

Tesina di Architettura dei Sistemi di Elaborazione
Gruppo GF(2)

Giamattei Luca - Mat. M63/000825 Farina Giorgio - Mat. M63/000861

27 marzo 2019

Indice

1 Minimizzazione reti combinatorie	1
1.1 Esercizio 01	1
1.1.1 Traccia	1
1.1.2 Soluzione	1
1.1.2.1 CLASSE 1	1
1.1.2.2 CLASSE 2	2
1.1.2.3 CLASSE 3	2
1.1.2.4 CLASSE 4-5-6	3
1.2 Esercizio 02	4
1.2.1 Traccia	4
1.2.2 Ottimizzazione esatta tramite Mc Cluskey in modo manuale	4
1.2.2.1 Ottimizzazione esatta tramite Mc Cluskey in modo manuale di Y0	4
1.2.2.2 Ottimizzazione esatta tramite Mc Cluskey in modo manuale di Y1	4
1.2.2.3 Ottimizzazione esatta tramite Mc Cluskey in modo manuale di Y2	6
1.2.3 Ottimizzazione esatta tramite Mc Cluskey con utilizzo di CAD	8
1.2.4 Mapping Tecnologico	11
1.3 Esercizio 03	12
1.3.1 Traccia	12
1.3.2 Esercizio	12
1.4 Esercizio 04	13
1.4.1 Traccia	13
1.4.2 Esercizio	13
1.5 Esercizio 05	16
1.5.1 Traccia	16
1.5.2 Contatore di 6 bit implementato tramite Contatori di 3bit (Full Adder)	17
1.5.3 Implementazione in VHDL	18
1.6 Esercizio 06	19
1.6.1 Traccia	19
1.6.2 Modello strutturale di Sommatore di 6 stringhe da tre bit	20
1.6.3 Implementazione in VHDL	20
2 Latch/Flip Flop	23
2.1 Latch RS	23
2.1.1 Traccia	23
2.1.2 Soluzione	23
2.1.2.1 Simulazione Behavioral	24

2.1.2.2	Simulazione Post Route	25
2.1.2.3	Condizioni di Modo fondamentale e Funzionamento	26
2.1.2.4	Durata minima di permanenza dell'ingresso per passare da uno stato totale stabile all'altro stato totale stabile	28
2.1.2.5	Conseguenze quando la durata minima di permanenza dell'ingresso non è rispettata	28
2.1.3	Traccia	29
2.1.4	Soluzione	29
2.1.4.1	Simulazione Post Route	30
2.2	Latch T	31
2.2.1	Traccia	31
2.2.2	Esercizio	31
2.2.2.1	Più commutazioni delle uscite	32
2.2.2.2	Caso in cui vi sono oscillazioni	33
2.2.2.3	Caso in cui si ha una seconda commutazione	33
2.2.2.4	Conclusione	33
2.3	Flip Flop D	33
2.3.1	Traccia	33
2.3.2	Esercizio	34
2.3.3	Funzionamento corretto	38
2.3.3.1	Struttura con alea	38
2.3.3.2	Struttura senza alea	38
2.4	Flip Flop T	39
2.4.1	Traccia	39
2.4.2	Schematic	39
2.4.3	Simulazione Behavioral	40
2.5	FlipFlop D Master Slave	40
2.5.1	Traccia	40
2.5.2	Esercizio	40
3	Display a 7 segmenti	46
3.1	Display a 7 Segmenti	46
3.1.1	Traccia	46
3.1.2	Soluzione	46
3.1.2.1	Descrizione strutturale del modulo Display_Seven_Segment	47
4	Clock Generator	51
4.0.1	Traccia	51
4.0.2	Soluzione	51
4.0.2.1	Schematici	52
4.0.2.2	Codice	52
4.0.3	Simulazione	54

5 Scan Chain	57
5.1 Scan Chain	57
5.1.1 Traccia	57
5.1.2 Soluzione	57
5.1.2.1 Modalità normale	58
5.1.2.2 Modalità Shift	58
5.1.2.3 Una panoramica strutturale alto livello della Scan Chian	59
5.1.3 Implementazione	59
5.1.3.1 Shift Register	59
5.1.3.2 Contatore Modulo N con Numero di Conteggi Massimo come ingresso	62
5.1.3.3 Control Unit	64
5.1.3.4 Scan Chain	68
5.1.3.5 Simulazioni Post Route	70
6 Finite State Machine	71
6.1 Finite State Machine	71
6.1.1 Traccia	71
6.1.2 Diagramma degli stati	71
6.1.3 Implementazione	72
6.1.3.1 Codice VHDL	73
6.1.3.2 Codifica degli stati e valutazioni	76
6.1.3.3 Simulazione Behavioral	81
7 Ripple Carry e Carry Look Ahead	82
7.1 Ripple Carry Adder/Carry Look Ahead	82
7.1.1 Traccia	82
7.1.2 Esercizio	82
7.1.2.1 Implementazione	83
7.1.2.2 Caratteristiche	86
8 Carry Save	88
8.1 Carry save	88
8.1.1 Traccia	88
8.1.2 Soluzione	88
8.1.3 Implementazione	89
8.1.3.1 Componente CSL	89
8.1.3.2 Componente CSL_LEVEL	90
8.1.3.3 Componente Contenitore	92
8.1.3.4 Componente CSL_TREE	93
8.1.3.5 Schematic RTL della soluzione	98
9 Carry Select	99
9.1 Carry Select Adder	99
9.1.1 Traccia	99
9.1.2 Esercizio	99
9.1.2.1 Funzionamento	99

9.1.2.2	Implementazione	100
9.1.2.3	Considerazioni su area occupata e tempo di calcolo	103
10 Moltiplicatori		107
10.1	Moltiplicatore a celle MAC	107
10.1.1	Traccia	107
10.1.2	Esercizio	107
10.1.2.1	Implementazione	108
10.1.2.2	Considerazioni area e ritardo	110
10.2	Robertson	110
10.2.1	Traccia	110
10.2.2	Soluzione	110
10.2.2.1	Interfaccia del Moltiplicatore di Robertson	111
10.2.3	Implementazione	111
10.2.3.1	Lo stato Idle	111
10.2.3.2	Lo stato Init	111
10.2.3.3	Lo stato Scelta_operazione	111
10.2.3.4	Lo stato Shift_stato	112
10.2.3.5	Lo stato Add_sub	112
10.2.3.6	Gestione dei quattro casi di prodotto di due numeri relativi	114
10.2.3.7	Come i quattro casi sono gestiti nella rete	115
10.3	Booth	116
10.3.1	Traccia	116
10.3.2	Soluzione	116
10.3.3	Implementazione	117
10.3.3.1	Lo Shift Register Q	117
10.3.3.2	La Control Unit	117
10.3.3.3	Il valore logico attribuito alla porta effettiva Scan_in dello Shift Register A	117
10.3.3.4	L' addizionatore sottrattore	118
10.3.3.5	Il risultato	118
10.4	Analisi di Booth e Robertson	119
10.4.1	Tempificazioni	119
10.4.2	Simulazioni Post-Route	119
11 Divisori		120
11.1	Divisore Restoring	120
11.1.1	Traccia	120
11.1.2	Soluzione	120
11.1.2.1	Implementazione	121
11.1.2.2	Codice	123
11.1.2.3	Simulazione	131
11.2	Divisore Non Restoring	132
11.2.1	Traccia	132
11.2.2	Soluzione	132
11.2.2.1	Implementazione	132

11.2.2.2 Codice	134
11.2.2.3 Simulazione e confronto	137
12 Uart	138
12.1 UART	138
12.1.1 Traccia	138
12.1.2 Soluzione	138
12.1.2.1 Implementazione	139
12.1.2.2 Codice	140
13 MIC	147
13.1 MIC 1	147
13.1.1 Traccia	147
13.1.2 Introduzione	147
13.1.3 Datapath	147
13.1.4 Controllo Microprogrammato	148
13.1.4.1 La funzione che calcola il Prossimo stato:	149
13.1.4.2 La funzione che calcola l'uscita (i segnali di controllo)	149
13.1.4.3 Conclusioni	149
13.1.4.4 Microlinguaggio e MAL	150
13.1.4.5 Descrizione delle variabili di decisione: come determinare quale sarà la microistruzione successiva	150
13.1.4.6 Descrizione dei Segnali di controllo	151
13.1.5 Tempificazione	151
13.1.6 Soluzione	152
13.1.6.1 Macroarchitettura	152
13.1.6.2 Decodifica indirizzi	153
13.1.6.3 Esempio di Programma	155
13.1.6.4 Ciclo di sviluppo	156

Capitolo 1

Minimizzazione reti combinatorie

1.1 Esercizio 01

1.1.1 Traccia

Si progetti una macchina M che, data una parola X di 6 bit in ingresso ($X_5, X_4, X_3, X_2, X_1, X_0$), restituisca una parola Y di 3 bit (Y_2, Y_1, Y_0) che rappresenta la codifica binaria del numero di bit alti in X.

1.1.2 Soluzione

Per la progettazione della macchina M si è pensato di definire la classica tabella di verità, però per essere coerenti con la soluzione trovata per l'esercizio due, si costruirà la tabella di verità in un modo leggermente diverso. La traccia richiede di contare quanti 1 si hanno in una stringa di sei bit, al posto di elencare tutte le combinazioni degli ingressi partendo da 0 fino a 63 e ponendo di fianco le uscite, ci si chiederà quali sono e quante sono le stringhe con zero 1, un 1,..sei 1.

La stringa che ha zero uno è quella composta da tutti zero.

Si utilizzerà la formula $n!/k!(n - K)!$ per calcolarsi quante sono le permutazioni per cui si hanno k valori alti su n=6 bits per poi alencarle.

1.1.2.1 CLASSE 1

Le permutazioni per cui ho un 1 ($k=1$), banalmente sono 6 e si ottengono facendo variare l'1 in una posizione tra le sei disponibili.

X_5	X_4	X_3	X_2	X_1	X_0
0	0	0	0	0	1
0	0	0	0	1	0
...
1	0	0	0	0	0

Table 1.1: Permutazioni Classe 1

1.1.2.2 CLASSE 2

Le permutazioni per cui ho due 1 ($k=2$) sono 15 e l'elenco si ricava a partire dalla CLASSE 1.

X5	X4	X3	X2	X1	X0
0	0	0	0	1	1
0	0	0	1	0	1
0	0	1	0	0	1
0	1	0	0	0	1
1	0	0	0	0	1
0	0	0	1	1	0
0	0	1	0	1	0
0	1	0	0	1	0
1	0	0	0	1	0
...
1	1	0	0	0	0

Table 1.2: Permutazioni Classe 2

1.1.2.3 CLASSE 3

Le permutazioni per cui ho tre 1 ($k=3$) sono 20 e l'elenco si ricava a partire dalla CLASSE 2.

In particolare le prime 5 permutazioni della classe due diventeranno $4+3+2+1$ permutazioni della classe 3, le successivi 4 permutazioni diventeranno $3+2+1$ permutazioni nella classe 3, ... la terzultima e la punultima permutazioni della

classe due diverranno una permutazione nella classe 3.

X5	X4	X3	X2	X1	X0		X5	X4	X3	X2	X1	X0		X5	X4	X3	X2	X1	X0
0	0	0	1	1	1		0	0	1	1	1	0		1	1	1	0	0	0
0	0	1	0	1	1		0	1	0	1	1	0							
0	1	0	0	1	1		1	0	0	1	1	0							
1	0	0	0	1	1														
0	0	1	1	0	1		0	1	1	0	1	0							
0	1	0	1	0	1		1	0	1	0	1	0							
1	1	0	1	0	1		1	1	0	0	1	0							
0	1	1	0	0	1														
1	0	1	0	0	1														
1	1	0	0	0	1														

Table 1.3: Permutazioni Classe 3

1.1.2.4 CLASSE 4-5-6

Per la classe 4 il ragionamento è analogo, ma bisogna far muovere gli zeri. Quindi le classi 4,5,6 si ottengono come complemento a 1 delle permutazioni delle classi 0,1,2.

La progettazione della macchina è definita in questo modo:

Y0 dovrà essere alta quando ho in ingresso una delle permutazioni appartenenti alla classe 1, 3, 5 ($6+20+6=32$ mintermini).

Y1 dovrà essere alta quando ho in ingresso una delle permutazioni appartenenti alla classe 2, 3, 6 ($15+20+1=36$ mintermini).

Y2 dovrà essere alta quando ho in ingresso una delle permutazioni appartenenti alla classe 4, 5, 6 ($15+6+1=22$ mintermini).

Di seguito è mostrata la rappresentazione sotto forma tabellare che esprime la relazione ingressi-uscite

CLASSE	Y2	Y1	Y0	#Perm	#Mintermini Y2	#Mintermini Y1	#Mintermini Y0
0	0	0	0	1			
1	0	0	1	6			6
2	0	1	0	15		15	
3	0	1	1	20		20	20
4	1	0	0	15	15		
5	1	0	1	6	6		6
6	1	1	0	1	1	1	

Table 1.4: Uscite

1.2 Esercizio 02

1.2.1 Traccia

Si derivi la forma minima (SOP) per ciascuna delle variabili in uscita dalla macchina M (considerate separatamente l'una dall'altra) utilizzando lo strumento SIS, e si confronti la soluzione trovata dal tool con quella ricavabile con una procedura esatta manuale (Karnaugh o Mc-Cluskey). Per una delle uscite si effettui anche il mapping su una delle librerie disponibili in SIS e si commentino i risultati ottenuti in diverse modalità di sintesi.

1.2.2 Ottimizzazione esatta tramite Mc Cluskey in modo manuale

Le classi di Mc Cluskey coincidono con le classi delle permutazioni. Di seguito facendo riferimento anche alla tabella dell'esercizio precedente verrà effettuata l'ottimizzazione esatta di Mc Cluskey separatamente per ogni uscita.

1.2.2.1 Ottimizzazione esatta tramite Mc Cluskey in modo manuale di Y0

Per l'uscita Y0 si hanno le classi numero uno, tre, e cinque. Poichè le classi non sono adiacenti non può essere applicata la fase di espansione (**non può essere applicata la proprietà di assorbimento**).

Di conseguenza non si avranno implicanti, o meglio **i mintermini sono i nostri implicanti primi**. Questo significa che è inutile anche fare la copertura poichè ogni implicante primo, che è un mintermine, coprirà esclusivamente il medesimo mintermine. Quindi gli **implicanti primi, ovvero i mintermini, risultano tutti essenziali**.

Di conseguenza la copertura minima della funzione è **la somma di tutti i mintermini delle classi uno, tre e cinque per un totale di 32 mintermini**

1.2.2.2 Ottimizzazione esatta tramite Mc Cluskey in modo manuale di Y1

Per l'uscita Y1 si hanno le classi numero due, tre, e sei. La fase di espansione potrà tenere in considerazione solo le due classi adiacenti due e tre.

Ogni mintermine della classe 2 ha un match con i mintermini della classe tre che hanno due uno nelle stesse posizioni del mintermine di classe 2 altrimenti ci sarà più di un bit che varia.

Ogni mintermine della classe 2 ha una corrispondenza con 4 elementi della classe 3. In particolare per ogni mintermine della classe 2 si ottengono quattro implicanti con due '1' (posizionati dove stavano gli '1' del mintermine considerato) più un don't care al posto di uno '0' che varia nelle quattro posizioni rimanenti (esempio per il mintermine 000011 ho 000-11,00-011,0-0011,-00011)

Si ricava quindi una copertura prima di **61 implicanti primi** di cui **15*4 implicanti primi** che ricoprono sia la classe 2 che la 3 più tutti i mintermini della classe 6 (il mintermine '**111111**').

Per ricavare la copertura minima al posto di rappresentarla essendo di 61 righe e 36 colonne si seguirà un ragionamento.

Il Mintermine della classe sei è l'unico a coprire se stesso e quindi funge da implicante essenziale.

Gli implicanti compriranno sempre un mintermine della classe 2 e uno della classe 3.

Quindi non vi è dominanza di righe perchè non vi è un implicante che implica 3 mintermini.

CAPITOLO 1. MINIMIZZAZIONE RETI COMBINATORIE

Per quanto riguarda la dominanza delle colonne r.

Ogni mintermine di classe due è coperto 4 volte e ogni mintermine di classe 3 è coperto 3 volte (come mostrato nella Tab. 1.5 per due mintermini generali rispettivamente uno di classe 2 e uno di classe 3). Non si ha una dominanza di colonne perchè non vi è mai una colonna di un mintermine di terza classe che è contenuta interamente in un'altra di seconda classe.

Allora per ottenere una copertura minima si potrebbero usare, essendo una tabella ciclica, i metodi di supporto alla copertura come Branch and Bound, considerando come metodo di costo la cardinalità.

La cardinalità migliore a cui si possa aspirare è 20: in quanto ogni implicante copre un solo elemento di classe due e uno di classe tre, quindi si può aspirare al massimo a scegliere 20 implicanti sperando di coprire i 20 mintermini di classe 3 e contemporaneamente i 15 di classe 2.

Si potrebbero anche scegliere i 20 implicanti prendendo 15 implicanti che coprano tutta la classe due, ma anche altri altrettanti 15 della classe 3, per poi sceglierne altri 5 che coprano 1:1 i rimanenti 5 mintermini della classe 3.

X5	X4	X3	X2	X1	X0	001010					001011	001110	011010	101010	
0	0	1	0	1	-	X				X					
0	0	1	-	1	0	X					X				
0	-	1	0	1	0	X						X			
-	0	1	0	1	0	X							X		
0	0	1	0	-	1					X					
0	0	-	0	1	1					X					

Table 1.5: Struttura parziale della tabella di Copertura prima per due mintermini generali '001010' di classe 2 e '001011' di classe 3

Per avere come risultato una copertura minima, ovvero di cardinalità 20, si seguirà il primo ragionamento.

Si riprenda la costruzione della classe 3 a partire dalla classe 2, si può ottenere la copertura minima prendendo tutte le permutazioni della classe tre con al posto dell'uno (che si era aggiunto dal passaggio della classe due alla classe tre) un don't care.

Questo procedimento è rappresentato nella Tab.1.6

Si sono ottenuti così i 20 implicanti primi, che coprono una volta la classe 3 e più volte la classe 2, i quali faranno parte della copertura minima della funzione Y1 insieme all'implicante essenziale (una copertura minima di 21 implicanti).

CAPITOLO 1. MINIMIZZAZIONE RETI COMBINATORIE

X5	X4	X3	X2	X1	X0	X5	X4	X3	X2	X1	X0
0	0	0	-	1	1	0	0	-	1	1	0
0	0	-	0	1	1	0	-	0	1	1	0
0	-	0	0	1	1	-	0	0	1	1	0
-	0	0	0	1	1						
						0	-	1	0	1	0
0	0	-	1	0	1	-	0	1	0	1	0
0	-	0	1	0	1						...
-	1	0	1	0	1	-	1	0	0	1	0
0	-	1	0	0	1						
-	0	1	0	0	1						
-	1	0	0	0	1						

Table 1.6: Copertura minima della classe 2 e 3

1.2.2.3 Ottimizzazione esatta tramite Mc Cluskey in modo manuale di Y2

Per l'uscita Y2 si hanno le classi numero quattro, cinque, e sei.

Le classi sono tutte e tre adiacenti. Le classe 4 e 5 sono il complemento a uno rispettivamente della classe 2 e della classe 1 . La classe sei è costituita dal solo mintermine '111111'.

X5	X4	X3	X2	X1	X0
1	1	1	1	0	0
1	1	1	0	1	0
1	1	0	1	1	0
1	0	1	1	1	0
0	1	1	1	1	0
1	1	1	0	0	1
1	1	0	1	0	1
1	0	1	1	0	1
0	1	1	1	0	1
...
0	0	1	1	1	1

Table 1.7: Classi 4, 5 e 6

Il risultato della corrispondenza tra la classe 4 con la 5 è che per ogni mintermine della classe 4 si ha un match con due mintermini della classe 5, quindi si hanno due nuovi implicanti per ogni mintermine di classe 4 mettendo un don't care alternativamente nei due 0 del mintermine di classe 4 considerato. Si ottiene quindi una tabella risultante con 30 righe.

Il risultato della corrispondenza tra la classe 5 con la classe 6 è la classe 5 con al posto degli zeri i don't care.

Questi due risultati sono mostrati nella Tab. 1.8.

X5	X4	X3	X2	X1	X0		X5	X4	X3	X2	X1	X0	
1	1	1	1	0	-		1	1	1	1	1	-	
1	1	1	1	-	0		1	1	1	1	-	1	
1	1	1	0	1	-		
1	1	1	-	1	0		-	1	1	1	1	1	
1	1	0	1	1	-								
1	1	-	1	1	0								
1	0	1	1	1	-								
1	-	1	1	1	0								
0	1	1	1	1	-								
-	1	1	1	1	0								
1	1	1	0	-	1								
1	1	1	-	0	1								
1	1	0	1	0	1								
1	1	-	1	-	1								
1	0	1	1	-	1								
1	-	1	1	0	1								
0	1	1	1	-	1								
-	1	1	1	0	1								
...								
0	-	1	1	1	1								
-	0	1	1	1	1								

Table 1.8: Risultati delle corrispondenze della classe 4 con la classe 5 e della classe 5 con la classe 6

Successivamente, come mostrato nella Tab. 1.9, gli implicanti primi risultanti dalla corrispondenza tra le due classi sono 30. Si osserva facilmente che l'implicante 1 è uguale al 2, il 3 è uguale al 4 e così via, quindi **sono 15 implicanti primi e coincidono con la classe 5 mettendo al posto degli 0 i don't care**.

The diagram illustrates the reduction of a Karnaugh map. On the left, a large Karnaugh map (15 implicants) is shown with columns labeled X5, X4, X3, X2, X1, X0. The rows are filled with binary values (1 or -). An arrow points from this map to a smaller Karnaugh map on the right (15 essential prime implicants), which also has columns X5, X4, X3, X2, X1, X0. This reduction indicates that all 15 implicants are essential for covering all minterms in the original map.

X5	X4	X3	X2	X1	X0
1	1	1	1	-	-
1	1	1	1	-	-
1	1	1	-	1	-
1	1	1	-	1	-
1	1	-	1	1	-
1	1	-	1	1	-
1	-	1	1	1	-
1	-	1	1	1	-
-	1	1	1	1	-
-	1	1	1	1	-
1	1	1	-	-	1
1	1	1	-	-	1
1	1	-	1	-	1
1	1	-	1	-	1
1	-	1	1	-	1
1	-	1	1	-	1
-	1	1	1	-	1
-	1	1	1	-	1
...
-	-	1	1	1	1
-	-	1	1	1	1

X5	X4	X3	X2	X1	X0
1	1	1	1	-	-
1	1	1	-	1	-
1	1	-	1	1	-
1	-	1	1	1	-
-	1	1	1	1	-
1	1	1	-	-	1
1	1	-	1	-	1
1	-	1	1	-	1
-	1	1	1	-	1
...
-	-	1	1	1	1

Table 1.9: Nella prima tabella sono mostrati gli implicanti risultanti dalla corrispondenza tra le due classi risultanti. Nella seconda tabella semplicemente si è eliminata una ridondanza evidente.

Posizionando sulle righe i 15 implicanti primi e sulle colonne i 22 mintermini si scopre che i 15 implicanti sono essenziali per tutti i mintermini di classe 4 e quindi li scelgo tutti e 15 per la copertura minima.

1.2.3 Ottimizzazione esatta tramite Mc Cluskey con utilizzo di CAD

Si è deciso di utilizzare lo strumento ESPRESSO perché rispetto a SIS è possibile avere informazioni più dettagliate riguardo i singoli passaggi della minimizzazione. Quindi, dopo aver ricavato il file in formato .pla, è stato possibile applicare la funzione “espresso -t -Dexact Y1.pla” dove:

- “-t” fornisce un ampio sommario relativo alla esecuzione
- “-Dexact” permette di eseguire l’algoritmo di minimizzazione esatta basato sui principi della procedura di Quine-Mc Cluskey

Di seguito si riporta il risultato ottenuto applicando l’algoritmo alla variabile di uscita Y1:

```

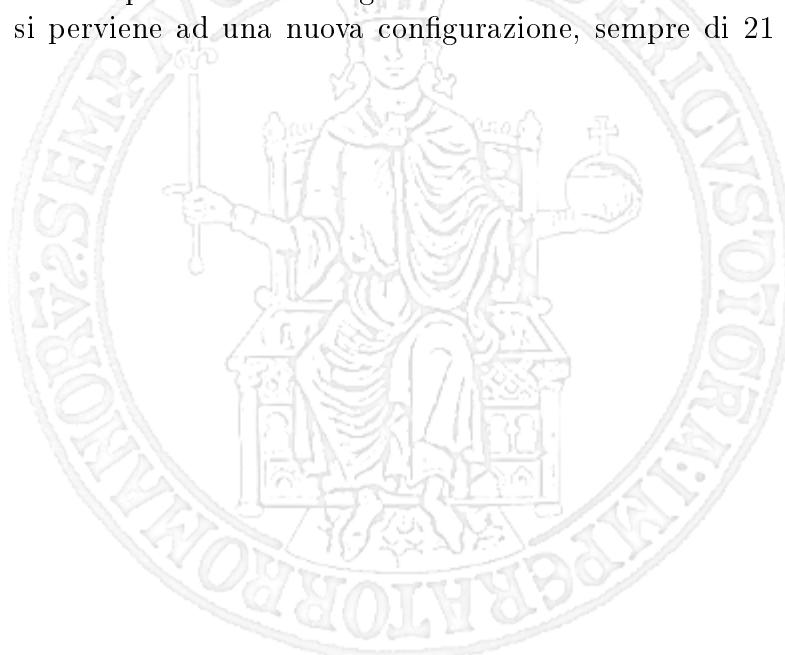
.i 6
.o 1
.ilb a b c d e f
.ob z
.p 21
111111 1
10010- 1
01100- 1
10100- 1
11000- 1
0001-1 1
0100-1 1
1000-1 1
00-011 1
001-01 1
010-01 1
001-10 1
100-10 1
0-0110 1
0-1010 1
-01010 1
-10010 1
-10100 1
1-1000 1
.e

# espresso -t -Dexact Y1.pla
# UC Berkeley, Espresso Version #2.3, Release date 01/31/88
# READ      Time was 0.00 sec, cost is c=36(36) in=216 out=36 tot=252
# COMPL     Time was 0.00 sec, cost is c=22(6) in=110 out=22 tot=132
# PLA is Y1.pla with 6 inputs and 1 outputs
# ON-set cost is  c=36(36) in=216 out=36 tot=252
# OFF-set cost is c=22(6) in=110 out=22 tot=132
# DC-set cost is c=0(0) in=0 out=0 tot=0
# PRIMES     Time was 0.00 sec, cost is c=61(61) in=306 out=61 tot=367
# ESSENTIALS Time was 0.00 sec, cost is c=1(1) in=6 out=1 tot=7
# PI-TABLE   Time was 0.00 sec, cost is c=60(60) in=300 out=60 tot=360
# MINCOV     Time was 0.00 sec, cost is c=61(61) in=306 out=61 tot=367
# MV_REDUCE  Time was 0.00 sec, cost is c=21(21) in=106 out=21 tot=127
# VERIFY     Time was 0.00 sec, cost is c=21(21) in=106 out=21 tot=127
# READ       1 call(s) for 0.00 sec ( 0.0%)
# COMPL     1 call(s) for 0.00 sec ( 0.0%)
# MV_REDUCE 1 call(s) for 0.00 sec ( 0.0%)
# VERIFY     1 call(s) for 0.00 sec ( 0.0%)
# exact Time was 0.00 sec, cost is c=21(21) in=106 out=21 tot=127
.e

```

Figure 1.1: Y1

Come ci si aspettava dopo aver effettuato la procedura manuale, partendo da un totale di 36 mintermini per l'ON-set (dovuti ai 15 di classe 2 + 20 di classe 3 + 1 di classe 6) in seguito alla fase di espansione si è arrivati ad un totale di 61 implicanti primi ($15 \times 4 + 1$) di cui uno soltanto essenziale. Dopo la fase di ricerca della copertura minima si è arrivati alla medesima soluzione trovata manualmente, ovvero un totale di 21 implicanti. Si vuole inoltre far notare che, questa configurazione di implicanti che individuano una copertura minima non è l'unica possibile, come precedentemente anticipato. Infatti eseguendo una nuova minimizzazione con l'algoritmo full_simplify di SIS si perviene ad una nuova configurazione, sempre di 21 implicanti, tuttavia differenti:



```
.model Y1
.inputs a b c d e f
.outputs z
.names a b c d e f z
000-11 1
00-011 1
0-0011 1
-00011 1
00-101 1
0-0101 1
-00101 1
01-001 1
-01001 1
1-0001 1
0011-0 1
0101-0 1
-00110 1
01-010 1
-01010 1
1-0010 1
011-00 1
101-00 1
1-0100 1
11-000 1
111111 1
.end
```

Figure 1.2: Nuova Configurazione

Per completezza si riportano di seguito le minimizzazioni effettuate anche per le variabili Y0 ed Y2:



```

# espresso -t -Dexact Y0.pla
# UC Berkeley, Espresso Version #2.3, Release date 01/31/88
# READ           Time was 0.00 sec, cost is c=32(32) in=192 out=32 tot=224
# COMPL          Time was 0.00 sec, cost is c=32(17) in=192 out=32 tot=224
# PLA is Y0.pla with 6 inputs and 1 outputs
# ON-set cost is c=32(32) in=192 out=32 tot=224
# OFF-set cost is c=32(17) in=192 out=32 tot=224
# DC-set cost is c=0(0) in=0 out=0 tot=0
# PRIMES          Time was 0.00 sec, cost is c=32(32) in=192 out=32 tot=224
# ESSENTIALS      Time was 0.00 sec, cost is c=32(32) in=192 out=32 tot=224
# PI-TABLE         Time was 0.00 sec, cost is c=0(0) in=0 out=0 tot=0
# MINCOV          Time was 0.00 sec, cost is c=32(32) in=192 out=32 tot=224
# MV_REDUCE        Time was 0.00 sec, cost is c=32(32) in=192 out=32 tot=224
# VERIFY           Time was 0.00 sec, cost is c=32(32) in=192 out=32 tot=224
# READ             1 call(s) for 0.00 sec ( 0.0%)
# COMPL            1 call(s) for 0.00 sec ( 0.0%)
# MV_REDUCE        1 call(s) for 0.00 sec ( 0.0%)
# VERIFY           1 call(s) for 0.00 sec ( 0.0%)
# exact Time was 0.00 sec, cost is c=32(32) in=192 out=32 tot=224
# espresso -t -Dexact Y2.pla
# UC Berkeley, Espresso Version #2.3, Release date 01/31/88
# READ           Time was 0.00 sec, cost is c=22(22) in=132 out=22 tot=154
# COMPL          Time was 0.00 sec, cost is c=20(20) in=60 out=20 tot=80
# PLA is Y2.pla with 6 inputs and 1 outputs
# ON-set cost is c=22(22) in=132 out=22 tot=154
# OFF-set cost is c=20(20) in=60 out=20 tot=80
# DC-set cost is c=0(0) in=0 out=0 tot=0
# PRIMES          Time was 0.00 sec, cost is c=15(15) in=60 out=15 tot=75
# ESSENTIALS      Time was 0.00 sec, cost is c=15(15) in=60 out=15 tot=75
# PI-TABLE         Time was 0.00 sec, cost is c=0(0) in=0 out=0 tot=0
# MINCOV          Time was 0.00 sec, cost is c=15(15) in=60 out=15 tot=75
# MV_REDUCE        Time was 0.00 sec, cost is c=15(15) in=60 out=15 tot=75
# VERIFY           Time was 0.00 sec, cost is c=15(15) in=60 out=15 tot=75
# READ             1 call(s) for 0.00 sec ( 0.0%)
# COMPL            1 call(s) for 0.00 sec ( 0.0%)
# MV_REDUCE        1 call(s) for 0.00 sec ( 0.0%)
# VERIFY           1 call(s) for 0.00 sec ( 0.0%)
# exact Time was 0.00 sec, cost is c=15(15) in=60 out=15 tot=75

```

Figure 1.3: Y0, Y2

Come ci si aspettava, non è stato possibile eseguire nessun tipo di minimizzazione sull'uscita Y0.

1.2.4 Mapping Tecnologico

Per effettuare il mapping si è deciso di utilizzare la libreria “mcnc.genlib” fornita dallo strumento SIS:

```

GATE inv1      1      0!=a;          PIN * INV 1 999 0.9 0.3 0.9 0.3
GATE inv2      2      0!=a;          PIN * INV 2 999 1.0 0.1 1.0 0.1
GATE inv3      3      0!=a;          PIN * INV 3 999 1.1 0.09 1.1 0.09
GATE inv4      4      0!=a;          PIN * INV 4 999 1.2 0.07 1.2 0.07
GATE nand2     2      0=!(a*b);    PIN * INV 1 999 1.0 0.2 1.0 0.2
GATE nand3     3      0=!(a*b*c);  PIN * INV 1 999 1.1 0.3 1.1 0.3
GATE nand4     4      0=!(a*b*c*d); PIN * INV 1 999 1.4 0.4 1.4 0.4
GATE nor2      2      0=!(a+b);    PIN * INV 1 999 1.4 0.5 1.4 0.5
GATE nor3      3      0=!(a+b+c);  PIN * INV 1 999 2.4 0.7 2.4 0.7
GATE nor4      4      0=!(a+b+c+d); PIN * INV 1 999 3.8 1.0 3.8 1.0
GATE and2      3      0=a*b;        PIN * NONINV 1 999 1.9 0.3 1.9 0.3
GATE or2       3      0=a+b;        PIN * NONINV 1 999 2.4 0.3 2.4 0.3
GATE xor       5      0=a!*b+!a*b; PIN * UNKNOWN 2 999 1.9 0.5 1.9 0.5
GATE xor       5      0=!(a*b+!a*b); PIN * UNKNOWN 2 999 1.9 0.5 1.9 0.5
GATE xnor      5      0=a*b+!a*b; PIN * UNKNOWN 2 999 2.1 0.5 2.1 0.5
GATE xnor      5      0=!(!a*b+a*b); PIN * UNKNOWN 2 999 2.1 0.5 2.1 0.5
GATE aoi21     3      0=!(a*b+c);  PIN * INV 1 999 1.6 0.4 1.6 0.4
GATE aoi22     4      0=!(a*b+c*d); PIN * INV 1 999 2.0 0.4 2.0 0.4
GATE oai21     3      0=!((a+b)*c); PIN * INV 1 999 1.6 0.4 1.6 0.4
GATE oai22     4      0=!((a+b)*(c+d)); PIN * INV 1 999 2.0 0.4 2.0 0.4
GATE zero      0      0=CONST0;
GATE one       0      0=CONST1;
    
```

Figure 1.4: mcnc.genlib

Si sono confrontati i risultati di 2 differenti modalità di mapping per l'uscita Y1, uno che favorisce una minore area utilizzata (map -W -m **0** -s), uno il ritardo massimo (map -W -m **1** -s). Inoltre questa analisi è stata effettuata prima e dopo la minimizzazione della rete combinatoria.

Si osservi come il mapping prima e dopo la minimizzazione dia il risultato atteso, infatti dal momento che la minimizzazione porta a un costo minore in termini di letterali ci si aspettava un'occupazione di area minore.

Y1	Area (min A)	tempo (min A)	Area (min T)	tempo (min T)
Mapping prima della minimizzazione	66	17.00	89	13.28
Mapping dopo la minimizzazione	61	18.30	83	12.60

Figure 1.5: mcnc.genlib

1.3 Esercizio 03

1.3.1 Traccia

Si calcoli la forma minima della macchina M come rete multi-uscita utilizzando lo strumento SIS e si disegni il grafo corrispondente.

1.3.2 Esercizio

Le 3 uscite della macchina M, che in precedenza erano state definite separatamente in file .blif indipendenti, sono state adesso unite sotto un unico file che rappresenti una macchina multi-uscita. Inoltre si è deciso di utilizzare lo script rugged su SIS per la minimizzazione della rete, e, già dopo la prima esecuzione dello script si è pervenuti alla soluzione “ottima” riportata in fig. 1.6.

```

INORDER = a b c d e f;
OUTORDER = y0 y1 y2;
y0 = [5]*[1][6]*![11] + [6]*[11] + ! [5]*[6];
y1 = [9]*[12]*[13] + !y2*[5]*[8] + b!*y2*[8] + !y2*[9] + !y2*[7];
y2 = b*[8]*[11] + [12]*[13] + [7]*[9];
[3] = a!*b + !a*b;
[4] = !c*[3] + c*[3];
[5] = f + e;
[6] = !d*[4] + d*[4];
[7] = [13] + [12];
[8] = d + c;
[9] = b*[5] + [11];
[11] = e*f;
[12] = a*y0;
[13] = c*d;
    
```

Figure 1.6: script.rugged macchina M

In particolare la rete risultante è costituita da 13 nodi e 54 sop, ed è possibile vederne la struttura tramite il grafo in fig. 1.7.

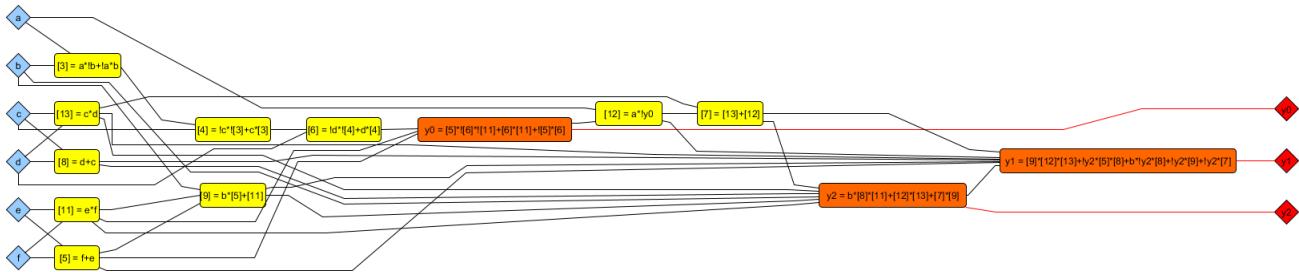


Figure 1.7: Grafo macchina M

1.4 Esercizio 04

1.4.1 Traccia

Si implementi la macchina M, nella forma ottenuta al punto 3, in VHDL seguendo una modalità di descrizione di tipo “data-flow”.

1.4.2 Esercizio

La macchina M minimizzata nell’ esercizio 3 è stata implementata in VHDL con modalità di descrizione “data-flow”. Ciò implica che si sono riportati i vari nodi di calcolo trovati nell’esercizio 3 nel corpo dell’entità rappresentante la macchina M, visibile in list. 1.1. Essendo una macchina combinatoria relativamente semplice e di pochi ingressi si è deciso di verificare il suo comportamento simulando in modo esaustivo tutte le possibili combinazioni delle variabili di ingresso. (list. 1.2). Si riporta inoltre la simulazione in fig. 2.21.



Figure 1.8: Simulazione M

```

1 library ieee;
2 use ieee.std_logic_1164.all;
3
4 entity Macchina_M is
5 port(
6     a,b,c,d,e,f : IN std_ulogic;
7     y0,y2 : INOUT std_ulogic;
8     y1: OUT std_ulogic);
9 end Macchina_M;
10
11 architecture M_DATAFLOW of Macchina_M is
12     signal s3,s4,s5,s6,s7,s8,s9,s11,s12,s13 : std_ulogic;
13 begin
14     s3<= (a and (not b)) or ((not a) and b);
15     s13<=c and d ;
16     s8<=d or c ;
17     s11<=e and f;
18     s5 <= f or e;
19     s4<= ((not c) and (not s3)) or (c and s3) ;
20     s9<=(b and s5) or s11;
21     s6<=((not d) and (not s4)) or (d and s4);
22     y0<=(s5 and (not s6) and (not s11)) or (s6 and s11) or ((not s5) and
23         s6) ;
24     s12<= a and (not y0);
25     s7<=s13 or s12 ;
26     y2<= (b and s8 and s11) or (s12 and s13) or (s7 and s9);
27     y1<=(s9 and s12 and s13) or ((not y2) and s5 and s8) or (b and (not
         y2) and s8) or ((not y2) and s9) or ((not y2) and s7) ;
end architecture;

```

Listing 1.1: VHDL Macchina M

```

1 library ieee; use ieee.std_logic_1164.all;
2 entity contatore_tb is end contatore_tb;
3 architecture BEHAVIORAL of contatore_tb is
4     component Macchina_M is
5         port(
6             a,b,c,d,e,f : IN std_ulogic;

```

```

7      y0 ,y2 : INOUT std _ulogic ;
8      y1: OUT std _ulogic );
9  end component;
10 signal i0 ,i1 ,i2 ,i3 ,i4 ,i5: std _ulogic := '0 ';
11 signal u0 ,u1 ,u2 : std _ulogic := '0 ';
12 begin
13   macchina : Macchina_M  port map ( i0 , i1 , i2 , i3 , i4 , i5 , u0 , u2 ,
14                                         u1 ) ;
15
16 process begin
17   for i in 0 to 63 loop
18     if (( i rem 2)=0)then
19       i0 <= '0' ;
20     else i0 <= '1' ;
21   end if;
22   wait for 5 NS;
23 end loop;
24 wait ;
25 end process;
26
27 process begin
28   for i in 0 to 31 loop
29     if (( i rem 2)=0)then
30       i1 <= '0' ;
31     else i1 <= '1' ;
32   end if;
33   wait for 10 NS;
34 end loop;
35 wait ;
36 end process;
37
38 process begin
39   for i in 0 to 15 loop
40     if (( i rem 2)=0)then
41       i2 <= '0' ;
42     else i2 <= '1' ;
43   end if;
44   wait for 20 NS;
45 end loop;
46 wait ;
47 end process;
48
49 process begin
50   for i in 0 to 7 loop
51     if (( i rem 2)=0)then

```

```

52      i3 <= '0' ;
53      else i3 <= '1' ;
54  end if ;
55  wait for 40 NS;
56 end loop ;
57 wait ;
58 end process ;

59
60 process begin
61 for i in 0 to 3 loop
62   if ((i rem 2)=0)then
63     i4 <= '0' ;
64     else i4 <= '1' ;
65   end if ;
66   wait for 80 NS;
67 end loop ;
68 wait ;
69 end process ;

70
71 process begin
72 for i in 0 to 1 loop
73   if ((i rem 2)=0)then
74     i5 <= '0' ;
75     else i5 <= '1' ;
76   end if ;
77   wait for 160 NS;
78 end loop ;
79 wait ;
80 end process ;
81
82 end architecture ;

```

Listing 1.2: TestBench contatore

1.5 Esercizio 05

1.5.1 Traccia

Si progetti la macchina M per composizione di macchine a partire da blocchi full-adder, e si implementi la soluzione trovata in VHDL

1.5.2 Contatore di 6 bit implementato tramite Contatori di 3bit (Full Adder)

Nella fig.1.9 è riportata la soluzione trovata per implementare il contatore a sei bit tramite contatori di 3 bit. L'idea è stata presa dall'algoritmo di Wallace.

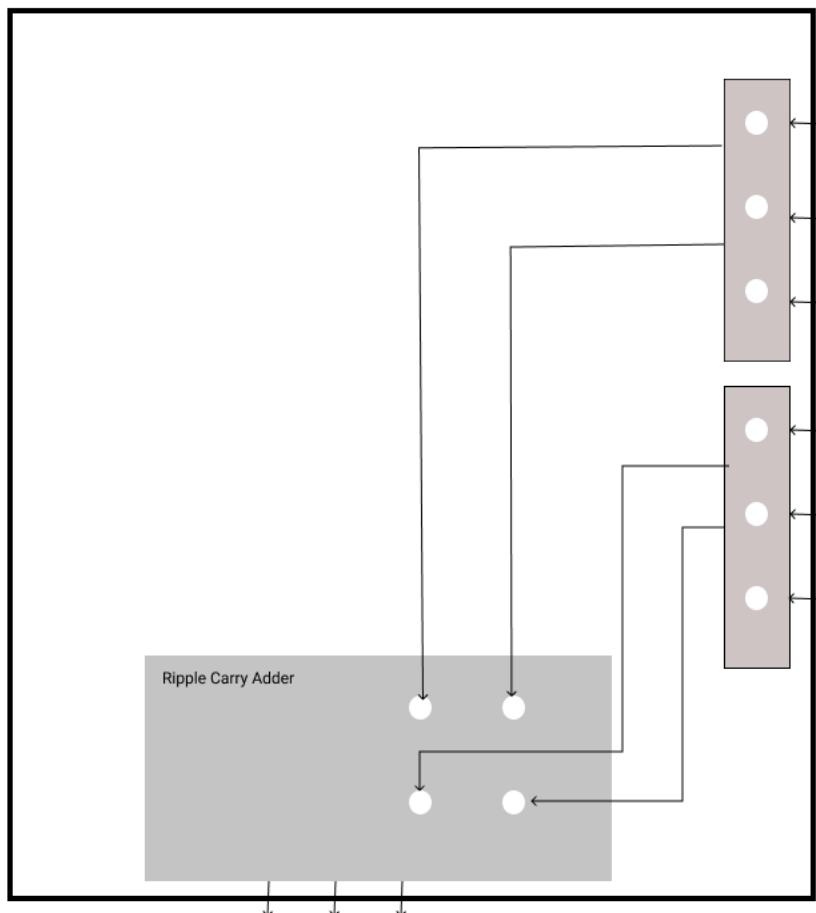


Figura 1.9: Contatore di 6 bit strutturato come contatori di 3 bit

Nella fig.1.10 si sono voluti mettere in evidenza i componenti Full Adder.

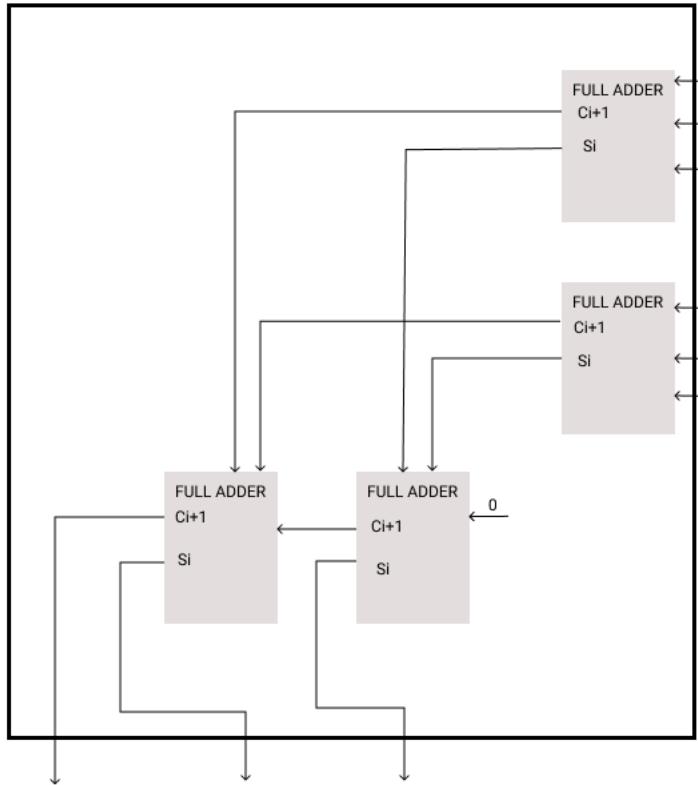


Figura 1.10: Contatore di 6 bit strutturato come quattro Full Adder

1.5.3 Implementazione in VHDL

L'entity Full_Adder è stata definita tramite un'architecture di tipo dataflow.

```

1 library ieee; use ieee.std_logic_1164.all;
2 entity FULL_ADDER is
3   port(x,y,c:IN std_ulogic; si, cp:OUT std_ulogic);
4 end FULL_ADDER;
5 architecture FULL_ADDER_DATAFLOW of FULL_ADDER is
6 begin
7   si<=x XOR y XOR c;
8   cp<=(x AND y) OR (x AND c) OR (y AND c);
9 end architecture;
```

L'entity contatore è stata definita tramite un'architecture di tipo strutturale come composizione di 4 Full_Adder. Infine è stato dichiarato un package “aritmetic” che verrà utilizzato nell'esercizio successivo.

```

2 library ieee; use ieee.std_logic_1164.all;
3 entity contatore is
4     port (i0,i1,i2,i3,i4,i5:IN std_ulogic;u0,u1,u2 :OUT std_ulogic);
5 end contatore;
6 architecture structural of contatore is
7     component FULL_ADDER is port(x,y,c:IN std_ulogic; si,cp:OUT
8         std_ulogic); end component;
9     signal temp: std_ulogic;
10    signal temp1: std_ulogic;
11    signal temp2: std_ulogic;
12    signal temp3: std_ulogic;
13    signal temp4: std_ulogic := '0';
14    signal temp5: std_logic;
15 begin
16 FA1: FULL_ADDER port map(i2 ,i1 , i0 , temp1 , temp);
17 FA2: FULL_ADDER port map(i5 ,i4 , i3 , temp3 , temp2);
18 FA3: FULL_ADDER port map(temp1,temp3 , temp4 , u0 , temp5);
19 FA4: FULL_ADDER port map(temp2,temp , temp5 , u1 , u2);
20 end architecture;
21
22 library ieee;
23 use ieee.std_logic_1164.all;
24
25 package aritmetic is
26     component contatore is
27         port (i0,i1,i2,i3,i4,i5:IN std_ulogic;u0,u1,u2 :OUT std_ulogic)
28             ;
29     end component;
30     component FULL_ADDER is
31         port(x,y,c:IN std_ulogic; si,cp:OUT std_ulogic);
32     end component;
33 end aritmetic;

```

1.6 Esercizio 06

1.6.1 Traccia

Si progetti una macchina S che, date 6 stringhe di 3 bit ciascuna in ingresso (A, B, C, D, E, F), rappresentanti la codifica binaria di numeri interi positivi, ne calcoli la somma W espressa su 6 bit. La macchina S deve essere progettata per composizione di macchine utilizzando la macchina M progettata al punto 5) e componenti full-adder, opportunamente collegati.

