Tesina di Architettura dei Sistemi di Elaborazione
Gruppo 4

Milo Saverio - Mat. M63/0773 Pommella Michele - Mat. M63/0790
Trimaldi Davide - Mat. M63/0799

29 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

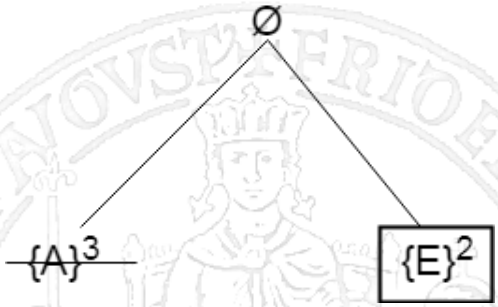
implicante	x	y	z	v	check
0,1,8,9	-	0	0	-	e
0,2,8,10	-	0	-	0	f
8,9,10,11	1	0	-	-	g

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:

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

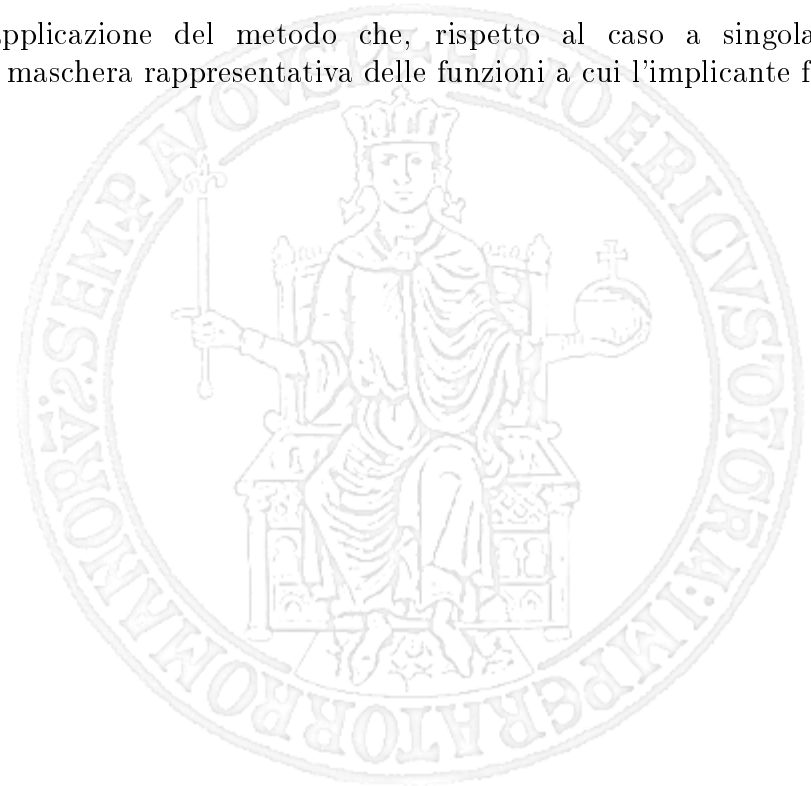
1.2.4 Minimizzazione 4

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

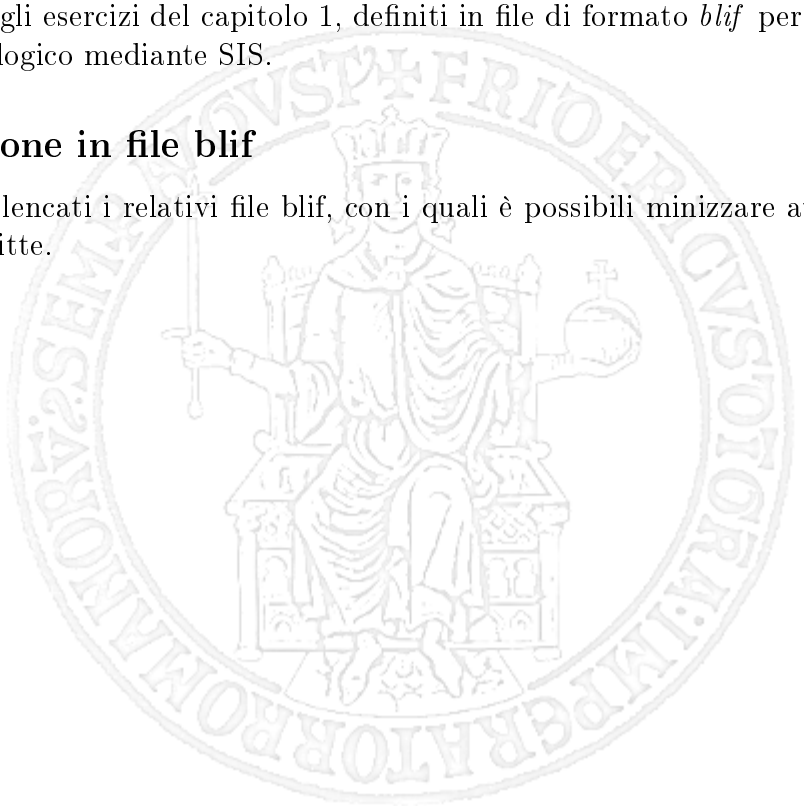
Utilizzare lo strumento automatico SIS per la minimizzazione di circuiti combinatori e il relativo mapping tecnologico.

