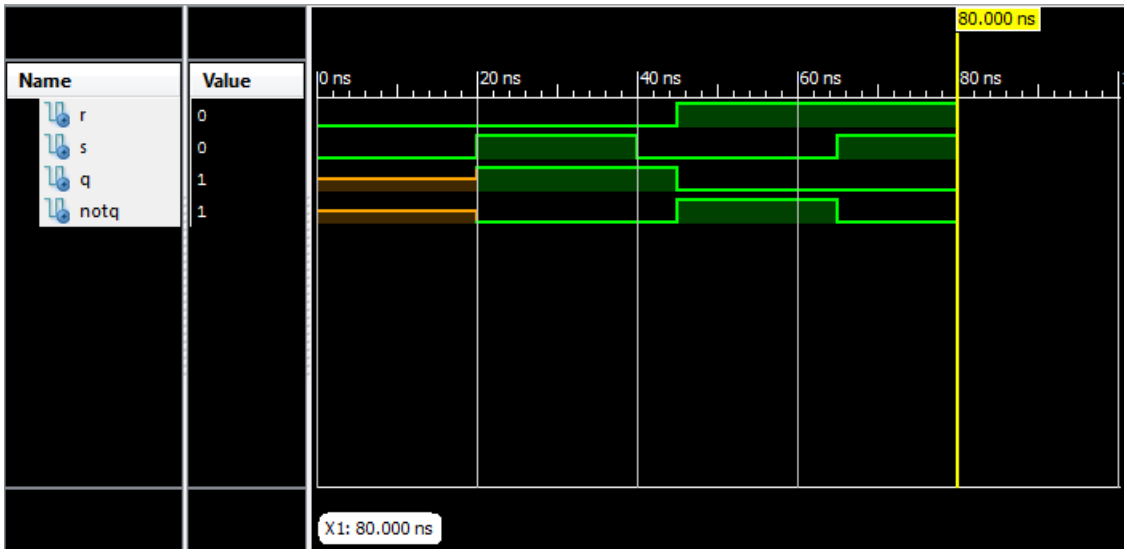


Figura 3.1: Simulazione del latch RS Behavioral

3.2.3.2 Post-Sintesi

La simulazione di Figura 3.2 rappresenta l'evoluzione del sistema in seguito all'operazione di Place & Route che, dopo aver utilizzato dei componenti della libreria Xilinx con i relativi ritardi dovuti ai tempi di commutazione, collega tali componenti, tenendo in considerazione i ritardi delle connessioni relative al routing effettuato. Proprio grazie a questi ritardi la simulazione continua e non si blocca come quella precedente quando gli ingressi passano dal valore R=S=1 a R=S=0, ma vengono introdotte delle asimmetrie nel circuito che portano le uscite ad assumere dei valori che rispettano i vincoli logici, nel nostro caso sono Q=1 e ¬Q=0 a 86 ns. Però è importante sottolineare che questi valori prodotti in uscita non sono deterministici, perchè generati da corse e quindi è preferibile non ottenerli rispettando il vincolo che proibisce l'utilizzo di entrambi gli ingressi alti.

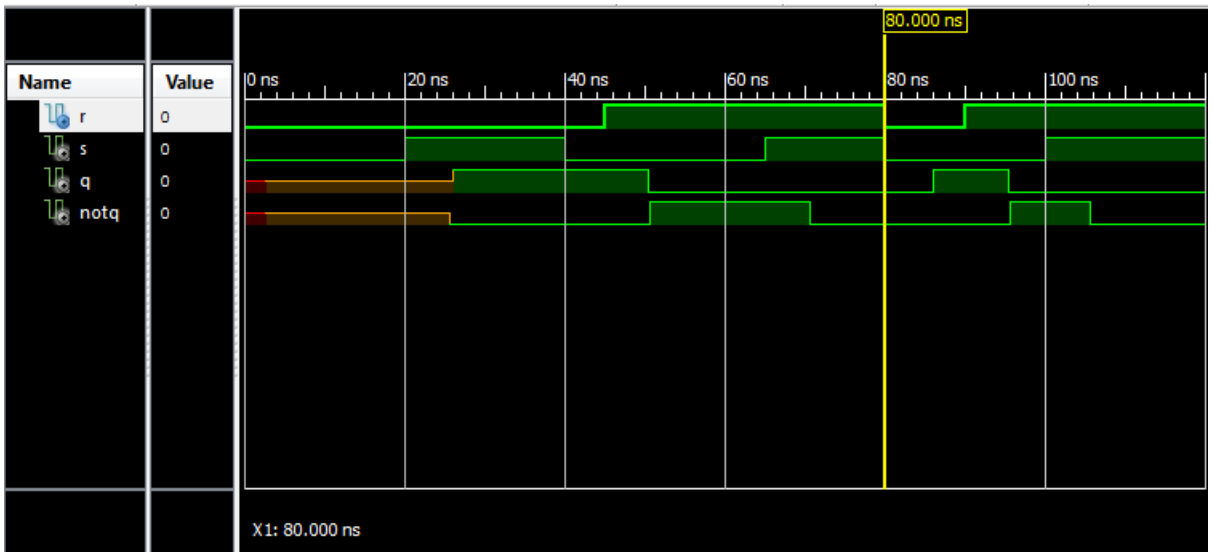


Figura 3.2: Simulazione del latch RS Post-Route

3.3 Latch RS abilitato

3.3.1 Schematico

Questo tipo di latch deriva dal latch RS precedente, al quale si antepone una rete combinatoria formata da due porte AND, i cui ingressi sono R ed S e in più un segnale di abilitazione, che può essere un clock, ma che in nessun caso rende sincrona la rete.

3.3.2 Codice

3.3.2.1 RS_Latch_Clocked

Tale componente è stato realizzato con un'architettura Structural, connettendo, tramite i due segnali *r_clocked* ed *s_clocked*, le uscite di due porte AND, realizzate con il componente *and_gate*, agli ingressi R ed S del componente *rs_latch* visto precedentemente (componente RS_Latch). Ovviamente in ingresso alle porte AND ci saranno i segnali di set e reset, nonché un clock che funge da segnale di abilitazione.
Il componente in questione è osservabile a questo link: RS_Latch_Clocked

3.3.3 Simulazione

3.3.3.1 Behavioral

In Figura 3.3 è rappresentata la simulazione behavioral del latch RS abilitato, che si comporta come un normale latch RS quando il clock è alto, altrimenti mantiene lo stato precedente. Inoltre si osserva, come nel caso del latch RS, che la simulazione si ferma (in questo caso a 50 ns) e viene generato un errore che indica che si è raggiunto il limite di iterazioni. Tutto questo è dovuto al fatto che, avendo utilizzanto nel testbench, a 45 ns, la configurazione degli ingressi non ammessa R=S=1, vengono generate delle oscillazioni non osservabili in questo tipo di simulazione e che

probabilmente sono causate dal successivo abbassamento del segnale di clock alla soglia dei 50 ns, che porterebbe il sistema a conservare lo stato precedente forzando i valori di ingresso R=S=0.

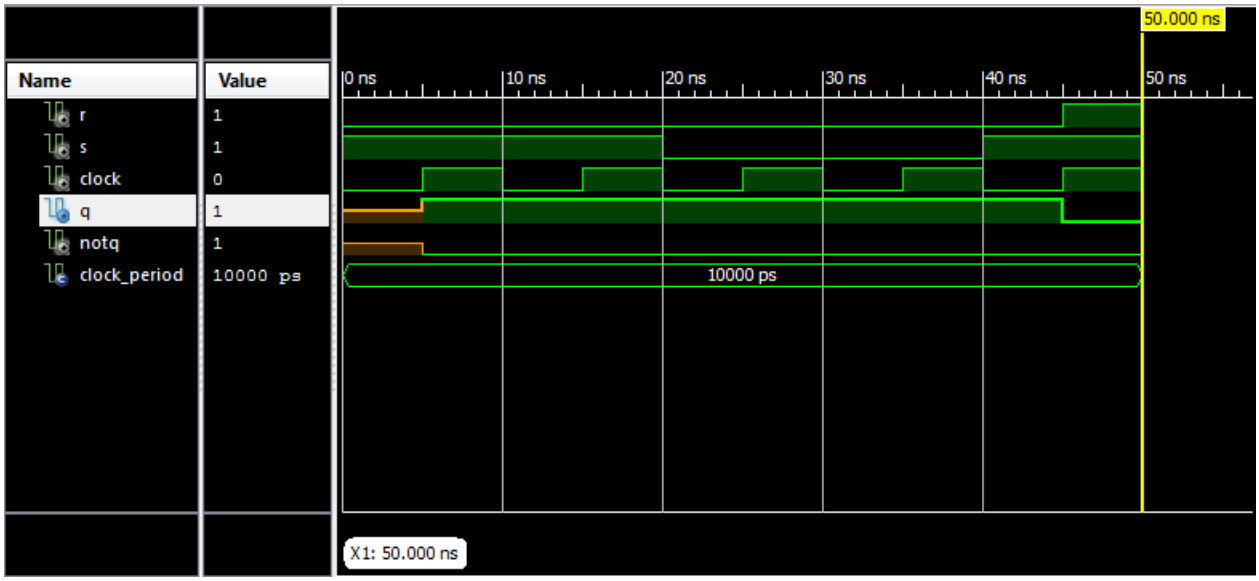


Figura 3.3: Simulazione del latch RS abilitato Behavioral

3.3.3.2 Post-Sintesi

In Figura 3.4 è mostrata la simulazione Post-Route del sistema. Come si è verificato per il lactch RS, anche qui, utilizzando i componenti della libreria Xilinx e considerando i loro ritardi e quelli delle connessioni, si osserva che la simulazione continua e non si arresta, grazie al fatto che vengono forzati dei valori in uscita (Q=1 e ¬Q=0 a 55ns) dovuti all'asimmetria introdotta dopo l'operazione di Place & Route.

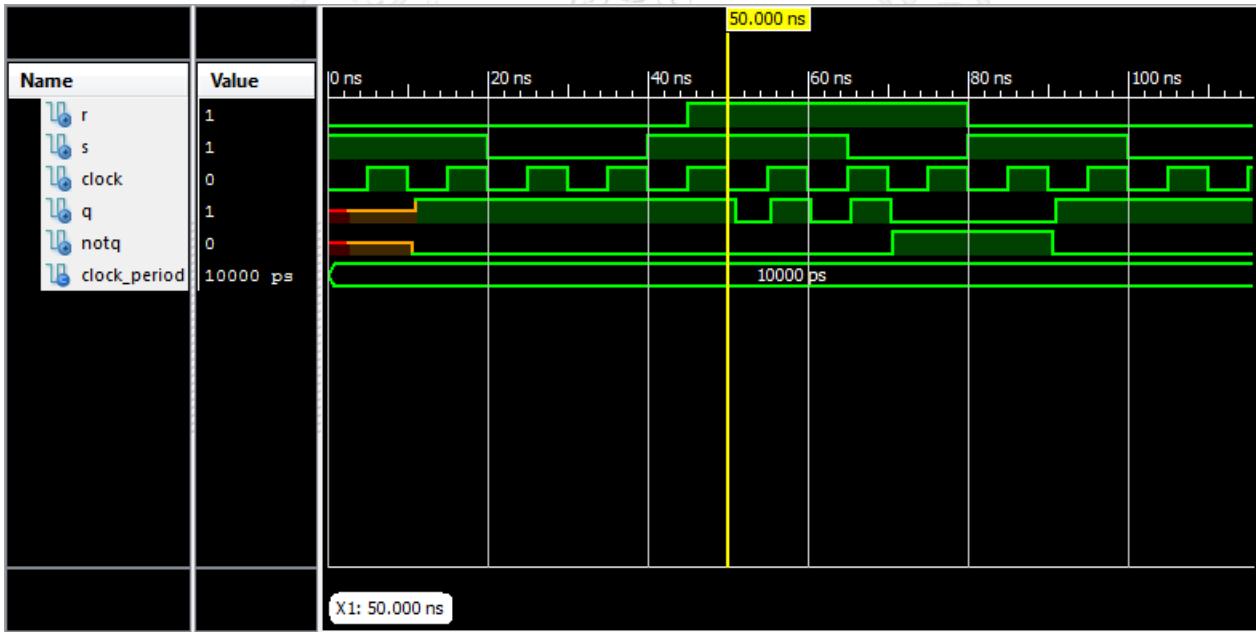


Figura 3.4: Simulazione del latch RS abilitato Post-Route

3.4 Latch D abilitato

3.4.1 Schematico

Il latch D abilitato può essere realizzato a partire da un latch RS abilitato i cui ingressi R ed S corrispondono al valore D in ingresso rispettivamente una volta negato tramite una porta NOT e una volta no. Questo latch permette di avere in uscita Q il valore D in ingresso, ma ritardato di un certo Δ.

3.4.2 Codice

3.4.2.1 D_Latch_Clocked

Questo componente è stato creato come un'architettura Structural a partire dal componente *rs_latch_clocked* visto in precedenza (componente RS_Latch_Clocked). L'ingresso D viene collegato all'ingresso S del latch rs abilitato e viene utilizzato un segnale *notd* per negare D e collegarlo all'ingresso R.
Il componente in questione è osservabile a questo link: D_latch_Clocked

3.4.3 Simulazione

3.4.3.1 Behavioral

Come si nota dalla simulazione behavioral di Figura 3.5 quando D e il clock sono alti il segnale in uscita Q è alto, altrimenti o viene mantenuto lo stato precedente quando il clock è basso oppure quando D è basso e il clock è altro, sarà l'uscita $\neg$Q ad essere alta. Anche in questo caso come in quelli precedenti, la simulazione si arresta a 40 ns a causa di un errore che indica il raggiungimento del limite di iterazioni dovuto alla generazione di cicli di delta cycle, che non permettono di giungere ad uno stato stabile.

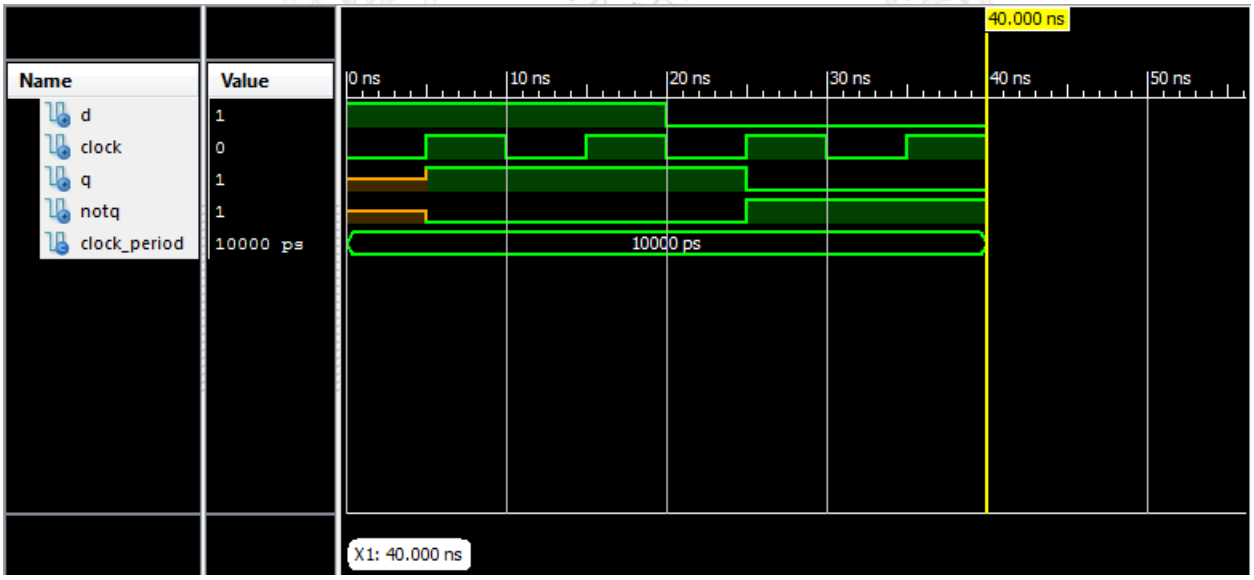


Figura 3.5: Simulazione del latch D abilitato Behavioral

3.4.3.2 Post-Sintesi

In Figura 3.6 è rappresentata la simulazione Post-Route del componente. Grazie all'introduzione dei ritardi dei componenti della libreria Xilinx e a quelli relatvi ai collegamenti tra gli stessi, la simulazione è in grado di continuare senza bloccarsi; ovviamente questo comporta un ritardo nella generazione delle uscite che commutano un po' dopo il sollevamento del segnale di clock. Ad aumentare ulteriormente il ritardo è l'aggiunta della porta NOT, che aumenta ulteriormente il tempo di commutazione rispetto al latch RS abilitato.

