Tesina di Architettura dei Sistemi di Elaborazione
Gruppo 14

Gabriele Previtera - Mat. M63/000834 Mirko Pennone - Mat. M63/000858
Simone Penna - Mat. M63/000876

20 febbraio 2019

Indice

Capitolo 1

Sintesi di macchine combinatorie

Eseguire gli esercizi riportati nella traccia Esercitazioni/Esercizi proposti/Esercitazione macchine combinatorie.pdf nell'area FTP.

1.1 Traccia

Eseguire gli esercizi riportati nel documento fornito.

1.2 Soluzione

1.2.1 Esercizio 1

Si progetti una macchina M che, data una parola X di 6 bit in ingresso ($X_5X_4X_3X_2X_1X_0$), restituisca una parola Y di 3 bit ($Y_2Y_1Y_0$) che rappresenta la codifica binaria del **numero di bit alti in X.**

Utilizzando una rappresentazione 4-2-1 per l'uscita **Y**, si riportano gli ON-SET ottenuti per ogni uscita:

Y_2: ON-SET = {15, 23, 27, 29, 30, 31, 39, 43, 45, 46, 47, 51, 53, 54, 55, 57, 58, 59, 60, 61, 62, 63};
Y_1: ON-SET = {3, 5, 6, 7, 9, 10, 11, 12, 13, 14, 17, 18, 19, 20, 21, 22, 24, 25, 26, 28, 33, 34, 35, 36, 37, 38, 40, 41, 42, 44, 48, 52, 56, 63};
Y_0: ON-SET = {1, 2, 4, 7, 8, 11, 13, 14, 16, 19, 21, 22, 25, 26, 28, 31, 32, 35, 37, 38, 41, 42, 44, 47, 49, 50, 52, 55, 56, 59, 61, 62}.

1.2.2 Esercizio 2

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.

Per poter effettuare tale esercizio, rispettando i requisiti forniti, è stato necessario suddividere la descrizione delle tre uscite in di tre file blif separati. Tale operazione si è resa necessaria in quanto il comando *simplify* di SIS, nel procedere alla minimizzazione di una rete combinatoria, utilizza anche trasformazioni globali come ad esempio *substitute*.

Si riportano i risultati ottenuti con lo strumento SIS: si è stampata la funzione non semplificata usando *write_eqn* (fig.1.1) e poi si è proceduto alla minimizzazione tramite comando *simplify* delle uscite separate (fig.1.2).

```
sis> read_blif Esercitazione1_2.blif
sis> print_stats
esercizio1_2    pi= 6   po= 3   nodes=  3       latches= 0
lits(sop)= 528
sis> write_eqn
INORDER = X5 X4 X3 X2 X1 X0;
OUTORDER = Y2 Y1 Y0;
Y2 = X5*X4*X3*X2*!X1*!X0 + X5*X4*X3*!X2*X1*!X0 + X5*X4*!X3*X2*X1*!X0 + X5*!X4*
X3*X2*X1*!X0 + !X5*X4*X3*X2*X1*!X0 + X5*X4*X3*X2*X1*!X0 + X5*X4*X3*!X2*!X1*X0
 + X5*X4*!X3*X2*!X1*X0 + X5*!X4*X3*X2*!X1*X0 + !X5*X4*X3*X2*!X1*X0 + X5*X4*X3*
X2*!X1*X0 + X5*X4*!X3*!X2*X1*X0 + X5*!X4*X3*!X2*X1*X0 + !X5*X4*X3*!X2*X1*X0 +
X5*X4*X3*!X2*X1*X0 + X5*!X4*!X3*X2*X1*X0 + !X5*X4*!X3*X2*X1*X0 + X5*X4*!X3*X2*
X1*X0 + !X5*!X4*X3*X2*X1*X0 + X5*!X4*X3*X2*X1*X0 + !X5*X4*X3*X2*X1*X0 + X5*X4*
X3*X2*X1*X0;
Y1 = X5*X4*!X3*!X2*!X1*!X0 + X5*!X4*X3*!X2*!X1*!X0 + !X5*X4*X3*!X2*!X1*!X0 +
X5*X4*X3*!X2*!X1*!X0 + X5*!X4*!X3*X2*!X1*!X0 + !X5*X4*!X3*X2*!X1*!X0 + X5*X4*!
X3*X2*!X1*!X0 + !X5*!X4*X3*X2*!X1*!X0 + X5*!X4*X3*X2*!X1*!X0 + !X5*X4*X3*X2*!
X1*!X0 + X5*!X4*!X3*!X2*X1*!X0 + !X5*X4*!X3*!X2*X1*!X0 + !X5*!X4*X3*!X2*X1*!X0
 + X5*!X4*X3*!X2*X1*!X0 + !X5*X4*X3*!X2*X1*!X0 + !X5*!X4*X3*!X2*X1*!X0 + X5*!X4*!X3*!
X2*!X1*X0 + !X5*X4*!X3*!X2*!X1*X0 + !X5*!X4*X3*!X2*!X1*X0 + X5*!X4*X3*!X2*!X1*
X0 + !X5*X4*X3*!X2*!X1*X0 + !X5*!X4*!X3*X2*!X1*X0 + X5*!X4*!X3*X2*!X1*X0 + !X5*
X4*!X3*X2*!X1*X0 + !X5*!X4*X3*X2*!X1*X0 + !X5*!X4*!X3*!X2*X1*X0 + X5*!X4*!X3*!
X2*X1*X0 + !X5*X4*!X3*!X2*X1*X0 + !X5*!X4*X3*!X2*!X1*X0 + !X5*!X4*!X3*X2*X1*X0
 + X5*X4*X3*X2*X1*X0;
Y0 = X5*!X4*!X3*!X2*!X1*!X0 + !X5*X4*!X3*!X2*!X1*!X0 + !X5*!X4*X3*!X2*!X1*!X0
 + X5*X4*X3*!X2*!X1*!X0 + !X5*!X4*!X3*X2*!X1*!X0 + X5*X4*!X3*X2*!X1*!X0 + X5*!
X4*X3*X2*!X1*!X0 + !X5*X4*X3*X2*!X1*!X0 + !X5*!X4*!X3*!X2*X1*!X0 + X5*X4*!X3*!
X2*X1*!X0 + X5*!X4*X3*!X2*X1*!X0 + !X5*X4*X3*!X2*!X1*X0 + X5*!X4*!X3*!X2*!X1*X0
 + !X5*X4*!X3*X2*X1*!X0 + !X5*!X4*X3*X2*X1*!X0 + X5*X4*X3*X2*X1*!X0 + !X5*!X4*!
X3*!X2*!X1*X0 + X5*X4*!X3*!X2*!X1*X0 + X5*!X4*X3*!X2*!X1*X0 + !X5*X4*X3*!X2*!
X1*X0 + X5*!X4*!X3*X2*!X1*X0 + !X5*X4*!X3*X2*!X1*X0 + !X5*!X4*X3*X2*!X1*X0 +
X5*X4*X3*X2*!X1*X0 + X5*!X4*!X3*!X2*X1*X0 + !X5*X4*!X3*!X2*!X1*X0 + !X5*!X4*X3*!
X2*X1*X0 + X5*X4*X3*!X2*X1*X0 + !X5*!X4*!X3*X2*X1*X0 + X5*X4*!X3*X2*X1*X0 + X5*!
X4*X3*X2*X1*X0 + !X5*X4*X3*X2*X1*X0;
sis>
```

Figura 1.1: Stampa della funzione non semplificata in SIS.

Tra il comando *print_stats* è possibile osservare come i numero di letterali sia sceso da 132 a 60 per Y_2 e da 204 a 96 per Y_1. Per quanto riguarda Y_0, invece, lo strumento SIS non è stato in grado di ridurre il numero di letterali, dunque la funzione è già in forma minima.

Per quanto concerne la procedura di minimizzazione manuale, si è utilizzato il metodo di **Mc-Cluskey** sulle tre uscite separate. Confrontando le due soluzioni, si è notato che i numeri di letterali ottenuti in entrambi i casi sono coerenti tra loro. Si è inoltre notato come l'uscita Y_0 ancora una volta non sia stata alterata durante il processo di minimizzazione, a differenza delle altre due uscite che risultano notevolmente ridotte in entrambi i casi. Ciò è perfettamente compatibile con il risultato ottenibile attraverso Mc-Cluskey (fig.1.3), dove si nota chiaramente che per l'uscita Y_0 non vengono generati consensi: questo è dovuto al fatto che non ci siano classi adiacenti aventi distanza di Hamming pari a 1, pertanto è impossibile che vengano generati consensi.

```
UC Berkeley, SIS 1.3 (compiled 20-Mar-00 at 1:56 PM)
sis> read_blif Esercitazione1_2_Y2.blif
sis> simplify
sis> write_eqn
INORDER = X5 X4 X3 X2 X1 X0;
OUTORDER = Y2;
Y2 = X3*X2*X1*X0 + X4*X2*X1*X0 + X5*X2*X1*X0 + X4*X3*X1*X0 + X5*X3*X1*X0 + X5*
X4*X1*X0 + X4*X3*X2*X0 + X5*X3*X2*X0 + X5*X4*X2*X0 + X5*X4*X3*X0 + X4*X3*X2*X1
 + X5*X3*X2*X1 + X5*X4*X2*X1 + X5*X4*X3*X1 + X5*X4*X3*X2;
sis> read_blif Esercitazione1_2_Y1.blif
sis> simplify
sis> write_eqn
INORDER = X5 X4 X3 X2 X1 X0;
OUTORDER = Y1;
Y1 = X5*X4*X3*X2*X1*X0 + X5*X4*!X2*!X1*!X0 + X5*!X3*X2*!X1*!X0 + !X4*X3*X2*!X1*!
X0 + !X5*X3*X2*!X1*!X0 + !X4*!X3*X2*X1*!X0 + !X5*!X4*X3*X1*!X0 + X5*!X4*X3*!X2*!
X0 + !X5*X4*X3*!X2*!X0 + !X5*X4*!X3*X2*!X0 + X5*!X4*!X2*!X1*X0 + !X5*X4*!X2*!
X1*X0 + !X4*!X3*X2*!X1*X0 + !X5*!X3*X2*!X1*X0 + !X5*!X4*X3*!X1*X0 + !X5*!X4*!
X2*X1*X0 + !X5*!X4*!X3*X1*X0 + X5*!X4*!X3*!X2*X1 + !X5*X4*!X3*!X2*X1;
sis> read_blif Esercitazione1_2_Y0.blif
sis> simplify
sis> write_eqn
INORDER = X5 X4 X3 X2 X1 X0;
OUTORDER = Y0;
Y0 = X5*!X4*!X3*!X2*!X1*!X0 + !X5*X4*!X3*!X2*!X1*!X0 + !X5*!X4*X3*!X2*!X1*!X0
 + X5*X4*X3*!X2*!X1*!X0 + !X5*!X4*!X3*X2*!X1*!X0 + X5*X4*!X3*X2*!X1*!X0 + X5*!
X4*X3*X2*!X1*!X0 + !X5*X4*X3*X2*!X1*!X0 + !X5*!X4*!X3*!X2*X1*!X0 + X5*X4*!X3*!
X2*X1*!X0 + X5*!X4*X3*!X2*X1*!X0 + !X5*X4*X3*!X2*X1*!X0 + X5*!X4*!X3*X2*X1*!X0
 + !X5*X4*!X3*X2*X1*!X0 + !X5*!X4*X3*X2*X1*!X0 + X5*X4*X3*X2*X1*!X0 + !X5*!X4*!
X3*!X2*!X1*X0 + X5*X4*!X3*!X2*!X1*X0 + X5*!X4*X3*!X2*!X1*X0 + !X5*X4*X3*!X2*!
X1*X0 + X5*!X4*!X3*X2*!X1*X0 + !X5*X4*!X3*X2*!X1*X0 + !X5*!X4*X3*X2*!X1*X0 +
X5*X4*X3*X2*!X1*X0 + X5*!X4*!X3*!X2*X1*X0 + !X5*X4*!X3*!X2*X1*X0 + !X5*!X4*X3*!
X2*X1*X0 + X5*X4*X3*!X2*X1*X0 + !X5*!X4*!X3*X2*X1*X0 + X5*X4*!X3*X2*X1*X0 + X5*!
X4*X3*X2*X1*X0 + !X5*X4*X3*X2*X1*X0;
sis>
```

Figura 1.2: Stampa della uscite semplificate separatamente in SIS.

	x_5	x_4	x_3	x_2	x_1	x_0	
1:	0	0	0	0	0	1	✓
2:	0	0	0	0	1	0	✓
4:	0	0	0	1	0	0	✓
8:	0	0	1	0	0	0	✓
16:	0	1	0	0	0	0	✓
32:	1	0	0	0	0	0	✓
7:	0	0	0	1	1	1	✓
11:	0	0	1	0	1	1	✓
13:	0	0	1	1	0	1	✓
14:	0	0	1	1	1	0	✓
19:	0	1	0	0	1	1	✓
21:	0	1	0	1	0	1	✓
22:	0	1	0	1	1	0	✓
25:	0	1	1	0	0	1	✓
26:	0	1	1	0	1	0	✓
28:	0	1	1	1	0	0	✓
35:	1	0	0	0	1	1	✓
37:	1	0	0	1	0	1	✓
38:	1	0	0	1	1	0	✓
41:	1	0	1	0	0	1	✓
42:	1	0	1	0	1	0	✓
44:	1	0	1	1	0	0	✓
49:	1	1	0	0	0	1	✓
50:	1	1	0	0	1	0	✓
52:	1	1	0	1	0	0	✓
56:	1	1	1	0	0	0	✓
31:	0	1	1	1	1	1	✓
47:	1	0	1	1	1	1	✓
55:	1	1	0	1	1	1	✓
59:	1	1	1	0	1	1	✓
61:	1	1	1	1	0	1	✓
62:	1	1	1	1	1	0	✓

Figura 1.3: Minimizzazione di Y_0 con Mc-Cluskey.

Per il mapping tecnologico, si è utilizzata la libreria *mcnc.genlib*, contenente le caratteristiche di ogni porta in termini di equazioni, area e ritardi. Come riportato in fig.1.4, sono stati effettuati diversi esperimenti variando la funzione di costo rispetto alla quale viene ottimizzata in base alla tecnologia scelta per il mapping. Ciò è stato fatto utilizzando l'opzione –**m** del comando **map**: in particolare, con -m 1 si è preferito ottimizzare il ritardo, con -m 0 l'area, mentre con -m 0.5 si è effettuata un mapping più bilanciato.

```
sis> read_blif Esercitazione1_2.blif
sis> read_library ~/sis-1.3/sis/sis_lib/mcnc.genlib
sis> map -W -m 1 -s                          sis> map -W -m 0 -s                          sis> map -W -m 0.5 -s
>>> before removing serial inverters <<<     >>> before removing serial inverters <<<     >>> before removing serial inverters <<<
# of outputs:          3                     # of outputs:          3                     # of outputs:          3
total gate area:       244.00                total gate area:       161.00                total gate area:       223.00
maximum arrival time: (16.41,16.41)          maximum arrival time: (21.50,21.50)          maximum arrival time: (19.10,19.10)
maximum po slack:     (-13.21,-13.21)        maximum po slack:     (-14.30,-14.30)        maximum po slack:     (-15.50,-15.50)
minimum po slack:     (-16.41,-16.41)        minimum po slack:     (-21.50,-21.50)        minimum po slack:     (-19.10,-19.10)
total neg slack:      (-43.76,-43.76)        total neg slack:      (-53.20,-53.20)        total neg slack:      (-50.10,-50.10)
# of failing outputs:  3                     # of failing outputs:  3                     # of failing outputs:  3
>>> before removing parallel inverters <<<  >>> before removing parallel inverters <<<  >>> before removing parallel inverters <<<
# of outputs:          3                     # of outputs:          3                     # of outputs:          3
total gate area:       196.00                total gate area:       161.00                total gate area:       169.00
maximum arrival time: (15.51,15.51)          maximum arrival time: (21.50,21.50)          maximum arrival time: (17.90,17.90)
maximum po slack:     (-12.31,-12.31)        maximum po slack:     (-14.30,-14.30)        maximum po slack:     (-14.30,-14.30)
minimum po slack:     (-15.51,-15.51)        minimum po slack:     (-21.50,-21.50)        minimum po slack:     (-17.90,-17.90)
total neg slack:      (-41.13,-41.13)        total neg slack:      (-53.20,-53.20)        total neg slack:      (-46.50,-46.50)
# of failing outputs:  3                     # of failing outputs:  3                     # of failing outputs:  3
# of outputs:          3                     # of outputs:          3                     # of outputs:          3
total gate area:       196.00                total gate area:       159.00                total gate area:       169.00
maximum arrival time: (15.51,15.51)          maximum arrival time: (21.30,21.30)          maximum arrival time: (17.90,17.90)
maximum po slack:     (-12.31,-12.31)        maximum po slack:     (-14.30,-14.30)        maximum po slack:     (-14.30,-14.30)
minimum po slack:     (-15.51,-15.51)        minimum po slack:     (-21.30,-21.30)        minimum po slack:     (-17.90,-17.90)
total neg slack:      (-41.13,-41.13)        total neg slack:      (-53.00,-53.00)        total neg slack:      (-46.50,-46.50)
# of failing outputs:  3                     # of failing outputs:  3                     # of failing outputs:  3
sis> print_map_stats                         sis> print_map_stats                         sis> print_map_stats
Total Area              = 196.00             Total Area              = 159.00             Total Area              = 169.00
Gate Count              = 72                 Gate Count              = 52                 Gate Count              = 58
Buffer Count            = 0                  Buffer Count            = 0                  Buffer Count            = 0
Inverter Count          = 6                  Inverter Count          = 9                  Inverter Count          = 10
Most Negative Slack     = -15.51             Most Negative Slack     = -21.30             Most Negative Slack     = -17.90
Sum of Negative Slacks  = -41.13             Sum of Negative Slacks  = -53.00             Sum of Negative Slacks  = -46.50
Number of Critical PO   = 3                  Number of Critical PO   = 3                  Number of Critical PO   = 3
```

Figura 1.4: Mapping tecnologico effettuato tramite libreria *mcnc.genlib* fornendo come parametri di bilanciamento, rispettivamente, 1, 0 e 0.5.