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

```
.model esercizio2
.inputs a b c d
.outputs f

.names a b c d f
0000 1
0001 1
0010 1
0111 1
1000 1
1010 1
1111 1

.exdc
.inputs a b c d
.outputs f

.names a b c d f
0101 1
1001 1
1011 1

.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, 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 progetto ISE del componente in questione è osservabile a questo link: RS_Latch ISE

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.

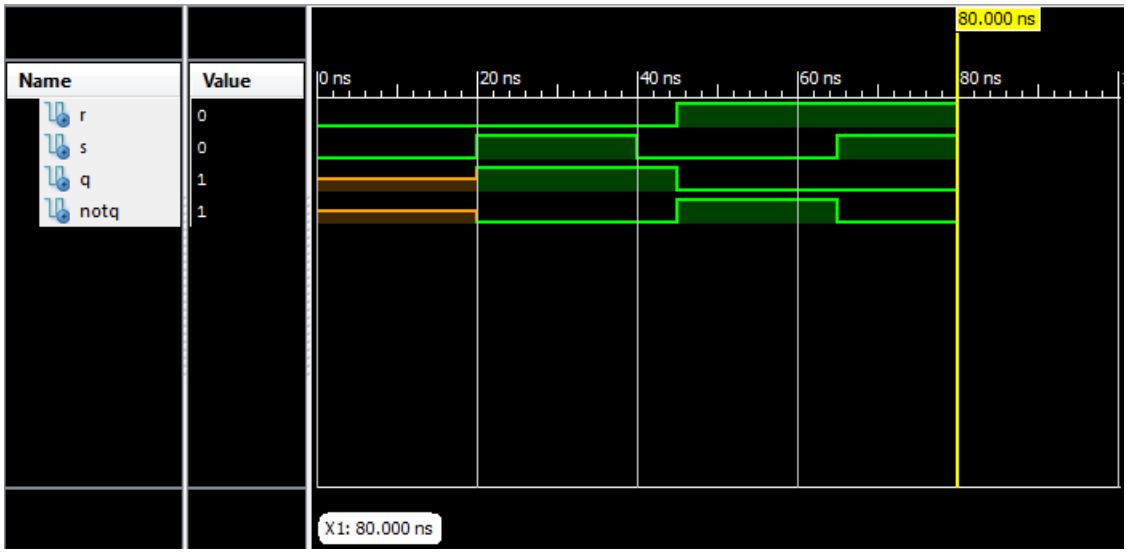


Figura 3.1: Simulazione del latch RS Behavioral

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.

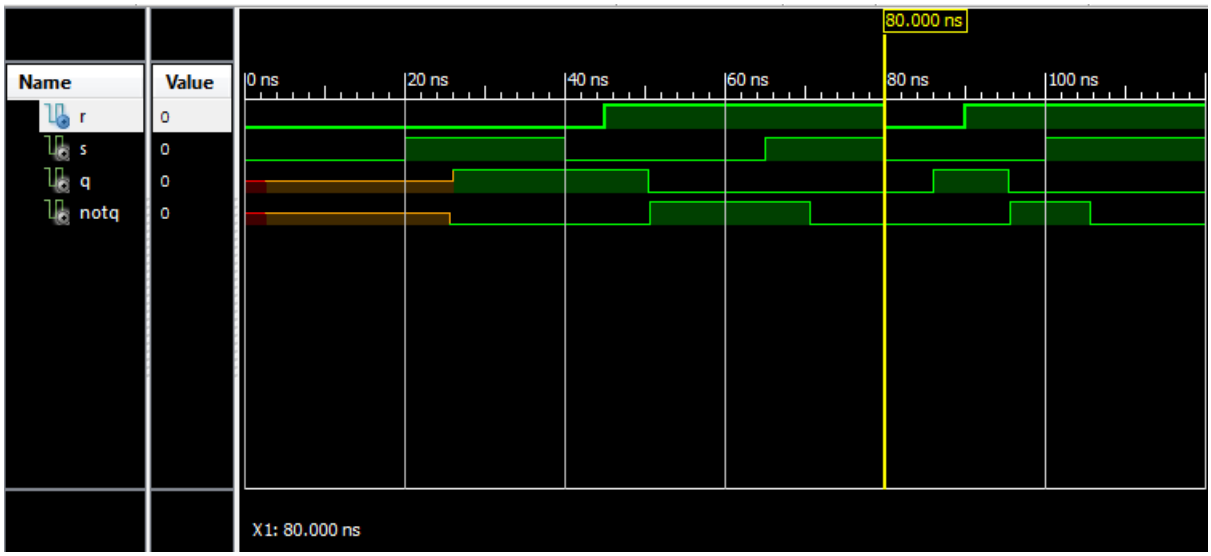


Figura 3.2: Simulazione del latch RS Post-Route

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 progetto ISE del componente in questione è osservabile a questo link: RS_Latch_Clocked ISE

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.

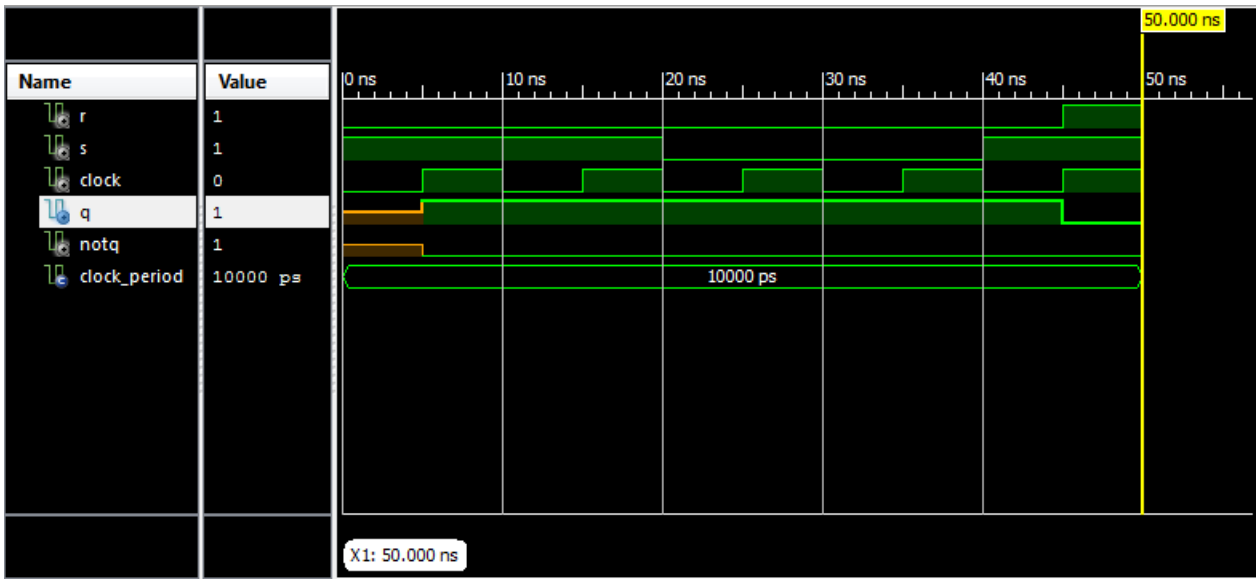


Figura 3.3: Simulazione del latch RS abilitato Behavioral

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.

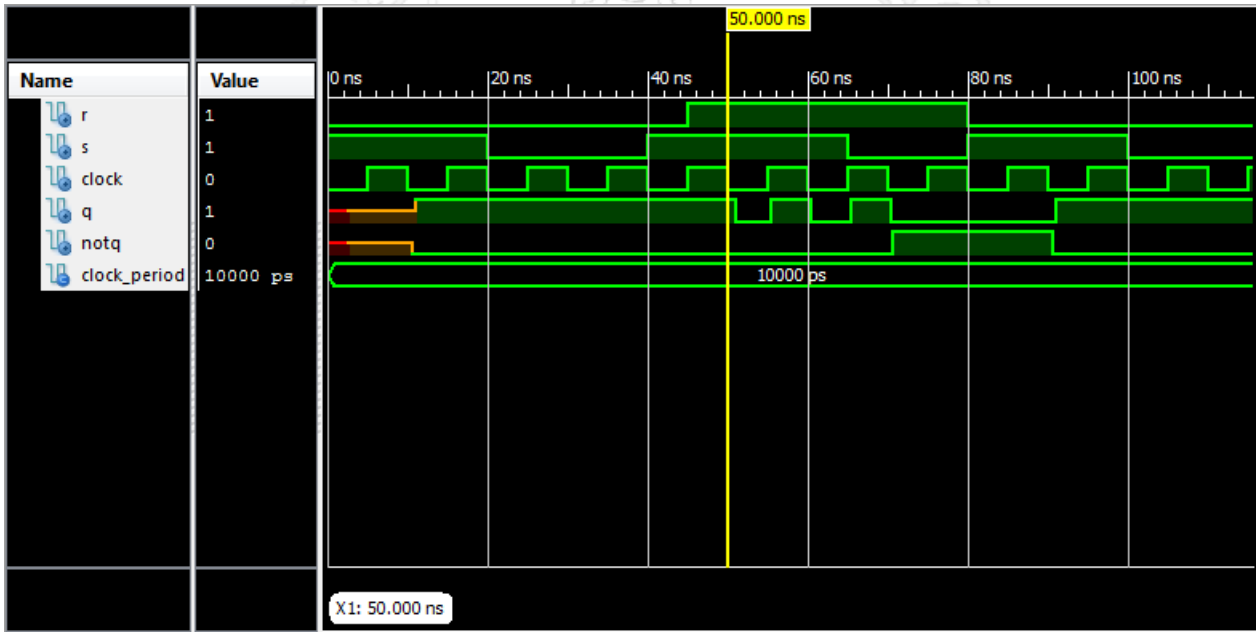


Figura 3.4: Simulazione del latch RS abilitato Post-Route

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 vengono fatti corrispondere 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 progetto ISE del componente in questione è osservabile a questo link: D_latch_Clocked ISE.

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.

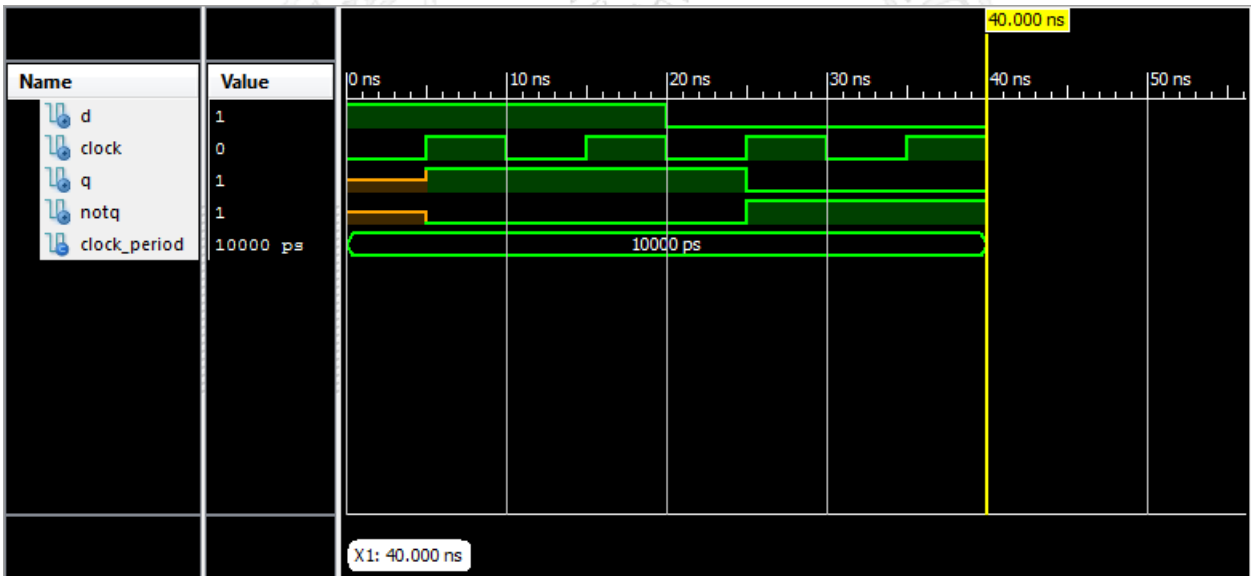


Figura 3.5: Simulazione del latch D abilitato Behavioral

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.

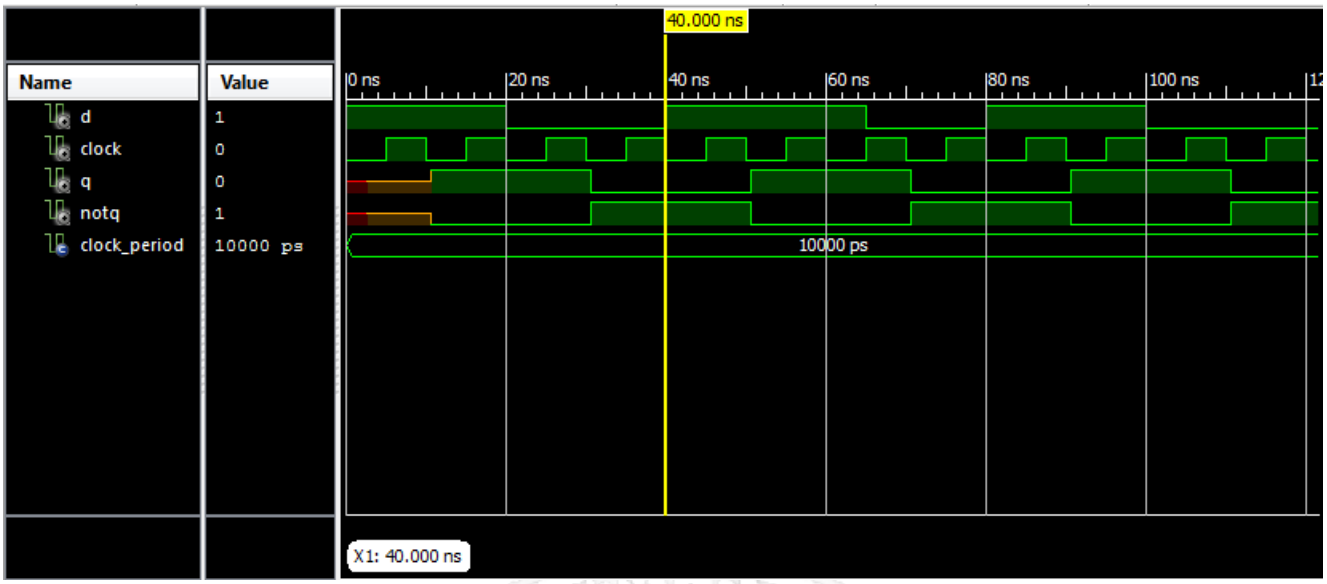


Figura 3.6: Simulazione del latch D abilitato Post-Route

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 progetto ISE del componente in questione è osservabile a questo link: T_Latch ISE

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.

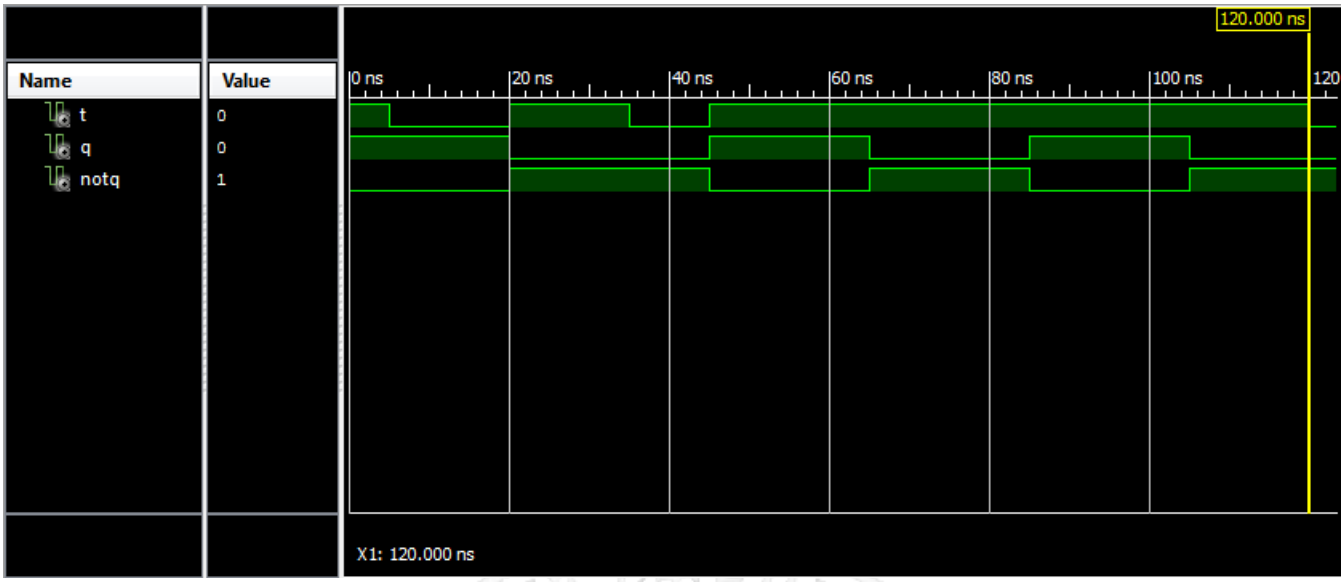


Figura 3.7: Simulazione del latch T Behavioral

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_Clocked

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 progetto ISE del componente in questione è osservabile a questo link: JK_Latch ISE

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.

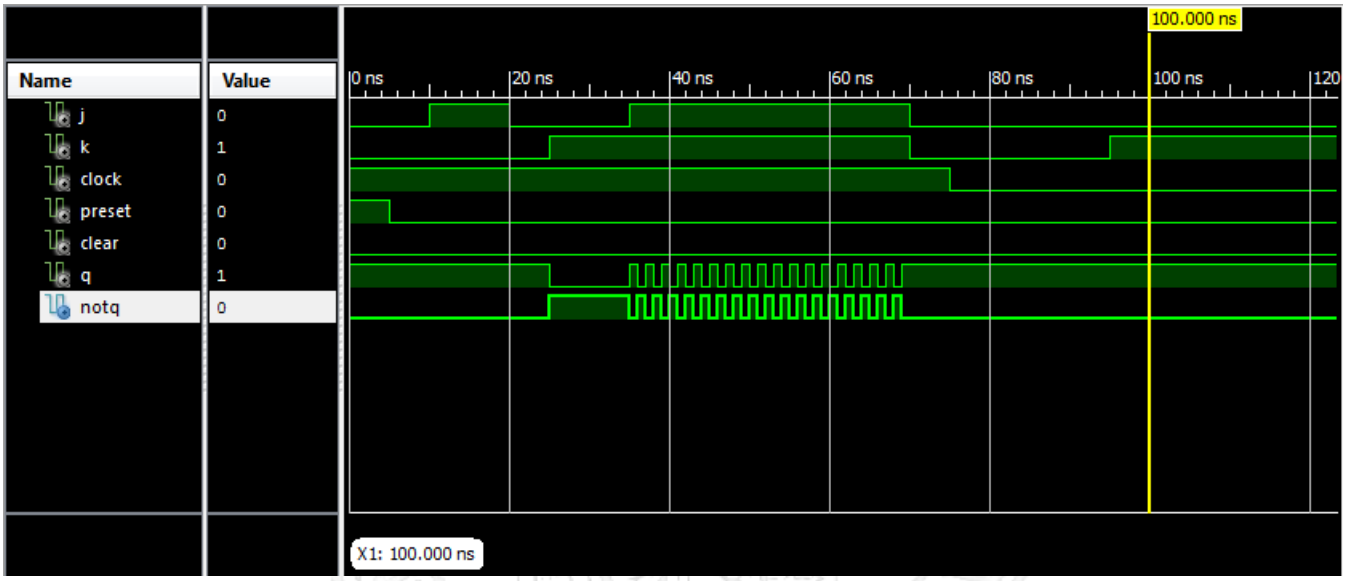


Figura 3.8: Simulazione del latch JK abilitato Behavioral

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.

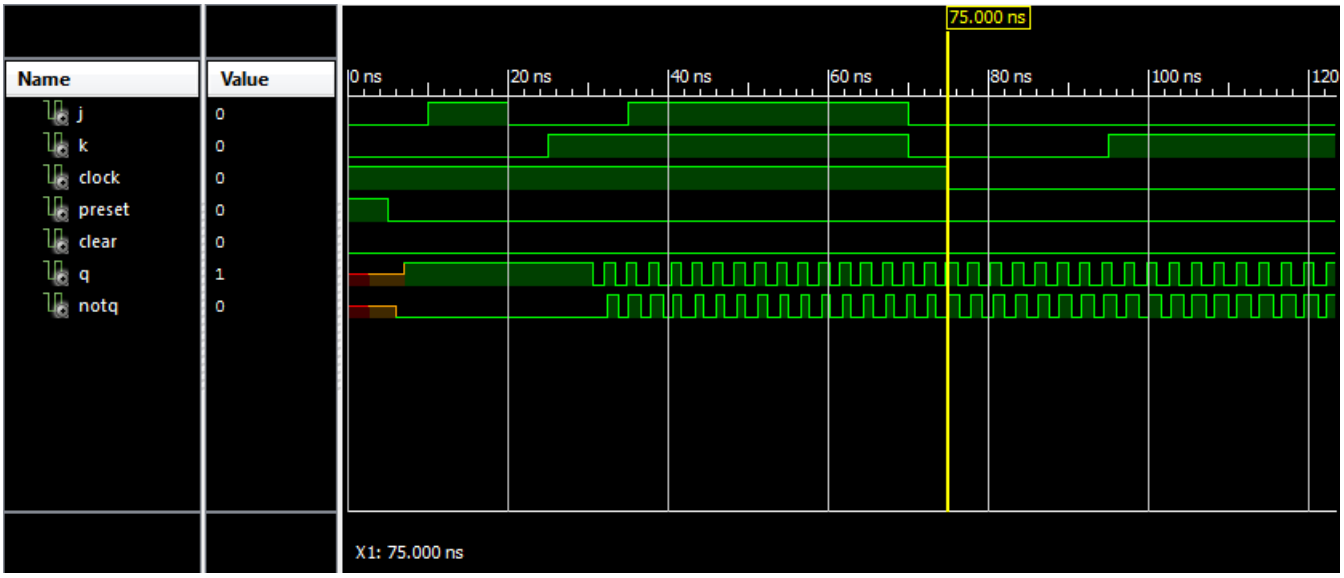


Figura 3.9: Simulazione del latch JK abilitato Post-Route

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.

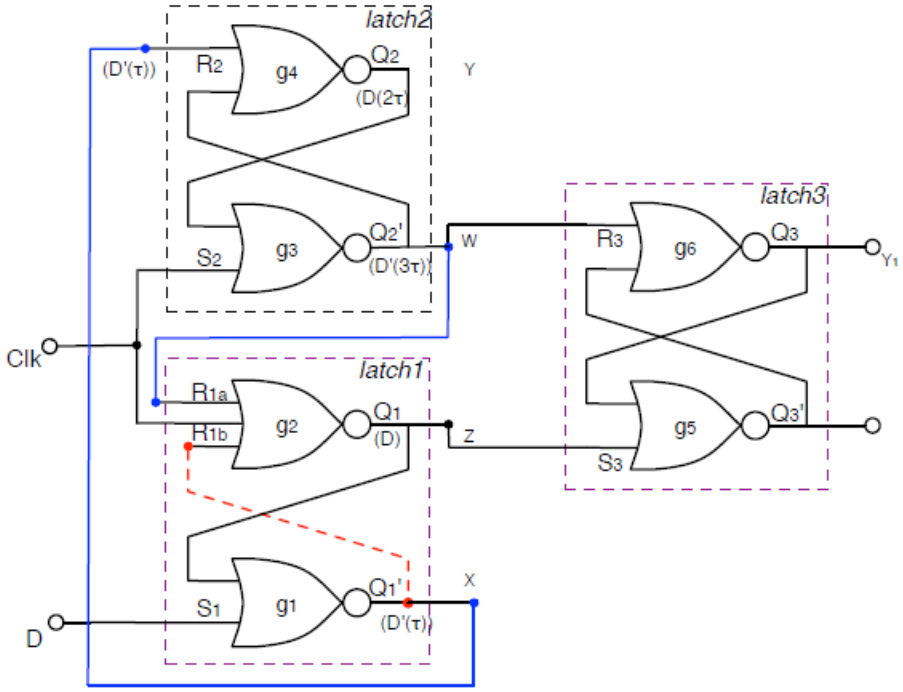


Figura 3.10: Schematico Flip-Flop D Edge Triggered

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 progetto ISE del componente in questione è osservabile a questo link: FlipFlop_D_Edge ISE

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.

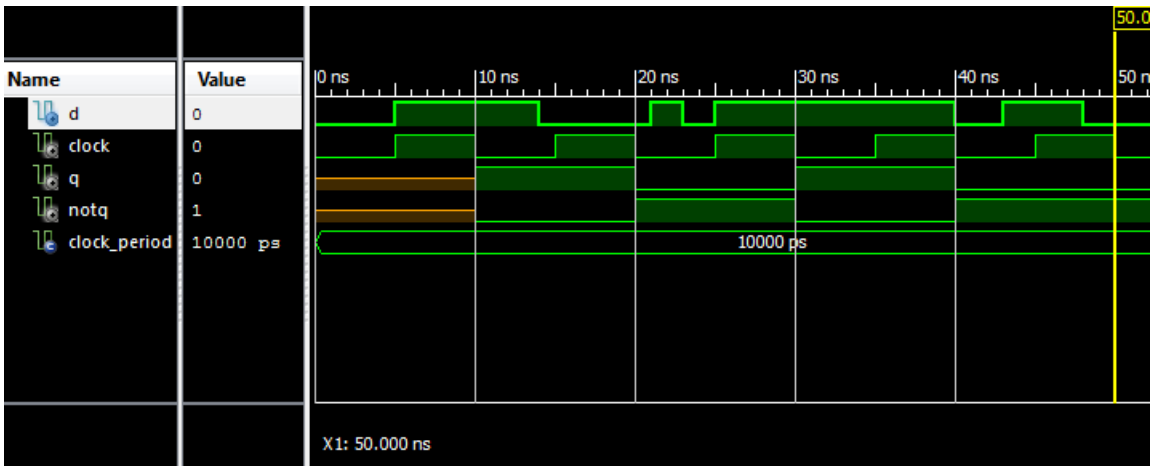


Figura 3.11: Simulazione del Flip-Flop D Edge Triggered Behavioral

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.

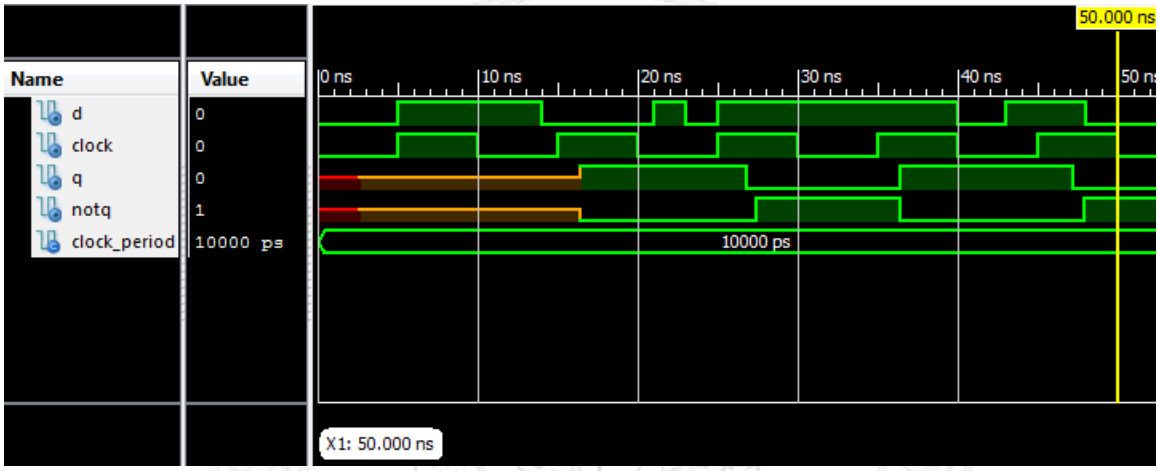


Figura 3.12: Simulazione del Flip-Flop D Edge Triggered Post-Route

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 ISE

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.

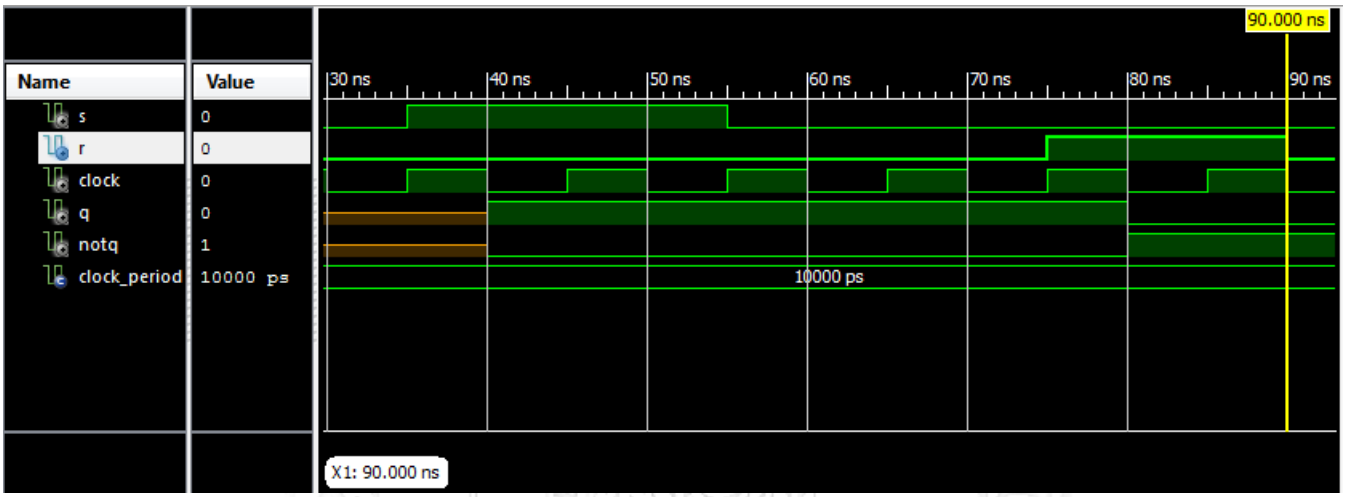


Figura 3.13: Simulazione del Flip-Flop RS Master-Slave Behavioral

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.

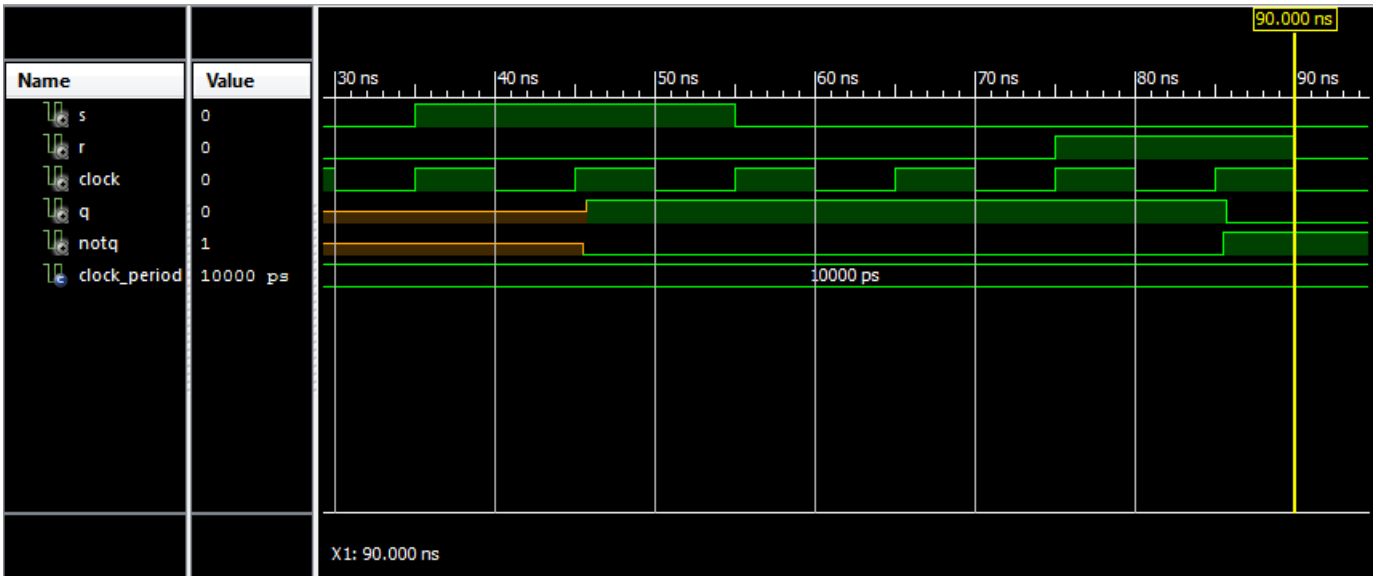


Figura 3.14: Simulazione del Flip-Flop RS Master-Slave Post-Route

Capitolo 4

Display a 7 segmenti

4.1 Traccia

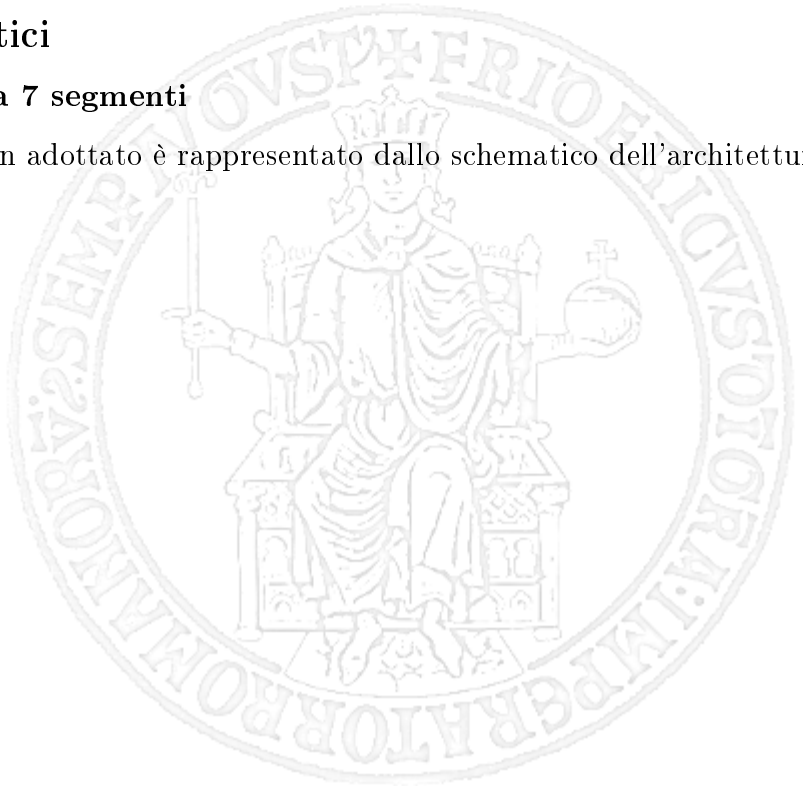
Illustrare la realizzazione di un'architettura che consenta di mostrare su un array di 4 display a 7 segmenti un valore intero. Tale puo essere una parola da 16 bit, composta cioè di 4 cifre esadecimali, ciascuna espressa su di un nibble (4 bit). Sviluppare la traccia discutendo l'approccio di design 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 5.1.



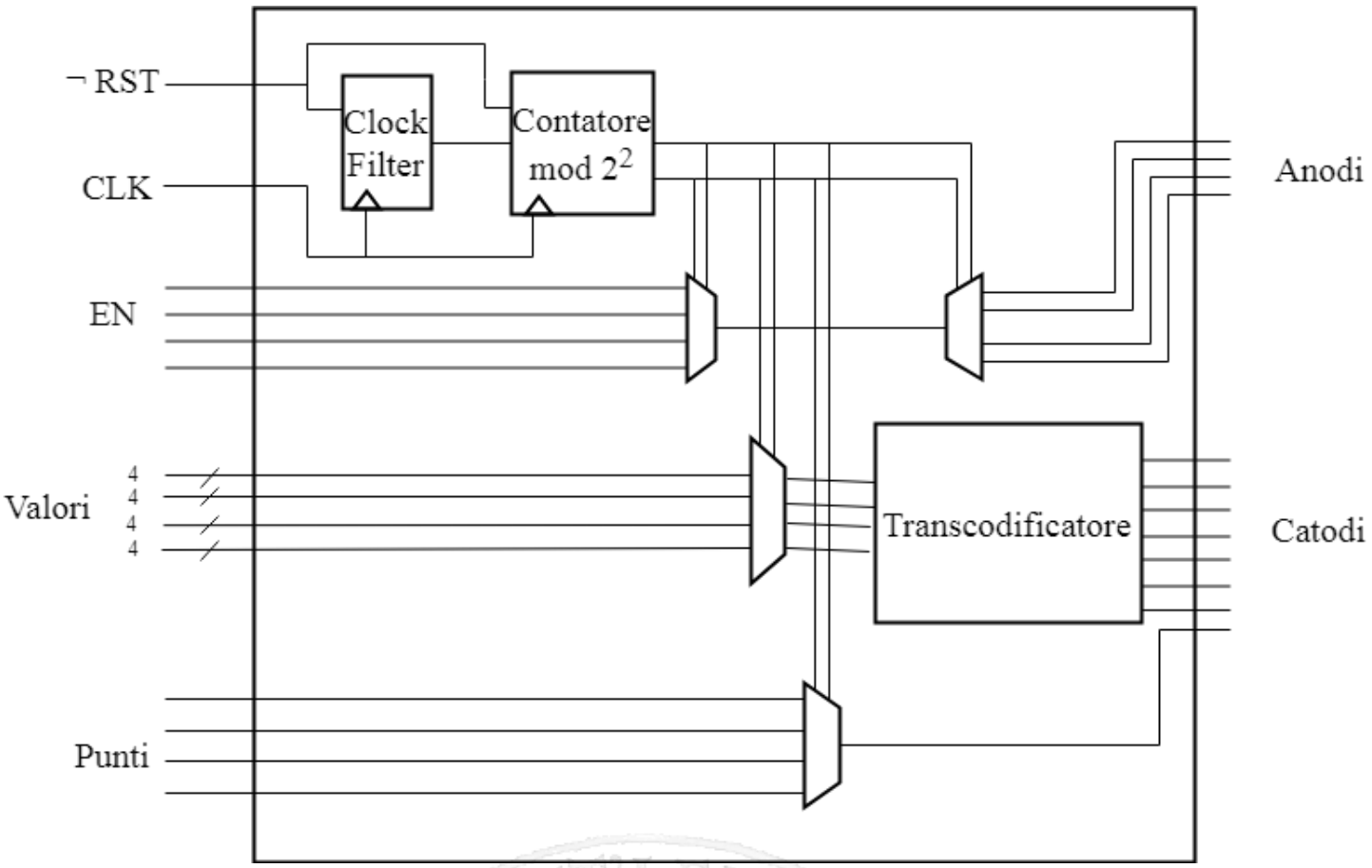


Figura 4.1: Architettura del display a sette segmenti

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.

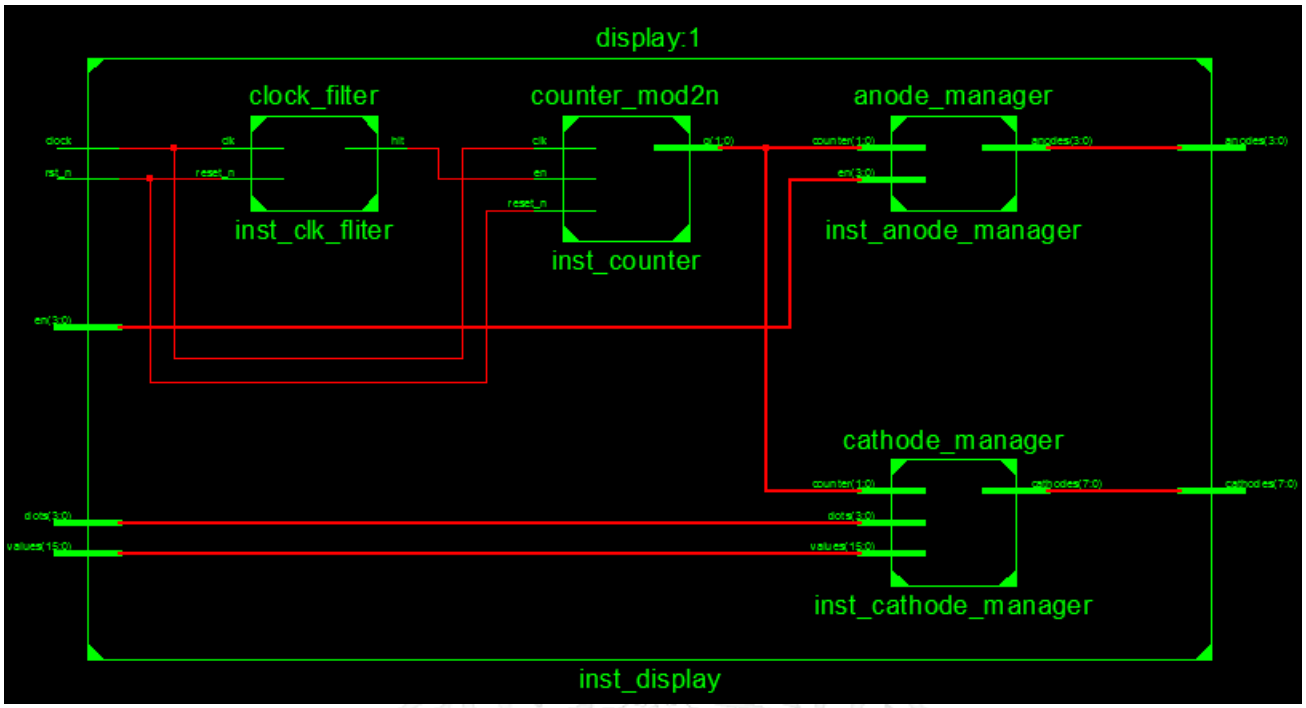


Figura 4.2: Display RTL Schematic

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.

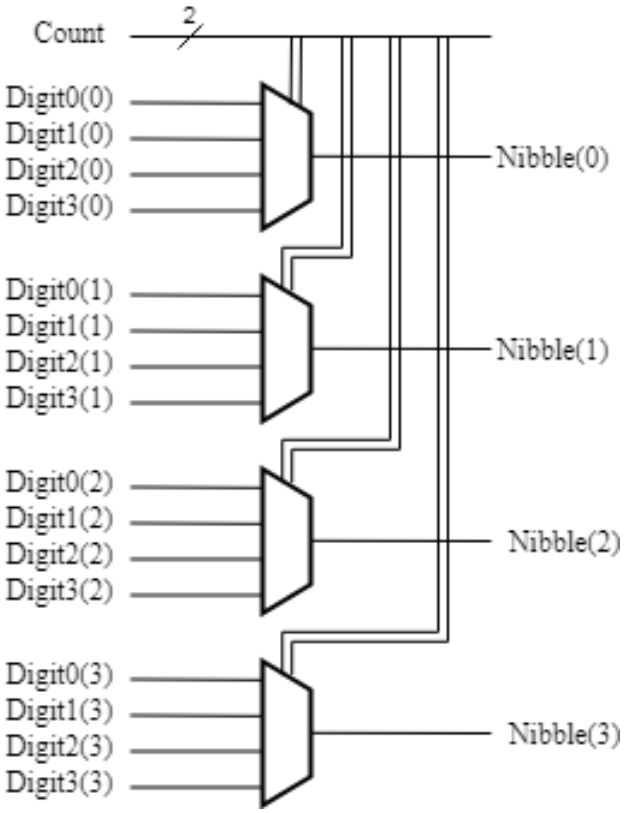


Figura 4.3: Architettura di multiplexing 16x4

4.2.2 Codice

Display sette segmenti ISE

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
14 architecture Behavioral of clock_filter is
15 constant freq_ratio : NATURAL := freq_clock/freq_hit;
16 begin
17 -- Ho bisogno di un contatore N
18   hit_counter_n : process(clk, reset_n)
19   variable count : NATURAL := 0;
20   begin
21     if reset_n = '0' then
22       count := 0;
23       hit <= '0';
24     elsif rising_edge(clk) then
25       if count = (freq_ratio-1) then
26         count := 0;
27         hit <= '1';
28       else
29         count := count + 1;
30         hit <= '0';
31       end if;
32     end if;
33   end process;
34 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.