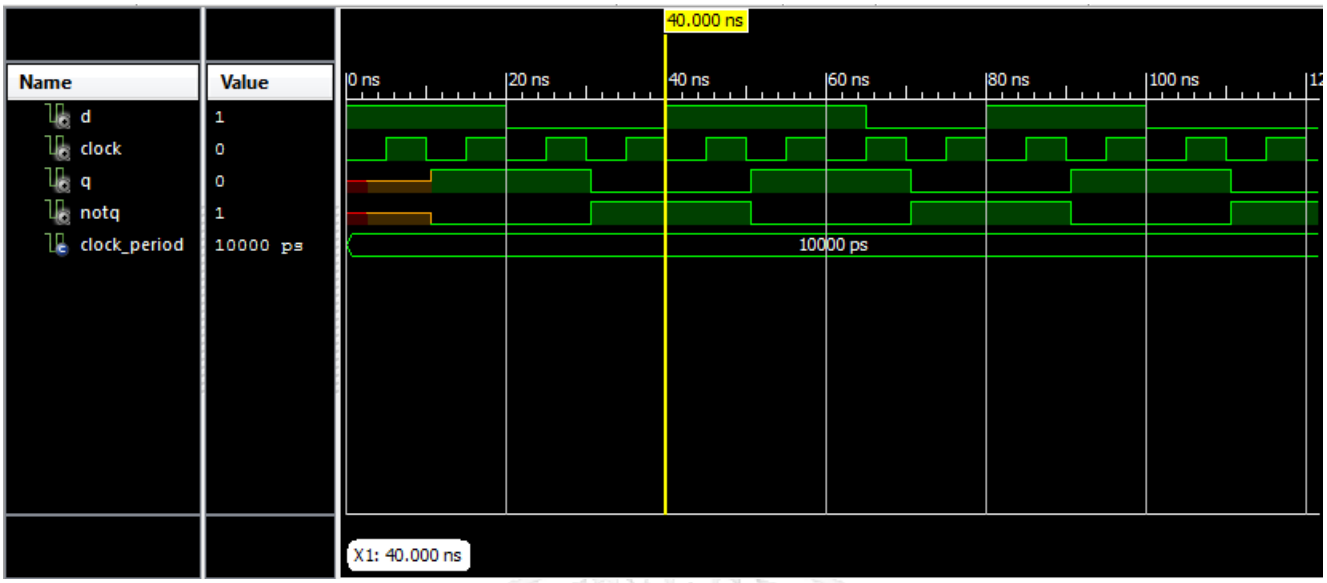


Figura 3.6: Simulazione del latch D abilitato Post-Route

3.5 Latch T

3.5.1 Schematico

Il latch T può essere realizzato similmente al latch RS abilitato, solo che, in ingresso alle porte AND, vanno le uscite retroazionate al posto degli ingressi di reset e set e il segnale T che, quando è abilitato, non fa altro che commutare l'uscita Q.

3.5.2 Codice

3.5.2.1 T_Latch

L'approccio utilizzato per la realizzazione di questo tipo di latch è quello strutturale, in base al quale, a partire dal componente RS_Latch visto in precedenza, sono state aggiunte due porte AND, in ingresso alle quali vanno il segnale T e le uscite Q e ¬Q del latch RS retroazionate. Per fare questo sono stati utilizzati quattro segnali: due (*retro_q* e *retro_notq*) utilizzati per le uscite del latch e altri due (*retro_q_delayed* e *retro_notq_delayed*) utilizzati per ritardare, tramite la parola chiave after, tali uscite in modo che possano essere utilizzate in ingresso alle porte AND, questo perchè altrimenti le uscite non riescono a commutare.
Il componente in questione è osservabile a questo link: T_Latch

3.5.3 Simulazione

3.5.3.1 Behavioral

In figura 3.7 si osserva la simulazione Behavioral del latch_t. Così come si nota quando T è alto possono avvenire una o più commutazione delle uscite in base alla sua durata, infatti T dovrebbe essere alto per un tempo sufficientemente grande affinchè avvenga una commutazione ma sufficientemente piccolo affinchè non ne avvengano altre. Si noti inoltre, che quando T è basso, il sistema mantine lo stato precedente.

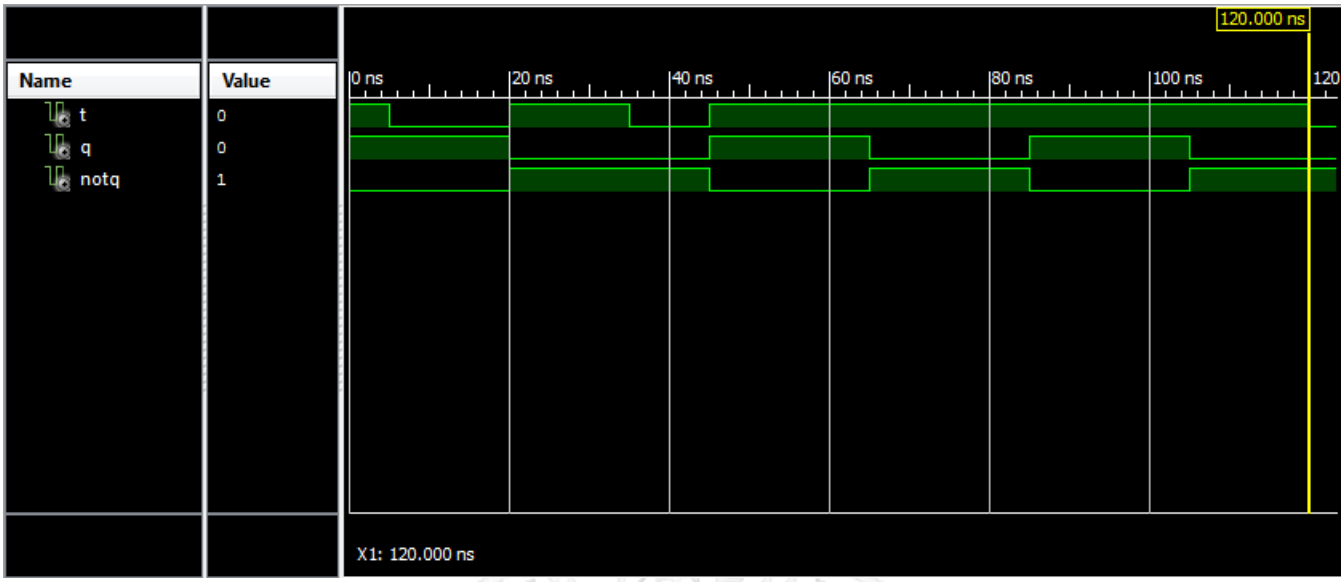


Figura 3.7: Simulazione del latch T Behavioral

3.5.3.2 Post-Sintesi

Nella simulazione Post-Sintesi invece, dato che la parola chiave after, usata precedentemente per permettere la commutazione delle uscite, non è sintetizzabile, allora le uscite non cambiano stato e permangono per tutta la durata della simulazione in uno stato "uninitialized".

3.6 Latch JK abilitato

3.6.1 Schematico

Il latch JK è strutturato in maniera simile al latch RS abilitato, ciò che cambia è che le porte AND prendono in ingresso le uscite retroazionate, così come avviene per il latch T, il clock e due segnali J e K. In base alla configurazione di questi due segnali cambia il funzionamento del latch, poiché se K=J=1, esso si comporta come un latch T, altrimenti si comporta come un latch RS in cui R=K e S=J.

3.6.2 Codice

3.6.2.1 JK_Latch

Questo tipo di componente è stato realizzato con un'architettura Structural, in cui si antepongono ad un latch RS (componente RS_Latch), due porte AND a tre ingressi, cioè il clock, K o J, e una delle due uscite retroazionata. Questo avviene tramite l'utilizzo di quattro segnali, ovvero *q_feedback* e *notq_feedback*, utilizzati per le uscite del latch RS, e *q_feedback_delayed* e *notq_feedback_delayed*, utilizzati per ritardare tali segnali, in modo che possano essere retroazionati in ingresso alle porte AND e quindi permettere la commutazione. Sono presenti inoltre due ingressi di *clear* e *preset*, messi in OR con le uscite delle porte AND, in modo da inizializzare il dispositivo, tramite i segnali *in_latch_r* e *in_latch_s*, messi in ingresso al latch RS.
Il componente in questione è osservabile a questo link: JK_Latch

3.6.3 Simulazione

3.6.3.1 Behavioral

In Figura 3.8 è rappresentata la simulazione Behavioral del funzionamento del latch JK abilitato. Si nota che, così come ci aspettiamo, quando J è alto, Q è alto, mentre quando K è alto, ¬Q è alto; quando invece sia J che K sono alti, il sistema funziona come un latch T e quindi le uscite commutano un certo numero di volte, che dipende dalla durata dello stato J=K=1. Ovviamente quando sia J che K sono bassi oppure il clock è basso, il sistema permane nello stato precedente.

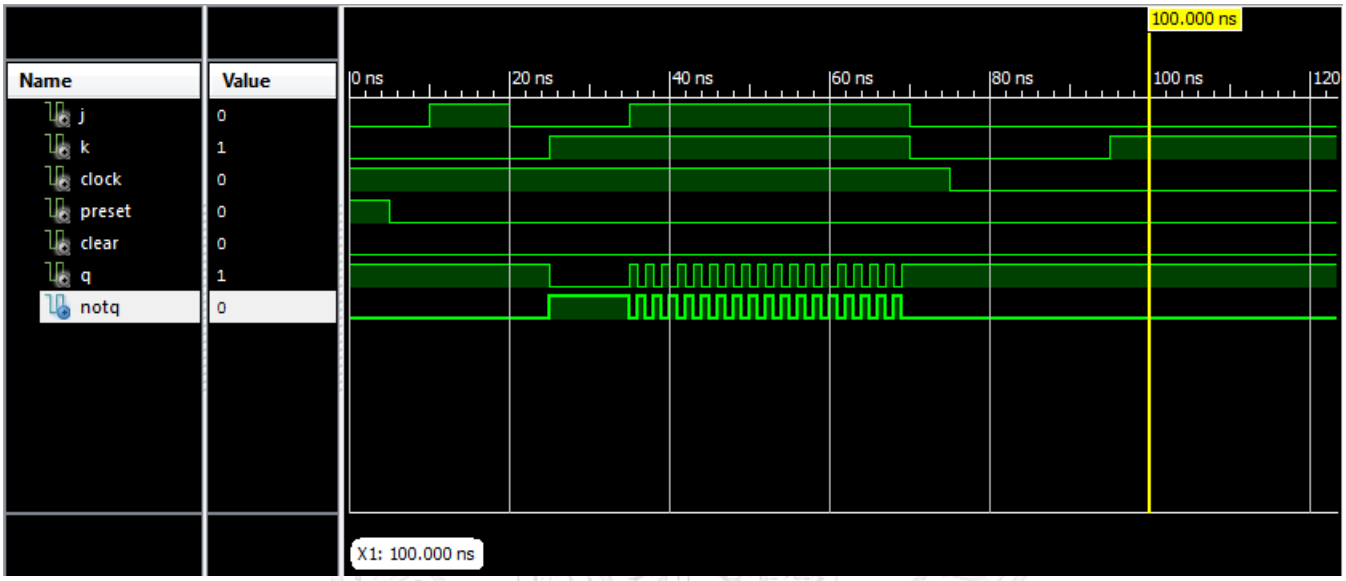


Figura 3.8: Simulazione del latch JK abilitato Behavioral

3.6.3.2 Post-Sintesi

Nella simulazione Post-Route di Figura 3.9 vengono utilizzati i componenti della libreria Xilinx con i relativi ritardi e quelli dovuti ai collegamenti tra gli stessi e quindi notiamo un certo delay nella commutazione delle uscite rispetto al caso Behavioral. Si nota inoltre che in seguito alla

fine della condizione in cui J=K=1, anzichè permanere in uno stato, le uscite commutano sempre, probabilmente a causa di alcune asimmetrie introdotte nel circuito dalla fase di sintesi.

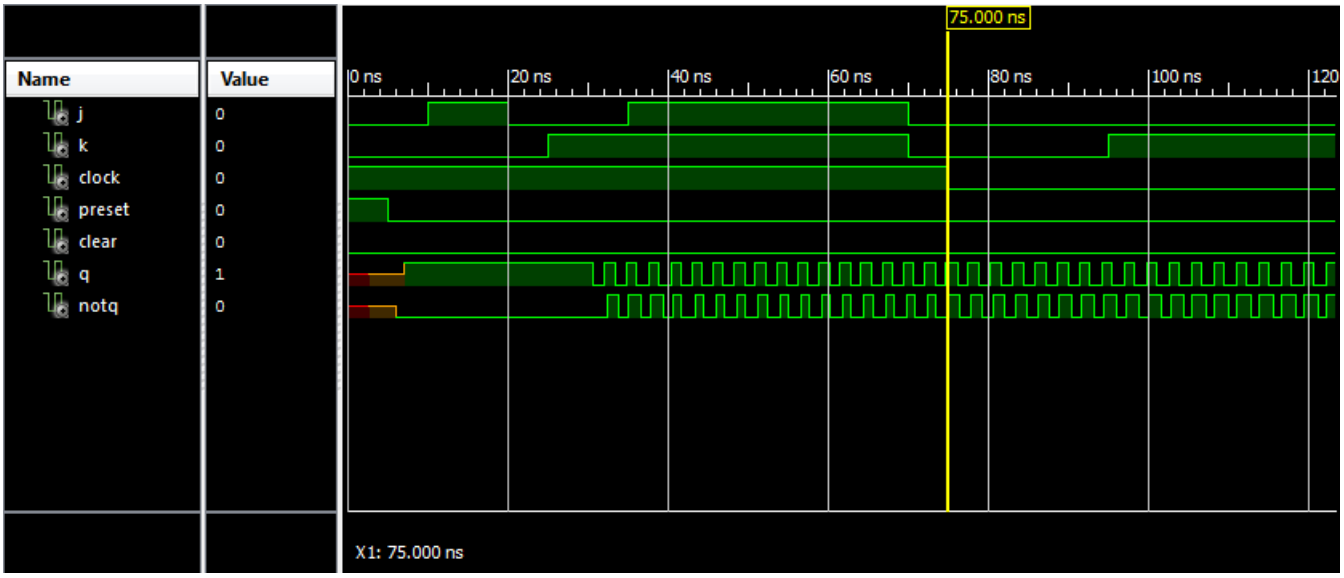


Figura 3.9: Simulazione del latch JK abilitato Post-Route

3.7 Flip-Flop D Edge Triggered

3.7.1 Schematico

Un Flip-Flop D Edge Triggered che commuta sul fronte di discesa del clock può essere realizzato, così come si evince dalla Figura 3.10, attraverso 6 porte NOR opportunamente interconnesse. Quando il clock è alto, l'uscita delle porte G2 e G3 viene forzata a 0, quindi il latch 3 mantiene il proprio stato delle uscite e il latch 1 e il latch 2 seguono il valore D e ¬D rispettivamente. All'atto della transizione del clock dal valore alto a quello basso, i valori di D e ¬D vengono propagati agli ingressi del latch 3 e quindi in uscita. Considerando τ il ritardo di ogni porta, la rete impiega 5τ per commutare. Si noti inoltre, un ulteriore ingresso alla porta G2, che non è altro che l'uscita della porta G1 retroazionata; questo elemento, può sembrare ridondante, in realtà evita il presentarsi di un'alea che, all'atto della transizione del clock 1->0, causerebbe un valore inatteso all'uscita della porta G2, dovuto al ritardo di 3τ per propagare l'ingresso R1a, infatti l'uscita della porta G1 ha lo stesso valore dell'uscita della porta G3, solo che è possibile ottenerla con un ritardo pari solo a τ.

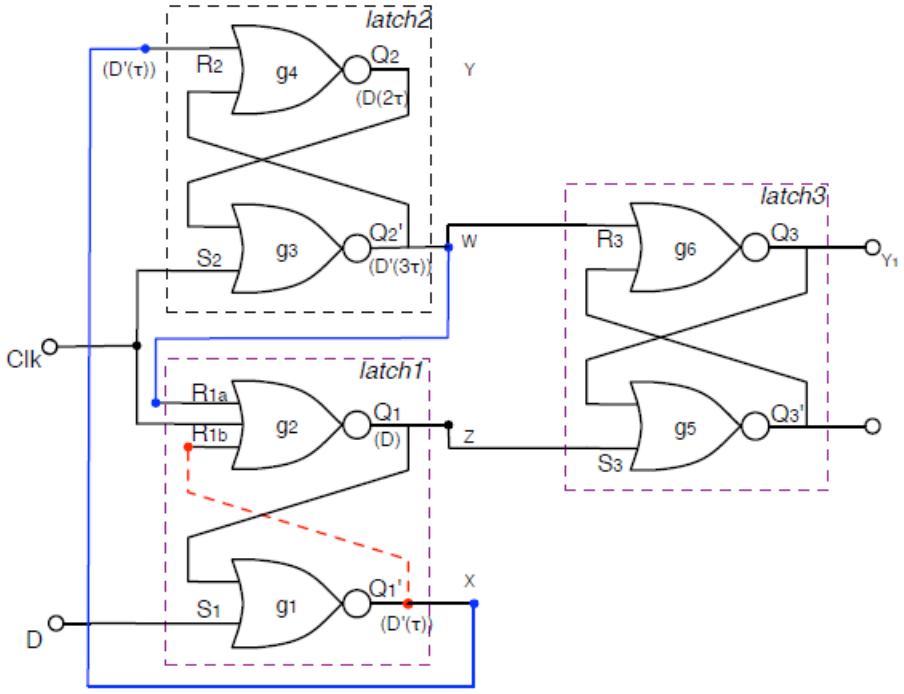


Figura 3.10: Schematico Flip-Flop D Edge Triggered

3.7.2 Codice

3.7.2.1 FlipFlop_D_edge

Il Flip-Flop D Edge Triggered è stato realizzato con un'architettura Structural componendo 6 porte NOR, opportunamente connesse con dei segnali da G1 a G6 che rappresentano le uscite delle omonime porte. Si noti che per settare l'uscita della porta NOR G2, che ha tre ingressi, è stata applicata una porta NOT al risultato dell'OR tra l'uscita della porta G3, il clock e l'uscita della porta G1, utilizzata per eliminare l'alea.
Il componente in questione è osservabile a questo link: FlipFlop_D_Edge

3.7.3 Simulazione

3.7.3.1 Behavioral

Nella simulazione Behavioral di Figura 3.11 si osserva il funzionamento del Flip Flop D. Come si nota la commutazione delle uscite avviene in corrispondenza del fronte di discesa del clock in maniera istantanea, non avendo usato a questo livello i componenti della libreria Xilinx.