I risultati ottenuti sono perfettamente coerenti con quanto stabilito: nel primo caso, ottimizzando il ritardo, lo slack negativo totale è di -41.13, ma l'area totale risulta essere 196. Nel secondo caso invece, ottimizzando l'area, questa risulta essere scesa a 159, ma lo slack negativo totale raggiunge -53.00. Nel terzo caso infine, dove si è scelta una mediazione tra tempo e area, si è ottenuta una rete la cui area e slack negativo totale si assestano ad un valore "intermedio" rispetto ai due casi precedenti: in particolare, la rete avrà un'area di 169 e uno slack negativo totale pari a -46.50.

1.2.3 Esercizio 3

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

Per effettuare quest'operazione è stato possibile scegliere tra i diversi algoritmi visti a lezione in grado di minimizzare una funzione a due livelli multiuscita fornendoci una funzione a due livelli o multilivello. Si è deciso di utilizzare lo script *rugged.script*, in grado di operare sia su funzioni multilivello che a due livelli applicando una serie di trasformazioni prestabilite e fornendo, in uscita, una funzione multilivello che ben si presta alla rappresentazione grafica mediante grafo. In

fig.1.5 è possibile osservare il risultato.

```
sis> write_eqn
INORDER = X5 X4 X3 X2 X1 X0;
OUTORDER = Y2 Y1 Y0;
Y2 = X5*!Y0*[10] + X4*[8]*[9] + X4*[6]*[7] + [7]*[9];
Y1 = X4*![3]*![6]*![10] + X4*![3]*[9]*[10] + !Y2*!Y0*[8] + !Y2*[3]*[6] + !Y2*
[10] + !Y2*[9];
Y0 = ![5]*[6]*![9] + [5]*[9] + [5]*![6];
[3] = X5*!X4 + !X5*X4;
[4] = !X3*![3] + X3*[3];
[5] = !X2*![4] + X2*[4];
[6] = X0 + X1;
[7] = X5*!Y0 + [10];
[8] = X2 + X3;
[9] = X1*X0;
[10] = X3*X2;
esercizio1_2    pi= 6   po= 3   nodes= 11       latches= 0
lits(sop)=  59
sis>
```

Figura 1.5: Risultato della minimizzazione con *rugged.script*.

Si noti come minimizzando tutte le uscite contemporaneamente, e dunque grazie al riutilizzo di alcuni dei nodi della rete per la realizzazione di più uscite, il numero totale di letterali sia sceso a 59, mentre nel caso della minimizzazione delle uscite separate si erano ottenuti 60, 96 e 192 letterali rispettivamente per Y_2, Y_1 e Y_0.

Il grafo ottenuto da questo risultato è consultabile in fig.1.6.

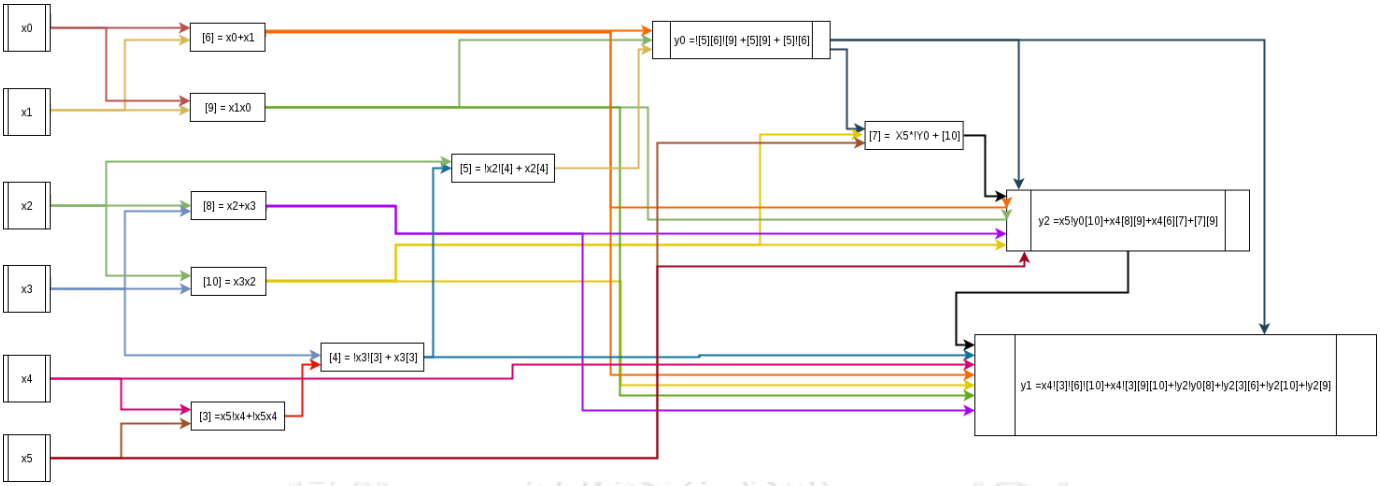

Figura 1.6: Grafo della funzione minimizzata tramite *rugged.script*.

1.2.4 Esercizio 4

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

Di seguito è riportata l'implementazione in VHDL della macchina M. Si noti come, descrivendo la macchina in modalità data-flow, sono stati riportati i nodi forniti da *rugged.script* come segnali

d'appoggio da utilizzare per la realizzazione di Y_2, Y_1 e Y_0. Sono stati inoltre utilizzati dei segnali temporanei d'uscita *y2_temp*, *y1_temp* e *y0_temp* per permettere la definizione di Y_2 in funzione di Y_0 e di Y_1 in funzione di Y_2. Il codice è disponibile qui: M_dataflow.vhd.

Si è poi proceduto alla realizzazione di un *testbench* per simulare la macchina tramite il tool *GHDL*. In tale testbench, i sei ingressi vengono portati da 0 ad 1 a distanza di 10 ns da una transizione all'altra. Il risultato è visibile in fig.1.7.

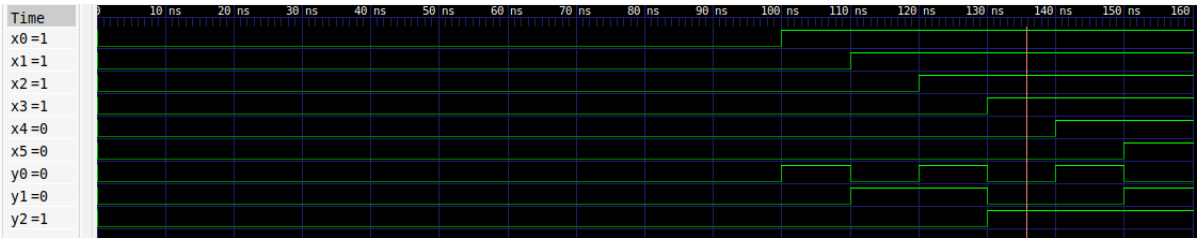

Figura 1.7: Simulazione della macchina M in *gtkwave*.

1.2.5 Esercizio 5

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

Ricordando che un full-adder è in grado di sommare 3 bit riportando in uscita il bit meno significativo s_i e quello più significativo r_i, possiamo procedere come segue: scomponendo la somma di 6 bit in due somme di 3 bit, effettuabili tramite 2 full-adder, otterremo due somme parziali s_0 e s_1 che andranno a loro volta sommate tra loro per ottenere il bit meno significativo dell'uscita y_0. Per quanto riguarda i riporti r_0 e r_1, aventi entrambi peso 1, questi andranno sommati tra loro tenendo anche conto del riporto ottenuto calcolando y_0 (ossia r_2, di peso 1). Il risultato di questa ultima operazione di somma sarà la cifra di peso 1 (y_1) della nostra soluzione, mentre il riporto sarà la cifra di peso 2 (y_2). Usando dunque 4 full-adder, lo schema ottenuto è consultabile in fig.1.8.

Per quanto concerne l'implementazione in VHDL, si è dapprima proceduto all'implementazione di un full-adder seguendo una modalità di descrizione di tipo "behavioural".

```
1  entity full_adder is
2  Port (  X : in  STD_LOGIC;
3      Y : in  STD_LOGIC;
4      CIN : in  STD_LOGIC;
5      S : out  STD_LOGIC;
6      C : out  STD_LOGIC
7    );
8  end full_adder;
9
10 architecture dataflow of full_adder is
11   begin
12     S <=  (X xor Y xor CIN);
13     C <=  ((X and Y) or ((X xor Y) and CIN));
```

```
14  end dataflow;
```

Codice Componente 1.1: Implementazione in VHDL di un full-adder.

Dopodiché, utilizzando questi componenti, si è proceduto a costruire la macchina M seguendo una modalità di descrizione di tipo "structural". Il codice è visualizzabile qui: M.vhd.

Il risultato della simulazione è analogo a quello dell'esercizio 4.

1.2.6 Esercizio 6

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.

Come descritto nell'esercizio 5, la macchina M è in grado di determinare, dati 6 bit in ingresso, il numero di bit alti. Dal momento che si può considerare tale macchina come sommatore in grado di sommare 6 bit, si è deciso di utilizzarla per sommare tra loro le cifre dello stesso peso delle 6 stringhe fornite in ingresso alla macchina S. Essendo tali stringhe composte da 3 bit ciascuna (di peso 2, 1 e 0), si è scelto di usare 3 macchine M per sommare le cifre di stesso peso tra loro. Una volta ottenute tali somme (ciascuna, rispettivamente, espressa su 3 bit in codifica binaria), si è proceduto con tali osservazioni: il bit di peso 0 della somma dei 6 bit di peso 0 non è altro che la cifra di peso 0 del risultato della macchina S, ossia della somma delle 6 stringhe. Il bit di peso 1 della stessa somma, invece, rappresenta invece un bit di peso 1 della somma totale delle stringhe, e lo stesso ragionamento è valido per il bit di peso 2. Passando alla somma dei 6 bit di peso 1 delle stringhe di partenza, si noti come la cifra di peso 0 di tale somma non è altro che un bit di peso 1 della somma totale delle stringhe, mentree la cifra di peso 1 è un bit di peso 2 per la somma totale, e così via.

Seguendo questo ragionamento, è stato possibile combinare le cifre delle somme di peso analogo utilizzando dei full-adder, ottenendo lo schema consultabile in fig.1.9.

Dopodiché si è proceduto alla sua realizzazione in VHDL utilizzando una modalità di descrizione "structural". Il codice è visualizzabile qui: S.vhd.

Il risultato della simulazione è riportato in fig.1.10.

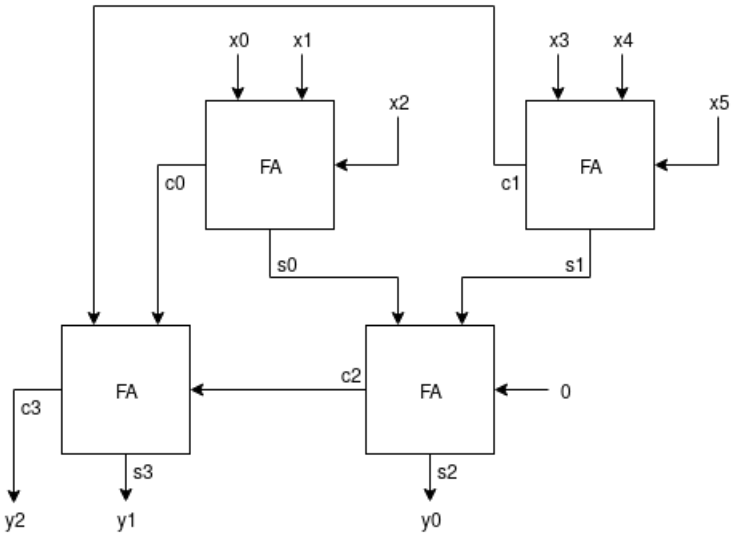

Figura 1.8: Schema della macchina M a partire da blocchi full-adder.

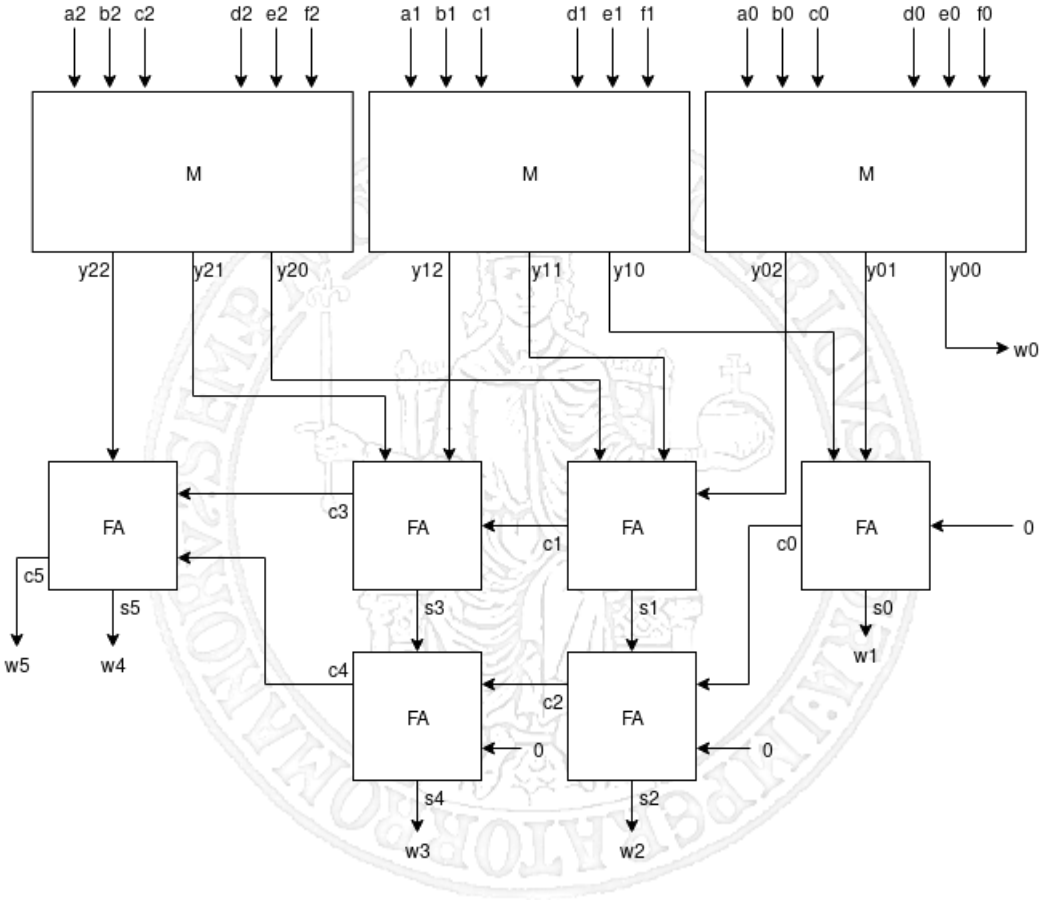

Figura 1.9: Schema della macchina S.

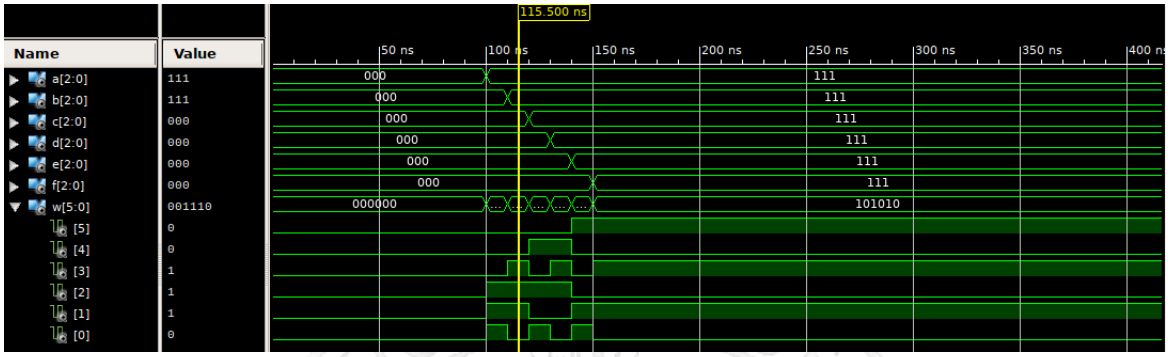

Figura 1.10: Risultato della simulazione della macchina S.

Capitolo 2

Latch e Flip-Flop

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

2.1 Latch RS

2.1.1 Traccia

Implementare e simulare un latch RS 1-attivo abilitato.

2.1.2 Soluzione

2.1.2.1 Implementazione

Il latch RS abilitato è stato realizzato collegano opportunamente le porte NOR e AND, queste ultime necessarie per l'abilitazione. Lo schema di collegamento è riportato in fig.2.1.