```
1  library IEEE;
2  use IEEE.STD_LOGIC_1164.ALL;
3
4  entity anode_manager is
5      Port (  counter : in STD_LOGIC_VECTOR (1 downto 0);
6          en : in  STD_LOGIC_VECTOR (3 downto 0);
7        anodes : out  STD_LOGIC_VECTOR (3 downto 0)
8        );
9  end anode_manager;
10
11 architecture structural of anode_manager is
12
13 component muxn_1 is
14   generic(address_width : natural := 3);
15   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;

component demux1_n is
  generic(address_width : natural := 2);
  port(
    a : in std_logic;
    sel : in std_logic_vector(address_width-1 downto 0);
    x : out std_logic_vector(2**address_width-1 downto 0)
  );
end component;
for all : demux1_n use entity WORK.demux1_n(dataflow);

signal enable_anode : STD_LOGIC := '0';

begin

  inst_mux4_1 : muxn_1
    generic map ( address_width => 2 )
    port map (  SEL => counter,
            A => en,
            X => enable_anode
          );

  inst_demux4_1 : demux1_n
    generic map ( address_width => 2 )
    port map (  sel => counter,
            a => enable_anode,
            x => anodes
          );

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.

```
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),
```

```
            X => nibble(i)
          );
  end generate;

  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)
           );

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.

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

entity cathode_encoder is
    Port ( nibble : in  STD_LOGIC_VECTOR (3 downto 0);
           cathodes : out  STD_LOGIC_VECTOR (6 downto 0)
         );
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.

```
architecture Behavioral of cathode_encoder is

constant digit0 : std_logic_vector(6 downto 0) := "1000000";
constant digit1 : std_logic_vector(6 downto 0) := "1111001";
constant digit2 : std_logic_vector(6 downto 0) := "0100100";
constant digit3 : std_logic_vector(6 downto 0) := "0110000";
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.

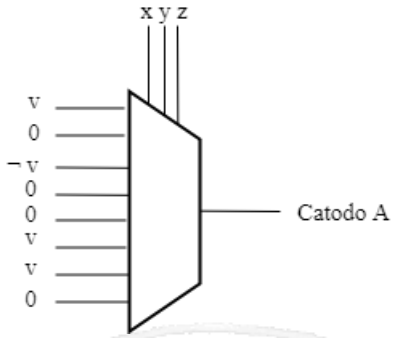


Figura 4.4: Multiplexer 8x1 che realizza la funzione booleana del segmento A

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"*.

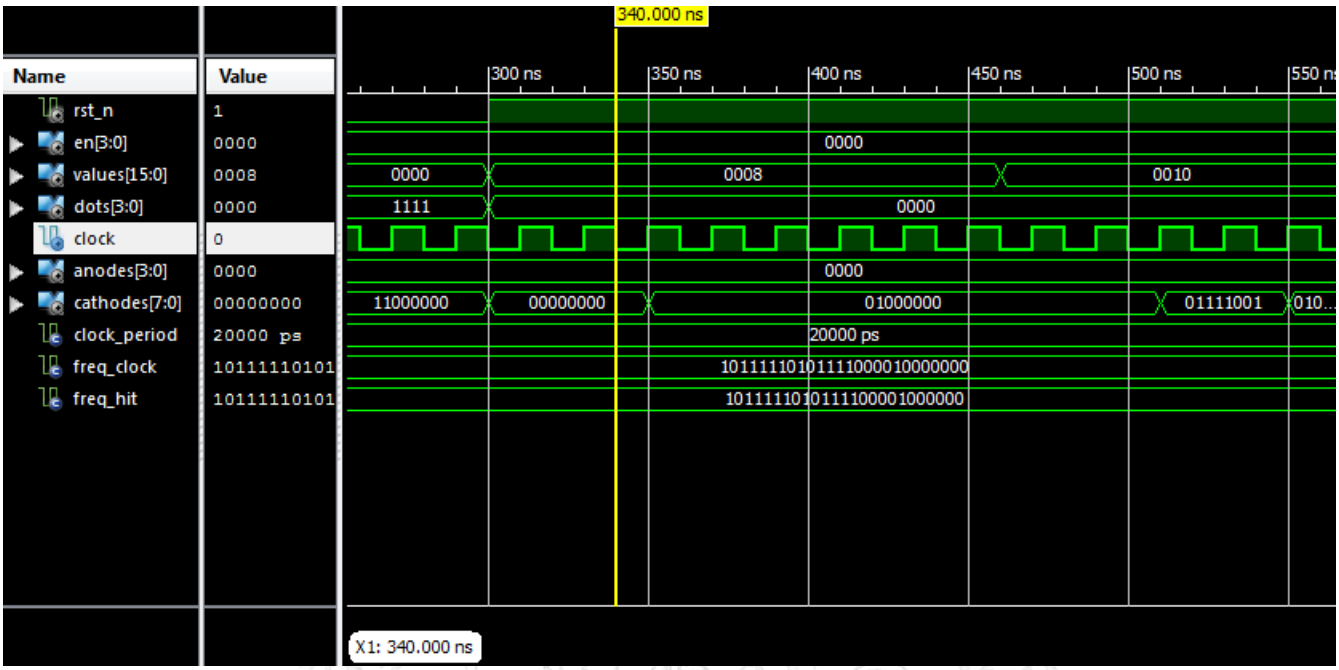


Figura 4.6: Simulazione del display: i catodi seguono correttamente il valore di ingresso per i sette segmenti delle diverse cifre

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.

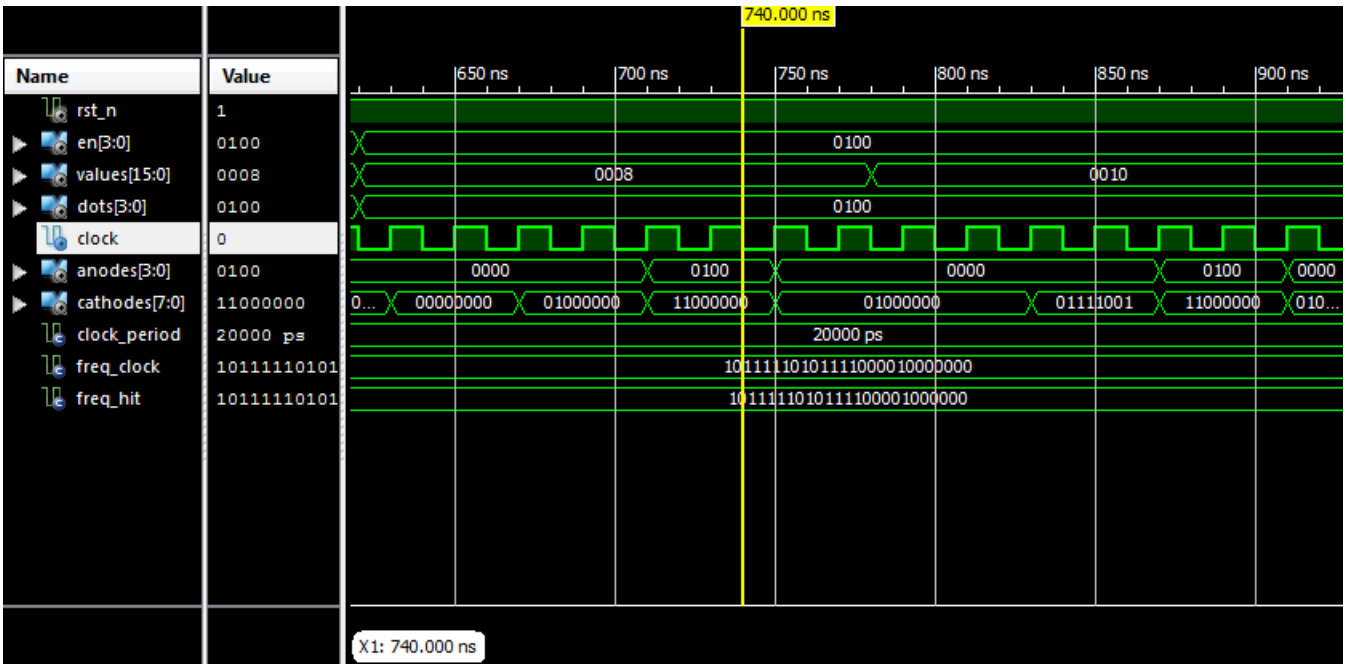


Figura 4.7: Simulazione del display: gli anodi seguono correttamente i valori di abilitazione; il catodo del punto segue correttamente i valori dei punti

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"*).

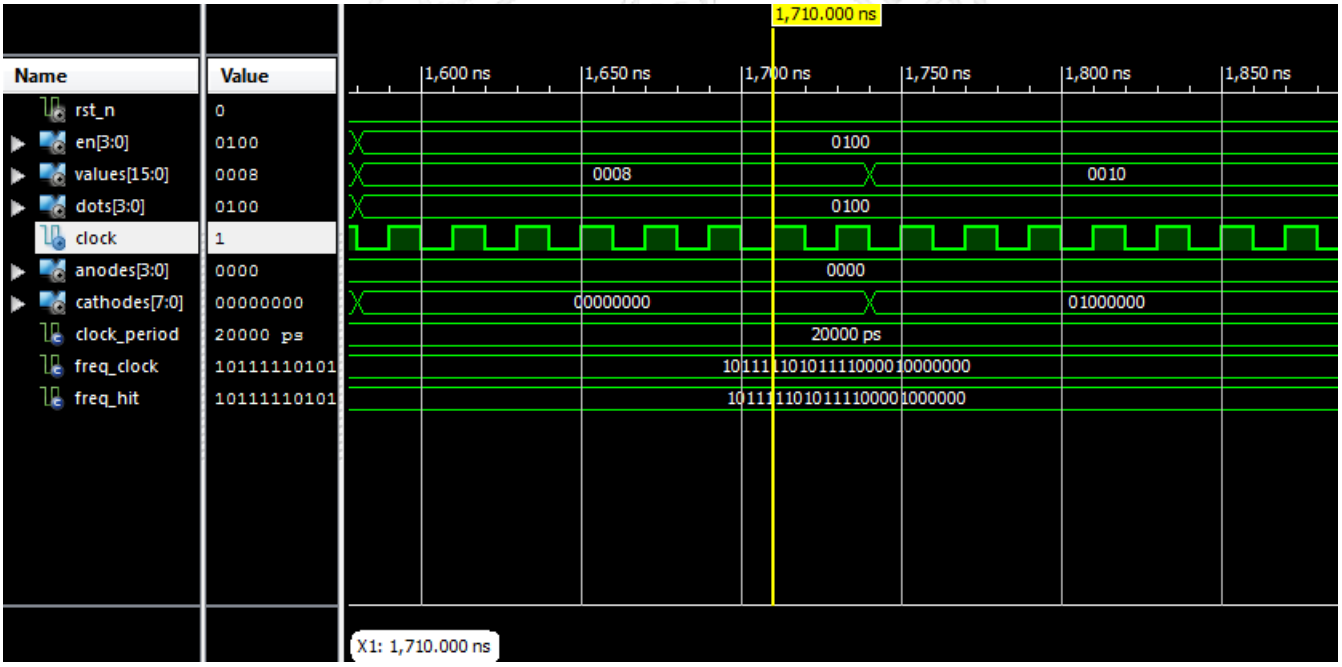


Figura 4.8: Simulazione del display: il reset basso blocca il conteggio alla prima cifra

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. Il segnale di *clock* è associato al clock della board, il *reset* ed i tre bit di *loader* sono associati ai quattro pulsanti, *in_byte* agli otto interruttori, *anodes* 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 caricamento (*button)*, 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, di questo se ne occupa il componente display on board che con un process controlla se è stato premuto il primo pulsante così da usare l' input in ingresso per abilitare solo determinate cifre o punti; con il secondo invece si carica il valore da voler visualizzare sul display .

```
1  entity display_top_level is
2    Generic ( freq_clock : integer := 50000000;
3            freq_hit : integer := 250
4           );
5      Port ( clock : in  STD_LOGIC;
6             reset : in  STD_LOGIC;
7  --           load_lsb_value : in  STD_LOGIC;
8  --           load_msb_value : in  STD_LOGIC;
9  --           load_dots_enable : in  STD_LOGIC;
10          load_conf :in STD_LOGIC;
11          load_value : in STD_LOGIC;
12          number: in STD_LOGIC_VECTOR (15 downto 0);
13             in_byte : in  STD_LOGIC_VECTOR (7 downto 0);
14             anodes : out  STD_LOGIC_VECTOR (3 downto 0);
15             cathodes : out  STD_LOGIC_VECTOR (7 downto 0));
16 end display_top_level;
17
18 architecture structural of display_top_level is
19
20 component display_on_board is
21     Port ( clock : in  STD_LOGIC;
22            reset : in  STD_LOGIC;
23 --           load_lsb_value : in  STD_LOGIC;
24 --           load_msb_value : in  STD_LOGIC;
25 --           load_dots_enable : in  STD_LOGIC;
26          load_conf :in STD_LOGIC;
27          load_value : in STD_LOGIC;
28          number: in STD_LOGIC_VECTOR (15 downto 0);
29             in_byte : in  STD_LOGIC_VECTOR (7 downto 0);
30             value : out  STD_LOGIC_VECTOR (15 downto 0);
31             dots : out  STD_LOGIC_VECTOR (3 downto 0);
32             en : out  STD_LOGIC_VECTOR (3 downto 0));
```

```
33 end component;
34 component display is
35   Generic ( freq_clock : integer := 50000000;
36           freq_hit : integer := 250
37         );
38    Port ( rst_n : in  STD_LOGIC;
39        clock : in STD_LOGIC;
40           en : in  STD_LOGIC_VECTOR (3 downto 0);
41           values : in  STD_LOGIC_VECTOR (15 downto 0);
42           dots : in  STD_LOGIC_VECTOR (3 downto 0);
43           anodes : out  STD_LOGIC_VECTOR (3 downto 0);
44           cathodes : out  STD_LOGIC_VECTOR (7 downto 0));
45 end component;
46 signal value : STD_LOGIC_VECTOR (15 downto 0) := (others => '0');
47 signal dots : STD_LOGIC_VECTOR (3 downto 0) := (others => '1');
48 signal en : STD_LOGIC_VECTOR (3 downto 0) := (others => '1');
49 signal rst_n : STD_LOGIC := '1';
50 begin
51   rst_n <= not(reset);
52   inst_on_board : display_on_board
53     port map ( clock => clock,
54              reset => reset,
55 --             load_lsb_value => load_lsb_value,
56 --             load_msb_value => load_msb_value,
57 --             load_dots_enable => load_dots_enable,
58              load_conf => load_conf,
59              load_value => load_value,
60              number => number,
61              in_byte => in_byte,
62              value => value,
63              dots => dots,
64              en => en
65            );
66   inst_display : display
67   generic map( freq_clock => freq_clock,
68               freq_hit => freq_hit
69              )
70     port map ( clock => clock,
71              rst_n => rst_n,
72              values => value,
73              dots => dots,
74              en => en,
75              anodes => anodes,
76              cathodes => cathodes
77            );
78 end structural;
```

Codice Componente 4.10: Architettura della Control Unit

Capitolo 5

Clock Generator

5.1 Traccia

Illustriamo come un DCM, posso essere sostituito al clock_filter del Display a 7 segmenti, per avere un segnale di abilizatione, per sostiturlo al clock_filter utilizzato nell' esercizio precedente.

5.2 Soluzione

5.2.1 Schematici

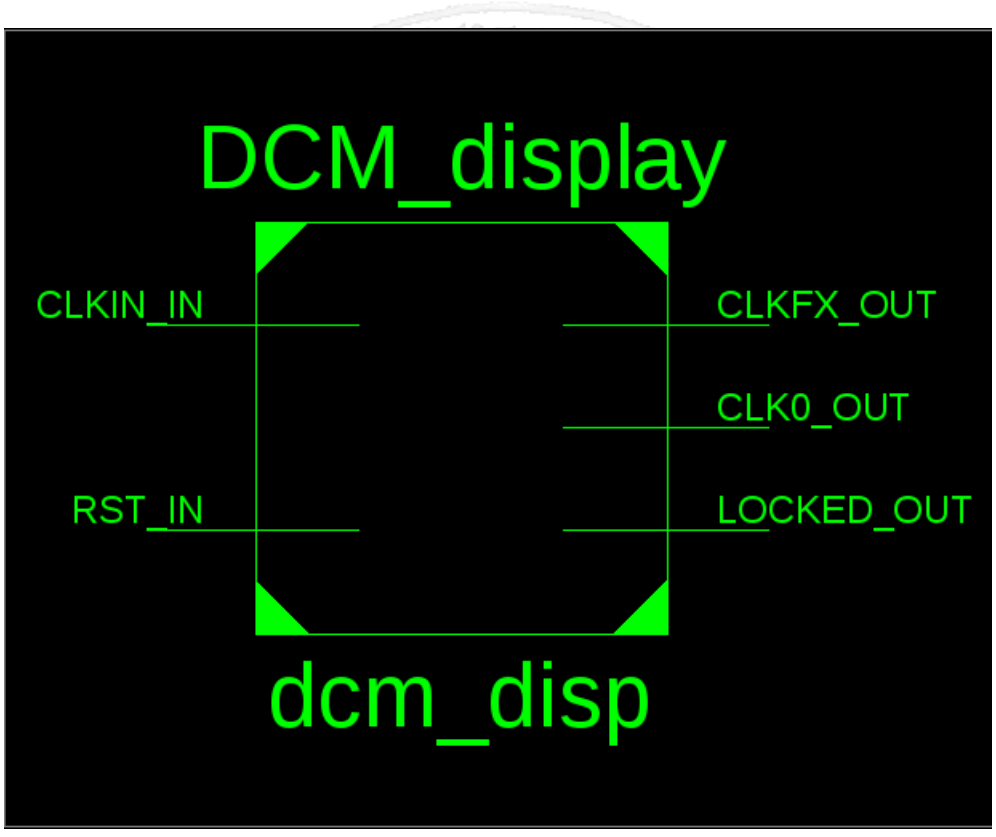


Figura 5.1: Architettura del display a sette segmenti

Utilizzando il wizard di xilinx per la creazione degli ip core, possiamo chiedere di creare un DCM che con un a frequenza in input di 50 MHz ci restituisca in uscita un segnale periodico con frequenza di 5 MHz, la frequenza desiderata è molto alta rispetto a quella che basterebbe per visualizzare le cifre sul display, infatti si notano alcuni sfarfalii, si potrebbe risolvere utilizzando una struttura a doppio DCM, per far si che il numero di possibili frequenze a disposizioni aumentino, ci accontentiamo della soluzione ad un singolo DCM essendo le cifre visibili sul display.