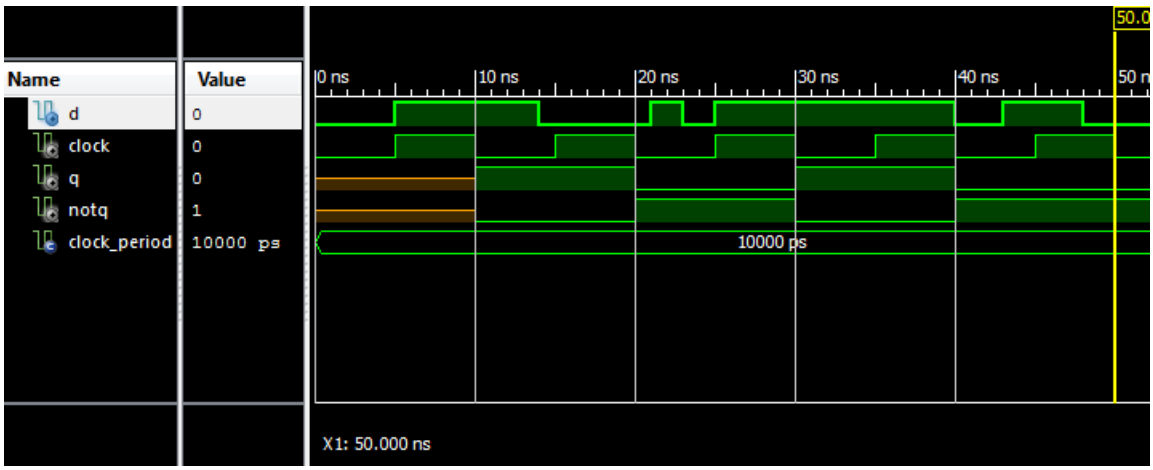


Figura 3.11: Simulazione del Flip-Flop D Edge Triggered Behavioral

3.7.3.2 Post-Sintesi

In Figura 3.14 è rappresetata una simulazione Post-Route del Flip-Flop D. Come c'era da aspettarsi, tenendo in considerazione i ritardi dei componenti utilizzati e quelli relativi al routing, la commutazione delle uscite non avviene istantanemente rispetto al fronte di discesa del clock, ma con un ritardo che è circa 6 ns, che quindi è in linea con il ritardo di 5τ predetto precedentemente. Si noti inoltre l'assenza di alee grazie all'aggiunta del collegamento dell'uscita della porta G1 all'ingresso della porta G2.

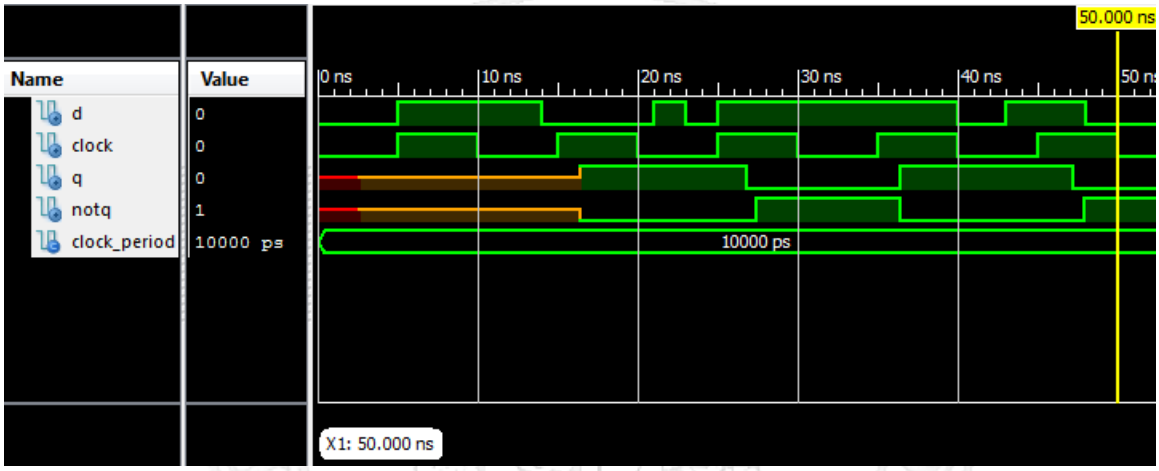


Figura 3.12: Simulazione del Flip-Flop D Edge Triggered Post-Route

3.8 Flip-Flop RS Master-Slave

3.8.1 Schematico

Un Flip-Flop RS Master-Slave viene realizzato collegando in cascata due latch RS abilitati da un segnale di clock nel caso del primo latch (master) è affermato, mentre nel caso del secondo (slave) è negato. Questo comporta che i due latch non funzionano contemporaneamente ma quando uno è attivo l'altro è disattivo e viceversa.

3.8.2 Codice

3.8.2.1 FlipFlop_RS_MS

Il componente *flipflop_rs_ms* è stato realizzato con un approccio Structural, poichè sono stati utilizzati due componenti rs_latch_clocked visti in precedeza (componente RS_Latch_Clocked), collegati tramite i due segnali *q_master* e *notq_master* che vanno in ingresso rispettivamente al set e al reset del latch slave. Infine è stata utilizzata una porta NOT per negare il segnale di clock che è stato poi collegato al secondo latch.
Il componente in questione è osservabile a questo link: FlipFlop_RS_MS

3.8.3 Simulazione

3.8.3.1 Behavioral

In Figura 3.13 viene mostrata la simulazione Behavioral quindi il funzionamento di questo tipo di circuito. Come si nota, quello che di fatto realizza è un flip-flop RS pilotato sul fronte di discesa del clock, in corrispondeza del quale avviene la commutazione delle uscite Q e ¬Q. In questo caso la simulazione continua fino alla fine poichè non è stata utilizzata la configurazione R=S=1 non ammessa e quindi non si sono generati eventi oscillatori.

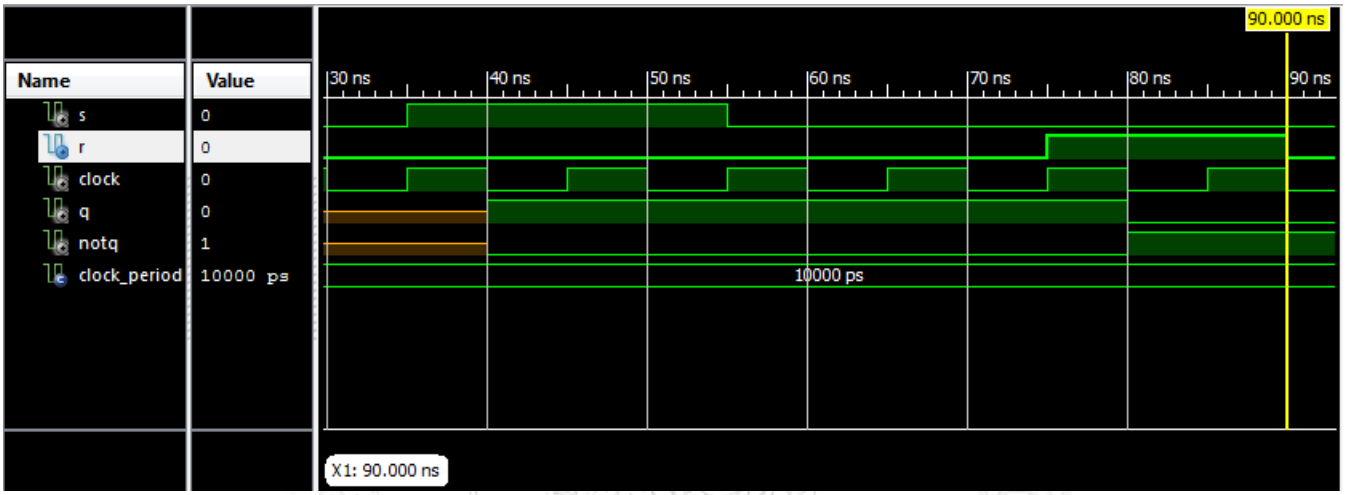


Figura 3.13: Simulazione del Flip-Flop RS Master-Slave Behavioral

3.8.3.2 Post-Sintesi

Nella simulazione Post-Route di Figura 3.14, come nei casi precedenti notiamo che la commutazione delle uscite avviene con circa 5-6 ns di ritardo, dovuto all'utilizzo dei componenti della libreria Xilinx e ai ritardi dei loro collegamenti. Fortunatamente, in questo caso, il ritardo della porta NOT sembra non pregiudicare il funzionamento del sistema, infatti il master e lo slave funzionano alternativamente, cosa che potrebbe non accadere se il ritardo della porta NOT fosse troppo grande e quindi non si riuscisse a determinare l'uscita. Questo però è un caso limite poiché di solito il ritardo della porta NOT è un τ piccolissimo.

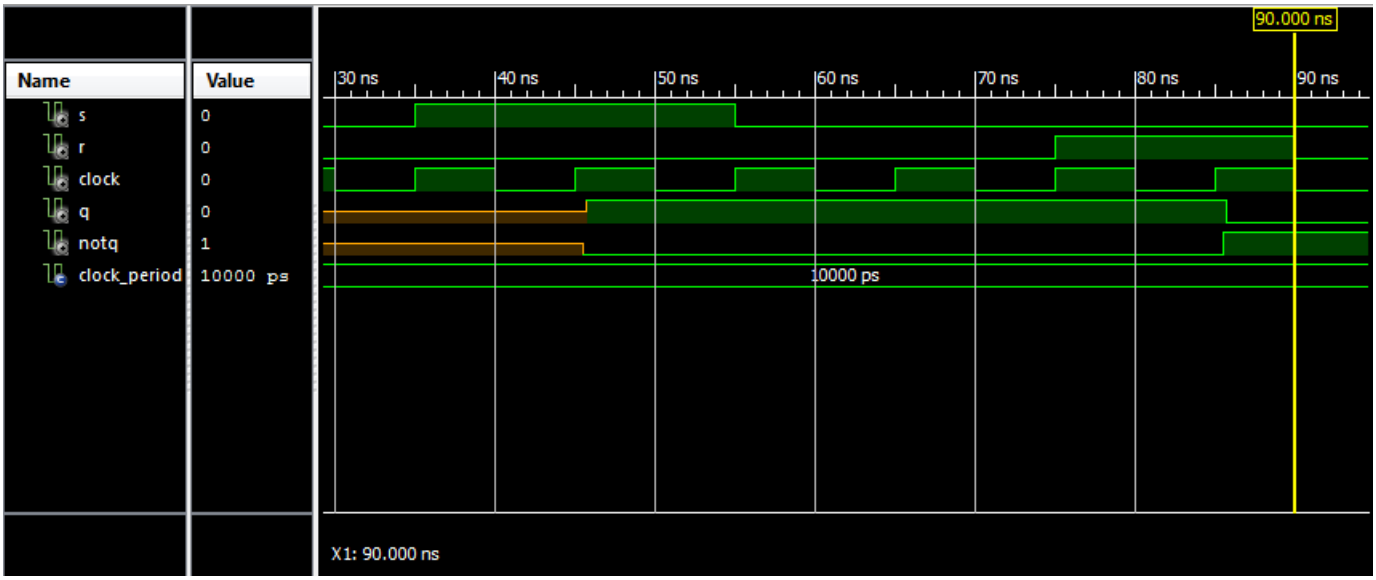


Figura 3.14: Simulazione del Flip-Flop RS Master-Slave Post-Route

Capitolo 4

Display a 7 segmenti

4.1 Traccia

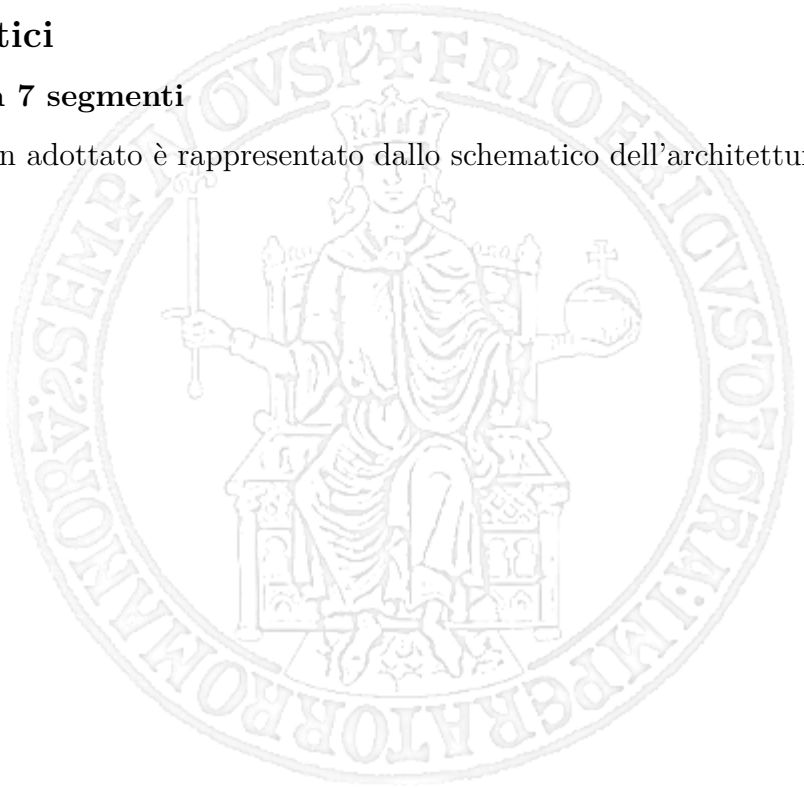
Illustrare la realizzazione di un'architettura che consenta di mostrare su un array di 4 display a 7 segmenti un valore intero. Tale puo essere una parola da 16 bit, composta cioè di 4 cifre esadecimali, ciascuna espressa su di un nibble (4 bit). Sviluppare la traccia discutendo l'approccio di design adottato.

4.2 Soluzione

4.2.1 Schematici

4.2.1.1 Display a 7 segmenti

L'approccio di design adottato è rappresentato dallo schematico dell'architettura in figura 4.1.



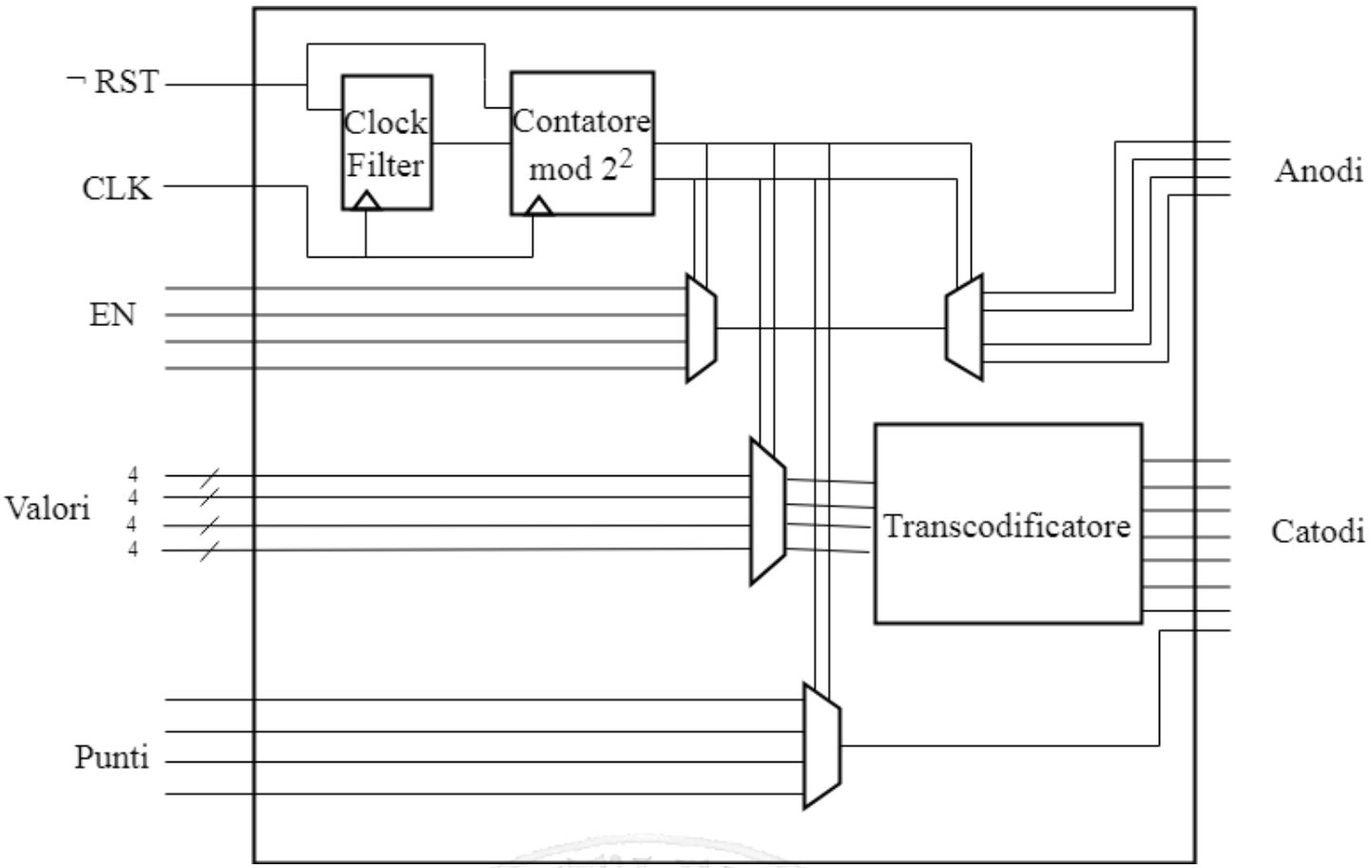


Figura 4.1: Architettura del display a sette segmenti

Gli input sono rappresentati da un segnale di clock, uno di reset in logica 0 attiva, le abilitazioni per le quattro cifre del display, i valori che le stesse cifre devono assumere e l'eventuale posizione dei punti da accendere. Gli output sono costituiti dai segnali degli anodi comuni ai sette segmenti, uno per ciascuna cifra, e dai segnali dei catodi, sette dei quali in riferimento ai segmenti della cifra in questione, l'ultimo relativo al punto ad essa associato.
Il segnale di clock è filtrato da un clock filter, che restituisce un segnale di hit a frequenza minore. Il clock permette, inoltre, il conteggio al contatore modulo 4, abilitato dal segnale di hit del clock filter. Il reset, se negato, consente il ripristino del clock filter e del contatore. L'uscita del contatore consente di selezionare una cifra del display ad ogni conteggio. Infatti, avendo le quattro cifre i catodi dei segmenti in comune, per mostrare un valore differente su ciascuna di esse è necessario un refresh del valore con una frequenza sufficientemente elevata.
I segnali di abilitazione permettono di decidere quali cifre del display utilizzare. Essi, in ingresso ad un multiplexer 4x1, sono selezionati dal contatore per considerare in ogni istante l'abilitazione legata alla cifra corrente. L'abilitazione così individuata identifica l'anodo relativo attraverso un demultiplexer 1x4, sempre tramite la selezione del contatore. Ad ogni conteggio, la cifra selezionata avrà anodo basso o alto, a seconda dei valori di abilitazione passati in ingresso.
I 16 bit di ingresso dei valori rappresentano i quattro nibble relativi alle quattro cifre del display. Una struttura di multiplexing seleziona, in riferimento ai valori di conteggio, il nibble relativo alla cifra corrente. I 4 bit del nibble entrano in un transcodificatore per la corretta accensione dei sette segmenti della cifra.