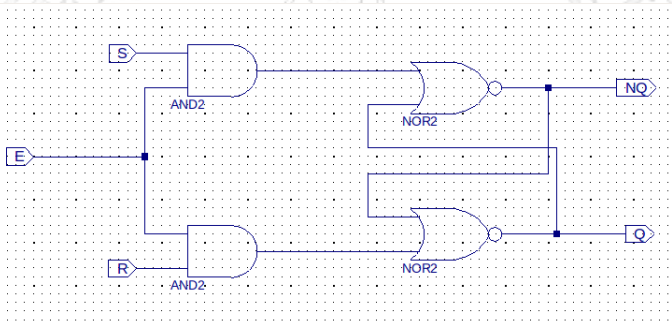

Figura 2.1: Schematico del latch RS abilitato.

Tale soluzione porterà il nostro componente latch RS a funzionare secondo una logica 1-attiva: in particolare, quando S=1 e R=0 il valore di Q si porta a 1 (Set), mentre per portare Q a 0 (Reset) bisognerà portare R a 1 e S a 0. Si noti come, in tale componente, la combinazione S=1 e R=1 non è ammessa, in quanto porterebbe a cambiamenti di stato non definiti.

2.1.2.2 Simulazione

Per la simulazione behavioural di tale componente, dapprima si è testato il comportamento a fronte delle due tipologie di ingresso ammesse. In particolare si è posto prima S=1 e R=0 e, come atteso, il valore di Q risulta 1. Dopodiché, ponendo R=1 e S=0, il valore di Q risulta 0. Anche il caso neutro, rispetto alle variazioni di Q , S=0 e R=0, come previsto, fa in modo che Q non cambi il suo valore. Per quanto concerne la combinazione non ammessa S=1 e R=1, si noti come tale coppia di ingressi porti il latch ad assumere un comportamento non deterministico, in quanto sia la rete di set che quella di reset lavorano per cambiare i valori delle uscite. In simulazione osseviamo come, da 80 ns, sia Q che NQ siano entrambi a 0, ovviamente ciò indica che il nostro componente non sta lavorando correttamete in quanto per come è definito NQ deve valere che "Q XOR NQ = 1" . Un caso di particolare interesse da osservare è la transizione da R=1, S=1 a R=0, S=0, in cui dallo stato non ammesso si passa allo stato neutro. Tale transizione porta nelle ipotesi di idealità del circuito a dei fenomeni oscillatori per le uscite, dovute alla simmetria del circuito [1] e alla retroazione positiva. Infatti nella simulazione behavioral si verifica che ci sono le oscillazioni ma non sono visibili, in quanto tale simulazione assume ritardi di propagazione nulli, e ciò comporta ad avere oscilalzioni con un periodo infinitesimamente piccolo. Infatti il simulatore raggiunge il limite massimo di delta cycle a causa di queste oscillazione che si accumulano nello stesso istante di commutazione e la simulazione termina con un errore. Tale simulazione è visibile in fig.2.1 .

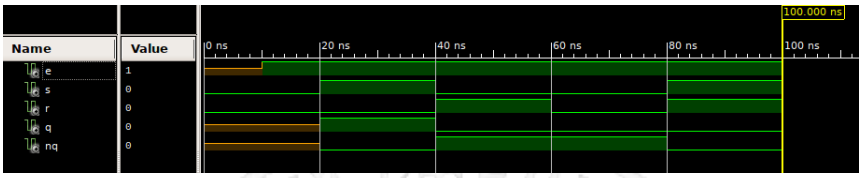

Figura 2.2: Simulazione behavioural del latch RS.

Tali oscillazioni sono visibili chiaramente nella simulazione post-map riportata in fig.2.1, dove ad ogni porta logico è associato il relativo ritardo di porta, secondo la libreria di ISE, ma non sono considerati i ritardi di propagazione sulle linee di collegamento.

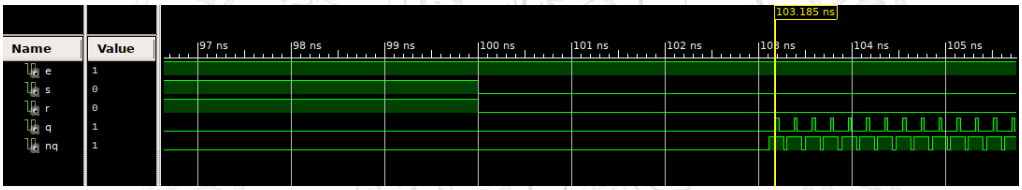

Figura 2.3: Simulazione post-map del latch RS.

Per quanto riguarda la simulazione post-route, le connessioni tra i componenti non sono più considerate ideali, cioè prive di ritardi, ma i ritardi sono modellati secondo le librerie di ISE e secondo la scelte effettuare durante il processo di sintesi da ISE. In questo caso, il simulatore, tenendo conto che i ritardi di propagazione sono diversi per le due retroazioni, dovuto alla reale asimmetria del circuito non dovrebbe generare più oscillazioni persistenti quando si passa dal caso

[1] ritardi delle porte e delle linee identici per le due porte NOR e le due retroazioni

non ammesso a quello neutro. Nel caso in cui la simulazione venga effettuata su board Nexys 4 DDR (fig.2.1), le oscillazioni restano poiché i ritardi di propagazione sono praticamente identici, e si riesce ad osservare anche dalla simulazione, i periodi di occilazione di Q e NQ sono praticamente uguli. Non c'è una linea nettamente più veloce dell'altra e tale da forzare il nostro latch ad assumere un valore stabile dopo un certo intervallo di tempo. Infatti utillazando lo strumento fpga editor di ISE in figura , da aggiungere la foto di gab, si dovrebbe notare che i due percorsi delle retroazioni sono simmetrici o che differiscano di poco. Ciò porterebbe all'identificazione di una probabile PUF (Physical Unclonable Function), che si può generare utilizzando una cascata di latch RS, ma quando tale componente viene sintetizzato e provato sulla Board, si ha che l'uscita va a 0 molto velocemente per tutti i latch della batteria, si ha che il comportamento viene viziato a 0.

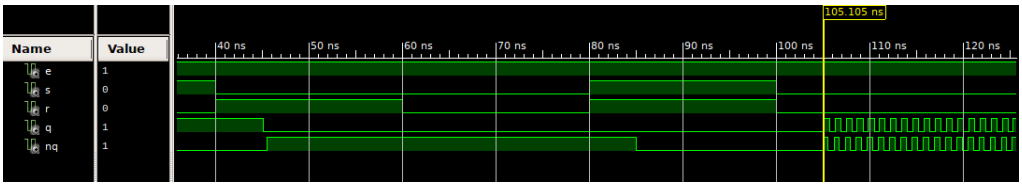

Figura 2.4: Simulazione post-route del latch RS su board Nexys 4 DDR.

Nel caso di simulazione su board Basys 2 (fig.2.1), invece, i ritardi di propagazione sono diversi e le oscillazioni non sono più presenti, tale risultato ci mosta quello che accade nel caso reale.

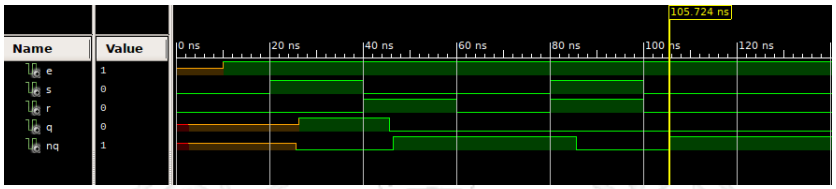

Figura 2.5: Simulazione post-route del latch RS su board Basys 2.

2.2 Latch T

2.2.1 Traccia

Implementare e simulare un latch T abilitato.

2.2.2 Soluzione

2.2.2.1 Implementazione

Si è implementato il latch T abilitato tramite descrizione behavioural. Come da comportamento previsto, il valore di Q verrà invertito solo in caso di abilitazione (en=1) e di valore in ingresso T=1, condizione espressa tramite struttura if then.

```
1 architecture behavioural of latch_T is
2 signal Qtemp : std_logic := '0';
```

```
3  begin
4    process(en,T) is
5    begin
6      if (T='1' and en='1') then
7        Qtemp <= not(Qtemp);
8      end if;
9    end process;
10   Q <= Qtemp;
11   QN <= not Qtemp;
12 end behavioural;
```

Codice Componente 2.1: Implementazione behavioural di un latch T abilitato.

2.2.2.2 Simulazione

Dalla simulazione behavioural (fig.2.6), si può osservare come il valore Q (e dunque notQ) cambi solamente in corrispondenza di T=1 ed en=1, invertendo il valore precedente.

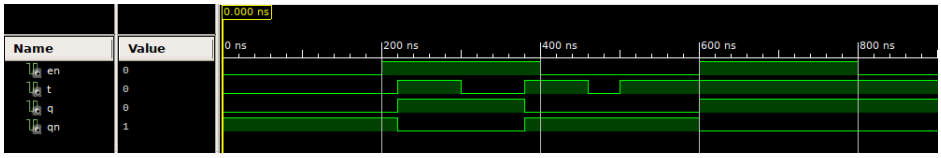

Figura 2.6: Simulazione behavioural del latch T abilitato.

E' interessante notare cosa succede nel caso di simulazione post-map (fig.2.15): dopo aver effettuato il mapping tecnologico, in corrispondenza di T=1 ed en=1 si verificano eventi oscillatori sull'uscita. Una volta che tale coppia di ingressi viene cambiata, infine, il segnale mantenuto resta indeterminato a seguito degli eventi oscillatori.

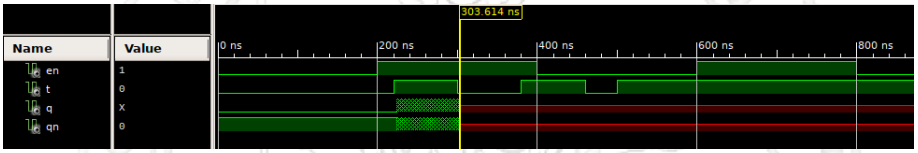

Figura 2.7: Simulazione post-map del latch T abilitato.

Tale fenomeno non si verifica nella simulazione post-route (fig.2.8), dove l'introduzione dei ritardi di propagazione fa sì che l'uscita Q torni al valore corretto dopo le oscillazioni.

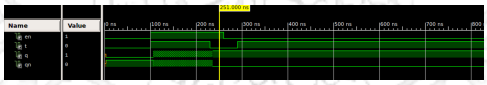

Figura 2.8: Simulazione post-route del latch T abilitato.

2.3 Latch JK

2.3.1 Traccia

Implementare e simulare un latch JK.

2.3.2 Soluzione

2.3.2.1 Implementazione

Il latch JK è stato realizzato mediante descrizione behavioural. Le condizioni if-else utilizzate controllano i valori di J e K in ingresso: per J alto e K basso, Q vien eportato a 1; nel caso contrario, Q è portato a 0. Se entrambi sono alti, invece, il comportamento del latch JK è assimilabile a quello del latch T, e dunque il valore di Q viene invertito.

```
architecture behavioral of latch_jk is
  signal Qtemp: std_logic := '0';
  begin
    p: process(J,K) is
    begin
      if(J='1' and K='0') then
        Qtemp<='1';
      else
        if(J='0' and K='1') then
          Qtemp<='0';
        else
          if(J='1' and K ='1') then
            Qtemp<= not Qtemp;
          end if;
        end if;
      end if;
  end process;
  Q <= Qtemp;
  Qnot <= not Qtemp;
end behavioral;
```

Codice Componente 2.2: Implementazione behavioural di un latch T abilitato.

2.3.2.2 Simulazione

Durante la simulazione behavioural (fig. 2.9), il valore di Q segue perfettamente il comportamento descritto precedentemente. In particolare si noti come, in corrispondenza di J=1 e K=1, il valore di Q venga invertito come nel latch T.

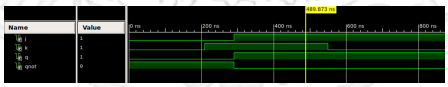

Figura 2.9: Simulazione behavioural del latch JK.

Si noti invece il comportamento del latch in simulazione post-map (fig.2.10): in corrispondenza di J=1 e K=1 si ripresentano gli stessi eventi oscillatori tipici del latch T, proprio perché il comportamento del latch JK è assimilabile a quello del latch T per questi due ingressi. Una volta posto J=0, invece, il comportamento ritorna ad essere quello descritto precedentemente e Q viene abbassato.

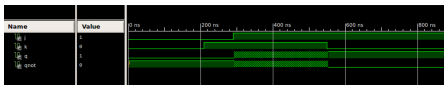

Figura 2.10: Simulazione post-map del latch JK.

2.4 Flip-flop D edge-triggered

2.4.1 Traccia

Implementare e simulare un flip-flop D edge-triggered abilitato che commuta sul fronte di salita con reset asincrono.

2.4.2 Soluzione

2.4.2.1 Implementazione

Il flip-flop D edge-triggered è stato realizzato tramite implementazione behavioural. Si noti come il process, sensibile solo al cambiamento di CLK e reset, porta il valore di D in Q solamente sul fronte di salita di CLK, descrivendo proprio il comportamento atteso del flip-flop. Nel caso in cui invece il reset venga portato al valore reset_level, il valore di Q viene resettato a prescindere dal comportamento del clock (reset asincrono).

```
1  architecture behavioural of flipflop_d_risingEdge_asyncReset is
2  signal q_temp : STD_LOGIC :=init_value;
3  begin
4    q <= q_temp;
5    ff : process(clk, reset)
6    begin
7      if ( reset = reset_level ) then
8        q_temp <= init_value;
9      elsif ( rising_edge(clk) and (enable = enable_level) ) then
10       q_temp <= d;
11     end if;
12   end process ff;
13 end behavioural;
```

Codice Componente 2.3: Implementazione behavioural di un flip-flop D edge-triggered.

2.4.2.2 Simulazione

I risultati della simulazione behavioural sono consultabili in figura 2.11. Si noti come i risultati ottenuti sono perfettamente coerenti con il comportamento previsto: in particolare, al fronte di salita del clock, il valore di Q si porta al valore di D solamente se enable è alto.

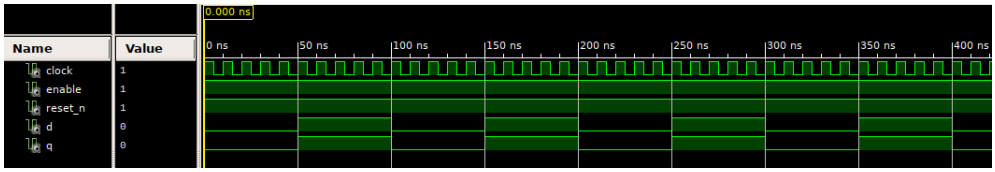

Figura 2.11: Simulazione behavioural del flip-flop RS master-slave.

Si noti inoltre come, nella simulazione post-map (fig. 2.12), a causa del ritardo di propagazione, Q si porta al valore di D solamente qualche istante dopo l'effettivo fronte di salita del clock.

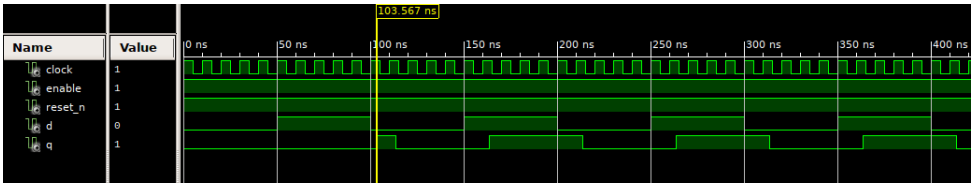

Figura 2.12: Simulazione post-map del flip-flop RS master-slave.

2.5 Flip-flop RS master-slave

2.5.1 Traccia

Implementare e simulare un flip-flop RS master-slave che commuta sul fronte di discesa.

2.5.2 Soluzione

2.5.2.1 Implementazione

Il flip-flop RS master-slave è stato realizzato tramite l'utilizzo di due latch RS collegati in cascata, come riportato in (fig.2.13). In particolare, il primo viene abilitato quando CLK = 1, mentre il secondo quando CLK = 0. Le uscite dunque saranno calcolate sul fronte di salita di clock, ma i risultati saranno visibili sul fronte di discesa.

2.5.2.2 Simulazione

Per la simulazione behavioural di tale componente (fig.2.14), si è proceduto analogamente al caso latch RS. In particolare, si è prima posto R=0 e S=1, in modo tale che, al fronte di discesa del CLK, Q si porta a 1. Anche il caso opposto, con R=1 e S=0, porta come previsto Q a 0, mentre per R=0 e S=0 la Q mantiene il proprio valore ad ogni fronte di discesa. Per quanto concerne il caso particolare R=1 e S=1, si noti come al fronte di discesa del clock si verifichino eventi oscillatori su

Q dovuti all'utilizzo dei latch RS, che il simulatore behavioural non è in grado di mostrare a causa dell'elevatissimo numero di delta cycle.

Tali oscillazioni sono osservabili invece in simulazione post-map (fig.2.15).

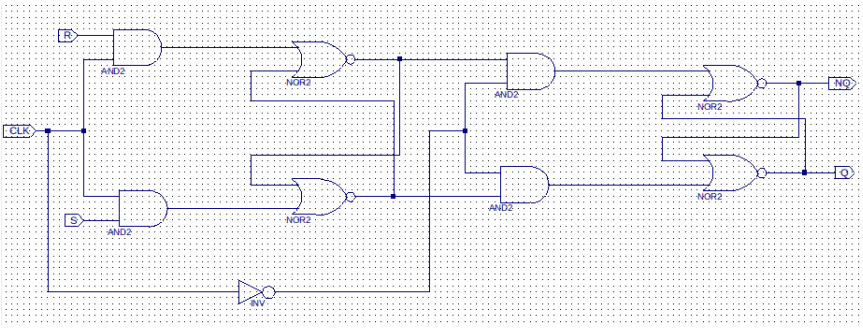

Figura 2.13: Schematico del flip-flop RS master-slave.