5.2.2 Codice

DCM ISE

5.3 Simulazione

Fare riferimento alla simulazione del display 4.3.

5.4 Sintesi su board FPGA

Fare riferimento a 4.4.

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.

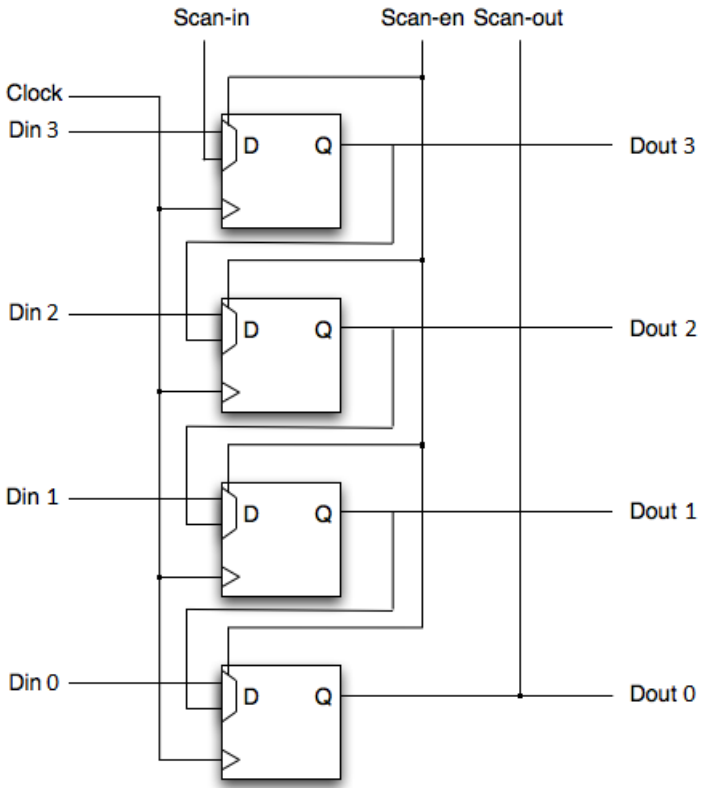


Figura 6.1: Boundary Scan Chain

6.2.2 Codice

Progetto ISE: Boundary Scan Chain ISE

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. Quando *scan_en* è basso, il segnale *din* viene salvato nel registro composto dai vari flip flop, nel caso in cui *scan_en* sia attivo i vari flip flop vengono connessi modo tale che ad ogni colpo di clock, il valore di *scan_in*venga salvato nel primo flip flop, il valore salvato nel primo venga propagato nel secondo e così via.

```
1  library IEEE;
2  use IEEE.STD_LOGIC_1164.ALL;
3  USE ieee.std_logic_arith.ALL;
4
5  entity boundary_scan_chain is
6    generic(n : natural := 4);
7      Port ( scan_in : in STD_LOGIC;
8             clk : in  STD_LOGIC;
9          reset_n : in STD_LOGIC;
10            din : in  STD_LOGIC_VECTOR (n-1 downto 0);
11            scan_en : in  STD_LOGIC;
12            scan_out : out  STD_LOGIC;
```

```
13              dout : out  STD_LOGIC_VECTOR (n-1 downto 0)
14       );
15  end boundary_scan_chain;
16
17  architecture Structural of boundary_scan_chain is
18
19    component latch_d_en is
20    generic(width:natural:=1);
21      Port ( clk : in  STD_LOGIC;
22             reset : in  STD_LOGIC;
23             en : in  STD_LOGIC;
24             d : in  STD_LOGIC_VECTOR (width-1 downto 0);
25             q : out  STD_LOGIC_VECTOR (width-1 downto 0));
26    end component;
27
28    component mux2_1 is
29      Port ( SEL : in STD_LOGIC;
30           A   : in STD_LOGIC;
31           B   : in STD_LOGIC;
32           X   : out STD_LOGIC
33      );
34    end component;
35
36  signal q : std_logic_vector(n-1 downto 0) := (others => '0');
37  signal x,dinapp : std_logic_vector(n-1 downto 0) := (others => '0');
38  signal s_out: std_logic:='0';
39
40  begin
41    chain_gen: for i in 0 to n-1 generate
42      sc_in: if i=n-1 generate
43        inst_mux2_1: mux2_1
44        Port map( SEL => scan_en,
45                A => q(i-1),
46                B => din(i),
47                X => x(i)
48        );
49
50      inst_edge_triggered: latch_d_en generic map(width =>1)
51        Port map( clk => clk,
52                reset => reset_n,
53                en => '1',
54                d(0) => x(i),
55                q(0) => q(i)
56        );
57      end generate sc_in;
58      sc_ch: if i>0 and i<n-1 generate
59        inst_mux2_1: mux2_1
60          Port map( SEL => scan_en,
61                 A => q(i-1),
```

```
62                B => din(i),
63                X => x(i)
64          );
65
66        inst_edge_triggered: latch_d_en generic map(width =>1)
67          Port map( clk => clk,
68                reset => reset_n,
69                en => '1',
70                d(0) => x(i),
71                q(0) => q(i)
72        );
73        end generate sc_ch;
74      sc_out: if i=0 generate
75      inst_mux2_1: mux2_1
76        Port map( SEL => scan_en,
77                A => scan_in,
78                B => din(i),
79                X => x(i)
80        );
81      inst_edge_triggered: latch_d_en generic map(width =>1)
82        Port map( clk => clk,
83                reset => reset_n,
84                en => '1',
85                d(0) => x(i),
86                q(0) => q(i)
87        );
88      end generate sc_out;
89    end generate;
90    dout<=q;
91    scan_out<=q(n-1);
92  end Structural;
```

Codice Componente 6.1: Definizione della 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*.

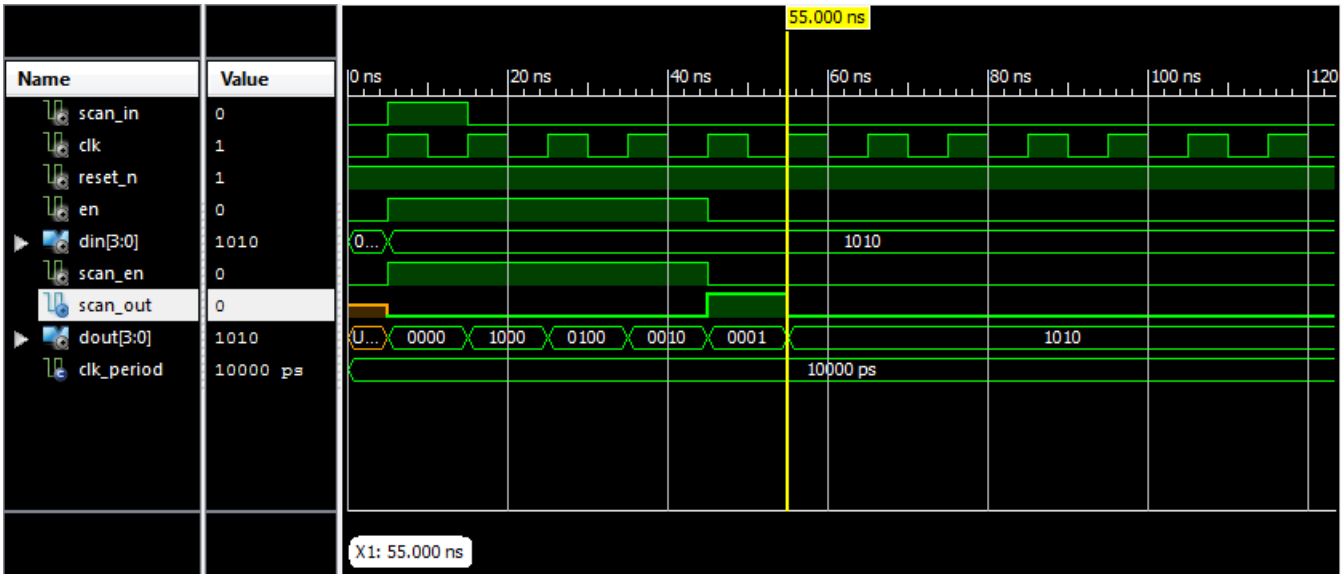


Figura 6.2: Simulazione della Boundary Scan Chain Behavioral

Capitolo 7

Finite State Machine

7.1 Traccia

Si voglia realizzare un macchina a stati finiti che permetta di riconoscere una stringa di bit, di dimensione generica.

7.2 Soluzione

7.2.1 Schematici

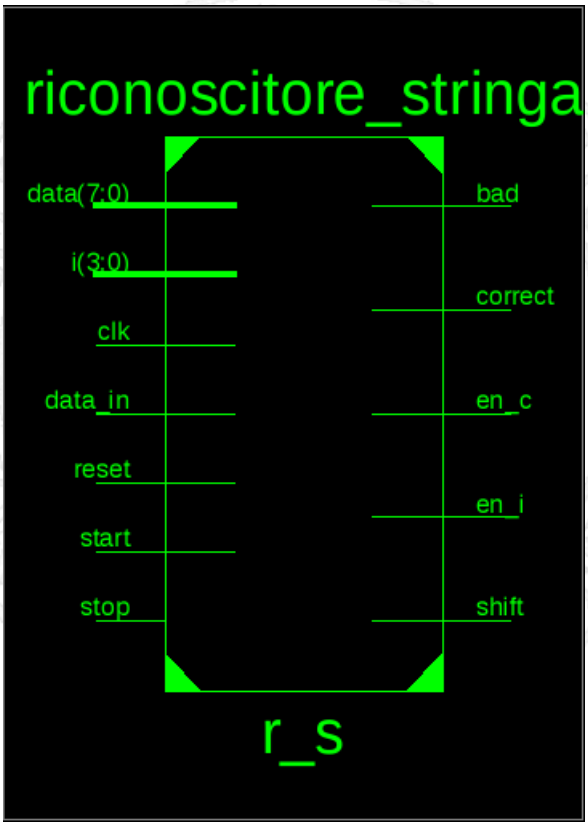


Figura 7.1: Riconoscitore di stringa

La macchina è composta semplicente da quattro stati:

- idle, in cui la macchina permane fintantoché non viene avviata la procedura di riconoscimento;
- carica_stringa, in cui viene salvato in una scan chain il valore della stringa da riconoscere;
- shifting, in cui viene fatto shiftare il bit della stringa e far si ponga in uscita alla scan_chain;
- riconosci_bit, si determina se il bit in uscita della scan_chain è uguale al bit della stringa da riconoscere (quest' ultimo identificato dal valore di un contatore che opportunamente identifica il valore del bit da confrontare): se i due bit sono uguali, si procedere allo shifting ed al riconoscimento fino a quando non vengono confrontati tutti i bit ed un segnale indica che la stringa riconosciuta è proprio quella voluta; altrimenti viene abilitato un segnale che indica il fallimento della comparazione indicando la dissimilirità della stringa posta in ingresso da quella che si vuole riconoscere.

7.2.2 Codice

7.2.2.1 Riconoscitore_stringa

Riconoscitore Stringa ISE

```
entity riconoscitore_stringa is
   generic(width: NATURAL:=8);
    Port ( clk : in STD_LOGIC;
        start: in STD_LOGIC;
        reset : in STD_LOGIC;
        data : in STD_LOGIC_VECTOR(width-1 downto 0);
        data_in : in  STD_LOGIC;
        i: in STD_LOGIC_VECTOR(natural(ceil(log2(real(width)))) downto 0);
        en_res : out STD_LOGIC;
           shift : out STD_LOGIC;
           en_i : out  STD_LOGIC;
        en_c : out STD_LOGIC;
        bad: out  STD_LOGIC;
        correct: out STD_LOGIC);
end riconoscitore_stringa;

architecture Behavioral of riconoscitore_stringa is
type state is (idle,riconosci_bit,shifting,carica_stringa);
signal current_state : state;
signal n:STD_LOGIC_VECTOR(natural(ceil(log2(real(width)))) downto 0):=(
   others=>'0');
begin
change_state: process (clk,reset)
  begin
    if (reset = '0') then
      current_state <= idle;
    elsif rising_edge(clk) then
    case current_state is
```

```
28        when idle=> if start='1' then
29                  current_state <= carica_stringa;
30                end if;
31        when carica_stringa=> current_state<=shifting;
32        when shifting=> current_state<=riconosci_bit;
33        when riconosci_bit=> if i=std_logic_vector(to_unsigned(width,n'length)
              ) then
34                        current_state<=idle;
35                      elsif data(to_integer(unsigned(i)))=data_in then
36                        current_state<=shifting;
37                      elsif data(to_integer(unsigned(i)))/=data_in then
38                        current_state<=idle;
39                      end if;
40      end case;
41      end if;
42    end process;
43    recognize_bit: process(clk,current_state,i,data_in,start)
44     begin
45     shift<='0';
46     correct<='0';
47     bad<='0';
48     en_c<='0';
49     en_i<='0';
50     en_res<='0';
51      case current_state is
52        when idle =>
53          shift<='0';
54          --n<=std_logic_vector(to_unsigned(width,n'length));
55          if start='1' then
56            correct<='0';
57            bad<='0';
58            en_i<='1';
59            shift<='1';
60            --next_state<=shifting;
61          end if;
62        when carica_stringa =>
63            en_i<='0';
64        when shifting =>
65            en_i<='1';
66            shift<='0';
67            en_c<='0';
68            --next_state<=riconosci_bit;
69        when riconosci_bit =>
70            en_i<='1';
71            if i=std_logic_vector(to_unsigned(width,n'length)) then
72            correct<='1';
73            en_res<='1';
74            --next_state<=idle;
75            elsif data(to_integer(unsigned(i)))=data_in then
```

```
76            en_c<='1';
77             shift<='1';
78            --next_state<=shifting;
79            else
80            bad<='1';
81            en_res<='1';
82            --next_state<=idle;
83            end if;
84       end case;
85    end process;
86  end Behavioral;
```

Codice Componente 7.1: Definizione della macchina a stati finiti

Notiamo che tutti i segnali di uscita, per le varie abilitazioni, come lo shifting o l' abilitazione dei conteggi vengono fissati ad un valore ben preciso prima del case all' interno del secondo process, questo per far si che il sintetizzatore, non rilevi dei latch durante la sua esecuzione, (ciò può essere dovuto al fatto che i segnali non essendo fissati in dei registri il sintetizzatore vuole salvarli così da poter effettuare una corretta sintesi del codice, questo vale anche per le altre macchine a stati finiti sviluppate successivamente).

Di seguito vengono riportati i differenti risultati operativi nel caso in cui si utilizzino diverse codifiche di rappresentazione degli stati si una macchina a stati finiti.

Codifica	Numero di slice	Numero di flip flop	Numero di four lut	Frequenza massima
One-hot	19	16	37	184.805MHz
Speed1	19	16	37	184.805MHz
Compact	15	14	29	191.031MHz
Sequential	15	14	29	191.031MHz
Gray	17	14	31	180.665MHz
Johnson	17	14	31	180.665MHz

Dato il numero di stati molto ridotto, i risultati ottenuti utilizzando le varie codifiche in alcuni casi sono risultati perfettamente gli stessi, le codifiche che hanno dato gli stessi risultato sono state messe una successivamente all' altra, difatti notiamo che One-hot e Speed1 ottengono gli stessi risultati, facendo riferimento alla manuale di XST di Xilinx la codifica Speed1 è orientata alla velocità ma i bit usati per identificare lo stato di solito sono in numero maggiore al numero degli stati della macchina, quindi il sintetizzatore alla fine si riconduce alla codica One-hot ed è quello che occupa maggior area, dato che utilizza un singolo flip flop per codificare ogni stato quindi la sua velocità al crescere degli stati e indipendete, anche se l' area occupata crescerà linearmente con il numero di stati;

compact è quella che richiede meno spazio cerca di utilizzare il minore numero di bit possibili per codificare gli stati, in questo caso risulta la più efficiente, perchè oltre ad essere minore lo spazio occupato essendo il numero di stati davvero esiguo il tutto riesce ad essere mappato in slice presenti nella stessa CLB, rendendo la frequenza massima operativa maggiore;

Sequential utilizzato un approccio radix-two sugli stati presenti su path molto lunghi, ma in questo caso ci si riconduce ad una soluzione compact essendo gli stati solo due;

Gray permetteno di far variare un solo bit alla volta occupa lo stesso spazio della compact, ma la logica combinatoriale per far variare tra gli stati e più complessa ed implica anche una minore frequenza operativa, per rendere sicuro il cambio di stato;

Johnson comunque permette la variazione di un solo bit alla volta, ma in questo caso coincide con la codifica di gray, anche se di solito i bit per la rappresentazione sono maggiori.

7.3 Simulazione

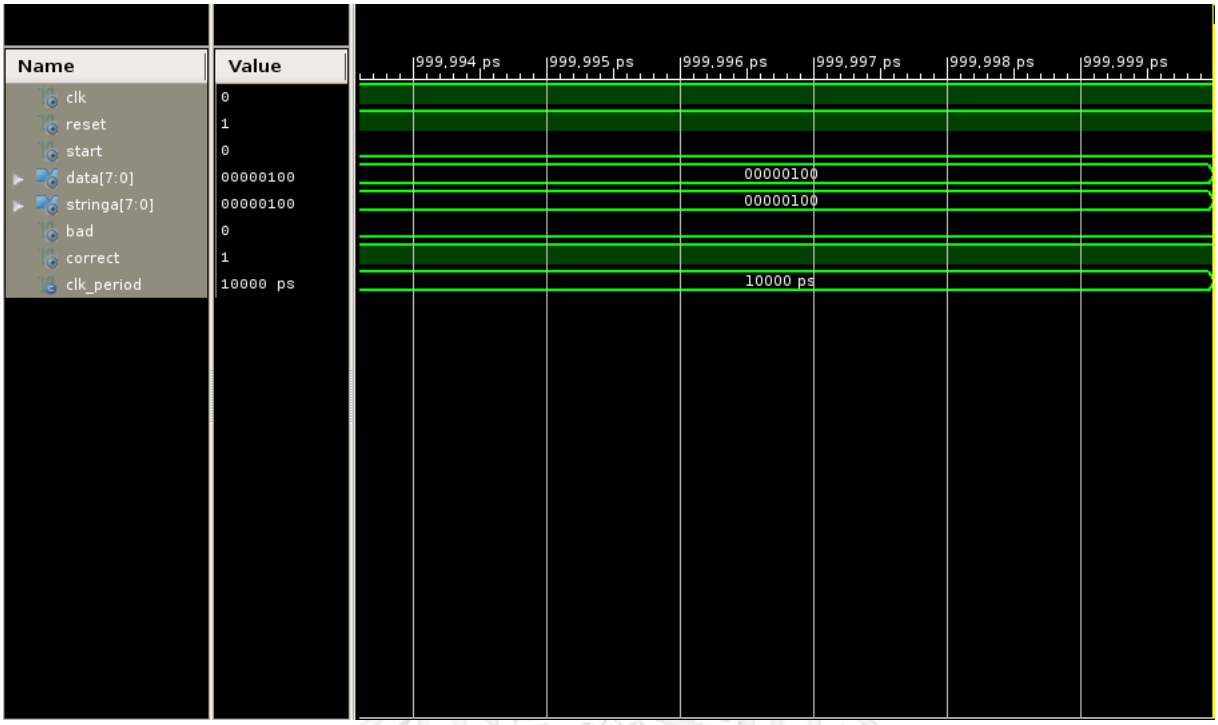


Figura 7.2: Stringa riconosciuta correttamente

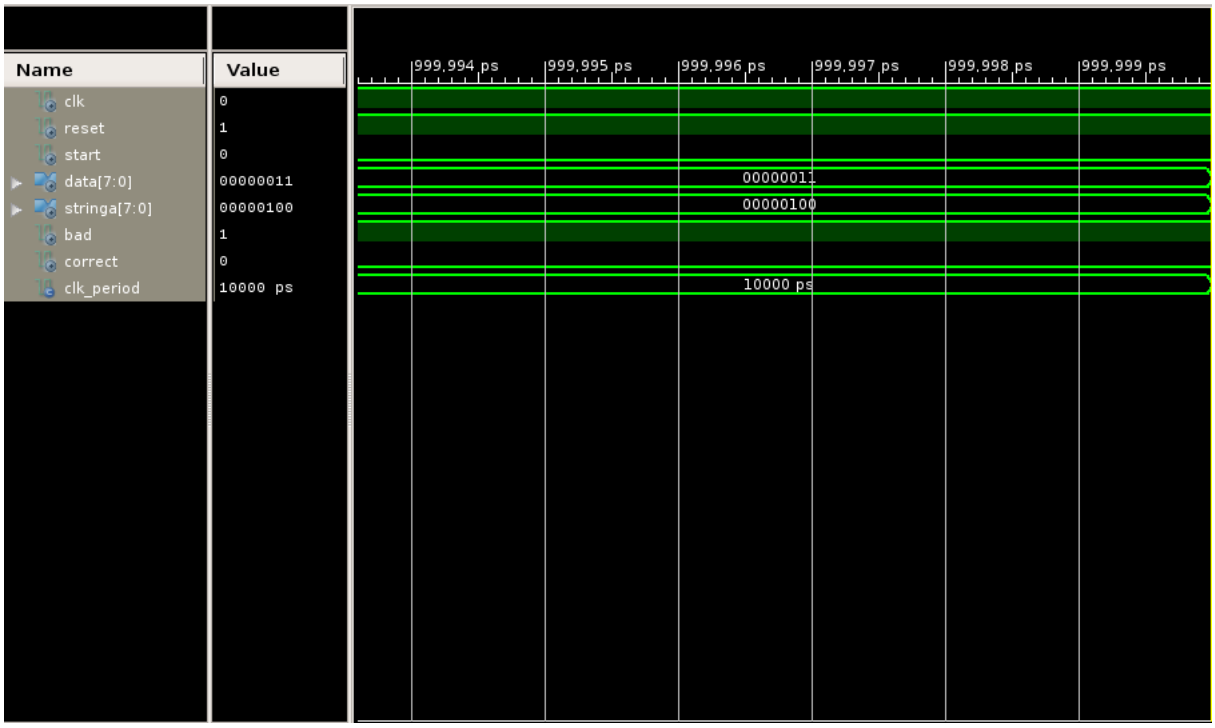


Figura 7.3: Stringa non riconosciuta

Nei due esempi: in alto, viene mostrato un caso in cui la stringa da riconoscere è quella in ingresso sono le stesse in basso invece, in cui le due stringhe sono dissimili.