L'ottavo catodo della cifra è costituito dal punto ad essa associato. I quattro segnali di ingresso dei punti, infatti, decretano se il punto, associato alla cifra selezionata, debba essere acceso, o meno. Un multiplexer 4x1 seleziona, grazie al conteggio, il valore corretto da assegnare al punto della cifra corrente.
L'implementazione dell'architettura è stata realizzata mediante la rappresentazione RT Level di figura 4.2.

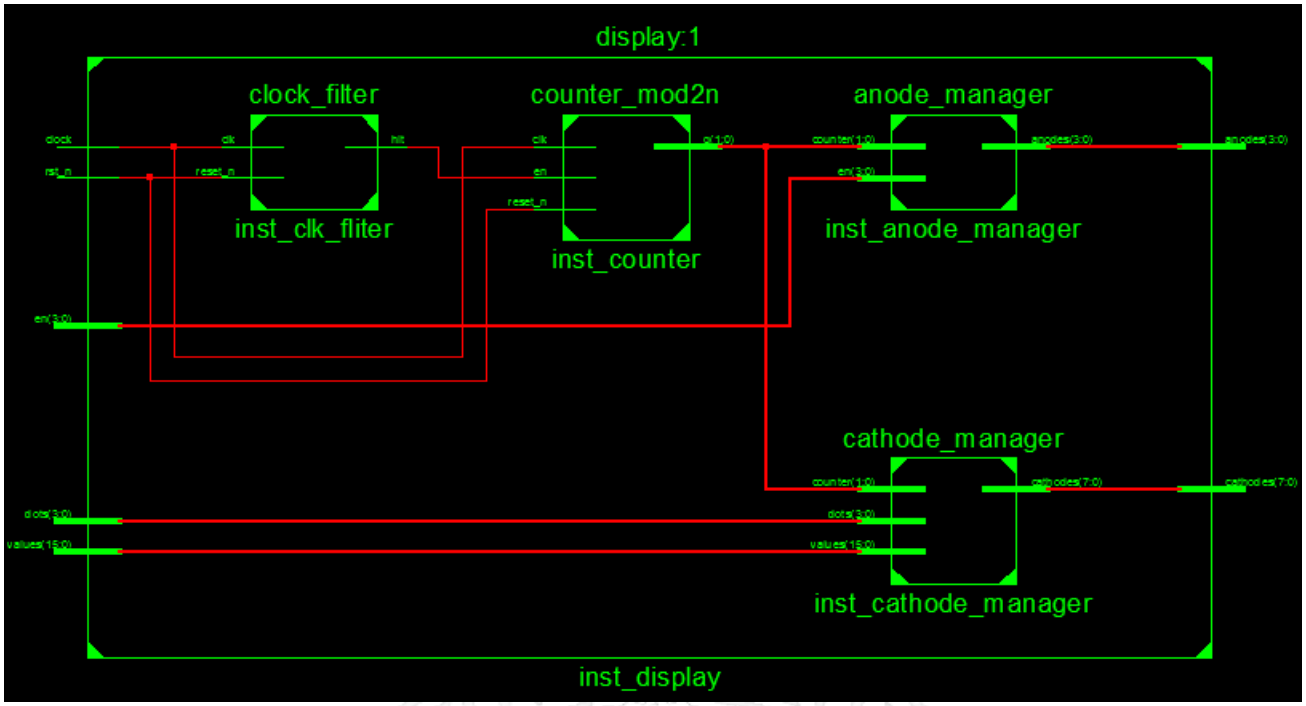


Figura 4.2: Display RTL Schematic

4.2.1.2 Struttura di multiplexing 16x4

Esplodiamo l'architettura che consente la selezione del nibble relativo alla cifra corrente del conteggio. Essa è realizzata mediante 4 multiplexer 4x1, ognuno dei quali riceve in ingresso un bit di pari peso dei nibble associati alle diverse cifre. L'ingresso di selezione è costituito dall'uscita del contatore.

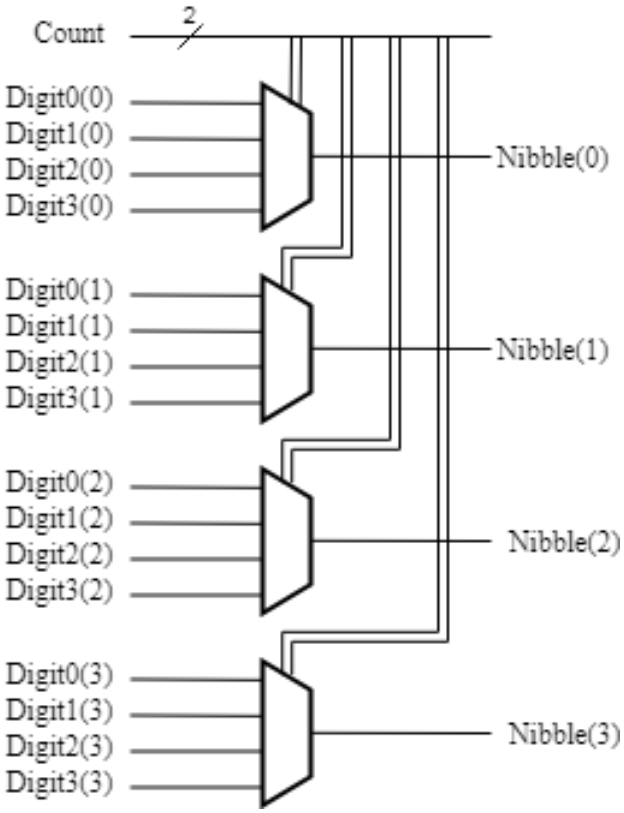


Figura 4.3: Architettura di multiplexing 16x4

4.2.2 Codice

4.2.2.1 Clock Filter

Il clock filter riceve in ingresso un segnale di reset in logica 0 attiva per ritornare ad uno stato neutro in modo asincrono all'elaborazione del conteggio. Esso funge da divisore di frequenza, restituendo il segnale hit con una frequenza sottomultipla di quella del clock in ingresso. Il rapporto tra le due frequenze rappresenta il numero di conteggi da effettuare dopo il quale il filtro deve alzare hit (nemero di colpi di clock in un periodo di hit). Non si è potuto sfruttare un contatore modulo 2^N perché non è detto che il conteggio sia potenza di 2.

```
1  library IEEE;
2  use IEEE.STD_LOGIC_1164.ALL;
3
4  entity clock_filter is
5    Generic (    freq_clock : NATURAL := 50000000;
6           freq_hit : NATURAL := 250
7         );
8      Port ( reset_n : in  STD_LOGIC;
9             clk : in  STD_LOGIC;
10            hit : out  STD_LOGIC
11     );
12 end clock_filter;
13
```

```
architecture Behavioral of clock_filter is
constant freq_ratio : NATURAL := freq_clock/freq_hit;
begin
-- Ho bisogno di un contatore N
  hit_counter_n : process(clk, reset_n)
  variable count : NATURAL := 0;
  begin
    if reset_n = '0' then
      count := 0;
      hit <= '0';
    elsif rising_edge(clk) then
      if count = (freq_ratio-1) then
        count := 0;
        hit <= '1';
      else
        count := count + 1;
        hit <= '0';
      end if;
    end if;
  end process;
end Behavioral;
```

Codice Componente 4.1: Definizione del componente Clock Filter

4.2.2.2 Anode Manager

Il gestore degli anodi utilizza una rete mux-demux per abilitare ad ogni conteggio l'anodo corretto. L'uscita del contatore rappresenta l'ingresso di selezione sia per il multiplexer, che per il demultiplexer. Gli ingressi di abilitazione entrano in un mux 4x1. L'uscita del multiplexer è posta in ingresso al demultiplexer. Lo stesso segnale di selezione garantisce che l'abilitazione sarà portata fino all'anodo a cui essa fa riferimento.

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity anode_manager is
    Port (  counter : in STD_LOGIC_VECTOR (1 downto 0);
            en : in  STD_LOGIC_VECTOR (3 downto 0);
       anodes : out  STD_LOGIC_VECTOR (3 downto 0)
       );
end anode_manager;

architecture structural of anode_manager is

component muxn_1 is
  generic(address_width : natural := 3);
  port (
    SEL : in STD_LOGIC_VECTOR(address_width-1 downto 0);
    A : in STD_LOGIC_VECTOR(2**address_width-1 downto 0);
```

```
18      X : out STD_LOGIC
19    );
20  end component;
21
22  component demux1_n is
23    generic(address_width : natural := 2);
24    port(
25      a : in std_logic;
26      sel : in std_logic_vector(address_width-1 downto 0);
27      x : out std_logic_vector(2**address_width-1 downto 0)
28    );
29  end component;
30  for all : demux1_n use entity WORK.demux1_n(dataflow);
31
32  signal enable_anode : STD_LOGIC := '0';
33
34  begin
35
36    inst_mux4_1 : muxn_1
37      generic map ( address_width => 2 )
38      port map (  SEL => counter,
39                  A => en,
40                  X => enable_anode
41                );
42
43    inst_demux4_1 : demux1_n
44      generic map ( address_width => 2 )
45      port map (  sel => counter,
46                  a => enable_anode,
47                  x => anodes
48                );
49
50  end structural;
```

Codice Componente 4.2: Definizione del componente Anode Manager

4.2.2.3 Cathode Manager

Il gestore dei catodi riceve in ingresso il conteggio, i valori che i catodi dei sette segmenti devono assumere e quelli del catodo dei punti. L'uscita è costituita dagli 8 segnali dei catodi. Il componente sfrutta la struttura di multiplexing descritta in 4.2.1.2. Per fare ciò riordina i valori di ingresso nel modo richiesto dalla batteria di mux 4x1. Il nibble così ottenuto entra nel transcodificatore, realizzato col componente cathode encoder, dal quale si ottengono i segnali dei catodi dei sette segmenti. Il segnale dell'ultimo catodo è ricavato dal multiplexing dei valori dei punti in ingresso: un mux 4x1 determina quale valore porre in uscita sul catodo.

```
1  library IEEE;
2  use IEEE.STD_LOGIC_1164.ALL;
3
```

```
4  entity cathode_manager is
5      Port (  counter : in STD_LOGIC_VECTOR (1 downto 0);
6           values : in STD_LOGIC_VECTOR (15 downto 0);
7           dots : in  STD_LOGIC_VECTOR (3 downto 0);
8         cathodes : out  STD_LOGIC_VECTOR (7 downto 0)
9         );
10 end cathode_manager;
11
12 architecture structural of cathode_manager is
13
14 component muxn_1 is
15   generic(address_width : NATURAL := 3);
16   port (
17     SEL : in STD_LOGIC_VECTOR(address_width-1 downto 0);
18     A : in STD_LOGIC_VECTOR(2**address_width-1 downto 0);
19     X : out STD_LOGIC
20   );
21 end component;
22
23 component cathode_encoder is
24     Port ( nibble : in  STD_LOGIC_VECTOR (3 downto 0);
25            cathodes : out  STD_LOGIC_VECTOR (6 downto 0)
26     );
27 end component;
28 for all : cathode_encoder use entity WORK.cathode_encoder(behavioral);
29
30 signal nibble : STD_LOGIC_VECTOR (3 downto 0) := (others => '0');
31 alias digit0 : STD_LOGIC_VECTOR(3 downto 0) is values(3 downto 0);
32 alias digit1 : STD_LOGIC_VECTOR(3 downto 0) is values(7 downto 4);
33 alias digit2 : STD_LOGIC_VECTOR(3 downto 0) is values(11 downto 8);
34 alias digit3 : STD_LOGIC_VECTOR(3 downto 0) is values(15 downto 12);
35 signal in_mux : STD_LOGIC_VECTOR (15 downto 0) := (others => '0'); --per
       ordinare i valori da porre in ingresso ai mux 4x1
36
37 begin
38
39   mux16_4 : for i in 0 to 3 generate
40     in_mux((i*4+3) downto i*4) <= (digit3(i), digit2(i), digit1(i), digit0(i
         ));
41     -- i=0 --> in_mux(3 downto 0)
42     -- i=1 --> in_mux(7 downto 4)
43     -- i=2 --> in_mux(11 downto 8)
44     -- i=3 --> in_mux(15 downto 12)
45     inst_mux4_1 : muxn_1
46       generic map ( address_width => 2 )
47       port map (  SEL => counter,
48             A => in_mux((i*4+3) downto i*4),
49             X => nibble(i)
50            );
```

```
51    end generate;
52
53    inst_encoder : cathode_encoder
54      port map ( nibble => nibble,
55              cathodes => cathodes (6 downto 0)
56            );
57
58    inst_dots_manager : muxn_1
59      generic map ( address_width => 2 )
60        port map ( SEL => counter,
61                A => dots,
62                X => cathodes(7)
63              );
64
65  end structural;
```

Codice Componente 4.3: Definizione del componente Cathode Manager

4.2.2.4 Cathode_encoder

Questo componente effettua la funzione di transcodifica di un nibble nelle funzioni booleane dei sette segmenti. Esso, dunque, riceve in input un nibble e restituisce il valore assunto dai catodi dei sette segmenti.

```
1  library IEEE;
2  use IEEE.STD_LOGIC_1164.ALL;
3
4  entity cathode_encoder is
5      Port ( nibble : in  STD_LOGIC_VECTOR (3 downto 0);
6             cathodes : out  STD_LOGIC_VECTOR (6 downto 0)
7           );
8  end cathode_encoder;
```

Codice Componente 4.4: Interfaccia del componente Cathode Encoder

A scopo didattico, la sua implementazione è stata effettuata seguendo differenti approcci.

Behavioral Questo approccio comportamentale prevede, appunto, la specifica del comportamento dell'uscita in concomitanza a ciascuna combinazione degli ingressi mediante costrutto *case-when*. Sono state definite delle costanti per ogni caso al fine di migliorare la leggibilità.

```
1  architecture Behavioral of cathode_encoder is
2
3  constant digit0 : std_logic_vector(6 downto 0) := "1000000";
4  constant digit1 : std_logic_vector(6 downto 0) := "1111001";
5  constant digit2 : std_logic_vector(6 downto 0) := "0100100";
6  constant digit3 : std_logic_vector(6 downto 0) := "0110000";
7  constant digit4 : std_logic_vector(6 downto 0) := "0011001";
8  constant digit5 : std_logic_vector(6 downto 0) := "0010010";
9  constant digit6 : std_logic_vector(6 downto 0) := "0000010";
```

```
10 constant digit7 : std_logic_vector(6 downto 0) := "1011000";
11 constant digit8 : std_logic_vector(6 downto 0) := "0000000";
12 constant digit9 : std_logic_vector(6 downto 0) := "0010000";
13 constant digitA : std_logic_vector(6 downto 0) := "0001000";
14 constant digitB : std_logic_vector(6 downto 0) := "0000011";
15 constant digitC : std_logic_vector(6 downto 0) := "1000110";
16 constant digitD : std_logic_vector(6 downto 0) := "0100001";
17 constant digitE : std_logic_vector(6 downto 0) := "0000110";
18 constant digitF : std_logic_vector(6 downto 0) := "0001110";
19
20 begin
21   main_cathode : process(nibble)
22   begin
23     case nibble is
24       when x"0" =>
25         cathodes <= digit0;
26       when x"1" =>
27         cathodes <= digit1;
28       when x"2" =>
29         cathodes <= digit2;
30       when x"3" =>
31         cathodes <= digit3;
32       when x"4" =>
33         cathodes <= digit4;
34       when x"5" =>
35         cathodes <= digit5;
36       when x"6" =>
37         cathodes <= digit6;
38       when x"7" =>
39         cathodes <= digit7;
40       when x"8" =>
41         cathodes <= digit8;
42       when x"9" =>
43         cathodes <= digit9;
44       when x"A" =>
45         cathodes <= digitA;
46       when x"B" =>
47         cathodes <= digitB;
48       when x"C" =>
49         cathodes <= digitC;
50       when x"D" =>
51         cathodes <= digitD;
52       when x"E" =>
53         cathodes <= digitE;
54       when x"F" =>
55         cathodes <= digitF;
56       when others =>
57         cathodes <= (others => '1');
58     end case;
```

```
59    end process;
60
61  end Behavioral;
```

Codice Componente 4.5: Architettura Behavioral del componente Cathode Encoder

Structural Questo approccio realizza il transcodificatore con 7 mux 8x1, mostrando l'utilizzo del multiplexer come rete universale. I 3 bit più significativi del nibble vanno a costituire l'ingresso di selezione dei multiplexer, mentre il bit meno significativo permette la codifica degli ingressi. In figura 4.4 viene rappresentato il caso per la funzione booleana del segmento A. Si assumono gli ingressi denominati come x,y,z,v, a partire dal bit più significativo.

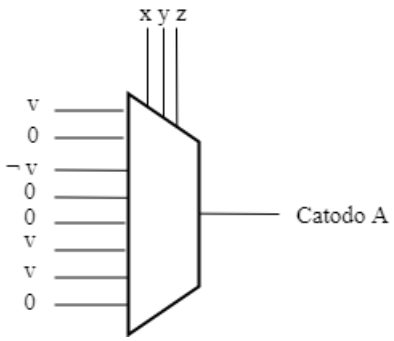


Figura 4.4: Multiplexer 8x1 che realizza la funzione booleana del segmento A

Dopo la costruzione dei vettori degli ingressi per ogni multiplexer, si è definito un vettore di vettori per consentire una generazione della struttura mediante il costrutto *for-generate*.