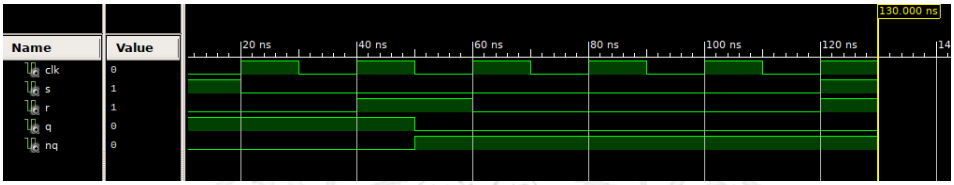

Figura 2.14: Simulazione behavioural del flip-flop RS master-slave.

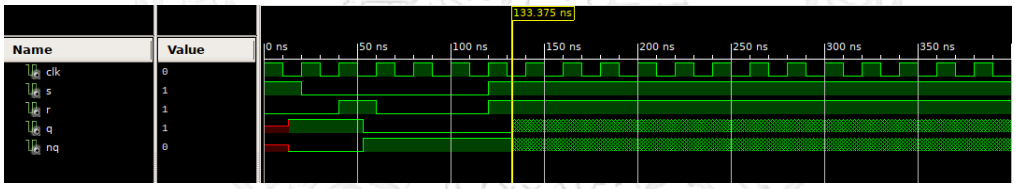

Figura 2.15: Simulazione post-map del flip-flop RS master-slave.

Capitolo 3

Display a 7 segmenti

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

3.1 Architettura

Sulla board che abbiamo in dotazione per risparmiare sul numero di segnali necessari per pilotare 8 digit, la Digiland ha provveduto a collegare i catodi in comune a tutte le digit tra di loro e collegare gli anodi di ogni led della digit tutti ad un anodo comune per ognidigit, come riportato in fig 3.1.

Pertanto per abilitare una digit si bisognerà innanzitutto alimentare opportunamente l'anodo e poi pilotando i catodi opportunamente verrà mostrato sul display il valore. Tale soluzione così fatta presenta però un problema di fondo, alimentando tutte le digit si mostrerà su di esse la stessa cifra. Pertanto si deve provvedere a realizzare una soluzione più ingegnosa, in particolare, sfruttando la persistenza dell immagini sulla retina e la velocità a cui può funzionare la board si riesce a mostrare valori diversi su ogni digit. In particolare ci saranno due componenti che molto velocemente attivano una sola digit alla volta e pilotano i catodi per mostrare il valore che si vuole mostrare su quella digit.

La seguente descrizione si riferisce all'architettura in grado di pilotare soltanto 4 digit, le modifiche necessarie per poterne pilotare 8 sono descritte nell'apposito paragrafo.

L'architettura del componente è mostrata in fig.3.2. I segnali in ingresso saranno:

- *clk* - segnale di clock per la tempificazione;
- *reset* - segnale di reset, per resettare il valore del display quando è alto (tramite eventuale pressione di un pulsante);
- *values* - segnale di 16 bit per detereminare il valore da visualizzare sul display;
- *dots* - segnale di 4 bit per l'abilitazione dei punti decimali sul display;
- *enable_ digit* - segnale di 4 bit per l'abilitazione degli 4 anodi corrispondenti alle 4 cifre sul display (logica 1-attivo);

I segnali in uscita saranno invece:

- *anodes* - bus per l'abilitazione delle 4 digit della batteria di display (0-attivo);
- *cathodes* - bus per pilotare i dei segmenti di ogni cifra (0-attivo);

Il componente è stato realizzato tramite la composizione dei seguenti componenti:

- *clock_divisor* - divisore di frequenza per il clock, necessario perchè se utilizzaimo una frequenza elevata vedremmo lo stesso valore per ogni digit;
- *counter_mod2n* - contatore modulo 2^n, con n=2, per la selezione della cifra da attivare in base al valore di conteggio;
- *anodes_manager* - componente per la selezione degli anodi delle cifre da attivare, formato da un decoder 1-4;
- *cathod_manager* - componente per la selezione dei segmenti da attivare per ogni cifra, formato da un multiplexer 4-1 e un nibble selector/cathod coder.

L'implementazione completa è consultabile qui: display_7_segmenti.vhd.

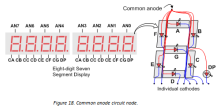

Figura 3.1: Display 7 segmente sulla Nexys 4

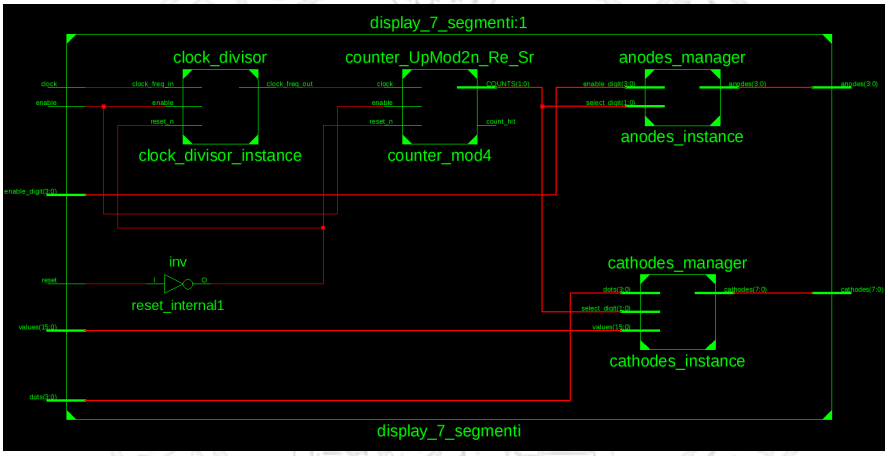

Figura 3.2: Schematico del latch RS abilitato.

3.2 Clock divisor e contatore modulo 2^n

3.2.1 Contatore modulo 2^n

Il contatore è un componente che conta il numero di impulsi applicati in ingresso (sul fronte di salita del clock). Oltre al *clock*, in ingresso c'è un segnale di abilitazione (1-attivo) e un segnale di *reset_n* per resettare il conteggio. In uscita, il segnale *counter* riporta il valore di conteggio corrente, mentre *counter_hit* diventa 1 solamente se il valore del conteggio è costituito da tutti 1 (valore massimo). L'implementazione, effettuata tramite descrizione behavioural, è consultabile qui: counter_mod2n.vhd.

3.2.2 Clock divisor

Prima di utilizzarlo, il segnale di clock in ingresso al componente viene filtrato tramite un clock divisor, che si occupa di filtrare i fronti del clock ad una frequenza *clock_frequency_in* per averli ad una frequenza più bassa *clock_frequency_out*. Il funzionamento di tale componente è del tutto analogo a quello di un contatore modulo 2^n, dove *clock_frequency_out* non è altro che il *counter_hit*, ossia un valore che diventa alto solamente quando il contatore ha raggiunto il suo valore massimo (calcolabile come *clock_frequency_in*/*clock_frequency_out* - 1). Ovviamente sappiamo che il clock generato con questo metodo non è esente da problemi di sincronizzazione e di scue, ma per i nostri scopi e per il fatto che avevamo bisogno di frequenze minori di 5 Mhz, tale scelta risulta più adatta .

L'implementazione, effettuata tramite descrizione behavioural, è consultabile qui: clock_divisor.vhd.

3.3 Anodes manager

L'obiettivo di tale componente è la gestione degli anodi relativi alle cifre del display. Dal momento che i singoli anodi relativi a ciascuna delle 4 cifre del display sono 0-attivi (la Nexys4 DDR utilizza i transistor per pilotare abbastanza corrente nel punto di anodo comune, le abilitazioni dell'anodo sono invertite), l'anodes manager dovrà attivare uno solo dei 4 diverso anodi mantenendo basso uno solo dei 4 bit relativi agli anodi, utilizzando in ingresso il valore fornito dal contatore. Il componente dovrà inoltre tenere conto del segnale enable in ingresso, che permette di attivare e disattivare manualmente i singoli anodi.

Per realizzare l'anodes manager si è utilizzato un decoder 2-4: ricevuto in ingresso il valore del contatore, il decoder alza solo uno dei 4 bit relativi agli anodi. Tali uscite sono poi messe in AND con il segnale di enable per pilotare soltanto le digit che si è degiso di abilitare. Infine, i bit vengono invertiti per rispettare la logica 0-attiva. L'implementazione completa, effettuata tramite descrizione data-flow, è consultabile qui: anodes_manager.vhd.

E' interessante notare come, nell'implementazione del decoder, oltre alle 4 possibili combinazioni di ingressi è stato aggiunto il caso "others", che genera in uscita tutti bit alti (che verranno poi negati successivamente). Tale tecnica serve ad evitare il fault masking, poiché avendo tutti 1 in uscita (caso non previsto dal normale funzionamento di un decoder) posso riconoscere subito la presenza di comportamenti imprevisti nel componente, che verrano poi manifestati all'esterno

mostrando su tutte le digit lo stesso valore anche quando i valori dovrebbero essere diversi, perchè tutti gli anodi sono abilitati nello stesso istante. Una soluzione duale consisterebbe nello spegnere tutte le digit, basta sostituire x"F" con x"0".

```
architecture dataflow of anodes_manager is
  signal anodes_swhitching : STD_LOGIC_VECTOR (3 downto 0) := (others =>
     '0');
  begin
    anodes <= not anodes_swhitching OR not enable_digit;
    with select_digit select anodes_swhitching <=
      x"1"    when "00",
      x"2"    when "01",
      x"4"    when "10",
      x"8"    when "11",
      x"F"    when others;
end dataflow;
```

Codice Componente 3.1: Implementazione data-flow dell'anodes manager.

3.4 Cathodes manager

Il cathodes manager permette di gestire i catodi associati ad ogni segmento omologo di ogni cifra del display a 7 segmenti. Per accendere il giusto segmento è necessario che il catodo corrispondente sia posto a 0, poichè i catodi sono pilotati da segnali 0-attivi. Il componente prende in ingresso:

- il bus *counter*, uscita del contatore (come l'anodes manager), che serve per segliere quale nibble mostrare sulla digit;
- il bus *values* (16 bit) per determinare il valore di ogni cifra e dunque i segmenti da accendere;
- il bus *dots* (4 bit) per determinare quali dei 4 punti decimali accendere.

In uscita abbiamo un bus ad 8 bit *cathodes* che indica la configurazione dei catodi relativi alla cifra attiva in quel momento e all'eventuale punto da accendere.

Per realizzare tale componente, si è utilizzata un'implementazione di tipo behavioural. In particolare, abbiamo due *process*:

- *digit_switching* - in base al valore di *select_digit* (contatore), si occupa di settare i bit del bus interno *nibble*[1] corrispondenti alla rappresentazione del valore che si vuole mostrare su quella digit ;
- *decoder* - in base alla *digit* presente nel *nibble*, imposta *cathodes_for_digit* al valore necessario per accendere i segmenti nel modo corretto per rappresentare il valore richiesto. Tali valori sono espressi come costanti e ricavabili dal reference manual.

[1] Un nibble è una stringa di 4 bit.

Infine, per determinare l'accensione dei dots, si utilizza un *multiplexer 4-1* generico. Il valore di cathodes è dunque determinato come segue: si calcola dapprima la parte dei dots, selezionando solo quelli relativi alle cifre selezionate e poi negandoli (per logica 0-attiva), e infine concatena aggiunge *cathodes_for_digit*.

```
    digit_switching: process (select_digit,values)
    begin
      case select_digit is
        when "00" => nibble <= digit_0;
        when "01" => nibble <= digit_1;
        when "10" => nibble <= digit_2;
        when "11" => nibble <= digit_3;
        when others => nibble <= (others => '0');
      end case;
    end process;
    decoder : process (nibble)
    begin
      case nibble is
        when "0000" => cathodes_for_digit <= zero;
        when "0001" => cathodes_for_digit <= one;
        when "0010" => cathodes_for_digit <= two;
        [...]
           end case;
  end process;

cathodes <= not dots(to_integer(unsigned(select_digit))) &
   cathodes_for_digit;
```

Codice Componente 3.2: Determinazione del valore di cathods..

L'implementazione completa è consultabile qui: cathodes_manager.vhd.

3.5 Display su Nexys 4

La board Nexys 4 DDR ha installato a bordo due batterie di 4 digit ciascuna, per un totale di 8 digit. La soluzione che abbiamo visto precedentemente però permette di controllare una sola batteria di 4 digit, e non potendo instanziare due componenti che controllano ciascuno 4 digit, è necessario apportare alcune modifiche agli elementi che costituiscono il componente che pilota i display a 7 segmenti.

In particolare, di seguito, vediamo alcune modifiche che abbiamo apportato ai componenti per poter pilotare tutte le digit che mette a disposizione la Nexys 4 DDR.

```
entity display_7_segments is PORT ( enable      : in STD_LOGIC;
                  clock       : in STD_LOGIC;
                  reset      : in STD_LOGIC;
                  values       : in STD_LOGIC_VECTOR  (31 downto 0);
                  dots       : in STD_LOGIC_VECTOR  (7 downto 0) ;
```

```
                    enable_digit    : in STD_LOGIC_VECTOR  (7 downto 0);
                    anodes         : out STD_LOGIC_VECTOR (7 downto 0);
                    cathodes       : out STD_LOGIC_VECTOR (7 downto 0)
                  );
end display_7_segments;

...

component counter_UpMod2n_Re_Sr is
        GENERIC (   n    : NATURAL :=3     );
          PORT (  enable      : in STD_LOGIC ;
            reset_n     : in STD_LOGIC;
            clock       : in STD_LOGIC;
            count_hit   : out STD_LOGIC;
            COUNTS      : out STD_LOGIC_VECTOR ((n-1) downto 0)         );
end component;
```

Codice Componente 3.3: display$_{7s}egments$

Nella top level entity del componente che permette di mostrare i valori sui display, il numero di segnali che indicano il la cifra da mostrare passano da 16 a 32, in quanto per ogni digit servono 4 bit per poter codificare i valori, in esadecimale, che possiamo mostrare su di essi. Poichè il numero di display da pilotare passa da 4 a 8, anche il numero di segnali che pilotano i punti , le digit e gli anodi aumentano. Inoltre il contatore non è più un contatore modulo 4 ma è un contatore modulo 8, in quanto deveno essere abilitate 8 digit.

Tali modifiche riguradano anche l'anodes_manager che deve pilotare 8 digit e non più 4, pertanto il decoder da un decoder 2:4 diventa un decoder 3:8 come si nota di seguito.

```
   with select_digit select anodes_switching <=
    x"01"     when "000",
    x"02"     when "001",
    x"04"     when "010",
    x"08"     when "011",
    x"10"  when "100",
    x"20"  when "101",
    x"40"  when "110",
    x"80"  when "111",
    (others => '0')    when others;
  end case;
```

Codice Componente 3.4: Abilitazione degli anodi

Per il cathodes_manager cambia il numero di nibble che dobbiamo gestire, infatti anche essi passano da 4 a 8, ognuna di esse è costituita da 4 bit di values, stringa del valore da mostrare sul display codificato in codifica binaria classica. In particolare partendo dal LSB ogni 4 bit di values codifica il valore da mostrare su una delle digit. Anche il process che si accupa di assegnare ai catodi la corretta codifica del valore viene estesa per poter gestire 8 digit e non più 4.

```
    alias digit_0 is values (3 downto 0);
  alias digit_1 is values (7 downto 4);
```

```
  alias digit_2 is values (11 downto 8);
  alias digit_3 is values (15 downto 12);
  alias digit_4 is values (19 downto 16);
  alias digit_5 is values (23 downto 20);
  alias digit_6 is values (27 downto 24);
  alias digit_7 is values (31 downto 28);

...

    digit_switching: process (select_digit,values)
    begin
      case select_digit is
        when "000" => nibble <= digit_0;
        when "001" => nibble <= digit_1;
        when "010" => nibble <= digit_2;
        when "011" => nibble <= digit_3;
        when "100" => nibble <= digit_4;
        when "101" => nibble <= digit_5;
        when "110" => nibble <= digit_6;
        when "111" => nibble <= digit_7;
        when others => nibble <= (others => '0');
      end case;
    end process;
```

Codice Componente 3.5: cathodes$_m$*anager.vhd*..