1.6.2 Modello strutturale di Sommatore di 6 stringhe da tre bit

Nella fig.1.11 è riportata la soluzione trovata per implementare il Sommatore di 6 stringhe da tre bit. L'idea è stata presa dall'algoritmo sempre dall'algoritmo "somma per colonne" che viene applicato nella seconda fase dell'implementazione di un moltiplicatore parallelo. Nel primo livello si utilizzano prima tre contatori di sei bit per contare quanti '1' vi sono in ogni colonna. Nel secondo livello il problema si riduce una matrice di tre righe, e quindi si utilizzano dei contatori a tre bit (i Full Adder).

Nel terzo livello si hanno solo due righe e quindi si utilizza un sommatore ripple carry per sommare due stringhe da tre bit.

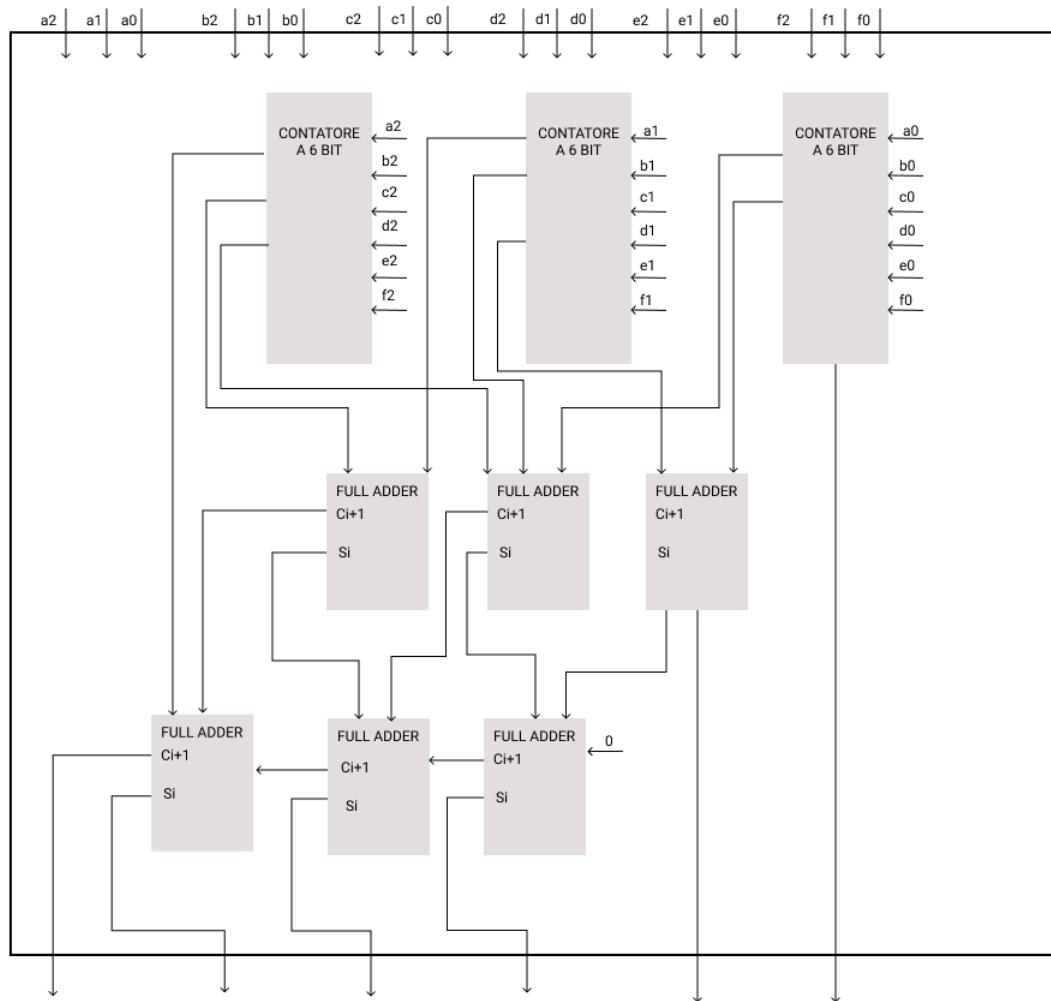


Figura 1.11: Contatore di 6 bit strutturato come quattro Full Adder

1.6.3 Implementazione in VHDL

L'entity ADD6 è stata definita tramite un'architecture di tipo strutturale facendo riferimento a entity definite nell'esercizio precedente (si è scelto di definire un package per referenziare entity situate in cartelle diverse).

```

1 —ghdl -s --workdir=/home/giorgio/Documenti/esercizighdl/
2   Contatoreaseibit
3
4 library ieee;
5 use ieee.std_logic_1164.all;
6 library work;
7 use work.aritmetic.all;
8
9 entity ADD6 is
10   port (a0,a1,a2,b0,b1,b2,c0, c1,c2,d0, d1,d2,e0,e1,e2,f0, f1,f2:IN
11         std_ulogic; o5,o4,o3,o2,o1,o0:OUT std_ulogic);
12 end ADD6;
13
14 architecture Structural of ADD6 is
15   signal temp, temp1, temp2,temp11,temp22, temp3,temp222,temp33, temp4,
16   tempsec2,tempsec22,tempsec3,tempsec33,tempsec4,c4,c5: std_ulogic;
17
18 component contatore is
19   port (i0,i1,i2,i3,i4,i5:IN std_ulogic; u0,u1,u2 :OUT std_ulogic);
20 end component;
21
22 component FULL_ADDER is
23   port( x,y,c:IN std_ulogic; si,cp:OUT std_ulogic);
24 end component;
25
26 begin
27   cont1: work.aritmetic.contatore port map (a0,b0,c0,d0,e0,f0,o0,temp1,
28                                             temp2);
29   cont2: work.aritmetic.contatore port map (a1,b1,c1,d1,e1,f1,temp11,
30                                             temp22, temp3);
31   cont3: work.aritmetic.contatore port map (a2,b2,c2,d2,e2,f2,temp222,
32                                             temp33, temp4);
33
34   AD1: work.aritmetic.FULL_ADDER port map (temp1,temp11,'0',o1,
35                                             tempsec2);
36   AD2: work.aritmetic.FULL_ADDER port map(temp2,temp22,temp222,
37                                             tempsec22,tempsec3);
38   AD3: work.aritmetic.FULL_ADDER port map(temp3,temp33,'0',tempsec33,
39                                             tempsec4);
40   AD4: work.aritmetic.FULL_ADDER port map(tempsec2,tempsec22,'0',o2,c4
41 );
42   AD5: work.aritmetic.FULL_ADDER port map(tempsec3,tempsec33,c4,o3,c5)
43 ;
44   AD6: work.aritmetic.FULL_ADDER port map (temp4,tempsec4,c5,o4,o5);

```

34

35 end architecture ;

Per il test bench si è definita un'architecture comportamentare simile a quella mostrata precedentemente per provare che la macchina definita rispondesse in modo corretto ad ogni sollecitazione in ingresso.



Capitolo 2

Latch/Flip Flop

2.1 Latch RS

2.1.1 Traccia

Si eseguono per il latch RS una simulazione comportamentale e post-sintesi, illustrando i passaggi salienti.

2.1.2 Soluzione

Il latch RS ha due ingressi detti di Set (S) e di reset (R) e funge da dispositivo a memorizzazione dello stato.

In particolare si farà riferimento a una logica **1-attiva**. Ovvero l'ingresso $SR='10'$ porrà lo stato di Q al valore attivo, ovvero 1 (e Q' al valore non attivo), invece l' ingresso $SR = '01'$ porrà lo stato Q al valore non attivo, in questo caso '0'.

L'ingresso neutro $RS='00'$ lascerà lo stato del latch invariato.

R ed S non devono assumere contemporaneamente il valore attivo (**vincolo $R*S=0$**).

Il latch RS trattato di seguito non sarà quello dinamico che verrà presentato successivamente.

Il latch RS trattato invece sfrutterà il fatto che tra le due porte NOR del latch dinamico è presente già l'uscita Q' . Porre un pin di uscita tra i due NOR significa aggiungere una linea di reazione e quindi una variabile di stato.

Lo stato memorizzato affinchè sia uguale a quello presentato in uscita, ciò impone che i due stati memorizzati siano codificati come '10' e '01'. Quindi per evitare corse critiche tra variabili di stato è di estrema importanza aggiungere uno stato instabile ('00').

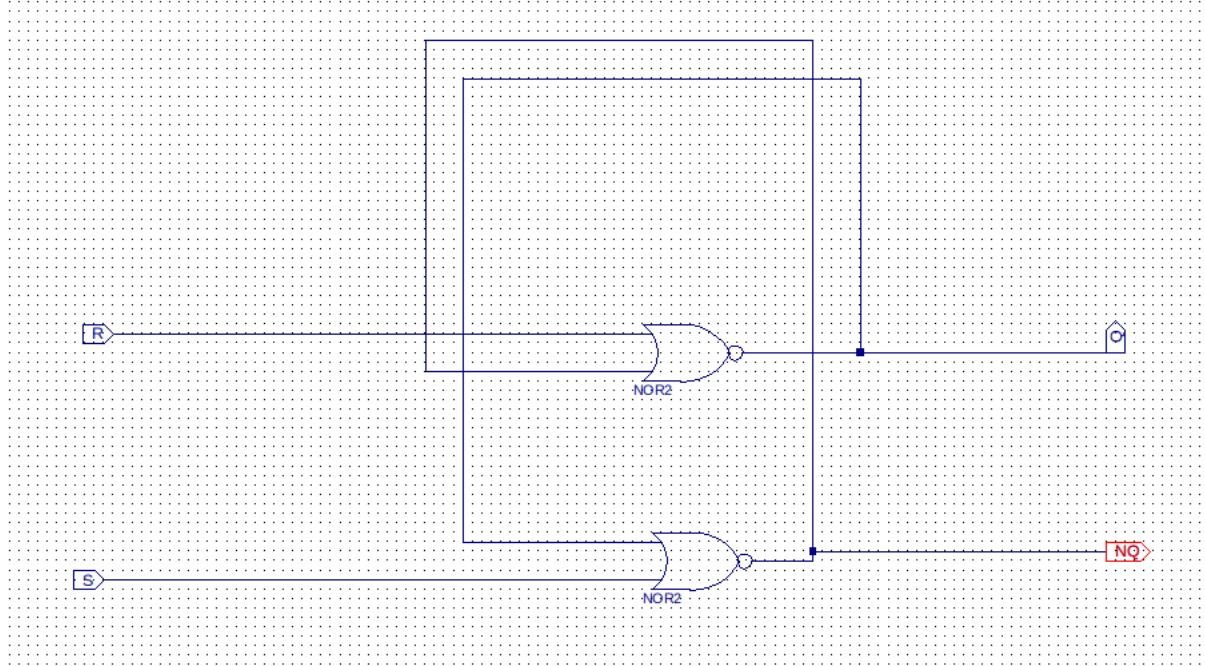


Figure 2.1: Schematic Latch RS 1-attivo realizzato con porte NOR

2.1.2.1 Simulazione Behavioral

La simulazione Behavioral sarà utilizzata per verificare che il comportamento del dispositivo, così progettato, risponda adeguatamente alle sollecitazioni in ingresso e che quindi rispetti la specifica formale.

Osservando lo schematic in Fig.2.1 , se poniamo in ingresso il valore $RS='11'$ le porte NOR forzano le due variabili di uscita al valore $QQ='00'$ (che è la codifica dello stato instabile aggiunto per evitare le corse critiche tra variabili di stato).

Ponendo in ingresso successivamente $RS='00'$, Q e Q' saranno pari a '11'. Lasciando l'ingresso costante ($RS='00'$), le due porte NOR fungeranno da inverter, e in uscita si creerà un ciclo che alternerà il valore delle variabili di stato (o di uscita) tra '00' e '11'.

Questo comportamento è stato definito dai progettisti della macchina nella fase di sintesi. Infatti si è riportata per chiarezza la Tab. definita in fase di sintesi della rete sequenziale dopo la fase di progettazione della rete combinatoria.

Qp/SR	00	01	11	10	$QQ'(y_1y_2)$
S0	S0	S0	S	S	01
S	11	S0	S	S1	00
S1	S1	S	S	S1	10
11	S	S	S	S	11

Table 2.1: Tabella delle transazioni di stato dopo la fase di progettazione della rete combinatoria della rete sequenziale

Poichè in questo tipo di simulazione i ritardi di porta e di collegamento sono nulli, giustamente la simulazione rappresenterà questo ciclo con un periodo pari a zero o se vogliamo con una frequenza infinita.

Questo comportamento oscillatorio è completamente volontario e risponde alla sintesi effettuata dall'utente.

Il VHDL in realtà per gestire la concorrenza utilizza un suo ritardo interno, il delta delay. Quindi, in modo trasparente all'utente, la sequenza in ingresso '1100' genera un ciclo con un periodo pari al delta delay.

Si riproduce questo ciclo fino a un numero limite di volte, che viene presentato come errore all'utente.

Quindi possiamo concludere tramite la simulazione Behavioral che il comportamento della macchina è conforme con quello scelto in fase di progettazione.



Figure 2.2: Simulazione Behavioral

2.1.2.2 Simulazione Post Route

Questo tipo di simulazione introduce i ritardi delle porte e dei collegamenti ed è quella che rispecchia di più un comportamento reale della macchina, introducendo delle asimmetrie, e quindi ad esempio dei ritardi diversi nei due collegamenti di reazione.

Ciò comporta un comportamento diverso della macchina rispetto a quello riscontrato nella simulazione Behavioral, infatti, riproponendo la stessa sequenza di ingresso, quando si passa dall'ingresso RS='11' a RS='00' non si genera un'oscillazione infinita, ma si stabilizza in uno dei due stati '01' o '10'.

Questo perchè con RS='11' si memorizza lo stato '00', al quale se si ripropone in ingresso RS='00', si produce come stato successivo (o come uscita) QQ' ='11'. Queste due variabili di stato entreranno in corsa tra loro, e si presenteranno quasi sempre come variabili di stato corrente in due istanti di tempo differenti.

Ciò significa che nel passare dallo stato '00' a uno stato '11', non essendo due codifiche adiacenti, si genererà una corsa critica tra le due variabili di stato, e, a seconda che arrivi prima una o l'altra, lo stato della macchina si stabilizzerà in '01' o in '10'.

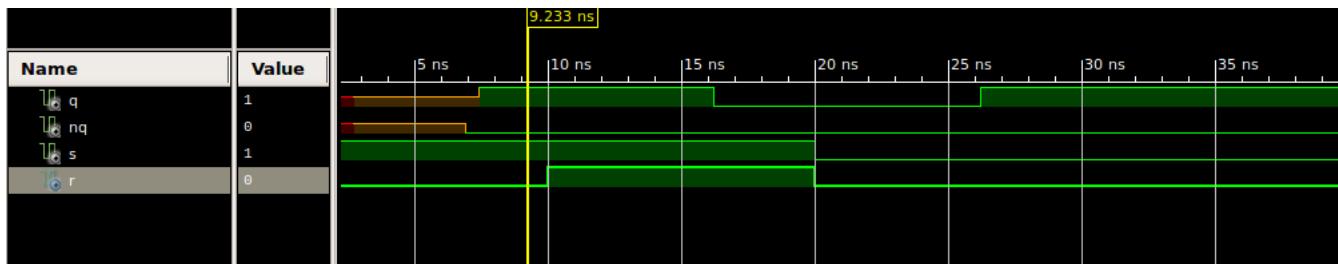


Figure 2.3: Simulazione Post Route

2.1.2.3 Condizioni di Modo fondamentale e Funzionamento

Per quanto riguarda le sequenze di ingressi, essendo il latch RS una macchina asincrona, quest'ultima deve rispettare le condizioni di Modo fondamentale.

La prima condizione è che non vi deve essere corsa tra le variabili di ingresso, quindi si imporrà in questo caso di utilizzare la macchina alternando l'ingresso ('01' o '10') con l'ingresso '00' in questo modo si evita anche un'involontaria sequenza di ingresso proibita.

A questo punto se il comportamento fosse stato come nella simulazione Behavioral si sarebbe potuto presentare il comportamento del dispositivo con l'ingresso '11' come un oscillatore, ma invece non è così, in una simulazione più vicina alla realtà si è riscontrato che il comportamento del latch non è determinabile.

La seconda condizione del modo fondamentale riguarda la durata degli ingressi per garantire che la macchina si assesti in uno stato totale stabile e quindi il corretto funzionamento della macchina.

Per risolvere questo problema si è considerata un'implementazione in VHDL dove si sono introdotti i ritardi inerziali e di trasporto delle porte NOR e il ritardo di trasporto delle due linee di reazione (si è considerato uguale il ritardo delle due linee di reazione sempre perché supponendo che la sequenza in ingresso sia alternata con '00' non si passerà mai per l'ingresso '11').

```

1 library ieee; use ieee.std_logic_1164.all;
2 entity Latch_rs is
3 generic (DELAYQT: time := 1.0 ns; DELAYNQT: time := 1.0 ns;
4          DELAY_INERTIAL_NOR: time := 3 ns; DELAY_TRANSPORT_NOR: time := 5
5          ns);
6 port( R,S,Clear,preset: IN std_ulogic ; Q,NQ: inout std_ulogic );
7 end Latch_rs;
8 architecture dataflow of Latch_rs is
9 signal DeltaQ: std_ulogic; signal DeltaNQ: std_ulogic;
10 begin
11   DeltaQ <= TRANSPORT Q after DELAYQT ;
12   DeltaNQ <= TRANSPORT NQ after DELAYNQT ;
13   Q <= REJECT DELAY_INERTIAL_NOR INERTIAL (NOT(R OR DeltaNQ OR Clear))
14     after DELAY_TRANSPORT_NOR ;
15   NQ <= REJECT DELAY_INERTIAL_NOR INERTIAL (NOT(S OR DeltaQ OR preset))
16     after DELAY_TRANSPORT_NOR ;
17 end architecture;

```

Adesso per intuire quando si ha il corretto funzionamento della macchina si rappresentera un grafo degli stati (senza l'ingresso '11' in quanto stiamo ipotizzando delle condizioni sulle sequenze di ingresso):

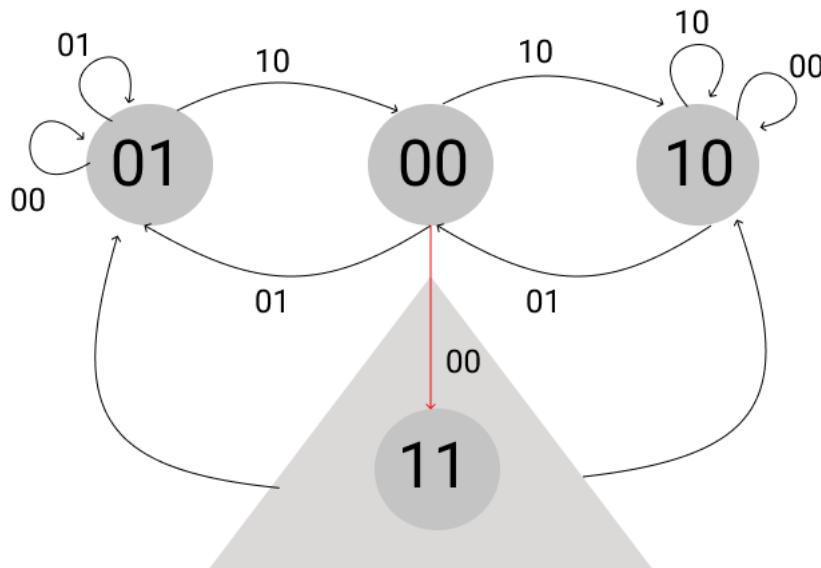


Figura 2.4: Grafo degli stati supponendo in ingresso una sequenza alternata con l'ingresso '00' (quindi non è presente l'ingresso '11')

```

1 LIBRARY ieee;
2 USE ieee.std_logic_1164.ALL;
3 ENTITY Rs_tb IS
4 END Rs_tb;
5
6 ARCHITECTURE behavior OF Rs_tb IS
7   COMPONENT Latch_rs
8     generic (DELAYQT: time := 1.0 ns; DELAYNQT: time := 1.0 ns;
9               DELAY_INERTIAL_NOR: time := 3 ns; DELAY_TRANSPORT_NOR: time := 5
10              ns);
11     PORT(R : IN std_logic; S : IN std_logic; Clear : IN std_logic;
12           preset : IN std_logic; Q : INOUT std_logic; NQ : INOUT
13           std_logic);
14   END COMPONENT;
15
16 BEGIN      — Instantiate the Unit Under Test (UUT)
17   uut: Latch_rs PORT MAP ( R => R,S => S, Clear => Clear , preset =>
18                             preset , Q => Q,NQ => NQ);
19   — Stimulus
20   process      stim_proc: process      begin
21     preset <= '0';
  
```

```

17  Clear <= '1', '0' after 10 ns;
18  S <= '0', '1' after 30 ns, '0' after 39 ns, '0' after 60 ns;
19  R <= '0', '0' after 30 ns, '0' after 39 ns, '1' after 60 ns;
20      wait;
21  end process;
22 END;
```

2.1.2.4 Durata minima di permanenza dell'ingresso per passare da uno stato totale stabile all'altro stato totale stabile

Si definisca la durata minima di permanenza dell'ingresso RS='10' affinchè da QQ='01' si passi nello stato '10'.

Come si può osservare dal grafo di Fif. 2.4 ho bisogno di mantenere l'ingresso stabile affinchè si passi prima per lo stato instabile '00' per poi arrivare nello stato '10'. Ciò significa, osservando la Fig.2.1 che vi sarà la necessità di mantenere l'ingresso stabile affinchè l'uscita si definisca dopo due passaggi (cicli) per la rete combinatoria.

Dopo un tempo pari alla somma del tempo di trasporto della porta NOR e di quello delle linee di reazione (nel nostro caso 5 ns + 1ns), si ha un giro per la rete combinatoria e il nuovo stato instabile '00' si presenta in ingresso come stato presente (sempre alla rete combinatoria).

Adesso nel secondo giro, con stato corrente '00', l'ingresso deve essere mantenuto stabile ancora per un intervallo di tempo minimo che garantisca la commutazione dell'uscita, e quindi un tempo almeno pari al ritardo inerziale della porta NOR (nel nostro caso pari a 3 ns).

Come si è verificato tramite una simulazione di Fig. 2.5, se l'ingresso dura per una durata almeno pari a 9 ns la rete sequenziale ha un funzionamento corretto.

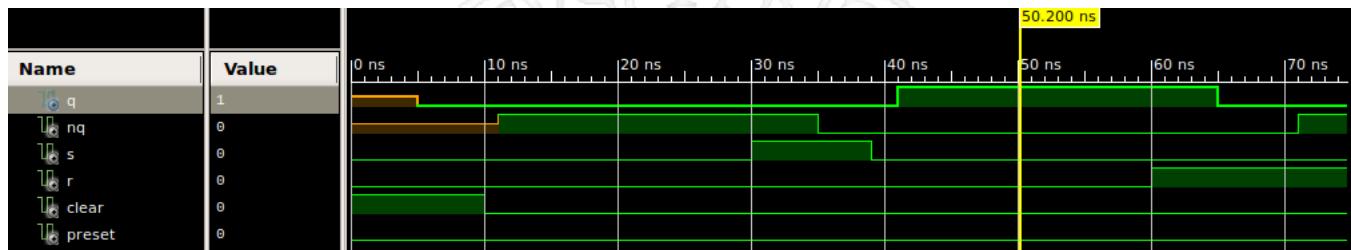


Figura 2.5: Simulazione con l'ingresso che dura per una durata maggiore di 9 ns

2.1.2.5 Conseguenze quando la durata minima di permanenza dell'ingresso non è rispettata

Quando la durata minima di permanenza dell'ingresso è minore di quella minima che garantisce un funzionamento corretto, la rete viene lasciata nello stato instabile '00'.

Poichè dopo uno dei due ingressi di set e di reset si ha l'ingresso RS ='00', ci si ritrova che lo stato corrente è '00' e l'ingresso è '00'.

Come discusso precedentemente, a causa di una corsa tra le due variabili di stato, la rete si stabilizzerà in modo non determinabile in uno dei due stati totali stabili.

Questa situazione è mostrata in Fig. 2.4 con il triangolo.

2.1.3 Traccia

Si eseguono per il latch RS dinamico una simulazione comportamentale e post-sintesi, illustrando i passaggi salienti.

2.1.4 Soluzione

Il latch dinamico è diretta conseguenza della tabella caratteristica. Si hanno due soli stati ed è quindi possibile codificarli con una variabile di stato (una linea di reazione). L'uscita per presentare Q e Q' utilizzerà successivamente un inverter sullo stato Q.

Si approfondirà sempre il funzionamento per una logica 1-attiva. Di seguito sono presentate due strutture che implementano la logica '1' attiva. Entrambe differiscono per la definizione scelta per i don't care nella fase di progettazione della rete combinatoria:

Qp\RS	00	01	11	10	QQ'
S0	S0	S1	-	S0	01
S1	S1	S1	-	S0	10

Tabella 2.2: Tabella delle transizioni di stato del latch RS fondamentale

In questo caso particolare non vi è corsa critica tra le variabili di stato visto che ne è una sola. Allora i due stati possono essere codificati banalmente S0 come '0' e S1 come '1'.

Se, stando nello stato '0' o '1', per l'ingresso '11' si definisce il prossimo stato (ovvero anche l'uscita Q) pari a '1' (ovvero S1) si ottiene una struttura a porte Nand come quella mostrata nello schematic di figura 2.6

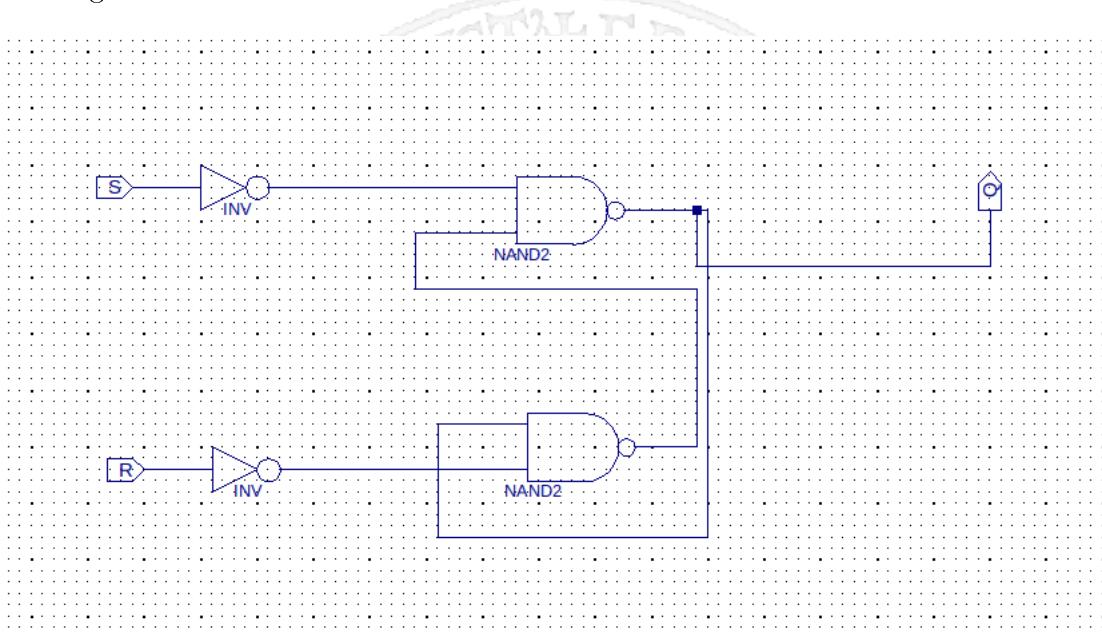


Figura 2.6: Latch RS dinamico 1-attivo a porte NAND

Se, stando nello stato '0' o '1', per l'ingresso '11' si definisce il prossimo stato (ovvero anche l'uscita Q) pari a '0' (ovvero S1) si ottiene una struttura a porte Nor come quella mostrata nello schematic di figura

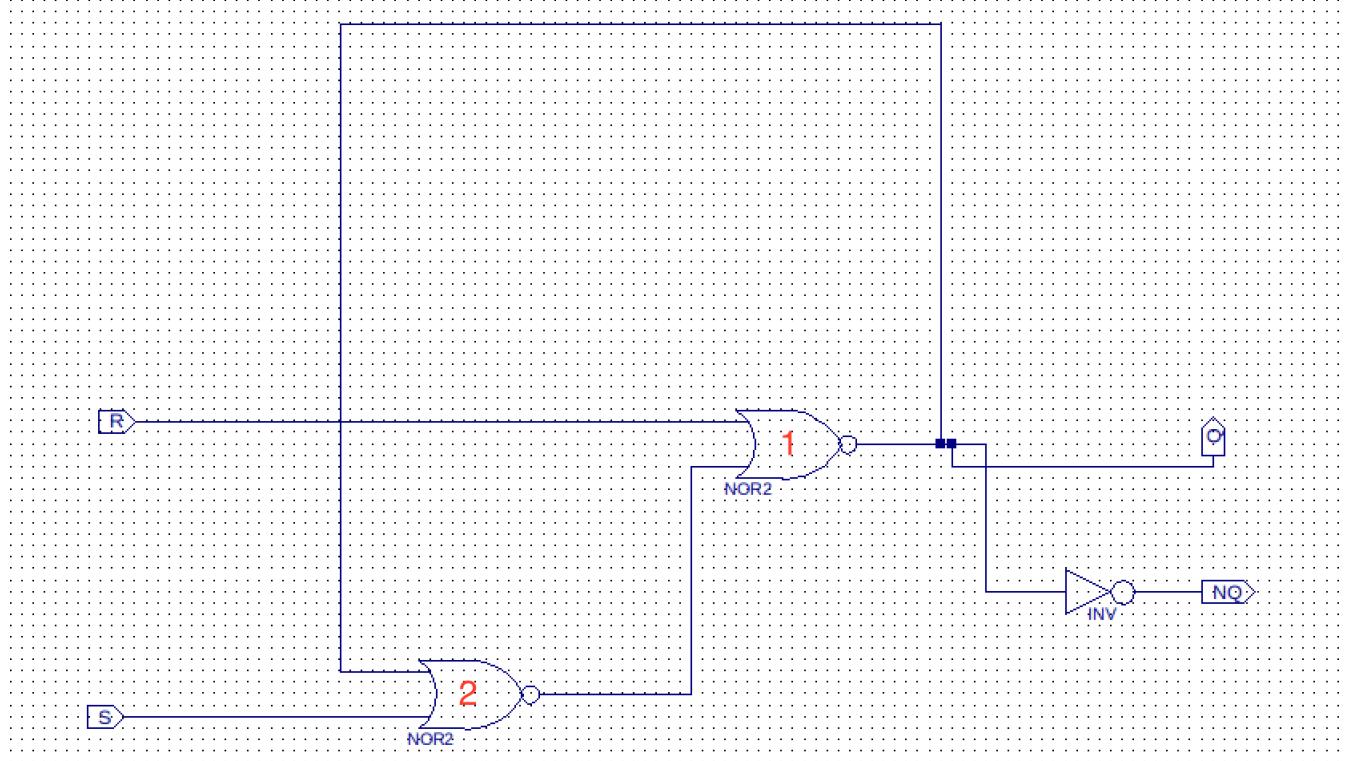


Figura 2.7: Latch RS dinamico 1-attivo a porte NOR

2.1.4.1 Simulazione Post Route

La simulazione Post Route per il latch RS a porte NOR di Fig. 2.7. Il comportamento è conforme alle scelte progettuali se, ponendo in ingresso la sequenza '11 00' con qualsiasi stato corrente, lo stato successivo è S0 ($Q=0'$).

Si ponga l'attenzione sulla rete combinatoria:

Si osservi prima di procedere che la NOR con un '1' in ingresso indipendentemente dall'altro ingresso risponde con '0'.

Posta una variazione di ingresso RS la risposta Q della macchina possiamo dividerla in due fasi.

Nella prima fase risponde alla variazione dell'ingresso Reset alla porta NOR 1.

Nella fase due risponde alla variazione dell'altro ingresso (ovvero l'uscita della porta NOR 2 avente come ingresso il Set).

Allora quando pongo come ingresso '11' ho il caso particolare per l'osservazione precedente che la fase 1 e la fase 2 siano uguali e pari a '0'.

Appena l'ingresso varia istantaneamente a '00' :

Nella fase 1 ho la risposta Q pari a '1'.

Nella fase 2 risponde con '0'.

Di conseguenza si genera un glich.

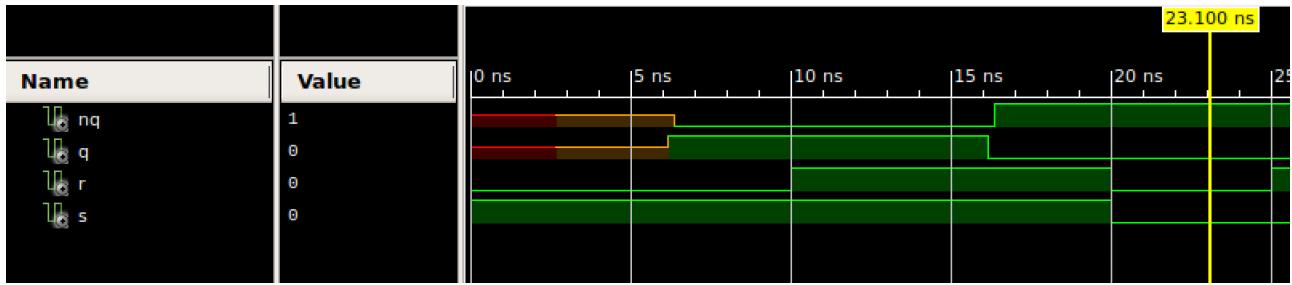


Figura 2.8: Post-Route Latch Dinamico RS a porte NOR

Nella maggior parte delle volte il glitch è così piccolo infatti da non creare problemi, e come si mostra in Fig.2.8 la Q acquisita è '0'.

2.2 Latch T

2.2.1 Traccia

Sviluppare il circuito del Latch di tipo T. Eseguire una simulazione illustrando i passaggi salienti.

2.2.2 Esercizio

Il Latch di tipo T è un dispositivo di memorizzazione di un bit che prevede un ingresso e due uscite. L'ingresso è dato da T che, quando viene posto al livello alto permette la commutazione del circuito. Le uscite sono il dato memorizzato Q e NotQ, il suo negato, che coincidono con le uscite del latch RS posto a valle della rete combinatoria che implementa la logica di funzionamento.

Con riferimento alla fig. 2.9 infatti è possibile notare che il dispositivo è sintetizzato a partire proprio da un latch RS utilizzato come elemento di memorizzazione. Inoltre sono illustrati i ritardi sulle connessioni ed i ritardi inerziali e di trasporto per le porte utilizzate durante la simulazione, in particolare:

- Ritardo inerziale: 3 ns per le porte NOR, 1 ns per le porte AND.
- Ritardo di trasporto: 5 ns per le porte NOR, 3 ns per le porte AND.

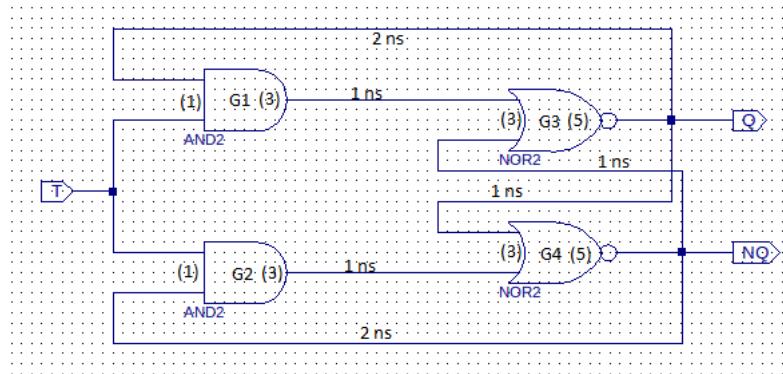


Figura 2.9: Schematico Latch T

Per la simulazione ci si è focalizzati in un primo momento sul mostrare il corretto funzionamento del latch, ponendo l'ingresso T alto per un periodo temporale compreso delta necessario ad ottenere un'unica commutazione del circuito e successivamente sui casi limite. Tale simulazione è visibile in fig. 2.10 e a tal proposito si è deciso di tenere il livello di T alto per un periodo di 11 ns. Sappiamo infatti, dall'esercizio precedente sul Latch RS con stessi ritardi di porta, che il tempo necessario a far stabilizzare l'elemento di memorizzazione è di 9 ns.

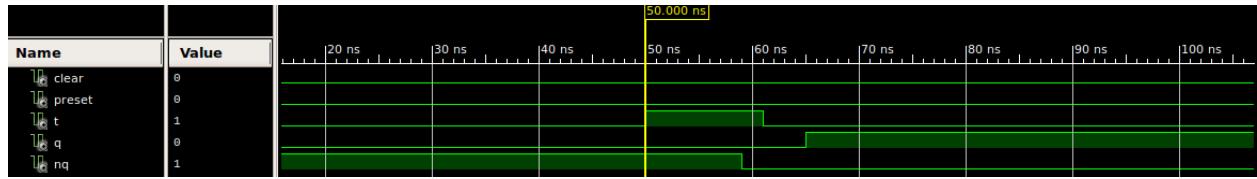


Figura 2.10: Simulazione corretta

Il caso limite di funzionamento sarà quindi proprio un ingresso T di durata 9 ns.

In fig. 2.11 è possibile infatti vedere due simulazioni svolte sui casi limite inferiori, in alto con un ingresso T di durata 8 ns ed in basso di soli 3 ns. Si può constatare che un ingresso di durata inferiore al tempo minimo di 9 ns non consente all'uscita di commutare. In particolare per ingressi di durata compresa tra i 4 ns e 8 ns si presentano delle particolari oscillazioni dovute alla simmetria del circuito.

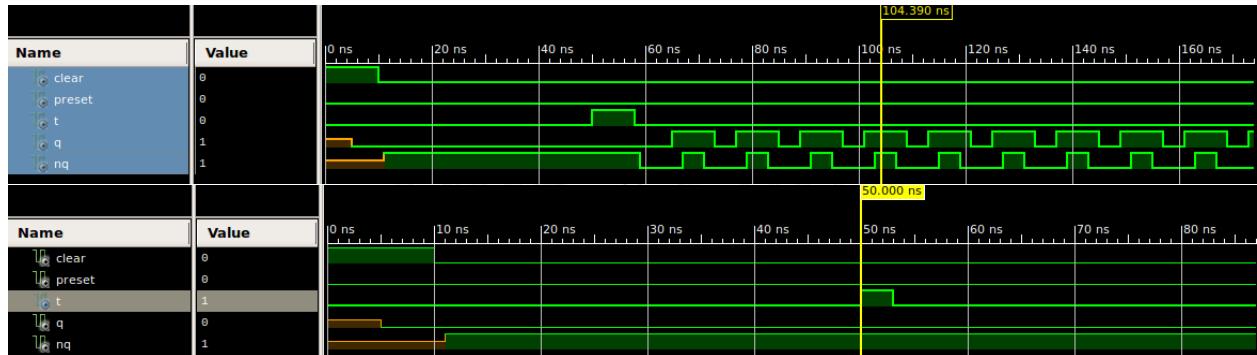


Figura 2.11: Simulazioni limiti inferiori

Simili considerazioni possono essere effettuate per la simulazione nei casi limite superiori.

Ponendo in ingresso un livello alto di durata eccessiva potrebbero verificarsi più commutazioni delle uscite.

2.2.2.1 Più commutazioni delle uscite

Supponendo di partire da una condizione stabile in cui è memorizzato il livello logico '0', apprezziamo la commutazione a '1' di Q dopo un tempo di 15 ns; successivamente questo nuovo valore assunto da Q si presenterà in ingresso alla AND con un ritardo ulteriore di 2 ns dovuto alla linea di reazione (per un totale di 17 ns).

Se questo impulso dovesse durare più del ritardo inerziale della NOR del latch RS(di calcolo della Q) si osserverà un'ulteriore commutazione della singola uscita Q.

Si sono distinti due casi che verranno illustrati di seguito.

2.2.2.2 Caso in cui vi sono oscillazioni

T continua ad essere alto, in più dei 17 ns, per ulteriori 3-6 ns. Così facendo si abbassa T prima che il latch RS commuti nello stato stabile lasciando il latch RS nello stato instabile '00'.

Non solo ma con l'abbassare di T in ingresso al latch RS vi sarà il valore '00'. Per la simmetria del circuito (le linee di reazione dell'RS hanno lo stesso ritardo di propagazione) si genereranno delle oscillazioni.

Ci si trova in queste condizioni, quindi, se T, ad esempio, dovesse durare un tempo compreso nell'intervallo 20-26 ns.

Si osservi a tal proposito la fig. 2.12: si pone un ingresso di durata 21 ns.

2.2.2.3 Caso in cui si ha una seconda commutazione

Viceversa se T dovesse durare un tempo maggiore di 9 ns oltre i 17 ns e quindi un totale di almeno 26 ns, osserviamo, per quanto detto precedentemente, una seconda commutazione (immagine in basso con T di durata 27 ns).

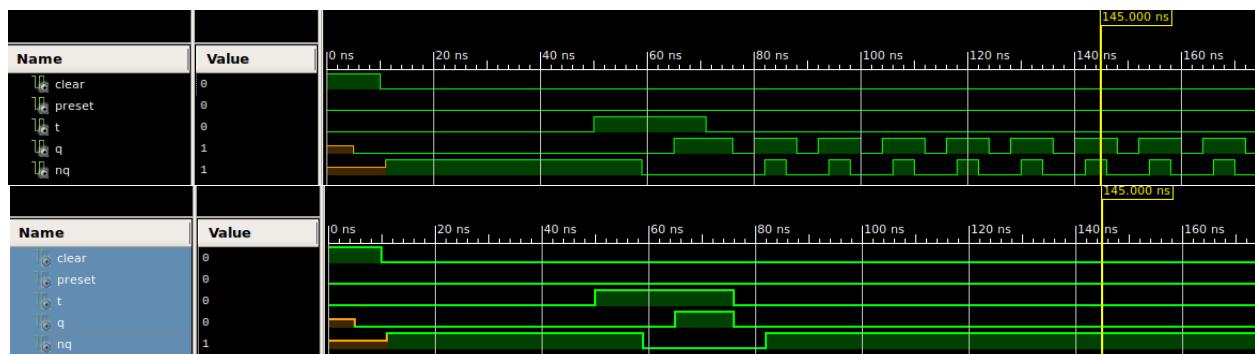


Figura 2.12: Simulazioni limiti superiori

2.2.2.4 Conclusione

Precedentemente si sono effettuate delle simulazioni dove il circuito è perfettamente simmetrico. In un caso reale si può supporre che l'impulso debba durare assolutamente dai 9 ns ai 17 ns. Nel "caso in cui vi sono oscillazioni" non si genererebbero oscillazioni, bensì si generebbe una corsa tra le due variabili di stato, a causa di una asimmetria delle linee di reazione del latch RS, assestando il circuito in uno dei due stati stabili.

2.3 Flip Flop D

2.3.1 Traccia

Sviluppare il circuito del FlipFlop di tipo D. Eseguire una simulazione comportamentale e post-sintesi, illustrando i passaggi salienti.

2.3.2 Esercizio

Il flip flop di tipo D è un dispositivo di memorizzazione di un bit che prevede due ingressi e due uscite. Gli ingressi sono il dato in ingresso D ed il clock, che, nel caso in esame, ne sincronizza la memorizzazione con il fronte di discesa $1 \rightarrow 0$. Le uscite sono il dato memorizzato Q e NotQ, il suo negato.

Come accennato si è deciso di implementare il dispositivo attivo sul fronte di discesa del clock, ciò implica che le porte utilizzate saranno delle NOR, porta 0-attiva ovvero che presenta uscita alta solo in presenza di ingressi bassi. Viceversa se si vuole implementare un flip flop D sensibile al fronte di salita sarà necessario utilizzare dualmente delle porte NAND. Lo schematico del circuito quindi sarà quello riportato in fig. 2.13.

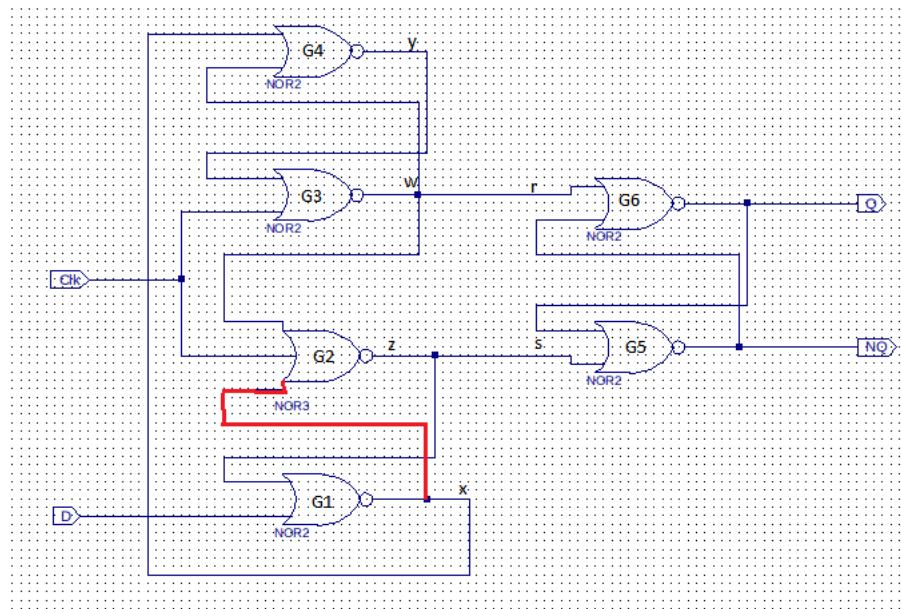


Figura 2.13: Schematico FlipFlop D

In particolare si è deciso di sviluppare, in moduli VHDL, le porte necessarie a realizzare lo structural del FlipFlop secondo due distinte implementazioni, quella in fig. 2.13 che prevede la connessione evidenziata in rosso, e una seconda che non presenta tale linea di connessione e come vedremo presenterà un'alea statica sull'uscita z della NOR g2.

I moduli implementati sono quelli della NOR a 2 ingressi e quella a 3 ingressi, utilizzate nella descrizione structural del flip flop nel listato 2.1.