```
1  architecture structural of cathode_encoder is
2    COMPONENT muxn_1
3     generic(address_width : natural := 3);
4    PORT(
5       SEL : IN std_logic_vector(address_width-1 downto 0);
6       A : IN std_logic_vector(2**address_width-1 downto 0);
7       X : OUT std_logic
8       );
9    END COMPONENT;
10 alias v : std_logic is nibble(0);
11 constant width : natural := 3;
12 signal in_A : std_logic_vector(7 downto 0) := (others => '1');
13 signal in_B : std_logic_vector(7 downto 0) := (others => '1');
14 signal in_C : std_logic_vector(7 downto 0) := (others => '1');
15 signal in_D : std_logic_vector(7 downto 0) := (others => '1');
16 signal in_E : std_logic_vector(7 downto 0) := (others => '1');
17 signal in_F : std_logic_vector(7 downto 0) := (others => '1');
18 signal in_G : std_logic_vector(7 downto 0) := (others => '1');
19 type array_bidim is array (6 downto 0) of std_logic_vector(7 downto 0);
20 signal ingressi : array_bidim := (others => (others => '1'));
21 begin
```

```
22    in_A(7 downto 0) <= (0 => v, 1 => '0', 2 => not v, 3 => '0', 4 => '0', 5
          => v, 6 => v, 7 => '0');
23    in_B(7 downto 0) <= (0 => '0', 1 => '0', 2 => v, 3 => not v, 4 => '0', 5
          => v, 6 => not v, 7 => '1');
24    in_C(7 downto 0) <= (0 => '0', 1 => not v, 2 => '0', 3 => '0', 4 => '0', 5
           => '0', 6 => not v, 7 => '1');
25    in_D(7 downto 0) <= (0 => v, 1 => '0', 2 => not v, 3 => v, 4 => '0', 5 =>
          not v, 6 => '0', 7 => v);
26    in_E(7 downto 0) <= (0 => v, 1 => v, 2 => '1', 3 => v, 4 => v, 5 => '0', 6
           => '0', 7 => '0');
27    in_F(7 downto 0) <= (0 => v, 1 => '1', 2 => '0', 3 => '0', 4 => '0', 5 =>
          '0', 6 => v, 7 => '0');
28    in_G(7 downto 0) <= (0 => '1', 1 => '0', 2 => '0', 3 => v, 4 => '0', 5 =>
          '0', 6 => not v, 7 => '0');
29    ingressi <= (in_G, in_F, in_E, in_D, in_C, in_B, in_A);
30
31    setteseg : for i in 0 to 6 generate
32      Inst_muxn_1: muxn_1 GENERIC MAP(width) PORT MAP(
33        SEL => nibble(3 downto 1),
34        A => ingressi(i),
35        X => cathodes(i)
36      );
37    end generate;
38  end structural;
```

Codice Componente 4.6: Architettura Structural del componente Cathode Encoder

Dataflow Denominati gli ingressi come nel paragrafo precendete, si è provveduto a definire un file blif per la funzione multi-uscita dei sette segmenti. In tal modo è stata possibile una minimizzazione con SIS attraverso l'uso del *rugged-script*.

```
sis> set autoexec print_stats
7segmenti  pi= 4  po= 7  node=  7  latch= 0  lits(sop)= 132  lits(ff)=  96
sis> source -x script.rugged
sweep; eliminate -1
7segmenti  pi= 4  po= 7  node=  7  latch= 0  lits(sop)= 132  lits(ff)=  96
7segmenti  pi= 4  po= 7  node=  7  latch= 0  lits(sop)= 132  lits(ff)=  96
simplify -m nocomp
7segmenti  pi= 4  po= 7  node=  7  latch= 0  lits(sop)=  59  lits(ff)=  51
eliminate -1
7segmenti  pi= 4  po= 7  node=  7  latch= 0  lits(sop)=  59  lits(ff)=  51

sweep; eliminate 5
7segmenti  pi= 4  po= 7  node=  7  latch= 0  lits(sop)=  59  lits(ff)=  51
7segmenti  pi= 4  po= 7  node=  7  latch= 0  lits(sop)=  59  lits(ff)=  51
simplify -m nocomp
7segmenti  pi= 4  po= 7  node=  7  latch= 0  lits(sop)=  59  lits(ff)=  51
resub -a
7segmenti  pi= 4  po= 7  node=  7  latch= 0  lits(sop)=  59  lits(ff)=  51

fx
7segmenti  pi= 4  po= 7  node=  9  latch= 0  lits(sop)=  55  lits(ff)=  51
resub -a; sweep
7segmenti  pi= 4  po= 7  node=  9  latch= 0  lits(sop)=  55  lits(ff)=  51
7segmenti  pi= 4  po= 7  node=  9  latch= 0  lits(sop)=  55  lits(ff)=  51

eliminate -1; sweep
7segmenti  pi= 4  po= 7  node=  9  latch= 0  lits(sop)=  55  lits(ff)=  51
7segmenti  pi= 4  po= 7  node=  9  latch= 0  lits(sop)=  55  lits(ff)=  51
full_simplify -m nocomp
7segmenti  pi= 4  po= 7  node=  9  latch= 0  lits(sop)=  55  lits(ff)=  51
7segmenti  pi= 4  po= 7  node=  9  latch= 0  lits(sop)=  55  lits(ff)=  51
```

Figura 4.5: Funzione multi-uscita dei sette segmenti minimizzata col rugged-script

Ogni uscita è calcolata in base alla forma minimizzata della funzione corrispondente.

```
1  architecture dataflow of cathode_encoder is
2
3  alias x : std_logic is nibble(3);
4  alias y : std_logic is nibble(2);
5  alias z : std_logic is nibble(1);
6  alias v : std_logic is nibble(0);
7
8  -- generati dalla minimizzazione
9  signal t7 : std_logic := '1';
10 signal t8 : std_logic := '1';
11 -- cathodes(i) non puo' essere usato per definire un'altra uscita, essendo
       un segnale di output
12 signal A : std_logic := '1';
13 signal B : std_logic := '1';
14 signal C : std_logic := '1';
15 signal D : std_logic := '1';
16 signal E : std_logic := '1';
17 signal F : std_logic := '1';
18 signal G : std_logic := '1';
19
20 begin
21   t7 <= (not z) or (not y);
22   t8 <= (not x) and (not y);
23
24   cathodes(0) <= A;
25   cathodes(1) <= B;
```

```
26    cathodes(2) <= C;
27    cathodes(3) <= D;
28    cathodes(4) <= E;
29    cathodes(5) <= F;
30    cathodes(6) <= G;
31
32    A <= (v and (not E) and t7) or ((not z) and E and G) or ((not v) and E);
33    B <= (v and (not E) and (not F)) or (y and (not D) and (not F));
34    C <= ((not E) and (not G) and t8) or (x and y and (not A));
35    D <= ((not y) and z and (not v) and (not C)) or (v and (not t7)) or (A and
          E);
36    E <= ((not y) and (not z) and v) or ((not x) and y and (not z)) or ((not x
          ) and v);
37    F <= ((not z) and A and (not E)) or (E and t8) or ((not y) and C);
38    G <= (y and (not z) and (not v) and (not E)) or ((not z) and t8) or (E and
           (not t7));
```

Codice Componente 4.7: Architettura Dataflow del componente Cathode Encoder

4.2.2.5 Display

Il display non fa altro che realizzare la struttura descritta in 4.2.1.1. Sono infatti istanziati e collegati nel modo descritto i componenti cui abbiamo fatto riferimento.

```
1  library IEEE;
2  use IEEE.STD_LOGIC_1164.ALL;
3
4  entity display is
5    Generic (    freq_clock : NATURAL := 50000000;
6          freq_hit : NATURAL := 250
7          );
8     Port (    rst_n : in  STD_LOGIC;
9        clock : in STD_LOGIC;
10       en : in  STD_LOGIC_VECTOR (3 downto 0);
11       values : in  STD_LOGIC_VECTOR (15 downto 0);
12       dots : in  STD_LOGIC_VECTOR (3 downto 0);
13       anodes : out  STD_LOGIC_VECTOR (3 downto 0);
14       cathodes : out  STD_LOGIC_VECTOR (7 downto 0)
15       );
16 end display;
17
18 architecture structural of display is
19
20 component counter_mod2n is
21   generic ( width : NATURAL := 2);
22   port (  en : in  STD_LOGIC;
23       reset_n : in  STD_LOGIC;
24       clk : in  STD_LOGIC;
25       q : out  STD_LOGIC_VECTOR (width-1 downto 0)
```

```
26        );
27  end component;
28
29  component clock_filter is
30    Generic (    freq_clock : NATURAL := 50000000;
31             freq_hit : NATURAL := 250
32          );
33      Port ( reset_n : in  STD_LOGIC;
34             clk : in  STD_LOGIC;
35             hit : out  STD_LOGIC
36      );
37  end component;
38
39  component cathode_manager is
40      Port (  counter : in STD_LOGIC_VECTOR (1 downto 0);
41          values : in STD_LOGIC_VECTOR (15 downto 0);
42          dots : in  STD_LOGIC_VECTOR (3 downto 0);
43        cathodes : out  STD_LOGIC_VECTOR (7 downto 0)
44        );
45  end component;
46
47  component anode_manager is
48      Port (  counter : in STD_LOGIC_VECTOR (1 downto 0);
49          en : in  STD_LOGIC_VECTOR (3 downto 0);
50        anodes : out  STD_LOGIC_VECTOR (3 downto 0)
51        );
52  end component;
53
54
55  signal hit : STD_LOGIC := '0';
56  signal sel: STD_LOGIC_VECTOR(1 downto 0) := (others => '0');
57
58  begin
59    inst_clk_fliter : clock_filter
60      generic map (    freq_clock => freq_clock,
61               freq_hit => freq_hit )
62      port map (  reset_n => rst_n,
63             clk => clock,
64             hit => hit
65             );
66    inst_counter : counter_mod2n
67      generic map ( width => 2 )
68      port map (  en => hit,
69             reset_n => rst_n,
70             clk => clock,
71             q => sel
72             );
73
74    inst_cathode_manager : cathode_manager
```

```
75      port map (  counter => sel,
76              values => values,
77              dots => dots,
78              cathodes => cathodes
79              );
80
81    inst_anode_manager : anode_manager
82      port map (  counter => sel,
83              en => en,
84              anodes => anodes
85              );
86
87  end structural;
```

Codice Componente 4.8: Definizione del componente Display

4.3 Simulazione

Per la simulazione sono state valutate differenti configurazioni degli ingressi mediante il costrutto *for-loop*.

```
1     -- Stimulus process
2     stim_proc: process
3     begin
4      rst_n <= '0';
5        -- hold reset state for 100 ns.
6        wait for 100 ns;
7
8        wait for clock_period*10;
9
10       -- insert stimulus here
11     for t in std_logic range '0' to '1' loop
12       rst_n <= not t;
13       for i in 0 to 2 loop
14         en <= std_logic_vector(to_unsigned(i*4, 4));
15         dots <= std_logic_vector(to_unsigned(i*4, 4));
16         -- i = 0 --> en = dots = 0000
17         -- i = 1 --> en = dots = 0100
18         -- i = 2 --> en = dots = 1000
19         for j in 1 to 2 loop
20           values <= std_logic_vector(to_unsigned(j*8, 16));
21           -- j = 1 --> values = 0000 0000 0000 1000
22           -- j = 2 --> values = 0000 0000 0001 0000
23           wait for freq_clock/freq_hit*clock_period*4; -- per osservare un
                 ciclo del contatore
24         end loop;
25       end loop;
26     end loop;
```

```
27
28          wait;
29      end process;
30
31  END;
```

Codice Componente 4.9: Simulazione del componente Display

Di seguito il segnale *values* sarà rappresentato in esadecimale. In figura 4.6 osserviamo come al variare di *values* segue la commutazione del segnale *cathodes* in modo coerente al conteggio. Quando infatti il valore di ingresso è pari a *x"0008"*, al primo conteggio i catodi seguono l'ingresso diverso per la prima cifra. Analogamente per la seconda cifra quando il valore di ingresso è pari a *x"0010"*.

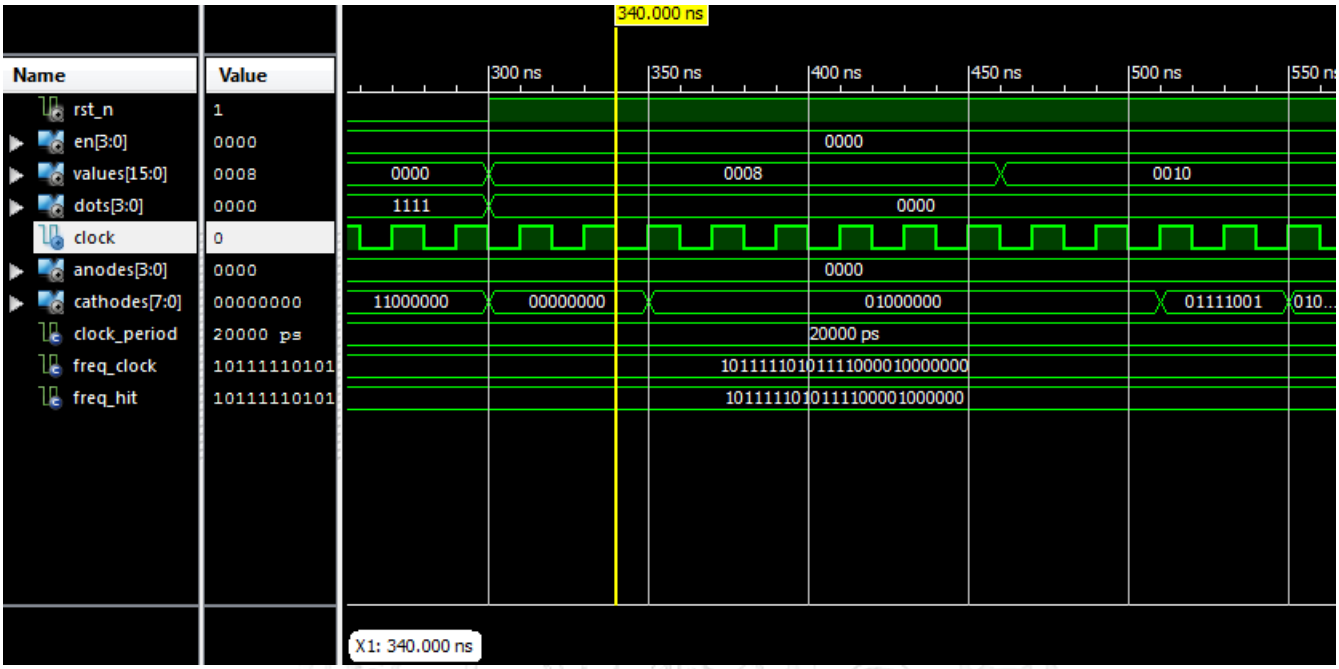


Figura 4.6: Simulazione del display: i catodi seguono correttamente il valore di ingresso per i sette segmenti delle diverse cifre

In figura 4.7 notiamo che, al variare di *en* e *dots*, segue la commutazione di *anodes* e *cathodes*, coerentemente al valore di conteggio. Quando, infatti, i segnali di abilitazione e dei punti sono pari a x*"0100"*, al conteggio associato alla terza cifra seguono nella commutazione anche il relativo anodo e il catodo che rappresenta il punto.

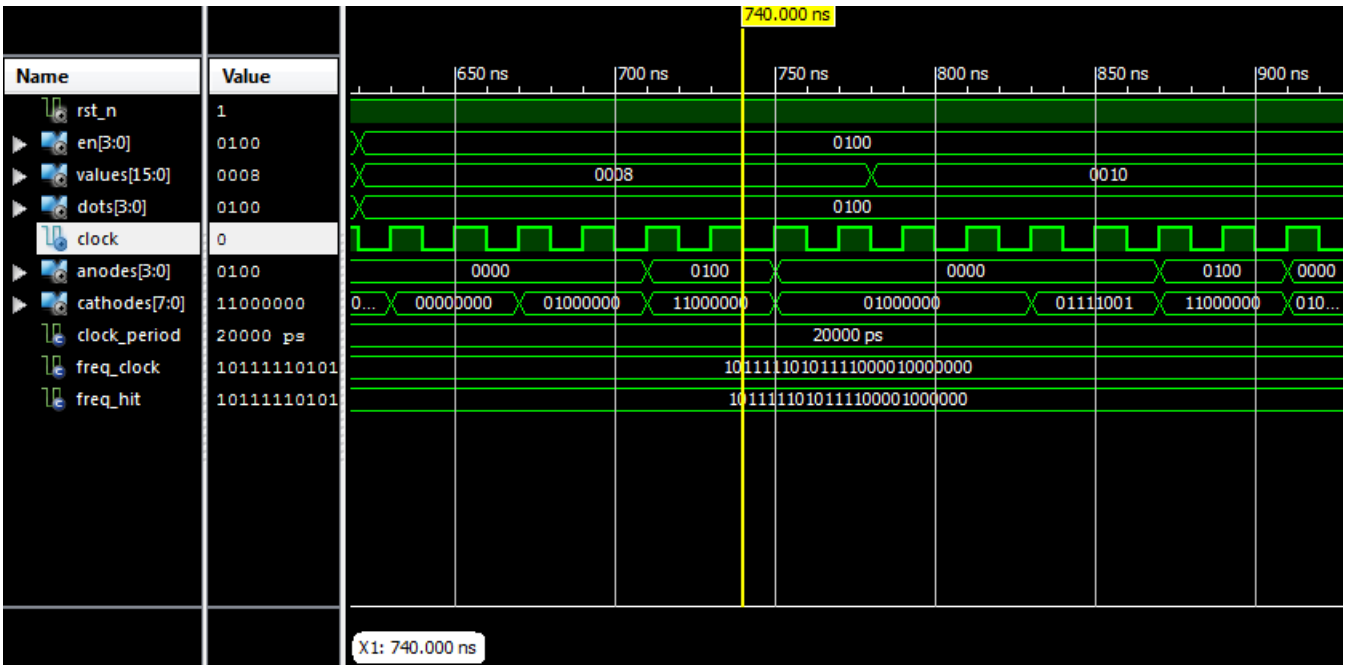


Figura 4.7: Simulazione del display: gli anodi seguono correttamente i valori di abilitazione; il catodo del punto segue correttamente i valori dei punti

Con *rst_n* pari a 0, ovvero il segnale di reset attivo, il conteggo si blocca alla prima cifra. Osserviamo in figura 4.8 che le uscite segnano identicamente i valori relativi alla prima cifra, indipendentemente dalla variazione degli altri ingressi. Esse infatti non sono sensibili alle variazioni di *en* e *dots* sulla terza cifra (*"0100"*), ma notiamo una commutazione al variare di *values* sulla prima cifra (*x"0008"*).

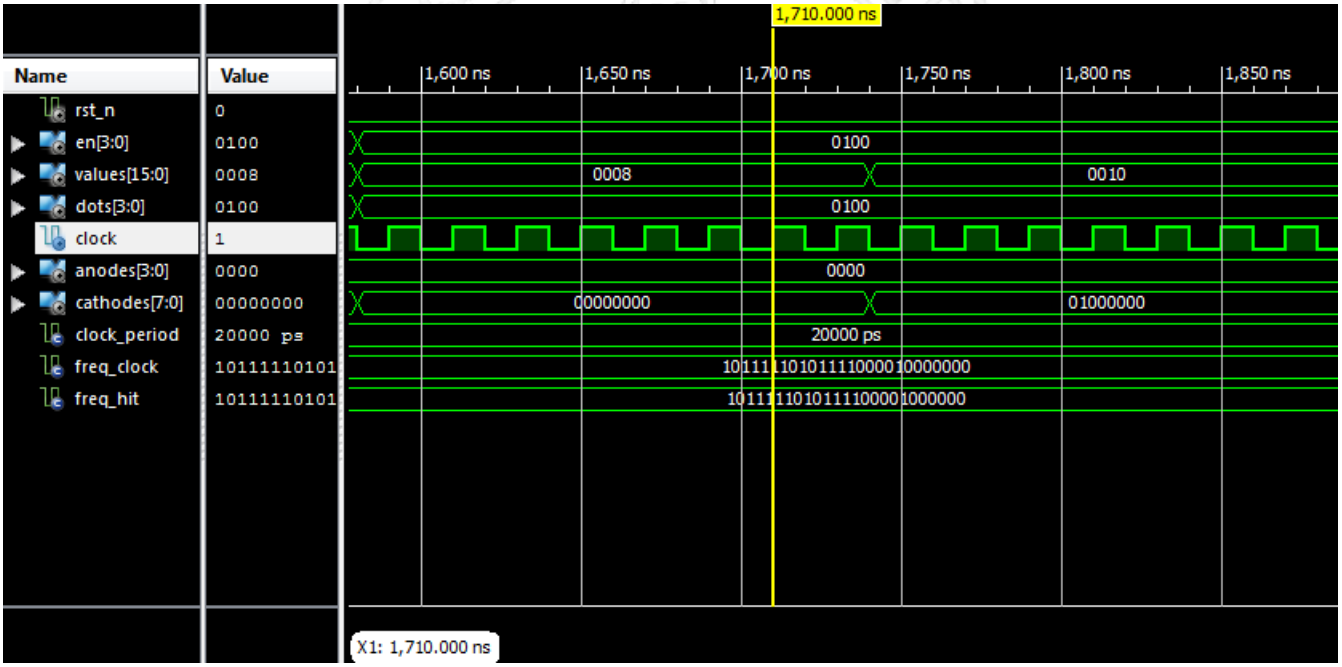


Figura 4.8: Simulazione del display: il reset basso blocca il conteggio alla prima cifra

4.4 Sintesi su board FPGA

La soluzione adottata per interfacciare il design con la board fa uso di una semplice unità di controllo. Essa consente di ricevere i valori di input tramite gli interruttori della board, e caricarli nei registri relativi mediante i pulsanti.