3.6 Approfondimento: Double display on-board

Osserviamo il funzionamento del display a 7 segmenti su board. Per poter testare il componente descritto sulla board Nexys 4 utilizzando tutte e 8 le digit, si è realizzata un'entità di più alto di livello denominata *DoubleDisplayOnBoard* che utilizza il componente descritto nel paragrafo precedente. L'idea che abbiamo seguito consiste nel testare il funzionamento del display mostrando su di esso il valore, in esadecimale, dato in input facendo commutare i 16 switch presenti sulla board. Il primo problema che abbiamo incontrato consiste nel fatto che per poter mostrare il valore su tutte le 8 digit, abbiamo bisogno di un bus con parallelismo di 32 bit, quindi con gli switch possiamo pilotare a massimo 4 digit. Per risolvere tale problema abbiamo deciso di seguire la seguente idea. Suddividere il bus values in ingresso al componente display_7_segments e collegare le due metà a dei registri di tipo d abilitati con parallelismo di 16 bit, in ingresso a tali registri abbiamo collegato i 16 switch e collegando gli enable di tali registri con due pulsanti sulla board, possiamo caricare la rappresentazione in binario del valore che vogliamo mostrare nei registri in modo tale che tale metà verrà sostenuta in ingresso al componente fino a quando non si provvederà ad un nuovo aggiornamento rieseguendo la procedura di load, premendo il pulsante.

```
[...]
architecture structural of display_onBoard is
  entity DoubleDisplayOnBoard is
```

```
4       Port (  clock         : in  STD_LOGIC;
5           values      : in  STD_LOGIC_VECTOR (15 downto 0);
6           load_reg_0_3    : in  STD_LOGIC;
7           load_reg_4_7    : in  STD_LOGIC;
8           anodes      : out  STD_LOGIC_VECTOR (7 downto 0);
9           cathodes      : out  STD_LOGIC_VECTOR (7 downto 0)
10      );
11    end DoubleDisplayOnBoard;
12  [...]
13
14  alias values_0_3 is values_int (15 downto 0) ;
15  alias values_4_7 is values_int (31 downto 16) ;
16
17  begin
18    register_0_3: register_d_Re_Ar PORT MAP (   clock   => clock,
19                            load   => load_reg_0_3,
20                            reset   => reset,
21                            d    => values,
22                            q     => values_0_3
23                  );
24    register_4_7: register_d_Re_Ar PORT MAP (   clock    => clock,
25                            load  => load_reg_4_7,
26                            reset    => reset,
27                            d   => values,
28                            q      => values_4_7
29                  );
30
31  [...]
```

Codice Componente 3.6: DoubleDisplayOnBoard.vhd

L'implementazione completa è consultabile qui: display_onBoard.vhd.

Inoltre, è stato opportunamente configurato il file Nexys4DDR_master.ucf per effettuare il mapping tra i physical della board e le interfaccia di ingresso/uscita del componente che abbiamo sintetizzato, un punto interessante è il collegamento dei segnali di load dei registri ai bottoni presenti sulla scheda. Ciò ci permette di poter scegliere in quale registro e di conseguenza su quale batteria di display mostrare il valore che stiamo dando in input con gli switch.

```
1  [...]
2  #bottone destro
3    NET "load_reg_0_3"          LOC=P17 | IOSTANDARD=LVCMOS33; #
       IO_L12P_T1_MRCC_14
4  #bottone sinistro
5    NET "load_reg_4_7"          LOC=M17 | IOSTANDARD=LVCMOS33; #
       IO_L10N_T1_D15_14
6  [..]
```

Codice Componente 3.7: DoubleDisplayOnBoard.vhd

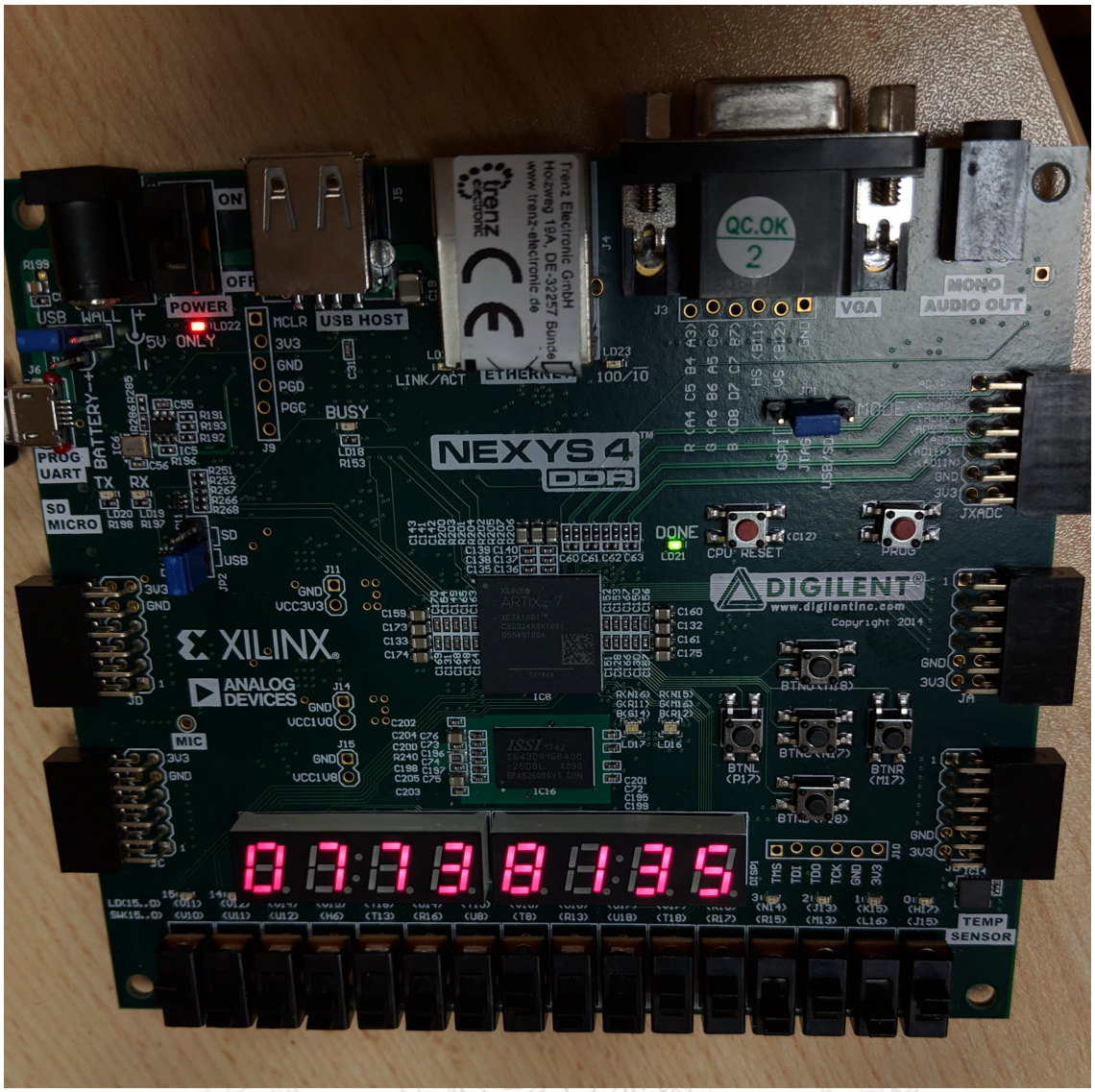

Figura 3.3: Display a 7 segmenti su board Nexys 4.

Capitolo 4

Clock generator

Sviluppare un progetto di sintesi di un DCM.

4.1 Implementazione

Sulle fpga Artix7 il DCM[1] è stato sostiuito dal CMT [2], tale componente include all'interno un MMCM [3] e un PLL [4].

Per poter sintetizzare un generatore di clock con una frequenza minore o multipla di 100 Mhz limitando lo scew, possiamo utilizzare il CMT. Per utilizzare tale componente, Xilinx fornisce un IP-core che deve essere opportunamente personalizzato,mediante il clocking-wizard , al fine di poter generare un modulo vhdl che utilizza/ configura correttamente il CMT sulla board al fine di fornire un clokc alla frequenza.

Durante la personalizzazione del componente il clocking wizard ci permette di scegliere se avere o meno un segnale, LOCKED, che ci indica quando CMT è riuscito ad agganciare la fase del clock principale e di conseguenza possiamo iniziare a utilizzare correttamente tutti i dispositivi che usano il clock in uscita al CMT.

Nella figura .4.1 si vede come abbiamo configurato il nostro componente che utilizza il CMT, in particolare abbiamo deciso di utilizzare tre uscite di tale componente configurate in modo tale da fornire segnali di clock con frequenza pari alla metà, un quarto e un decimo della frequenza in ingresso [5].

Una volta conclusa la procedura il Wizard genera un file vhd. Per poter simulare e verificare che il componente funzionasse correttamente, abbiamo deciso di realizzare un piccolo componente che utilizza il componente generato e un left shifter register che esegue un operazione di shift ad ogni colpo di clock. La top level entity presenta la seguente interfaccia:

```
1 [...]
2 entity clk_tester is
3   GENERIC( N  : integer := 8     );
4   Port (  clock_in      : in  STD_LOGIC;
5       enable      : in  STD_LOGIC;
```

[1] DCM : Digital Clock Manager
[2] Clock Management Tile
[3] mixed-mode clock manager
[4] phase-locked loop
[5] La board Nexys4 è dotata di un oscillatore con frequenza pari a 100 Mhz

```
6       reset_n         : in  STD_LOGIC;
7       d_in          : in  STD_LOGIC;
8       q_out        : out STD_LOGIC;
9       Q           : out STD_LOGIC_VECTOR (N-1 downto 0);
10      half_clock     : out STD_LOGIC;
11      quarter_clock       : out STD_LOGIC;
12      tenth_clock    : out STD_LOGIC;
13      locked       : out STD_LOGIC
14    );
15  end clk_tester;
16
17  [...]
```

Codice Componente 4.1: clock$_t$*ester.vhd*

C'è da precisare, che tale interfaccia ha come uscita i segnali di clock soltanto per rendere più agevole la fase di analisi della simulazione.

Mentre il collegamento tra i componenti è stato realizzato come di seguito :

```
1   [...]
2
3   signal enable_int      : STD_LOGIC :='1';
4
5   begin
6     half_clock     <= half_clock_int;
7     quarter_clock       <= quarter_clock_int;
8     tenth_clock       <= tenth_clock_int;
9     locked       <= enable_int;
10
11  clock_Ints: my_clock port map(  CLK_IN1 => clock_in,
12                    CLK_OUT1  => half_clock_int,
13                    CLK_OUT2 => quarter_clock_int,
14                    CLK_OUT3 => tenth_clock_int,
15                    LOCKED  => enable_int
16              );
17
18  shifter_register_inst: shifterRegister generic map ( N => N)
19                       port map (   clock    => half_clock_int,
20                            enable     => enable_int,
21                            reset_n    => reset_n,
22                            left    => left,
23                            d_in     => d_in,
24                            q_out   => q_out,
25                            Q       => Q
26                 );
27  [...]
```

Codice Componente 4.2: clock$_t$*ester.vhd*

In particolare lo shifter registrer è abilato dal segnale di locked del generatore di clock, pertanto appena il componente ci segnalerà che ha aggangiato correttamente la fase del segnale in ingresso

lo shifter register inizierà a funzionare, e funzionerà ad una frequenza pari alla metà di quella del segnale in ingresso al componente che usa il CMT.

Di seguito analizziamo i risultati della simulazione, 4.2, di tale componente.

Dopo circa 40 cicli di clock, il segnale locked va alto,marker blu, e il lo shifter register inizia a lavorare, e non appena riceve un valore in ingresso alto, osserviamo che ad ogni qualvolta il segnale half_clock va alto il bit viene shiftato verso sinistra. Inoltre osserviamo che poichè d_in resta alto per circa un periodo e mezzo, il valore in ogni cella dello shifter register viene mantunuto alto per due periodi di half_clock, dopo il terzo periodo il valore ritorna basso perchè si inizia a propagare il valore zero da d_in in tutte le celle a partire dall'instante segnalato dal marker giallo.

L'implementazione completa è consultabile qui: clock_tester.vhdleft_right_shift_register.vhdmy_clock.v

4.2 Simulazione

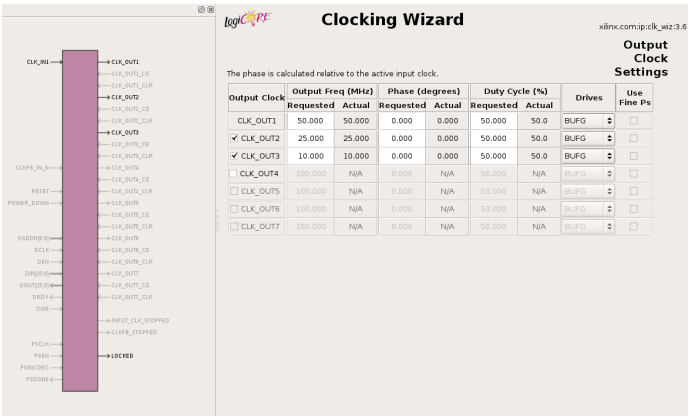

Figura 4.1: DClocking_wizard

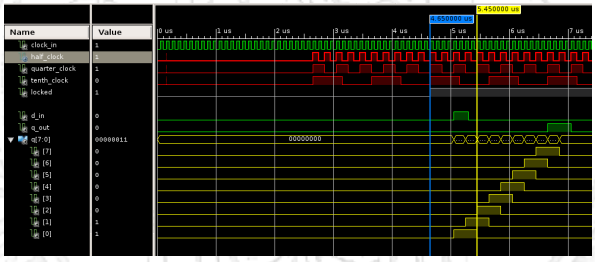

Figura 4.2: Risultati simulazione

Capitolo 5

Scan Chain

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

1. Modalita normale: l'array si comporta come un registro di N posizioni;
2. Modalita controllo: i flip-flop possono essere scritti e letti individualmente congurandoli 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 e la distanza del primo stadio dalla cella da raggiungere.

5.1 Implementazione

L'interfaccia del componente è la seguente:

```
entity scan_chain is
  generic(
    width : integer := 8;
    shift_direction : std_logic := '1'
  );  Port (
    clock : in  STD_LOGIC;
    en : in  STD_LOGIC;
    reset_n : in STD_LOGIC;
    scan_en : in  STD_LOGIC;
    d_reg : in  STD_LOGIC_VECTOR (width-1 downto 0);
    scan_in : in  STD_LOGIC;
    q_reg : out  STD_LOGIC_VECTOR (width-1 downto 0);
    scan_out : out  STD_LOGIC
  );
end scan_chain;
```

Codice Componente 5.1: Implementazione data-flow dell'anodes manager.

In ingresso, oltre ai segnali di *clock*, *enable* e *reset*, il componente avrà i seguenti segnali:

- *scan_ en*: bit di selezione della modalità di funzionamento (1 per modalità normale, 1 per modalità controllo);
- *d_ reg*: valori in ingresso dei flip-flop nel registro;
- *scan_ in*: valore in ingresso da inserire nel registro in caso di shift;

In uscita, invece, *q_ reg* sarà l'uscita del registro, mentre *scan_ out* sarà il bit tirato fuori dal registro in caso di shift.

Per quanto riguarda l'implementazione, è stata utilizzata una descrizione di tipo *structural*. In particolare, è stato generato un numero di flip-flop multiplexati pari al valore generico *width*, che rappresenta il parallelismo del registro. Tale componente è un tipo particolare di flip-flop D il cui ingresso viene prima selezionato tramite multiplexer. Per generare un registro che effettui shift a destra, si fissa come *scan_ in* di ogni flip-flop il valore in uscita del flip-flop precedente (alla sua sinistra), dopodiché si utilizza il multiplexer per decidere quale ingresso portare nel flip-flop. In particolare, *scan_ en* sarà il segnale di selezione: se 0 (modalità normale), in ingresso avremo il corrispettivo valore in ingresso di *d_ reg*, se 1 (modalità controllo) l'ingresso del flip-flop sarà *scan_ in*, e dunque l'uscita di quello precedente. Il discorso è analogo nel caso di shift a sinistra: in base al valore generico *shift_ direction* si stabilisce quale tipologia di shift register si vuole generare e dunque come collegare tra loro i flip-flop. L'implementazione completa è consultabile qui: scan_chain.vhd.

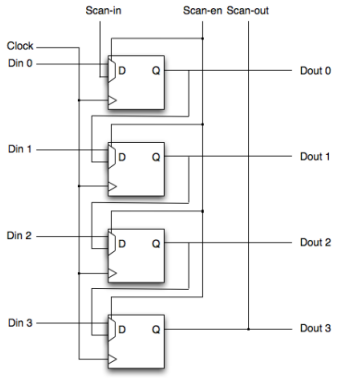

Figura 5.1: Architettura del componente scan_chain.

5.2 Simulazione

Per testing, si è effettuata una simulazione behavioural del componente: durante tale simulazione, si è utilizzato il componente in entrambe le sue modalità di funzionamento con la stringa '00000111'. In particolare, dapprima si è utilizzata la modalità normale (*scan_ en*=0) per caricare i valore nel registro. Dopodiché, quando *scan_ en*=1, il registro si comporta correttamente come uno shift register, shiftando i bit a sinistra ad ogni fronte di salita del clock. Una volta posto di nuovo *scan_ en*=0, il registro smette di shiftare le cifre e, al successivo fronte di salita del clock, carica di nuovo il valore in ingresso nel registro. I risultati sono consultabili in fig. 5.2.

Per completezza, in fig. 5.3 è stato riportato il risultato della simulazione con shift a destra (*shift_direction*=1), perfettamente coerente con il comportamento atteso.

5.3 Approfondimento: Scan Chain On Board

Oltre alla classica simulazione, per testare il corretto funzionamento della scan chain abbiamo deciso di provare tale componente sulla board.

In particolare si è realizzata una top level entity che utilizza la scan chain per sostenere il valore in ingresso alla batteria di display a 7 segmenti presenti sulla board.

L'interfaccia della top level entity è la seguente :

```
[...]

entity scan_chain_on_board is
      Port (  clock     : in  STD_LOGIC;
                scan_in   : in  STD_LOGIC;
           scan_clk  : in  STD_LOGIC;
           scan_en   : in  STD_LOGIC;
           scan_out    : out STD_LOGIC;
           anodes   : out STD_LOGIC_VECTOR (7 downto 0);
           cathodes  : out STD_LOGIC_VECTOR (7 downto 0)
);
end scan_chain_on_board;

[...]
```

Codice Componente 5.2: Interfaccia top level entity .