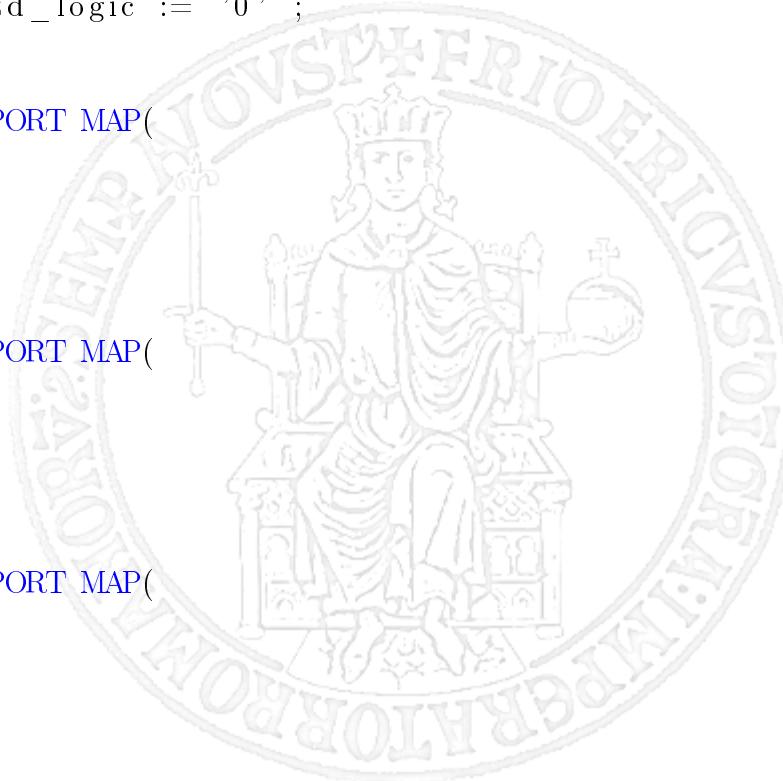
```

1 library IEEE;
2 use IEEE.STD_LOGIC_1164.ALL;
3
4 entity FFD_edg is
5     Port ( D : in STD_LOGIC;
6             Clk : in STD_LOGIC;
7             Q : out STD_LOGIC;
8             QN : out STD_LOGIC);
9 end FFD_edg;
```

```

10
11 architecture Structural of FFD_edg is
12
13 COMPONENT nor_2_in
14 PORT(
15     a : IN std_logic;
16     b : IN std_logic;
17     o : OUT std_logic
18 ); END COMPONENT;
19
20 COMPONENT nor_3_in
21 PORT(
22     a : IN std_logic;
23     b : IN std_logic;
24     c : IN std_logic;
25     o : OUT std_logic
26 ); END COMPONENT;
27
28 signal x : std_logic := '0';
29 signal z : std_logic := '0';
30 signal y : std_logic := '0';
31 signal w : std_logic := '0';
32 signal tQ : std_logic := '0';
33 signal tQN : std_logic := '0';
34
35 begin
36 G1 : nor_2_in PORT MAP(
37     a => D,
38     b => z,
39     o => x
40 );
41
42 G4 : nor_2_in PORT MAP(
43     a => w,
44     b => x,
45     o => y
46 );
47
48 G3 : nor_2_in PORT MAP(
49     a => y,
50     b => Clk,
51     o => w
52 );
53
54 G2 : nor_3_in PORT MAP(
55     a => x,

```



```

56      b => Clk ,
57      c => w,
58      o => z );
59
60  --G2 : nor_2_in PORT MAP(
61    a => Clk ,
62    b => w,
63    o => z
64  );
65
66 G6 : nor_2_in PORT MAP(
67   a => w,
68   b => tQN,
69   o => tQ
70 );
71
72 G5 : nor_2_in PORT MAP(
73   a => z ,
74   b => tQ ,
75   o => tQN
76 );
77
78 Q <= tQ;
79 QN <= tQN;
80
81 end Structural ;

```

Listato 2.1: Structural

Come da traccia si è eseguita una prima simulazione di tipo behavioral impostando i ritardi delle porte a 0. Infatti dalla fig. 2.14, che rappresenta tale simulazione, è possibile notare questo comportamento istantaneo ed inoltre il corretto funzionamento del FlipFlop.

Infatti al fronte di discesa del segnale clk il dato (D) viene memorizzato e presentato in uscita (Q) istantaneamente.

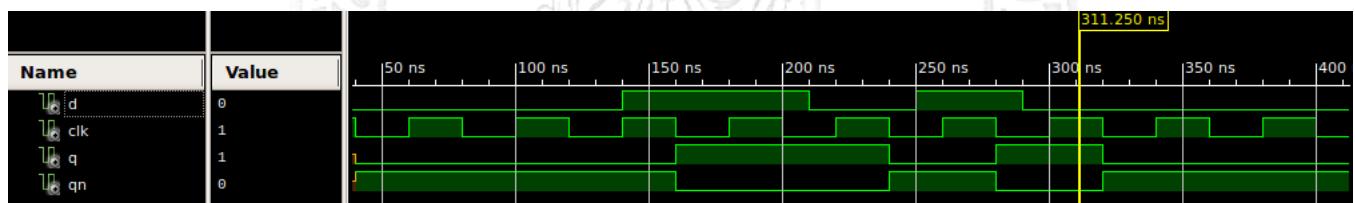


Figura 2.14: Simulazione Behavioral

Successivamente si è deciso di impostare i ridardi di porta a 6 ns (sia il ritardo inerziale che quello di trasporto), ignorando però i ritardi dovuti alle connessioni tra le porte ritenute, quindi, istantanee.

Si riportano in fig. 2.15 ed in fig. 2.16 rispettivamente le simulazioni effettuate con e senza la linea di connessione evidenziata in fig. 2.13.

Nella prima figura è possibile notare che il funzionamento è quello corretto e con un ritardo massimo sull'uscita pari a 3 tau (dove per tau si intende il ritardo di porta) dovuto al fatto che, dal momento del fronte di discesa i segnali vengono calcolati sulle uscite w e z con un tau e successivamente propagati come segnali di set e reset nelle due porte di memorizzazione che portano ulteriori 2 tau di ritardo.

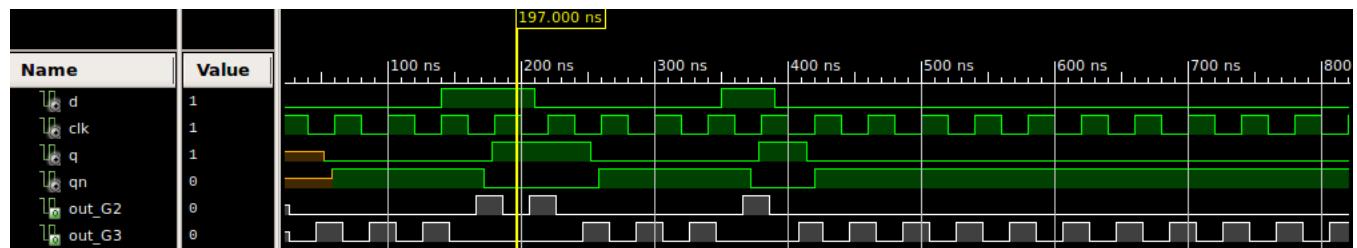


Figura 2.15: Simulazione senza alea

Il caso di fig. 2.16 è il caso in cui non è presente la linea “rossa” di connessione. Ciò comporta che l’elaborazione del dato arriverà a G2 successivamente tramite la porta G3.

A causa di questo ritardo ulteriore viene a crearsi nel nodo Z una alea statica del tipo 0->1->0. Ciò accade perchè, con il clock alto, il segnale w è forzato al valore '0' (sempre indipendentemente dall’ingresso D).

Nel momento di fronte di discesa con clock pari a '0' in una prima fase, di durata tau, la porta G2 fungerà da invertitore e quindi invertirà il valore '0' di w presentando in Z il livello logico '1' (sempre indipendentemente dall’ingresso D).

Se il dato D campionato è '0' la porta g3 presenta alla fine della prima fase in uscita (in W) il livello logico '1'.

Nella seconda fase la porta G2 prendendo come ingresso il livello logico “corretto” '1' presenterà dopo un altro tau il livello logico '0'.

Si osservi che se il dato campionato fosse stato '1' il glitch non si sarebbe verificato.

Il glitch quindi nel nostro caso si presenta dopo 1 tau e finisce dopo 2 tau dall’istante in cui si ha il fronte.

Nella Fig. 2.16 si osservi come il glitch, durando solo un tau, porti temporaneamente lo stato del latch RS nello stato instabile '00'.

Successivamente al glitch il valore basso del clk deve solo un tau (e non due) in quanto la macchina con il glitch è già nello stato instabile '00' e quindi basta solo un tau affinche il latch RS commuti il proprio stato nello stato stabile corretto.

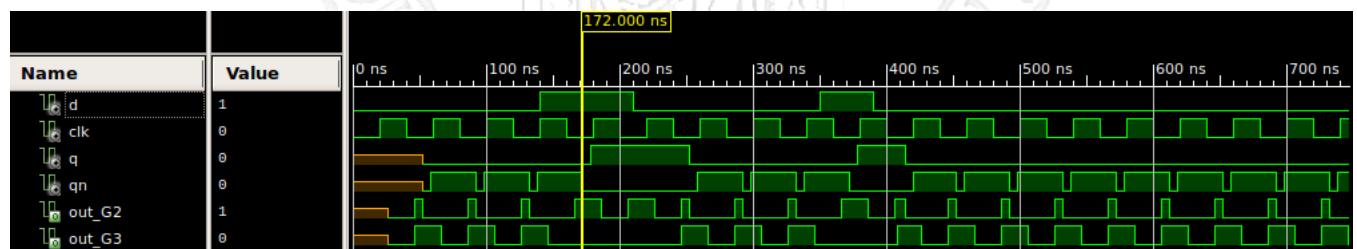


Figura 2.16: Simulazione con alea

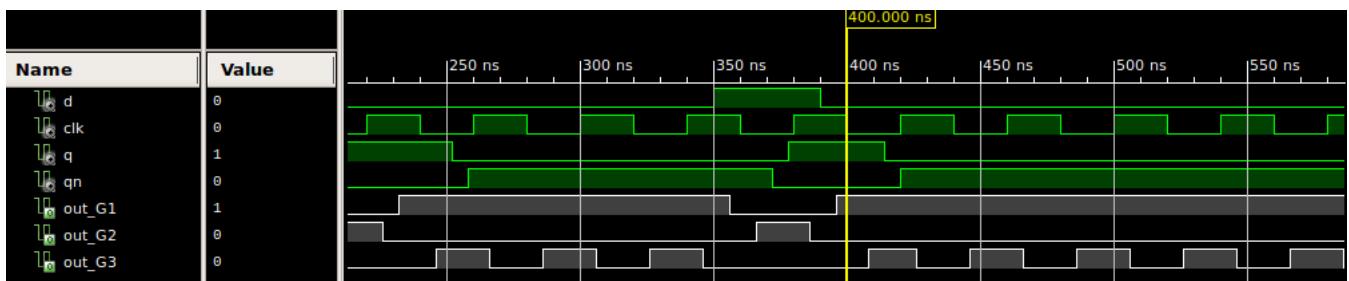


Figura 2.17: Simulazione senza alea 2

E' possibile inoltre notare in fig. 2.17 il perchè inserendo una linea di reazione tra le porte G1 e G2 l'alea scompare.

In questo caso il dato D deve commutare solo un tau prima del fronte di discesa affinchè si propaghi come ingresso di G2.

Quando si ha il fronte di discesa x e y, pari rispettivamente a D' e D, attraverso le porte g2 e g3, che fungono da invertitori, commutano presentando sul nodo S il dato D e sul nodo R il dato D'.

2.3.3 Funzionamento corretto

Tempo di set-up: è il tempo minimo per cui deve rimanere stabile l'input D prima del fronte di clock.

Tempo di hold: è il tempo minimo per cui deve rimanere stabile l'input D dopo il fronte di clock (solitamente trascurabile).

Ulteriori casi test sono stati presi in esame per definire la distanza temporale tra due fronti (quindi la frequenza di funzionamento della macchina) e il duty cycle affinchè si abbia un corretto funzionamento.

2.3.3.1 Struttura con alea

Si è notato che nel caso con alea (e con le ipotesi fatte sui ritardi) è necessario per il funzionamento corretto del dispositivo:

un tempo di 3 tau con livello '0' del clock.

Quando il livello è basso il Dato non si riesce a propagare nel circuito (proprio perchè non è trasparente al dato), allora il Dato ha la possibilità di propagarsi nel circuito solo quando il clk è sul livello alto.

Affinchè la variazione dell'ingresso venga letto il prima possibile è necessario che il clk sia alto e che il segnale D di ingresso commuti 2 tau prima del fronte , quindi il tempo di setup è pari a 2 Tau.

Di conseguenza il livello alto deve essere almeno pari a 2 Tau.

Quindi un Duty Cycle pari a 2/3 (con una frequenza massima di funzionamento pari a $2/3 * \tau_{\text{tau}}$)

2.3.3.2 Struttura senza alea

Si è notato che nella struttura senza alea (e con le ipotesi fatte sui ritardi) è necessario per il funzionamento corretto del dispositivo:

un tempo di 3 tau con livello '0' del clock (1 tau necessari per presentare $R=D'$ e $S=D$ e 2 tau per permettere al latch 3 di memorizzare).

Affinché la variazione dell'ingresso venga letta il prima possibile è necessario che il segnale D di ingresso questa volta però commuti solo 1 tau prima del fronte (sempre con clk alto), quindi il tempo di setup è pari a 2 tau.

Quindi un Duty Cycle pari a 2/3 (con una frequenza massima di funzionamento pari a $2/3*\tau_{\text{au}}$).

Il tempo di hold è trascurabile per entrambi i casi.

2.4 Flip Flop T

2.4.1 Traccia

Si sviluppi il circuito del FlipFlop T sul fronte di salita tramite un Flip Flop D. Eseguire una simulazione comportamentale e post-sintesi, illustrando i passaggi salienti.

2.4.2 Schematic

Nella fig. 2.1 è mostrato tramite lo Schematic la struttura utilizzata per realizzare il Flip FLop T.

E' una Rete sequenziale impulsiva che realizza la sequenza impulsiva tramite i due ingressi, il segnale T e la variabile di sincronismo clk, e tramite un elemento di memoria, il Flip FLop D, pilotato dalla variabile di sincronismo clk.

In particolare l'impulso si avrà solo sul fronte di salita della variabile di sincronismo clk, mentre nel periodo restante la macchina permane nello stato precedente.

Questa volta le commutazioni si possono avere solo sul fronte di salita della variabile di sincronismo clk, e non rispetto al livello di T.

La rete combinatoria utilizzata per calcolare lo stato successivo è una semplice porta XOR.

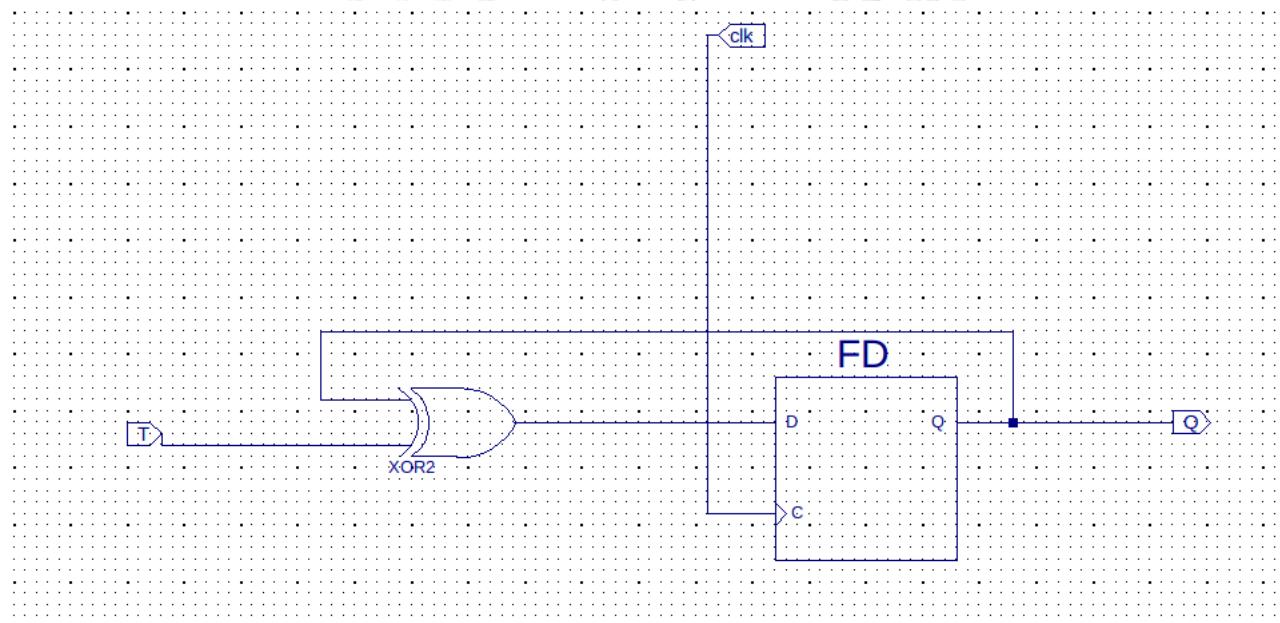


Figura 2.18: Schematic del Flip Flop T

2.4.3 Simulazione Behavioral

In fig. è riportata una simulazione Behavioral del Flip Flop T rappresentato nella Fig.2.1

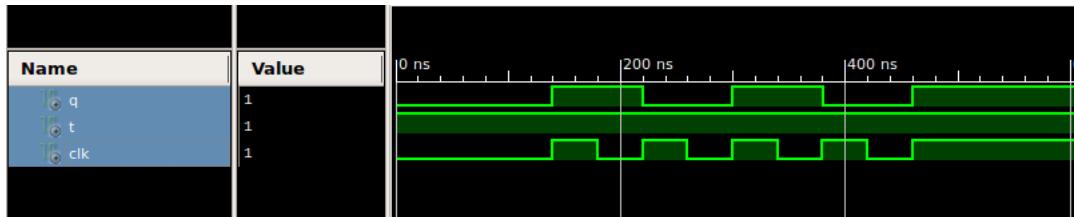


Figura 2.19: Simulazione Behavioral Flip Flop T

2.5 FlipFlop D Master Slave

2.5.1 Traccia

Sviluppare il circuito del FlipFlop di tipo D Master Slave. Eseguire una simulazione comportamentale e post-sintesi, illustrando i passaggi salienti.

2.5.2 Esercizio

Un flip-flop può essere facilmente realizzato ponendo in cascata due latch, uno attivo su un livello del clock e l'altro attivo sul livello opposto, questa configurazione è detta master-slave ed è evidenziata in Fig. 2.20a. Inoltre in Fig. 2.20b è stato rappresentato nel dettaglio il circuito del Latch D utilizzato.



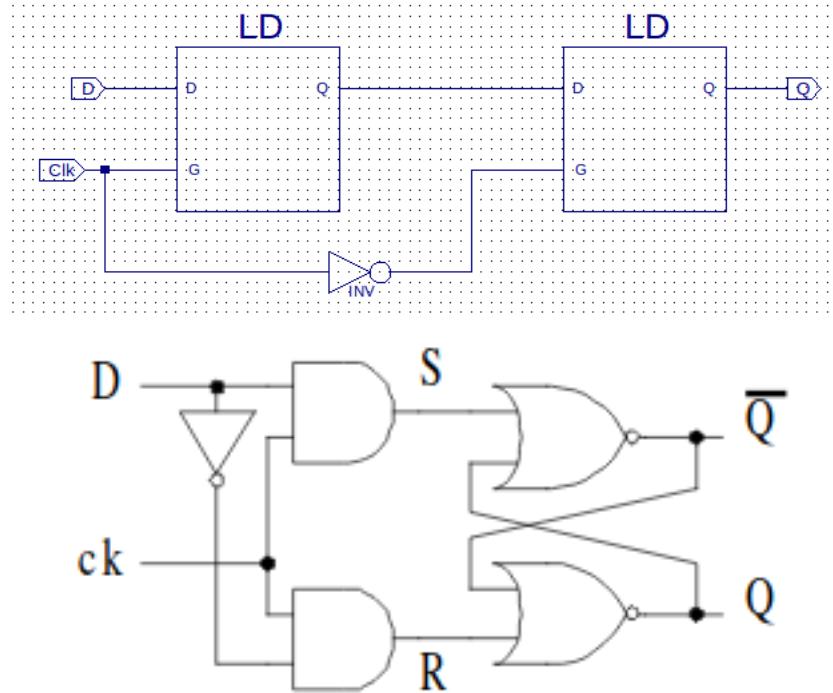


Figura 2.20: Schematic Flip Flop D e D Latch

Si spiegherà adesso brevemente il funzionamento della cascata tra i due Latch D in base al livello del Clock:

- Per $\text{Clk}=1$ il primo latch (master) è trasparente. Le variazioni del segnale D vengono riportate in uscita e si presentano in ingresso al secondo latch (slave). Poiché lo slave è in fase di hold la sua uscita resta costante. Quando il clock commuta dal livello logico alto al livello logico basso il latch slave diviene trasparente ed il dato presente in ingresso viene riportato in uscita.
- Nell'intervallo di tempo in cui $\text{Clk}=0$ il latch master è in fase di hold ed è pertanto insensibile ad eventuali variazioni di D. Si ottiene in questo modo un flip-flop sensibile al fronte di discesa del clock.

Ovviamente, se volessimo ottenere un flip-flop sensibile al fronte di salita del clock è sufficiente invertire la posizione dei due latch, in modo tale che il master sia trasparente per $\text{Clk}=0$ e lo slave per $\text{Clk}=1$.

Per mostrare il funzionamento del dispositivo si è eseguita dapprima una simulazione in modo da constatarne il corretto funzionamento. Difatti, osservando la Fig. 2.21 è possibile notare che nonostante il dato D vari nell'intervallo di trasparenza del master, ovvero con $\text{Clk} = '1'$, il valore che viene effettivamente propagato in uscita (sempre che rispetti il vincolo del tempo di Setup) è quello “catturato” sul fronte di discesa del Clock.

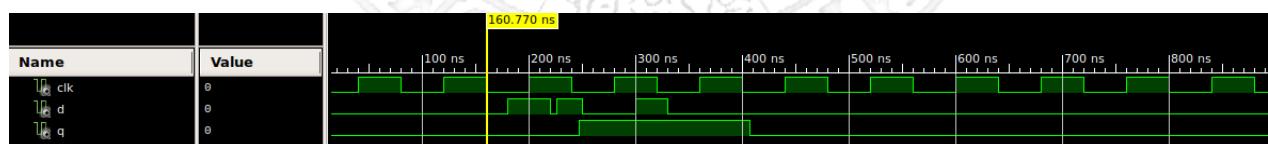


Figura 2.21: Simulazione

E' importante notare, osservando la Fig. 2.20a, che il circuito presenta una "asimmetria" sulla propagazione del clock ai due Latch. Infatti l'inverter che porta il segnale Clk allo slave introduce un ritardo. Ciò potrebbe portare ad una anomalia sulla variazione 0 -> 1 del Clock in quanto per una breve finestra temporale detta "di trasparenza" entrambi i Latch trovano il segnale di abilitazione attivo.

Sebbene sia importante porre l'attenzione su questo particolare, difficilmente si creano problemi stante la piccola durata della finestra.

Tuttavia si è deciso di mostrare una simulazione di questa anomalia riprendendo il Latch RS sviluppato precedentemente e sostituendoli ai Latch D trattati fino ad ora.

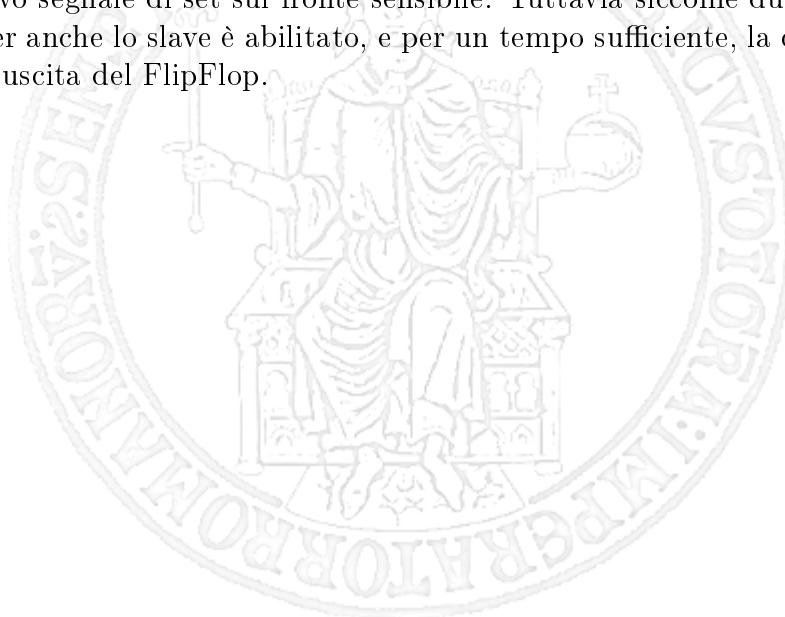
Quello che si utilizzerà, risultante dalla composizione dei Latch RS, sarà dunque un **Flip Flop RS Master Slave sensibile al fronte di discesa**.

Nella simulazione illustrata in Fig. 2.22 si è volutamente introdotto un ritardo molto grande nell'inverter così da avere una finestra di trasparenza di durata 25 ns, equivalente ad un quarto di periodo di clock. Affinchè si verifichino delle vere e proprie anomalie è necessario infatti che il clock che abilita lo slave abbia un ritardo sufficientemente grande da mantenerlo in trasparenza per tutta la durata che è somma di due tempi:

- il tempo che permette la commutazione del master (nel caso si presenti un ingresso che lo forzi a variare le uscite)
- la durata dell'impulso in ingresso allo slave necessaria a farlo commutare

Tuttavia, nonostante si sia brevemente dimostrato che raramente questo fenomeno porta ad anomalie, per completezza si è simulata in Fig. 2.22a una possibile configurazione di ingressi-uscite errata e che si discosta, quindi, dalla logica di funzionamento del dispositivo. Per comprendere meglio i nomi dei segnali rappresentati fare riferimento alla Fig. 2.22b.

Nell'esempio in figura infatti, il normale funzionamento del FlipFlop sarebbe quello di mantenere costante ed alta l'uscita, nonostante l'impulso di reset. Questo perchè in successione a questo impulso vi è un nuovo segnale di set sul fronte sensibile. Tuttavia siccome durante la prima commutazione del master anche lo slave è abilitato, e per un tempo sufficiente, la commutazione viene riportata anche sull'uscita del FlipFlop.



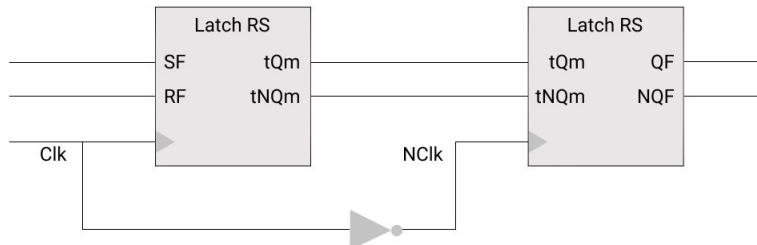
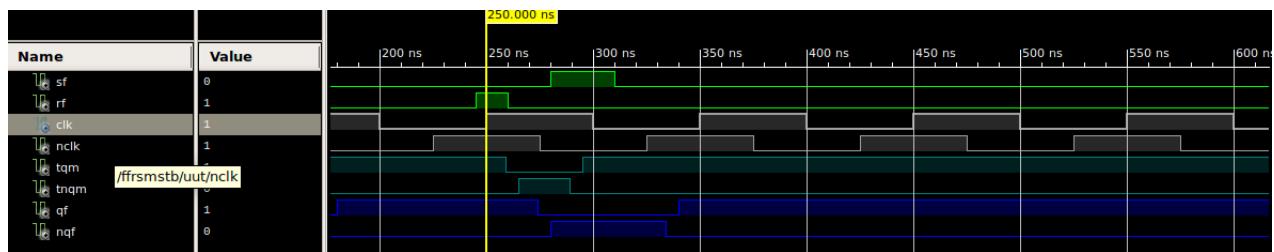


Figura 2.22: Simulazione con anomalia

Per completezza si riporta il codice sviluppato per implementare il dispositivo. I componenti che sono stati utilizzati sono il latch RS precedentemente illustrato aggiungendo un segnale di abilitazione ed un inverter. Per semplicità tuttavia non se ne riportano nuovamente le implementazioni.

```

1 library IEEE;
2 use IEEE.STD_LOGIC_1164.ALL;
3
4
5 entity FFRSMS is
6 Port ( RF : in STD_LOGIC;
7        SF : in STD_LOGIC;
8        Clk : in STD_LOGIC;
9        QF : out STD_LOGIC;
10       NQF : out STD_LOGIC);
11 end FFRSMS;
12
13
14 architecture Structural of FFRSMS is
15
16 COMPONENT Latch_rs
17 generic (
18   DELAYQT: time := 1.0 ns;
19   DELAYNQT: time := 1.0 ns;
20   DELAY_INERTIAL_NOR: time := 3 ns;
21   DELAY_TRANSPORT_NOR: time := 5 ns;
22   DELAY_TAN : time := 1.0 ns;
```

```

23  DELAY_INERTIAL_AND: time := 1 ns;
24  DELAY_TRANSPORT_AND: time := 3 ns );
25  PORT( E,R,S: IN std_ulogic ;
26  Q,NQ: inout std_ulogic );
27 END COMPONENT;

28
29 COMPONENT invert
30 generic (
31  DELAY_INERTIAL: time := 3 ns;
32  DELAY_TRANSPORT: time := 25 ns );
33  PORT( I: IN std_ulogic ;
34  O: OUT std_ulogic );
35 END COMPONENT;

36
37 signal tQm : std_logic := '0' ;
38 signal tNQm : std_logic := '0' ;
39 signal tQF : std_logic := '0' ;
40 signal tNQF : std_logic := '0' ;
41 signal NClk : std_logic := '0' ;

42
43 begin
44
45
46 Master : Latch_rs PORT MAP(
47  E => Clk ,
48  R => RF,
49  S => SF ,
50  Q => tQm,
51  NQ => tNQm ) ;

52
53 Inverter: invert PORT MAP(
54  I => Clk ,
55  O => NClk
56 ) ;

57
58 Slave : Latch_rs PORT MAP(
59  E => NClk ,
60  R => tNQm,
61  S => tQm,
62  Q => tQF ,
63  NQ => tNQF ) ;

64
65 QF <= tQF ;
66 NQF <= tNQF ;
67
68

```

69 | **end Structural;**

Listato 2.2: FlipFlop RS master-slave



Capitolo 3

Display a 7 segmenti

3.1 Display a 7 Segmenti

3.1.1 Traccia

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

3.1.2 Soluzione

Si è scelto di definire il numero intero da mostrare sul display tramite 8 switches e 4 buttons della board. Gli 8 switches serviranno per definire una stringa di 8 bit. Questi, al primo fronte di salita del clock avranno una funzione differente a seconda di quale bottone si sia premuto e di conseguenza quale bit tra quelli associati ai bottoni sia alto.

La control unit si occuperà di presentare dei bit di configurazione su un banco di registri (come si può vedere nella figura) che potranno essere letti, interpretati e mostrati sul display a sette segmenti.

Il secondo modulo Display_seven_segment dopo aver letto la configurazione presentata dalla control_unit presenterà in uscita 12 bit, i cui valori, tramite due bus, andranno ad illuminare dei led specifici.

Un bus composto da 8 linee servirà per configurare quali led accendere di una singola cifra (7 led per la rappresentazione di un valore più 1 led del dot). In particolare i led di ogni cifra in posizione analoga hanno il catodo in comune. Questi 8 bit secondo una logica 0-attiva definiranno quali degli otto catodi totali saranno posti a massa, tuttavia solo i led appartenenti a un anodo abilitato condurranno e di conseguenza si illumineranno. Il secondo bus, composto da quattro linee, è utilizzato per pilotare quali anodi sono abilitati.

Si sfrutteranno i limiti di campionamento dell'occhio umano e si alternerà ogni cifra con il suo corrispettivo valore con una frequenza tale da sembrare sempre presente.

Ogni periodo è diviso in quattro fasi, in ogni fase verrà abilitata una e una sola cifra che mostrerà il suo corrispettivo valore.

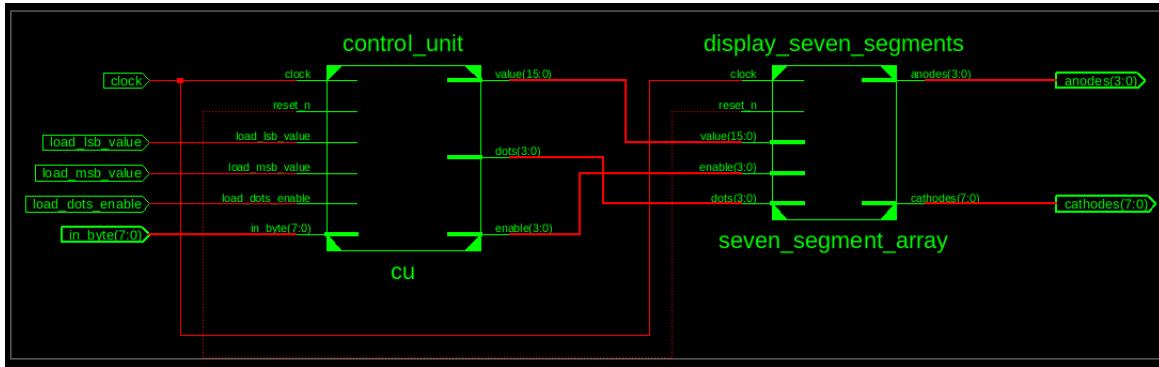


Figure 3.1: Schematic RTL a macrocomponenti

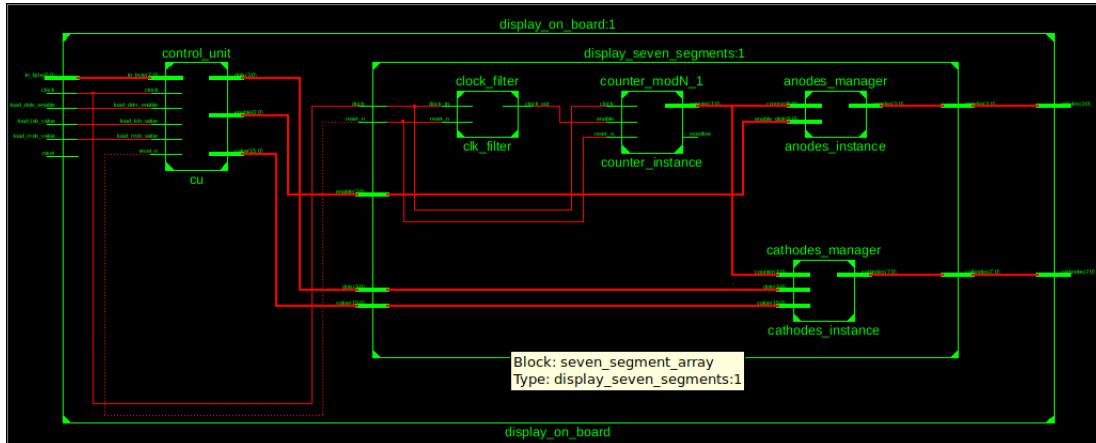


Figure 3.2: Schematic che evidenzia le componenti interne del Display_seven_segments

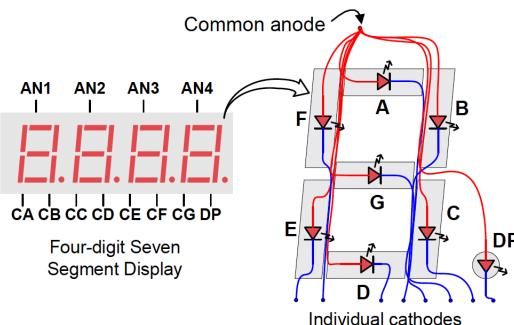


Figure 3.3: Seven Segment Display

3.1.2.1 Descrizione strutturale del modulo Display_Seven_Segment

Per dividere la frequenza di N si utilizza un contatore modulo N. In particolare ad ogni fronte di salita del clock il contatore viene incrementato. Quando arriva ad N prima di resettarsi invia in uscita un impulso di abilitazione a un contatore di modulo 4. Ogni conteggio del contatore

modulo 4 andrà a scandire una delle quattro fasi citate precedentemente, andando ad abilitare sia una delle quattro cifre e sia quali dei 4 bit di 16 dovranno essere mostrati nella cifra.

Si è utilizzato il progetto contenuto nell'ftp apportando alcune modifiche. Si è creata una entity contatore_Mod_N lasciando non definito il numero di conteggi, ciò favorendo il riuso del componente già implementato del contatore mod_4.

In totale quindi sono stati istanziati 2 contatori:

- Uno modulo N come realizzazione del divisore di frequenza
- Un secondo contatore modulo 4 utile a scandire le 4 fasi di abilitazione delle 4 cifre con rispettivi valori per i 7 segmenti e il dot

Si riporta quindi di seguito il codice del contatore generalizzato nel caso modulo_N, quindi la sua istanziazione all'interno della entity "clock filter" che definisce la divisione della frequenza entrante del clock.

```

1 library IEEE;
2 use IEEE.STD_LOGIC_1164.ALL;
3 use IEEE.math_real.all;
4 use IEEE.std_logic_arith.all ;
5 use IEEE.NUMERIC_STD.ALL;

6
7 entity counter_modN is
8 generic (n: Integer := 4);
9 Port (
10    clock : in STD_LOGIC;
11    reset_n : in STD_LOGIC;
12    enable : in STD_LOGIC;
13    counter : out STD_LOGIC_VECTOR(integer (floor (LOG2( real (N-1) )))-
14    Downto 0);
15    overflow: out std_logic );
16 end counter_modN;

17 architecture Behavioral of counter_modN is
18
19 signal c : STD_LOGIC_VECTOR(integer (floor (LOG2( real (N-1) )))Downto 0)
20 := (others => '0');
21 signal reset : std_logic;
22
23 begin
24
25 counter <= c;
26 reset <= not reset_n;
27 counter_process: process (clock , reset)
28
29 begin

```

```

30 if reset = '1' then
31   c <= (others => '0');
32   overflow <= '0';
33 elsif clock'event AND clock = '1' AND enable = '1' then
34   c <= CONV_STD_LOGIC_VECTOR(unsigned(c) + 1, integer(floor(LOG2(real(
35     N))))));
36   if unsigned(c) = 0 then
37     overflow <= '1';
38   else
39     overflow <= '0';
40   end if;
41 end if;
42
43 end process;
44 end Behavioral;

```

Listing 3.1: Contatore modulo N

```

1 library IEEE;
2 use IEEE.STD_LOGIC_1164.ALL;
3 use IEEE.math_real.all;
4 use IEEE.std_logic_arith.all ;
5 use IEEE.NUMERIC_STD.ALL;
6
7 entity clock_filter is
8 generic(
9   clock_frequency_in : integer := 50000000;
10  clock_frequency_out : integer := 5000000);
11
12 Port ( clock_in : in STD_LOGIC;
13   reset_n : in STD_LOGIC;
14   clock_out : out STD_LOGIC);
15 end clock_filter;
16
17 architecture Structural of clock_filter is
18
19 component counter_modN is
20 generic (n: Integer := 4);
21 Port (
22   clock : in STD_LOGIC;
23   reset_n : in STD_LOGIC;
24   enable : in STD_LOGIC;
25   counter : out STD_LOGIC_VECTOR(integer(floor(LOG2(real(N-1)))))-
26   DOWNTO 0);
27   overflow: out std_logic);
end component;

```

```
28
29 constant count_max_value : integer := clock_frequency_in /(
   clock_frequency_out) - 1;
30
31 begin
32
33 counter_modx : counter_modN
34 generic map(n => count_max_value)
35 port map(
36   clock => clock_in,
37   reset_n => reset_n,
38   enable = > '1',
39   counter => open,
40   overflow => clock_out);
41
42 end Structural;
```

Listing 3.2: Clock Filter



Capitolo 4

Clock Generator

4.0.1 Traccia

Sviluppare un progetto di sintesi di un DCM.

4.0.2 Soluzione

Il *Digital Clock Manager (DCM)* è un componente *IP-CORE* fornito da ISE che ci permette di manipolare i segnali di clock e di eliminare l'effetto di *skew* che causerebbe errori nel circuito. L'architettura di un DCM è composta di quattro componenti principali come in figura:

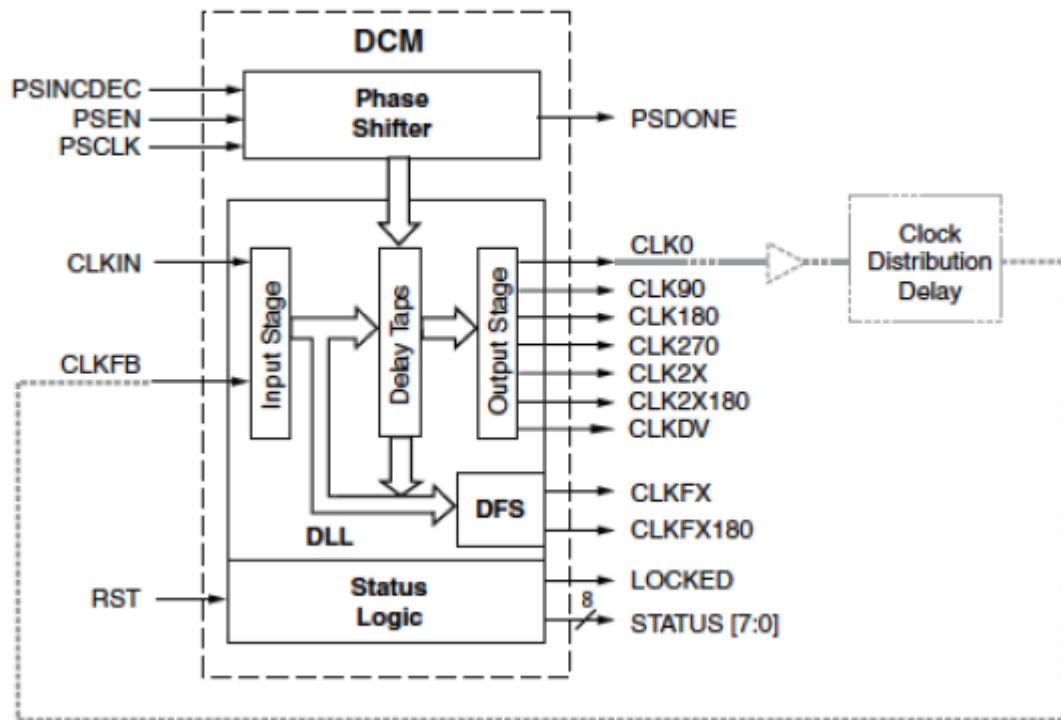


Figura 4.1: Schema DCM

Il *DLL* è la componente adibita al controllo dello skew dei segnali. Ci interessa porre enfasi sulla componente *Digital Frequency Synthesizer (DFS)* che ci permette di generare un clock di frequenza preassegnata partendo da quello in ingresso CLKIN modificato tramite un moltiplicatore CLKFX_MULTIPLY e un divisore CLKFX_DIVIDE. Dato che questi due fattori devono essere interi, non è possibile ottenere in uscita una qualsiasi frequenza utilizzando un solo DCM.

$$F_{clkfx} = F_{clkin} \frac{CLKFX_{MULTIPLY}}{CLKFX_{DIVIDE}}$$

Il nostro progetto vuole far operare un Moltiplicatore di Booth alla sua frequenza massima. Quest'ultima è ottenuta dal report di ISE dopo aver effettuato la sintesi ed è pari a 193.979MHz. Il componente DCM è importato all'interno del progetto come un IP-CORE ed è facilmente configurabile tramite un tool grafico. In questo tool è possibile scegliere la frequenza del CLKIN, nel nostro caso 50MHz, e scegliere quella di uscita. Abbiamo scelto come frequenza di uscita 190MHz. Il tool grafico ci permette di sapere anche quali sono stati i fattori di moltiplicazione e divisione utilizzati. In questo caso il DCM ha effettuato la seguente operazione $50 \div 5 = 10 \times 19 = 190$.

4.0.2.1 Schematici

Lo schematico risultante dell'inclusione del DCM è il seguente

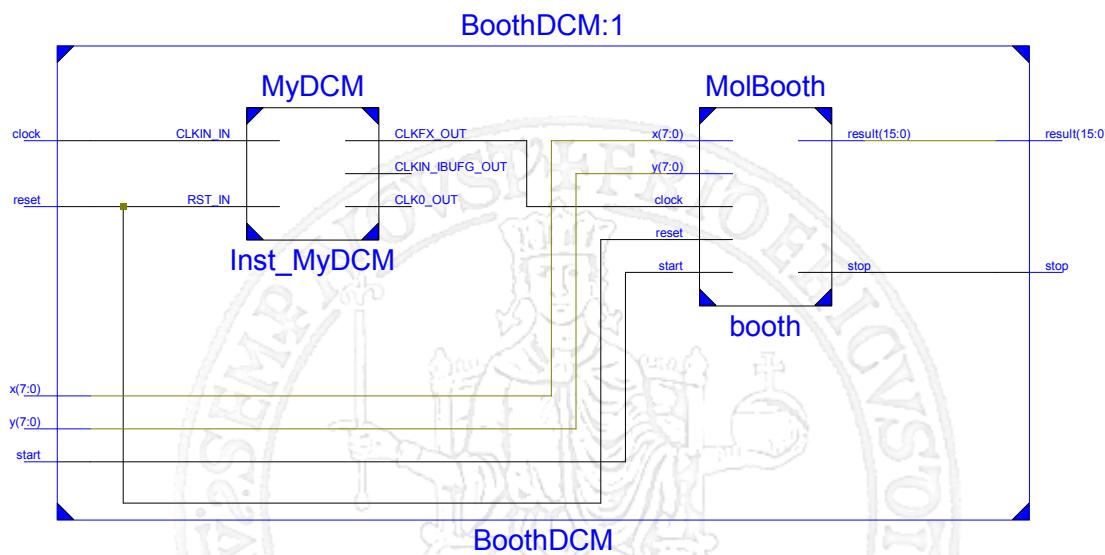


Figura 4.2: Booth con DCM

4.0.2.2 Codice

```

1 library IEEE;
2
3 library IEEE;
4 use IEEE.STD_LOGIC_1164.ALL;
5
6 -- Uncomment the following library declaration if using
7 -- arithmetic functions with Signed or Unsigned values
8 --use IEEE.NUMERIC_STD.ALL;
9
10 -- Uncomment the following library declaration if instantiating
11 -- any Xilinx primitives in this code.
12 --library UNISIM;
13 --use UNISIM.VComponents.all;
14
15 entity BoothDCM is
16
17
18 Generic ( N: integer := 8 );
19 port (
20     x : in STD_LOGIC_VECTOR (N-1 downto 0);
21     y : in STD_LOGIC_VECTOR (N-1 downto 0);
22     start : in STD_LOGIC;
23     reset : in STD_LOGIC;
24     stop : out STD_LOGIC;
25     clock : in STD_LOGIC;
26     result : out STD_LOGIC_VECTOR ((N+N)-1 downto 0)
27 );
28
29 end BoothDCM;
30
31 architecture Structral of BoothDCM is
32
33 component MolBooth is
34 Generic ( N: integer := 8 );
35 port (
36     .
37     .
38     .
39     .
40     .
41     .
42     .
43 );
44 end component;
45

```

```

46
47 COMPONENT MyDCM
48 PORT(
49   CLKIN_IN : IN std_logic;
50   RST_IN : IN std_logic;
51   CLKFX_OUT : OUT std_logic;
52   CLKIN_IBUFG_OUT : OUT std_logic;
53   CLK0_OUT : OUT std_logic
54 );
55 END COMPONENT;
56
57
58 signal clock_fixed : std_logic := '0';
59
60 begin
61 booth : MolBooth port map(
62
63   x=>x,
64   y=>y,
65   start=>start,
66   stop=>stop,
67   reset=>reset,
68   result=>result,
69   clock=>clock_fixed
70 );
71
72 );
73
74 Inst_MyDCM: MyDCM PORT MAP(
75   CLKIN_IN => clock,
76   RST_IN => reset,
77   CLKFX_OUT => clock_fixed,
78   CLKIN_IBUFG_OUT => open,
79   CLK0_OUT => open
80 );
81
82
83 end Structural;

```

Listing 4.1: Implementazione Structural Booth con DCM

4.0.3 Simulazione

Mostriamo ora i risultati delle simulazioni ottenute con e senza l'utilizzo del DCM. Attraverso i grafici è possibile apprezzare l'aumento di prestazioni ottenute dal Moltiplicatore di Booth con DCM.