```
entity display_cu is
    Generic ( freq_clock : NATURAL := 50000000;
              freq_hit : NATURAL := 250
            );
     Port ( clock : in  STD_LOGIC;
            reset : in  STD_LOGIC;
            loader : in  STD_LOGIC_VECTOR (2 downto 0);
            input_byte : in  STD_LOGIC_VECTOR (7 downto 0);
            anodes_n : out  STD_LOGIC_VECTOR (3 downto 0);
            cathodes : out  STD_LOGIC_VECTOR (7 downto 0)
          );
end display_cu;
```

Codice Componente 4.10: Entità della Control Unit

Il segnale di *clock* è associato al clock della board, il *reset* ed i tre bit di *loader* sono associati ai quattro pulsanti, *input_byte* agli otto interruttori, *anodes_n* ai quattro anodi e *cathodes* agli otto catodi. Sono utilizzate tre batterie di otto flip flop D edge triggered per salvare l'input nel giusto registro a seconda del pulsante premuto. Esse, infatti, hanno per abilitazione ciascuna un differente bit di *loader*, e quindi un differente pulsante. Il reset negato ed i vari registri sono poi collegati al display. Per i punti, il registro è collegato al display in forma negata. Ciò perché sono presi in input i punti da abilitare, abilitazione che corrisponde a valle ad un abbassamento del catodo relativo. Infine, si noti che gli anodi della board sono associati al valore negato di quelli in uscita dal display: essi sono pilotati con una logica 0 attiva.

```
architecture Structural of display_cu is

component display is
  Generic ( freq_clock : NATURAL := 50000000;
            freq_hit : NATURAL := 250
          );
   Port ( rst_n : in  STD_LOGIC;
          clock : in STD_LOGIC;
             en : in  STD_LOGIC_VECTOR (3 downto 0);
             values : in  STD_LOGIC_VECTOR (15 downto 0);
             dots : in  STD_LOGIC_VECTOR (3 downto 0);
             anodes : out  STD_LOGIC_VECTOR (3 downto 0);
             cathodes : out  STD_LOGIC_VECTOR (7 downto 0)
        );
end component;

component edge_triggered_d_n is
  Generic ( width : NATURAL := 4);
    Port ( d : in  STD_LOGIC_VECTOR (width-1 downto 0);
```

```
20              clk : in  STD_LOGIC;
21              reset_n : in  STD_LOGIC;
22              en : in  STD_LOGIC;
23              q : out  STD_LOGIC_VECTOR (width-1 downto 0)
24          );
25  end component;
26
27  signal reset_n : STD_LOGIC := '1';
28  alias loader_values : STD_LOGIC_VECTOR(1 downto 0) is loader(1 downto 0);
29  alias loader_en_dots : STD_LOGIC is loader(2);
30  signal values_reg : STD_LOGIC_VECTOR(15 downto 0) := (others => '0');
31  signal en_dots_reg : STD_LOGIC_VECTOR(7 downto 0) := (others => '0');
32  alias dots_reg : STD_LOGIC_VECTOR(3 downto 0) is en_dots_reg(3 downto 0);
33  alias en_reg : STD_LOGIC_VECTOR(3 downto 0) is en_dots_reg(7 downto 4);
34  signal dots_cathode_values : STD_LOGIC_VECTOR(3 downto 0) := (others => '1')
        ;
35  signal anodes : STD_LOGIC_VECTOR(3 downto 0) := (others => '0');
36
37  begin
38    reset_n <= not reset;
39    dots_cathode_values <= not dots_reg;
40    anodes_n <= not anodes;
41
42    edge_triggered_lsbvalues : edge_triggered_d_n
43      Generic map (width => 8)
44      Port map (
45              d => input_byte,
46              clk => clock,
47              reset_n => reset_n,
48              en => loader_values(0),
49              q => values_reg(7 downto 0)
50            );
51
52    edge_triggered_msbvalues : edge_triggered_d_n
53      Generic map (width => 8)
54      Port map (
55              d => input_byte,
56              clk => clock,
57              reset_n => reset_n,
58              en => loader_values(1),
59              q => values_reg(15 downto 8)
60            );
61
62    edge_triggered_en_dots : edge_triggered_d_n
63      Generic map (width => 8)
64      Port map (
65              d => input_byte,
66              clk => clock,
67              reset_n => reset_n,
```

```
68              en => loader_en_dots,
69              q => en_dots_reg
70            );
71
72    inst_display : display
73      Generic map( freq_clock => freq_clock,
74                   freq_hit => freq_hit
75                 )
76      Port map(
77             rst_n => reset_n,
78             clock => clock,
79             en => en_reg,
80             values => values_reg,
81             dots => dots_cathode_values,
82             anodes => anodes,
83             cathodes => cathodes
84             );
85
86  end Structural;
```

Codice Componente 4.11: Architettura della Control Unit

Capitolo 5

Clock Generator

5.1 Traccia

Progettare una cella di un comparatore a maggioranza (magnitude comparator) ad 1 bit la cui tabella di verità è riportata in fig.17.1. La sintesi è basata sull'uso di una porta NXOR come mostrato in fig.

Truth Table

Inputs		Outputs		
B	A	A > B	A = B	A < B
0	0	0	1	0
0	1	1	0	0
1	0	0	0	1
1	1	0	1	0

Figura 5.1: Tabella di verità magnitude comparator

Descrivere il circuito in VHDL e simularlo nell'ambiente Xilinx ISE costruendo un ambiente di test per verificarne tutte le funzionalità e gli aspetti di tempificazione.

5.2 Soluzione

5.2.1 Schematici

5.2.2 Codice

5.2.2.1 Bit String Comparator

Il componente Top Module è ottenuto con costrutto *Generate* e le connessioni sono fatte sfruttando tre array monodimensionali (aGT, aLT, aEQ).

I valori MSB di questi array sono posti a (0,0,1) rispettivamente per garantire che sui bit più significativi della stringa in ingresso sia fatta una comparazione assoluta, e non relativa.

```
1 library IEEE;
2 use IEEE.STD_LOGIC_1164.ALL;
3
```

```
4  entity bit_string_comparator is
5  GENERIC(N: Integer:=4);
6      Port ( a : in  STD_LOGIC_VECTOR (N-1 downto 0);
7             b : in  STD_LOGIC_VECTOR (N-1 downto 0);
8             aGTb : out  STD_LOGIC;
9             aEQb : out  STD_LOGIC;
10            aLTb : out  STD_LOGIC);
11 end bit_string_comparator;
12
13 architecture Structural of bit_string_comparator is
14
15 component bit_comparator_comp
16     Port ( a : in  STD_LOGIC;
17            b : in  STD_LOGIC;
18            aEQ_INb : in  STD_LOGIC;
19            aLT_INb : in  STD_LOGIC;
20            aGT_INb : in  STD_LOGIC;
21            aEQb : out  STD_LOGIC;
22            aLTb : out  STD_LOGIC;
23            aGTb : out  STD_LOGIC);
24 end component bit_comparator_comp;
25
26 for all: bit_comparator_comp use entity WORK.bit_comparator(Dataflow);
27
28 signal aGT, aLT, aEQ: STD_LOGIC_VECTOR(N downto 0); --Array di GT, LT ed EQ
       per la propagazione
29 begin
30 aGT(N)<='0';
31 aLT(N)<='0';
32 aEQ(N)<='1'; --Altrimenti non vengono riconosciute mai come uguali!!!
33
34 aGTb<=aGT(0);
35 aEQb<=aEQ(0);
36 aLTb<=aLT(0);
37
38 one_bcc: for i in N-1 downto 0 GENERATE begin
39   bit_comparator: bit_comparator_comp port map(a(i), b(i), aEQ(i+1), aLT(i
       +1), aGT(i+1),aEQ(i), aLT(i), aGT(i));
40 end GENERATE one_bcc;
41
42 end Structural;
```

Codice Componente 5.1: Definizione del componente Bit String Comparator Generic

5.3 Simulazione

5.4 Sintesi su board FPGA

Capitolo 6

Scan Chain

6.1 Traccia

Progettare una rete composta da una serie di N Flip Flop D abilitati ad operare nei seguenti due modi:

1. Modalità normale: l'array si comporta come un registro di N posizioni;
2. Modalità controllo: i flip flop possono essere scritti e letti individualmente configurandoli in cascata come uno shift register.

Utilizzare una rete di controllo in grado di alimentare il primo stadio con un valore e generare tanti colpi di clock quanto è la distanza del primo stadio dalla cella da raggiungere.

6.2 Soluzione

6.2.1 Schematici

La seguente Boundary Scan Chain è realizzata a partire da quattro flip flop edge triggered che in ingresso prendono l'uscita di un multiplexer che, pilotato da un segnale di *scan_en*, decide di selezionare o un ingresso utente *din* oppure l'uscita del flip flop precedente. Quindi quando scan_en è basso il sistema funziona in modalità registro, conservando il dato din, altrimenti funziona da shift-register, shiftando i valori in ingresso al segnale *scan_in*, fino a quello di scan_out dopo un numero di colpi di clock pari al numero di registri.

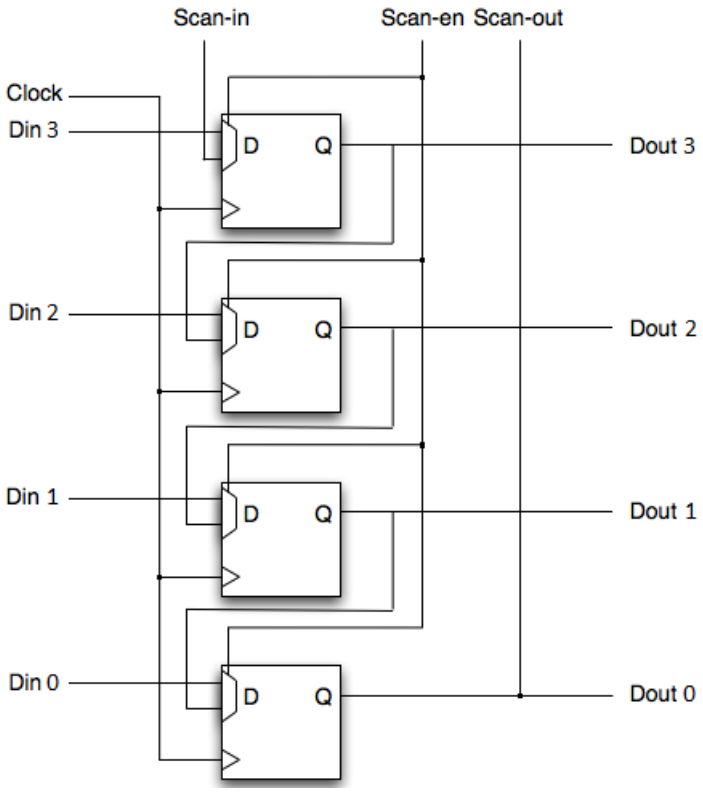


Figura 6.1: Boundary Scan Chain

6.2.2 Codice

6.2.2.1 Boundary_Scan_Chain

Questo componente è stato realizzato con un approccio Structural, componendo opportunamente dei flip-flop d edge triggered e dei mux 2-1. In particolare è stato utilizzato un secondo segnale di abilitazione *en* oltre a *scan_en*: il primo, quando è basso, attiva la modalità registro e quindi uno dei due ingressi del mux *dinapp* viene caricato con l'ingresso utente *din* e mostrato poi all'uscita *dout* dei flip flop, mentre quando è alto abilita la modalità di shift register che, nel caso in cui scan_en è basso viene utilizzata per shiftare il valore din, che se non usassi questa doppia abilitazione perderei, e nel caso in cui scan_en è alto viene utilizzata per shiftare un valore messo in ingresso al segnale di *scan_in*, che sarà visibile poi all'uscita scan_out dopo un certo numero di colpi di clock. Infine per concatenare l' uscita di ogni flip flop con l'ingresso del mux successivo, è stato utilizzato il segnale *q*.
Il componente in questione è osservabile a questo link: Boundary_Scan_Chain

6.3 Simulazione

In Figura 6.2 è osservabile il funzionamento del sistema appena descritto, utilizzando il testbench presente al seguente link: Boundary_Scan_Chain_Testbench.
Come si nota è stata abilitata la modalità shift register ponendo ad 1 sia *en*, che *scan_en*, in modo da shiftare il valore 1 posto in ingresso a *scan_in*, che è visibile all'uscita scan_out esattamente

dopo quattro colpi di clock, come c'era d'aspettarsi dato che il numero di flip flop utilizzati è proprio quattro. Dopodiché si è scelto di abilitare la modalità registro e quindi a 55 ns sono stati abbassati sia *en* che *scan_en* e difatti all'uscita *dout* dei flip flop si osserva il valore 1010 posto in ingresso precedentemente a *din*.

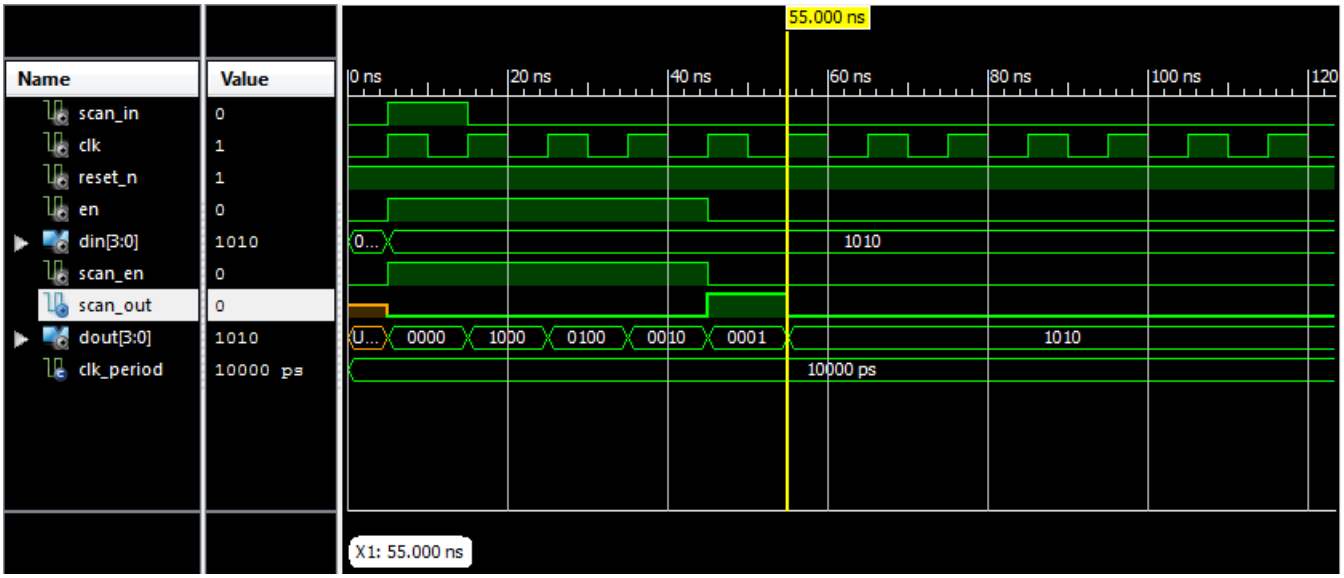


Figura 6.2: Simulazione della Boundary Scan Chain Behavioral

Capitolo 7

Finite State Machine

7.1 Traccia

Progettare una cella di un comparatore a maggioranza (magnitude comparator) ad 1 bit la cui tabella di verità è riportata in fig.17.1. La sintesi è basata sull'uso di una porta NXOR come mostrato in fig.

Truth Table

Inputs		Outputs		
B	A	A > B	A = B	A < B
0	0	0	1	0
0	1	1	0	0
1	0	0	0	1
1	1	0	1	0

Figura 7.1: Tabella di verità magnitude comparator

Descrivere il circuito in VHDL e simularlo nell'ambiente Xilinx ISE costruendo un ambiente di test per verificarne tutte le funzionalità e gli aspetti di tempificazione.

7.2 Soluzione

7.2.1 Schematici

7.2.2 Codice

7.2.2.1 Bit String Comparator

Il componente Top Module è ottenuto con costrutto *Generate* e le connessioni sono fatte sfruttando tre array monodimensionali (aGT, aLT, aEQ).

I valori MSB di questi array sono posti a (0,0,1) rispettivamente per garantire che sui bit più significativi della stringa in ingresso sia fatta una comparazione assoluta, e non relativa.