In ingresso, oltre ai segnali di *clock, scan_in e scan_en abbiamo anche scan_clk. Quest'ultimo segnale è il clock per la scan chain, ma essendo quello della board un segnale a frequenza troppo elevata, che ci avrebbe impedito di apprezzare il funzionamento della scan chain* [1]. *Pertanto abbiamo deciso, anche se sbagliato, di dare manualmente un impulso "per simulare un clock personalizzato" premendo un pulsante presente sulla board. Quest'ultima scelta errata ci è stata segnalata anche dal tool di sintesi, infatti mappando scan_clk su un pulsante viene generato un errore, ma lo stesso tool suggerisce l'opportuna modifica da effettuare al file ucf per poter bypassare tale errore*[2].

Di seguito il mapping di scan_en e di scan_clk sull'ucf:

```
[...]
#bottone centrale
  NET "scan_en"          LOC=N17 | IOSTANDARD=LVCMOS33; #IO_L9P_T1_DQS_14
#bottone inferiore
  NET "scan_clk"          CLOCK_DEDICATED_ROUTE = FALSE | LOC=P18 |
      IOSTANDARD=LVCMOS33; #IO_L9N_T1_DQS_D13_14
[...]
```

[1] anche con un clock negli ordini dei khz premendo il pulsante associato a scan_en venivano eseguite troppe operazioni di shif

[2] *Aggiungere al file ucf* "CLOCK_DEDICATED_ROUTE = FALSE" alla riga dove si effettua il mapping del bottone.

Codice Componente 5.3: Interfaccia top level entity .

Per testare il funzionamento del componente, bisogna settare il bit in ingresso alla scan chain spostando il primo switch e poi premere il pulsante centrale per abilitare l'operazione di shift e simulare i colpi di clock premedo il pulsante in basso. Se il bit in uscita alla scan chain è alto allora il primo led della batteria si illuminerà.

L'implementazione completa della top level_entity è consultabile qui: scan_chain_on_board.vhd.

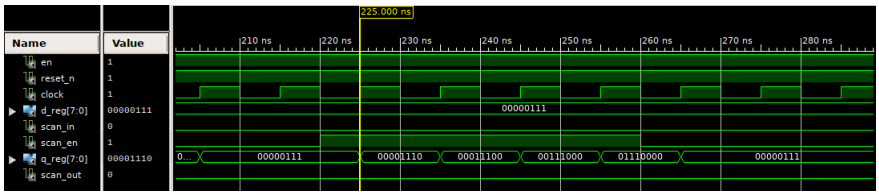

Figura 5.2: Simulazione del componente scan chain (shift a sinistra).

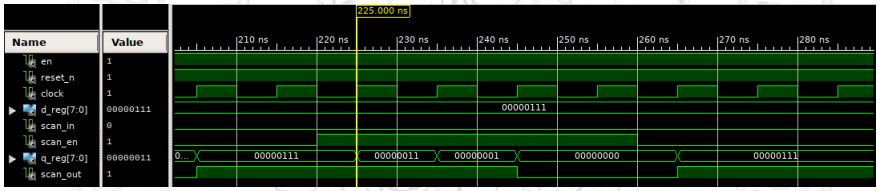

Figura 5.3: Simulazione del componente scan chain (shift a sinistra).

Capitolo 6

Finite State Machine

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 (opzionale: poi i costrutti con guardia e simulare con GHDL).

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 1 specicando al tool di sintesi Xilinx ISE diverse codiche per gli stati. Per quelle rilevanti estrapolare la codica assegnata dal sintetizzatore, area occupata e frequenza massima di lavoro, apportando eventuali commenti.

6.1 Implementazione

6.1.1 Implementazione con descrizione a doppio processo (Moore)

Si è deciso di realizzare una macchina di Moore per il riconoscimento della sequenza '1011001'. L'automa a stati finiti realizzato è rappresentato in fig.6.1.

Per l'implementazione di questo automa in VHDL, si è deciso di utilizzare una descrizione di tipo behavioural. In particolare, si è utilizzata la tecnica con due processi:

- *update_ state*, che si occupa di aggiornare lo stato corrente ad ogni colpo di clock;
- *choose_ next_ state*, che rappresenta la parte combinatoria della macchina, atta a determinare lo stato prossimo e l'uscita usando un costrutto *case*;

Tale scelta, sebbene renda il codice meno compatto, risulta più facilmente leggibile nella parte di determinazione dello stato prossimo. Si noti come, essendo questa una macchina di Moore, la determinazione dell'uscita dipende solamente dallo stato in cui si trova la macchina: in particolare, come osservabile anche nel codice, l'uscita sarà alta solamente quando la macchina avrà rilevato l'intera stringa.

```
1 when s1011001 =>
2   value_out <= '1';
3   if value_in = '0' then
4     next_state <= init;
```

```
5   else next_state <= s1;
6 end if;
```

Codice Componente 6.1: Determinazione dell'uscita in VHDL (Moore).

L'implementazione completa è consultabile qui: fsm_moore.vhd

6.1.2 Implementazione con guardie (Mealy)

Si è deciso di realizzare tale macchina anche come Mealy: in particolare, dal momento che in questa implementazione l'uscita dipende direttamente dall'ingresso, il valore in uscita sarà 1 quando, trovandoci nello stato s101100, si riceve in ingresso il valore 1. Fatto ciò, è stato dunque possibile rimuovere lo stato s1011001 ed aggiungere due transizioni da s101100 a init e s1 a fronte degli ingressi 0 e 1. L'automa a stati finiti realizzato è rappresentato in fig.6.1.

Per l'implementazione di questo automa in VHDL, si è ricorso all'utilizzo di costrutti di assegnazione con guardia e funzione di risoluzione. In particolare, sono stati utilizzati due guardie innestate: quella esterna per far commutare lo stato solo sul fronte di salita del clock e quelle interne per determinare stato corrente e uscita in funzione degli ingressi. La funzione di risoluzione, invece, permette di determinare lo stato *current_state* in caso di assegnazioni concorrenti (si sceglie lo stato più a sinistra nel vettore degli stati concorrenti). L'implementazione completa è consultabile qui: fsm_mealy.vhd

6.2 Sintesi e simulazione

6.2.1 Sintesi e simulazione con descrizione a doppio processo (Moore)

Si è proceduto alla sintesi e alla simulazione della macchina tramite il tool Xilinx ISE. Per quanto riguarda la codifica degli stati, sono state utilizzate le codifiche One hot, Compact, Sequential, Gray, Johnson e Speed1. Le codifiche per ciascuno stato sono riportate nella tabella seguente:

stato	One hot	Compact	Sequential	Gray	Johnson	Speed1
init	00000001	000	000	000	0000	100000100
s1	00000010	010	001	001	0001	100000010
s10	00000100	101	010	011	0011	000000101
s101	00001000	011	011	010	0111	010000010
s1011	00010000	110	100	110	1111	101000000
s10110	00100000	111	101	111	1110	000100001
s101100	01000000	001	110	101	1100	000010100
s1011001	10000000	100	111	100	1000	100001100

Una volta sintetizzata la macchina, si sono osservati i risultati ottenuti: in particolare, la frequenza massima di lavoro e l'area occupata in termini di numero di slice (registri e LUT) e di flip-flop sono rappresentati nella seguente tabella:

parametro	One hot	Compact	Sequential	Gray	Johnson	Speed1
Freq. massima (MHz)	968.898	1008.776	1008.770	1008.776	964.971	1177.163
n. di Slice	16	7	7	7	9	14
n. di Flip-Flop	8	3	3	3	4	7

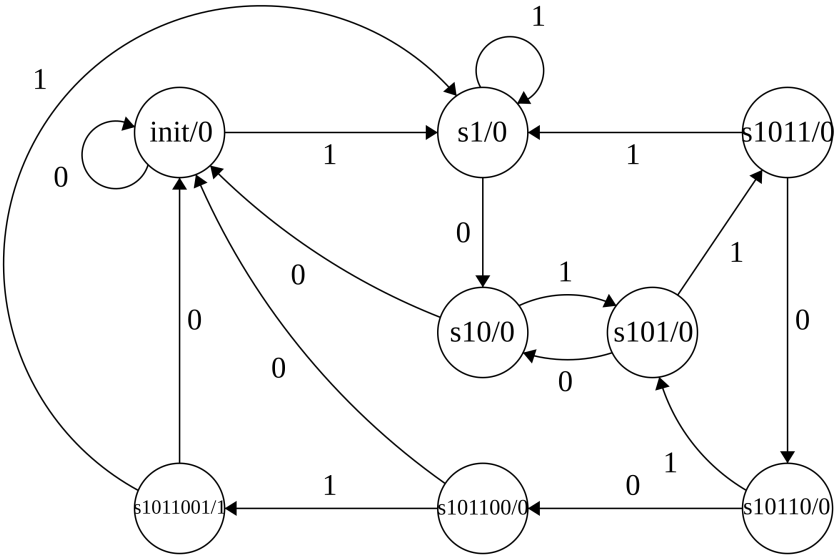

Figura 6.1: Automa a stati finiti della macchina (Moore).

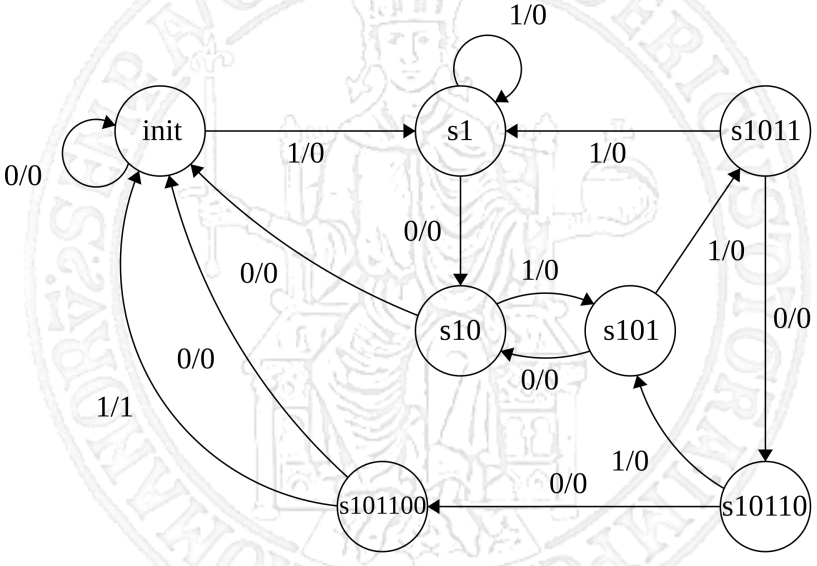

Figura 6.2: Automa a stati finiti della macchina (Mealy).

Dato il numero esiguo di stati, le diverse tipologie di codifica non presentano significative differenze. In generale, possiamo dire che la Speed1, come previsto, ottimizia la frequenza massima di lavoro, mentre per ottenere migliori prestazioni in termini di area si preferiscono le codifiche Compact, Sequential e Gray.

I risultati della simulazione sono riportati in fig.6.3. Si noti come l'uscita diventa alta solamente dopo che, in corrispondenza dei fronti di salita del clock, l'ingresso presenta la sequenza '1011001', per poi abbassarsi al successivo colpo di clock in quanto la macchina è tornata nello stato *init*.

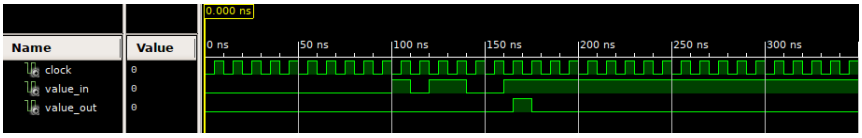

Figura 6.3: Risultati della simulazione della macchina (Moore).

6.2.2 Simulazione con guardie (Mealy)

Per quanto concerne la macchina Mealy, dal momento che sono stati utilizzati i costrutti di guardia, non è stato possibile sintetizzarla. I risultati della simulazione, effettuata dunque tramite il tool GHDL, sono riportati in fig.6.4. Si noti come questi sono completamente analoghi al caso Moore.

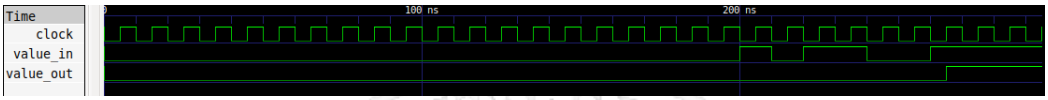

Figura 6.4: Risultati della simulazione della macchina (Mealy).

Capitolo 7

Ripple Carry Adder

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 sul- l'area occupata e tempo di calcolo al variare di N e commentare il risultato con le formule teoriche.

7.1 Architettura

Il ripple carry adder è stato realizzato a partire da una serie di full-adder in cascata: ogni full adder *i* prenderà in ingresso due cifre dello stesso peso *i* e il carry uscente dallo stadio precedente *i-1* e darà in uscita la cifra *i* del risultato. L'architettura può essere osservata in fig.7.1.

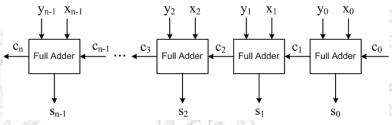

Figura 7.1: Architettura del Ripple Carry Adder.

A causa dell'architettura a cascata utilizzata, tale circuito presenta tempi di calcolo più elevati delle sue controparti parallele. Se si considera il suo path critico (gli n+1 carry dei full-adder), il tempo di calcolo aumenta linearmente con il numero di full adder, e dunque con il numero di bit n delle stringhe da sommare. In particolare, $T = nT_{FA}$, dove T_{FA} è il tempo di propagazione del full adder impiegato. Il vantaggio dell'utilizzo di questo circuito è la sua ridotta area: anche questa, infatti, dipende linearmente dal numero di full adder e dunque da n: $A = nA_{FA}$, dove A_{FA} è l'area del full adder.

7.2 Implementazione

7.2.1 Full Adder

Per la realizzazione del RCA, si è dapprima proceduto alla realizzazione di un full adder. Per tale implementazione, si è utilizzata una descrizione di tipo data-flow:

```
architecture dataflow of full_adder is
begin
  S <= (X xor Y xor CIN);
  C <= ((X and Y) or ((X xor Y) and CIN));
end dataflow;
```

Codice Componente 7.1: Implementazione data-flow di un full adder.

L'implementazione completa è consultabile qui: full_adder.vhd

7.2.2 Ripple Carry Adder

Per implementare il RCA, invece, si è utilizzata una descrizione di tipo structural. Tramite un *generate* si generano *width* full adder, collegati secondo lo schema visto nel paragrafo precedente:

```
architecture structural of rippleCarry_adder is
[...]
begin
  S <= S_temp;
  carries (0) <= c_in;
  c_out <= carries(width);
  rippleCarry_adder : for i in 0 to width-1 generate
    f_adder: full_adder port map (
      x => X(i),
      y => Y(i),
      c_in => carries(i),
      s => S_temp(i),
      c_out => carries(i+1)
    );
    end generate rippleCarry_adder;
end structural;
```

Codice Componente 7.2: Implementazione structural di un RCA.

L'implementazione completa è consultabile qui: ripple_carry_adder.vhd

7.3 Simulazione e sintesi

7.3.1 Simulazione

Per tale componente è stata effettuata una simulazione behavioural, durante la quale sono stati cambiati sia gli operandi in ingresso che il carry in ingresso. I risultati ottenuti sono osservabili in fig.7.2.

7.3.2 Sintesi

Si è proceduto poi alla sintesi del componente utilizzando diversi valori di lunghezza in bit delle stringhe, ottenuti cambiando il parametro generico *width*: a fronte di ogni valore n scelto, attraverso l'utilizzo del report di sintesi, sono stati ricavati i seguenti termini:

- *numero di slices*, relativo dunque all'area occupata;
- *minimum period* (inteso come reciproco della massima frequenza di funzionamento), relativo dunque al ritardo.

Si sono inoltre utilizzati due registri per i valori in ingresso ed in uscita, in modo tale da evitare eventuali ritardi dovuti all'utilizzo di blocchi I/O dell'FPGA che avrebbero potuto alterare i risultati dell'esperimento. I risultati, in funzione del numero di bit, sono riportati in fig.7.3. Si noti come, nel caso dell'area, i risultati siano perfettamente coerenti con l'andamento lineare teorico già descritto precedentemente. Nel caso del minimum period, invece, l'andamento risulta migliore nel caso pratico che in quello teorico: ciò è dovuto al fatto che, in fase di sintesi, il tool effettua un'ottimizzazione dell'architettura del componente, sfruttando a pieno le matrici di interconnessione tra i CLB presenti nell'FPGA per ridurre i ritardi del circuito.

7.4 Approfondimento: Ripple Carry Adder Add-Sub

7.4.1 Architettura e implementazione

Si è deciso inoltre di implementare un Ripple Carry Adder in grado di effettuare sia addizioni che sottrazioni su operandi rappresentati in complemento a 2. Tale componente è stato realizzato nel seguente modo: prima di portare il secondo operando in ingresso al RCA, viene effettuata la XOR di tale operando con *subtract*: se *subtract*=0 (modalità addizione), il segnale entrerà inalterato nell'RCA, altrimenti se *subtract*=1 (modalità sottrazione) la XOR ne effettuerà il complemento. All'interno dell'RCA il carry in ingresso sarà proprio *subtract*: se questo è 0, verrà effettuata una normale addizione A+B; se invece *subtract*=1 verrà effettuata l'addizione A+(-B), in quanto B è stato prima complementato e poi incrementato di 1 e dunque invertito grazie al carry in ingresso al RCA.