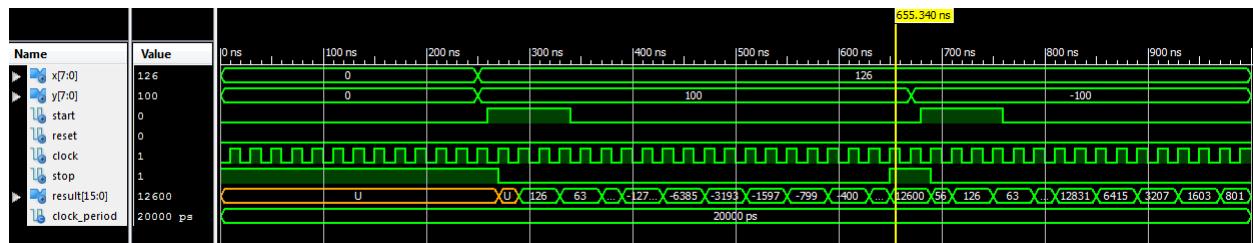
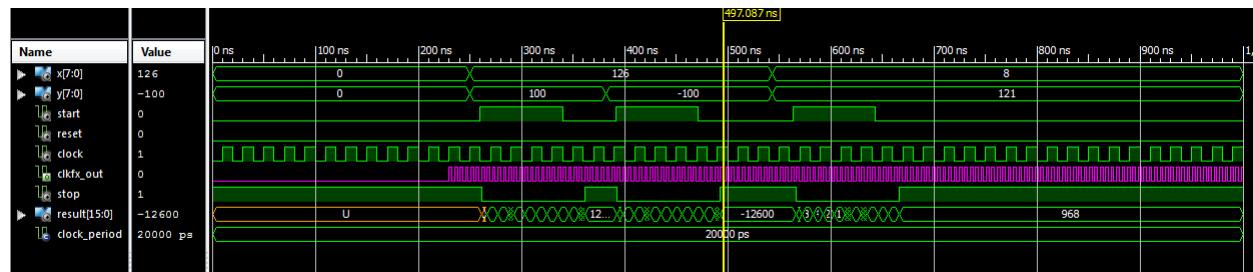
```

1  — Stimulus process
2  stim_proc: process
3  begin
4
5      — test1
6      X <= x"7E"; — 126
7      Y <= x"64"; — 100
8      wait for 10 ns;
9          start <= '1';
10     wait for 80 ns;
11     start <= '0';
12     wait until stop = '1';
13     — risultato atteso: 12600 = 0x3138
14     wait for 20 ns;
15
16     — test2
17     Y <= x"9C"; — -100
18     wait for 10 ns;
19     start <= '1';
20     wait for 80 ns;
21     start <= '0';
22     wait until stop = '1';
23     — risultato atteso: -12600 = 0xCEC8
24     wait for 50 ns;
25
26     — test3
27     X <= x"08"; — 8
28     Y <= x"79"; — -128
29     wait for 20 ns;
30     start <= '1';
31     wait for 80 ns;
32     start <= '0';
33     wait until stop = '1';
34     — risultato atteso: -1024 = 0xFC00
35
36     wait;
37 end process;

```

Listing 4.2: Simulazione Booth

Segue il risultato del testbench:


 Figura 4.3: *Testbench Booth*

 Figura 4.4: *Testbench Booth con DCM*


Capitolo 5

Scan Chain

5.1 Scan Chain

5.1.1 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.

5.1.2 Soluzione

La traccia è stata interpretata in questo modo:

```
1 entity ScanChain is
2     —N è la dimensione dello shift register e quindi definisce anche il
3     —modulo del contatore
4     generic (N : integer := 8) ;
5     port (
6         —ingressi di dato
7         —valore logico "scan_in" da inserire in una posizione codificata
8         —su "posizione"
9         scan_in:in std_logic;
10        posizione: in std_logic_vector(integer(Floor(log2(real(N-1)))) 
11        downto 0);
12        —ingressi logici da imporre al registro quando si opera in
13        —modalità normale (mode='0')
14        reg_in: in std_logic_vector(N-1 downto 0);
15        —ingressi di sincronismo
16        —è la variabile di sincronismo
17        clock:in std_logic;
18        reset_n:in std_logic;
```

```

15  — ingressi di controllo
16      start:in std_logic;
17      — valore logico basso (mode='0') per fare la semplice scrittura
18          sul registro altrimenti
19      — valore logico alto per lavorare nella seconda modalità (mode
20          ='1')
21      mode:in std_logic;
22
23  — uscite di dato Q
24      Q:out std_logic_vector( N-1 downto 0 );
25  — uscite di Stato
26      stop:out std_logic
27  ); end ScanChain;

```

Listing 5.1: Interfaccia della ScanChain

Nel Lis.5.1 è mostrata l'interfaccia del componente Scan_Chian implementato in questo elaborato.

La rete ha come ingresso un segnale di controllo “mode” e a seconda del valore logico che assume questa variabile di ingresso la Scan Chain si comporterà nei due possibili modi descritti di seguito.

5.1.2.1 Modalità normale

Ponendo Mode al valore logico basso la Scan Chain opera in modalità Normale, ovvero effettua la semplice scrittura sul registro.

In questo caso il vettore di valori logici applicati alle porte di ingresso Reg_in sarà memorizzato nel registro della Scan Chain.

5.1.2.2 Modalità Shift

Ponendo Mode al valore logico alto la Scan Chain opera nellamodalità Shift. In tal caso dovranno essere configurati anche gli ingressi “scan_in” e “posizione”.

Scan_in è il valore logico che dovrà essere posto all'elemento di memoria del registro di posizione pari a quella codificata nell'ingresso “posizione”.

Per fare ciò si opera in questa modo:

- in un primo passaggio il registro viene shiftato di una posizione utilizzando come valore logico l'ingresso di dato “scan_in” eliminando il bit meno significativo del registro.

- in un secondo passaggio vengono effettuate tante operazioni di shift quante quelle necessarie affinchè il valore logico memorizzato in prima posizione arrivi nella posizione desiderata (codificata nell'ingresso “posizione”).

Lo Shift Register in questo secondo passaggio dovrà essere configurato in modo tale da essere una catena a ciclo chiuso (e non come una catena aperta), ovvero l'uscita di Scan_out dello Shift Register dovrà essere collegata fisicamente all'ingresso di Scan_in. Per realizzare questa parte operativa si è utilizzato un componente MUX a due ingressi come ingresso di Scan_in dello Shift register.

5.1.2.3 Una panoramica strutturale alto livello della Scan Chian

Nella Fig.5.1 è mostrata una panoramica alto livello della struttura della Scan Chian implementata in questo elaborato.

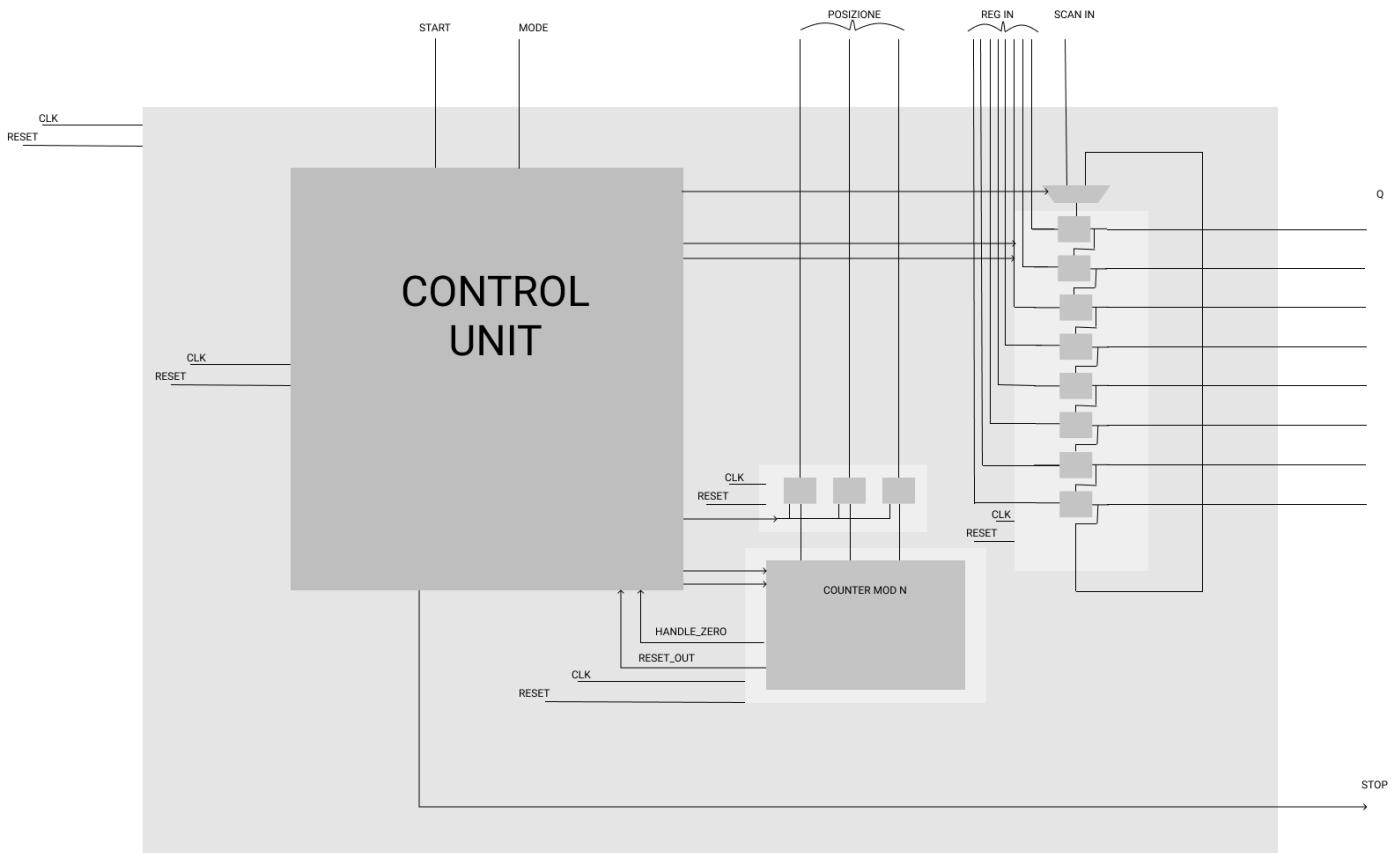


Figura 5.1: Panoramica strutturale di alto livello della Scan Chain

Per tenere il conto dei colpi di clock si è utilizzato un contatore modulo N, questa volta però è leggeremnte diverso da quello visto nel Display a sette segmenti.

Esso prende in ingresso un numero massimo di conteggi, oltre il quale, l'uscita assumerà un valore logico alto.

5.1.3 Implementazione

5.1.3.1 Shift Register

Per risolvere la traccia si è utilizzato nella parte operativa uno Shift Register costituito da N flip flop D Multiplexati.

Un flip flop multiplexato è un flip flop D il cui ingresso di Dato è multiplexato da un multiplexer a due ingressi.

A seconda del segnale di abilitazione ogni flip flop D multiplexato o prende in ingresso l'uscita di quello posizionato prima nella catena per operare come uno shift register (con shift a destra) oppure quello di Dato al fine di far operare lo Shift Register da semplice registro.

```

1 entity Shift_register is
2   generic( N:integer:= 8; shift_right_left:std_logic := '0') ;
3   port(
4     mux_scan_in: std_logic ;
5     reg_in: in std_logic_vector(N-1 downto 0) ;
6     scan_in_dato: in std_logic ;
7     scan_in_controllo: in std_logic ;
8     scan_enable: in std_logic ;
9     clk: in std_logic ;
10    reset_n: in std_logic ;
11    enable: in std_logic ;
12    Q: out std_logic_vector(N-1 downto 0) ;
13    scan_out:out std_logic
14  );
15 end Shift_register ;
16
17 architecture Behavioral of Shift_register is
18   component flipflopmux
19   port(
20     scan_in: in std_logic ;
21     D: in std_logic ;
22     scan_enable: in std_logic ;
23     clk: in std_logic ;
24     reset_n: in std_logic ;
25     enable: in std_logic ;
26     Q: out std_logic
27   );
28 end component ;
29
30 component MUX2
31   port(
32     in0: in std_logic ;
33     in1: in std_logic ;
34     sel: in std_logic ;
35     out_mux: out std_logic );
36 end component ;
37
38   signal temp_vector :std_logic_vector(N downto 0) ;
39   signal out_temp:std_logic ;
40 begin
41   —shift verso destra
42   mux: MUX2 port map( in0=>scan_in_dato , in1=>scan_in_controllo , sel=>
43                         mux_scan_in , out_mux=>out_temp ) ;
44
45   LOOP_FOR_INSTANTIATE_FFM: for i in N-1 downto 0 generate

```

```

45
46     shift_right: if( shift_right_left= '0') generate
47         ffm: flipflopmux port map( scan_in=>temp_vector(i+1), D=>
48             reg_in(i),
49             scan_enable=>scan_enable,
50             clk=>clk, reset_n=>reset_n,
51             enable=>enable, Q=>temp_vector(i));
52             temp_vector(N)<=out_temp;
53             scan_out<= temp_vector(0);
54             Q<=temp_vector(N-1 downto 0);
55     end generate shift_right;
56     --shift verso sinistra
57     shift_left: if( shift_right_left= '1') generate
58         ffm: flipflopmux port map( scan_in=>temp_vector(i-1), D=>
59             reg_in(i),
60             scan_enable=>scan_enable,
61             clk=>clk, reset_n=>reset_n,
62             enable=>enable, Q=>temp_vector(i));
63             temp_vector(0)<=out_temp;
64             scan_out<= temp_vector(N);
65             Q<=temp_vector(N-1 downto 0);
66     end generate shift_left;
67
68
69
70
71
72 end Behavioral;

```

Listing 5.2: Shift Register con Scan_in Multiplexato

```

1 library IEEE;
2 use IEEE.STD_LOGIC_1164.ALL;
3
4 entity flipflopmux is
5     port(
6         scan_in: in std_logic;
7         D: in std_logic;
8         scan_enable: in std_logic;
9         clk: in std_logic;
10        reset_n: in std_logic;
11        enable: in std_logic;
12        Q: out std_logic
13    );
14 end flipflopmux;

```

```

15
16 architecture Structural of flipflopmux is
17 component MUX2
18 port(
19   in0: in std_logic;
20   in1: in std_logic;
21   sel: in std_logic;
22   out_mux: out std_logic);
23 end component;
24
25 component FFD
26 Port ( clk : in STD_LOGIC;
27         en : in STD_LOGIC;
28         reset_n : in STD_LOGIC;
29         D : in STD_LOGIC;
30         Q : out STD_LOGIC);
31 end component;
32
33 signal out_temp:std_logic;
34
35 begin
36
37 mux: MUX2 port map( in0=>D, in1=>scan_in , sel=>scan_enable , out_mux=>
38 out_temp );
39
40 flipflop: FFD port map( clk=>clk ,en=>enable , reset_n=>reset_n , D=>
41 out_temp ,Q=>Q);
42
43 end Structural;

```

Listing 5.3: Flip Flop D Multiplexato

5.1.3.2 Contatore Modulo N con Numero di Conteggi Massimo come ingresso

Questo Contatore Modulo N differentemente da quello mostrato nel Display a 7 segmenti ha un vettore di variabili di ingresso su cui è possibile codificare con una rappresentazione Modulo (positiva) un valore da 0 a N-1. Il contatore ha due variabili di uscita:

“Handle_zero” sarà alto se Massimo Numero di Conteggi è codificato come tutti zero. Questa variabile di uscita sarà un segnale di stato utile alla Control Unit per gestire il caso in cui si voglia solo effettuare lo Shift di una posizione.

“res_out” sarà alto quando il Numero di Conteggi Massimo definito in ingresso è pari al conteggio corrente.

```

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

```

```

3 | use IEEE.math_real.all;
4 | use IEEE.std_logic_arith.all ;
5 |
6 |
7 entity Counter is
8   generic (N:integer:=8);
9   port(
10    clock: in std_logic;
11    reset_n: in std_logic;
12    enable:in std_logic;
13    count_max: in std_logic_vector(integer(Floor((log2(real(N-1))))))
14      downto 0);
15    res_out: out std_logic:='0';
16    Handle_zero: out std_logic:='0';
17  );
18 |
19 end Counter;
20 |
21 architecture Behavioral of Counter is
22   —signal count: std_logic_vector(integer(Floor((log2(real(N))))))-1
23   —downto 0):=(others=>'0');
24   signal count :integer range 0 to N-1 ;
25   signal temp_out: std_logic;
26   signal temp_out1: std_logic;
27   signal c_max: integer;
28 |
29 begin
30   c_max <= conv_integer(unsigned(count_max));
31   res_out <= temp_out;
32   Handle_zero<= temp_out1;
33   counter_process: process(clock,enable,reset_n) begin
34     if (reset_n='0') then
35       count<=0;
36       temp_out <= '0';
37       temp_out1<= '0';
38     elsif (rising_edge(clock) and enable ='1') then
39       count <= count+1;
40       if(c_max=0) then
41         count<=0;
42         temp_out1<= '1';
43       elsif(count = c_max-1) then
44         count<=0;
45         temp_out<= '1';
46

```

```

47      temp_out1<= '0';
48      else
49          temp_out<= '0';
50          temp_out1<= '0';
51      end if;
52
53  end if;
54
55
56 end process;

```

Listing 5.4: Contatore modulo N

5.1.3.3 Control Unit

In Fig.5.2 è mostrato il diagramma degli stati della macchina sequenziale Sincrona di tipo Moore.

Invece nel Lis. 5.5 è mostrato il codice in VHDL.

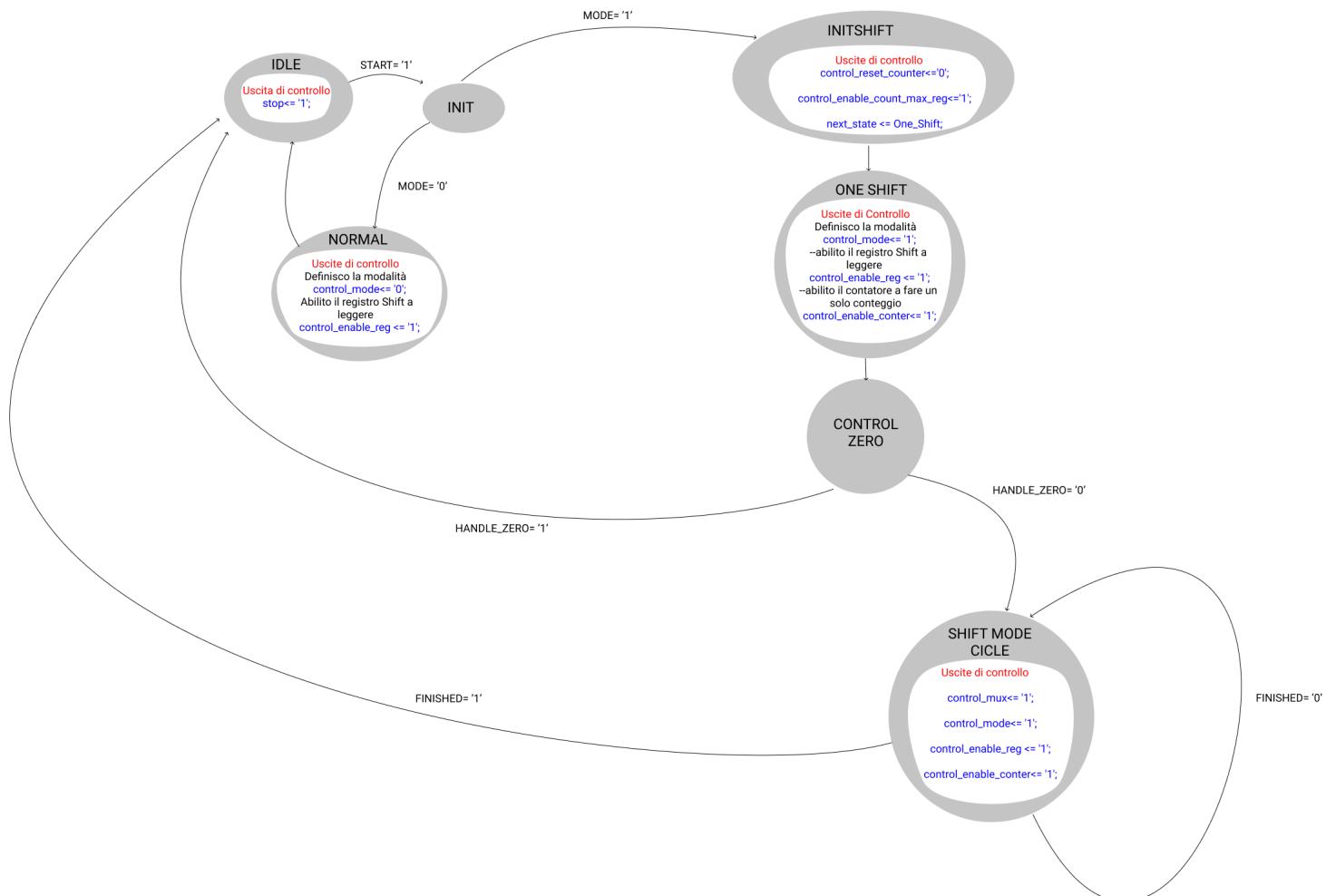


Figura 5.2: State Chart della Control Unit

```

1 library IEEE;
2 use IEEE.STD_LOGIC_1164.ALL;
3 use IEEE.math_real.all;
4 use IEEE.std_logic_arith.all ;
5 use IEEE.NUMERIC_STD.ALL;
6
7
8 entity ControlUnit is
9
10 port(
11
12     reset_n:in std_logic;
13     --segnali di stato
14     --mi dice quando il contatore ha finito il conteggio
15     finished:in std_logic;
16     --mi serve per gestire il caso in cui si debba fare un solo Shift
17     Handle_zero:in std_logic;
18     --ingressi di controllo
19     --è il segnale di start per avviare una delle due operazioni
20     start: in std_logic;
21     --0 è per fare la semplice scrittura su registro altrimenti 1 per
22     --fare la modalità shifter
23     mode:in std_logic;
24     --è la variabile di sincronismo
25     clock:in std_logic;
26     --segnali di controllo in uscita
27     --segnale di controllo che comanda il modo di operare dello shift
28     --register
29     control_mode: out std_logic;
30     --segnali di controllo che abilitano rispettivamente il contatore
31     --e il registro
32     control_enable_counter: out std_logic;
33     control_enable_reg: out std_logic;
34     --segnali di reset contatore e di reset registro
35     control_reset_counter:out std_logic;
36     control_reset_reg: out std_logic;
37     --segnale di controllo che permette di cambiare il segnale in
38     --ingresso scan_in allo shift register
39     --0 come scan_in si ha l'ingresso di dato
40     --1 come scan_in si ha la catena chiusa
        control_mux:out std_logic;
        --segnale di controllo che dice quando è finita l'operazione
        --richiesta
        stop: out std_logic;
        --abilitazione registro che contiene il valore del count_max

```

```

41     control_enable_count_max_reg: out std_logic
42   );
43
44 end ControlUnit;
45
46 architecture Behavioral of ControlUnit is
47
48 type state is (idle , One_Shift , NormalMode, init ,ControlZero ,
49   initShift ,shiftModeCicle);
50   signal current , next_state: state := idle ;
51
52 begin
53   process(clock ,reset_n) begin
54     if(reset_n = '0' ) then
55       current<=idle ;
56     elsif( rising_edge(clock )) then
57       current<= next_state ;
58     end if ;
59   end process ;
60
61   process(current , start ,Handle_zero , mode ,finished , reset_n) begin
62     control_enable_count_max_reg<='0';
63     stop <='0';
64
65     control_enable_conter<= '0';
66     control_enable_reg <='0';
67
68     control_reset_counter<='1';
69     control_reset_reg <='1';
70
71     control_mode<='0';
72     control_mux<='0';
73
74     case current is
75       when idle =>
76         stop<= '1';
77         if(start='1') then
78           next_state <= init ;
79         else
80           next_state <= idle ;
81         end if ;
82
83       when init =>
84         if(mode = '0') then
85           next_state <= NormalMode;

```

```

86         next_state <= initShift ;
87     end if ;
88 when initShift =>
89     —resetto il contatore
90     control_reset_counter <='0';
91     —abilita un registro che dovrà contenere il valore massimo di
92     —conteggi
93     —a leggere il valore massimo di conteggi definito dai segnali di
94     —ingresso
95     —per poi presentare tale valore come ingresso del contatore
96     control_enable_count_max_reg <='1';
97     next_state <= One_Shift ;
98 when NormalMode =>
99     —modalità Normal
100    control_mode<= '0';
101    —abilito il registro a leggere di Shift
102    control_enable_reg <= '1';
103    next_state<=idle ;
104
105 when One_Shift =>
106     —modalità Shift in questo caso viene utilizzato come scan_in l'
107     —ingresso di dato
108     control_mode<= '1';
109     —abilito il registro di Shift a leggere
110     control_enable_reg <= '1';
111     —abilito il registro di Shift a fare un solo conteggio
112     control_enable_conter<= '1';
113     — next_state <= shiftMode ;
114
115
116 when ControlZero =>
117     if(handle_zero='1') then
118         next_state <= idle ;
119     else
120         next_state <= shiftModeCicle ;
121     end if ;
122
123
124 when shiftModeCicle =>
125     —scollego lo scan_in dello shift register dall 'ingresso
126     —di dato e lo corto circuito allo Scan_out realizzando
127     —una catena chiusa
128     control_mux<= '1';

```

```

129      —modalità di shift
130      control_mode<= '1';
131      —registro in modalità abilitata
132      control_enable_reg <= '1';
133      —abilito il contatore a fare un conteggio
134      control_enable_conter<= '1';

135
136      if(finished='1') then
137          next_state <= idle;
138      else
139          next_state <= shiftModeCicle;
140      end if;
141      end case;
142  end process;

143
144
145 end Behavioral;

```

Listing 5.5: Control Unit

5.1.3.4 Scan Chain

Nel Lis. 5.6 è mostrato principialmente solo la definizione dell'architecture (non sono mostrate le dichiarazioni dei componenti essendo ridondanti in quanto le loro interfacce già sono state presentate precedentemente)

```

1  signal reset_counter: std_logic;
2  signal reset_register:std_logic; Handle_zero_temp,scan_out_temp,
3      finished_temp,res_out_temp,control_mode_temp,
4      control_enable_conter_temp,control_enable_reg_temp,
5      control_reset_counter_temp,control_reset_reg_temp,control_mux_temp :
6      std_logic;
7  signal reset_reg: std_logic;
8  signal enable_temp: std_logic;
9  signal Count_max_temp: std_logic_vector(integer(Floor(log2(real(N-1)))
10    )) downto 0);

begin
  reset_counter <= reset_n and control_reset_counter_temp;
11  reset_register <= reset_n and control_reset_reg_temp;
12  countMaxReg: registro
13  generic map (N=>integer(Floor(log2(real(N-1))))+1)
14  port map(
15    valore_in=>posizione,
16    reset_n=>reset_n,
17    enable=>enable_temp,
    clk=>clock,

```

```

18     valore_out=>Count_max_temp
19   );
20
21 cu: ControlUnit
22 port map(
23   --ingressi di sincronismo
24   clock=>clock,
25   reset_n=>reset_n,
26   --segnali di stato provenienti dalla parte operativa
27   finished=>finished_temp,
28   Handle_zero=>handle_zero_temp,
29   --ingressi di controllo
30   start=>start,
31   mode=>mode,
32   --segnali di controllo
33   control_mode=>control_mode_temp,
34   control_enable_conter=>control_enable_conter_temp,
35   control_enable_reg=>control_enable_reg_temp,
36   control_reset_counter=>control_reset_counter_temp,
37   control_reset_reg=>control_reset_reg_temp,
38   control_mux=>control_mux_temp,
39   stop=>stop,
40   --abilitazione registro che contiene il valore del count_max
41   control_enable_count_max_reg=>enable_temp
42
43 );
44
45 sr : Shift_register
46 generic map( N=>N, shift_right_left=>'0')
47 port map(
48   mux_scan_in=>control_mux_temp,
49   reg_in=>reg_in,
50   scan_in_dato=>scan_in,
51   scan_in_controllo=>scan_out_temp,
52   scan_enable=>control_mode_temp,
53   clk=>clock,
54   reset_n=>reset_register,
55   enable=>control_enable_reg_temp,
56   Q=>Q,
57   scan_out=>scan_out_temp
58 );
59
60 contatore: Counter
61 generic map(N=>N)
62 port map(
63   clock=>clock,

```

```

64      reset_n=>reset_counter ,
65      enable=>control_enable_conter_temp ,
66      — se il contatore è mod N vuol dire che può contare fino a 8
       quindi ho bisogno di log2(N+1) bit
67      count_max=>Count_max_temp ,
68      res_out=>finished_temp ,
69      Handle_zero=>Handle_zero_temp
70      );
71 end Structural ;

```

Listing 5.6: Architecture dell'entity Scan Chain

5.1.3.5 Simulazioni Post Route

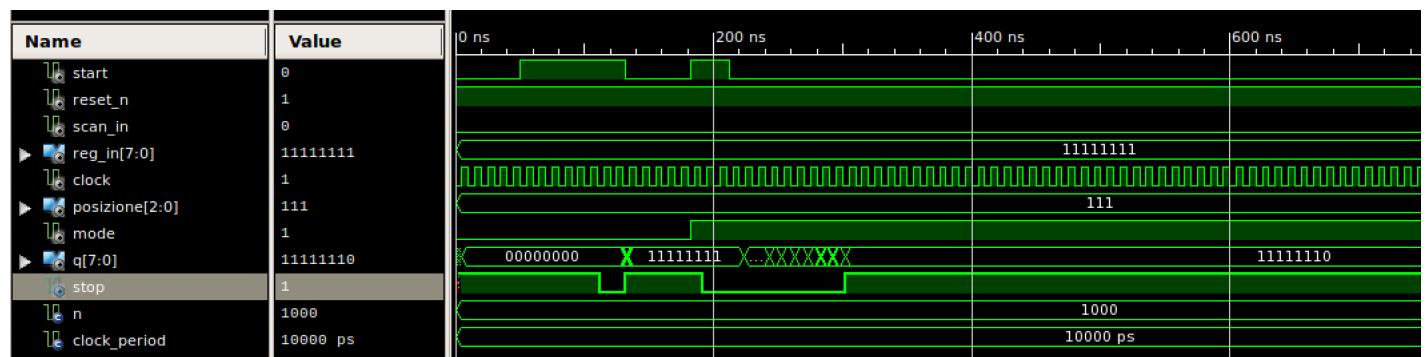


Figura 5.3: Esempio di Simulazione Post Route

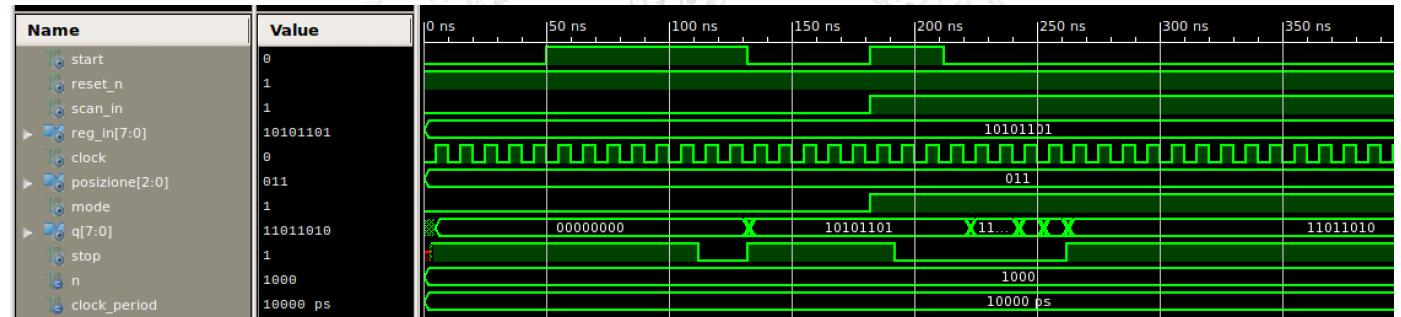


Figura 5.4: Esempio di Simulazione Post Route

Capitolo 6

Finite State Machine

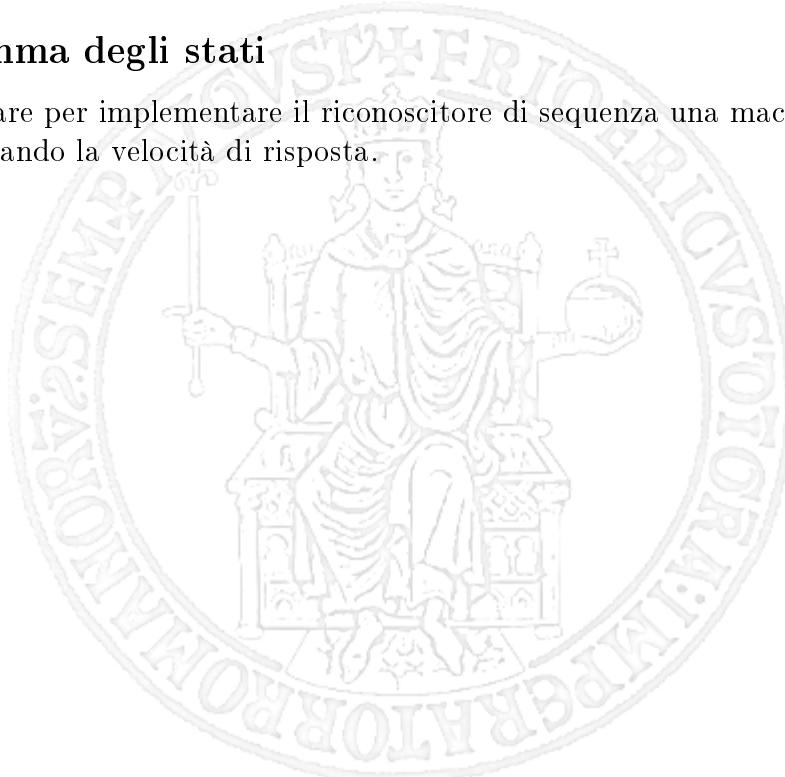
6.1 Finite State Machine

6.1.1 Traccia

1. Realizzare un riconoscitore di una generica sequenza a N bit (e.g. 1011001) in VHDL utilizzando dapprima la descrizione behavioral a singolo o doppio processo.
2. A partire dal riconoscitore di sequenza realizzato al punto 1 con i costrutti behavioral a singolo o doppio processo del VHDL, sintetizzare la macchina specificando al tool di sintesi Xilinx ISE diverse codic平 per gli stati. Per quelle rilevanti estrapolare la codice assegnata dal sintetizzatore, area occupata e frequenza massima di lavoro, apportando eventuali commenti.

6.1.2 Diagramma degli stati

Si è scelto di utilizzare per implementare il riconoscitore di sequenza una macchina sequenziale di tipo Mealy privilegiando la velocità di risposta.



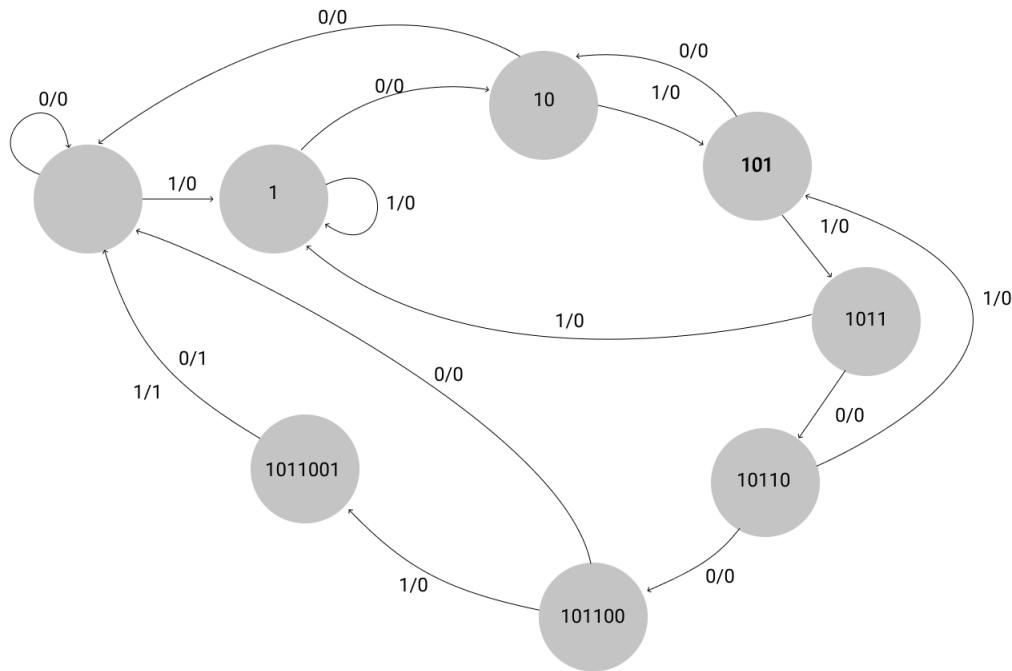
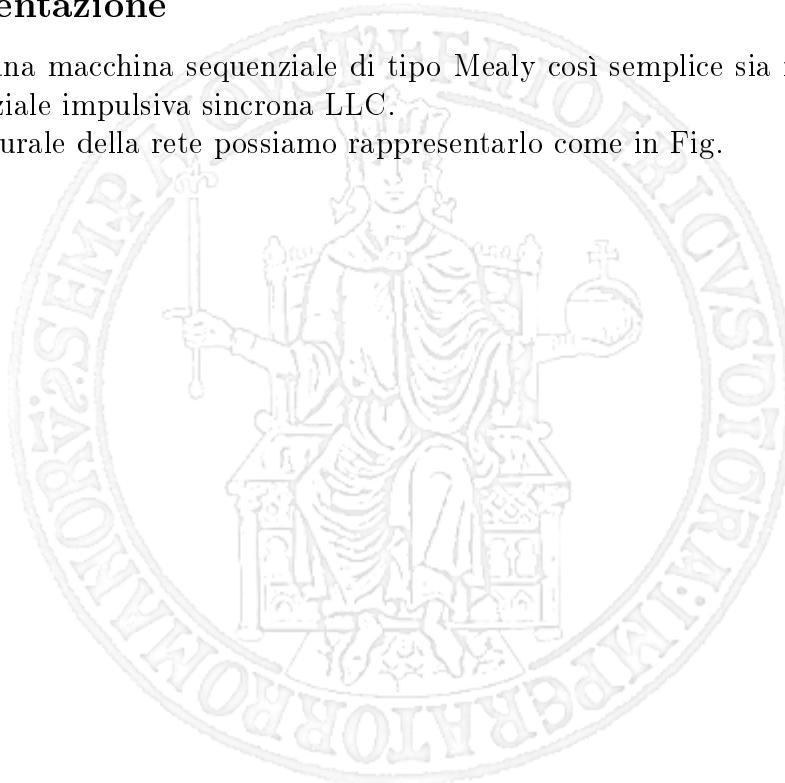


Figura 6.1: Diagramma degli stati secondo un modello sequenziale di tipo Mealy del riconositore di sequenza “1011001”

6.1.3 Implementazione

Ci aspettiamo che una macchina sequenziale di tipo Mealy così semplice sia realizzata come una classica rete sequenziale impulsiva sincrona LLC.

Il modello strutturale della rete possiamo rappresentarlo come in Fig.



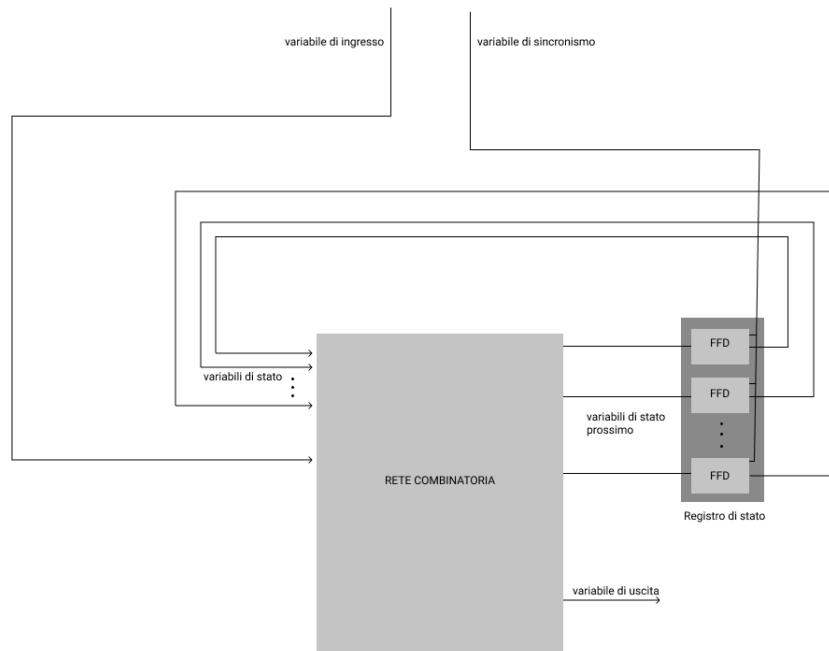


Figura 6.2: Rappresentazione strutturale di una rete sequenziale sincrona impulsiva

6.1.3.1 Codice VHDL

Di seguito è riportato il codice VHDL che descrive tramite la descrizione behavioral a doppio processo il modello della rete sequenziale.

```

1 library IEEE;
2 use IEEE.STD_LOGIC_1164.ALL;
3
4 -- Uncomment the following library declaration if using
5 -- arithmetic functions with Signed or Unsigned values
6 --use IEEE.NUMERIC_STD.ALL;
7
8 -- Uncomment the following library declaration if instantiating
9 -- any Xilinx primitives in this code.
10--library UNISIM;
11--use UNISIM.VComponents.all;
12
13 entity FSM is
14     port(
15         i: in std_logic;
16         clk: in std_logic;
17         reset_n : in std_logic;
18         Q: out std_logic
19     );
20 end FSM;
21

```

```

22 | architecture Behavioral of FSM is
23 | --1011001
24 | type state is ( idle , stat01 , stat010 , stat0101 , stat01011 , stat010110 ,
25 |           stat0101100 , stat01011001 );
26 | signal current , next_state: state := idle ;
begin
27 |
28 | NS_TO_CS: process ( clk , reset_n ) begin
29 |   if( reset_n = '0' ) then
30 |     current <= idle ;
31 |   elsif(rising_edge(clk)) then
32 |     current<= next_state ;
33 |   end if ;
34 | end process ;
35 |
36 | Combinatorial: process( current , i ) begin
37 | case current is
38 |   when idle=>
39 |     if( i='0')then
40 |       next_state<= idle ;
41 |       Q<='0';
42 |     else
43 |       next_state<=stat01 ;
44 |       Q<='0';
45 |     end if ;
46 |   when stat01=>
47 |     if( i='0')then
48 |       next_state<= stat010 ;
49 |       Q<='0';
50 |     else
51 |       next_state<=stat01 ;
52 |       Q<='0';
53 |     end if ;
54 |
55 |   when stat010 =>
56 |     if( i='0')then
57 |       next_state<= idle ;
58 |       Q<='0';
59 |     else
60 |       next_state<=stat0101 ;
61 |       Q<='0';
62 |     end if ;
63 |
64 |   when stat0101 =>
65 |     if( i='0')then
66 |       next_state<= stat010 ;

```

```

67      Q<='0';
68      else
69          next_state<=stato1011;
70          Q<='0';
71      end if;
72
73  when stato1011 =>
74      if ( i='0') then
75          next_state<= stato10110 ;
76          Q<='0';
77      else
78          next_state<=stato1 ;
79          Q<='0';
80      end if;
81
82  when stato10110 =>
83      if ( i='0') then
84          next_state<= stato101100 ;
85          Q<='0';
86      else
87          next_state<=stato101 ;
88          Q<='0';
89      end if;
90
91  when stato101100 =>
92      if ( i='0') then
93          next_state<= idle ;
94          Q<='0';
95      else
96          next_state<=stato1011001 ;
97          Q<='0';
98      end if;
99
100 when stato1011001 =>
101     if ( i='0') then
102         next_state<= idle ;
103         Q<='1';
104     else
105         next_state<=idle ;
106         Q<='1';
107     end if;
108 end case;
109
110 end process;
111
112

```

```

113
114
115 end Behavioral;

```

Listing 6.1: Implementazione di un CSL

6.1.3.2 Codifica degli stati e valutazioni

Sequential In questo tipo di codifica si utilizza il minor numero di bit per codificare gli N stati della FSM rappresentandoli in modo sequenziale.

In Fig. è mostrata la codifica con la rispettiva temporizzazione.

L'intera rete combinatoria può essere mappata su una LUT a 4 ingressi (l'ingresso i più le tre variabili di stato corrente).

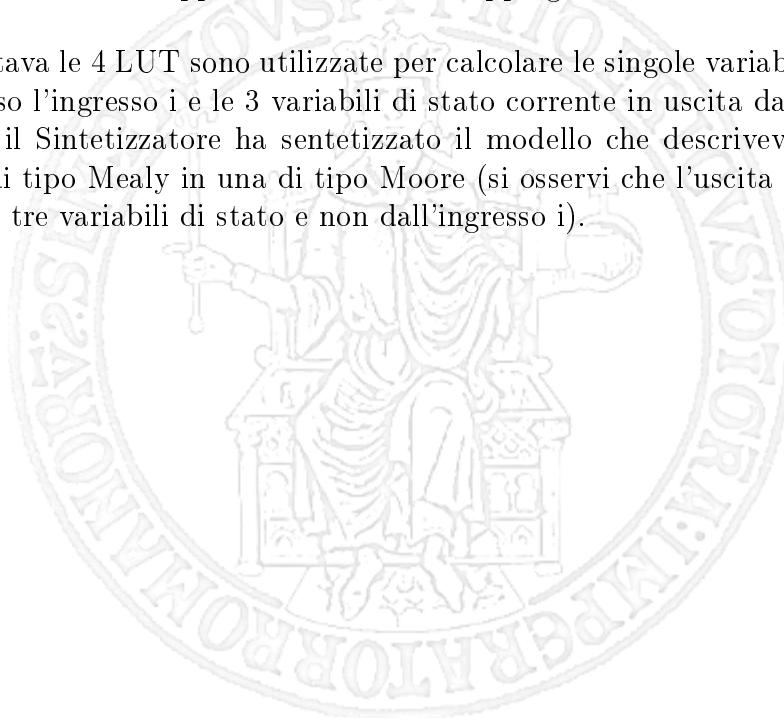
State	Encoding	Cell:in->out	fanout	Delay	Delay	Logical Name (Net Name)
idle	000					
stato1	001					
stato10	010	FDC:C->Q	4	0.591	0.666	current_FSM_FFd3 (current_FSM_FFd3)
stato101	011	LUT4:I1->0	1	0.704	0.000	current_FSM_FFd3-In1 (current_FSM_FFd3-In)
stato1011	100	FDC:D		0.308		current_FSM_FFd3
stato10110	101					
stato101100	110					
stato1011001	111	Total		2.269ns	(1.603ns logic, 0.666ns route)	

Figura 6.3: Schematic Technology con rispettivo mapping sul CLB utilizzando la Codifica di Default

Per vedere più da vicino la connessione tra le LUT e i flip flop si è eseguito lo schematico tecnologico post sintesi e anche una rappresentazione del Mapping tramite PlanAhead. (rappresentato in Fig.).

Come ci si aspettava le 4 LUT sono utilizzate per calcolare le singole variabili di stato prossimo prendendo in ingresso l'ingresso i e le 3 variabili di stato corrente in uscita dai 3 Flip FLop.

La novità è che il Sintetizzatore ha sentetizzato il modello che descriveva una macchina sequenziale sincrona di tipo Mealy in una di tipo Moore (si osservi che l'uscita è calcolata a partire esclusivamente dalle tre variabili di stato e non dall'ingresso i).