7.4 Sintesi su board FPGA

```
entity tester_dispositivi is
    Port ( clock : in  STD_LOGIC;
        button : in STD_LOGIC_VECTOR (3 downto 0);
        led : out STD_LOGIC_VECTOR (7 downto 0);
           in_byte : in  STD_LOGIC_VECTOR (7 downto 0));
end tester_dispositivi;

architecture Behavioral of tester_dispositivi is
COMPONENT contatore_modulo_2n
  PORT(
    clk : IN std_logic;
    enable : IN std_logic;
    reset : IN std_logic;
    hit : OUT std_logic;
    output : OUT std_logic_vector(1 downto 0)
    );
  END COMPONENT;
  component latch_d_en is
  generic(width:natural:=8);
    Port ( clk : in  STD_LOGIC;
```

```
21            reset : in  STD_LOGIC;
22            en : in  STD_LOGIC;
23            d : in  STD_LOGIC_VECTOR (width-1 downto 0);
24            q : out  STD_LOGIC_VECTOR (width-1 downto 0));
25    end component;
26    COMPONENT riconoscitore_stringa_generico
27    PORT(
28      clk : IN std_logic;
29      reset : IN std_logic;
30      start : IN std_logic;
31      data : IN std_logic_vector(7 downto 0);
32      stringa : IN std_logic_vector(7 downto 0);
33      bad : OUT std_logic;
34      correct : OUT std_logic
35      );
36    END COMPONENT;
37    COMPONENT display_top_level
38    PORT(
39      clock : IN std_logic;
40      reset : IN std_logic;
41      load_conf : IN std_logic;
42      load_value : IN std_logic;
43      number : IN std_logic_vector(15 downto 0);
44      in_byte : IN std_logic_vector(7 downto 0);
45      anodes : OUT std_logic_vector(3 downto 0);
46      cathodes : OUT std_logic_vector(7 downto 0)
47      );
48    END COMPONENT;
49    COMPONENT debounce
50    PORT(
51      clk : IN std_logic;
52      button : IN std_logic;
53      result : OUT std_logic
54      );
55    END COMPONENT;
56    signal sel :STD_LOGIC_VECTOR(1 downto 0):=(others=>'0');
57    signal scelta: STD_LOGIC_VECTOR( 3 downto 0):=(others=>'0');
58    signal value:STD_LOGIC_VECTOR(15 downto 0) := (others=>'0');
59    signal en_c,en_ric,hit,en_c1,reset_c,bad,correct:STD_LOGIC:='0';
60  begin
61    deb:debounce port map(clock,button(0),en_c);
62    change: process (clock,en_c)
63    begin
64    if rising_edge(clock) then
65      if en_c='1' then
66        case sel is
67          when "00" => scelta<="0001";
68          when "01" => scelta<="0010";
69          when "10" => scelta<="0100";
```

```
70          when "11" => scelta<="1000";
71          when others => scelta<="1111";
72        end case;
73      else
74        scelta<="0000";
75      end if;
76    end if;
77    end process;
78    selettore: contatore_modulo_2n port map(en_c,'1',not( button(3)),open,sel)
        ;
79    data : latch_d_en port map(clock,not(button(3)),scelta(1),in_byte(7 downto
         0),value(7 downto 0));
80    stinga : latch_d_en port map(clock,not(button(3)),scelta(2),in_byte(7
        downto 0),value(15 downto 8));
81    counter: contatore_modulo_2n port map(clock,en_c1,reset_c,hit,open);
82    st: process(scelta,clock,hit)
83      begin
84      reset_c<='1';
85        if scelta(3)='1' and hit='0' then
86          en_ric<='1';
87          en_c1<='1';
88        elsif scelta(3)='1' and hit='1' then
89          en_c1<='0';
90          en_ric<='0';
91        else
92          reset_c<='0';
93        end if;
94      end process;
95    led(7)<=sel(1);
96    led(6)<=sel(0);
97    led(0)<=correct;
98    led(1)<=bad;
99    ric_stringa_generico : riconoscitore_stringa_generico port map(clock,not(
        scelta(0)),en_ric,value(7 downto 0),value(15 downto 8),bad,correct);
100 end Behavioral;
```

Codice Componente 7.2: Definizione del componente che gestisce il caricamento dei dati

Il codice mostrato descrive un tester utilizzato, con piccole differenze dove occorre, utilizzato per tutti i dispositivi non solo combinatoriali presenti in tale elaborato, difatti al suo interno è presente un process che genera un segnale di start attivo fino a quando il contatore al di sopra di esso non termina, questo perchè tale segnale non deve essere attivo continuamente altrimenti i dispositivi ricomincerebbero a computare non appena tornerebbero nello stato di riposo, ma deve essere attivo per un tempo tale affinchè venga rilevato dal dispositivo che deve essere avviato.

Vengono utilizzati gli switch per inserire sia la stringa da riconoscere, oltre a quella con cui si deve effettuare il confronto, tali valori sono sorretti da registri;

il led zero si attiva se il riconoscimento è andato a buon fine, il led uno nel caso contrario;

i led sei e sette occorrono a determinare: quali registri per contenere i dati si sono selezionati (se acceso il led sei il dato da voler confrontare, se acceso il sette la stringa da riconoscere), se

si è avviata la computazione (entrambi i led accesi) o nel caso in cui sono spenti entrambi venga effettuato il reset del dispositivo per effettuare una successiva computazione;

il reset degli altri componenti invece è abilitato della pressione del pulsante tre, invece il pulsante zero permette la selezione dei registri e l' avvio della macchina a stati finiti;

per caricare i dati gli switch devono essere impostati prima che il led relativo al registro venga acceso, vale anche per gli altri tester utilizzati.

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

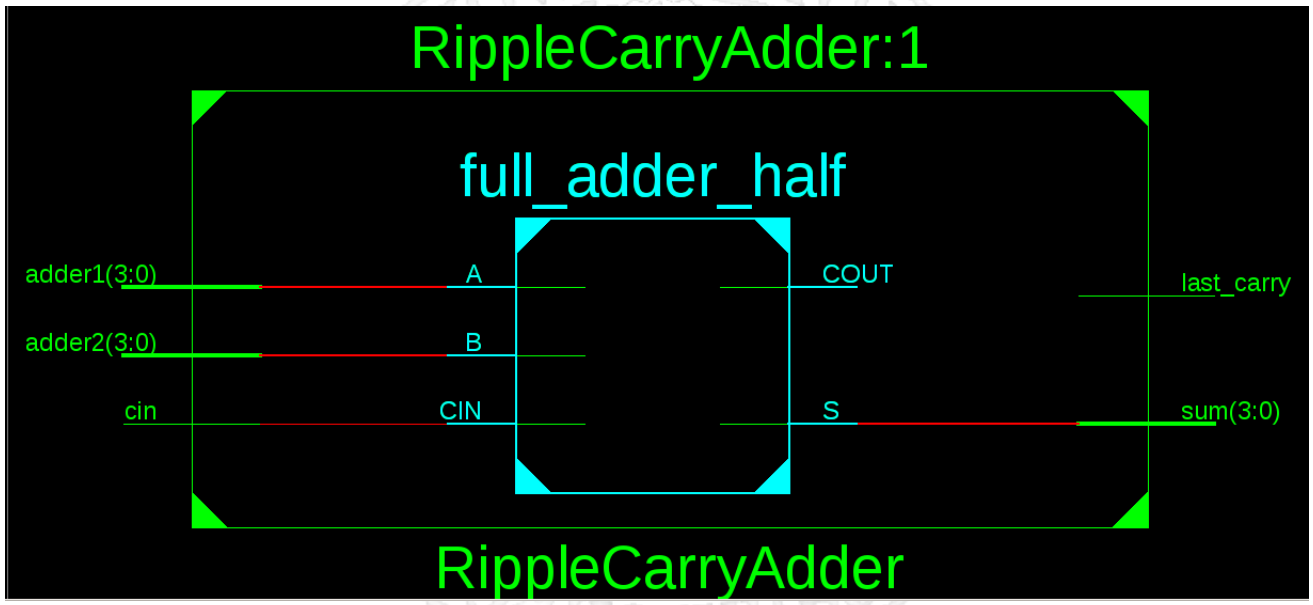


Figura 8.1: Ripple Carry Adder

Per realizzare il circuito si è ricorsi al costrutto for generate, per colleggare i vari full adder per far sì che i riporti generati dal full adder precedente vengano propagati al successivo, purtroppo dallo

schematico non è evidente, perché i costrutti di questo tipo quando viene chiesto ad ISE di creare lo schematico non lo espande mostrando tutti i singoli full_adder, ma ne crea uno solo che però ha in ingresso ed uscita un vettore di bit.

Numero di bit operandi	Numero di slice	Numero di four LUT	Tempo di calcolo
4	6	8	7,225 ps
8	12	16	7,735 ps
16	24	32	9,358 ps
32	48	64	9,985 ps

Secondo le formule dell' area e del ritardo, l' area è pari a $5n$ (dove n indica il numero di full-adder) , ed $2\delta n$ per il ritardo, la realizzazione per FPGA invece presenta dimensioni e ritardi diversi, difatti l' area raddoppia se raddoppiano il numero di bit per la somma, molto probabilmente perché utilizzerà ogni slice come un full adder, invece il tempo di calcolo all' aumentare del numero di bit cresce ma meno che linearmente, determinante dal fatto che il sintetizzatore riuscirà a disporre le slice molto vicine tra di loro e il tempo di propagazione dei riporti diverrà molto piccolo, si nota però che da 8 a 16 bit il tempo di calcolo incrementa di una quantità maggiore rispetto al passaggio da a 4 a 8 o da 16 a 32, perchè le connessioni non avvengono tra CLB vicini tra loro, per determinare il tempo si è scelto come input sempre il massimo valore accettabile in ingresso.

Non è stato possibile testare il componente con numero di bit di operandi a 64 bit, poichè quando si cerca di mappare sulla scheda i vari pin di ingresso e di uscita dell ' addizionatore, questa operazione non riesce, una soluzione sarebbe quella di evitare che I/O venga mappato, però ISE ci avverte che i tempi di propagazione potrebbero essere non veritieri, allora si potrebbe utilizzare un altro componente per far passare gli input non contemporaneamente all' addizionatore, ma l' area del dispositivo verrebbe influenzata da tutte le connessioni che occorrono per questo dispositivo e ci potrebbero essere delle ottimizzazioni che minimizzano l' area.

8.2.2 Codice

Ripple Carry Adder ISE

8.3 Simulazione

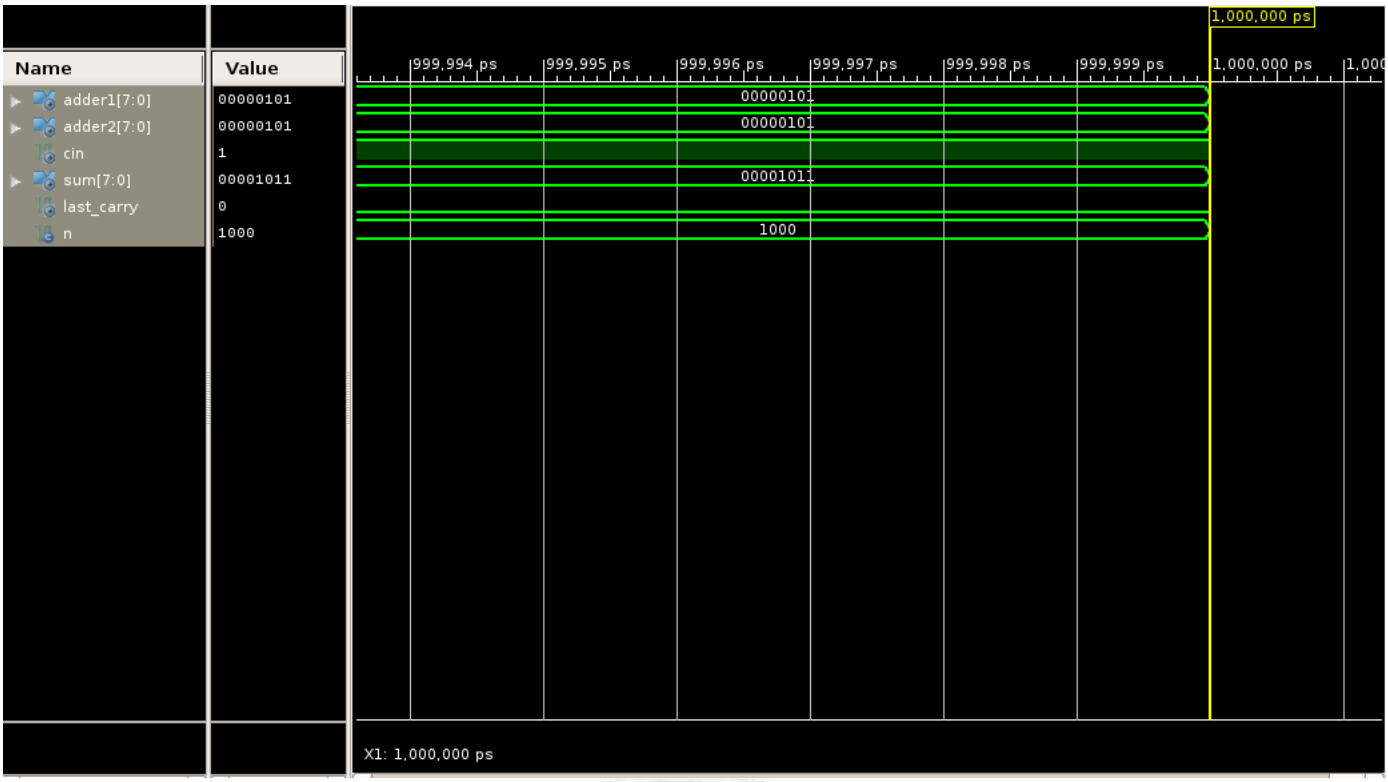


Figura 8.2: Ripple Carry Adder esempio di somma

8.4 Sintesi su board FPGA

Per la sintesi si è utilizzato un tester simile a questo 7.4, con la differenza che non vi è un process per la gestione dello start, essendo il componente da testare combinatoriale, gli switch occorrono per l' inserimento degli operandi i led sette , sei e cinque indicano, nel caso in cui il led cinque o il led sei sia acceso che abbiamo selezionato un registro per caricare i dati, se sono accesi entrambi stiamo assegnando il valore uno al carry in ingresso, se si abilita il led sette utilizziamo l' addizionatore come sottrattore, mentre l' accensione del led zeo determina se è presente il carry in uscita o meno, quella del led uno che vi è una situazione d' errore dovuta al sottrattore, a dispetto del tester messo in riferimento, qui viene anche utilizzato il display per visualizzare il risultato, per decidere quante cifre del display vogliamo utilizzare basta abilitare gli ultimi quattro switch (a partire dal quinto all' ottavo questi mettono in funzione dalla prima alla quarta cifra) e premere il pulsante due, se abbiamo bisogno dei punti questi vengono abilitati dai primi quattro switch (stesso ragionamento fatto per le cifre) dopodichè bisogna sempre premere il pulsante due, per visionare la somma bisogna premere il pulsante uno.

Capitolo 9

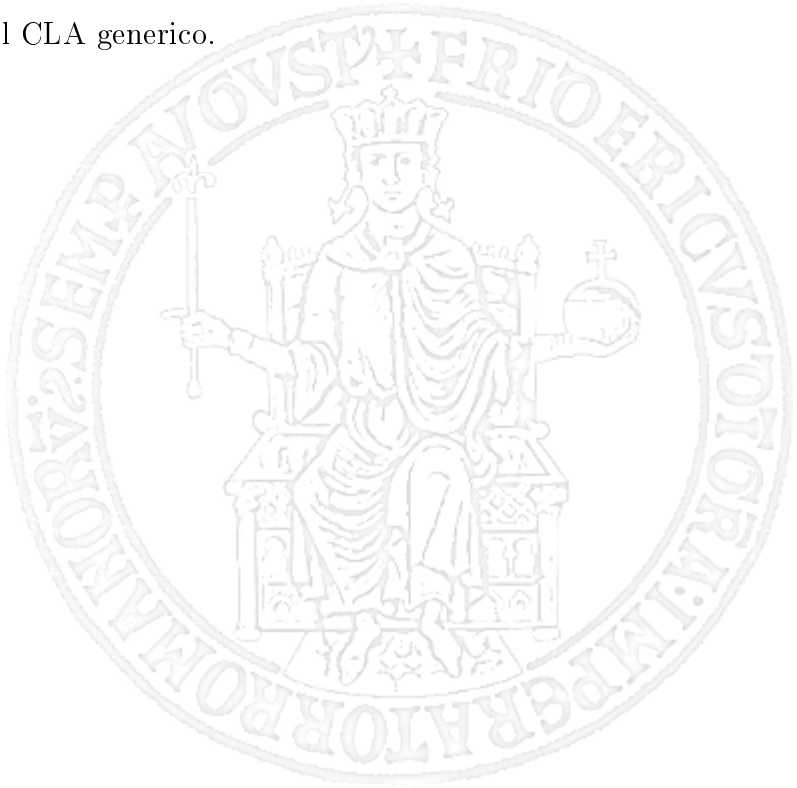
Carry Look Ahead

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

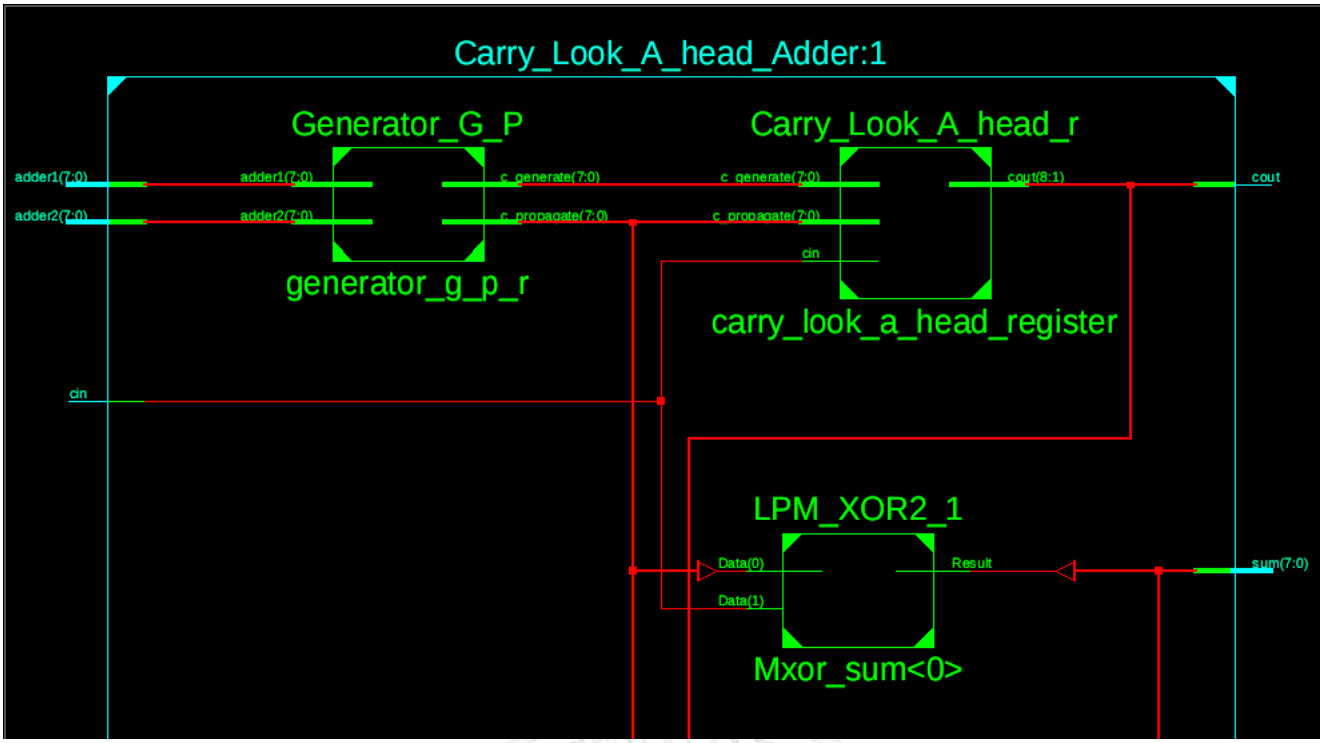


Figura 9.1: Carry Look a Head

L' addizionatore è costituito, da una rete che permette di calcolare i riporti generati e propagati, questi poi vengono propagati ad una rete Carry Look A Head che contemporaneamente combina i riporti ricevuti, altrimenti ci si riconducerebbe al caso Ripple Carry, dopodiché il riporto generato dalla rete Carry Look A Head viene messo in ingresso ad un XOR insieme al ritardo propagato, calcolato precedentemente.

Numero di bit operandi	Numero di slice	Numero di four LUT	Tempo di calcolo
4	6	8	7,190 ps
8	12	19	7,444 ps
16	24	32	9,006 ps
32	48	64	9,139 ps

Secondo le formule dell' area e del ritardo, l' area è pari a $(n^2 + 9n)/2$ volte(dove n indica il numero di porte logiche) , ed $5/delta$, ma se confrontiamo i risultati con quelli ottenuti sintetizzando il Ripple Carry Adder 8.2.1, possiamo osservare che trascurando il caso 8 bit, l' area occupata è pressoché la stessa, ma i tempi di calcolo sono sensibilmente minore, aiutando ad affermare con certezza che sulla nostra FPGA tale soluzione risulta migliore.