Si noti come tale addizionatore può presentare una condizione di overflow/underflow, dal momento che gli operandi rappresentati in complemento a 2 perdono un bit utile alla rappresentazione del valore per utilizzarlo come segno. In particolare, si verificherà una condizione di overflow/underflow in questi casi:

- somma di due positivi con risultato negativo;
- somma di due negativi con risultato positivo;
- differenza di positivo e negativo con risultato negativo;
- differenza di negativo e positivo con risultato positivo.

Per identificare tale condizione, è stata utilizzata una macchina denominata overflow_checker, progettata utilizzando la mappa di Karnaugh in fig.7.4.

L'architettura di questa macchina è riportata in fig.7.5.

Tale macchina è stata implementata tramite descrizione structural. L'implementazione completa è consultabile qui: rippleCarry_addsub.vhd

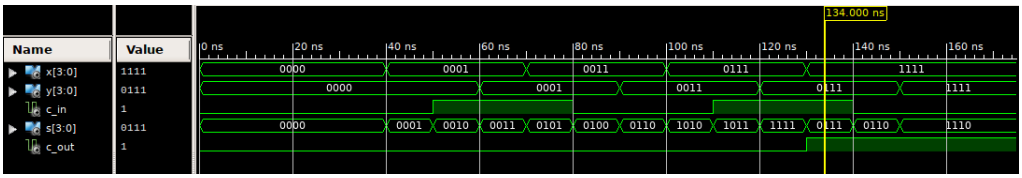

Figura 7.2: Simulazione behavioural del Ripple Carry Adder.

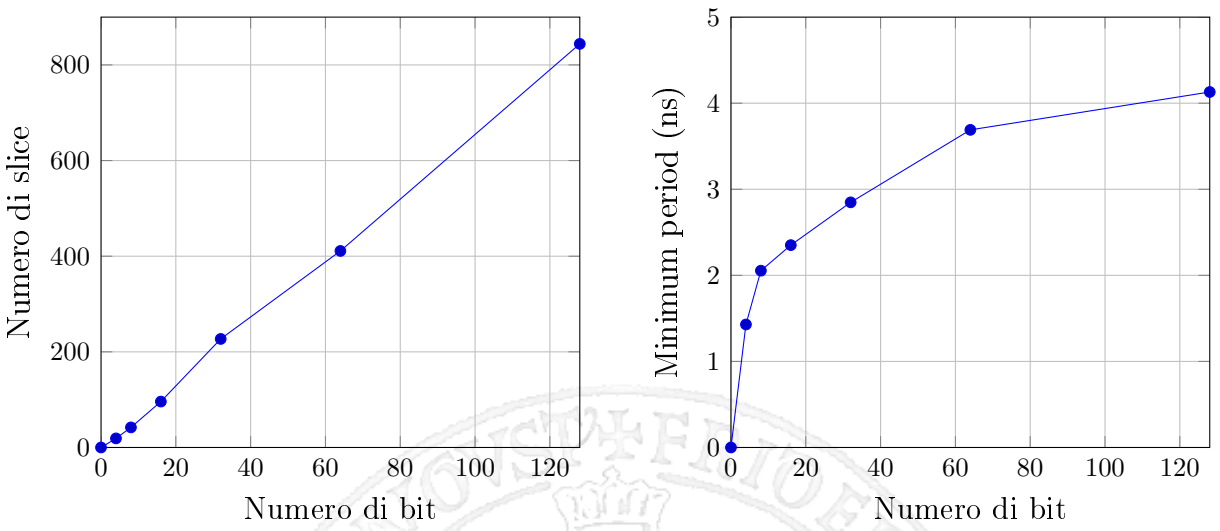

Figura 7.3: Grafici dei risultati ottenuti post-sintesi in funzione del numero di bit.

sub s / a b	00	01	11	10
00		1		
01			1	
11	1			
10				1

Figura 7.4: Mappa di Karnaugh dell'overflow checker.

7.4.2 Simulazione

Per tale componente è stata effettuata una simulazione behavioural, durante la quale sono stati cambiati sia gli operandi in ingresso che il valore *subtract* per effettuare sia addizioni che sottrazioni. I risultati ottenuti sono osservabili in fig.7.6. Si noti come il bit di overflow in uscita viene impostato correttamente ad 1 a fronte delle combinazioni che generano overflow.

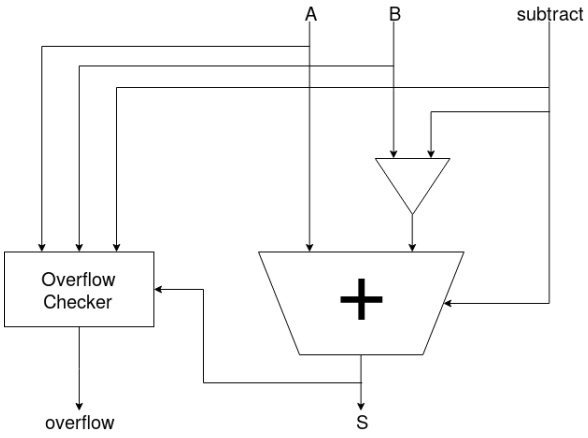

Figura 7.5: Architettura del Ripple Carry Adder Add-Sub.

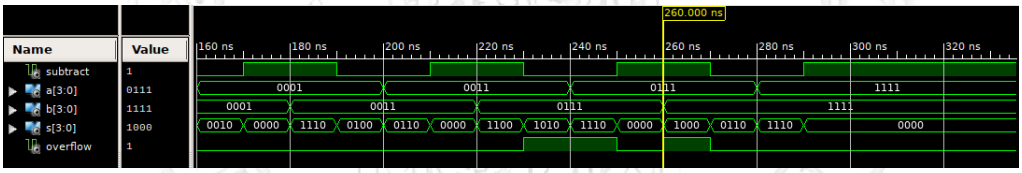

Figura 7.6: Simulazione behavioural del Ripple Carry Adder.

Capitolo 8

Carry Look Ahead

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.)

8.1 Architettura

L'architettura di tale componente è riportata in fig.8.1. Si noti come, per ottimizzare i ritardi rispetto al RCA, si calcolano in anticipo i riporti sugli stadi successivi utilizzando le condizioni di propagazione (X and Y) e generazione (X or Y). In particolare, i componenti utilizzati sono:

1. *Propagation/Generation calculator*, che si occupa del calcolo delle condizioni di propagazione (P_i) e generazione (G_i);
2. *Carry Look Ahead*, che si occupa del calcolo dei riporti. In particolare, $C_{i+1} = G_i + P_i C_i$;
3. *Full adder*, che si occupano del calcolo della somma tra i valori di X e Y e i carry in ingresso.

8.2 Implementazione

Si è deciso di implementare il componente *carry_ look_ ahead_ adder* direttamente in forma generica, permettendo di specificare il numero di bit delle stringhe tramite parametro generico *width*. Per l'implementazione, si è utilizzata una descrizione di tipo structural. Di seguito vengono riportati i componenti utilizzati.

8.2.1 Propagation/Generation calculator

L'implementazione di questo componente è stata realizzata tramite descrizione dataflow. In particolare, di seguito è riportata la parte di codice relativa alla generazione delle condizioni di propagazione e generazione:

```
architecture dataflow of propagation_generation_calculator is
begin
  G <= X and Y;
  P <= X or Y;
end;
```

Codice Componente 8.1: Descrizione dataflow del Propagation/Generation calculator.

L'implementazione completa è consultabile qui: propagation_generation_calculator.vhd

8.2.2 Carry Look Ahead Adder

Per realizzare l'addizionatore, oltre al propagation/generation calculator, si sono utilizzati altri due componenti generati tramite costrutto generate: *carry_look_ahead*, per il calcolo dei riporti, e *full_adders* per la rete di full adder atta al calcolo delle somme:

```
carry_look_ahead: for i in 0 to (width-1) generate
  carry_ahead :
    C(i+1) <= G(i) or (P(i) and C(i));
end generate carry_look_ahead;

full_adders: for i in 0 to (width-1) generate
  fullAdder : full_adder port map (
    x => X(i),
    y => Y(i),
    c_in => C (i),
    s => S_TEMP(i)
  );
end generate full_adders;
```

Codice Componente 8.2: Generazione del $carry_{l}ook_{a}headedeifulladder$.

Si noti come, all'interno del port map di ciascun full_adder, non siano stati utilizzati i riporti in uscita *c_out*, dal momento che tali riporti sono già stati calcolati precedentemente dal *carry_look_ahead*. L'implementazione completa è consultabile qui: carry_look_ahead_adder.vhd

8.3 Simulazione e sintesi

8.3.1 Simulazione

Per tale componente è stata effettuata una simulazione behavioural, durante la quale sono stati fatti variare i due operandi e il riporto in ingresso. I risultati ottenuti sono osservabili in fig.8.2.

8.3.2 Sintesi

Si è proceduto infine alla sintesi del componente utilizzando diversi valori di lunghezza delle stringhe tramite manipolazione del parametro generico *width*. Come nel caso del Ripple Carry Adder, sono stati ottenuti il *numero di slices* e *minimum period* (reciproco della massima frequenza di funzionamento) in funzione del numero n di bit per valutare le prestazioni di tale macchina. I risultati sono riportati in fig.8.3.

Si osservi come, rispetto ai risultati ottenuti con il Ripple Carry Adder, il circuito occupi un numero di slices (e dunque area) sensibilmente maggiore. Ciò è dovuto al fatto che, introducendo il calcolo dei riporti in parallelo, il numero di componenti è aumentato: tuttavia, tale differenza è apprezzabile solo con numero più elevato di bit (dai 32 in poi). Per quanto riguarda i ritardi, invece, non si registrano particolari differenze rispetto al caso RCA: i periodi minimi risultano pressoché identici. Possiamo dunque concludere che, con l'utilizzo del tool di sintesi per board Nexys 4, non sono particolarmente apprezzabili le differenze tra l'utilizzo di un RCA e di un CLA grazie alle capacità di ottimizzazione del suddetto tool. Si noti infine che, a fronte di sintesi del componente con un numero di bit maggiore di 32, il tool riporta il seguente warning: "*WARNING:Xst:1336 - (*) More than 100% of Device resources are used*". Questo è dovuto al fatto che le risorse necessarie per sintetizzare il componente sulla board non sono sufficienti, e dunque tali risultati sono da considerarsi solo in teoria.

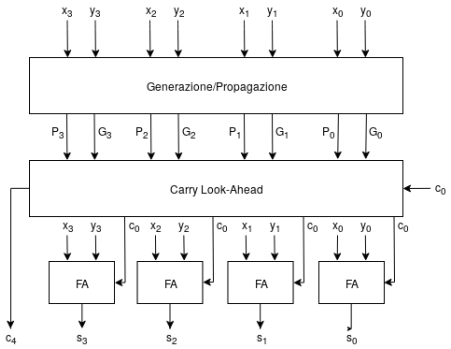

Figura 8.1: Architettura del Carry Look Ahead.

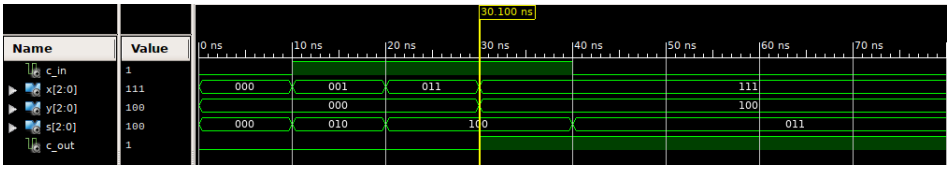

Figura 8.2: Simulazione behavioural del Carry Look Ahead.

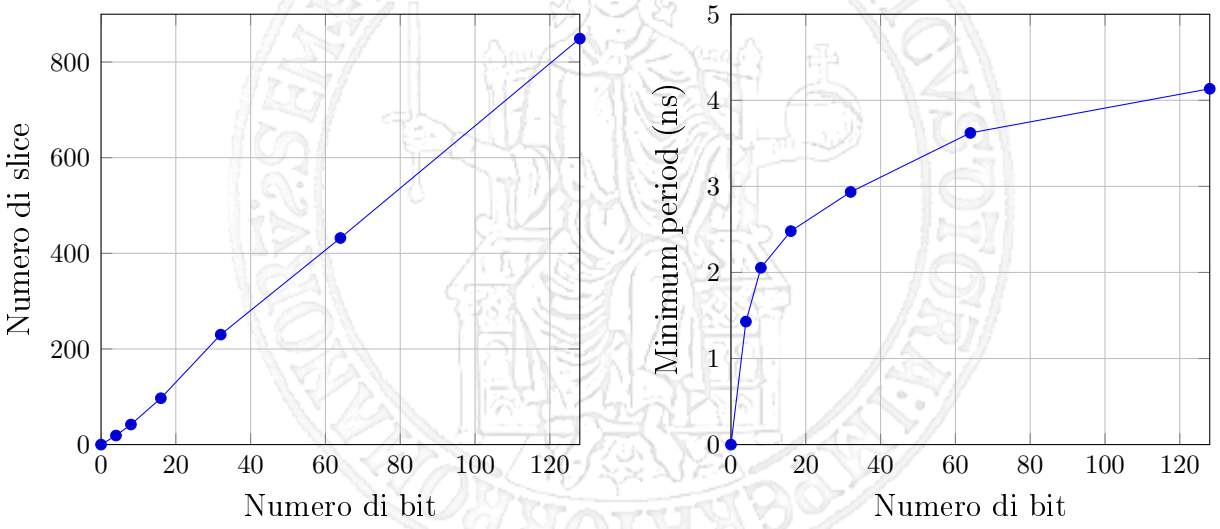

Figura 8.3: Grafici dei risultati ottenuti post-sintesi in funzione del numero di bit.

Capitolo 9

Carry Save

Realizzare un esempio di addizionatore basato sulla modalità Carry Save.

9.1 Architettura

Il Carry Save è un addizionatore in grado di effettuare la somma tra 3 stringhe di bit di lunghezza n. In particolare, tale macchina è formata da:

1. blocchi *carry save*, ossia dei full adder che si occupano di sommare 3 bit dello stesso peso delle tre stringhe;
2. blocchi *full adder* che sommano al risultato del CSL_i il riporto uscente dal blocco CSL_{i-1} e dal full adder precedente.

L'architettura della macchina è osservabile in fig.9.1. In particolare, si noti come la seconda catena di full adder vada a formare un *ripple carry adder*. Per semplificarne l'implementazione possiamo dunque sostituire a tale schema quello in fig.9.2, formato da un *carry save logic* (serie di carry save block) e un RCA.

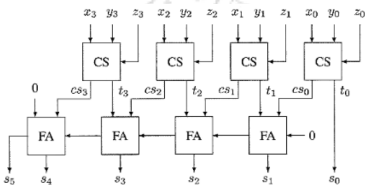

Figura 9.1: Architettura del Carry Save.

9.2 Implementazione

9.2.1 Carry Save Logic

Per l'implementazione della Carry Save Logic, ossia della catena di Carry Save Block, si è utilizzata una descrizione di tipo structural. La generazione dei singoli blocchi full adder è stata realizzata come segue:

```
T <= T_temp;
CS <= CS_temp;
carry_save_blocks: for i in 0 to (width-1) generate
  carry_save_block: full_adder port map (
    x => X(i),
    y => Y(i),
    c_in => Z(i),
    s => T_temp(i),
    c_out => CS_temp(i)
  );
end generate carry_saves;
```

Codice Componente 9.1: Generazione dei Carry Save Block.

L'implementazione completa è consultabile qui: carry_save_logic.vhd

9.2.2 Carry Save

Per realizzare il Carry Save, è stata una descrizione di tipo structural con un parametro generico *width* per definire la il numero di bit dei tre operandi. Data *width*, la somma sarà espressa su *width*+2 bit. Oltre al componente Carry Save Logic già descritto, si è fatto uso di un Ripple Carry Adder per sommare CS con i valori di T (shiftati a destra) e ottenere le cifre più significative del risultato S:

```
S(0) <= T(0);
A <= '0' & T(width-1 downto 1);
RCA: ripple_carry_adder GENERIC MAP (
  width => width )
  X => A,
  Y => CS,
  c_in => '0',
  S => S(width downto 1),
  c_out => S(width+1)
);
```

Codice Componente 9.2: Utilizzo del Ripple Carry Adder nel Carry Save.

L'implementazione completa del Carry Save è consultabile qui: carry_save.vhd

9.3 Simulazione e sintesi

9.3.1 Simulazione

Per tale componente è stata effettuata una simulazione behavioural, durante la quale sono stati fatti variare i tre operandi da sommare. I risultati ottenuti sono osservabili in fig.9.3.

9.3.2 Sintesi

Si è proceduto infine alla sintesi del componente utilizzando diversi valori di lunghezza delle stringhe tramite manipolazione del parametro generico *width*. Come per gli addizionatori precedenti, sono stati ottenuti il *numero di slices* e *minimum period* in funzione del numero di bit per valutare le prestazioni di tale macchina. I risultati sono riportati in fig.9.6.

Come ci si aspettava, il circuito occupa molta più area rispetto alle sue controparti con solo due operandi. Tuttavia, i tempi di elaborazione risultano molto simili, grazie alle capacità di ottimizzazione del tool di sintesi.

9.4 Approfondimento: confronto con RCA a tre operandi

Per poter effettuare una valutazione delle prestazioni del Carry Save, si è deciso di realizzare un altro componente per la somma di tre operandi. In particolare, tale componente è stato realizzato mediante l'utilizzo di due Ripple Carry Adder in cascata. L'architettura del componente è visibile in fig.9.5.