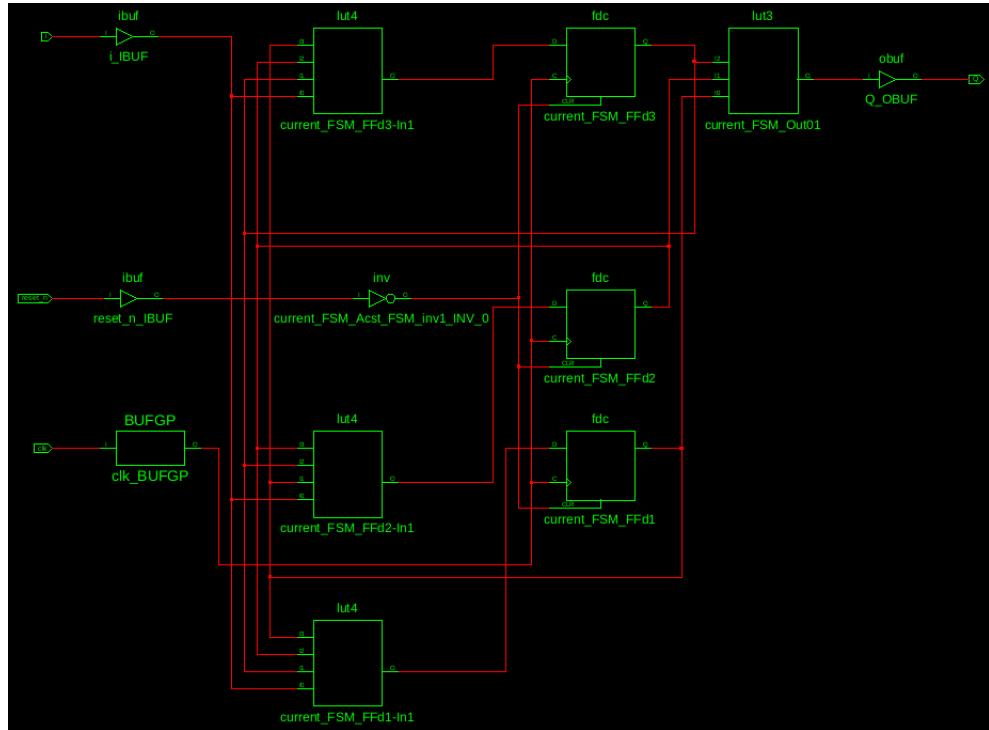


Figura 6.4: Schematic Technology utilizzando la Codifica di Default Sequential

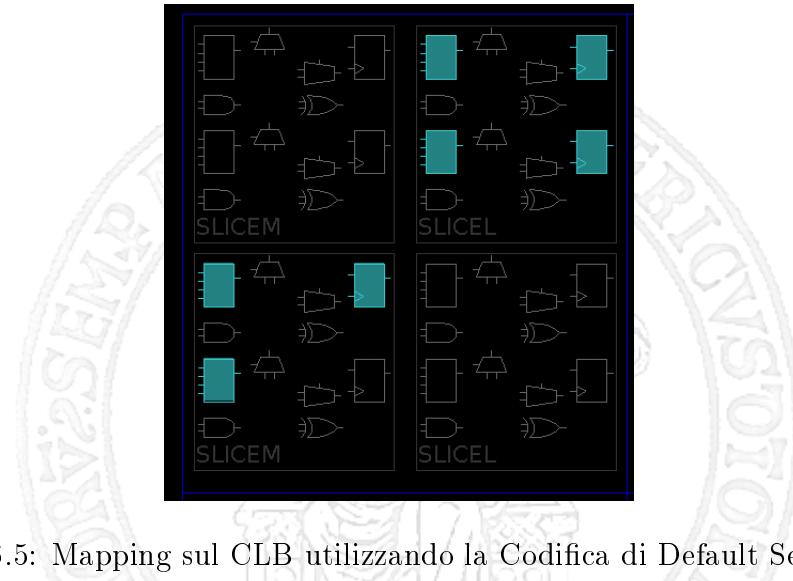


Figura 6.5: Mapping sul CLB utilizzando la Codifica di Default Sequential

One Hot State Encoding Associa ad ogni stato una codifica in cui un solo bit è alto, e quindi si avranno n variabili di stato, dove n è il numero degli stati, ovvero n flip flop.

Commutano due bit quindi durante una transizione tra due stati. One-Hot State Encoding richiede di utilizzare più FlipFlop (occupando più area) a favore, in genere, di una maggiore velocità.

State	Encoding	Cell:in->out	fanout	Delay	Delay	Logical Name (Net Name)
idle	00000001	-	-	-	-	-
state01	00000010	FDC:C->Q	2	0.591	0.622	current_FSM_FFd6 (current_FSM_FFd6)
state010	00000100	LUT2_L:I0->LO	1	0.704	0.104	current_FSM_FFd8-In_SW0 (N4)
state0101	00001000	LUT4:I3->0	1	0.704	0.000	current_FSM_FFd8-In (current_FSM_FFd8-In)
state01011	00010000	FDP:D	-	0.308	-	current_FSM_FFd8
state0101100	00100000	-	-	-	-	-
state01011001	10000000	-	-	-	-	-
Total				3.033ns	(2.307ns logic, 0.726ns route)	

Figura 6.6: Schematic Technology con rispettivo mapping sul CLB utilizzando la Codifica di Default

Questo però non è sempre vero sicuramente quando la tecnologia è FPGA. Infatti poiché la rete combinatoria deve essere mappata principalmente sulle LUT e MUX, in questo caso avranno un maggior peso ad esempio il numero degli ingressi della funzione (se si superano i 4 ingressi vi è bisogno di utilizzare il MUX aggiungenti un ulteriore piccolo ritardo).

Di seguito è mostrato sia uno Schematico Tecnologico del riconoscitore di sequenza sintetizzato con la codifica One Hot con il seguente mapping sulla FPGA.

Ogni variabile di stato con la codifica One Hot rappresenta in modo univoco uno stato e quindi il vantaggio è che una variabile di stato prossima (che rappresenta lo stato X) dipenderà esclusivamente dal valore assunto dalle variabili di stato che rappresentano quegli stati per cui con un valore dell'ingresso si transita nello stato X e ovviamente dal valore dell'ingresso.

Allora osservando il Grafo degli stati rappresentato precedentemente si noti come tutti i nodi abbiano al più solo due archi entranti e quindi con LUT a 4 ingressi sarebbe possibile rappresentare queste funzioni di variabile di stato prossima. Il problema è lo stato Idle che ha 4 archi entranti più il valore dell'ingresso.

Si osservi di conseguenza come aumentare la dimensione del registro di stato quindi non sempre corrisponde ad aumentare la frequenza del sistema, almeno per quanto riguarda la tecnologia FPGA.



Figura 6.7: Schematic Technology utilizzando la Codifica di One Hot

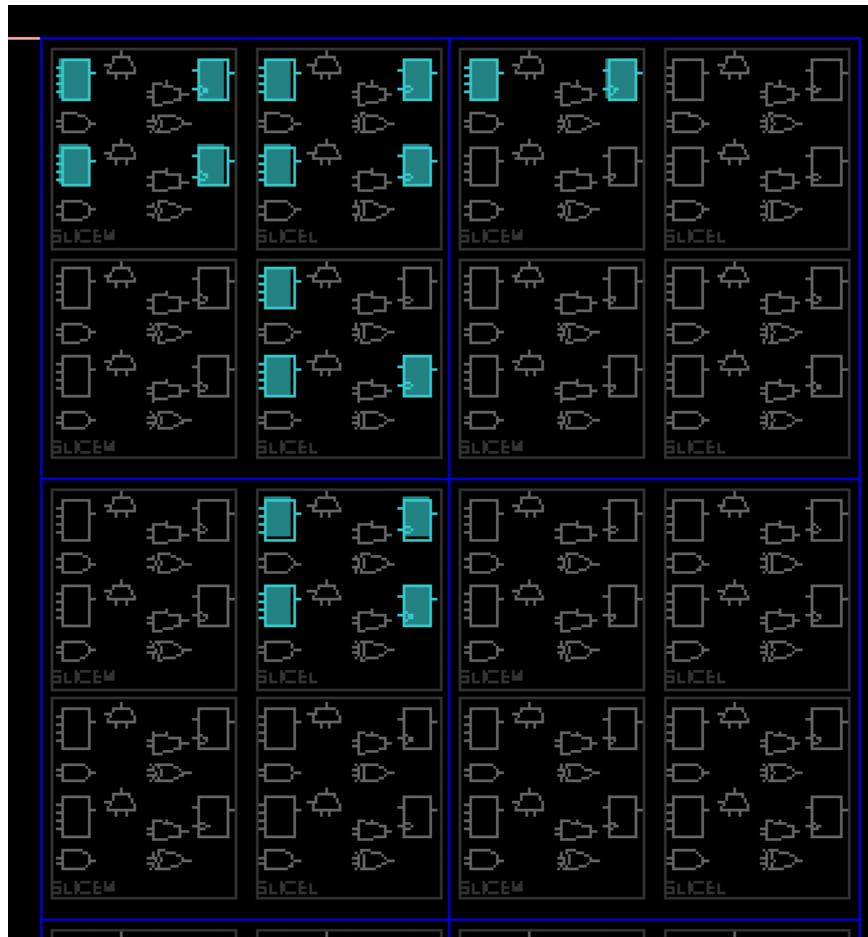


Figura 6.8: Mapping su 4 CLB utilizzando la Codifica di One Hot

Gray State Encoding Garantisce che solo un bit cambia tra due stati consecutivi riducendo al minimo i rischi e i glitch.

Qui si è scoperto che è possibile codificare gli stati adiacenti con variazione di un solo bit tramite soli 3 bit.

Ovviamente le prestazioni e le considerazioni sono le stesse che si sono fatte per la Sequential State Encoding.

A questo punto si è pensato che è più conveniente utilizzare la codifica di Gray al posto di quella Sequential.

State	Encoding	Cell:in->out	fanout	Gate Delay	Net Delay	Logical Name (Net Name)
idle	000					
stato1	001					
stato10	011	FDC:C->Q	4	0.591	0.666	current_FSM_FFd3 (current_FSM_FFd3)
stato101	010	LUT4:I1->0	1	0.704	0.000	current_FSM_FFd3-In1 (current_FSM_FFd3-In)
stato1011	110	FDC:D		0.308		current_FSM_FFd3
stato10110	111					
stato101100	101					
stato1011001	100	Total		2.269ns	(1.603ns logic, 0.666ns route)	

Figura 6.9: Schematic Technology con rispettivo mapping sul CLB utilizzando la Codifica di Default

Compact State Encoding Consiste nel ridurre al minimo il numero di bit per le variabili di stato con lo scopo di ottimizzare l'area.

Anche qui (nel caso particolare del problema trattato) si ottengono delle prestazioni analoghe a quelle ottenute con la Sequential.

Jhonson Come quella di Gray mostra benefici grandi benefici con FSM contenenti lunghi percorsi senza ramificazioni.

La differenza tra Gray e Jhonson sta nel come ricavarsi una codifica degli stati adiacenti con variazione di un solo bit.

In questo caso quella di Jhonson non conviene in quanto sia l'area che la velocità peggiorano come mostrato in Fig.

State	Encoding	Cell:in->out	fanout	Gate Delay	Net Delay	Logical Name (Net Name)
idle	0000					
stato1	0001	FDC:C->Q		8	0.591	current_FSM_FFd2 (current_FSM_FFd2)
stato10	0011	LUT4:I0->0		1	0.704	0.000 current_FSM_FFd1-In_F (N19)
stato101	0111	MUX5:I0->0		1	0.321	0.000 current_FSM_FFd1-In (current_FSM_FFd1-In)
stato1011	1111	FDC:D			0.308	current_FSM_FFd1
stato10110	1110					
stato101100	1100					
stato1011001	1000					
Total						2.856ns (1.924ns logic, 0.932ns route)

Figura 6.10: Schematic Technology con rispettivo mapping sul CLB utilizzando la Codifica di Default

Speed1 Ha lo scopo di ottimizzare la velocità. Il numero di bits per uno registro di stato dipende dalla particolare FSM, ma generalmente è più grande del numero degli stati.

Anche in questo caso abbiamo sia un peggioramento di area che di velocità.

State	Encoding	Cell:in->out	fanout	Gate Delay	Net Delay	Logical Name (Net Name)
idle	100000100					
stato1	100000010	FDP:C->Q		2	0.591	0.622 current_FSM_FFd7 (current_FSM_FFd7)
stato10	000000101	LUT3:I0->0		1	0.704	0.455 current_FSM_FFd1-In5 (current_FSM_FFd1-In5)
stato101	010000010	LUT4:I2->0		1	0.704	0.000 current_FSM_FFd1-In21 (current_FSM_FFd1-In)
stato1011	101000000	FDP:D			0.308	current_FSM_FFd1
stato10110	000100001					
stato101100	000010100					
stato1011001	000001000					
Total						3.384ns (2.307ns logic, 1.077ns route)

Figura 6.11: Schematic Technology con rispettivo mapping sul CLB utilizzando la Codifica di Default

6.1.3.3 Simulazione Behavioral

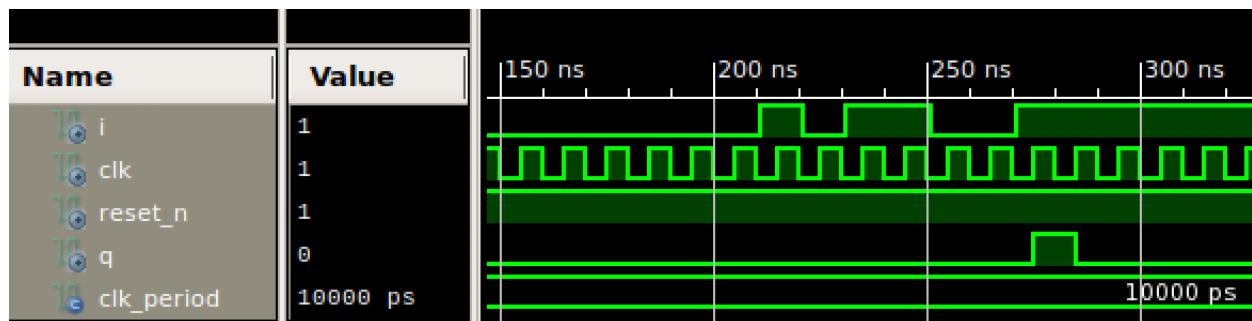


Figura 6.12: Schematic Technology con rispettivo mapping sul CLB utilizzando la Codifica di Default



Capitolo 7

Ripple Carry e Carry Look Ahead

7.1 Ripple Carry Adder/Carry Look Ahead

7.1.1 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 tempo di calcolo al variare di N e commentare il risultato con le formule teoriche.
- 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.

7.1.2 Esercizio

Per rispondere agli esercizi riguardanti il Ripple Carry Adder ed il Carry Look Ahead si è volutamente deciso di implementare un'unica struttura con la versatilità di entrambi gli addizionatori. Si è giunti a questa conclusione ipotizzando di mantenere per il Carry Look Ahead la possibilità di decidere, su N bit per cifra in ingresso, gli M di cui effettuare il calcolo a priori delle funzioni di generazione e propagazione. Conseguenza immediata di questa decisione è che i restanti N-M bit verranno inseriti in una struttura di tipo Ripple Carry. Con queste premesse, quindi, possono presentarsi essenzialmente 3 casi:

M=0) In questo caso si ha un semplice Ripple Carry Adder, tutti i bit della somma vengono calcolati propagando il riporto dopo ogni somma come in fig. 7.1.

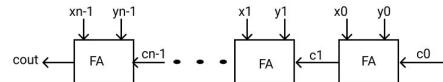


Figura 7.1: Ripple Carry Adder

M=N) Si tratta del caso opposto a M=0. Infatti con M=N tutti i riporti saranno calcolati in anticipo e la struttura risultante sarà proprio quella del Carry Look Ahead in fig. 7.2.

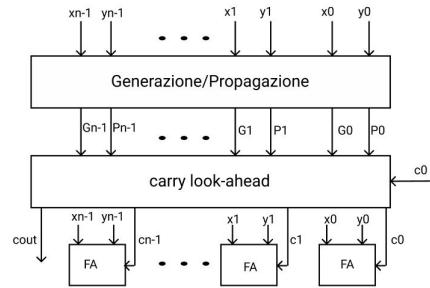


Figura 7.2: Carry Look Ahead

0<M<N) E' questo il caso in cui si decide, spesso obbligatoriamente, di usufruire di entrambe le architetture. Al crescere della dimensione degli operandi, infatti, aumenta sensibilmente anche la complessità delle funzioni di generazione e propagazione. Ciò implica che è necessario l'utilizzo di porte con sempre un maggior numero di ingressi di difficile realizzazione. Per questo motivo si pone un limite al numero di riporti che possono essere calcolati in anticipo. La struttura risultante da questa riflessione viene portata in fig. 7.3.

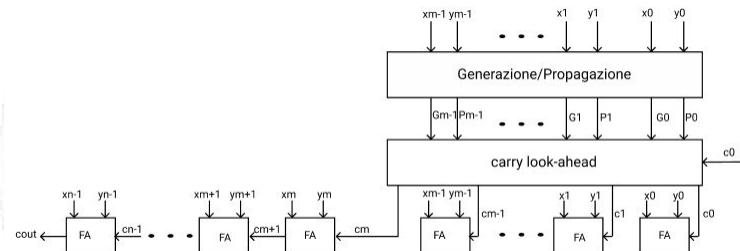


Figura 7.3: Cla e Rc

7.1.2.1 Implementazione

Si è deciso, come da traccia, di implementare per il Carry Look Ahead moduli separati per Generazione/Propagazione, CLA e gli M-1 FullAdder (il cui modulo è visibile nel Cap.1). E' stata effettuata la stessa scelta per il modulo del Ripple Carry, che istanzia N FullAdder opportunamente interconnessi.

Entrambe le architetture, con le dovute condizioni di istanziazione, sono state utilizzate nel top module. Osservando il List. 7.3 è possibile notare come per modificare il numero di bit da assegnare al Cla o al Rc basti modificare il numero N rispetto ad SP(che rappresenta il numero totale di bit nelle stringhe). Per creare corrispondenze tra il codice e la trattazione precedente si fa notare che, rispettivamente, ad SP corrisponde N ed a N corrisponde M.

```

1 library IEEE;
2 use IEEE.STD_LOGIC_1164.ALL;
3
4 entity Sommatore is
5 generic (N: integer := 5;
6          SP: integer := 8
7          );
8 port (
9    x : in STD_LOGIC_VECTOR(SP-1 downto 0);
10   y: in STD_LOGIC_VECTOR(SP-1 downto 0);
11   cin : in STD_LOGIC;    cout: out STD_LOGIC;
12   s : out STD_LOGIC_VECTOR(SP-1 downto 0) );
13 end Sommatore;
14
15
16 architecture Structural of Sommatore is
17
18 component CLA
19 generic(N: integer );
20 port (
21   p: in STD_LOGIC_VECTOR(N-1 downto 0);
22   g : in STD_LOGIC_VECTOR(N-1 downto 0);
23   cin : in STD_LOGIC ;
24   c : out STD_LOGIC_VECTOR(N downto 0)
25 );
26 end component;
27
28 component GenerazionePropagazione
29 generic (N: integer := 4);
30 port (
31   x : in STD_LOGIC_VECTOR(N-1 downto 0);
32   y: in STD_LOGIC_VECTOR(N-1 downto 0);
33   p : out STD_LOGIC_VECTOR(N-1 downto 0);
34   g : out STD_LOGIC_VECTOR(N-1 downto 0)
35 );
36 end component;
37
38 component full_adder
39 Port (
40   x : in STD_LOGIC;
```

```

41  y : in STD_LOGIC;
42  ci : in STD_LOGIC;
43  s : out STD_LOGIC;
44  co : out STD_LOGIC);
45 end component;

46

47 component ripple_carry
48 generic (N: integer:= 8);
49 Port (
50  op1 : in STD_LOGIC_VECTOR(N-1 downto 0);
51  op2 : in STD_LOGIC_VECTOR(N-1 downto 0);
52  ci : in STD_LOGIC;
53  co : out STD_LOGIC;
54  s : out STD_LOGIC_VECTOR(N-1 downto 0));
55 end component;

56

57 signal p_signal : STD_LOGIC_VECTOR(N-1 downto 0); —condizioni di
      propagazione
58 signal g_signal : STD_LOGIC_VECTOR(N-1 downto 0); —condizioni di
      generazione
59 signal c_signal : STD_LOGIC_VECTOR(N downto 0); —riporti calcolati del
      blocco Carry Look-Ahead
60 signal co_signal: STD_LOGIC_VECTOR(N-1 downto 0); —riporti in uscita
      dai full adder (non utilizzati)

61
62 signal s_cla: STD_LOGIC_VECTOR(N-1 downto 0);
63 signal s_rc: STD_LOGIC_VECTOR(SP-N-1 downto 0);

64 begin

65 — caso in cui vi è solo CLA
66 cla_gen : if N = SP generate
67   GP_instance: GenerazionePropagazione generic map(N => N)
68   port map ( x => x(N-1 downto 0) , y => y(N-1 downto 0) , p => p_signal ,
69             g=> g_signal);

70   CLA_instance: CLA generic map(N=>N)
71   port map(p=>p_signal , g=>g_signal , cin => cin , c =>c_signal);

72   fa_instances: for i in 0 to N-1 generate
73     pm_fa: full_adder port map(x => x(i) , y=> y(i) , ci=>c_signal(i) , co
74       => co_signal(i) , s=>s_cla(i));
75   end generate;

76   cout <= co_signal(N-1);
77 end generate cla_gen;

```

```

81
82 — caso in cui vi è solo RC
83 rc_gen : if N = 0 generate
84   rc_instance: ripple_carry generic map(N => SP-N)
85   port map ( op1 => x(SP-1 downto N) , op2 => y(SP-1 downto N) , ci =>
86             cin , co=> cout , s => s_rc ) ; —cout da mettere fuori
87 end generate rc_gen ;
88
89 — caso in cui si utilizzano sia CLA che RC
90 cla_rc_gen : if N /= 0 and N /= SP generate
91   GP_instance: GenerazionePropagazione generic map(N => N)
92   port map ( x => x(N-1 downto 0) , y => y(N-1 downto 0) , p => p_signal ,
93             g=> g_signal );
94
95 CLA_instance: CLA generic map(N=>N)
96 port map(p=>p_signal , g=>g_signal , cin => cin , c =>c_signal );
97
98 fa_instances: for i in 0 to N-1 generate
99   pm_fa: full_adder port map(x => x(i) , y=> y(i) , ci=>c_signal(i) , co
100            => co_signal(i) , s=>s_cla(i));
101 end generate ;
102
103 rc_instance: ripple_carry generic map(N => SP-N)
104 port map ( op1 => x(SP-1 downto N) , op2 => y(SP-1 downto N) , ci =>
105             co_signal(N-1) , co=> cout , s => s_rc );
106
107 end generate cla_rc_gen ;
108
109 s <= s_rc & s_cla ;
110
111 end Structural ;

```

Listato 7.1: Adder Cla e Rc

7.1.2.2 Caratteristiche

Ci si è focalizzati, come da traccia, sulle caratteristiche dei sommatori in termini di area occupata e tempi di calcolo cercando un riscontro dalle simulazioni.

In particolare, dai calcoli teorici:

- Nel Ripple Carry ci si aspetta un area di $5*N$ dove N indica il numero di FA utilizzati e 5 è il numero di porte utilizzate per formarlo (1 XOR a 3 ingressi per la somma, 3 AND + 1 OR per il calcolo del riporto). Ci si aspetta inoltre un ritardo di $2\Delta N$ con Δ ritardo della generica porta logica (2Δ per FA * N FA in serie).
- Nel Carry Look Ahead invece ci si aspetta un area molto maggiore al crescere di N e infatti è possibile dimostrare che sia pari a $(N^2+9N)/2$. Al contrario, il maggior vantaggio di questa

N	# of 4 input LUTs
4	8
8	16
16	32
32	64

N (M=0) RC	Max comb. path delay	N, M	Max comb. path delay	N=M CLA	Max comb. path delay
4	9.794 ns	4,3	9.934 ns	4	9.934 ns
8	14.846 ns	8,5	14.818 ns	8	14.818 ns
16	24.686 ns	16,10	24.306 ns	16	24.166 ns
32	44.366 ns	32,20	43.362 ns	32	43.142 ns

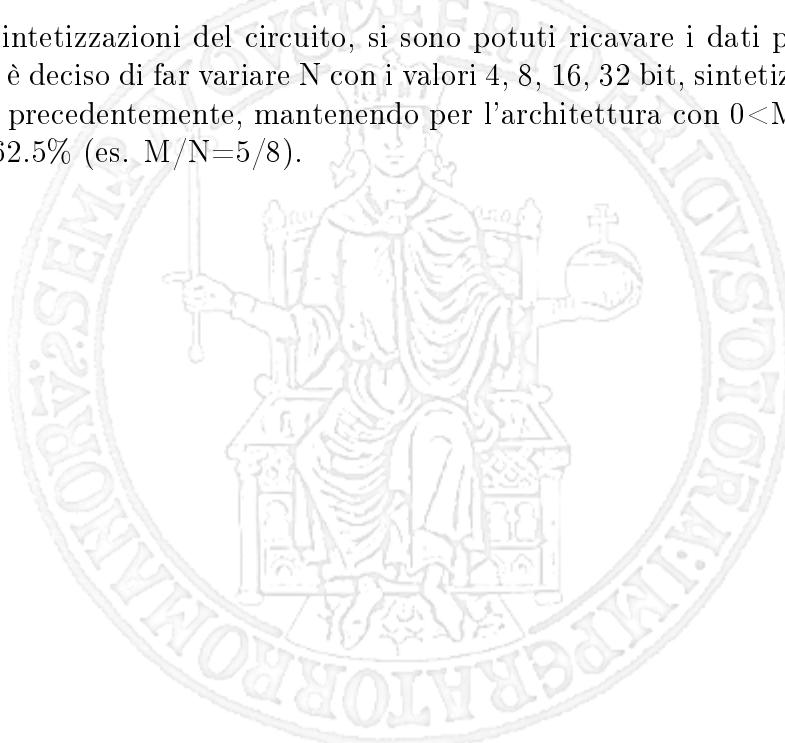
Tabella 7.1: Ritardi

CLA	Min period	Max Frequency	RC	Min period	Max Frequency
4	4.223 ns	236.798 MHz	4	4.731 ns	211.385 MHz
8	5.600 ns	178.571 MHz	8	5.987 ns	167.033 MHz
16	7.212 ns	138.658 MHz	16	7.310 ns	136.790 MHz
32	8.291 ns	120.613 MHz	32	8.750 ns	97.890 MHz

Tabella 7.2: Minimum Period e Maximum Frequency

architettura rispetto al Ripple Carry è il ritardo. Infatti il ritardo è di 5Δ (Δ per il calcolo di P e G, 2Δ per il calcolo dei riporti ed infine 2Δ del FA) e questo è dovuto al fatto che i FA lavorano in parallelo e non in serie come nel caso precedente.

Procedendo con le sintetizzazioni del circuito, si sono potuti ricavare i dati presenti nelle tabelle 9.1. In particolare si è deciso di far variare N con i valori 4, 8, 16, 32 bit, sintetizzando le 3 strutture differenti presentate precedentemente, mantenendo per l'architettura con $0 < M < N$ un rapporto di M rispetto a N del 62.5% (es. $M/N=5/8$).



Capitolo 8

Carry Save

8.1 Carry save

8.1.1 Traccia

Realizzare un esempio di addizionatore basato sulla modalità Carry Save.

8.1.2 Soluzione

Si è scelto di implementare il Carry Save rendendolo generico rispetto al numero degli operandi e al numero di bit degli addendi al fine di analizzare al crescere degli operandi il tempo di risposta.

Di seguito è riportata solo l'implementazione del CSL_TREE (Fig.8.1) che prenderà in ingresso m addendi a n bit e restituirà in uscita solo due addendi riducendo la complessità della somma da m addendi a 2. Le due stringhe in uscita dovranno essere date come ingressi a un sommatore a due addendi, preferibilmente veloce, per ottenere il risultato finale.

Sempre in Fig.8.1 è riportato un esempio di CSL_TREE.

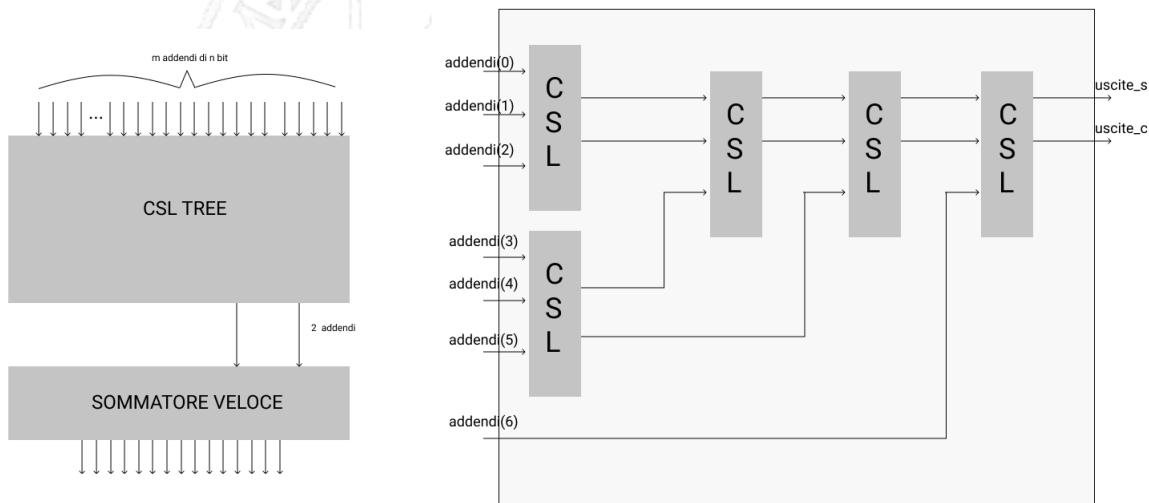


Figura 8.1: Il carry save visto come una struttura composta da un CSL_TREE e un sommatore veloce

8.1.3 Implementazione

Al fine di implementare una soluzione che fosse scalabile rispetto al numero di operandi ci si è accontentati di una soluzione di poco sub-ottima, in particolare rispetto all'area, che fosse però di natura più omogenea e ricorsiva.

8.1.3.1 Componente CSL

Si è definito in vhdl prima una entity CSL che ha come ingressi 3 addendi di n bit e presenta in uscita due nuove stringhe di $n+1$ bit.

Il componente CSL implementa una somma per colonne riportando in uscita con il giusto posizionamento, su due stringhe distinte, il bit di resto e di somma di ciascuna colonna.

Un esempio del funzionamento del CSL è possibile vederlo in Fig. 8.2.

In 8.1 è riportata la dichiarazione e la definizione dell'entity CSL tramite un'architecture strutturale.

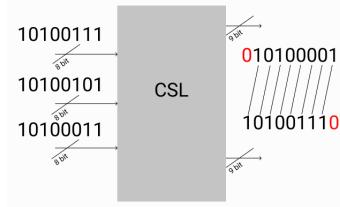


Figura 8.2: Esempio di funzionamento del componente CSL

```

1  entity CSL is
2      generic( nbit: integer := 3);
3      port(
4          ingresso_uno: in std_logic_vector(nbit-1 downto 0);
5          ingresso_due: in std_logic_vector(nbit-1 downto 0);
6          ingresso_tre: in std_logic_vector(nbit-1 downto 0);
7          uscite_s: out std_logic_vector(nbit downto 0);
8          uscite_c: out std_logic_vector(nbit downto 0)
9      );
10
11 end CSL;
12 architecture Structural of CSL is
13     component FULL_ADDER is
14         port(
15             x,y,c:IN std_logic;
16             si ,cp:OUT std_logic);
17         end component;
18
19 begin
20     uscite_s(nbit) <= '0';
21     uscite_c(0) <= '0';
22

```

```

23
24
25 GEN: for i in nbit-1 downto 0 generate
26   FA: FULL_ADDER port map(x=> ingresso_uno(i),y=> ingresso_due(i),c=>
27     ingresso_tre(i),si=>uscite_s(i),cp=>uscite_c(i+1));
28 end generate GEN;
end Structural;

```

Listing 8.1: Implementazione di un CSL

8.1.3.2 Componente CSL_LEVEL

L'entity CSL_Level è definita come una structural con componenti CSL. In particolare avendo in ingresso m (con m divisibile per tre) addendi di n bit essa presenta in uscita $m^*3/2$ stringhe di $n+1$ bit, utilizzando in parallelo $m/3$ componenti CSL.

Nel listato 8.2 è riportata la dichiarazione e la definizione dell'entity CSL_LEVEL.

Per rendere il numero di addendi e il numero di bit definibili quando si istanzia il componente, si è utilizzata una matrice di bit (il tipo vett_au dichiarato nel package “extentionwork” visibile nel listato 8.1) ritardando la definizione delle due dimensioni.

```

1 library IEEE;
2 use IEEE.STD_LOGIC_1164.ALL;
3 library ieee; use ieee.numeric_std.all;
4 library ieee; use ieee.math_real.all;
5 library ieee; use IEEE.std_logic_arith.all;
6 library work; use work.extensionwork.all;
7
8 entity LEVEL_CSL is
9
10 generic (
11   n_addendi: integer := 6;
12   N : integer := 8
13 );
14
15 port (
16   addendi: in vett_au((n_addendi-1) downto 0, N-1 downto 0);
17   uscite_s: out vett_au((integer(Floor(real(n_addendi/3)))-1) downto
18     0, N downto 0);
19   uscite_c: out vett_au(Integer(Floor(real(n_addendi/3)))-1 downto
20     0, N downto 0)
21 );
22
23
24 end LEVEL_CSL;

```

```

26 architecture Structural of LEVEL_CSL is
27 type vet_addendi_temp is array( (n_addendi-1) downto 0 ) of
28     std_logic_vector(N-1 downto 0);
29 signal c : vet_addendi_temp ;
30 type vet_uscite_temp is array( (integer(Floor(real(n_addendi/3)))-1)
31     downto 0) of std_logic_vector(N downto 0);
32 signal vet_uscite_c_temp:vet_uscite_temp; signal vet_uscite_s_temp
33     : vet_uscite_temp;
34
35 component CSL is
36     generic(
37         nbit: integer := 10);
38     port(
39         ingresso_uno: in std_logic_vector(nbit-1 downto 0);
40         ingresso_due: in std_logic_vector(nbit-1 downto 0);
41         ingresso_tre: in std_logic_vector(nbit-1 downto 0);
42         uscite_s: out std_logic_vector(nbit downto 0);
43         uscite_c: out std_logic_vector(nbit downto 0)
44     );
45
46 end component;
47 begin
48 process (addendi) begin
49     for i in n_addendi-1 downto 0 loop
50         for j in N-1 downto 0 loop
51             c(i)(j)<= addendi(i,j) ;
52         end loop;
53     end loop;
54 end process;
55
56 process (vet_uscite_c_temp , vet_uscite_s_temp) begin
57     for i in (integer(Floor(real(n_addendi/3)))-1) downto 0 loop
58         for j in N downto 0 loop
59             uscite_c(i,j)<=vet_uscite_c_temp(i)(j);
60             uscite_s(i,j)<=vet_uscite_s_temp(i)(j);
61         end loop;
62     end loop;
63 end process;
64
65 GEN: for i in integer(Floor(real((n_addendi/3))))-1 downto 0 generate
66     csl_instantiate: CSL
67     generic map( nbit=> N)
68     port map(
69         ingresso_uno =>c( i*3) ,
70         ingresso_due=>c( i*3+1) ,
71         uscite_s=>vet_uscite_s_temp,
72         uscite_c=>vet_uscite_c_temp
73     );
74 end;

```

```

69      ingresso_tre=>c(i*3+2),
70      uscite_s => vet_uscite_s_temp(i),
71      uscite_c => vet_uscite_c_temp(i)
72  );
73 end generate GEN;
74
75 end Structural;

```

Listing 8.2: Implementazione di un CSL_LEVEL

```

1 library IEEE; use IEEE.STD_LOGIC_1164.ALL; use ieee.math_real.all;
2 use IEEE.std_logic_arith.all;
3 package extentionwork is
4
5     type vett_au is array (integer range <>, integer range <>) of
6         std_logic;
7     Function nome_funzione ( n_addendi : in integer;      n_bit : in
8         integer) return integer ;
9 end extentionwork;
10 package body extentionwork is
11     Function nome_funzione ( n_addendi : in integer;      n_bit : in
12         integer) return integer
13     is
14         variable na:integer := n_addendi;
15         variable nb:integer:= n_bit+1;
16         begin
17             while(na/-3) loop
18                 na:= Integer(Floor( real(na/3)))*2 + (na rem 3);
19                 nb:=nb+1;
20             end loop;
21             return nb;
22         end nome_funzione;
23 end extentionwork;

```

Listing 8.3: Package EXTENTIONWORK

8.1.3.3 Componente Contenitore

L'entity Contenitore ha come parametri generic il numero di addendi e il numero di bit.

E' il componente che ha il suo interno tutta la logica del CSL_TREE.

In particolare esso è composto da un CSL_TREE.

8.1.3.4 Componente CSL_TREE

Per rendere l'intero componente CSL_TREE generico rispetto al numero di operandi stante la non-omogeneità tra un livello (LEVEL_CSL) e il successivo si è utilizzata un'istanziazione ricorsiva.

Per capire l'impostazione del progetto si osservi la fig.8.3 che mostra come la struttura di fig.8.1 venga sintetizzata utilizzando l'implementazione ricorsiva sviluppata in questo elaborato.

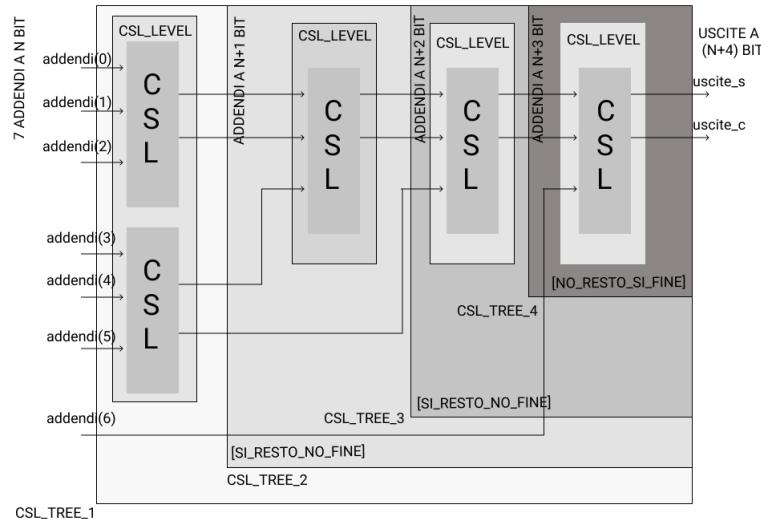


Figura 8.3: Sintesi della fig.8.1.

Implementazione del CSL_TREE Il componente Contenitore istanzia il primo CSL_TREE definendo i parametri generici di quest'ultimo, in particolare il parametro “bit_in_uscita” che verrà opportunamente calcolato da una funzione che non verrà sintetizzata, ma servirà solo per la configurazione.

Di seguito, nel List.8.4, è mostrato il codice VHDL che definisce l'entity CSL_TREE.

Si può osservare che l'istanziazione ricorsiva è ripetuta finché non si verifica la condizione “NO_RESTO_SI_FINE” nel costrutto if-generate che genererà l'ultimo CSL_LEVEL per poi presentare l'uscita di tutti i CSL_TREE e del Contenitore.

Le altre due condizioni “NO_RESTO_NO_FINE” e “SI_RESTO_SI_FINE” invece genereranno prima l'istanziazione di un CSL_LEVEL e poi di un nuovo CSL_TREE (“SI_RESTO” significa che dopo l'istanziazione del CSL_LEVEL si ottiene un numero di addendi non divisibile per tre).

La complessità del CSL_TREE contenuto sarà minore di quello che lo contiene.

```

1 library IEEE;
2 use IEEE.STD_LOGIC_1164.ALL;
3 library work; use work.extensionwork.all;
4 use ieee.math_real.all;
5 use IEEE.std_logic_arith.all;
6
7 entity CSL_TREE is
8     generic (
9         n_addendi: integer := 7 ;

```

```

10      n_addendi_uscita:integer:=1;
11      N :integer:=8;
12      bit_in_uscita:integer:=12
13      );
14  port (
15      addendi: in vett_au((n_addendi-1) downto 0, N-1 downto
16          0);
17      uscite_s: out vett_au(0 downto 0, bit_in_uscita-1 downto 0)
18          ;
19      uscite_c: out vett_au(0 downto 0, bit_in_uscita-1 downto 0)
20          );
21 end CSL_TREE;
22 architecture Structural of CSL_TREE is
23
24 component LEVEL_CSL
25     generic (
26         n_addendi:integer := 100;
27         N : integer := 5
28         );
29     port (
30         addendi: in vett_au((n_addendi-1) downto 0, N-1 downto 0);
31         uscite_s: out vett_au((integer(Floor(real(n_addendi/3)))-1)
32             downto 0, N downto 0);
33         uscite_c: out vett_au(Integer(Floor(real(n_addendi/3)))-1
34             downto 0, N downto 0)
35         );
36     end component;
37 component CSL_TREE
38     generic (
39         n_addendi:integer := 448 ;
40         n_addendi_uscita:integer:=1;
41         N :integer:=8;
42         bit_in_uscita:integer:=12
43         );
44     port (
45         addendi: in vett_au((n_addendi-1) downto 0, N-1
46             downto 0);
47         uscite_s: out vett_au(0 downto 0, bit_in_uscita-1 downto 0)
48             ;
49         uscite_c: out vett_au(0 downto 0, bit_in_uscita-1 downto 0)
50         );
51     end component;
52
53 signal temp_uscite_s, temp_uscite_c: vett_au((integer(Floor(real(
54         n_addendi/3)))-1) downto 0, N downto 0);

```

```

49 signal concatenate: vett_au((integer(Floor(real(n_addendi/3)))*2-1)
50   downto 0, N downto 0);
51 signal concatenate1: vett_au(((integer(Floor(real(n_addendi/3)))*2)+
52   n_addendi_uscita)-1 downto 0, N downto 0);
53 type vet_uscrite_temp is array( (n_addendi-1) downto 0) of
54   std_logic_vector(N downto 0);    signal vet_uscrite_c_temp:
55   vet_uscrite_temp;
56   signal vet_uscrite_s_temp: vet_uscrite_temp;
57 begin
58   NO_RESTO_SI_FINE: if ((n_addendi_uscita) = 0) and (integer(Floor(
59     real(n_addendi/3))) = 1) generate
60     livelloFinale: LEVEL_CSL
61       generic map (n_addendi=>n_addendi ,N=>N)
62       port map (addendi=>addendi , uscite_s=>temp_uscite_s ,
63         uscite_c=>temp_uscite_c
64       );
65       process (temp_uscite_c ,temp_uscite_s) begin
66         for i in (integer(Floor(real(n_addendi
67           /3)))-1) downto 0 loop
68           for j in N downto 0 loop
69             uscite_s(i,j)<=temp_uscite_s(i,j)
70             ;
71             uscite_c((integer(Floor(
72               real(n_addendi/3)))-1)+i,j)<=
73             temp_uscite_c(i,j);
74           end loop;
75         end process;
76       end generate NO_RESTO_SI_FINE;
77
78   NO_RESTO_NO_FINE: if ((n_addendi_uscita) = 0) and (integer(Floor(
79     real(n_addendi/3))) /= 1) generate
80     livelloNonFinale: LEVEL_CSL
81       generic map (n_addendi=>n_addendi , N => N)
82       port map (addendi=>addendi , uscite_s=>temp_uscite_s ,
83         uscite_c=>temp_uscite_c --addendi-uscita non ci
84         sono qui
85       );
86       process (temp_uscite_s ,temp_uscite_c) begin
87         for i in (integer(Floor(real(n_addendi/3)))
88           -1) downto 0 loop
89           for j in N downto 0 loop
90             concatenate(i,j)<=temp_uscite_s(i,j);
91             end loop;
92           end loop;
93         end process;
94       end generate NO_RESTO_NO_FINE;
95
96   end architecture Behavioral;

```



```

112      end loop;
113      for i in (integer(Floor(real(n_addendi/3)))
114          *2-1) downto (integer(Floor(real(
115              n_addendi/3)))) loop
116          for j in N downto 0 loop
117
118              concatenate1(i,j)<=temp_uscite_c(i-
119                  integer(Floor(real(n_addendi/3))),j
120                  );
121          end loop;
122      end loop;
123      for i in (((integer(Floor(real(n_addendi/3)))
124          *2)+n_addendi_uscita)-1) downto (integer(
125              (Floor(real(n_addendi/3))))*2) loop
126          for j in N-1
127              downto 0 loop
128
129              concatenate1(i,j)<=addendi(i-
130                  integer(Floor(real(n_addendi/3)))*2 +
131                  n_addendi-n_addendi_uscita),j);
132
133          end loop;
134          concatenate1(i,N)<='0';
135          end loop;
136      end process;
ricorsione1: CSL_TREE
    generic map(
        N=>N+1,
        n_addendi => (integer(Floor(real(n_addendi/3)))*2+
            n_addendi_uscita),
        n_addendi_uscita =>(integer(Floor(real(n_addendi/3)))
            )*2+n_addendi_uscita)rem 3,
        bit_in_uscita=>bit_in_uscita
    ) --
port map(
    addendi => concatenate1,
    uscite_s=>uscite_s,
    uscite_c=>uscite_c
);
end generate SI_RESTO_NO_FINE;
end Structural;

```

Listing 8.4: Implementazione di un CSL_TREE

8.1.3.5 Schematic RTL della soluzione

Di seguito sono mostrati due Schematic RTL della soluzione. Nel primo in Fig. è riportata la sintesi di un Contenitore con numero di addendi pari a 7 e numero di bit pari a 8, invece nel secondo, quello in Fig, con numero di addendi pari a 100 e numero di bit pari a 8.

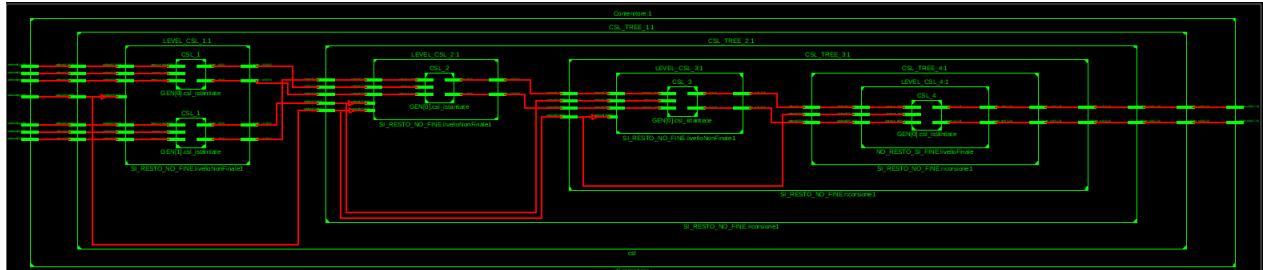


Figura 8.4: Schematic di un CSL_TREE con 7 operandi da 8 bit che presenta in uscita due operandi da 12 bit



Figura 8.5: Schematic di un CSL_TREE con 100 operandi da 8 bit che presenta in uscita due operandi da 18 bit

Capitolo 9

Carry Select

9.1 Carry Select Adder

9.1.1 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.

9.1.2 Esercizio

9.1.2.1 Funzionamento

L'addizionatore Carry Select è un dispositivo appartenente alla categoria dei "sommatori veloci". Questo infatti è appositamente progettato per migliorare i ritardi del Ripple Carry dovuti alla propagazione lineare dei riporti, a discapito di una maggiore occupazione di area. Per la trattazione sulle caratteristiche del dispositivo si riporta ai paragrafi successivi, si osservi adesso la Fig. 9.1 che rappresenta uno schematico del sommatore.

Il funzionamento del Carry Select Adder limita considerevolmente la problematica della propagazione dei riporti, prevedendo la suddivisione di un unico Ripple Carry a N bit in P blocchi di 2 RippleCarry l'uno (ciascuno a $M=N/P$ bit). In particolare, i due addizionatori all'interno del blocco avranno gli stessi M ingressi ma saranno configurati differentemente per quanto riguarda il riporto in ingresso. Così facendo un blocco avrà a disposizione i calcoli effettuati per due casi distinti, per riporto in ingresso pari a '1' oppure pari a '0'. Questo procedimento introduce sicuramente una ridondanza (dal momento che un solo risultato verrà scelto) ma aumenta sensibilmente la velocità del dispositivo in caso vi sia un numero N considerevole di bit in ingresso. Chiaramente il primo blocco avendo in ingresso un riporto certo, presenterà un solo addizionatore.