9.2.2 Codice

Carry Look A Head ISE

9.2.2.1 Carry Look A Head

```
entity Carry_Look_A_head_r is
   generic(width: Natural :=8);
    Port ( c_generate : in  STD_LOGIC_VECTOR (width-1 downto 0);
           c_propagate : in  STD_LOGIC_VECTOR (width-1 downto 0);
           cin : in  STD_LOGIC;
           cout : out  STD_LOGIC_VECTOR(width downto 1));
end Carry_Look_A_head_r;

architecture Structural of Carry_Look_A_head_r is
  COMPONENT Carry_Look_A_head_unit
  generic(width: NATURAL:=8);
  PORT(
    c_generate : IN std_logic_vector(width-1 downto 0);
    c_propagate : IN std_logic_vector(width-1 downto 0);
    cin : IN std_logic;
    cout : OUT std_logic:='0');
  END COMPONENT;
begin
  C_L_A:for i in 1 to width generate
    inst_C_L_A_U: Carry_Look_A_head_unit generic map (width=>i) port map(
       c_generate(i-1 downto 0),c_propagate(i-1 downto 0),cin,cout(i));
  end generate;
end Structural;
```

Codice Componente 9.1: Definizione del Carry Look a Head

Del Carry Look a Head ne viene generato uno di una dimensione pari al numero di riporti propagati e generati da gestire.

9.3 Simulazione

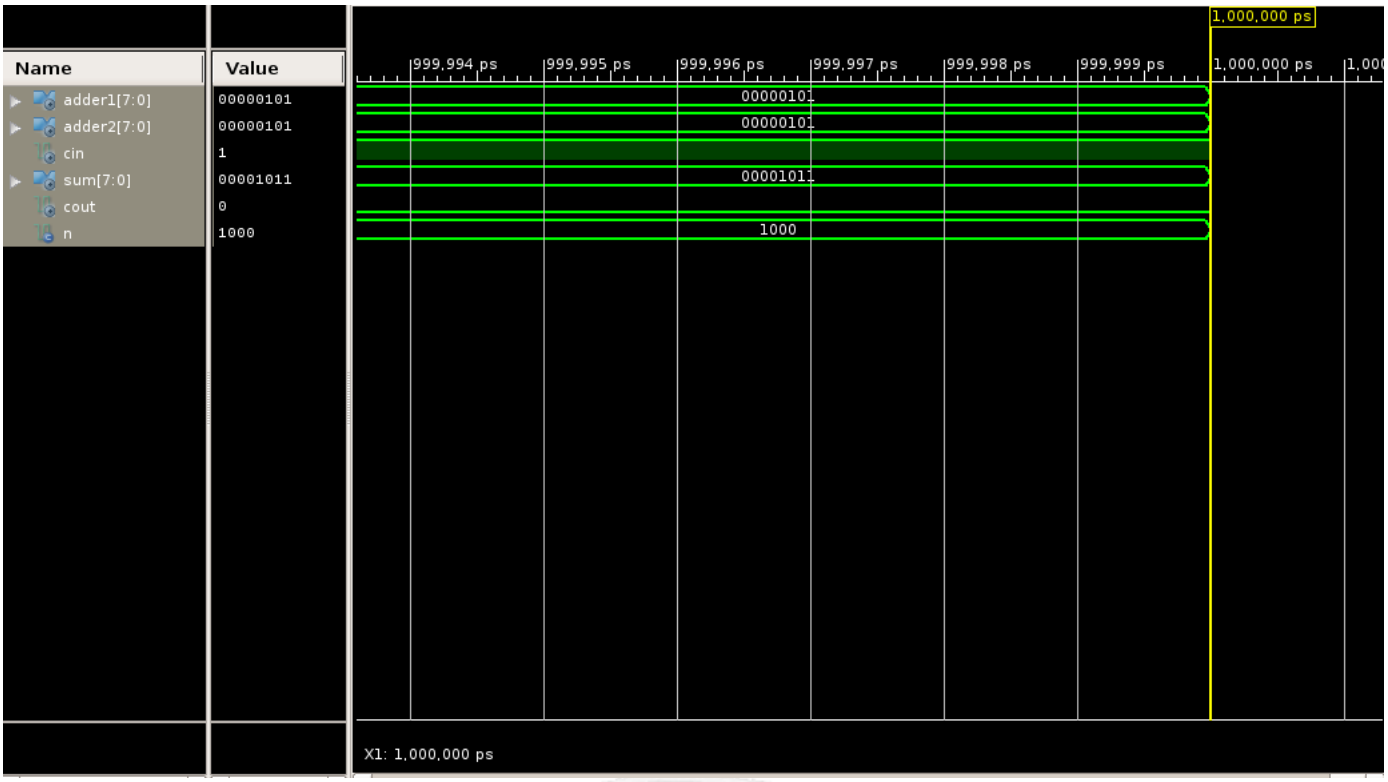


Figura 9.2: Carry Look a Head esempio di somma

9.4 Sintesi su board FPGA

Valgono le stesse considerazioni fatte per il Ripple Carry Adder 8.4.

Capitolo 10

Carry Save

10.1 Traccia

Realizzare un esempio di addizionatore basato sulla modalita Carry Save.

10.2 Soluzione

10.2.1 Schematici

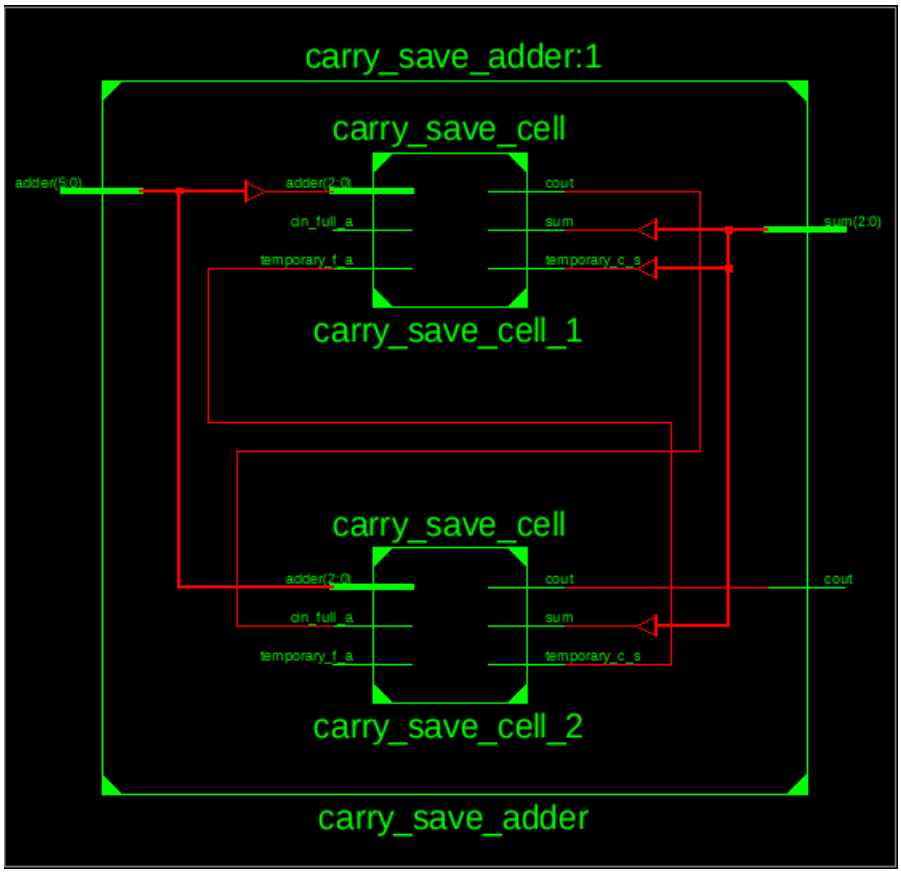


Figura 10.1: Carry Save Adder

Si è scelto di realizzare un semplice addizionatore di tipo carry save che effettua la somma di due operandi da tre bit, nello schematico sono mostrati due carry save cell, queste sono formate da due full adder, uno utilizzato come carry save ed un altro come un full adder, dopodiché i valori calcolari dal primo carry save cell vengono propagati al secondo e la somma è determinata dalla uscita dei full adder.

10.2.2 Codice

Carry Save Adder ISE

10.2.2.1 Carry Save Cell

```
1  entity carry_save_cell is
2      Port ( adder : in  STD_LOGIC_VECTOR (2 downto 0);
3             cin_full_a: in STD_LOGIC;
4          temporary_f_a:in STD_LOGIC;
5          temporary_c_s : out  STD_LOGIC;
6             cout : out  STD_LOGIC;
7             sum : out  STD_LOGIC);
8  end carry_save_cell;
9
10 architecture structural of carry_save_cell is
11 component full_adder_half is
12     PORT( A: in STD_LOGIC;
13       B: in STD_LOGIC;
14       CIN: in STD_LOGIC;
15       S: out STD_LOGIC;
16       COUT: out STD_LOGIC
17     );
18 end component;
19 signal c0:STD_LOGIC:='0';
20 begin
21   carry_save:full_adder_half port map(adder(0), adder(1),adder(2),
        temporary_c_S,c0);
22   full_adder:full_adder_half port map(c0,cin_full_a,temporary_f_a,sum,cout);
23
24 end structural;
```

Codice Componente 10.1: Definizione della carry save cell

Come è realizzata una cella carry save citata sopra.

10.3 Simulazione

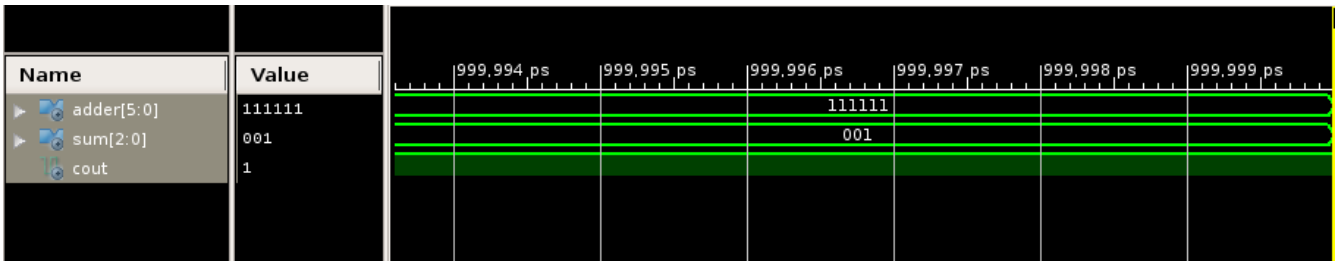


Figura 10.2: Carry Save Adder

10.4 Sintesi su board FPGA

Valgono le stesse considerazioni fatte per il Ripple Carry Adder 8.4, eccezione fatta per gli switch che ne vengono utilizzati solo sei inceve di otto, (tre per un operando e tre per un altro), il led cinque è disabilitato non avendo la bisogno di provare anche quando il carry in ingresso è alto e la sottrazione viene effettuata quando led sette e 6 sono alti.

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

Il sommattore essenzialmente è costituito da un numero di carry select cell, di dimensione identiche (è possibile scegliere la dimensione della cella) che calcolano le somma parziali e propagano i riporti alle celle successive, il numero di celle da istanziare è data dalla dimensione della cella e dal numero di bit su cui viene effettuata la somma, una cella è costituita da due Ripple Carry Adder uno avente carry in ingresso pari ad uno ed un altro pari a zero, di seguito si determinano valori caratteristici del sommatore considerando una carry select cell di dimensione pari a quattro bit.

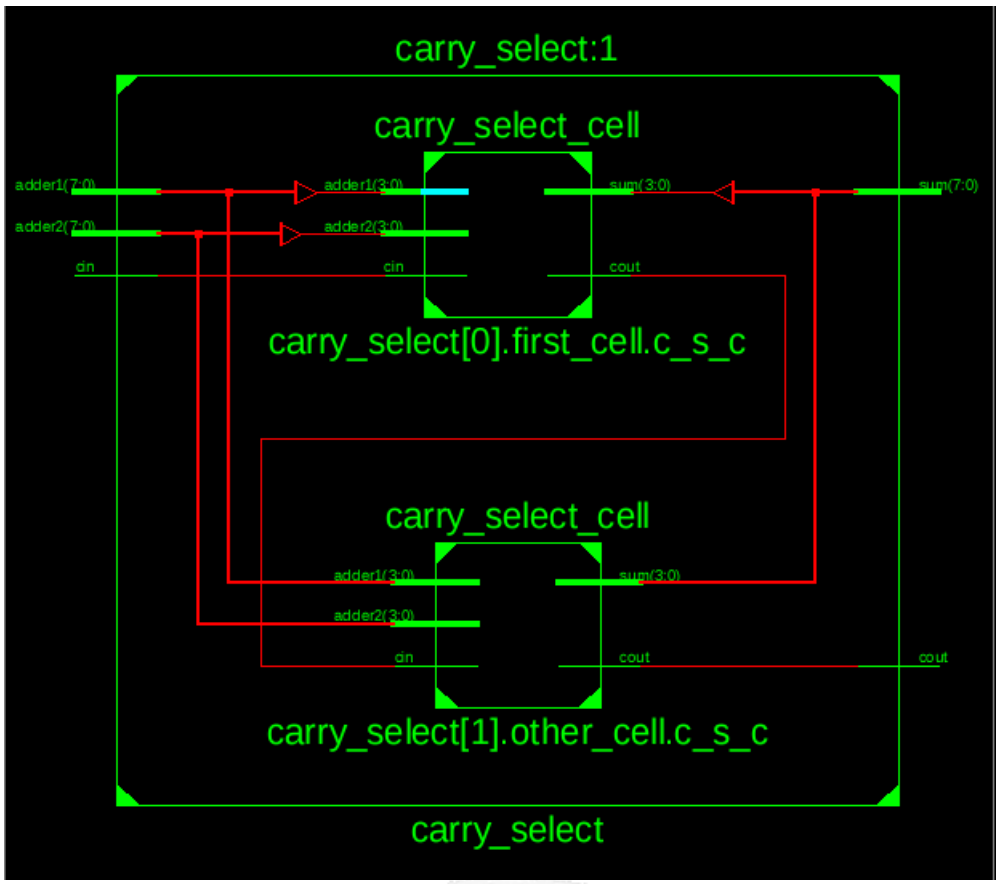


Figura 11.1: Carry Select Adder

Numero di bit operandi	Numero di slice	Numero di four LUT	Tempo di calcolo
4	5	10	6,853 ps
8	12	21	7,467 ps
16	24	43	8,521 ps
32	48	87	9,984 ps

Confrontando i risultati con il sommatore Ripple Carry Adder, 8.2.1 il numero di slice occupate è pressochè lo stesso tranne nel primo caso, invece il numero delle LUT è maggiore, dovute alle interconnessioni che permettono ai Ripple Carry Adder di avere carry in ingresso pari ad uno o zero e a quelle per collegare i vari carry in uscita. I tempi di calolo sono migliori in tutti i casi tranne che nell 'ultimo che sono pressoché gli stessi, l' uguaglianza del tempo di calcolo nell' ultimo caso con quello del Ripple Carry Adder è dovuto che il carry select avendo una cella di dimensione quattro all' aumentare del numero di operandi, aumentano il numero di celle istanziate, il tempo di attesa dell' ultima cella diventa sempre maggiore e tende ad essere comparabile con quello di un Ripple Carry Adder.

11.2.2 Codice

Carry Select Adder ISE

11.3 Simulazione

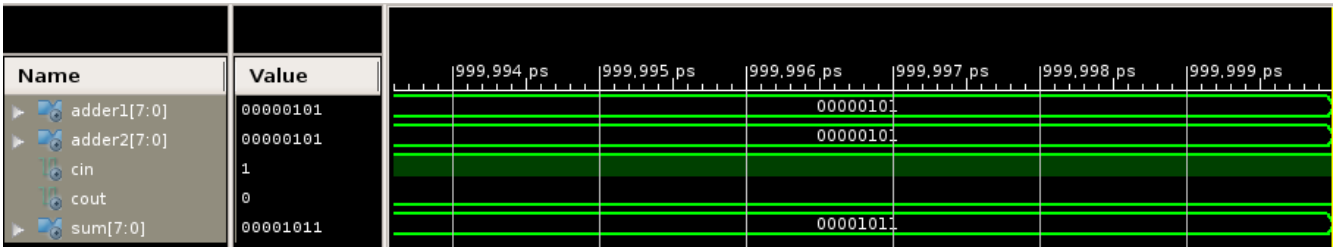


Figura 11.2: Carry Select Adder

11.4 Sintesi su board FPGA

Valgono le stesse considerazioni fatte per il Ripple Carry Adder 8.4.

Capitolo 12

Addizionatore a 7 operandi

12.1 Traccia

Progettare in VHDL un sommatore ad N bit capace di sommare 7 operandi. Il risultato della somma deve essere completo senza overflow, cio'e la somma in uscita deve essere espressa su un numero di bit tali da consentire il non verificarsi di condizioni di overflow. Nel caso specifico i bit in uscita devono essere pari ad N+3. Il progetto pu'o essere realizzato mediante composizione di Full Adder. Per ciascuna colonna si pu'o effettuare un conteggio, producendo in uscita un valore binario espresso su 3 bit. Tenendo conto della posizione dei riporti un sommatore (e.g. ripple carry) pu'o sommare tutti i riporti generati, restituendo il risultato finale. I riporti generati possono propagarsi sino alla cifra i+2 (e.g. la somma di 5 volte 1 genera 1 con riporto 10). Eventuali altre cifre binarie, come i riporti generati da somme precedenti, vanno considerati: quindi si verifica che un riporto pu'o generarsi fino alla i+4-esima cifra binaria.

12.2 Soluzione

12.2.1 Schematici

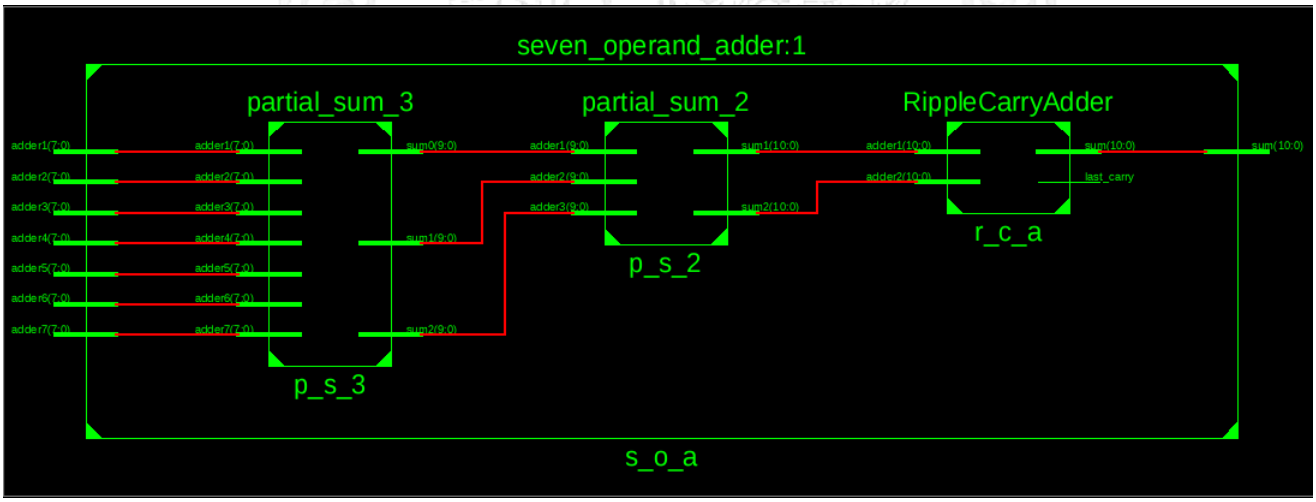


Figura 12.1: Seven Operand Adder

L' addizionatore è costituito dapprima da un dispositivo che trasforma i sette input in ingresso in soli tre input da nove bit ognugno, grazie al componente seven to three il quale conta il numero di uno presenti in ingresso, di questi ne sono presenti otto ed ognugno ha in ingresso bit dello stesso peso appartenenti a ciascuno operando, i valori in uscita di tali componenti prima di essere passati al successivo vengono incolonnati introducendo degli zero fittizzi per fa sì che le tre stringhe di bit in uscita abbiano lo stesso peso(se pensiamo al primo seven to three genera un bit di peso zero, uno di peso uno e l' ultimo di peso due, questi vanno associati a tre strighe diverse ma le stringhe con il bit di peso uno e due hanno bisogno di zero antecedenti per poter incollonare le somme). Il partial sum due non sono altro che dei full adder utilizzati come carry save, per produrre due stringhe che vengono fatte sommare da dei Ripple Carry Adder, facendo le stesse considerazione sui pesi discusse nel passo precedente.

12.2.2 Codice

Seven Adder ISE

12.3 Simulazione

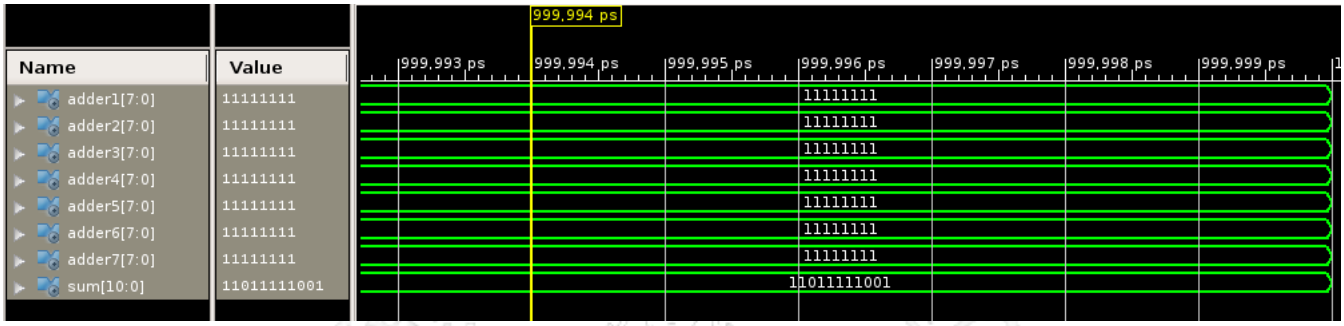


Figura 12.2: Seven Adder esempio di somma

12.4 Sintesi su board FPGA

Valgono le stesse considerazioni fatte per il Ripple Carry Adder 8.4, con la differenza che ci sono sette registri da poter caricare.

Capitolo 13

Moltiplicatori

13.1 Traccia

13.1.1 Moltiplicatore a celle Mac

Realizzare in VHDL un circuito di moltiplicazione a celle MAC di N bit. La cella MAC deve contenere un Full Adder (descritto gi'a in esercizi pre- cedenti) ed una porta AND per la moltiplicazione parziale. Tale cella deve essere replicata in una struttura ordinata (per righe e colonne) per comporre il circuito intero di moltiplicazione. Effettuare considerazioni di occupazione di area e di tempi di propagazione dei segnali al variare di N per valori significativi, apportando eventuali commenti salienti.