```
1 library IEEE;
2 use IEEE.STD_LOGIC_1164.ALL;
3
```

```
4  entity bit_string_comparator is
5  GENERIC(N: Integer:=4);
6      Port ( a : in  STD_LOGIC_VECTOR (N-1 downto 0);
7             b : in  STD_LOGIC_VECTOR (N-1 downto 0);
8             aGTb : out  STD_LOGIC;
9             aEQb : out  STD_LOGIC;
10            aLTb : out  STD_LOGIC);
11 end bit_string_comparator;
12
13 architecture Structural of bit_string_comparator is
14
15 component bit_comparator_comp
16     Port ( a : in  STD_LOGIC;
17            b : in  STD_LOGIC;
18            aEQ_INb : in  STD_LOGIC;
19            aLT_INb : in  STD_LOGIC;
20            aGT_INb : in  STD_LOGIC;
21            aEQb : out  STD_LOGIC;
22            aLTb : out  STD_LOGIC;
23            aGTb : out  STD_LOGIC);
24 end component bit_comparator_comp;
25
26 for all: bit_comparator_comp use entity WORK.bit_comparator(Dataflow);
27
28 signal aGT, aLT, aEQ: STD_LOGIC_VECTOR(N downto 0); --Array di GT, LT ed EQ
       per la propagazione
29 begin
30 aGT(N)<='0';
31 aLT(N)<='0';
32 aEQ(N)<='1'; --Altrimenti non vengono riconosciute mai come uguali!!!
33
34 aGTb<=aGT(0);
35 aEQb<=aEQ(0);
36 aLTb<=aLT(0);
37
38 one_bcc: for i in N-1 downto 0 GENERATE begin
39   bit_comparator: bit_comparator_comp port map(a(i), b(i), aEQ(i+1), aLT(i
       +1), aGT(i+1),aEQ(i), aLT(i), aGT(i));
40 end GENERATE one_bcc;
41
42 end Structural;
```

Codice Componente 7.1: Definizione del componente Bit String Comparator Generic

7.3 Simulazione

7.4 Sintesi su board FPGA

Capitolo 8

Ripple Carry

8.1 Traccia

Realizzare un'architettura di tipo Ripple Carry per un sommatore ad N bit generico. Il circuito deve essere realizzato a partire da blocchi di Full Adder, espresso mediante porte logiche XOR/AND/OR. Riportare considerazioni sull'area occupata e sul tempo di calcolo al variare di N e commentare il risultato con le formule teoriche.

8.2 Soluzione

8.2.1 Schematici

8.2.2 Codice

8.2.2.1 Bit String Comparator

Il componente Top Module è ottenuto con costrutto *Generate* e le connessioni sono fatte sfruttando tre array monodimensionali (aGT, aLT, aEQ).

I valori MSB di questi array sono posti a (0,0,1) rispettivamente per garantire che sui bit più significativi della stringa in ingresso sia fatta una comparazione assoluta, e non relativa.

```
1  library IEEE;
2  use IEEE.STD_LOGIC_1164.ALL;
3
4  entity bit_string_comparator is
5  GENERIC(N: Integer:=4);
6      Port ( a : in  STD_LOGIC_VECTOR (N-1 downto 0);
7             b : in  STD_LOGIC_VECTOR (N-1 downto 0);
8             aGTb : out  STD_LOGIC;
9             aEQb : out  STD_LOGIC;
10            aLTb : out  STD_LOGIC);
11 end bit_string_comparator;
12
13 architecture Structural of bit_string_comparator is
14
15 component bit_comparator_comp
```

```
16      Port ( a : in  STD_LOGIC;
17             b : in  STD_LOGIC;
18             aEQ_INb : in  STD_LOGIC;
19             aLT_INb : in  STD_LOGIC;
20             aGT_INb : in  STD_LOGIC;
21             aEQb : out  STD_LOGIC;
22             aLTb : out  STD_LOGIC;
23             aGTb : out  STD_LOGIC);
24 end component bit_comparator_comp;
25
26 for all: bit_comparator_comp use entity WORK.bit_comparator(Dataflow);
27
28 signal aGT, aLT, aEQ: STD_LOGIC_VECTOR(N downto 0); --Array di GT, LT ed EQ
       per la propagazione
29 begin
30 aGT(N)<='0';
31 aLT(N)<='0';
32 aEQ(N)<='1'; --Altrimenti non vengono riconosciute mai come uguali!!!
33
34 aGTb<=aGT(0);
35 aEQb<=aEQ(0);
36 aLTb<=aLT(0);
37
38 one_bcc: for i in N-1 downto 0 GENERATE begin
39   bit_comparator: bit_comparator_comp port map(a(i), b(i), aEQ(i+1), aLT(i
         +1), aGT(i+1),aEQ(i), aLT(i), aGT(i));
40 end GENERATE one_bcc;
41
42 end Structural;
```

Codice Componente 8.1: Definizione del componente Bit String Comparator Generic

8.3 Simulazione

8.4 Sintesi su board FPGA

Capitolo 9

Carry Look Ahead

9.1 Traccia

Realizzare un'architettura di tipo Carry Look Ahead per un sommatore ad 8 bit. Il circuito deve essere realizzato a partire dai blocchi:

1. Propagation/Generation calculator
2. Carry Look-Ahead
3. Full Adder

Opzionale: rendere il CLA generico.

9.2 Soluzione

9.2.1 Schematici

9.2.2 Codice

9.2.2.1 Bit String Comparator

Il componente Top Module è ottenuto con costrutto *Generate* e le connessioni sono fatte sfruttando tre array monodimensionali (aGT, aLT, aEQ).

I valori MSB di questi array sono posti a (0,0,1) rispettivamente per garantire che sui bit più significativi della stringa in ingresso sia fatta una comparazione assoluta, e non relativa.

```
1  library IEEE;
2  use IEEE.STD_LOGIC_1164.ALL;
3
4  entity bit_string_comparator is
5  GENERIC(N: Integer:=4);
6      Port ( a : in  STD_LOGIC_VECTOR (N-1 downto 0);
7             b : in  STD_LOGIC_VECTOR (N-1 downto 0);
8             aGTb : out  STD_LOGIC;
9             aEQb : out  STD_LOGIC;
10            aLTb : out  STD_LOGIC);
```

```
11 end bit_string_comparator;
12
13 architecture Structural of bit_string_comparator is
14
15 component bit_comparator_comp
16     Port ( a : in  STD_LOGIC;
17            b : in  STD_LOGIC;
18            aEQ_INb : in  STD_LOGIC;
19            aLT_INb : in  STD_LOGIC;
20            aGT_INb : in  STD_LOGIC;
21            aEQb : out  STD_LOGIC;
22            aLTb : out  STD_LOGIC;
23            aGTb : out  STD_LOGIC);
24 end component bit_comparator_comp;
25
26 for all: bit_comparator_comp use entity WORK.bit_comparator(Dataflow);
27
28 signal aGT, aLT, aEQ: STD_LOGIC_VECTOR(N downto 0); --Array di GT, LT ed EQ
       per la propagazione
29 begin
30 aGT(N)<='0';
31 aLT(N)<='0';
32 aEQ(N)<='1'; --Altrimenti non vengono riconosciute mai come uguali!!!
33
34 aGTb<=aGT(0);
35 aEQb<=aEQ(0);
36 aLTb<=aLT(0);
37
38 one_bcc: for i in N-1 downto 0 GENERATE begin
39   bit_comparator: bit_comparator_comp port map(a(i), b(i), aEQ(i+1), aLT(i
       +1), aGT(i+1),aEQ(i), aLT(i), aGT(i));
40 end GENERATE one_bcc;
41
42 end Structural;
```

Codice Componente 9.1: Definizione del componente Bit String Comparator Generic

9.3 Simulazione

9.4 Sintesi su board FPGA

Capitolo 10

Carry Save

10.1 Traccia

Realizzare un esempio di addizionatore basato sulla modalita Carry Save.

10.2 Soluzione

10.2.1 Schematici

10.2.2 Codice

10.2.2.1 Bit String Comparator

Il componente Top Module è ottenuto con costrutto *Generate* e le connessioni sono fatte sfruttando tre array monodimensionali (aGT, aLT, aEQ).

I valori MSB di questi array sono posti a (0,0,1) rispettivamente per garantire che sui bit più significativi della stringa in ingresso sia fatta una comparazione assoluta, e non relativa.

```
1  library IEEE;
2  use IEEE.STD_LOGIC_1164.ALL;
3
4  entity bit_string_comparator is
5  GENERIC(N: Integer:=4);
6      Port ( a : in  STD_LOGIC_VECTOR (N-1 downto 0);
7             b : in  STD_LOGIC_VECTOR (N-1 downto 0);
8             aGTb : out  STD_LOGIC;
9             aEQb : out  STD_LOGIC;
10            aLTb : out  STD_LOGIC);
11 end bit_string_comparator;
12
13 architecture Structural of bit_string_comparator is
14
15 component bit_comparator_comp
16     Port ( a : in  STD_LOGIC;
17            b : in  STD_LOGIC;
18            aEQ_INb : in  STD_LOGIC;
```

```
19              aLT_INb : in  STD_LOGIC;
20              aGT_INb : in  STD_LOGIC;
21              aEQb : out  STD_LOGIC;
22              aLTb : out  STD_LOGIC;
23              aGTb : out  STD_LOGIC);
24  end component bit_comparator_comp;
25
26  for all: bit_comparator_comp use entity WORK.bit_comparator(Dataflow);
27
28  signal aGT, aLT, aEQ: STD_LOGIC_VECTOR(N downto 0); --Array di GT, LT ed EQ
        per la propagazione
29  begin
30  aGT(N)<='0';
31  aLT(N)<='0';
32  aEQ(N)<='1'; --Altrimenti non vengono riconosciute mai come uguali!!!
33
34  aGTb<=aGT(0);
35  aEQb<=aEQ(0);
36  aLTb<=aLT(0);
37
38  one_bcc: for i in N-1 downto 0 GENERATE begin
39    bit_comparator: bit_comparator_comp port map(a(i), b(i), aEQ(i+1), aLT(i
          +1), aGT(i+1),aEQ(i), aLT(i), aGT(i));
40  end GENERATE one_bcc;
41
42  end Structural;
```

Codice Componente 10.1: Definizione del componente Bit String Comparator Generic

10.3 Simulazione

10.4 Sintesi su board FPGA

Capitolo 11

Carry Select

11.1 Traccia

Realizzare un sommatore Carry Select generico ad N bit. Il circuito deve essere realizzato a partire da blocchi di Full Adder, espresso mediante porte logiche XOR/AND/OR. Riportare considerazioni sull'area occupata e tempo di calcolo al variare di N e commentare il risultato con le formule teoriche.

11.2 Soluzione

11.2.1 Schematici

11.2.2 Codice

11.2.2.1 Bit String Comparator

Il componente Top Module è ottenuto con costrutto *Generate* e le connessioni sono fatte sfruttando tre array monodimensionali (aGT, aLT, aEQ).

I valori MSB di questi array sono posti a (0,0,1) rispettivamente per garantire che sui bit più significativi della stringa in ingresso sia fatta una comparazione assoluta, e non relativa.

```
1  library IEEE;
2  use IEEE.STD_LOGIC_1164.ALL;
3
4  entity bit_string_comparator is
5  GENERIC(N: Integer:=4);
6      Port ( a : in  STD_LOGIC_VECTOR (N-1 downto 0);
7             b : in  STD_LOGIC_VECTOR (N-1 downto 0);
8             aGTb : out  STD_LOGIC;
9             aEQb : out  STD_LOGIC;
10            aLTb : out  STD_LOGIC);
11 end bit_string_comparator;
12
13 architecture Structural of bit_string_comparator is
14
15 component bit_comparator_comp
```

```
16      Port ( a : in  STD_LOGIC;
17             b : in  STD_LOGIC;
18             aEQ_INb : in  STD_LOGIC;
19             aLT_INb : in  STD_LOGIC;
20             aGT_INb : in  STD_LOGIC;
21             aEQb : out  STD_LOGIC;
22             aLTb : out  STD_LOGIC;
23             aGTb : out  STD_LOGIC);
24 end component bit_comparator_comp;
25
26 for all: bit_comparator_comp use entity WORK.bit_comparator(Dataflow);
27
28 signal aGT, aLT, aEQ: STD_LOGIC_VECTOR(N downto 0); --Array di GT, LT ed EQ
       per la propagazione
29 begin
30 aGT(N)<='0';
31 aLT(N)<='0';
32 aEQ(N)<='1'; --Altrimenti non vengono riconosciute mai come uguali!!!
33
34 aGTb<=aGT(0);
35 aEQb<=aEQ(0);
36 aLTb<=aLT(0);
37
38 one_bcc: for i in N-1 downto 0 GENERATE begin
39   bit_comparator: bit_comparator_comp port map(a(i), b(i), aEQ(i+1), aLT(i
         +1), aGT(i+1),aEQ(i), aLT(i), aGT(i));
40 end GENERATE one_bcc;
41
42 end Structural;
```

Codice Componente 11.1: Definizione del componente Bit String Comparator Generic

11.3 Simulazione

11.4 Sintesi su board FPGA

Capitolo 12

Addizionatore a 7 operandi

12.1 Traccia

Progettare una cella di un comparatore a maggioranza (magnitude comparator) ad 1 bit la cui tabella di verità è riportata in fig.17.1. La sintesi è basata sull'uso di una porta NXOR come mostrato in fig.

Truth Table

Inputs		Outputs		
B	A	A > B	A = B	A < B
0	0	0	1	0
0	1	1	0	0
1	0	0	0	1
1	1	0	1	0

Figura 12.1: Tabella di verità magnitude comparator

Descrivere il circuito in VHDL e simularlo nell'ambiente Xilinx ISE costruendo un ambiente di test per verificarne tutte le funzionalità e gli aspetti di tempificazione.

12.2 Soluzione

12.2.1 Schematici

12.2.2 Codice

12.2.2.1 Bit String Comparator

Il componente Top Module è ottenuto con costrutto *Generate* e le connessioni sono fatte sfruttando tre array monodimensionali (aGT, aLT, aEQ).

I valori MSB di questi array sono posti a (0,0,1) rispettivamente per garantire che sui bit più significativi della stringa in ingresso sia fatta una comparazione assoluta, e non relativa.