Per ogni blocco vi saranno posti due mux in uscita:

1. Il primo prende gli M bit di somma provenienti dai 2 addizionatori interni ed è pilotato dal riporto del blocco precedente. Nel caso in cui riporto di selezione sia '1' si sceglierà il risultato della somma effettuata con riporto entrante pari a '1', e viceversa.

2. Il secondo ha la funzione di propagare il riporto al blocco successivo ed è ancora una volta pilotato dal riporto precedente. Similmente al mux precedente, seleziona il riporto uscente dall'addizionatore opportuno.

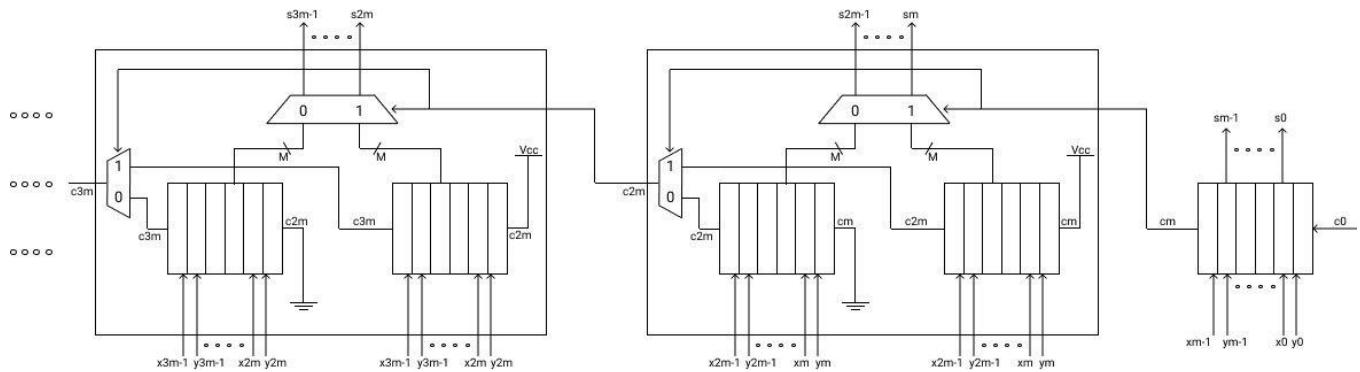


Figura 9.1: Carry Select Adder

9.1.2.2 Implementazione

Si è implementato il modulo del MUX e riutilizzato l'addizionatore Ripple Carry impiegato negli esercizi precedenti composto dal Full Adder visibile nel Capitolo 1. Si è dunque deciso di implementare un modulo “*Blocco2RC*” che identificasse il blocco (List. 9.1) composto da due Ripple Carry Adder e due Mux opportunamente interconnessi, come in Fig.9.1. Quindi si è implementato il modulo più di alto livello “*Csel_adder*” (List. 9.2) che provvede ad istanziare l’intera struttura, un primo semplice Ripple Carry e P-2 blocchi definiti precedentemente. E’ possibile scegliere i parametri per l’istanziazione tramite due generic che indicano M ed il numero di Blocchi totale. Si riportano di seguito i codici.

```

1 library ieee;
2 use ieee.std_logic_1164.all;
3 entity Blocco2RC is
4   generic (N: integer := 4);
5   port (
6     operatore1 : in STD_LOGIC_VECTOR(N-1 downto 0);
7     operatore2 : in STD_LOGIC_VECTOR(N-1 downto 0);
8     risultato : out STD_LOGIC_VECTOR(N-1 downto 0);
9     cin : in STD_LOGIC;
10    cout : out STD_LOGIC);
11 end Blocco2RC;
12
13 architecture Structural of Blocco2RC is
14
15 component ripple_carry
16   generic (N: integer);
17   port (
18     op1 : in STD_LOGIC_VECTOR(N-1 downto 0);
19     op2 : in STD_LOGIC_VECTOR(N-1 downto 0));

```

```

20      ci : in STD_LOGIC;
21      co : out STD_LOGIC;
22      s: out STD_LOGIC_VECTOR(N-1 downto 0));
23  end component;
```

```

24
25  component MUX generic (N_bit: integer); port(
26      op1 : in STD_LOGIC_VECTOR(N-1 downto 0);
27      op2 : in STD_LOGIC_VECTOR(N-1 downto 0);
28      ris : out STD_LOGIC_VECTOR(N-1 downto 0);
29      select_uscita : in STD_LOGIC);
30  end component;
```

```

31
32  signal cout_to_mux : STD_LOGIC_VECTOR(1 downto 0);
33  signal s0 : STD_LOGIC_VECTOR(N-1 downto 0);
34  signal s1 : STD_LOGIC_VECTOR(N-1 downto 0);
35
36  begin
37
38  add1_instance : ripple_carry generic map(N =>N)
39  port map (
40      op1 => operatore1(N-1 downto 0),
41      op2 => operatore2(N-1 downto 0),
42      ci => '1',
43      co=> cout_to_mux(1),
44      s => s1
45  );
46  add0_instance : ripple_carry generic map(N =>N)
47  port map (
48      op1 => operatore1(N-1 downto 0),
49      op2 => operatore2(N-1 downto 0),
50      ci => '0',
51      co=> cout_to_mux(0),
52      s=>s0
53  );
54  Mux_ris_instance : MUX generic map(N => N)
55  port map (
56      op1 => s0,
57      op2 => s1,
58      ris => risultato,
59      select_uscita => cin );
60
61  cout <= cout_to_mux(1) when cin = '1' else cout_to_mux(0); — "Mux 2
62      in -> 1 out"
63  end Structural;
```

Listato 9.1: Blocco2RC

```

1 library IEEE;
2 use IEEE.STD_LOGIC_1164.ALL;
3 —M è il numero totale di bit di uno dei due operandi , mentre P è il
4 — numero di blocchi
5 entity Csel_adder is generic (M: integer := 12;P: integer := 1);
6   port (
7     operatore1: in STD_LOGIC_VECTOR(M-1 downto 0);
8     operatore2: in STD_LOGIC_VECTOR(M-1 downto 0);
9     risultato : out STD_LOGIC_VECTOR(M-1 downto 0);
10    cin: in STD_LOGIC;
11    cout: out STD_LOGIC);
12 end Csel_adder;
13
14
15
16
17 component Blocco2RC
18   generic (N: integer);
19   port (operatore1 : in STD_LOGIC_VECTOR(N-1 downto 0);
20         operatore2 : in STD_LOGIC_VECTOR(N-1 downto 0);
21         risultato : out STD_LOGIC_VECTOR(N-1 downto 0);
22         cin : in STD_LOGIC;
23         cout : out STD_LOGIC);
24 end component;
25
26 component ripple_carry generic (N: integer);
27   port ( op1 : in STD_LOGIC_VECTOR(N-1 downto 0);
28         op2 : in STD_LOGIC_VECTOR(N-1 downto 0);
29         ci : in STD_LOGIC;
30         co : out STD_LOGIC;
31         s: out STD_LOGIC_VECTOR(N-1 downto 0)
32 );
33 end component;
34
35
36 signal segnale_cout : STD_LOGIC_VECTOR(P-1 downto 0);
37
38
39 begin
40
41 cout <= segnale_cout(Blocchi-1);

```

```

43
44 first_RC_instance : ripple_carry generic map(N => M/P)
45     port map(
46         op1 => operatore1(M/P-1 downto 0) ,
47         op2 => operatore2(M/P-1 downto 0) ,
48         s => risultato(M/P-1 downto 0) ,
49         ci => cin ,
50         co =>segnale_cout(0)) ;
51
52
53 generate_CSeA: for i in 0 to P-2 generate
54     Blocco2RC_instances : Blocco2RC generic map(N => M/P)
55         port map(
56             operatore1 => operatore1(M-(M/P*i)-1 downto M-(M/P*(i
57                 +1))) ,
58             operatore2 => operatore2(M-(M/P*i)-1 downto M-(M/P*(i
59                 +1))) ,
60             risultato => risultato (M-(M/P*i)-1 downto M-(M/P*(i
61                 +1))) ,
62             cin=>segnale_cout(P-2-i) ,
63             cout=>segnale_cout(P-1-i)) ;
64 end generate;
65
66 end Structural;

```

Listato 9.2: Csel_adder

9.1.2.3 Considerazioni su area occupata e tempo di calcolo

E' possibile studiare facilmente le caratteristiche del circuito considerando i 2 elementi base:

1. Full Adder con tempo di propagazione t_{FA}
2. Multiplexer con tempo di propagazione t_{MUX}

Con queste premesse, per studiare il ritardo complessivo è necessario effettuare una separazione tra il primo blocco ed i successivi. Infatti nel primo blocco assistiamo ad un ritardo di Mt_{FA} equivalente al ritardo del singolo addizionatore Ripple Carry, nei successivi invece è da aggiungere il ritardo del MUX che propaga il riporto al blocco successivo. Il ritardo nei blocchi intermedi sarà dunque di $Mt_{FA} + t_{MUX}$, ciò implica che il ritardo totale dell'addizionatore può essere scritto come $T = M * t_{FA} + (P - 1) * t_{MUX}$. In particolare il primo termine della somma è dovuto ai singoli Ripple Carry in ogni blocco ai quali è concesso di lavorare in parallelo, il secondo dovuto ai $P - 1$ MUX in serie dovuti ai P blocchi.

Tuttavia è lecito chiedersi quale sia la combinazione ottima per scegliere i parametri P ed M in modo da minimizzare i ritardi. Ricordandosi che $M = \frac{N}{P}$ è possibile riscrivere l'espressione precedente $T = Mt_{FA} + (P - 1)t_{MUX} = \frac{N}{P}t_{FA} + (P - 1)t_{MUX}$.

Derivando in dP si ottiene $\frac{dT}{dP} = -\frac{nt_{FA}}{P^2} + t_{MUX}$ e ponendo la derivata =0 è possibile ricavare i valori:

- $P = \sqrt{\frac{Nt_{FA}}{t_{MUX}}}$
- $M = \frac{N}{P} = \sqrt{\frac{Nt_{MUX}}{t_{FA}}}$

Questi valori sono quindi quelli che consentono di minimizzare il ritardo totale T. Sostituendo queste espressioni otteniamo $T = Mt_{FA} + (P - 1)t_{MUX} = 2\sqrt{Nt_{FA}t_{MUX}} - t_{MUX}$.

Dall'ultimo passaggio, che mette in risalto il rapporto tra il ritardo T ed il numero di bit N, possiamo notare che il primo cresce con la radice del secondo. A differenza del singolo ripple carry, per il quale ricordiamo la crescita era lineare con N, per un numero di bit molto alto possiamo avere un notevole vantaggio in termini di tempo di l'elaborazione.

Viceversa lo svantaggio principale del dispositivo si ottiene in termini di area occupata. E' possibile infatti supporre da subito, essendo i blocchi formati da 2 ripple carry per sottoinsieme di bit, che l'area occupata sia circa il doppio rispetto al semplice ripple carry. Possiamo infatti vedere l'area totale come somma di 3 contributi distinti, l'area del primo stadio composto dal singolo ripple carry (5M), l'area dei MUX e l'area dei P-1 blocchi composti da 2 addizionatori rc.

Tramite i tool di Ise è stato possibile studiare le caratteristiche del circuito, sintetizzandolo al variare dei parametri indici del numero di bit per addendo e numero di blocchi totali.

In particolare, in Tab. 9.1, sono stati effettuati diversi casi test con M(numero di bit) che varia tra 8, 16, 32 e 64 bit e P(numero di blocchi) tra 1(forma un ripple carry),2, 4, 8 e 16. Questa scelta è stata effettuata considerando la limitazione dell'implementazione che obbliga a dividere gli M bit ugualmente nei blocchi(quindi M divisibile per P).

Possiamo immaginare però che una architettura ottima del dispositivo si ottenga eliminando il vincolo che tutti i blocchi siano della stessa dimensione. Nel caso di blocchi uguali, infatti, le somme calcolate dai vari RCA al loro interno sono disponibili tutte allo stesso tempo ma i segnali di selezione dei blocchi dal secondo al p-esimo sono disponibili solamente dopo un tempo T di propagazione da uno stadio all'altro. L'idea per ottimizzare il dispositivo sarebbe quindi quella di assegnare un carico computazionale maggiore ai blocchi il cui risultato è richiesto dopo un tempo maggiore. Tuttavia ciò va oltre la trattazione che si è ritenuta opportuna, ci limiteremo dunque a sperimentare i casi precedentemente citati.

Dai dati raccolti è possibile effettuare diverse osservazioni:

- Nel caso con P=1 l'unico blocco che si istanzia è il primo, l'architettura quindi si limita ad un semplice ripple carry. I dati sono coerenti a quelli raccolti negli esercizi precedenti e come ci si aspettava il ritardo ancora una volta cresce con M.
- Sono stati individuati i casi di ottimo(evidenziati in verde). Ne risulta che generalmente non è conveniente dividere in troppi blocchi di pochi bit.
- I casi evidenziati in rosso sono i casi in cui il rapporto M/P= 2 e quindi ogni blocco avrà soltanto 2 bit di ingresso per operando. Questi casi sono i peggiori riscontrati, aggiungendo infatti i circuiti di selezione a differenza di un semplice ripple carry, risulta essere più lento di quest'ultimo.

M,P (RC)	Max comb. path delay	M,P	Max comb. path delay	M,P	Max comb. path delay
8,1	13.203 ns	8,2	12.505 ns	8,4	13.413 ns
16,1	21.690 ns	16,2	15.637 ns	16,4	16.869 ns
32,1	38.665 ns	32,2	24.048 ns	32,4	21.589 ns
64,1	72.616 ns	64,2	40.593ns	64,4	31.438 ns
M,P	Max comb. path delay	M,P	Max comb. path delay	M,P	Max comb. path delay
16,8	22.232 ns	16,16	—		
32,8	24.425 ns	32,16	39.954 ns		
64,8	33.343 ns	64,16	41.680 ns		

Tabella 9.1: Ritardi

Simili esperimenti sono stati effettuati per il calcolo delle aree. Infatti si è preso nota del numero di LUT a 4 ingressi utilizzate in fase di implementazione. I risultati riportati in Tab. 9.2 confermano che l'occupazione dell'area nel caso della suddivisione in blocchi è molto maggiore rispetto a quella nel caso di un semplice ripple carry. Si noti inoltre come ai casi ottimi in termini di ritardo evidenziati precedentemente corrispondano i casi peggiori in termini di area occupata.

M,P	4 in LUTs	M,P	4 in LUTs	M,P	4 in LUTs
8,1	16	16,1	32	32,1	64
8,2	17	16,2	44	32,2	92
8,4	16	16,4	39	32,4	99
—		16,8	32	32,8	83

Tabella 9.2: Aree

E' stata infine realizzata una architettura aggiungendo dei registri per gli ingressi e le uscite dell'addizionatore mostrata in Fig. . Così facendo è stato possibile raccogliere i dati riportati in Tab. di periodo minimo e quindi frequenza massima associati al circuito realizzato.

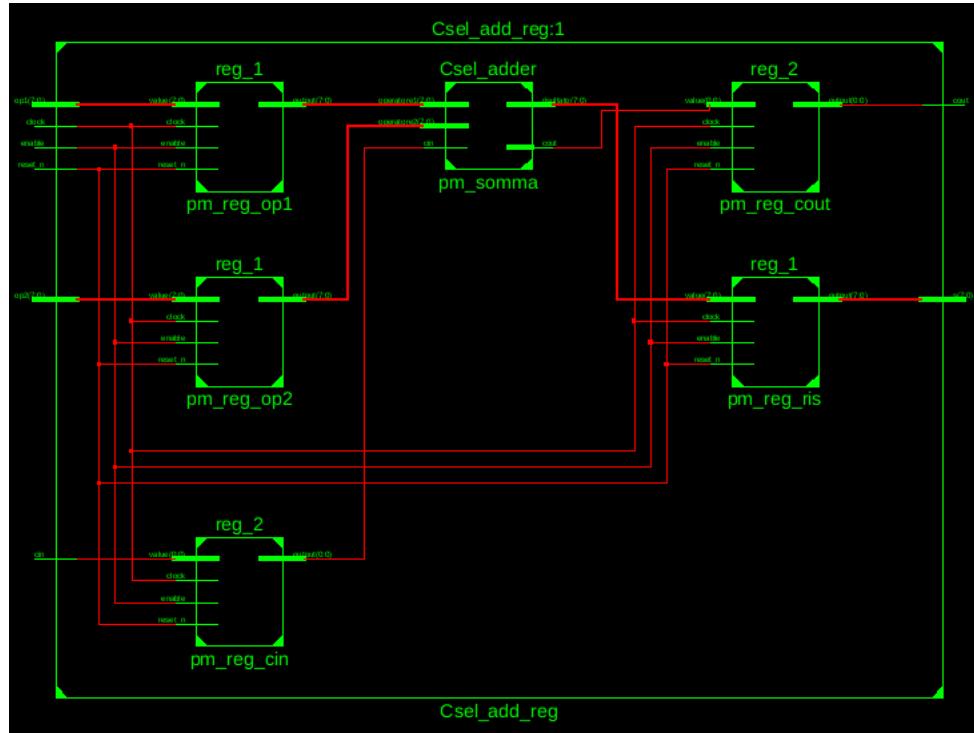


Figura 9.2: Carry Select con registri

M,P	Min Period	Max Freq	M,P	Min Period	Max Freq	M,P	Min Period	Max Freq
8,1	4.851 ns	206.160 MHz	16,1	6.235 ns	160.377 MHz	32,1	7.168 ns	139.511MHz
8,2	4.497 ns	222.366 MHz	16,2	6.233 ns	160.443 MHz	32,2	8.575ns	116.615MHz
8,4	4.851 ns	206.160 MHz	16,4	5.976 ns	167.336 MHz	32,4	7.363ns	135.811MHz
—	—	—	16,8	6.235 ns	160.377 MHz	32,8	7.504ns	133.259MHz

Tabella 9.3: Periodo minimo e massima frequenza

Capitolo 10

Moltiplicatori

10.1 Moltiplicatore a celle MAC

10.1.1 Traccia

Realizzare in VHDL un circuito di moltiplicazione a celle MAC di N bit. La cella MAC deve contenere un Full Adder (descritto già in esercizi precedenti) 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.

10.1.2 Esercizio

Il moltiplicatore a Celle MAC è un particolare tipo di moltiplicatore che consiste nel “distribuire” l’operazione di somma dei singoli bit dei prodotti parziali. Lo schema che si viene a formare da questo tipo di moltiplicatore è detto “a matrice”, dove la singola cella al suo interno implementa un’operazione. Questa operazione di base è Multiply-And-Accumulate che è la sequenza di due passaggi elementari:

1. Moltiplicazione tra due operandi B e C
2. Sommare il risultato in un accumulatore

E’ possibile riassumere quanto detto in un’unica, compatta, espressione $A \leftarrow A + BC$.

Tale calcolo può essere effettuato da una “cella MAC” il cui insieme di dati in ingresso-uscita è riassunto nello schema di Fig. , insieme allo schema matriciale complessivo.

Gli elementi in ingresso al singolo blocco sono:

- Il prodotto parziale calcolato tramite una porta AND
- Il risultato della somma parziale degli elementi che si trovano sulla stessa colonna
- Il riporto proveniente dalla analoga somma immediatamente a destra, cioè sulla stessa riga

Con questo set in ingresso, verranno prodotti i seguenti risultati:

- La nuova somma parziale
- Il riporto da propagare alla somma a sinistra sulla stessa riga

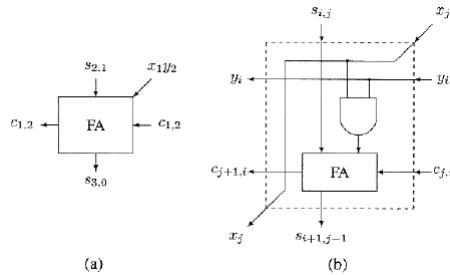


Figura 10.1: Cella di base MAC

In particolare si fa notare che tutte le celle appartenenti alla stessa riga utilizzano per il prodotto lo stesso bit dell'operando Y e, viceversa, tutte quelle sulla stessa diagonale il bit dell'operando X. Questa omogeneità nella distribuzione dei dati permette di costruire lo schema matriciale complessivo di Fig. . Da tale figura è possibile anche osservare che, come è normale per un'operazione di moltiplicazione tra operandi di N bit, il risultato sarà rappresentato su $2N$ bit.

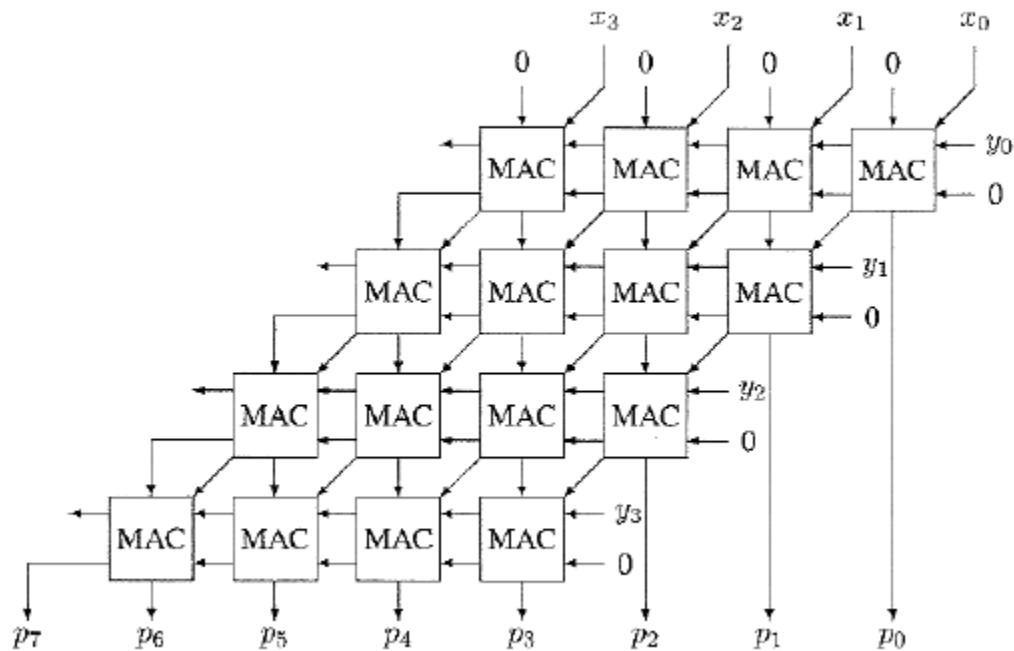


Figura 10.2: Schema matriciale moltiplicatore

10.1.2.1 Implementazione

Il modulo più significativo della struttura implementata è sicuramente quello della cella MAC. E' questo infatti il modulo fondamentale che consente di realizzare la logica di funzionamento se

opportunamente interconnesso come mostrato in Fig. . In aggiunta a questo componente ne sono stati sviluppati ulteriori 2, che per semplicità non verranno riportati. In particolare si è deciso di riutilizzare il FullAdder precedentemente mostrato e di scrivere un top module in grado di istanziare e collegare adeguatamente le celle MAC secondo la struttura matriciale.

```

1
2 entity Cella_MAC is
3 Port (
4     x_in : in STD_LOGIC;
5     y_in : in STD_LOGIC;
6     c_in : in STD_LOGIC;
7     s_in : in STD_LOGIC;
8     s_out : out STD_LOGIC;
9     x_out : out STD_LOGIC;
10    y_out : out STD_LOGIC;
11    c_out : out STD_LOGIC);
12 end Cella_MAC;
13
14
15 architecture Structural of Cella_MAC is
16
17 COMPONENT full_adder
18 PORT(
19     x : IN std_logic;
20     y : IN std_logic;
21     ci : IN std_logic;
22     s : OUT std_logic;
23     co : OUT std_logic
24 );
25 END COMPONENT;
26
27 signal and_out : STD_LOGIC;
28
29 begin
30
31     x_out <= x_in;
32     y_out <= y_in;
33
34     and_out <= x_in and y_in;
35
36     fa : full_adder port map(
37         x => s_in,
38         y => and_out,
39         ci => c_in,
40         co => c_out,
41         s => s_out

```

```

42 );
43
44 end Structural;

```

10.1.2.2 Considerazioni area e ritardo

La cella base illustrata in Fig. risulta essere articolata su 3 livelli, tuttavia è possibile ottimizzarla e portarla su 2. Così facendo il ritardo introdotto dal singolo blocco MAC risulta essere pari a $2T$. Con queste ipotesi, un semplice calcolo della propagazione dei ritardi tra le celle sull'esempio con $N=4$ bit di Fig. fornisce come risultato un ritardo totale di $20T$ in uscita sul bit più significativo. Analizzando e generalizzando ulteriormente i calcoli è possibile accorgersi del fatto che il ritardo è circa pari a $6(N - 1)T$.

Ulteriori analisi possono essere effettuate riguardo l'area del dispositivo. Infatti questa è in generale uguale AKN^2 con K dimensione della cella MAC ($K=7$ se su 3 livelli, $K=10$ se su 2).

Sono state effettuate diverse sintesi del circuito con l'ausilio del Tool ISE, in modo da trovare un riscontro con i dati teorici appena mostrati. Si è testata l'architettura con valori di N pari a 8,16,32,64 bit i cui risultati di area espressa in LUT da 4 ingressi sono mostrati in Tab. . In aggiunta per il calcolo dei ritardi della rete si è deciso di aggiungere alla struttura dei registri che presentino gli ingressi alla rete e prendano l'uscita. In questo modo si è potuto arrivare all'osservazione dei dati presenti in Tab. .

10.2 Robertson

10.2.1 Traccia

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

10.2.2 Soluzione

L'algoritmo di Robertson si fonda su una sequenza di operazioni che richiedono in ogni passaggio i-esimo una sola operazione di shift verso destra, contrariamente alle i-esime operazioni di shift richieste da quello manuale.

La caratteristica principale di Robertson è quella che può moltiplicare numeri relativi rappresentati in complementi a due.

Con una rappresentazione in modulo e segno, si sarebbe potuto pensare di utilizzare direttamente i moltiplicatori paralleli e calcolare alla fine il bit di segno risultante.

Invece con una rappresentazione in complementi a due, questa strada richiederebbe prima di calcolarsi il complemento a due degli operandi negativi per poi effettuare la moltiplicazione tra numeri positivi, e infine, se il segno del risultato sarebbe dovuto essere negativo, anche il complemento a due del risultato.

Per evitare questo modo di operare, si è introdotto l'algoritmo di Robertson.

10.2.2.1 Interfaccia del Moltiplicatore di Robertson

Il risultato sarà disponibile in uscita quando il bit di Stop è alto.

```

1 entity Moltiplicatore_NperNbit_Robertson is
2 generic (N: integer:=8);
3 Port ( op1 : in STD_LOGIC_VECTOR(N-1 downto 0);
4         op2 : in STD_LOGIC_VECTOR(N-1 downto 0);
5         start : in STD_LOGIC;
6         clock : in STD_LOGIC;
7         reset_n : in STD_LOGIC;
8         stop : out STD_LOGIC;
9         risultato : out STD_LOGIC_VECTOR(2*N-1 downto 0));

```

Listing 10.1: Entity Moltiplicatore_NperNbit_Robertson

10.2.3 Implementazione

Di seguito si presenterà l'algoritmo di Robertson partendo dalla Control Unit, una macchina sequenziale sincrona, e descrivendo come i comandi prodotti da essa scandiranno i vari passaggi del prodotto di due numeri relativi.

10.2.3.1 Lo stato Idle

Lo stato idle è lo stato iniziale/finale. Se il flag di Start è alto si procede con la sequenza di operazioni (`next_stato<= init`), inoltre quando si resetta il moltiplicatore Idle diviene lo stato presente.

Quando l'operazione di Moltiplicazione si è conclusa si ritorna nello stato Idle e si pone il flag di Stop a '1'.

10.2.3.2 Lo stato Init

Lo stato Init resetta sia il conteggio, che scandirà i passaggi della moltiplicazione, sia il registro A. Il registro A di N bit conterrà quindi tutti valori logici bassi.

Il registro Q ed M sono abilitati a leggere rispettivamente il moltiplicatore X e il moltiplicando Y.

Il prossimo stato è "scelta_operazione"

10.2.3.3 Lo stato Scelta_operazione

Se il bit meno significativo del moltiplicatore X è '0' si eseguirà direttamente l'operazione di shift (verso destra) del registro A e Q (`next_stato<= Shift_stato`). Altrimenti Se il bit meno significativo del moltiplicatore X è '1' bisognerà eseguire l'operazione di somma (`next_stato<= add_sub`).

10.2.3.4 Lo stato Shift _ stato

Si occuperà di abilitare lo shift ($\text{shift} \leq 1'$: il mux vincola a leggere esclusivamente il bit di scan_in e non il dato D risultante dalla somma) e i due registri a leggere.

Se il contatore non ha finito ($\text{counter_hit} = '0'$) si ritorna allo stato Scelta _ operazione altrimenti si ritorna nello stato iniziale Idle presentando in uscita il risultato finale.

10.2.3.5 Lo stato Add _ sub

Anche qui abilito il registro A a leggere, ma questa volta il dato D risultante dal sommatore (senza il bit di overflow).

In particolare il risultato della somma, che è già presente in uscita quando si abilita il registro A a leggere, è il risultato della somma (o sottrazione) del contenuto del registro A con il Moltiplicando contenuto nel registro M.

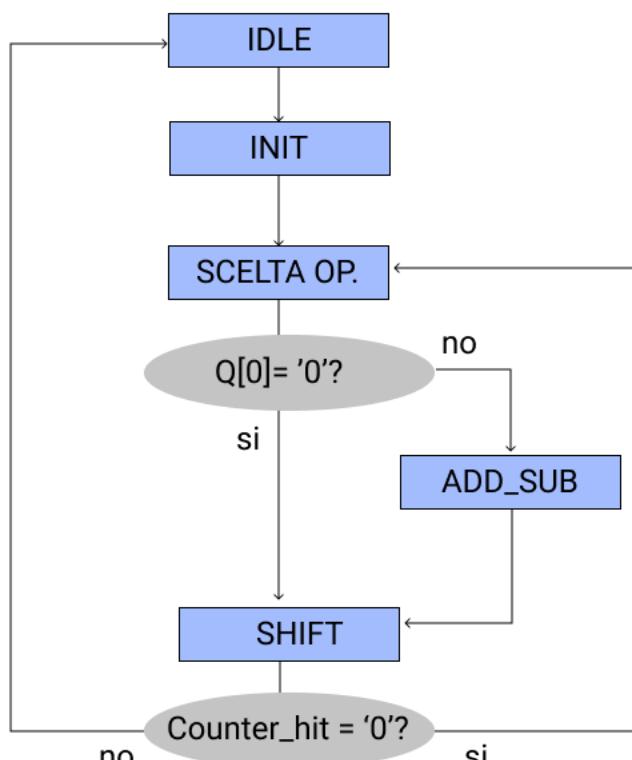


Figura 10.3: Evoluzione della rete

```

1 library IEEE;
2 use IEEE.STD_LOGIC_1164.ALL;
3 entity ControlUnit is port(
4     start: in STD_LOGIC;
5     clock: in STD_LOGIC;
6     reset_n : in STD_LOGIC;
7     reset_a : out STD_LOGIC;

```

```

8   reset conteggio : out STD_LOGIC;
9   stop : out STD_LOGIC;
10  en_a, en_q, en_m : out STD_LOGIC;
11  q_lessSignificantBit : in STD_LOGIC;
12  shift : out STD_LOGIC;
13  conteggio_up : out STD_LOGIC;
14  counter_hit : in STD_LOGIC—serve a stoppare il conteggio
15  );
16 end ControlUnit;
17 architecture Behavioral of ControlUnit is
18 Type Stati is (idle, init, scelta_operazione, shift_stato, add_sub);
19 signal stato_corrente, next_stato : Stati := idle;
20 begin
21 cambio_stato: process(clock, reset_n) begin
22  if reset_n = '0' then
23    stato_corrente <= idle;
24  elsif rising_edge(clock) then
25    stato_corrente <= next_stato;
26  end if; end process;
27
28 FSM: process (stato_corrente, start, reset_n, counter_hit,
29   q_lessSignificantBit) begin
30  shift <= '0';
31  conteggio_up <='0';
32  stop <= '0';
33  reset_a <= '0';
34  reset_conteggio <= '0';
35  en_a <= '0';
36  en_q <= '0';
37  en_m <= '0';
38
39  case stato_corrente is
40    when idle =>
41      stop <= '1';
42      if start = '1' then
43        next_stato <= init;
44      else
45        next_stato <= idle;
46      end if;
47    when init =>
48      reset_a <= '1';
49      reset_conteggio <= '1';
50      en_q <= '1';—abilitazione per leggere l'operando 2(
51      moltiplicatore)
52      en_m <= '1';—abilitazione per leggere l'operando 1(moltiplicando
53      )

```

```

51      next_stato <= scelta_operazione;
52
53      when scelta_operazione =>
54          if q_lessSignificantBit = '0' then
55              next_stato <= shift_stato;
56          else
57              next_stato <= add_sub;
58          end if;
59      when add_sub =>
60          en_a <= '1'; —abilita a ma lo shift 0
61          next_stato <= shift_stato;
62      when shift_stato =>
63          en_q <= '1'; —abilito sia a che q con shift 1 poich li devo
64          shiftare entrambi
65          shift <= '1';
66          en_a <= '1';
67          if counter_hit = '0' then — se non il contatore non ha ancora
68              terminato(N_bit-1), abilita il conteggio
69              conteggio_up <= '1';
70              next_stato <= scelta_operazione;
71          else
72              next_stato <= idle; — a conteggio terminato torna nello stato
73              idle!
74          end if;
    end case;
end process;
end Behavioral;
```

Listing 10.2: Control Unit del Moltiplicatore di Robertson

10.2.3.6 Gestione dei quattro casi di prodotto di due numeri relativi

Per garantire un funzionamento corretto il valore di scan_in deve essere inizializzato al valore logico basso e deve rimanere tale almeno finché non si incontra il primo bit alto del moltiplicatore X (finché Q[0]=1).

Dal primo bit di X alto il valore di Scan_in sarà costante fino al penultimo passaggio e pari a:

- Se il Moltiplicando è negativo ($M[7] = '1'$), si avrà una somma di due numeri negativi almeno fino al penultimo passaggio. La somma tra due numeri negativi rappresentati in complementi a due si svolge come una normale somma, ma è opportuno assicurarsi che il bit più significativo sia alto.

Per garantire che il risultato delle somme sia negativo si utilizza il segnale di Scan_in che dovrà assumere un valore logico alto almeno fino al penultimo passaggio assicurando con l'operazione di shift che il valore risultante dall'addizione di A con il Moltiplicatore (caso in cui $Q[0] = 1$ e quindi si ha l'addizione tra due numeri negativi) o con tutti '0' (quando $Q[0] = 0$ e si prosegue direttamente con lo shift) sia sempre negativo.

- Se il Moltiplicando è positivo ($M[7]=0'$) si hanno semplici addizioni tra numeri positivi in complementi a due e quindi per assicurarsi che il risultato in ogni passo sia positivo il valore logico assunto da Scan_in dovrà essere basso.

Nell'ultimo passo (counter_hit =1) vi è bisogno che il bit più significativo di A (e quindi del risultato totale) assuma il valore logico del bit di segno risultante dall'operazione di moltiplicazione:

- se il Moltiplicatore ($Q[0]=1$) è negativo bisogna fare la sottrazione di A con il Moltiplicando.

In questo caso lo shift seguente dovrà utilizzare come segnale di Scan_in il valore logico basso quando il Moltiplicando è negativo e quello alto quando è positivo.

- se il Moltiplicatore ($Q[0]=0$) è positivo bisogna fare, come al solito, direttamente solo lo shift utilizzando come scan_in il valore logico che si aveva nel passo precedente.

In questo modo il bit più significativo del risultato sarà uguale a quello del Moltiplicando Y.

10.2.3.7 Come i quattro casi sono gestiti nella rete

Si descriveranno di seguito come verranno utilizzati dalla rete un flip flop F e il counter_hit del contatore per gestire i quattro casi di prodotto di due numeri relativi.

Il flip flop F presenta in uscita il bit che contribuirà alla logica che genererà il segnale di Scan_in nell'operazione di Shift quando lo stato presente è "Shift_stato".

Esso è inizializzato al valore logico basso finché non si ha il primo bit del Moltiplicatore diverso da zero.

Dal primo bit di X alto il valore memorizzato dal flip flop F sarà costante fino al penultimo passaggio e pari

- al livello logico alto se il bit più significativo del Moltiplicatore è alto
- al livello logico basso in caso contrario.

Questo comportamento è ottenuto con l'entity flipflopF definita come nel List.10.3. Il valore in uscita del componente flipflopF è associato al segnale temporaneo "new_msb_a_temp" nel Lis. 10.4.

```

1 entity flipflopF is port(
2     Q_lsb: in std_logic;
3     M_msb: in std_logic;
4     reset_n: in std_logic;
5     clk: in std_logic;
6     F: out std_logic
7 );
8 end flipflopF;
9
10 architecture Behavioral of flipflopF is
11 signal temp: std_logic := '0';
12 begin
13     process(clk,reset_n)
14     begin
15         if(reset_n = '0') then
16             temp<= '0';
17         elsif(rising_edge(clk)) then
18             if( Q_lsb = '1' and M_msb = '1') then
19                 temp<= '1';

```

```

20      end if;
21      end if;
22  end process;
23  F <= temp;
24 end Behavioral;
```

Listing 10.3: FlipFlop F

Nell'ultimo passo (`counter_hit =1`) per gestire la sottrazione si è utilizzato come segnale di Subtract il segnale `Counter_hit` del contatore.

Quindi l'ADD-SUB negli N-1 passaggi si comporterà come un addizionatore e nell'ultimo come un sottrattore.

In questo ultimo passaggio il segnale di `Scan_in` (sarebbe il segnale `msb_a_temp` nel List.) affinchè assuma valore pari al bit di segno, non dipende dal valore in uscita dal FlipFlop F bensì dal risultato della XOR tra i due bit più significativi dei due operandi.

```

1 msb_a_temp <= (new_msba_temp and not counter_hit_temp) or ((m_temp(N
    -1) xor op1(N-1)) and counter_hit_temp);
```

Listing 10.4: Segnale “`msb_a_temp`” associato alla porta effettiva Scan In del componente Shift Register A

10.3 Booth

10.3.1 Traccia

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

10.3.2 Soluzione

L'algoritmo di Booth è equivalente a quello di Robertson: esegue il prodotto di due numeri relativi rappresentati in complementi a due.

La vera differenza tra i due sono le condizioni in cui risultano efficienti. Infatti con un moltiplicatore contenente tutti valori logici alti, per l'algoritmo di Robertson risulterà essere la condizione peggiore in termini di efficienza, poiché dovranno essere effettuate N-1 somme e 1 sottrazione oltre le N operazioni di Shift, mentre per l'algoritmo di Booth risulterà quella ottimale dovendo effettuare principialmente solo operazioni di Shift.

Ciò è ottenuto con la codifica di Booth. Mentre un bit può assumere solo due valori logici, l'idea di Booth è stata quella di sfruttare coppie adiacenti di bit in modo tale da lavorare con più valori in un'astrazione più alta. Ad esempio la stringa “11101111” rappresentata in complementi a due è equivalente semanticamente alla stringa “0,0,-1,1,0,0,0,-1” secondo la codifica di Booth.

Aumentare l'astrazione della semantica in generale potrebbe portare a un aumento della complessità della rete sequenziale.

Infatti ora in ogni passaggio bisogna valutare tre valori logici per i quali si hanno tre comportamenti diversi:

- se si ha '01' (sarebbe 1 nella semantica più ad alto livello secondo Booth) si deve effettuare l'addizione

- se si ha '10' (sarebbe -1 nella semantica più ad alto livello secondo Booth) bisogna effettuare l'operazione di sottrazione

- se si ha '00' o '11' (sarebbe 0 nella semantica più ad alto livello secondo Booth) si deve effettuare solo l'operazione di Shift

Per fortuna facendo delle opportune considerazioni si riuscirà addirittura ad ottenere una rete più semplice.

10.3.3 Implementazione

Di seguito si presenterà l'algoritmo di Booth descrivendo solo le modifiche apportate al progetto che implementava l'algoritmo di Robertson.

10.3.3.1 Lo Shift Register Q

Affinchè si abbia anche per il bit meno significativo del moltiplicatore una coppia di bit da leggere, il moltiplicatore, situato nello Shift Register Q, sarà affiancato da un ulteriore bit meno significativo che conserverà per tutto il processo un valore logico basso.

10.3.3.2 La Control Unit

La Control Unit rimane prevalentemente la stessa con l'unica differenza che l'operazione di Shift viene effettuata direttamente sorpassando lo stato Add_Sub quando il bit meno significativo di Q e il penultimo bit meno significativo di Q sono uguali (ovvero quando la coppia considerata è '00' o '11')

10.3.3.3 Il valore logico attribuito alla porta effettiva Scan_in dello Shift Register A

Non ha più senso utilizzare il flipflop F. Si osservi che nella sequenza di operazioni effettuate nell'algoritmo di Booth non vi potranno mai essere due somme o sottrazioni di seguito.

Inoltre la prima operazione, oltre eventuali singole operazioni di Shift dovute ai primi valori logici bassi meno significativi del moltiplicatore, sarà sempre una sottrazione.

Quindi se il moltiplicando è positivo (o negativo), al primo '1' meno significativo del Moltiplicatore, si avrà un'operazione di sottrazione (ovvero di somma con un numero negativo (o positivo)) con tutti valori logici bassi, e quindi il risultato è rispettivamente negativo (o positivo).

Successivamente se vi sono degli ulteriori '1' si esegue semplicemente l'operazione di shift, e il risultato deve rimanere rispettivamente negativo (o positivo).

Non vi può essere un'altra operazione di sottrazione se prima non vi è un'operazione di somma.

Con l'operazione di somma si andrà a sommare il moltiplicando positivo (o negativo) con il numero negativo (o positivo) contenuto nel registro A e così via.

Ciò implica quindi che vi è sempre un'operazione di somma di numeri di segno opposto rappresentati in complementi a due.

Quindi il valore logico attribuito alla porta effettiva Scan_in dello Shift Register A, dopo un'eventuale operazione di somma o sottrazione, potrà essere sempre associato al valore logico associato al bit più significativo dello Shift Register A.

```

1 registro_A: Shift_register
2     generic map( N=>N, shift_right_left=>'0')
3     port map(
4         reg_in=>somma_a_temp,
5         scan_in=>a_temp(N-1),
6         scan_enable=>shift_temp,
7         clk=>clock,
8         reset_n=>reset_a_temp,
9         enable=>en_a_temp,
10        Q=>a_temp,
11        scan_out=>msb_q_temp
12    );

```

Listing 10.5: Componente Shift Register A istanziato nel Moltiplicatore di Booth

10.3.3.4 L' addizionatore sottrattore

Adesso bisogna chiedersi quale segnale associare alla porta effettiva “add_sub_n” affinchè si comandi un’operazione di sottrazione o di addizione.

L’idea è quella di associaragli il bit più significativo della coppia di bit che costituiscono in quel momento i coefficienti per la codifica di Booth.

Questo perchè se la coppia contiene ‘00’ o ‘11’ anche se verranno effettuate rispettivamente o la somma o la sottrazione dalla rete combinatoria ADD_SUB, comunque il registro A sarà abilitato a leggere solo il segnale di Scan_in nell’operazione di Shift.

Invece se la coppia è costituita da ‘01’ o da ‘10’ verranno sempre effettuate (giustamente) la somma o la sottrazione, ma questa volta il registro A sarà abilitato a leggere il risultato della somma o della sottrazione.

```

1 Add_Sub_pm: RCA_ADD_SUB
2     generic map( N=>N)
3     port map(
4         OP1=>a_temp,
5         OP2=>m_temp,
6         add_sub_n=>Q_temp(1),
7         s=>somma_a_temp,
8         overflow=>open
9     );

```

Listing 10.6: Componente RCA_ADD_SUB istanziato nel Moltiplicatore di Booth

10.3.3.5 Il risultato

Ovviamente il risultato dovrà leggere solo il contenuto del registro A e dei primi N bit del registro Q scartando quello meno significativo.

10.4 Analisi di Booth e Robertson

10.4.1 Tempificazioni

Si confronteranno di seguito i due algoritmi con una Simulazione di tipo Post-route variando i due operandi

10.4.2 Simulazioni Post-Route

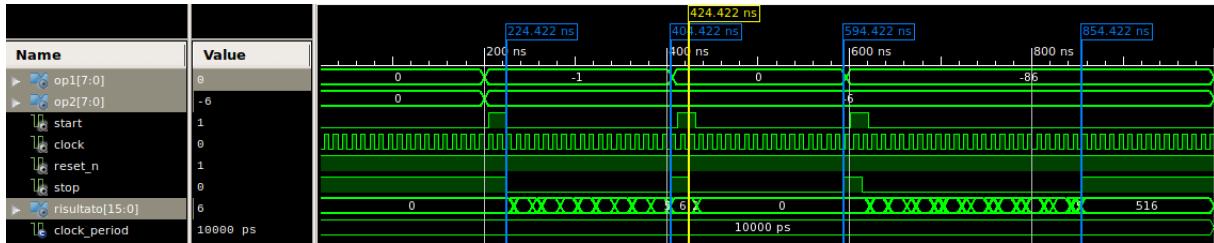


Figura 10.4: Simulazione Post Route di Booth

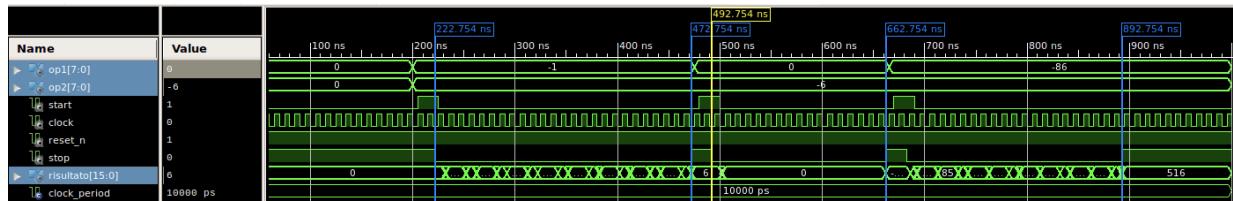


Figura 10.5: Simulazione Post Route di Robertson

Operando 2 (Y)	Operando 1 (X)	Tempo di risposta di Booth	tempo di risposta di Robertson
11111010	11111111	$ 224-404 = 180$ ns	$ 222-472 = 250$ ns
//	00000000	$ 424-594 = 170$ ns	$ 492-662 = 170$ ns
//	10101010	$ 614-854 = 200$ ns	$ 682-892 = 210$ ns

Tabella 10.1: Confronto dei tempi di risposta

Come ci si poteva aspettare:

- se il moltiplicatore è costituito da tutti valori logici alti l'algoritmo di Booth è 1/3 più veloce di quello di Robertson in quanto deve eseguire principalmente solo operazioni di Shift
- se il moltiplicatore è costituito da tutti valori logici bassi entrambi gli algoritmi devono eseguire principalmente solo operazioni di Shift e quindi avranno lo stesso tempo di risposta
- se il moltiplicatore è costituito da una sequenza alternata di valori logici bassi ed alti le due prestazioni si possono quasi considerare equivalenti.

Si può concludere che in generale è più conveniente utilizzare l'algoritmo di Booth perché non solo è equivalente a quello di Robertson, ma in più quando abbiamo sequenze continue di valori alti il tempo di risposta si riduce ulteriormente.

Capitolo 11

Divisori

11.1 Divisore Restoring

11.1.1 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).

11.1.2 Soluzione

Il divisore è un dispositivo in grado di calcolare il quoziente Q e il resto R di una divisione tra il dividendo D e il divisore V rispettivamente.

Supponendo di avere D e V rispettivamente di m e n bit, possiamo constatare che:

- Al più il resto risulterà pari a $V - 1$ e quindi espresso su n bit
- Il quoziente sarà rappresentato al massimo su m bit (nel caso $V = 1$ infatti il quoziente sarà proprio pari a D). In generale il quoziente viene espresso su $m - n + 1$ bit.