13.1.2 Moltiplicatore di Booth

Realizzare in hardware l'algoritmo della moltiplica- zione secondo Booth per operandi ad 8 bit. L'architettura deve essere rea- lizzata sulla base dello schema di progettazione PO/PC (Parte Operativa e Parte di Controllo).

13.2 Soluzione

13.2.1 Schematici

13.2.1.1 Moltiplicatore a celle Mac

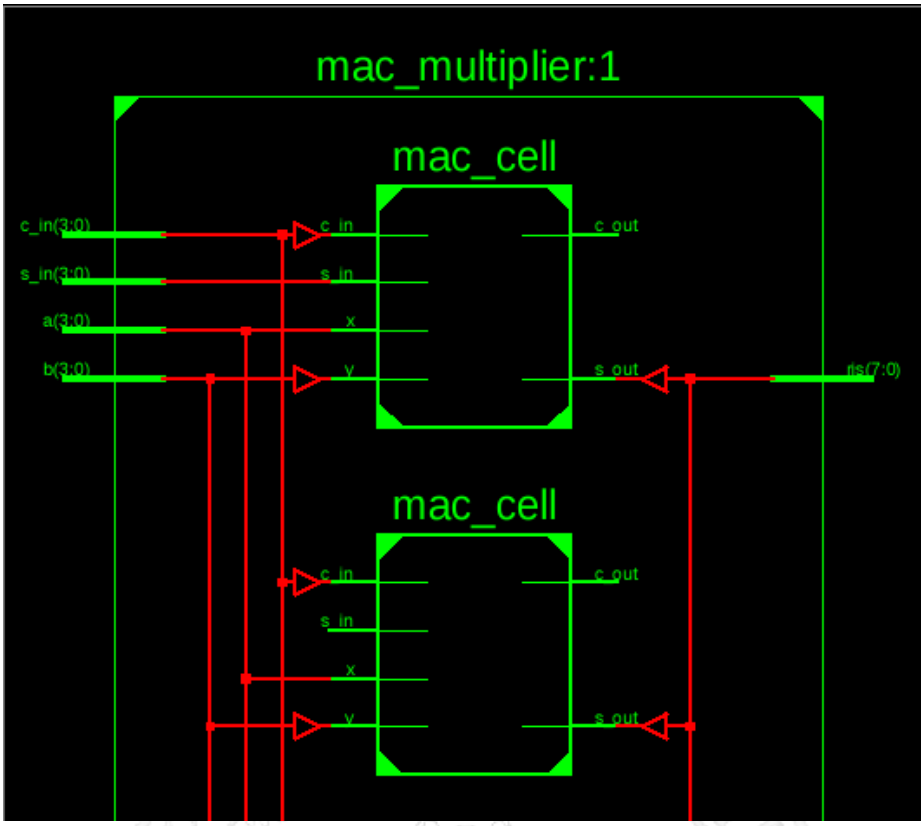


Figura 13.1: Mac Multiplier

Viene mostrato un esempio di moltiplicatore a celle MAC 2x2 (purtroppo lo schematico se preso nella sua interezza non rendeva visibili i segnali), per la gestione dei segnali di interconnessione si è ricorso all' uso di un costrutto matrix, data la simmetria del circuito in cui vengono gestiti i vari carry e somme parziali di ingresso ed uscita, si è scelto di instanziare prima le righe, gestendo le somme parziali in ingresso ed in carry in uscita, alla successiva riga invece vanno gestiti i carry in ingresso e le somme in uscita.

Numero di bit operandi	Numero di slice	Numero di four LUT	Tempo di calcolo	Tempo di calcolo per cel-la(stimato)
2x2	4	8	8,978 ps	1,122 ps
4x4	17	32	14,557 ps	0,7278 ps
8x8	65	128	25,026 ps	0,579 ps
16x16	256	512	51,175 ps	0,556 ps

Possiamo riscontrare che il sintettizzatore, mette in ogni slice una cella MAC, lo spazio occupato quindi per ogni raddoppio di riga e di colonna diventa quattro volte maggiore come anche il numero di LUT quadruplica, per il numero minimale di slice presente sulla BASYS non è possibile sintetizzare un 32x32.

I tempi di calcolo teoricamente dovrebbero essere pari a $6(n-1)T + 2T$ (dove n numero o di riga o di colonna essendo il moltiplicatore simmettrico e T ritardo di una porta), se dalla formula ricaviamo T al variare di n notiamo che il tempo per cella descresce, in realtà le celle vengono messe sempre più vicine durate l' operazione di place and routing, dato che il loro numero aumenta e lo spazio diminuisce, riducendo così la distanza fra di esse ed il tempo di propagazione dei segnali.

13.2.1.2 Moltiplicatore di Booth

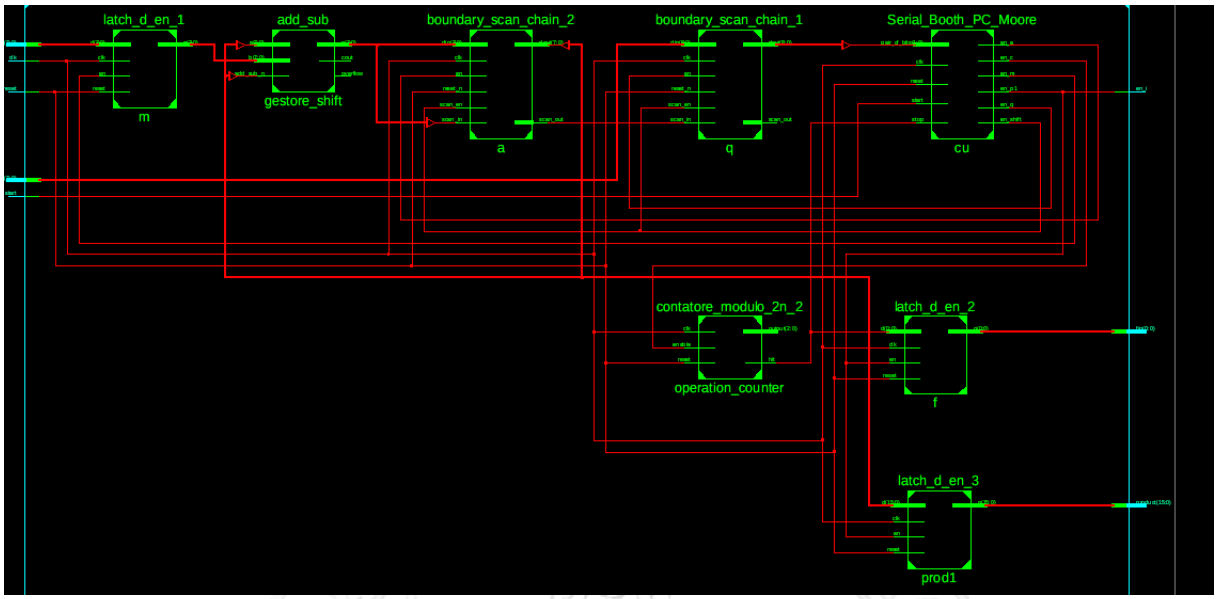


Figura 13.2: Booth Multiplier

Per realizzare il motiplicatore (che utilizza la codifica Booth-1), si è utilizzato una macchina a stati finiti con i seguenti stati:

- idle, stato in cui la macchina non opera; init, vi è l' inizializzazione dei registri e delle scan chain, il moltiplicatore va salvato in un registro, nella scan chain q il moltiplicando e a viene settata a 0;
- in getseq vengono letti gli ultimi due bit in uscita dalla scan chain q, in base ai valori ne si ricava la codifica di Booth e si decide se sommare, sottrarre prima di effettuare lo shifting o solo effettuare quest' ultimo;
- in inits si abilita lo shifting che verrà effettuato al passaggio nel successivo stato shift.

Le scan chain utilizzata per questo progetto e per gli altri dispositivi sequenziali non è quella mostrata nel paragrafo 6.2, in particolare è stato aggiunto un secondo segnale di abilitazione *en* oltre a *scan_ en*: il primo, permette di scegliere cosa shiftare se l' ingresso *din* oppure un valore già precedentemente shiftato, difatti viene sfruttato nel caso in cui avvenga una somma o una

sottrazione, perchè prima di fare lo shiting viene caricato il risultato di tale operazione in a e poi shiftato, se avessi utilizzato la semplice scan chain ogni volta che *scan_en* sarebbe stato attivo avrei shiftato il valore presente nel registro, ma in realtà lo shift avviene un numero di volte pari al numero di bit del moltiplicando che ci da cui si trae la codifica di Booth e si vuole anche decidere se shiftare il valore della somma corrente o di quella precedente. La terminazione si ha quando il contatore genera un segnale di hit, che avviene successivamente all' ultima operazione derivante dalla codifica di Booth.

13.2.2 Codice

Mac ISE

Booth ISE

13.2.2.1 Moltiplicatore a celle Mac

```
entity mac_multiplier is
  generic(
    N : natural := 2;
    M : natural := 2
  );
  port(
    a : in std_logic_vector(N-1 downto 0);
    b : in std_logic_vector(M-1 downto 0);
    s_in : in std_logic_vector(N-1 downto 0);
    c_in : in std_logic_vector(M-1 downto 0);
    ris : out std_logic_vector(M+N-1 downto 0)
  );
end mac_multiplier;

architecture structural of mac_multiplier is

component mac_cell is
  port(
    x : in std_logic;
    y : in std_logic;
    c_in : in std_logic;
    s_in : in std_logic;
    c_out : out std_logic;
    s_out : out std_logic
  );
end component;

type matrix is array(M downto 0, N downto 0) of std_logic;
signal carry : matrix := (others => (others => '0'));-- riga M non usata
signal partial_sum : matrix := (others => (others => '0'));-- (0,0) non
    usato

begin
```

```
33    inizializza_carry : for i in 0 to M-1 generate
34      carry(i,0) <= c_in(i);
35    end generate;
36    inizializza_sum : for j in 0 to N-1 generate
37      partial_sum(0,j+1) <= s_in(j);
38    end generate;
39
40    --partial_sum(1 to M,N) <= carry(0 to M-1, N);
41    --ris <= partial_sum(M,N downto 0) & partial_sum(M-1 downto 1,0); strongly
          typed
42    mac_matrix : for i in 0 to M-1 generate -- si genera una riga per tutte le
          colonne si gestiscono le somme parziali e i carry in uscita
43      mac_row : for j in 0 to N-1 generate -- dopodichÃš si procede alla
            successiva riga
44        mac : mac_cell
45          port map(
46            x => a(j),-- posiziona i bit di un operando sulla colonna
47            y => b(i), -- posiziona i bit di un operando su una righa
48            c_in => carry(i,j),-- i carry in ingresso di uno stadio precedente
                  per i primi mac di riga e colonna pari a 0
49            s_in => partial_sum(i,j+1), -- somma parziale ottenuta da uno
                stadio precedente
50            c_out => carry(i,j+1), -- carry in uscita per la prossima cella
                mac
51            s_out => partial_sum(i+1,j) -- somma in uscita per il successivo
                carry
52          );
53      end generate;
54      partial_sum(i+1,N) <= carry(i,N); -- riporto i carry in uscita presenti
          all' ultima colonna di ogni riga come somma parziale in ingresso
55      ris(i) <= partial_sum(i+1,0); -- i primi bit della moltiplicazione
          risultano dalla somma parziale in uscita dai primi elementi in
          colonna di ogni riga
56    end generate;
57
58    risultato_completo : for z in 1 to N generate
59      ris(z+M-1) <= partial_sum(M,z);-- gli ultimi valori di prodotto sono
          dati dai valori di somma partiale dell' ultima riga
60    end generate;
61  end structural;
```

Codice Componente 13.1: Definizione della carry save cell

13.3 Simulazione

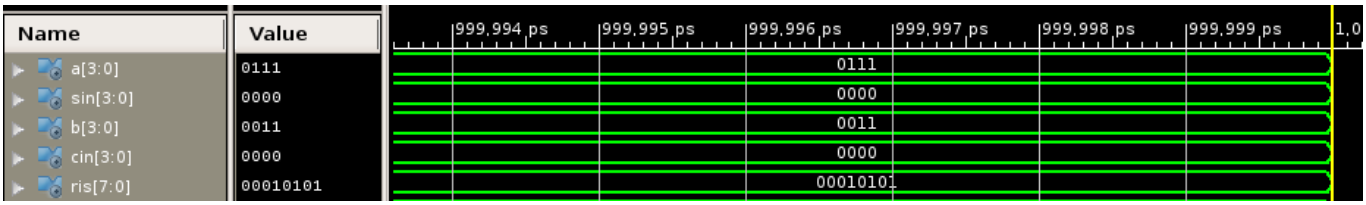


Figura 13.3: Esempio di moltiplicazione a celle Mac

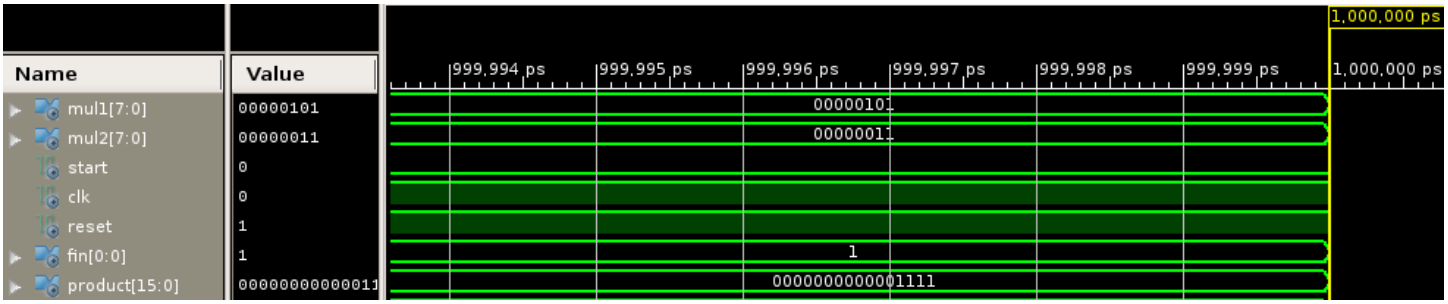


Figura 13.4: Esempio di moltiplicazione utilizzando Booth

13.4 Sintesi su board FPGA

13.4.0.1 Moltiplicatore a celle MAC

Valgono le stesse considerazioni fatte per il Ripple Carry Adder 8.4, però i led zero, uno e cinque non vengono utilizzati ed i vari segnali di abilitazione sottrazione e carry non sono presenti.

13.4.0.2 Moltiplicatore di Booth

Valgono le stesse considerazione per il riconoscitore di stringhe 7.4, dove l' accensione del led zero indica la fine dell' operazione, il led uno non viene utilizzato mentre invece il display è impiegato come in 8.4.

Capitolo 14

Divisori

14.1 Traccia

Realizzare in hardware l'algoritmo della divisione Restoring per operandi ad 8 bit. L'architettura deve essere realizzata sulla base dello schema di progettazione PO/PC (Parte Operativa e Parte di Controllo).

14.2 Soluzione

14.2.1 Schematici

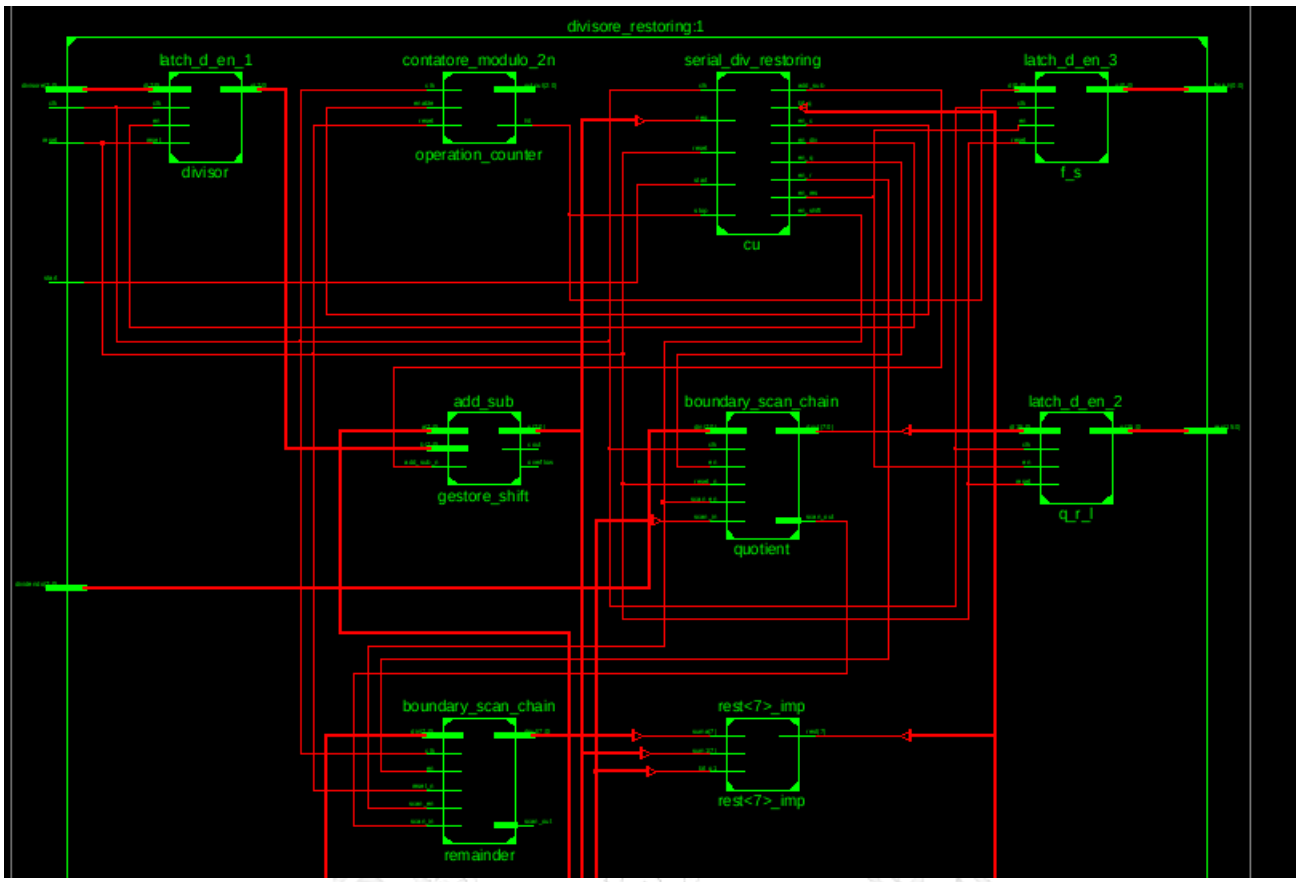


Figura 14.1: Divisore restoring

Il circuito è molto simile a quello utilizzato per sviluppare un moltiplicatore di Booth 13.2.1.2, cambia radicalmente però in che modo i bit vengono trattati difatti la macchina a stati finiti consiste di sei stati:

- idle stato di riposo, attende il segnale di avvio di una computazione;
- init, in cui avviene l' inizializzazione delle scan chain, in questo caso q ed r (q caricato con il valore del dividendo ed r inzializzato a 0) e un registro per salvare il valore del divisore;
- shift in cui i dati (q ed r) vengono shiftati a sinistra (in Booth i dati shiftano a destra);
- sub in cui viene effettuata la sottrazione, nel caso il risultato sia un numero negativo si ritorna in shift altrimenti, si procede nello stato set_1 che setta ad uno il bit da dover shiftare all' interno di q, tale shift avverrà nello stato shift1.

Il segnale di terminazione della computazione viene attivato quando il contatore segnala un numero di operazioni pari al numero di bit del dividendo.

14.2.2 Codice

Divisore Restoring ISE

14.3 Simulazione

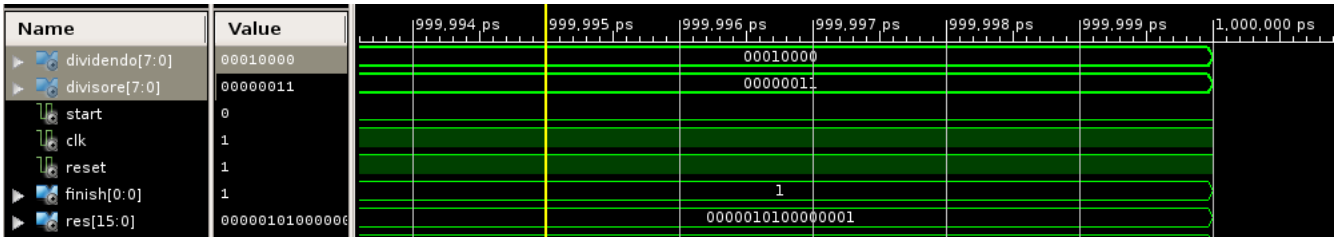


Figura 14.2: Esempio divisione

Il quoziente ed il resto vengono espresse in un unica stringa da 16 bit dove la metà superiore della stringa è il quoziente la metà inferiore il resto, questo per facilitare successivamente la scrittura sul display.

14.4 Sintesi su board FPGA

Valgono le stesse considerazione per il riconoscitore di stringhe 7.4, dove l' accensione del led zero indica la fine dell' operazione ed il led uno non viene utilizzato, viene caricato prima il dividendo e poi il divisore, infine le prime due cifre del display indicano il quoziente le seconde il resto ed è utilizzato come descritto qui 8.4.

Capitolo 15

UART

15.1 Traccia

Realizzare un dispositivo VHDL che implementa il protocollo UART (a partire da quello diffuso dalla Digilent). Collegare internamente, oppure tramite inter- faccia fisica esterna alla board stessa, ad un'altra board oppure ad un PC previo utilizzo di un physical RS232, due interfacce per trasmette e ricevere ottetti. Svolgere l'esercizio riutilizzando il VHDL messo a disposizione da Digilent (e disponbile nel materiale del corso) commentando eventuali ristrutturazioni del codice. (Opzionale) Sviluppare un'architettura per l'implementazione del protocollo UART secondo il paradigma PO/PC ex-novo, evidenziando le similarit'a/dissimilarit'a con il progetto Digilent.