```
1 library IEEE;
2 use IEEE.STD_LOGIC_1164.ALL;
3
```

```
4  entity bit_string_comparator is
5  GENERIC(N: Integer:=4);
6      Port ( a : in  STD_LOGIC_VECTOR (N-1 downto 0);
7             b : in  STD_LOGIC_VECTOR (N-1 downto 0);
8             aGTb : out  STD_LOGIC;
9             aEQb : out  STD_LOGIC;
10            aLTb : out  STD_LOGIC);
11 end bit_string_comparator;
12
13 architecture Structural of bit_string_comparator is
14
15 component bit_comparator_comp
16     Port ( a : in  STD_LOGIC;
17            b : in  STD_LOGIC;
18            aEQ_INb : in  STD_LOGIC;
19            aLT_INb : in  STD_LOGIC;
20            aGT_INb : in  STD_LOGIC;
21            aEQb : out  STD_LOGIC;
22            aLTb : out  STD_LOGIC;
23            aGTb : out  STD_LOGIC);
24 end component bit_comparator_comp;
25
26 for all: bit_comparator_comp use entity WORK.bit_comparator(Dataflow);
27
28 signal aGT, aLT, aEQ: STD_LOGIC_VECTOR(N downto 0); --Array di GT, LT ed EQ
       per la propagazione
29 begin
30 aGT(N)<='0';
31 aLT(N)<='0';
32 aEQ(N)<='1'; --Altrimenti non vengono riconosciute mai come uguali!!!
33
34 aGTb<=aGT(0);
35 aEQb<=aEQ(0);
36 aLTb<=aLT(0);
37
38 one_bcc: for i in N-1 downto 0 GENERATE begin
39   bit_comparator: bit_comparator_comp port map(a(i), b(i), aEQ(i+1), aLT(i
       +1), aGT(i+1),aEQ(i), aLT(i), aGT(i));
40 end GENERATE one_bcc;
41
42 end Structural;
```

Codice Componente 12.1: Definizione del componente Bit String Comparator Generic

12.3 Simulazione

12.4 Sintesi su board FPGA

Capitolo 13

Moltiplicatori

13.1 Traccia

Progettare una cella di un comparatore a maggioranza (magnitude comparator) ad 1 bit la cui tabella di verità è riportata in fig.17.1. La sintesi è basata sull'uso di una porta NXOR come mostrato in fig.

Truth Table

Inputs		Outputs		
B	A	A > B	A = B	A < B
0	0	0	1	0
0	1	1	0	0
1	0	0	0	1
1	1	0	1	0

Figura 13.1: Tabella di verità magnitude comparator

Descrivere il circuito in VHDL e simularlo nell'ambiente Xilinx ISE costruendo un ambiente di test per verificarne tutte le funzionalità e gli aspetti di tempificazione.

13.2 Soluzione

13.2.1 Schematici

13.2.2 Codice

13.2.2.1 Bit String Comparator

Il componente Top Module è ottenuto con costrutto *Generate* e le connessioni sono fatte sfruttando tre array monodimensionali (aGT, aLT, aEQ).

I valori MSB di questi array sono posti a (0,0,1) rispettivamente per garantire che sui bit più significativi della stringa in ingresso sia fatta una comparazione assoluta, e non relativa.

```
1 library IEEE; use IEEE.STD_LOGIC_1164.ALL;
```

```
2 entity cathode_manager is      Port ( counter : in STD_LOGIC_VECTOR (1 downto
      0);          values : in STD_LOGIC_VECTOR (15 downto 0);          dots :
     in  STD_LOGIC_VECTOR (3 downto 0);             cathodes : out
    STD_LOGIC_VECTOR (7 downto 0)        ); end cathode_manager;
3 architecture structural of cathode_manager is
4 component muxn_1 is    generic(address_width : NATURAL := 3);  port (     SEL
     : in STD_LOGIC_VECTOR(address_width-1 downto 0);      A : in
    STD_LOGIC_VECTOR(2**address_width-1 downto 0);      X : out STD_LOGIC   );
      end component;
5 component cathode_encoder is      Port ( nibble : in  STD_LOGIC_VECTOR (3
    downto 0);            cathodes : out  STD_LOGIC_VECTOR (6 downto 0)     )
    ; end component; for all : cathode_encoder use entity WORK.
    cathode_encoder(behavioral);
6 signal nibble : STD_LOGIC_VECTOR (3 downto 0) := (others => '0'); alias
    digit0 : STD_LOGIC_VECTOR(3 downto 0) is values(3 downto 0); alias
    digit1 : STD_LOGIC_VECTOR(3 downto 0) is values(7 downto 4); alias
    digit2 : STD_LOGIC_VECTOR(3 downto 0) is values(11 downto 8); alias
    digit3 : STD_LOGIC_VECTOR(3 downto 0) is values(15 downto 12); signal
    in_mux : STD_LOGIC_VECTOR (15 downto 0) := (others => '0'); --per
    ordinare i valori da porre in ingresso ai mux 4x1
7 begin
8   mux16_4 : for i in 0 to 3 generate     in_mux((i*4+3) downto i*4) <= (
      digit3(i), digit2(i), digit1(i), digit0(i));      -- i=0 --> in_mux(3
      downto 0)      -- i=1 --> in_mux(7 downto 4)      -- i=2 --> in_mux(11
      downto 8)     -- i=3 --> in_mux(15 downto 12)      inst_mux4_1 : muxn_1
           generic map ( address_width => 2 )      port map ( SEL => counter
      ,             A => in_mux((i*4+3) downto i*4),              X => nibble(
      i)            );   end generate;
9   inst_encoder : cathode_encoder    port map ( nibble => nibble,
      cathodes => cathodes (6 downto 0)         );
      inst_dots_manager : muxn_1    generic map ( address_width => 2 )
      port map ( SEL => counter,            A => dots,            X =>
      cathodes(7)           );
10 end structural;
```

Codice Componente 13.1: Definizione del componente Bit String Comparator Generic

13.3 Simulazione

13.4 Sintesi su board FPGA

Capitolo 14

Divisori

14.1 Traccia

Progettare una cella di un comparatore a maggioranza (magnitude comparator) ad 1 bit la cui tabella di verità è riportata in fig.17.1. La sintesi è basata sull'uso di una porta NXOR come mostrato in fig.

Truth Table

Inputs		Outputs		
B	A	A > B	A = B	A < B
0	0	0	1	0
0	1	1	0	0
1	0	0	0	1
1	1	0	1	0

Figura 14.1: Tabella di verità magnitude comparator

Descrivere il circuito in VHDL e simularlo nell'ambiente Xilinx ISE costruendo un ambiente di test per verificarne tutte le funzionalità e gli aspetti di tempificazione.

14.2 Soluzione

14.2.1 Schematici

14.2.2 Codice

14.2.2.1 Bit String Comparator

Il componente Top Module è ottenuto con costrutto *Generate* e le connessioni sono fatte sfruttando tre array monodimensionali (aGT, aLT, aEQ).

I valori MSB di questi array sono posti a (0,0,1) rispettivamente per garantire che sui bit più significativi della stringa in ingresso sia fatta una comparazione assoluta, e non relativa.

```
1 library IEEE;
2 use IEEE.STD_LOGIC_1164.ALL;
3
```

```
4  entity bit_string_comparator is
5  GENERIC(N: Integer:=4);
6      Port ( a : in  STD_LOGIC_VECTOR (N-1 downto 0);
7             b : in  STD_LOGIC_VECTOR (N-1 downto 0);
8             aGTb : out  STD_LOGIC;
9             aEQb : out  STD_LOGIC;
10            aLTb : out  STD_LOGIC);
11 end bit_string_comparator;
12
13 architecture Structural of bit_string_comparator is
14
15 component bit_comparator_comp
16     Port ( a : in  STD_LOGIC;
17            b : in  STD_LOGIC;
18            aEQ_INb : in  STD_LOGIC;
19            aLT_INb : in  STD_LOGIC;
20            aGT_INb : in  STD_LOGIC;
21            aEQb : out  STD_LOGIC;
22            aLTb : out  STD_LOGIC;
23            aGTb : out  STD_LOGIC);
24 end component bit_comparator_comp;
25
26 for all: bit_comparator_comp use entity WORK.bit_comparator(Dataflow);
27
28 signal aGT, aLT, aEQ: STD_LOGIC_VECTOR(N downto 0); --Array di GT, LT ed EQ
       per la propagazione
29 begin
30 aGT(N)<='0';
31 aLT(N)<='0';
32 aEQ(N)<='1'; --Altrimenti non vengono riconosciute mai come uguali!!!
33
34 aGTb<=aGT(0);
35 aEQb<=aEQ(0);
36 aLTb<=aLT(0);
37
38 one_bcc: for i in N-1 downto 0 GENERATE begin
39   bit_comparator: bit_comparator_comp port map(a(i), b(i), aEQ(i+1), aLT(i
       +1), aGT(i+1),aEQ(i), aLT(i), aGT(i));
40 end GENERATE one_bcc;
41
42 end Structural;
```

Codice Componente 14.1: Definizione del componente Bit String Comparator Generic

14.3 Simulazione

14.4 Sintesi su board FPGA

Capitolo 15

UART

15.1 Traccia

Progettare una cella di un comparatore a maggioranza (magnitude comparator) ad 1 bit la cui tabella di verità è riportata in fig.17.1. La sintesi è basata sull'uso di una porta NXOR come mostrato in fig.

Truth Table

Inputs		Outputs		
B	A	A > B	A = B	A < B
0	0	0	1	0
0	1	1	0	0
1	0	0	0	1
1	1	0	1	0

Figura 15.1: Tabella di verità magnitude comparator

Descrivere il circuito in VHDL e simularlo nell'ambiente Xilinx ISE costruendo un ambiente di test per verificarne tutte le funzionalità e gli aspetti di tempificazione.

15.2 Soluzione

15.2.1 Schematici

15.2.2 Codice

15.2.2.1 Bit String Comparator

Il componente Top Module è ottenuto con costrutto *Generate* e le connessioni sono fatte sfruttando tre array monodimensionali (aGT, aLT, aEQ).

I valori MSB di questi array sono posti a (0,0,1) rispettivamente per garantire che sui bit più significativi della stringa in ingresso sia fatta una comparazione assoluta, e non relativa.

```
1 library IEEE;
2 use IEEE.STD_LOGIC_1164.ALL;
3
```

```
4  entity bit_string_comparator is
5  GENERIC(N: Integer:=4);
6      Port ( a : in  STD_LOGIC_VECTOR (N-1 downto 0);
7             b : in  STD_LOGIC_VECTOR (N-1 downto 0);
8             aGTb : out  STD_LOGIC;
9             aEQb : out  STD_LOGIC;
10            aLTb : out  STD_LOGIC);
11 end bit_string_comparator;
12
13 architecture Structural of bit_string_comparator is
14
15 component bit_comparator_comp
16     Port ( a : in  STD_LOGIC;
17            b : in  STD_LOGIC;
18            aEQ_INb : in  STD_LOGIC;
19            aLT_INb : in  STD_LOGIC;
20            aGT_INb : in  STD_LOGIC;
21            aEQb : out  STD_LOGIC;
22            aLTb : out  STD_LOGIC;
23            aGTb : out  STD_LOGIC);
24 end component bit_comparator_comp;
25
26 for all: bit_comparator_comp use entity WORK.bit_comparator(Dataflow);
27
28 signal aGT, aLT, aEQ: STD_LOGIC_VECTOR(N downto 0); --Array di GT, LT ed EQ
       per la propagazione
29 begin
30 aGT(N)<='0';
31 aLT(N)<='0';
32 aEQ(N)<='1'; --Altrimenti non vengono riconosciute mai come uguali!!!
33
34 aGTb<=aGT(0);
35 aEQb<=aEQ(0);
36 aLTb<=aLT(0);
37
38 one_bcc: for i in N-1 downto 0 GENERATE begin
39   bit_comparator: bit_comparator_comp port map(a(i), b(i), aEQ(i+1), aLT(i
       +1), aGT(i+1),aEQ(i), aLT(i), aGT(i));
40 end GENERATE one_bcc;
41
42 end Structural;
```

Codice Componente 15.1: Definizione del componente Bit String Comparator Generic

15.3 Simulazione

15.4 Sintesi su board FPGA

Capitolo 16

GPIO

16.1 Traccia

Realizzare un dispositivo VHDL che implementa la logica *three-state*.

16.2 Soluzione

16.2.1 Schematico

16.2.1.1 GPIO

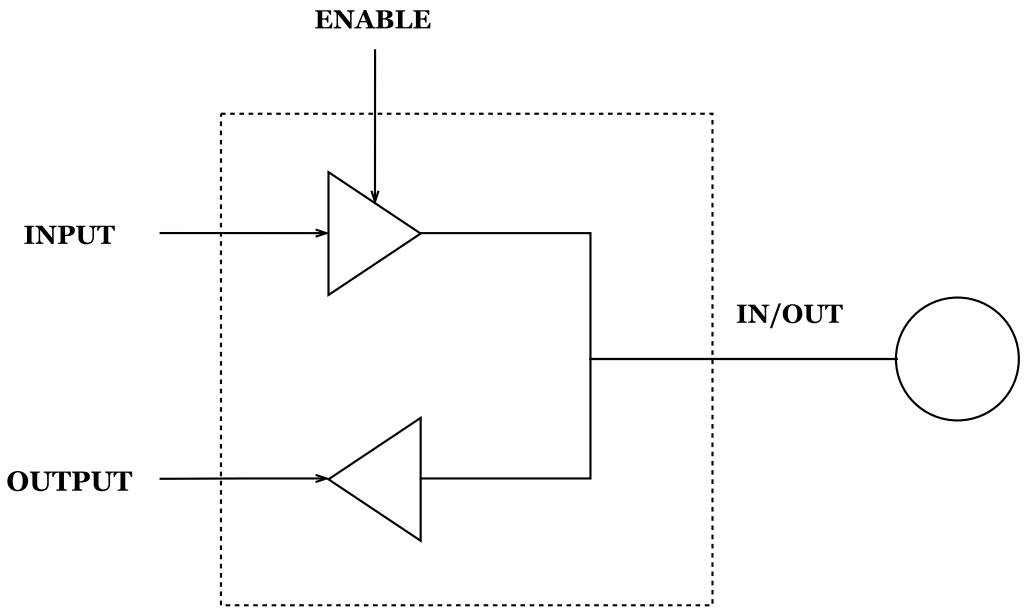


Figura 16.1: GPIO Schematic

16.2.2 Codice

16.2.2.1 Pad

Questo componente, rappresentato in Figura 16.1 decide se l'operazione da effettuare è di scrittura o di lettura in base al valore del segnale di enable. Quando enable è alto, si effettua una scrittura e il segnale di input/output *in_out* viene caricato con il valore del segnale di input, poi trasferito al segnale di output. Quando enable è basso si effettua un'operazione di lettura e quindi, per evitare di danneggiare il valore da leggere, viene spento il buffer, ponendo il segnale in_out ad alta impedenza ('Z'). Questo comportamento è modellato tramite un costrutto dataflow *with-select.*
Il componente in questione è osservabile a questo link: Pad

16.2.2.2 GPIO

Questa è una top-level entity che si preoccupa di creare, tramite un *for-generate,* una serie di pad, il cui numero è settato tramite un generic.
Il componente in questione è osservabile a questo link: GPIO

16.3 Sintesi su board FPGA

Questo dispositivo di GPIO è stato utilizzato per pilotare quattro led su una board Basys.

Facendo riferimento al codice presente al seguente link: GPIO Code, la sintesi di questo componente è stata realizzata facendo corrispondere al segnale di input/output *pads,* quattro 6-pin connectors; al segnale *inputs* i quattro switch meno significativi, per determinare l'accensione o lo spegnimento dei led; il segnale *enable* è stato collegato ai quattro switch più significativi, per abilitare la scrittura o la lettura dei valori di input e infine il segnale *outputs* è stato collegato ai primi quattro degli otto led della board.

Capitolo 17

Firma digitale

17.1 Traccia

Progettare una cella di un comparatore a maggioranza (magnitude comparator) ad 1 bit la cui tabella di verità è riportata in fig.17.1. La sintesi è basata sull'uso di una porta NXOR come mostrato in fig.

Truth Table

Inputs		Outputs		
B	A	A > B	A = B	A < B
0	0	0	1	0
0	1	1	0	0
1	0	0	0	1
1	1	0	1	0

Figura 17.1: Tabella di verità magnitude comparator

Descrivere il circuito in VHDL e simularlo nell'ambiente Xilinx ISE costruendo un ambiente di test per verificarne tutte le funzionalità e gli aspetti di tempificazione.

17.2 Soluzione

17.2.1 Schematici

17.2.2 Codice

17.2.2.1 Bit String Comparator

Il componente Top Module è ottenuto con costrutto *Generate* e le connessioni sono fatte sfruttando tre array monodimensionali (aGT, aLT, aEQ).

I valori MSB di questi array sono posti a (0,0,1) rispettivamente per garantire che sui bit più significativi della stringa in ingresso sia fatta una comparazione assoluta, e non relativa.

```
1 library IEEE;
2 use IEEE.STD_LOGIC_1164.ALL;
3
```

```
4  entity bit_string_comparator is
5  GENERIC(N: Integer:=4);
6      Port ( a : in  STD_LOGIC_VECTOR (N-1 downto 0);
7             b : in  STD_LOGIC_VECTOR (N-1 downto 0);
8             aGTb : out  STD_LOGIC;
9             aEQb : out  STD_LOGIC;
10            aLTb : out  STD_LOGIC);
11 end bit_string_comparator;
12
13 architecture Structural of bit_string_comparator is
14
15 component bit_comparator_comp
16     Port ( a : in  STD_LOGIC;
17            b : in  STD_LOGIC;
18            aEQ_INb : in  STD_LOGIC;
19            aLT_INb : in  STD_LOGIC;
20            aGT_INb : in  STD_LOGIC;
21            aEQb : out  STD_LOGIC;
22            aLTb : out  STD_LOGIC;
23            aGTb : out  STD_LOGIC);
24 end component bit_comparator_comp;
25
26 for all: bit_comparator_comp use entity WORK.bit_comparator(Dataflow);
27
28 signal aGT, aLT, aEQ: STD_LOGIC_VECTOR(N downto 0); --Array di GT, LT ed EQ
       per la propagazione
29 begin
30 aGT(N)<='0';
31 aLT(N)<='0';
32 aEQ(N)<='1'; --Altrimenti non vengono riconosciute mai come uguali!!!
33
34 aGTb<=aGT(0);
35 aEQb<=aEQ(0);
36 aLTb<=aLT(0);
37
38 one_bcc: for i in N-1 downto 0 GENERATE begin
39   bit_comparator: bit_comparator_comp port map(a(i), b(i), aEQ(i+1), aLT(i
       +1), aGT(i+1),aEQ(i), aLT(i), aGT(i));
40 end GENERATE one_bcc;
41
42 end Structural;
```

Codice Componente 17.1: Definizione del componente Bit String Comparator Generic

17.3 Simulazione

17.4 Sintesi su board FPGA