La soluzione adottata in questo esercizio utilizza una versione di calcolo alternativa a quella manuale per il calcolo di una divisione tra interi. Infatti al passo i -esimo, l'algoritmo manuale prevede, nel caso la cifra i -esima del quoziente fosse pari a '1', lo shift di i posizioni a destra del divisore in modo da poter essere confrontato correttamente con il risultato parziale del resto. Ciò è mostrato nell'esempio di fig. da cui è ricavata la formula $R_{i+1} = R_i - q_i 2^{-i} V$.

Al contrario l'algoritmo utilizzato, nonostante sia equivalente a quello derivato dalla procedura manuale, permette di effettuare ad ogni passo della divisione lo shift di una sola posizione a sinistra del resto parziale.

Infatti è possibile effettuare la seguente sequenza di operazioni:

1. Left shift di una posizione di R_i (Quindi $R_i = 2R_i$)
2. Confronto di $2R_i$ con V per determinare q_i (effettuando una sottrazione e valutando il segno, q_i sarà '0' con quando negativo e '1' nel caso opposto)
3. Sottrazione del prodotto $q_i V$ da $2R_i$ (Quindi $R_{i+1} = R_i - q_i V$ dove $R_i = 2R_i$).

11.1.2.1 Implementazione

Si è deciso di strutturare il sistema nel modo illustrato nello schematico in fig. . In particolare si è supposto che il dividendo D fosse di dimensione pari a $m = 2n$ bit dove n è il numero di bit sui quali è rappresentato il divisore V . Inoltre, come accennato in precedenza, il quoziente Q sarà espresso su $m - n + 1 = n + 1$ bit (per semplicità è stato scelto pari a n). Operazioni che richiedano un dimensionamento del quoziente maggiore non forniranno il risultato corretto.

A tal proposito nella top entity sono stati istanziati diversi componenti, i seguenti completano la **parte operativa**:

1. Registro V di n bit contenente il divisore.
2. Shift register S di 1 bit contenente il bit più significativo del dividendo (mantiene il segno dell'operazione di sottrazione).
3. Shift register A di n bit contenente in prima istanza gli n bit più significativi del dividendo, e successivamente i risultati delle operazioni fornite dall'adder. Questo registro conterrà alla fine il resto R della divisione.
4. MuxA adibito a selezionare l'ingresso al registro A. Infatti in fase di inizializzazione andrà caricata parte del dividendo, mentre negli stati intermedi il mux farà passare il risultato dell'adder.
5. Shift Register Q di n bit contenente prima gli n bit meno significativi del dividendo, shiftato ad ogni operazione di sottrazione con scan in la cifra i -esima del quoziente. Conterrà alla fine il valore del quoziente Q .
6. RippleCarryAdder/Sub utile ad effettuare le operazioni fondamentali di sottrazione ed addizione(nel caso di correzione).
7. Counter modulo n che terrà il conteggio dei passaggi effettuati, quindi quelli rimasti da effettuare per la fine della divisione.

Appartiene invece alla **parte di controllo**:

1. Control unit che fornisce in uscita tutti i segnali di coordinazione dei componenti in modo da implementare la logica di funzionamento.

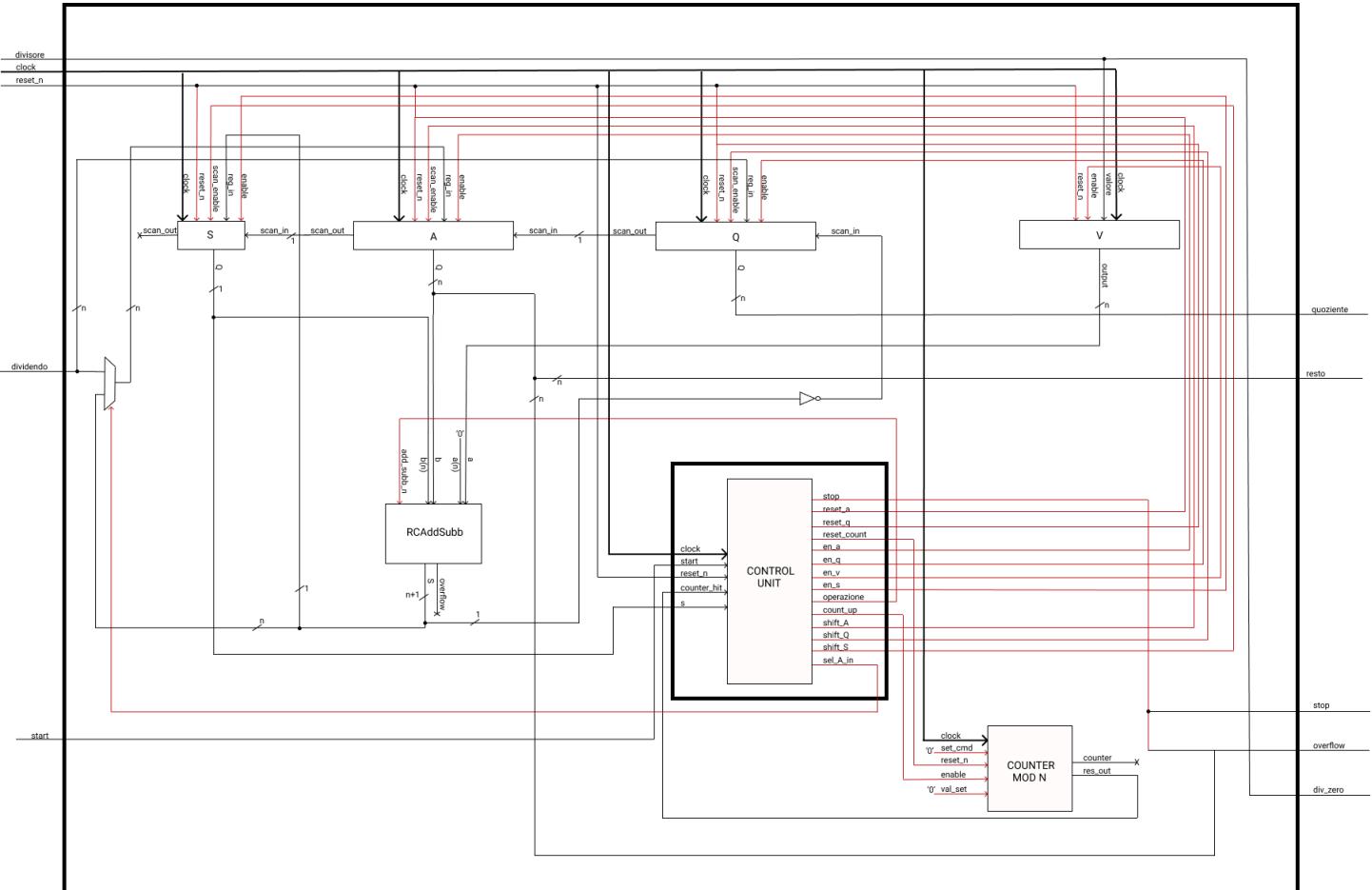


Figura 11.1: Schematico divisore

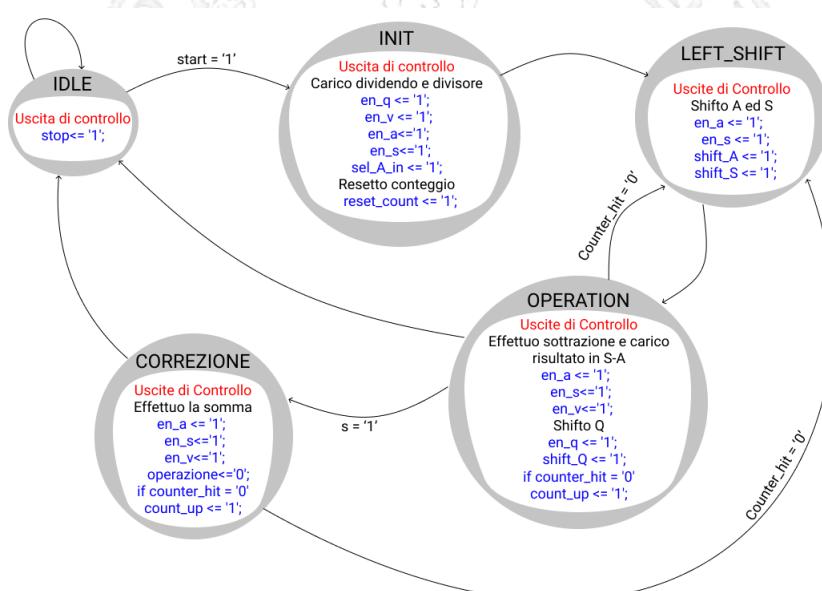
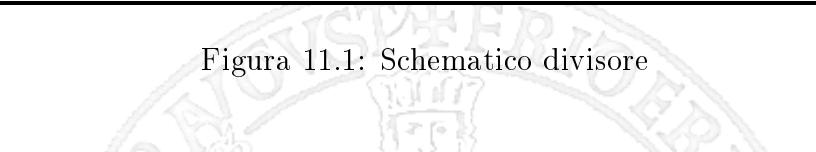


Figura 11.2: Funzionamento corretto CU

11.1.2.2 Codice

L'architettura è stata realizzata in VHDL riutilizzando principalmente componenti sviluppati in esercizi precedenti come il ripple carry adder, shift register, un registro ed un contatore modulo N. È stata poi implementata una control unit che fornisce in uscita un insieme di segnali di controllo in grado di coordinare tutte le parti dell'architettura.

Per semplicità si riportano nel seguito soltanto i codici riguardanti la CU e la top entity, la quale istanzia e interfaccia opportunamente i componenti.

```

1  entity control_unit is
2      Port ( clock : in STD_LOGIC;
3              start : in STD_LOGIC;
4              reset_n : in STD_LOGIC;
5                  counter_hit : in STD_LOGIC;
6                  s : in STD_LOGIC;
7                  div_zero : in STD_LOGIC;
8                      stop : out STD_LOGIC;
9                      reset_a : out STD_LOGIC;
10                     reset_q : out STD_LOGIC;
11                     reset_count : out STD_LOGIC;
12                     en_a : out STD_LOGIC;
13                     en_q : out STD_LOGIC;
14                     en_v : out STD_LOGIC;
15                     en_s : out STD_LOGIC;
16                     operazione: out STD_LOGIC;
17                     count_up : out STD_LOGIC;
18                     shift_A : out STD_LOGIC;
19                     shift_Q : out STD_LOGIC;
20                     shift_S : out STD_LOGIC;
21                     sel_A_in : out STD_LOGIC
22     );
23
24 end control_unit;
25
26 architecture Behavioral of control_unit is
27
28 type state is (idle, init, operation, left_shift, correzione);
29
30 signal stato_corrente, nxt_stato : state := idle;
31
32 begin
33
34 state_proc: process(clock, reset_n)
35 begin
36     if reset_n = '0' then
37         stato_corrente <= idle;
38     elsif rising_edge(clock) then

```

```

39     stato_corrente <= nxt_stato;
40   end if;
41 end process;
42
43 fsm: process(stato_corrente, start, reset_n, counter_hit, s)
44 begin
45
46   stop <= '0';
47   reset_a <= '0';
48   reset_q <= '0';
49   reset_count <= '0';
50   en_a <= '0';
51   en_q <= '0';
52   en_v <= '0';
53   en_s <= '0';
54   count_up <= '0';
55   shift_A <= '0';
56   shift_Q <= '0';
57   shift_S <= '0';
58   sel_A_in <= '0';
59   operazione <='1'; —ad ogni passo deve fare sottrazioni
60
61 case stato_corrente is
62   when idle =>
63     stop <= '1';
64     if start = '1' then
65       nxt_stato <= init;
66     else
67       nxt_stato <= idle;
68     end if;
69
70   when init =>
71     en_q <= '1';
72     en_v <= '1';
73     en_a <= '1';
74     en_s <= '1';
75     reset_count <= '1';
76     sel_A_in <= '1';
77     if div_zero = '1' then
78       stop <= '1';
79       nxt_stato <= idle;
80     else
81       nxt_stato <= left_shift;
82     end if;
83
84   when left_shift =>

```

```

85      en_a <= '1';
86      en_s <= '1';
87      shift_A <= '1';
88      shift_S <= '1';
89      nxt_stato <= operation;
90
91  when operation =>
92      en_a <= '1';
93      en_s <= '1';
94      en_v <= '1';
95      en_q <= '1';
96      shift_Q <= '1';
97
98      if s='1' then nxt_stato<=correzione;
99      elsif counter_hit = '0' then
100         count_up <= '1';
101         nxt_stato <= left_shift;
102     else nxt_stato <= idle;
103     end if;
104
105 when correzione =>
106     en_a <= '1';
107     en_s <= '1';
108     en_v <= '1';
109     operazione <='0'; — nelle stato correzione si effettua una somma
110
111     if counter_hit = '0' then
112         count_up <= '1';
113         nxt_stato <= left_shift;
114     else nxt_stato <= idle;
115     end if;
116
117 end case;
118
119 end process;
120
121 end Behavioral;

```

Listing 11.1: CONTROL UNIT

```

1 entity divisore_restoring is
2   generic (n : integer := 4);
3   Port ( dividendo : in STD_LOGIC_VECTOR (2*n-1 downto 0);
4          divisore : in STD_LOGIC_VECTOR (n-1 downto 0);
5          start : in STD_LOGIC;
6          clock : in STD_LOGIC;
7          reset_n : in STD_LOGIC;

```

```

8      stop : out STD_LOGIC;
9      div_zero : out STD_LOGIC;
10     overflow : out STD_LOGIC;
11     quoziante : out STD_LOGIC_VECTOR (n-1 downto 0);
12     resto : out STD_LOGIC_VECTOR (n-1 downto 0));
13 end divisore_restoring;
14
15 architecture Structural of divisore_restoring is
16
17
18 component control_unit
19   Port ( clock : in STD_LOGIC;
20         start : in STD_LOGIC;
21         reset_n : in STD_LOGIC;
22         counter_hit : in STD_LOGIC;
23         s : in STD_LOGIC;
24         div_zero : in STD_LOGIC;
25         stop : out STD_LOGIC;
26         reset_a : out STD_LOGIC;
27         reset_q : out STD_LOGIC;
28         reset_count : out STD_LOGIC;
29         en_a : out STD_LOGIC;
30         en_q : out STD_LOGIC;
31         en_v : out STD_LOGIC;
32         en_s : out STD_LOGIC;
33         operazione : out STD_LOGIC;
34         count_up : out STD_LOGIC;
35         shift_A : out STD_LOGIC;
36         shift_Q : out STD_LOGIC;
37         shift_S : out STD_LOGIC;
38         sel_A_in : out STD_LOGIC);
39 end component;
40
41
42 component Counter_ModN is generic (N : integer := 4);
43   port (
44     clock : in STD_LOGIC;
45     reset_n : in STD_LOGIC;
46     enable : in STD_LOGIC;
47     counter : out STD_LOGIC_VECTOR ( integer (( floor (LOG2 ( real(N -
48       1 )))) ) DOWNTO 0) ;
49     res_out : out std_logic := '0'
50   );
51 end component;
52
53 component ff_d

```

```

53 port( data_in:  in std_logic;
54     reset_n :  in std_logic;
55     clock:  in std_logic;
56     enable:  in std_logic;
57     data_out:  out std_logic
58 );
59 end component;
60
61 component reg is
62     Generic(width : integer := 8);
63     Port ( valore : in STD_LOGIC_VECTOR (width-1 downto 0);
64             clock : in STD_LOGIC;
65             enable : in STD_LOGIC;
66             reset_n : in STD_LOGIC;
67             output : out STD_LOGIC_VECTOR (width-1 downto 0));
68 end component;
69
70 component RCAddSub is
71     Generic(num_bit : integer := 8);
72     Port ( a : in STD_LOGIC_VECTOR(num_bit-1 downto 0);
73             b : in STD_LOGIC_VECTOR(num_bit-1 downto 0);
74             add_sub_n : in STD_LOGIC;
75             s : out STD_LOGIC_VECTOR (num_bit-1 downto 0);
76             overflow : out STD_LOGIC);
77 end component;
78
79 component Shift_register is
80     generic( N:integer:= 8; shift_dx_o_sx:std_logic := '1');
81     port(
82         reg_in:  in std_logic_vector(N-1 downto 0);
83         scan_in:  in std_logic;
84         scan_enable:  in std_logic;
85         clk:  in std_logic;
86         reset_n:  in std_logic;
87         enable:  in std_logic;
88         Q:  out std_logic_vector(N-1 downto 0);
89         scan_out:out std_logic
90     );
91 end component;
92
93 component MUXN is
94     generic (N: integer := 8);
95     port(
96         a :  in std_logic_vector(N-1 downto 0);
97         b :  in std_logic_vector(N-1 downto 0);
98         sel :  in std_logic;

```

```

99      o : out std_logic_vector(N-1 downto 0)
100    );
101  end component;

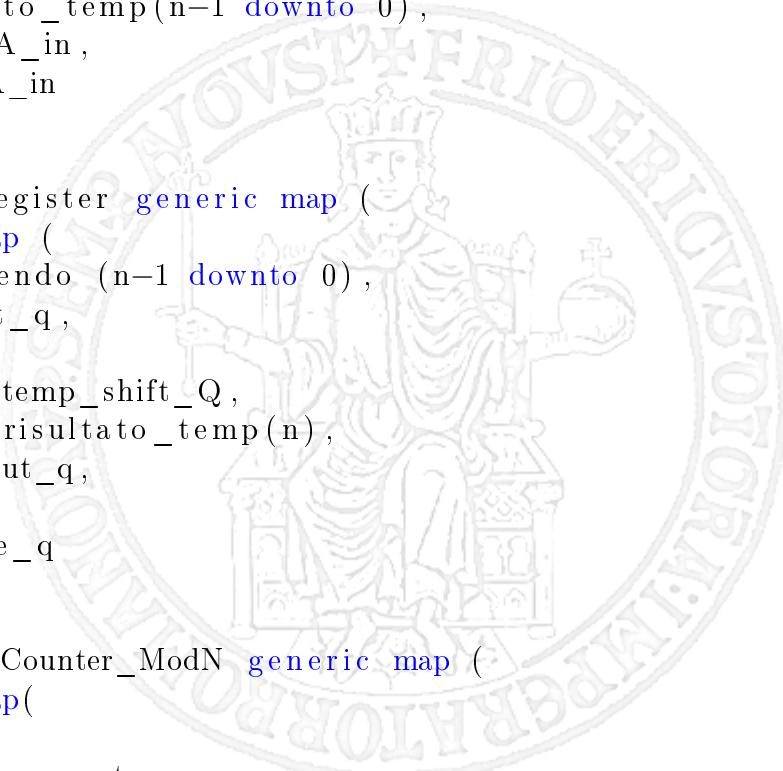
103
104 signal a, q, v, temp_A_in : std_logic_vector (n-1 downto 0);
105 signal operando1_temp, operando2_temp, risultato_temp :
106   std_logic_vector (n downto 0);
107 signal stop_temp, enable_v, enable_a, enable_s, enable_q, temp_shift_S,
108   temp_shift_A, temp_shift_Q, reset_a, reset_a_cu, reset_q,
109   reset_q_cu, reset_count, reset_count_cu, q_out_q, cnt_in, cnt_out:
110   std_logic;
111
112 begin
113
114 temp_s(0)<= risultato_temp(n);
115 s <= temp_s_q(0);
116 operando1_temp<=s & a;
117 operando2_temp<='0' & v;
118
119 reset_count <= reset_n and not reset_count_cu;
120 reset_a <= reset_n AND not reset_a_cu;
121 reset_q <= reset_n AND not reset_q_cu;
122
123 quoziante<=q;
124 resto<=a;
125 stop<=stop_temp;
126
127 reg_v: reg generic map (
128   width=>n) port map (
129     valore=>divisore,
130     clock=>clock,
131     enable=>enable_v,
132     reset_n=>reset_n,
133     output=>v
134   );
135
136
137 reg_s: shift_register generic map (n => 1) port map (
138   reg_in=>temp_s,
139   reset_n=>reset_n,
140   clk=>clock,

```

```

141      scan_enable=>temp_shift_S ,
142      scan_in=>s_in ,
143      scan_out=>open ,
144      q=>temp_s_q ,
145      enable=>enable_s
146    );
147
148
149 reg_a: Shift_Register generic map (
150   n=>n) port map (
151   reg_in=>temp_A_in ,
152   reset_n=>reset_a ,
153   clk=>clock ,
154   scan_enable=>temp_shift_A ,
155   scan_in=>q_out_q ,
156   scan_out=>s_in ,
157   Q=>a ,
158   enable=>enable_a
159 );
160
161 muxA : MUXN generic map(N=>N)
162   port map(
163     a => dividendo (2*n-1 downto n) ,
164     b => risultato_temp(n-1 downto 0) ,
165     sel => sel_A_in ,
166     o => temp_A_in
167   );
168
169 reg_q: Shift_Register generic map (
170   n=>n) port map (
171   reg_in=>dividendo (n-1 downto 0) ,
172   reset_n=>reset_q ,
173   clk=>clock ,
174   scan_enable=>temp_shift_Q ,
175   scan_in=>not risultato_temp(n) ,
176   scan_out=>q_out_q ,
177   Q=>q ,
178   enable=>enable_q
179 );
180
181 counter_mod_n: Counter_ModN generic map (
182   n=>n) port map(
183   clock=>clock ,
184   reset_n=>reset_count ,
185   enable=>cnt_in ,
186   counter=>open ,

```



```

187     res_out=>cnt_out
188 );
189
190
191 rca : RCAddSub
192   generic map ( num_bit=>n+1)
193   port map (
194     a=>operando1_temp,
195     b=>operando2_temp,
196     add_sub_n=>operazione,
197     s=>risultato_temp,
198     overflow=>open
199   );
200
201
202 cu: control_unit port map(
203   clock => clock,
204   start => start,
205   reset_n => reset_n,
206   counter_hit => cnt_out,
207   s=>risultato_temp(n),
208   div_zero => temp_div_zero,
209   stop => stop_temp,
210   reset_a => reset_a_cu,
211   reset_q => reset_q_cu,
212   reset_count => reset_count_cu,
213   en_a => enable_a,
214   en_q => enable_q,
215   en_v => enable_v,
216   en_s => enable_s,
217   operazione=>operazione,
218   count_up => cnt_in,
219   shift_A => temp_shift_A,
220   shift_Q => temp_shift_Q,
221   shift_S => temp_shift_S,
222   sel_A_in => sel_A_in
223 );
224
225
226 temp_div_zero<= '1' when divisore = (divisore'range => '0') else
227   '0';
228
229 — overflow se il resto maggiore o uguale al divisore
230 overflow<= '1' when unsigned(a) >= unsigned(divisore) and stop_temp =
231   '1' and divisore /= (divisore'range => '0') else '0';

```

```

232 |     div_zero <= temp_div_zero;
233 |
234 | end Structural;

```

Listing 11.2: DIVISORE RESTORING

11.1.2.3 Simulazione

Per testare l'architettura implementata si è deciso di utilizzare il tool ISE. Creando un opportuno testbench sono stati ricreati 3 casi di test, il primo mostra il funzionamento corretto, il secondo e il terzo rispettivamente mostrano gli stati di errore rispettivamente nel caso si vada in overflow oppure si tenti di effettuare una divisione per '0'.

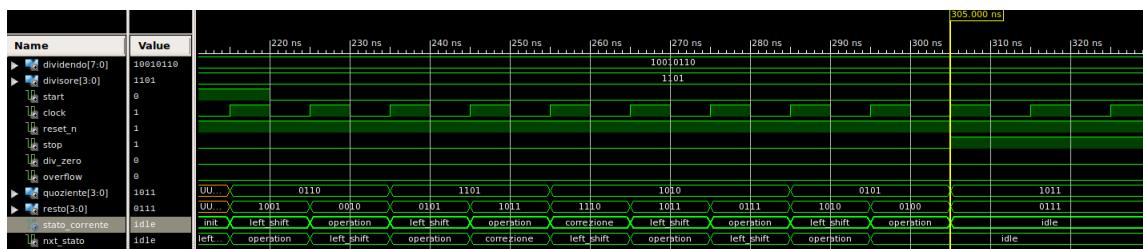


Figura 11.3: Simulazione corretta

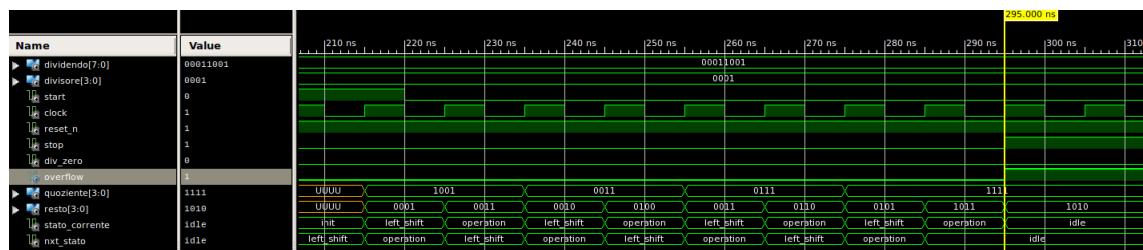


Figura 11.4: Simulazione overflow

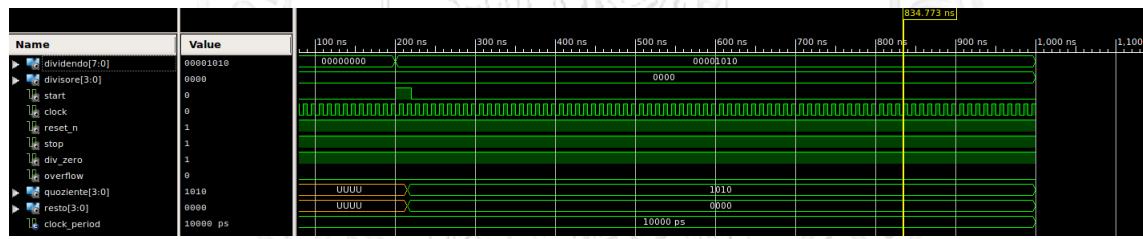


Figura 11.5: Simulazione divisione per zero

11.2 Divisore Non Restoring

11.2.1 Traccia

Realizzare in hardware l'algoritmo della divisione Non 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).

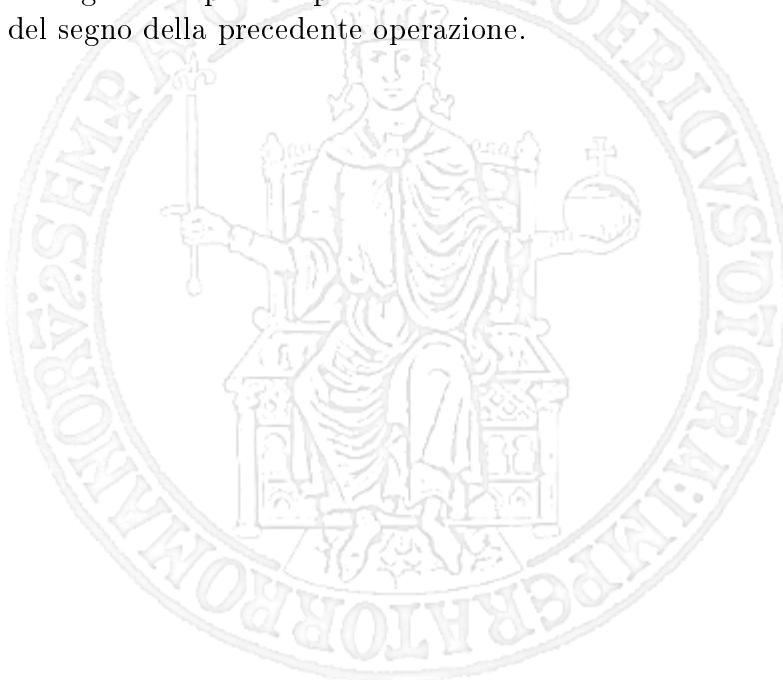
11.2.2 Soluzione

Per il divisore Non Restoring valgono le medesime considerazioni effettuate per il divisore restoring. La differenza principale consiste appunto nel fatto che questo divisore non prevede l'operazione di restoring ma un calcolo alternativo.

Infatti, ad una eventuale operazione di restoring $R_i = R_i + V$ seguirà sempre una sottrazione $R_{i+1} = 2R_i - V$ del passo successivo. Con una semplice sostituzione è facile intuire che le due operazioni possono essere fuse in una unica somma $R_{i+1} = 2R_i + V$. Ciò rende necessario prevedere esplicitamente per l'algoritmo la possibilità di avere risultati negativi per la sottrazione e quindi un bit per memorizzare il segno. Lo schematico della soluzione adottata risulta uguale a quella mostrata in precedenza, tuttavia viene ugualmente riportato in fig. per completezza.

11.2.2.1 Implementazione

I cambiamenti nell'implementazione riguardano principalmente il funzionamento della control unit. Come si può osservare dalla rappresentazione degli stati in fig. , la sequenza di operazioni da effettuare risulta sensibilmente diversa. Sostanzialmente come anticipato il passo di correzione non viene mai eseguito se non all'ultimo passo per correggere un eventuale resto negativo. Al contrario verrà sempre eseguito un passo “operazione” che consisterà in una somma oppure in una differenza a seconda del segno della precedente operazione.



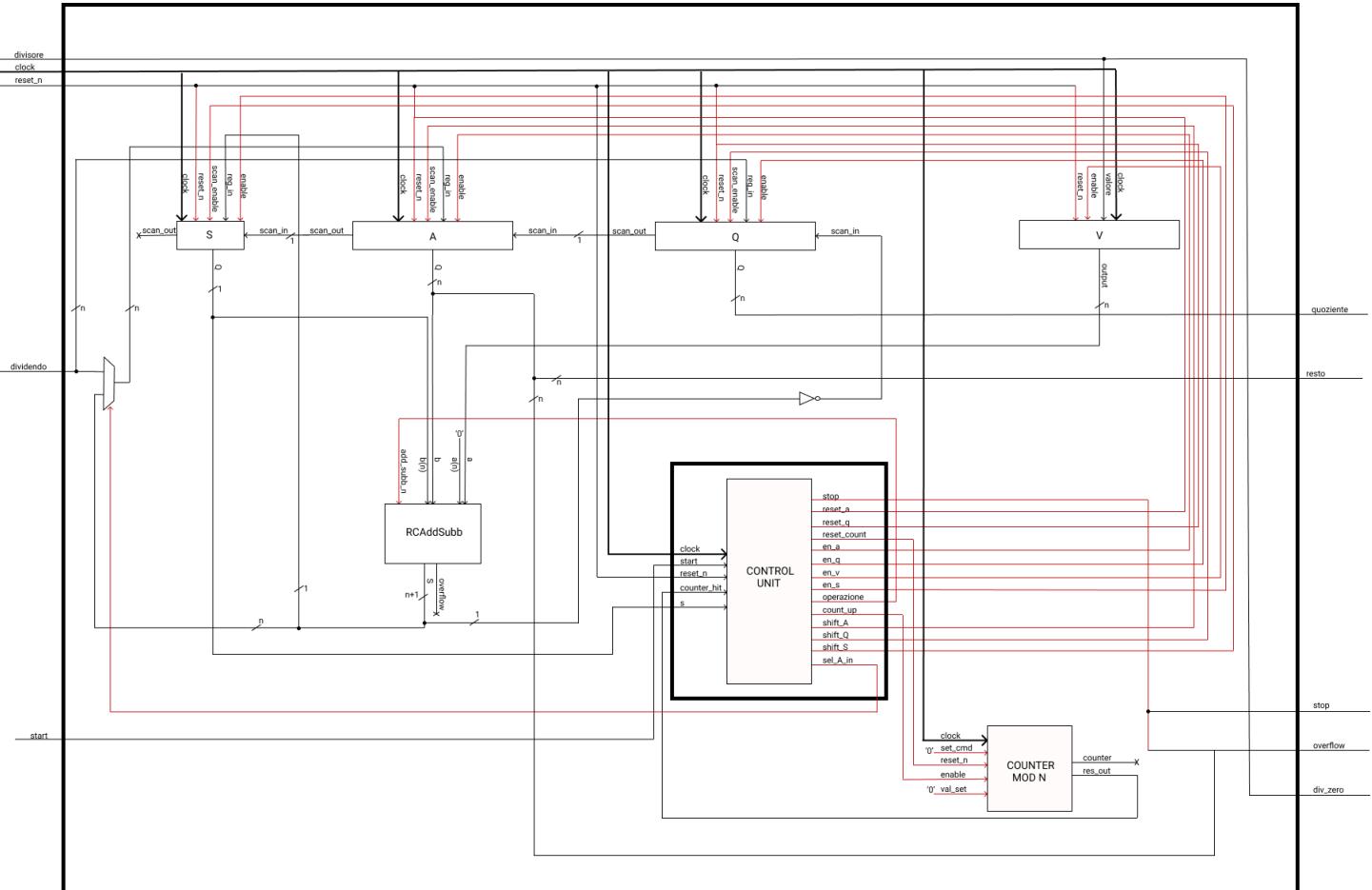


Figura 11.6: Schematico divisore

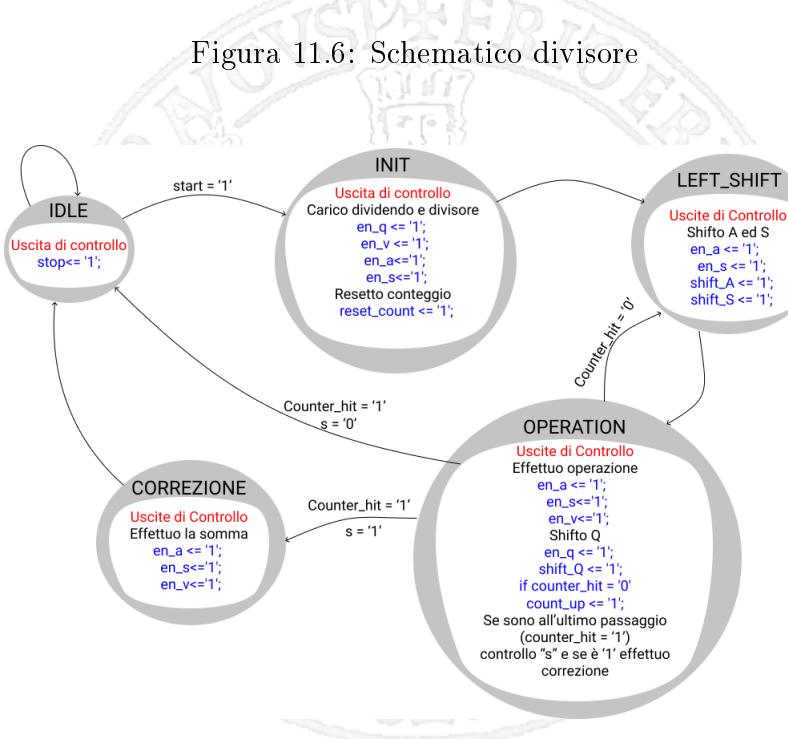


Figura 11.7: Funzionamento corretto CU

11.2.2.2 Codice

Come anticipato in precedenza i componenti sono gli stessi utilizzati nella versione Restoring eccetto per la control unit, sulla quale sono state apportate le modifiche maggiori. Per questo motivo si riporterà di seguito solo il codice relativo a questa entity. La differenza sostanziale è riscontrabile nel processo “op” che provvede a fornire in uscita il valore corretto del segnale “operazione”. Questo infatti, in ingresso all’adder permetterà di decidere se effettuare una somma oppure una sottrazione. Infine come detto in precedenza, il passo di correzione sarà eventualmente effettuato in ultima istanza.

```

1 library IEEE;
2 use IEEE.STD_LOGIC_1164.ALL;
3
4 -- Uncomment the following library declaration if using
5 -- arithmetic functions with Signed or Unsigned values
6 --use IEEE.NUMERIC_STD.ALL;
7
8 -- Uncomment the following library declaration if instantiating
9 -- any Xilinx primitives in this code.
10 library UNISIM;
11 --use UNISIM.VComponents.all;
12
13 entity control_unit is
14     Port ( clock : in STD_LOGIC;
15             start : in STD_LOGIC;
16             reset_n : in STD_LOGIC;
17             counter_hit : in STD_LOGIC;
18             s : in STD_LOGIC;
19             s_corr : in STD_LOGIC;
20             div_zero : in STD_LOGIC;
21             stop : out STD_LOGIC;
22             reset_a : out STD_LOGIC;
23             reset_q : out STD_LOGIC;
24             reset_count : out STD_LOGIC;
25             en_a : out STD_LOGIC;
26             en_q : out STD_LOGIC;
27             en_v : out STD_LOGIC;
28             en_s : out STD_LOGIC;
29             operazione: out STD_LOGIC;
30             count_up : out STD_LOGIC;
31             shift_A : out STD_LOGIC;
32             shift_Q : out STD_LOGIC;
33             shift_S : out STD_LOGIC;
34             sel_A_in : out STD_LOGIC
35         );
36
37 end control_unit;
```

```

38
39 architecture Behavioral of control_unit is
40
41 type state is (idle, init, operation, left_shift, correzione);
42
43 signal stato_corrente, nxt_stato : state := idle;
44
45 begin
46
47 state_proc: process(clock, reset_n)
48 begin
49   if reset_n = '0' then
50     stato_corrente <= idle;
51   elsif rising_edge(clock) then
52     stato_corrente <= nxt_stato;
53   end if;
54 end process;
55
56 op: process (clock, s)
57 begin
58   if (s='U') then
59     operazione <='1';
60   elsif (rising_edge(clock)) then
61     if (nxt_stato=operation) then
62       operazione<=not s; —se il segno negativo bisogna sommare,
63       viceversa sottrarre (tecnica non restoring)
64     elsif (nxt_stato=correzione) then
65       operazione<=not s_corr;
66     end if;
67   end if;
68 end process;
69
70 fsm: process(stato_corrente, start, reset_n, counter_hit, s_corr)
71 begin
72   stop <= '0';
73   reset_a <= '0';
74   reset_q <= '0';
75   reset_count <= '0';
76   en_a <= '0';
77   en_q <= '0';
78   en_v <= '0';
79   en_s <= '0';
80   count_up <= '0';
81   shift_A <= '0';
82   shift_Q <= '0';

```

```

83     shift_S <= '0';
84     sel_A_in <= '0';
85
86     case stato_corrente is
87         when idle =>
88             stop <= '1';
89             if start = '1' then
90                 nxt_stato <= init;
91             else
92                 nxt_stato <= idle;
93             end if;
94
95         when init =>
96             en_q <= '1';
97             en_v <= '1';
98             en_a <='1';
99             en_s <='1';
100            reset_count <= '1';
101            sel_A_in <= '1';
102            if div_zero = '1' then
103                stop <= '1';
104                nxt_stato<=idle;
105            else
106                nxt_stato <= left_shift;
107            end if;
108
109        when left_shift =>
110            en_a <= '1';
111            en_s <= '1';
112            shift_A <= '1';
113            shift_S <= '1';
114            nxt_stato <= operation;
115
116        when operation =>
117            en_a <= '1';
118            en_s <='1';
119            en_v <='1';
120            en_q <= '1';
121            shift_Q <= '1';
122
123            if counter_hit = '0' then
124                count_up <= '1';
125                nxt_stato <= left_shift;
126            else
127                if s_corr='1' then nxt_stato<=correzione;
128                else   nxt_stato <= idle;

```

```

129      end if;
130  end if;
131
132  when correzione =>
133    en_a <= '1';
134    en_s <= '1';
135    en_v <= '1';
136    nxt_stato <= idle;
137
138  end case;
139
140 end process;
141
142 end Behavioral;

```

Listing 11.3: CONTROL UNIT

11.2.2.3 Simulazione e confronto

Ancora una volta tramite il tool ISE è stata effettuata una simulazione che consente di mostrare il corretto funzionamento del dispositivo. In particolare, la prima simulazione mostra l'esecuzione con l'ultimo passo di correzione, la seconda il calcolo della divisione con i medesimi operandi del caso di test preso in considerazione per il divisore Restoring. Da questo confronto è possibile notare che il non restoring, a differenza del Restoring permette di non eseguire nessuna operazione di correzione, riducendo dunque i tempi di calcolo. Considerando dunque che il primo presenterà sempre al più un solo passo di correzione mentre il secondo potrebbe presentarne più di 1 (caso mostrato ha una sola correzione) possiamo concludere che il divisore Non Restoring risulta avere prestazioni sensibilmente migliori in termini di tempi di calcolo.

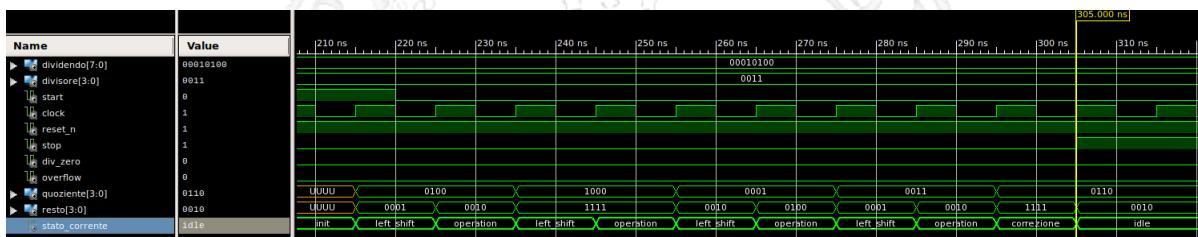


Figura 11.8: Simulazione con correzione

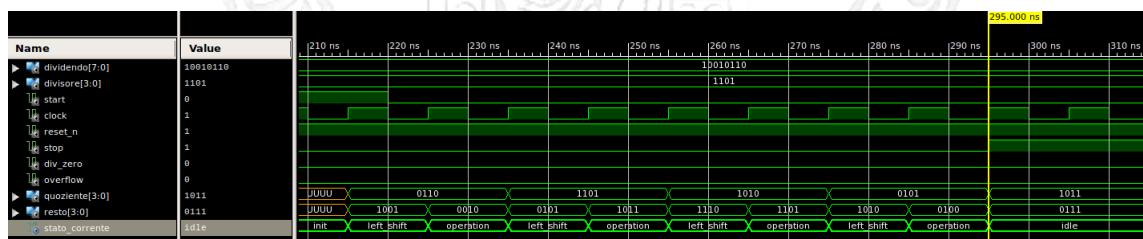


Figura 11.9: Simulazione confronto

Capitolo 12

Uart

12.1 UART

12.1.1 Traccia

Realizzare un dispositivo VHDL che implementa il protocollo UART (a partire da quello diffuso dalla Digilent). Collegare internamente, oppure tramite interfaccia fisica esterna alla board stessa, ad un'altra board oppure ad un PC previo utilizzo di un physical RS232, due interfacce per trasmettere e ricevere ottetti. Svolgere l'esercizio riutilizzando il VHDL messo a disposizione da Digilent (e disponibile nel materiale del corso) commentando eventuali ristrutturazioni del codice.

12.1.2 Soluzione

Quando si parla di UART ci si riferisce ad un dispositivo di comunicazione seriale. La trasmissione seriale consente di effettuare lo scambio dati tra due entità, una trasmittente e una ricevente, utilizzando un mezzo trasmissivo idoneo a trasferire la sequenza di segnali associati al dato da trasmettere. In particolare l'acronimo sta per “Universal Asynchronous Receiver and Transmitter”, ciò implica che è limitato alla sola trasmissione asincrona. Questo dispositivo è utilizzabile tramite una interfaccia “RS232” e provvede in autonomia alla serializzazione/parallelizzazione del dato rispettivamente in trasmissione/ricezione.

Una trattazione specifica sulla architettura dell'UART esula dagli scopi di questa esercitazione e per questo ci limiteremo a mostrare in fig. , ai fini della comprensione del capitolo, il particolare registro utilizzato per la trasmissione e la ricezione dei dati. Tale registro in fase di trasmissione riceve il dato in parallelo e lo fornisce sulla linea di uscita in seriale attuando degli shift, il discorso è duale nel caso della ricezione. In particolare viene mostrata una semplice soluzione per la sincronizzazione che consiste nel dotare il trasmettitore e il ricevitore di un unico clock comune, in modo da sincronizzare la serializzazione dei bit in trasmissione sul fronte di salita del clock e il campionamento dei bit in ricezione sul fronte di discesa.

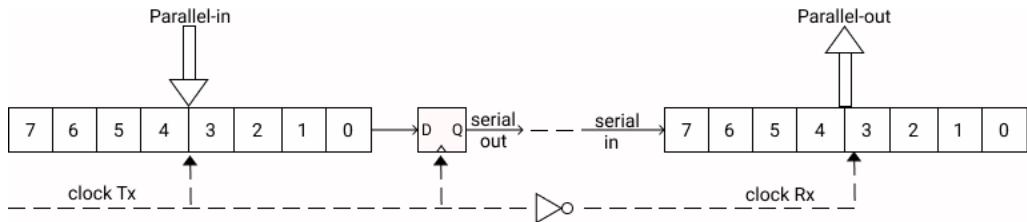


Figura 12.1: Registro

Nella comunicazione asincrona (nota come “start-stop”) la trasmissione dati è effettuata carattere per carattere di un numero prestabilito di bit. Ai bit attribuiti al carattere saranno aggiunti altri adibiti alla sincronizzazione e al controllo della parità. Un esempio di trasmissione è illustrata in fig. .

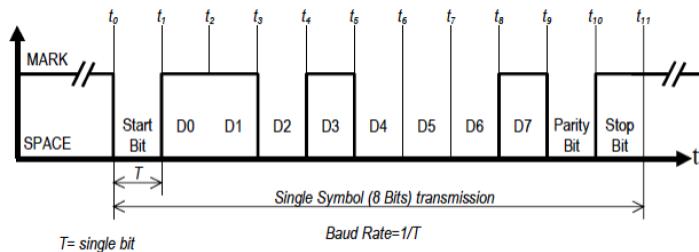


Figura 12.2: Trasmissione asincrona

12.1.2.1 Implementazione

Per questa esercitazione si è deciso di sintetizzare su una unica board due componenti messi in comunicazione tramite due pin. In particolare ognuno avrà a disposizione:

- una entity “Rs232RefComp” che implementa il protocollo UART. In trasmissione sarà caricato sul bus di ingresso il dato da inviare, e dopo opportune segnalazioni di controllo provvederà a serializzarlo e porlo sulla linea di uscita. Il discorso è duale per la ricezione.
- una entity “registro” sul quale andrà caricato il dato in ricezione, prelevandolo dal bus di uscita della entity precedente.

In aggiunta a questi componenti è stata istanziata una entity per la gestione del display a 7 segmenti, in modo da poter visualizzare i dati ricevuti dai due componenti. A tal proposito si è deciso di assegnare rispettivamente 2 delle 4 cifre del display ad ognuno dei due componenti. Il dato da trasmettere sarà dato dalla posizione degli switch e la trasmissione partì soltanto dopo la pressione del rispettivo bottone (uno per ogni dispositivo).

Il funzionamento complessivo del sistema è mostrato tramite una vista ad alto livello (non di dettaglio) dell’architettura in fig. .