15.2 Soluzione

15.2.1 Schematici

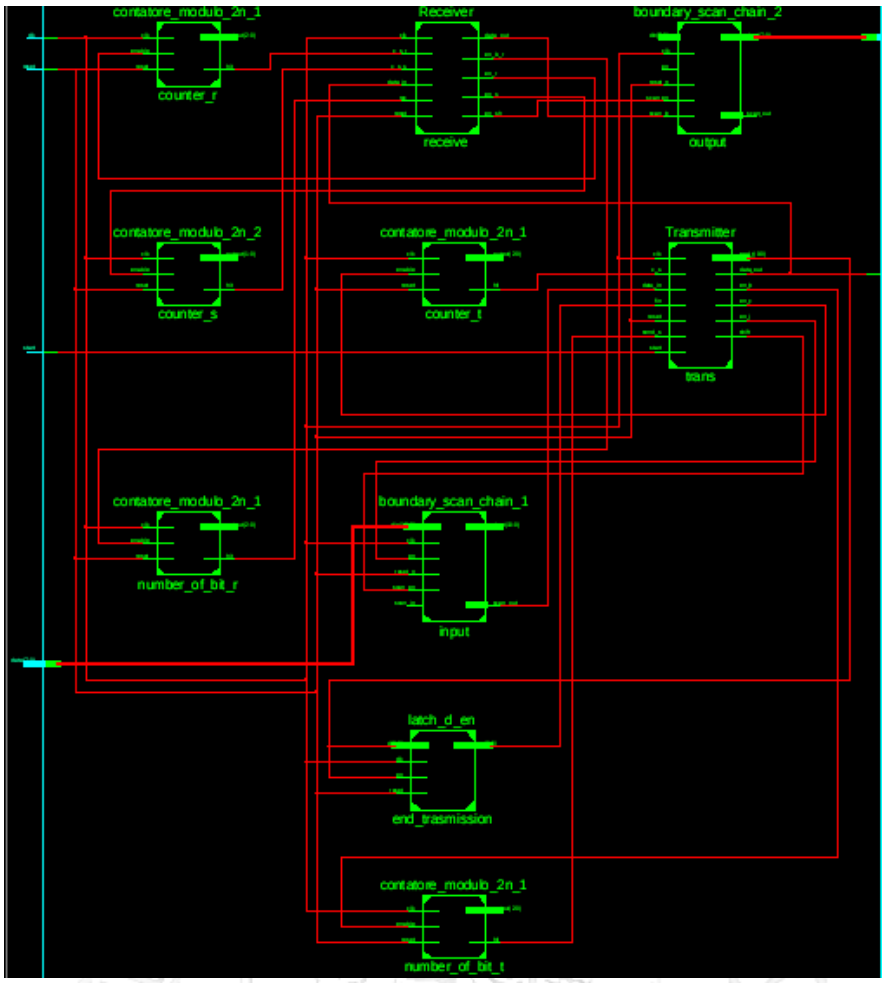


Figura 15.1: UART

Si è realizzata una versione di UART secondo lo schema PO/PC, i componenti principali solo le due macchine a stati finiti:

- il trasmitter che consiste di cinque stati:
  - idle, rimane in attesa fino a che non si decide di avviare una trasmissione;
  - send_start invia il segnale di start;
  - wait_for_next attende fino a che il contatore counter_t non genera un segnale di counter hit, per far sì prima che venga inviata la successiva informazione, quella attuale venga sorretta per sedici colpi di clock;
  - send che abilita lo shifting della scan chain con il dato ingresso per invare il successivo bit;
  - send_stop che invia il bit di stop, si abilita questo stato quando il numero di bit trasmetti è pari ad otto (il conteggio è fissato da un contatore).

- Il ricevitore invece è composto solo da quattro stati:
  - idle, si permane in questo stato fino a quando la linea dei dati in ingresso è alta;
  - sfasamento fa in modo di posizionare l' evento di lettura del dato a metà del tempo per il quale è sorretto il dato, così da essere sicuri di leggere correttamente il bit;
  - wait_for_next dove si attende per leggere il successivo bit;
  - receive dopo si effettua la lettura del bit in ingresso.

Il ricevitore ha diversi contatori, uno che tiene in conto del numero di bit letti affinchè dopo aver letto l' ottavo si ritorna in idle, un altro che conta il numero di cicli di clock per lo sfasamento ed infine uno che conta il numero di colpi di clock da attendere prima di leggere il successivo dato.

I dati in input ed in output sono sorretti da due scan chain come quelle utilizzate in Booth 13.2.1.2, quella dell' input è più grande di tre bit perchè vi è un primo bit alto per indicare che non vi è trasmissione di dati fino a che non vi è un invio di dati, il secondo è il bit di start ed il terzo messo in coda ai dati è quello di stop.

Premettendo che la soluzione di Digilent effettua vari controlli sulla trasmissione qui non effettuati, la riteniamo di non facile comprensione, scrivere in un unico file VHDL sia il ricevitore, trasmettitore, i contatori e tutti gli altri componenti a supporto non permette di rendere chiara la lettura, non è lampante quali parti siano a supporto del ricevitore e quali del ricevitore, non permettendo così una modifica veloce nel caso in cui volessimo attendere ad esempio trentadue cicli di clock prima di inviare ricevere un altro dato o estrapolare uno solo dei due componenti e riutilizzarlo (pensiamo al caso in cui dobbiamo solo inviare i dati e non riceverli), invece nella nostra soluzione possiamo estrapolare una delle due macchine a stati finiti all' occorrenza e nell' unico file in cui sono riacchiusi i componenti a corredo capire quali ci occorrono ed riutilizzarli, oppure in base ai segnali in ingresso alle due macchine sequenziali capire quali segnali ci possano occorrere, altro punto a sfavore è la realizzazione di diversi process per fare andare la macchina ad una frequenza voluta per trasmettere/ricevere ad un determinato baud rate, soluzione migliore consisterebbe di utilizzare un DCM e cambiare la frequenza in base alla necessità.

15.2.2 Codice

UART ISE

15.3 Simulazione

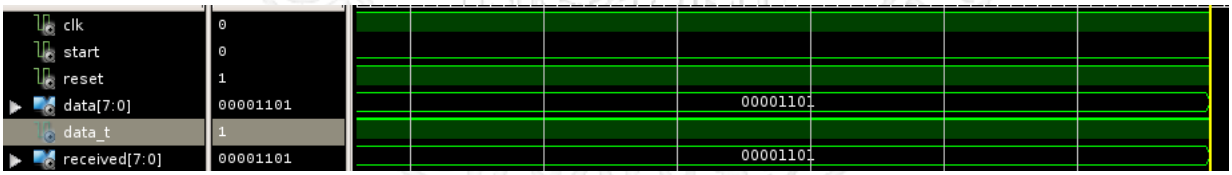


Figura 15.2: UART

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

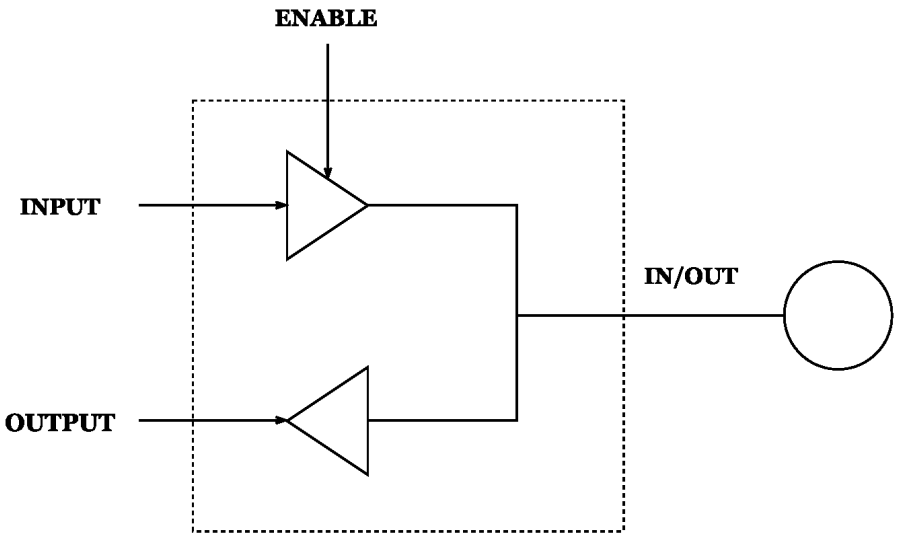


Figura 16.1: GPIO Schematic

16.2.2 Codice

Progetto ISE: GPIO ISE

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 quindi, per evitare di disturbare 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 pin di un 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 ed infine il segnale *outputs* è stato collegato ai primi quattro degli otto led della board.

Capitolo 17

Firma digitale

17.1 Traccia

Realizzare un'architettura per la realizzazione di un sistema che effettui l'autenticazione di un messaggio a lunghezza variabile tramite l'algoritmo di crittografia RSA. Il sistema deve consentire di effettuare sia la fase di firma del messaggio, sia la fase di verifica del messaggio firmato.

17.2 Soluzione

17.2.1 Schematici

Il seguente circuito implementa l' algortimo RSA per la firma di un messaggio, applica una funzione di hash sul messaggio e verifica che il messaggio ricevuto sia coretto.

L' intera procedura è divisa nelle seguenti fasi:

- vengono scelti i valori caratteristici per inviare il dato (p pari a 3, q 11, e 7 e d 3) ed il messaggio da inviare;
- si applica la funzione di hashing sul messaggio, utilizzando il metodo della moltiplicazione;
- viene applicata la firma sul messaggio originale utilizzando la chiave privata, il trasmettitore invia i due dati appena calcolati (anche se non è presente un vero e proprio invio essendo trasmettitore e ricevitore implementati sulla stessa board);
- Il ricevitore applica la chiave pubblica sul messaggio firmato, ne effettua l' hashing e verifica se la sua versione del messaggio a cui è stato applicato l' hashing è identico a quello che è stato ricevuto.

Così facendo garantiamo sia la segretezza(infatti non mandiamo il messaggio in chiaro), sia l' autenticazione(chi riceve il messaggio può verificare effettuando l' hashing con parametri convenuti non è stato modificato),

Di seguito vengono descritte le varie componenti che vengono utilizzate per effettuare l' hashing e firmare il messaggio.

17.2.1.1 Funzione hash

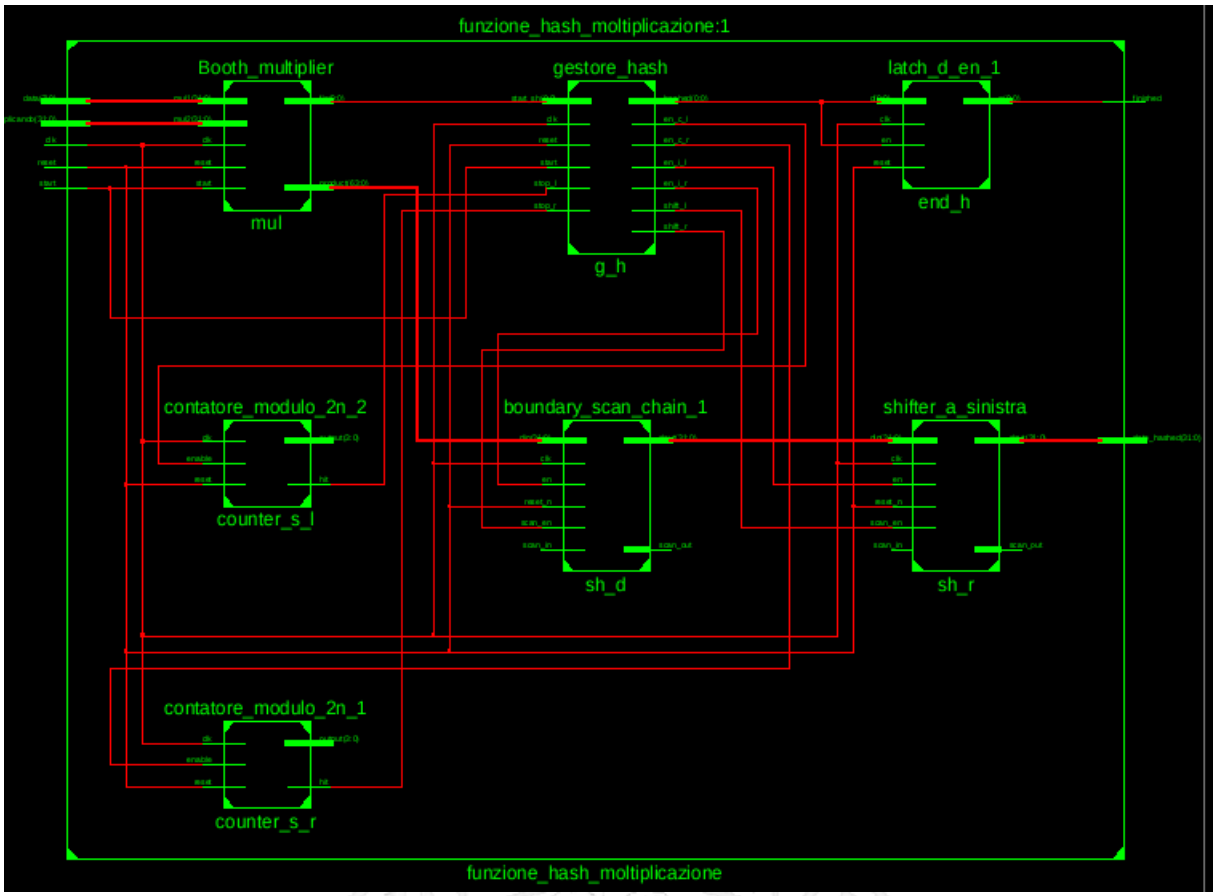


Figura 17.1: hasher

Il circuito è stato realizzato in due differenti modi:

- nel primo caso si utilizza una macchina a stati finiti descritta da cinque stati:
  - idle, stato di riposo dell' automa;
  - init in cui si attende che il moltiplicatore termini il suo compito;
  - shifting_r in cui il valore della moltiplicazione viene shiftato a destra;
  - shifting_l il valore viene shiftato a sinistra; ended per comunicare la fine dell' operazione di hash;
- nel secondo si è utilizzato lo stesso procedimento ma il tutto viene implementato in un process.

Viene utilizzata la seconda soluzione perché occupa meno spazio, dato che le operazioni di shifting consistono semplicemente, nel caso dello shifting a destra prendere i sedici bit più significativi dal risultato della moltiplicazione, quando si shifta a sinistra basta accodare sedici zeri per avere lo stesso risultato della prima soluzione.

Il circito viene così descritto poiché tale forma di hashing prevedere di moltiplicare il dato per un valore A il quale è un numero compreso tra 0 ed 1, di cui il denominatore è un multiplo di

due, per tale ragione di è scelto di moltiplicare per un valore W ed infine di shiftare a destra un numero di volte pari al valore dell' esponente, tale numero è la parte decimale del valore del dato per W (perché quando shiftiamo inseriamo degli zero fittizi), dopodichè si effettua uno shifting a sinistra di un numero pari di volte affinché la nostra parte deciamale rientri nella parte del registro da inviare.

17.2.1.2 Esponenziatore

Per effettuare l' elevazione a potenza con il modulo, ci siamo riferiti a questo algoritmo :

```
Modular-Exponentiation (a,b,n)
c ← 0
d ← 1
for i ← k downto 0
    do c ← 2c
        d ← (d·d) mod n
        if b_i = 1
            then c ← c + 1
            d ← (d·a) mod n
return d
```

Figura 17.2: Algoritmo per modular exponentiational di un messaggio

L ' algoritmo cicla sul numero di bit dell ' esponente, moltiplica il valore d prima per se stesso e ne fa il modulo, questo stesso valore di d per la base quando il valore dell' esponente in forma binaria assume valore uno e ne effettua il modulo, (il parametro c non occorre per il calcolo del valore finale, occorre solo a capire in realtà alla fine se il numero è stato elevato per il corretto esponente).

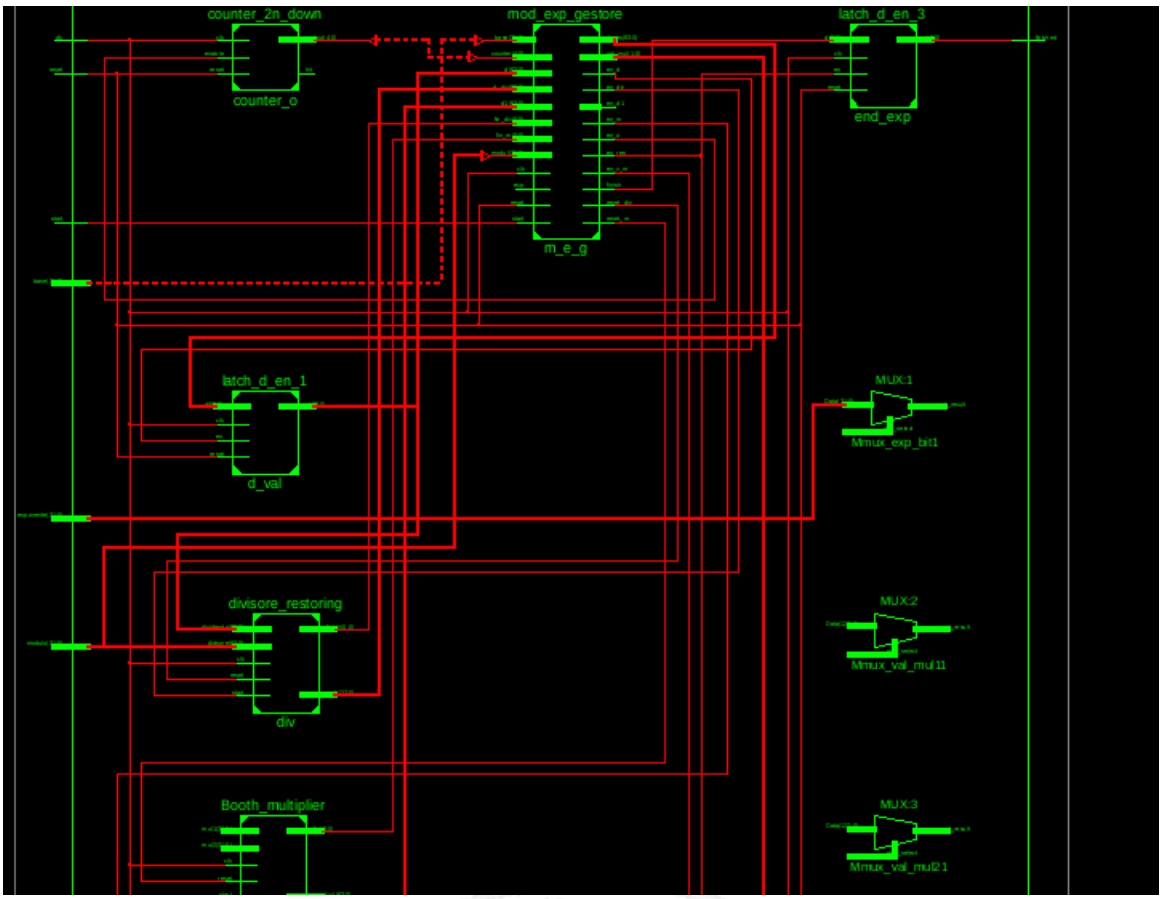


Figura 17.3: Hardware Exponentiational

Utiliziamo questo algoritmo perché ci permette di riutilizzare componenti già sviluppati negli esercizi precedenti, difatti osservando lo schematico, vi è presente un moltiplicatore di Booth e il divisore restoring per le varie operazioni prodotto e modulo, un contatore down per indicare quale dei bit dell' esponente dobbiamo analizzare e dei selettori descritti con il costrutto with select per selezionare quali valori devono essere moltiplicati. La macchina a stati finiti non fa altro che eseguire i vari passi dell' algoritmo, però per problemi relativi al timing è stata realizzata alla fine con un singolo process, difatti una prima realizzazione con due process determinava che alcuni registri contenti i dati dell' operazione assumessero un valore indefinito (veniva sintetizzata un macchina che doveva avere una frequenza di clock minore da quella generabile dalla scheda).

Per problemi di spazio, si è ricorsi a componenti per la moltiplicazione e divisione seriali, oltre ad riutilizzarli all' interno dello stesso progetto difatti il moltiplicatore di Booth è stato utilizzato per: calcolare il prodotto di pq ed l' esponenziazione.

17.2.2 Codice

RSA ISE

17.3 Simulazione

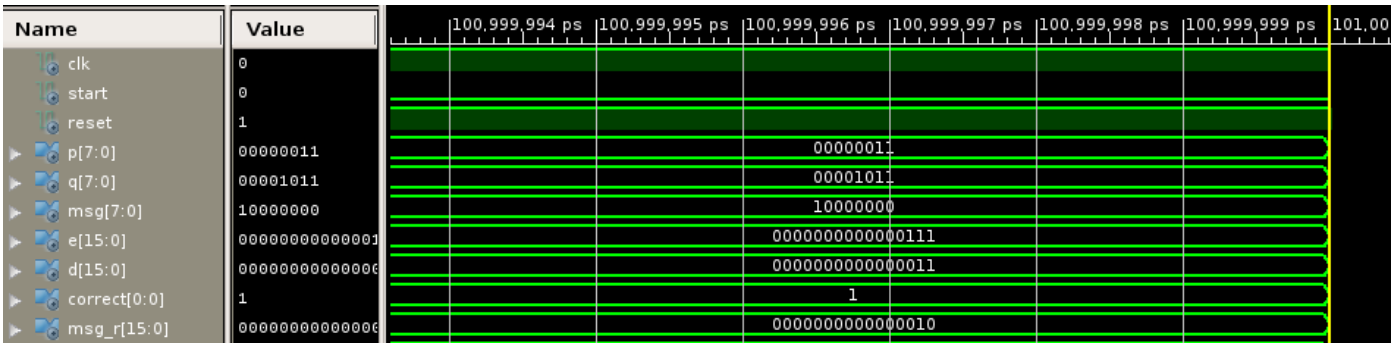


Figura 17.4: Esempio di computazione dei dati in cui il messaggio ricevuto ed inviato non coincidono

17.4 Sintesi su board FPGA

Valgono le stesse considerazione per il riconoscitore di stringhe 7.4, dove l' accensione del led zero indica che il messaggio ricevuto è diverso da quello di cui si è effettuato l' hashing, infatti se il messaggio è corretto tale led non viene abilitato. Dato il numero maggiore di dati in input i registri da caricare sono in numero superiore difatti viene accesso anche il led cinque per indicare quale registro si sta caricando, il caricamento del messaggio avviene quando led cinque e sei sono alti, si utilizza il display come descritto qui 8.4 e viene mostrata la versione firmata del messaggio inviato, il circuito che effettua la firma viene resettato quando i led sette e sei sono accesi.