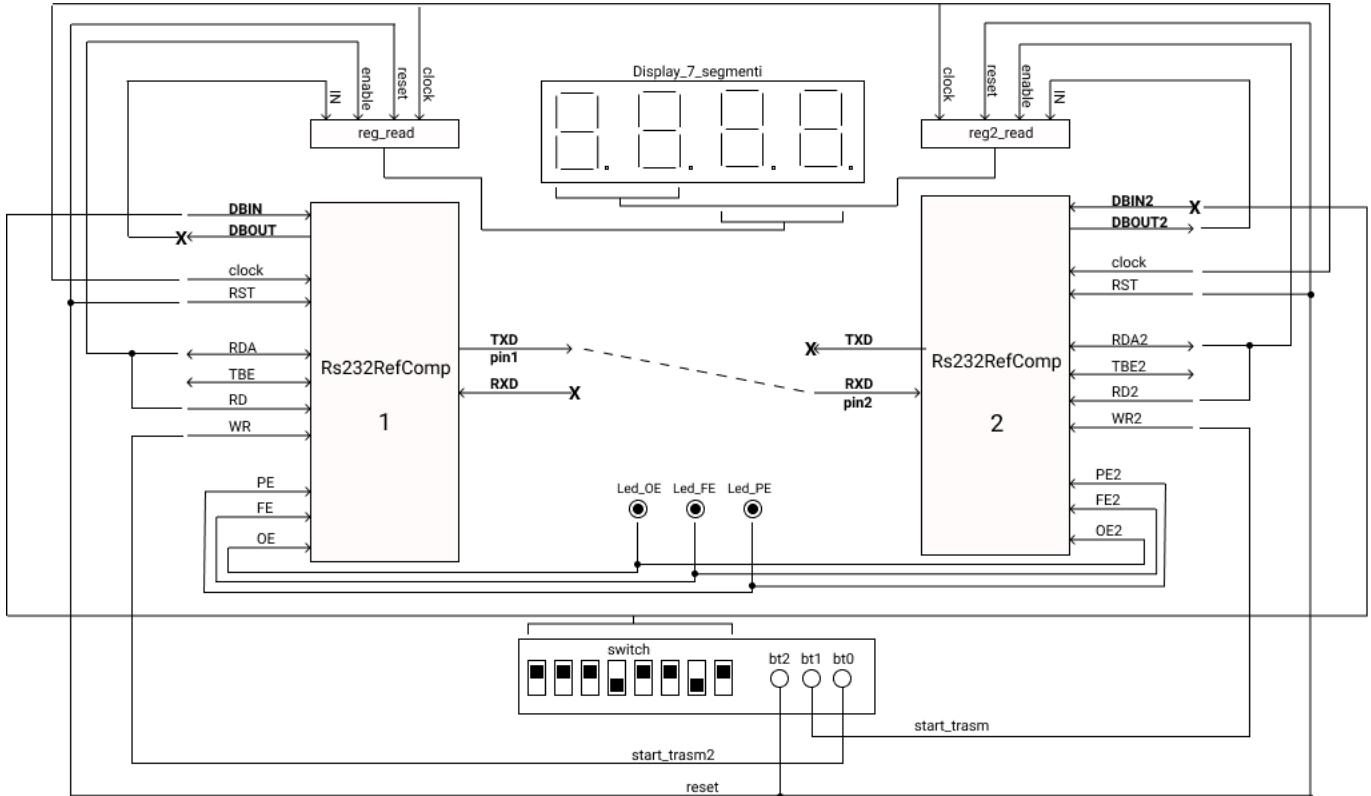


Figura 12.3: Architettura

12.1.2.2 Codice

Quanto detto nei paragrafi precedenti è stato scritto in codice VHDL. In particolare è stato riusato il modulo “Rs232RefComp” ed il componente registro, visibile negli esercizi precedenti. E’ stata quindi implementata una entity in grado di istanziare e connettere opportunamente tutti i componenti (mostrata nel list.). Infine è stato configurato il file .ucf in modo da consentire l’utilizzo di tutte le parti della board necessarie ad un corretto funzionamento (list.).

```

1  entity TOP_UART is port (
2    pin1 : inout std_logic; — Collegamento ingresso Uart —
3    pin2 : inout std_logic; —aggiunto per trasmissione
4    clock : in std_logic ;
5    reset : in std_logic ;
6    start_trasm: in std_logic; — Comando per avviare trasmissione —
7    start_trasm2: in std_logic; — Comando per avviare trasmissione —
8    switch : in STD_LOGIC_VECTOR(7 downto 0);
9    led_OE : out std_logic;
10   led_FE : out std_logic;
11   led_PE : out std_logic;
12   led_OE2 : out std_logic;
13   led_FE2 : out std_logic;
14   led_PE2 : out std_logic;
15   anodes : out STD_LOGIC_VECTOR (3 downto 0);

```

```

16      cathodes : out STD_LOGIC_VECTOR (7 downto 0)
17  );
18 end TOP_UART;
19
20 architecture Behavioral of TOP_UART is
21
22 Component display_seven_segments is
23 Generic(
24     clock_frequency_in : integer := 50000000;
25     clock_frequency_out : integer := 5000000
26 );
27 Port ( clock : in STD_LOGIC;
28         reset_n : in STD_LOGIC;
29         value : in STD_LOGIC_VECTOR (15 downto 0);
30         enable : in STD_LOGIC_VECTOR (3 downto 0);
31         dots : in STD_LOGIC_VECTOR (3 downto 0);
32         anodes : out STD_LOGIC_VECTOR (3 downto 0);
33         cathodes : out STD_LOGIC_VECTOR (7 downto 0));
34 end Component;
35
36 component Rs232RefComp is
37 Port (
38     TXD : out std_logic := '1';
39     RXD : in std_logic;
40     CLK : in std_logic; —Master Clock
41     DBIN : in std_logic_vector (7 downto 0); —Data Bus in (mi
42         serve per caricare i dati da trasmettere)
43     DBOUT : out std_logic_vector (7 downto 0); —Data Bus out (da cui
44         prelevo i dati ricevuti
45     RDA : inout std_logic;
46     TBE : inout std_logic := '1'; —Transfer Bus Empty
47     RD : in std_logic; —Read Strobe
48     WR : in std_logic; —Write Strobe
49     PE : out std_logic; —Parity Error Flag
50     FE : out std_logic; —Frame Error Flag
51     OE : out std_logic; —Overwrite Error Flag
52     RST : in std_logic := '0';
53 );
54 end component ;
55
56 component reg is
57 Generic(width : integer := 8);
58 Port ( valore : in STD_LOGIC_VECTOR (width-1 downto 0);
59         clock : in STD_LOGIC;
60         enable : in STD_LOGIC;
61         reset_n : in STD_LOGIC;

```

```

60         output : out STD_LOGIC_VECTOR (width-1 downto 0));
61 end component;

62
63 signal RDA,RDA2 : STD_LOGIC;
64 signal start : STD_LOGIC := '0' ;
65 signal TBE,TBE2 : STD_LOGIC ;
66 signal RD_TEMP,RD_TEMP2 : STD_LOGIC := '0' ;
67 signal WR_TEMP,WR_TEMP2 : STD_LOGIC := '0' ;
68 signal DBOUT,DBOUT2 : std_logic_vector (7 downto 0) := (Others => '0'
   ); —vettore di 8 bit dove carico valori ricevuti
69 signal temp_read,reg_read_out,reg2_read_out : std_logic_vector (7
   downto 0) := (Others => '0' );
70 signal value_display : std_logic_vector (15 downto 0) := (Others =>
   '0' );
71
72
73
74
75 type state_type is (init , lettura , scrittura ) ;
76 signal state , next_state : state_type := init;
77
78 begin
79
80 cambio_stato: process(clock , reset )
81 begin
82   if reset = '1' then
83     state <= init ;
84   elsif rising_edge(clock) then
85     state <= next_state;
86   end if;
87 end process;
88
89 prc_read : process (clock)
90 begin
91   if clock'event and clock = '1' then
92     if RDA = '1' then
93       RD_TEMP <= '1';
94     else
95       RD_TEMP <= '0';           — se ho disponibile da laggere il dato
96                               alzo RD_TEMP sarebbe il read_Strobe ()
97     end if ;
98   end if ;
99 end process;
100
101 prc_read2: process (clock , DBOUT, RDA, RD_TEMP)
102 begin

```

```

102  if clock'event and clock = '1' then
103    if RDA2 = '1' then
104      RD_TEMP2 <= '1';
105    else
106      RD_TEMP2 <= '0'; — se ho disponibile da laggere il dato
107      alzo RD_TEMP sarebbe il read_Strobe
108    end if ;
109  end if ;
110 end process;

111 prc_write : process (clock)
112 begin
113  if clock'event and clock = '1' then
114    if start_trasm = '1' then
115      WR_TEMP <= '1';
116    else
117      WR_TEMP <= '0';
118    end if ;
119  end if ;
120 end process;

121 prc_write2 : process (clock)
122 begin
123  if clock'event and clock = '1' then
124    if start_trasm2 = '1' then
125      WR_TEMP2 <= '1';
126    else
127      WR_TEMP2 <= '0';
128    end if ;
129  end if ;
130 end process;

132 reg_read: reg port map(
133   valore => DBOUT,
134   clock=> clock, —en_reg_read ,
135   enable=> RDA ,
136   reset_n=> not reset ,
137   output => reg_read_out); — da caricare sul display

139 reg2_read: reg port map(
140   valore => DBOUT2,
141   clock=> clock, —en_reg_read ,
142   enable=> RDA2 ,
143   reset_n=> not reset ,
144   output => reg2_read_out); — da caricare sul display
145
146

```

```

147
148
149 rs233 : Rs232RefComp
150     Port map (
151         TXD => pin1 ,
152         RXD => '0' ,
153         CLK => CLOCK,           —Master Clock
154         DBIN => switch ,      —Data Bus in
155         DBOUT => DBOUT ,      —Data Bus out
156         RDA => RDA ,          —Read Data Available
157         TBE => TBE ,          —Transfer Bus Empty
158         RD => RD_TEMP ,       —Read Strobe
159         WR => WR_TEMP ,       —Write Strobe
160         PE => led_PE ,        —Parity Error Flag _DISPARI_
161         FE => led_FE ,        —Frame Error Flag
162         OE => led_OE ,        —Overwrite Error Flag
163         RST => reset
164     );
165
166 rs2332 : Rs232RefComp
167     Port map (
168         TXD => open ,
169         RXD => pin2 ,
170         CLK => CLOCK,           —Master Clock
171         DBIN => "00000000" ,    —Data Bus in
172         DBOUT => DBOUT2 ,      —Data Bus out
173         RDA => RDA2 ,          —Read Data Available
174         TBE => TBE2,           —Transfer Bus Empty
175         RD => RD_TEMP2 ,       —Read Strobe
176         WR => WR_TEMP2 ,       —Write Strobe
177         PE => led_PE2 ,        —Parity Error Flag _DISPARI_
178         FE => led_FE2 ,        —Frame Error Flag
179         OE => led_OE2 ,        —Overwrite Error Flag
180         RST => reset
181     );
182
183
184 disp : display_seven_segments generic map ( clock_frequency_out => 400
185     )
186     port map (
187         clock => clock ,
188         reset_n => not reset ,
189         value => value_display ,
190         enable => "1111" ,
191         dots => "0000" ,
192         anodes => anodes ,

```

```

192         cathodes => cathodes
193     );
194
195 value_display <= reg2_read_out & reg_read_out;
196
197 end Behavioral;
```

Listing 12.1: Top UART

```

## clock pin for Basys rev E Board
NET "clock"      LOC = "P54"; # Bank = 2, Pin name = IO_L5N_2/D3/
GCLK31, Sch name = CLK1

## onBoard 7seg display
NET "cathodes<0>"    LOC = "P25"; # Bank = 3, Pin name = IO_L8P_3,
                           Sch name = CA
NET "cathodes<1>"    LOC = "P16"; # Bank = 3, Pin name = IO_L5P_3/
GCLK22, Sch name = CB
NET "cathodes<2>"    LOC = "P23"; # Bank = 3, Pin name = IO_L7N_3/
GCLK27, Sch name = CC
NET "cathodes<3>"    LOC = "P21"; # Bank = 3, Pin name = IO_L6N_3/
GCLK25, Sch name = CD
NET "cathodes<4>"    LOC = "P20"; # Bank = 3, Pin name = IO_L6P_3/
GCLK24/TRDY2, Sch name = CE
NET "cathodes<5>"    LOC = "P17"; # Bank = 3, Pin name = IO_L5N_3/
GCLK23/IRDY2, Sch name = CF
NET "cathodes<6>"    LOC = "P83"; # Bank = 1, Pin name = IO/VREF_1,
                           Sch name = CG
NET "cathodes<7>"    LOC = "P22"; # Bank = 3, Pin name = IO_L7P_3/
GCLK26, Sch name = DP
# anodo
NET "anodes<0>"     LOC = "P34"; # Bank = 3, Pin name = IO_L10P_3,
                           Sch name = AN1
NET "anodes<1>"     LOC = "P33"; # Bank = 3, Pin name = IO_L9N_3,
                           Sch name = AN2
NET "anodes<2>"     LOC = "P32"; # Bank = 3, Pin name = IO_L9P_3,
                           Sch name = AN3
NET "anodes<3>"     LOC = "P26"; # Bank = 3, Pin name = IO_L8N_3,
                           Sch name = AN4
#
#          0
## Leds
NET "led_OE2"        LOC = "P15"; # Bank = 3, Pin name = IO_L4N_3/GCLK21
                           Sch name = LD0
NET "led_FE2"        LOC = "P14"; # Bank = 3, Pin name = IO_L4P_3/GCLK20
                           Sch name = LD1
NET "led_PE2"        LOC = "P8";  # Bank = 3, Pin name = IO_L3N_3,
                           Sch name = LD2
```

```

23  NET "led_OE"      LOC = "P7"; # Bank = 3, Pin name = IO_L3P_3,
24          Sch name = LD3
25  NET "led_FE"      LOC = "P5"; # Bank = 3, Pin name = IO_L2N_3/VREF_3,
26          Sch name = LD4
27  NET "led_PE"      LOC = "P4"; # Bank = 3, Pin name = IO_L2P_3,
28          Sch name = LD5
29  #NET "led_exp_neg" LOC = "P3"; # Bank = 3, Pin name = IO_L1N_3,
30          Sch name = LD6
31  #NET "led_end_op"  LOC = "P2"; # Bank = 3, Pin name = IO_L1P_3,
32          Sch name = LD7
33  #
34  ## Switches
35  NET "switch<0>"  LOC = "P38"; # Bank = 2, Pin name = IP ,
36          Sch name = SW0
37  NET "switch<1>"  LOC = "P36"; # Bank = 3, Pin name = IP ,
38          Sch name = SW1
39  NET "switch<2>"  LOC = "P29"; # Bank = 3, Pin name = IO(3S100E)/IP
40          (3S250E),Sch name = SW2
41  NET "switch<3>"  LOC = "P24"; # Bank = 3, Pin name = IP
42          Sch name = SW3
43  NET "switch<4>"  LOC = "P18"; # Bank = 3, Pin name = IP ,
44          Sch name = SW4
45  NET "switch<5>"  LOC = "P12"; # Bank = 3, Pin name = IP/VREF_3,
46          Sch name = SW5
47  NET "switch<6>"  LOC = "P10"; # Bank = 3, Pin name = IO(3S100E)/IP
48          (3S250E),Sch name = SW6
49  NET "switch<7>"  LOC = "P6"; # Bank = 3, Pin name = IP ,
50          Sch name = SW7
51  #
52  ## Buttons
53  NET "start_trasm" LOC = "P69"; # Bank = 2, Pin name = IP ,
54          Sch name = BTN0
55  NET "start_trasm2" LOC = "P48"; # Bank = 2, Pin name = IP_L3N_2/
56          VREF_2           Sch name = BTN1
57  NET "reset"        LOC = "P47"; # Bank = 2, Pin name = IP_L3P_2,
58          Sch name = BTN2
59
60  ## 6 pin connectors
61  NET "pin1"         LOC = "P81"; # Bank = 1, Pin name = IO_L3P_1/A12 ,
62          Sch name = JA-1
63
64  NET "pin2"         LOC = "P92"; # Bank = 1, Pin name = IO_L6N_1/A5/GCLK9,
65          Sch name = JA-4

```

Listing 12.2: UCF

Capitolo 13

MIC

13.1 MIC 1

13.1.1 Traccia

1. Realizzare la sintesi di un'architettura MIC, di cui sono forniti i codici VHDL e gli strumenti di sviluppo, completando un ciclo di sviluppo. Effettuare, inoltre, la sostituzione del dispositivo per la comunicazione UART utilizzando, invece, led e switch delle board FPGA sfruttando le istruzioni di IN e OUT.

13.1.2 Introduzione

Il MIC 1 è una microarchitettura ovvero implementa la sua parte di controllo con un controllo microprogrammato ed è utilizzato per interpretare ed eseguire il livello ISA della IJVM.

Quindi ottenuto un codice operativo ISA IJVM il MIC1 sarà in grado di interpretarla ed eseguirla tramite un controllo microprogrammato e un Datapath.

Questi ultimi due saranno illustrati brevemente di seguito.

13.1.3 Datapath

Il datapath è la logica che ha a disposizione l'interprete per eseguire l'istruzione ISA.

I bus (B e C): - **il bus B** può leggere uno solo dei registri. Esso rappresenta l'operando destro dell'ALU.

- **il bus C** può essere scritto su 1 o più registri.

Si hanno a disposizione 9 registri da 32 bit: - **MAR e MDR** sono dei registri utilizzati per interfacciarsi con la memoria centrale insieme ai segnali di Read e di Write (saranno utilizzati per interfacciarsi anche con gli Switch e i Led)

- **PC e MBR** sono dei registri utilizzati per interfacciarsi con l'area programmi. Si osservi come il dato, ovvero il codice operativo ottenuto in ricezione dall'area programmi, sia di soli 8 bit, affinché possa essere trasferito sul bus B vi è bisogno di una estensione in termini di bit. Esso potrà essere esteso trattando il dato come se fosse rappresentato in modulo oppure come se fosse rappresentato in complementi a due e quindi dotato di segno. Nel primo caso vi è un'estensione

con tutti valori logici bassi, mentre nel secondo con un valore logico uguale a quello del bit più significativo degli 8 bit. La scelta è determinata da un segnale di controllo.

Il Datapath presenterà in uscita questi 8 bit affinchè possano essere utilizzati come ingressi dalla Control Unit.

PC quindi contiene un puntatore alla prossima istruzione da eseguire.

-SP, LV, CPP sono dei registri che durante l'esecuzione dei microprogrammi devono essere aggiornate affinchè contengano un puntatore rispettivamente uno alla cima dello stack, un altro alla base del blocco delle variabili locali, e infine CPP alla base della Memoria delle costanti.

- **H**: la sua uscita rappresenta l'unico operando sinistro dell'ALU.

L'ALU ha sei segnali di controllo: - F0 e F1 determinano l'operazione della ALU,

- EnA e ENB abilitano individualmente i due input,
- INVA inverte l'input di sinistra
- e INC forza la presenza di un riporto nel bit meno significativo, sommando quindi 1 al risultato

Lo Shifter gestisce l'output dell'ALU ed ha due linee di controllo: - SLL8 (Shift left logical, “scorrimento logico a sinistra”) trasla il valore a sinistra di un byte, impostando gli 8 bit meno significativi a zero

- SRA1 (Shift Right Arithmetic “scorrimento aritmetico a destra”) trasla invece il valore di 1 bit a destra, lasciando inalterato il bit più significativo.

13.1.4 Controllo Microprogrammato

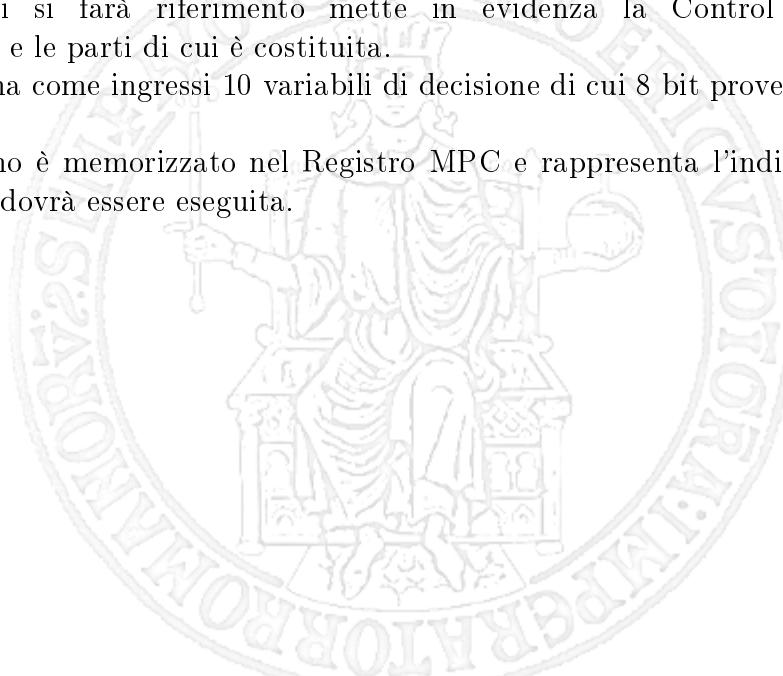
La control unit del Mic1 è stato implementata secondo la tecnica del controllo microprogrammato.

Di seguito sarà fatta un'analisi del tipo di controllo microprogrammato implementato nel MIC1.

La fig. a cui si farà riferimento mette in evidenza la Control Unit (il controllo microprogrammato) e le parti di cui è costituita.

La control unit ha come ingressi 10 variabili di decisione di cui 8 bit provengono dall'MBR e 2 dall'ALU.

Lo stato prossimo è memorizzato nel Registro MPC e rappresenta l'indirizzo della prossima microistruzione che dovrà essere eseguita.



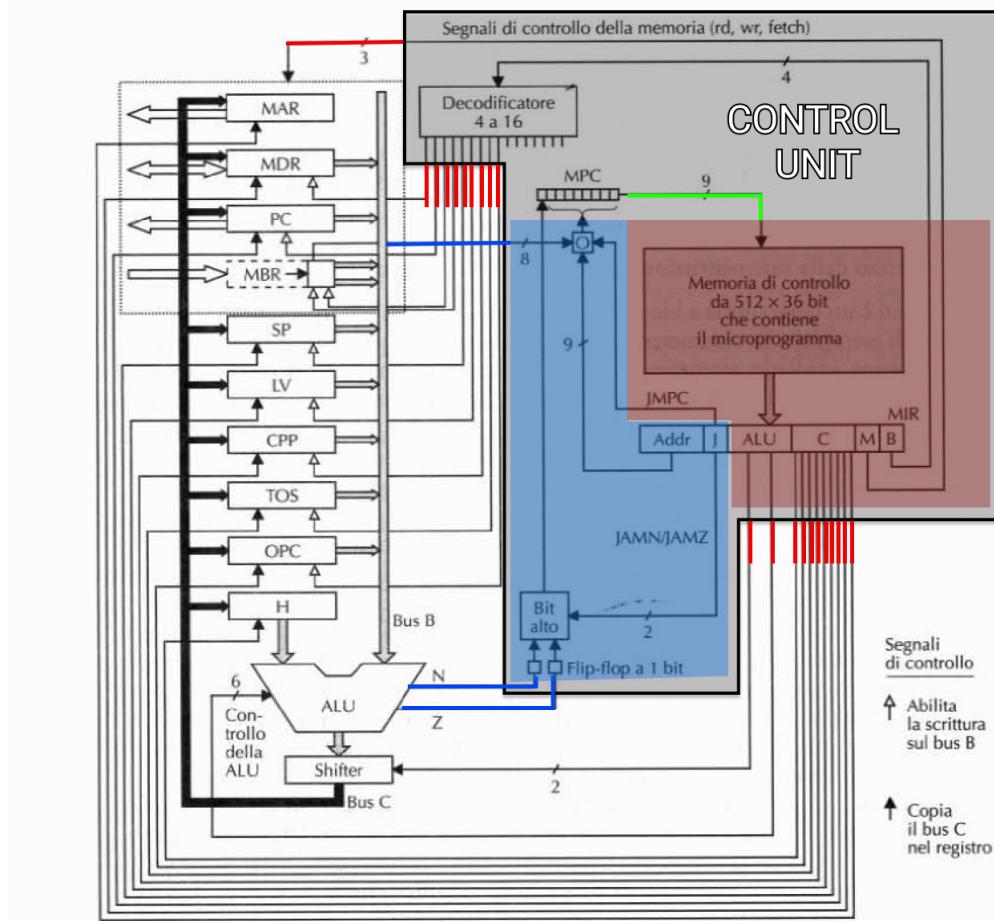


Figura 13.1: Rappresentazione della Microstruttura evidenziando il controllo microprogrammato a memorizzazione di Microistruzione (MM)

13.1.4.1 La funzione che calcola il Prossimo stato:

L'AREA ROSSA più l'AREA BLU si possono identificare nella logica che implementa la funzione che calcola il prossimo stato ovvero la prossima microistruzione da leggere; essa è in funzione delle variabili di decisione (gli ingressi rappresentati con le linee blu) e dello stato corrente (linea verde).

13.1.4.2 La funzione che calcola l'uscita (i segnali di controllo)

L'AREA ROSSA (da sola) presenta l'uscita ovvero i segnali di controllo; la funzione che calcola l'uscita è solo funzione dello stato corrente ovvero dell'indirizzo della microistruzione da leggere dalla memoria ROM.

13.1.4.3 Conclusioni

Si può concludere che questa control unit altro non fa che sintetizzare secondo la tecnica del controllo microprogrammato **mm (a memorizzazione di microistruzione)** una rete sequenziale di tipo Moore.

Da ciò è possibile derivare che il microlinguaggio utilizzato sarà sicuramente della **famiglia TS** (transfer structured: permette di fare trasferimenti condizionati).

Analizzando le variabili di condizione è possibile definire quante sono le condizioni che possono essere utilizzate nel linguaggio, ma bisogna ricordarsi che è possibile fare un'unica diramazione in una microistruzione (perchè le variabili di condizione al massimo possono alzare il bit più significativo dell' indirizzo della microistruzione quando è zero).

13.1.4.4 Microlinguaggio e MAL

Si è dedotto che il microlinguaggio deriva dalla famiglia TS limitandone la generalità nel modo seguente:

- uc | Oj <(C) un, (C') un OR 0x100> assumendo che un (next address è minore di 0xFF)

le condizioni possono essere molte e sono combinazioni delle variabili di decisione, però in una microistruzione, avendo a disposizione un'unica biforazione di trasferimento, potrà essere valutata una condizione e la sua negata.

Il MAL (Micro Assembler Language) è un linguaggio leggermente più astratto per esprimere Microprogrammi costituiti da Microistruzioni che rispettino il Microlinguaggio.

Per scrivere in MAL oltre a definire correttamente gli opportuni trasferimenti da una microistruzione all'altra, bisogna anche definire il microcordine, che dovrà essere impartito al datapath, quindi vi è bisogno di una buona conoscenza della parte operativa.

Infatti si vedrà di seguito che in un ciclo elementare è possibile fare più microoperazioni con un solo micrcordine (come lettura e scrittura su uno stesso registro)

13.1.4.5 Descrizione delle variabili di decisione: come determinare quale sarà la microistruzione successiva

Da quando MIR (Micro instruction Register) è stabile (dovrebbe essere all'inizio del ciclo, quindi sul fronte di discesa) inizia ad essere calcolato l'indirizzo della prossima microistruzione nel modo mostrato di seguito.

i primi 8 bit dell'indirizzo della prossima microistruzione:

se JMPC è impostato a 1 gli 8 bit di MBR sono collegati in OR con gli 8 bit meno significativi del campo NEXT_ADDRESS della microistruzione precedente.

In genere quando JMPC vale 1 gli 8 bit di NEXT ADDRESS valgono 0x000 o 0x100.

Tramite l'uso di JMPCsi fornisce un modo per definire l'indirizzo in cui trovare la successiva microistruzione da eseguire.

il bit più significativo (il nono) dell'indirizzo della prossima microistruzione:

Adesso se JAM = "000": MPC è quello appena calcolato

altrimenti se solo 1 bit di JAM è alto il bit più significativo di MPC sarà calcolato tramite la funzione logica mostrata di seguito:

$$F = (JAMZ \text{ AND } Z) \text{ OR } (JAMN \text{ AND } N) \text{ OR } \text{NEXT_ADDRESS}[8]$$

13.1.4.6 Descrizione dei Segnali di controllo

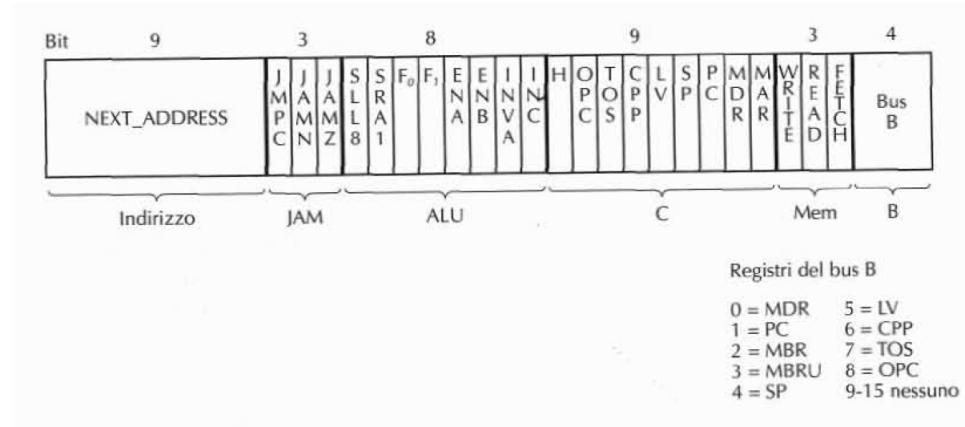


Figura 13.2: Microistruzione

13.1.5 Tempificazione

Tempificazione teorica del MIC 1 del tempo elementare ovvero di un periodo di clock:

- All'inizio del ciclo di un ciclo di clock (il fronte di discesa) la parola contenuta nella memoria di controllo e puntata da MPC viene trasferita in MIR. Il tempo necessario per effettuare questa operazione è indicato con ΔW

- subito dopo i vari segnali si propagano all'interno del percorso dati. Grazie a questi segnali il contenuto di un registro viene inserito nel bus B e la ALU sa quale operazione deve eseguire. Dopo $\Delta W + \Delta x$ dall'inizio del ciclo, gli input della ALU diventano stabili.

- dopo un Δy gli output della ALU diventano stabili
- dopo un ulteriore intervallo di tempo Δz , l'output dello shifter raggiunge i registri attraverso il bus C

I registri possono essere caricati quindi sul fronte di salita del clock. Tra i registri vi sono anche i flip flop N e Z che ritardano quindi il risultante valore corretto di MPC nel livello alto del clock.

Si è notato che non tutte le temporizzazioni mostrate teoricamente sono state implementate nel progetto ISE di Aiello, ad esempio i Flip Flop N e Z non ci sono e nemmeno il registro MIR, inoltre l'MPC è un registro che legge il valore in ingresso sul fronte di salita, al posto che durante il livello alto.

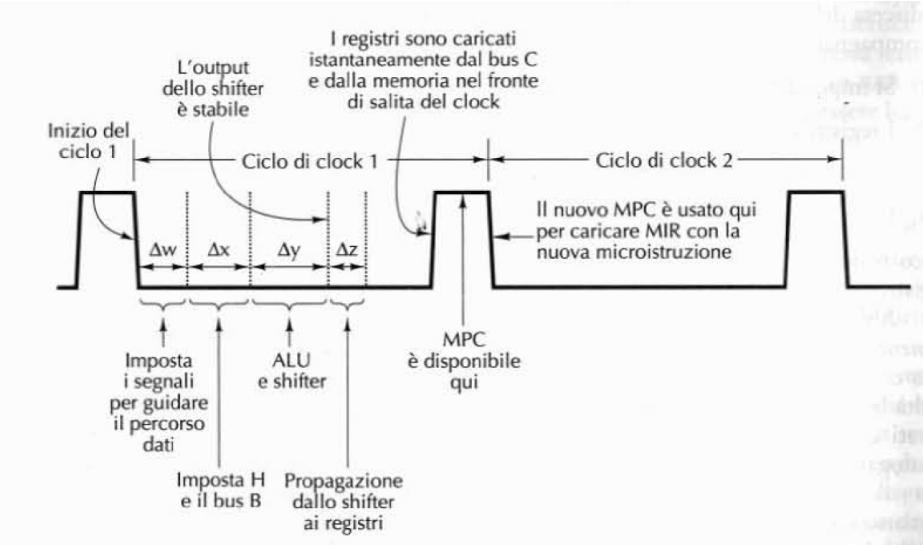


Figura 13.3: Temporizzazione di un ciclo di clock del percorso dati

13.1.6 Soluzione

13.1.6.1 Macroarchitettura

Di seguito è mostrata la macroarchitettura implementata da Aiello alla quale è stato sostituito il componente UART con il componente Switch_Led.

Il Mic 1 implementa la microarchitettura presentata nel paragrafo precedente.

La microarchitettura per interpretare ed eseguire sequenza di istruzioni ISA IJVM è affiancata da una Boot_Rom che implementa l'area programmi (byteaddressable), un'area di memoria per le costanti anch'essa una ROM (Word addressable) e una memoria RAM per le variabili locali (sempre word addressable). Questo insieme di componenti è chiamato macroarchitettura.

Inizialmente vi era anche il componente UART con il quale era possibile dare dei dati da 32 bit. Successivamente quest'ultimo è stato sostituito con il componente Switch_led in modo tale da impartire il dato da 32 bit con gli 8 Switches della Board (potranno essere definiti solo i primi 8 bit del dato, mentre i restanti 24 saranno sempre pari a '0').

In Fig. è mostrata la macroarchitettura avente il componente Switch_led.

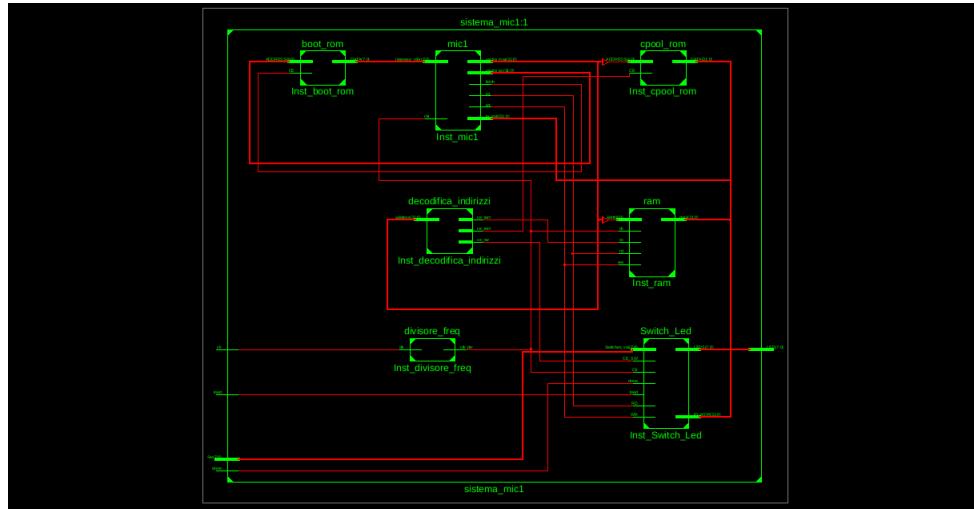


Figura 13.4: Macroarchitettura con il componente Switch_led sostituito

13.1.6.2 Decodifica indirizzi

La decodifica degli indirizzi funge un ruolo fondamentale nella macroarchitettura. Essa permette di mettere sullo stesso bus i vari elementi di memoria e anche il componente Switch_led, infatti garantisce che solo uno dei tre componenti sarà abilitato a scrivere sul bus (nel caso ovviamente di scrittura sul bus, ovvero di richiesta di lettura da parte del MIC).

Si può osservare che è stato assegnato al componente Switch_led un particolare indirizzo (x"FFFFFFF4"); questo indirizzo è generato automaticamente dalla operazione IJVM IN il quale verrà messo nel registro MAR con la seguente richiesta di lettura verso l'esterno.

A quel punto il decodificatore degli indirizzi alzerà un segnale di abilitazione che sarà intercettato dal componente Switch_led.

```

1 library IEEE;
2 use IEEE.STD_LOGIC_1164.ALL;
3 use IEEE.STD_LOGIC_ARITH.ALL;
4 use IEEE.STD_LOGIC_UNSIGNED.ALL;
5
6
7
8
9 entity decodifica_indirizzi is
10 port(
11   address:in std_logic_vector(31 downto 0);
12   ce_sw:out std_logic:='0';
13   ce_rom:out std_logic:='0';
14   ce_ram:out std_logic:='0';
15 );
16 end decodifica_indirizzi;
17
18 architecture Behavioral of decodifica_indirizzi is
19   signal app:std_logic_vector(2 downto 0):="000";

```

```

20 begin
21
22     app <= "010" when ( address>=x"00010000" and address<x"00020000") else
23         --Costant_pool_rom
24         "001" when ( address>=x"00020000" and address<x"FFFFFFFFFF4") else
25             --ram
26             "100" when ( address=x"FFFFFFFFFF4") else --Switch
27             "000";
28
29     ce_sw<=app(2);
30     ce_rom<=app(1);
31     ce_ram<=app(0);
32
33 end Behavioral;

```

Listing 13.1:

Quest'ultimo restituirà una stringa di tutti zeri nel caso in cui il Button 1 non è premuto, altrimenti restituirà una stringa con i primi 24 bits pari a '0' e gli 8 bits meno significativi pari agli 8 bit immessi con gli Switches.

Analogamente con l'operazione di Out verrà inoltrata dal Mic una richiesta di scrittura nell'indirizzo (x"FFFFFFFFFF4") che ancora una volta comporterà l'abilitazione dello Switch_led, il quale conserverà il dato in un registro per poi mostrarlo sui leds.

```

1 library IEEE;
2 use IEEE.STD_LOGIC_1164.ALL;
3
4 -- Uncomment the following library declaration if using
5 -- arithmetic functions with Signed or Unsigned values
6 --use IEEE.NUMERIC_STD.ALL;
7
8 -- Uncomment the following library declaration if instantiating
9 -- any Xilinx primitives in this code.
10 --library UNISIM;
11 --use UNISIM.VComponents.all;
12
13 entity Switch_Led is
14     Port (
15         CE_SW    : in std_logic := '0';
16         Ck : in STD_LOGIC;
17         Switches_val : in std_logic_vector(7 downto 0):= (others=>'0'); --
18             aggiungi alla toplevel entity
19         IO_MDR : inout std_logic_vector(31 downto 0) := "
20             ZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZ";
21         RD      : in std_logic := '0'; --SEGNALE DI LETTURA DAGLI SWITCH
22         load   : in Std_logic := '0';
23         done   : in std_logic := '0';
24         WR      : in std_logic := '0'; --segnale di scrittura SUI LED

```

```

23      LEDS : out std_logic_vector(7 downto 0) —aggiungi alla toplevel
24      entity;
25  end Switch_Led;
26
27 architecture Behavioral of Switch_Led is
28
29 signal reg_led : std_logic_vector(7 downto 0) := (others => '0');
30
31 begin
32
33 leggi_switches : process(ck)
34 begin
35     if falling_edge(ck) then
36         if wr='0' and rd ='1' and ce_sw = '1' and done = '1' then
37             io_mdr(7 downto 0) <= Switches_val;
38         elsif wr='0' and rd ='1' and ce_sw = '1' and done = '0' then
39             io_mdr<= (others =>'0');
40         else
41             io_mdr<= (others =>'Z');
42         end if;
43
44     end if;
45 end process;
46
47 scrivi_led : process(ck)
48 begin
49     if falling_edge(ck) then
50         if wr='1' and rd ='0' and ce_sw = '1' then
51             reg_led <= io_mdr(7 downto 0);
52         end if;
53
54     end if;
55 end process;
56
57     leds <= reg_led;
58
59 end Behavioral;

```

Listing 13.2: Componente Switch_led

13.1.6.3 Esempio di Programma

In Fig. vi è l'esempio di Programma costituito da 13 Istruzioni.

Come prima istruzione si ha IN (descritta brevemente nel paragrafo precedente).

Conseguentemente alla IN è stato caricato sullo Stack l'operando definito tramite gli Switches.

Nel caso in cui non dovesse andare a buon fine si caricherà un dato costituito da tutti valori bassi.

Con DUP viene duplicato il valore presente in cima allo stack e messo in cima allo stack.

BIPUSH inserisce 0x0 in cima allo stack.

Successivamente si verifica che i due operandi in cima allo stack siano uguali, ovvero se l'input prelevato dagli switches è tutti '0' (ovvero ha prelevato il valore quando il Button 1 non è stato alzato) si va in L2, si fa una POP e si ritorna ad IN.

Invece nel caso in cui il valore di input è diverso da zero, si prosegue il flusso del programma in modo sequenziale.

Si memorizza il valore prelevato in input nell'area delle variabili locali (politically correct per garantire un possibile riuso), si ricarica nello stack con ILOAD A, si preleva il secondo operando dall'area delle Costanti, e si effettua l'operazione di somma (si osservi come l'operazione non sia affiancata da operandi, operazione "a zero operandi").

L'istruzione di OUT effettua non solo una write (come descritto nel paragrafo precedente) ma anche una "pop" dallo stack.

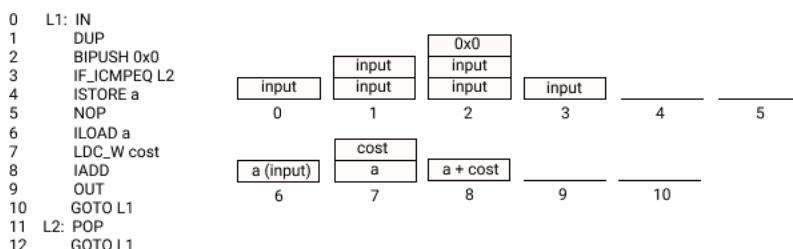


Figura 13.5: Macroarchitettura con il componente Switch_led sostituito

13.1.6.4 Ciclo di sviluppo

Esempio di programma Il programma descritto precedentemente è stato scritto in un file .jas, successivamente è stato assemblato per ottenere il .ijvm, in cui già abbiamo i codici operativi delle varie operazioni. Infine tramite il traduttore automatico del Prof. Mazzeo è stato possibile tradurre il formato ijvm in vhdl, pronto per essere inserito nell'area programmi.

Di seguito è riportato il risultato finale, ovvero il codice operativo inserito nell'area programmi.

```

1
2 library ieee;
3 use ieee.std_logic_1164.all;
4 use ieee.std_logic_arith.all;
5 use ieee.std_logic_unsigned.all;
6 use ieee.numeric_std.all;
7
8
9 library work;
10 -- use work.memory.all;
11
12 entity boot_rom is
13 port (

```

```

14 ADDRESS : in std_logic_vector(8 downto 0); —9 bit indirizzi =
15      512 locazioni
16 CE : in std_logic;
17 DATA : out std_logic_vector(7 downto 0)
18 );
19 end entity boot_rom;
20
21 architecture basic of boot_rom is
22 constant width : integer := 8; —larghezza word
23 constant memsize : integer := 512; —256 locazioni
24
25 —dichiarazione label della control store
26 constant BIPUSH :std_logic_vector(7 downto 0):= x"10";
27 constant DUP :std_logic_vector(7 downto 0):= x"59";
28 constant GOTO :std_logic_vector(7 downto 0):= x"A7";
29 constant IADD :std_logic_vector(7 downto 0):= x"60";
30 constant IAND :std_logic_vector(7 downto 0):= x"7E";
31 constant IFEQ :std_logic_vector(7 downto 0):= x"99";
32 constant IFLT :std_logic_vector(7 downto 0):= x"9B";
33 constant IF_ICMPEQ :std_logic_vector(7 downto 0):= x"9F";
34 constant IINC :std_logic_vector(7 downto 0):= x"84";
35 constant ILOAD :std_logic_vector(7 downto 0):= x"15";
36 constant INVOKEVIRTUAL :std_logic_vector(7 downto 0):= x"B6";
37 constant IOR :std_logic_vector(7 downto 0):= x"B0";
38 constant IRETURN :std_logic_vector(7 downto 0):= x"AC";
39 constant ISTORE :std_logic_vector(7 downto 0):= x"36";
40 constant ISUB :std_logic_vector(7 downto 0):= x"64";
41 constant LDC_W :std_logic_vector(7 downto 0):= x"13";
42 constant NOP :std_logic_vector(7 downto 0):= x"00";
43 constant POP :std_logic_vector(7 downto 0):= x"57";
44 constant SWAP :std_logic_vector(7 downto 0):= x"5F";
45 constant WIDE :std_logic_vector(7 downto 0):= x"C4";
46 constant HALT :std_logic_vector(7 downto 0):= x"FF";
47 constant ERR :std_logic_vector(7 downto 0):= x"FE";
48 constant OUTT :std_logic_vector(7 downto 0):= x"FD";
49 constant INN :std_logic_vector(7 downto 0):= x"FC";
50 constant IMUL :std_logic_vector(7 downto 0):= x"78";
51 constant IDIV :std_logic_vector(7 downto 0):= x"E0";
52
53
54
55 type rom_array is array(0 to memsize-1) of std_logic_vector(width-1
56      downto 0);
57

```

```

58      constant rom_data : rom_array :=  -----
59          (
60
61
62
63      X"FC",
64      X"59",
65      X"10",
66      X"00",
67      X"9F",
68      X"00",
69      X"10",
70      X"36",
71      X"00",
72      X"00",
73      X"15",
74      X"00",
75      X"13",
76      X"00",
77      X"00",
78      X"60",
79      X"FD",
80      X"A7",
81      X"FF",
82      X"EF",
83      X"57",
84      X"A7",
85      X"FF",
86      X"EB",
87
88      others=>(others=>'0')
89  );
90
91
92
93 begin
94
95     DATA <= rom_data(conv_integer(ADDRESS)) when CE = '1' else (others =>
96         'Z');
97 end architecture basic;

```

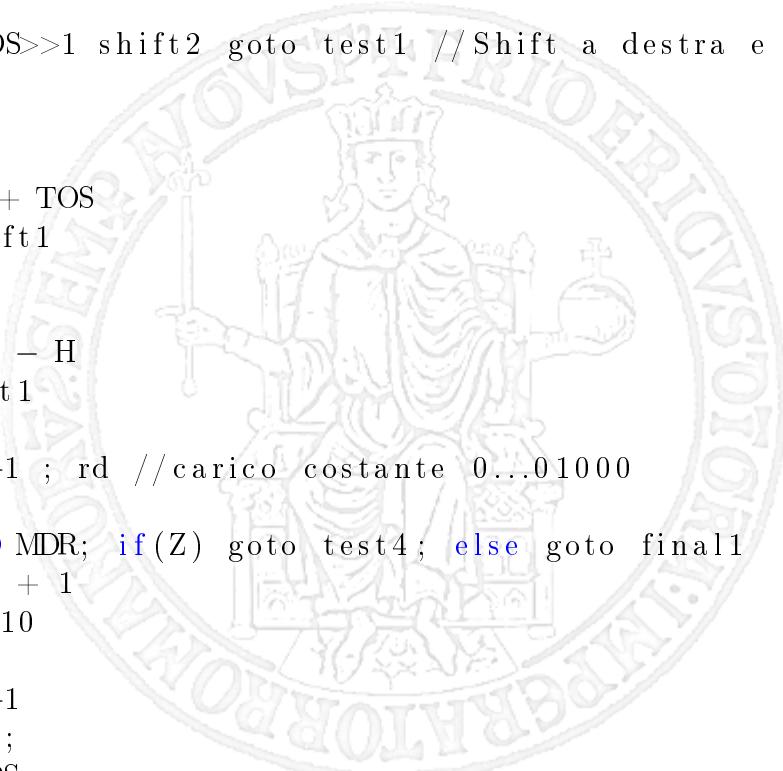
Listing 13.3: Boot_ROM

Esempio di microprogramma in linguaggio MAL: Booth

```

1 imul1 MAR = SP = SP-1;rd // carico AQ
2 imul2 MAR = SP = SP-1 // carico M
3 imul3 H = MDR; rd // carico AQ in H
4 imul4 H = H<<8 // shiftlo AQ in posiz corretta
5 imul5 TOS = H = H<<8
6 imul6 OPC = MDR<<8 // prendo M e shiftlo in posiz corretta
7 imul7 OPC = OPC<<8
8 imul8 OPC = OPC<<8
9 imul9 OPC = OPC + 1 // nella parte destra di OPC incrementiamo il
   conteggio
10           // registri occupati: TOS,H -> AQ ; OPC-> M
11
12 imul10 MAR = SP ; rd // carico stringa S ma non decremento SP; S ha
   valori alti solo nelle posizioni 24 e 25
13 imul11 H = TOS
14 imul12 H = H AND MDR; if (Z) goto shift1; else goto imul13 // Controllo
   coppia "00"
15 imul13 MDR = MDR>>1
16 imul14 Z = H AND MDR; if (Z) goto diff1; else goto imul15 // 
   Controllo coppia "10"
17 imul15 MDR = INV(MDR)
18 imul16 Z = H AND MDR; if (Z) goto somma1; else goto shift1 // Controllo
   coppia "01" altrimenti è "11"
19
20 shift1 TOS = TOS>>1 shift2 goto test1 // Shift a destra e si riplica l'
   ultimo bit
21
22 somma1 H = OPC
23 somma2 TOS = H + TOS
24 somma3 goto shift1
25
26 diff1 H = OPC
27 diff2 TOS = TOS - H
28 diff3 goto shift1
29
30 test1 MAR = SP-1 ; rd // carico costante 0...01000
31 test2 H = OPC
32 test3 Z = H AND MDR; if (Z) goto test4; else goto final1
33 test4 OPC = OPC + 1
34 test5 goto imul10
35
36 final1 SP = SP-1
37 final2 MAR = SP;
38 final3 MDR = TOS;wr
39 final4 goto Main1

```



Listing 13.4: Boot_ROM