L'implementazione dell'RCA a tre operandi, effettuata tramite descrizione structural, è consultabile qui: rca_tre_operandi.vhd

Si è poi sintetizzato il componente seguendo le stesse procedure del Carry Save. Nei grafici riportati in fig.**??**, è possibile osservare le differenze tra le prestazioni dei due.

Come previsto, le prestazioni del Carry Save sono molto migliori, specialmente all'aumentare del numero di bit. Grazie all'utilizzo dei blocchi carry save, infatti, i riporti vengono calcolati contemporaneamente in tutti gli stadi, ottimizzando i tempi del circuito rispetto al caso con gli RCA. Anche in termini di area il Carry Save è migliore, in quanto utilizza un full adder in meno per il bit più significativo del risultato.

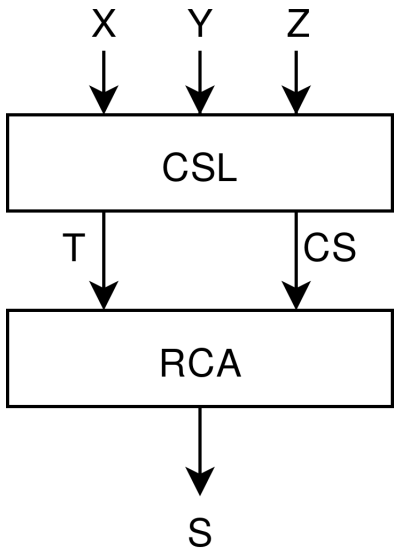

Figura 9.2: Architettura del Carry Save con CSB e RCA.

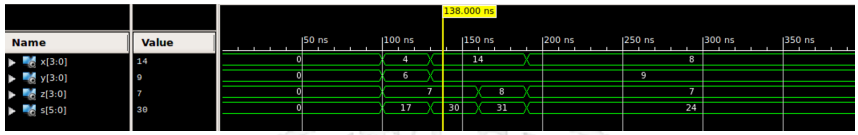

Figura 9.3: Simulazione behavioural del Carry Save.

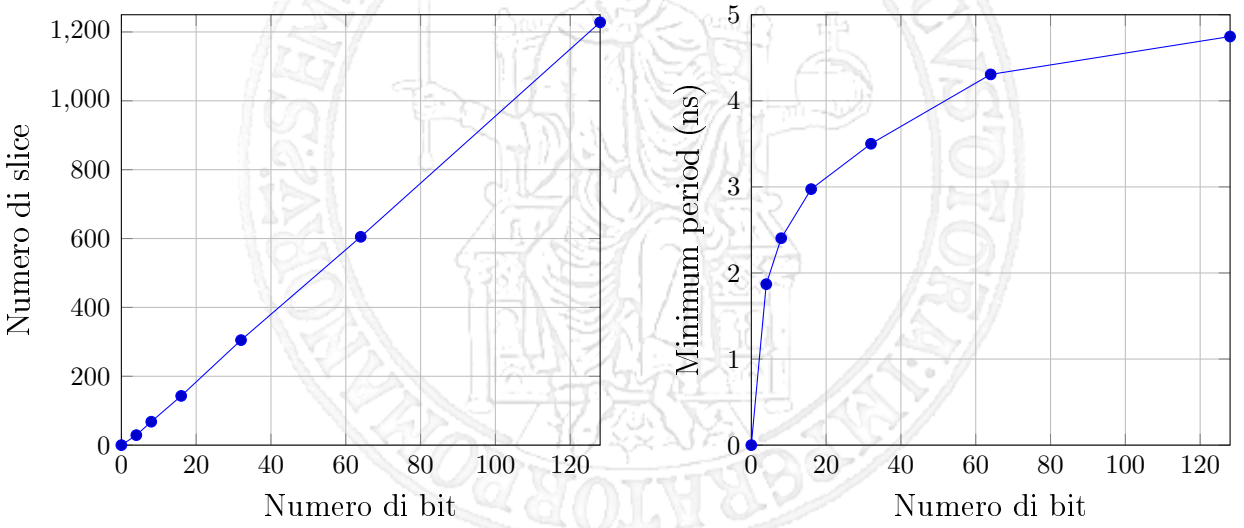

Figura 9.4: Grafici dei risultati ottenuti post-sintesi in funzione del numero di bit.

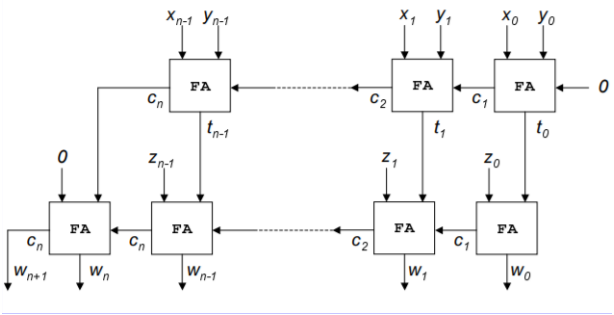

Figura 9.5: Architettura dell'RCA a tre operandi.

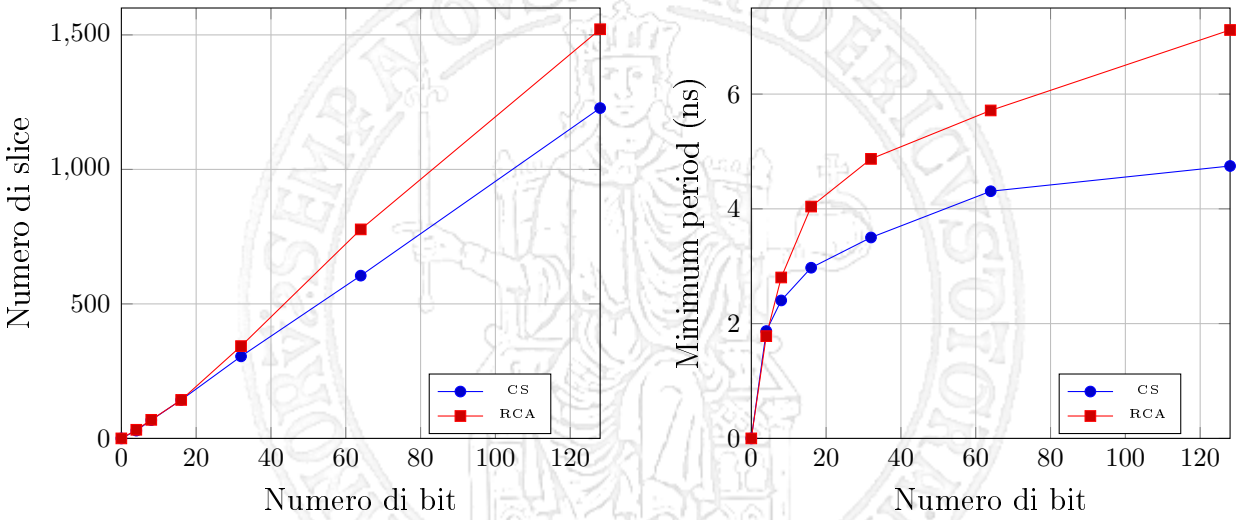

Figura 9.6: Grafici dei risultati ottenuti post-sintesi in funzione del numero di bit.

Capitolo 10

Carry Select

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 ris ultato con le formule teoriche.

10.1 Architettura

Il Carry Select può essere inteso come un'estensione del RCA al fine di migliorarne le prestazioni in caso di numero di bit elevato. Dal momento che le prestazioni del RCA peggiorano linearmente all'aumentare dl numero di bit, si può pensare di suddividere la carry chain in P blocchi, ciascuno dei quali lavora su M bit delle stringhe in ingresso. L'architettura del componente è raffigurata in fig.10.1.

In particolare, si noti come ogni blocco (ad eccezione del primo) sia composto da due RCA e due multiplexer: gli RCA sommano gli stessi M bit del blocco corrispondente, ma si distinguino per il valore di *c_ in*, ossia il riporto in ingresso. In base al valore effettivo del riporto in ingresso, calcolato allo stadio precedente, i due multiplexer selezioneranno i bit dell'uscita S e il *c_ out* dell'RCA corrispondente. In questo modo, siamo in grado di calcolare contemporaneamente gli M bit in uscita di ogni blocco P: il ritardo complessivo sarà dunque pari a quello di un unico RCA di M bit + quello delle P-1 coppie di multiplexer da pilotare, ossia $T_{CSEL} = M*T_{FA} + (P-1)*T_{MUX}$. Tale vantaggio in tempo denota tuttavia un enorme svantaggio in termini di area occupata, notevolmente superiore a quella di un normale RCA con un'unica carry chain di M*P blocchi.

10.2 Implementazione

10.2.1 Carry Select Block

Per la realizzazione del componente Carry Select, si è dapprima proceduto alla realizzazione di un *Carry Select Block*, ossia dei blocchi che andranno a formare il Carry Select. L'interfaccia di questo componente è la seguente:

```
1 entity carry_elect_block is
```

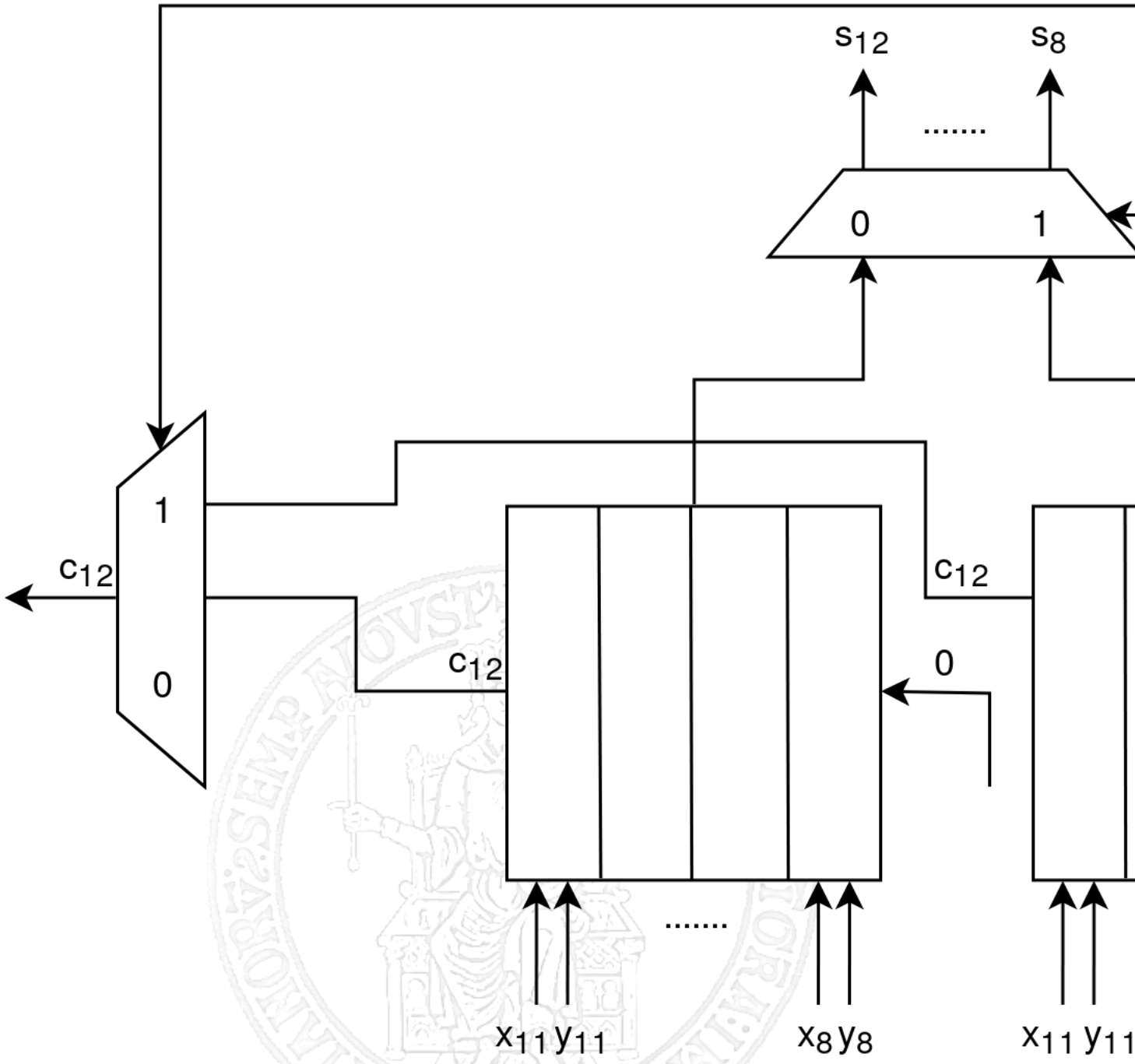

Figura 10.1: Architettura del Carry Select.

```
  generic (
    width : NATURAL := 4
  );  port (
    A : in STD_LOGIC_VECTOR ((width-1) downto 0);
    B : in STD_LOGIC_VECTOR ((width-1) downto 0);
    c_in : in STD_LOGIC;
    S : out STD_LOGIC_VECTOR ((width-1) downto 0);
    c_out : out STD_LOGIC);
end carry_select_cell;
```

Codice Componente 10.1: Interfaccia di un Carry Select Block.

Tale componente, come visto prima, è formato da due RCA e due multiplexer 2-1. I due addizionatori si occuperanno di sommare le due stringhe in ingresso A e B (di lunghezza generica *width*) con *c_ in* pari, rispettivamente, a 0 e 1. In base al valore *c_ in* effettivo in ingresso al blocco, i due multiplexer sceglieranno quali dei due valori *S* e *c_ out,* calcolati dai due RCA, riportare in uscita. L'implementazione completa, realizzata tramite descrizione structural, è consultabile qui: carry_select_block.vhd

10.2.2 Carry Select

Anche per realizzare il Carry Select è stata utilizzata una descrizione di tipo structural. Prima di tutto, si sono utilizzati due parametri generici: P, ossia il numero di blocchi della catena, ed M, ossia il numero di bit sommato da ciascun blocco. Dopodiché, per quanto concerne l'architettura, sono stati generati l'RCA iniziale e P-1 blocchi restanti per formare la carry chain del sommatore:

```
rca: rippleCarry_adder port map(
  X => A((M-1) downto 0),
  Y => B((M-1) downto 0),
  c_in => internal_carry(0),
  S => S_TEMP((M-1) downto 0),
  c_out => internal_carry(1)
);

blocks: for i in 1 to P-1 generate
  csel_block: carry_select_block port map (
    A => A ((((i+1)*M)-1) downto(i*M)),
    B => B ((((i+1)*M)-1) downto (i*M)),
    c_in  => internal_carry(i),
    S => S_TEMP((((i+1)*M)-1) downto (i*M)),
    c_ou => internal_carry(i+1)
  );
end generate blocks;
```

Codice Componente 10.2: Generazione dei blocchi del Carry Select.

L'implementazione completa è consultabile qui: carry_select.vhd

10.3 Simulazione e sintesi

10.3.1 Simulazione

Per tale componente è stata effettuata una simulazione behavioural, durante la quale sono stati fatti variare i due operandi da sommare. I risultati ottenuti sono osservabili in fig.10.2.

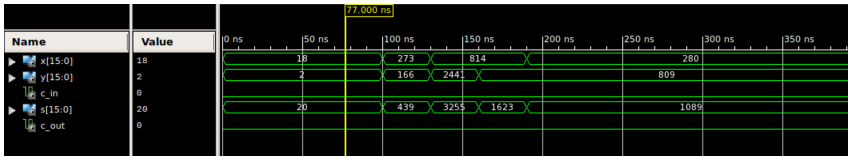

Figura 10.2: Simulazione behavioural del Carry Save.

10.3.2 Sintesi

10.3.2.1 Scelta di P ed M

Si è proceduto infine alla sintesi del componente. Per la scelta di P ed M, sono state fatte le seguenti considerazioni: data la formula e $T_{CSEL} = M * T_{FA} + (P-1) * T_{MUX}$, è possibile ottimizzare tale quantità scegliendo $M = \sqrt{N * \frac{T_{FA}}{T_{MUX}}}$ e $P = \frac{N}{M}$.

Si è deciso quindi di fissare $N = M * P = 32$ e di sintetizzare il componente per valutare le prestazioni temporali del circuito. Per ottenere il risultato ottimale, dati $T_{FA} = 0.893$ ns e $T_{MUX} = 0.889$ ns, possiamo trovare $M \simeq \sqrt{N} = 8$ e $P = 8$. Per effettuare una dimostrazione dell'efficacia di tale formula, sono riportati in tabella i *minimum period* di funzionamento del Carry Select ottenuti al variare di M e P (con N fissato pari a 32):

P=1, M=64	3.690 ns
P=2, M=32	3.760 ns
P=4, M=16	3.753 ns
P=8, M=8	3.618 ns
P=16, M=4	3.685 ns
P=32, M=2	3.690 ns
P=64, M=1	3.690 ns

E' possibile osservare come la scelta dei valori $M = 8$ e $P = 8$ ci restituisca i risultati ottimali in termini di tempi del circuito.

10.3.2.2 Prestazioni all'aumentare del numero di bit

Si è proceduto infine alla sintesi del componente utilizzando diversi valori N e scegliendo M e P tramite la formula vista precedentemente. Sono stati quindi ottenuti il *numero di slices* e *minimum period* in funzione del numero di bit per valutare le prestazioni di tale macchina. I risultati sono riportati in fig.10.3.

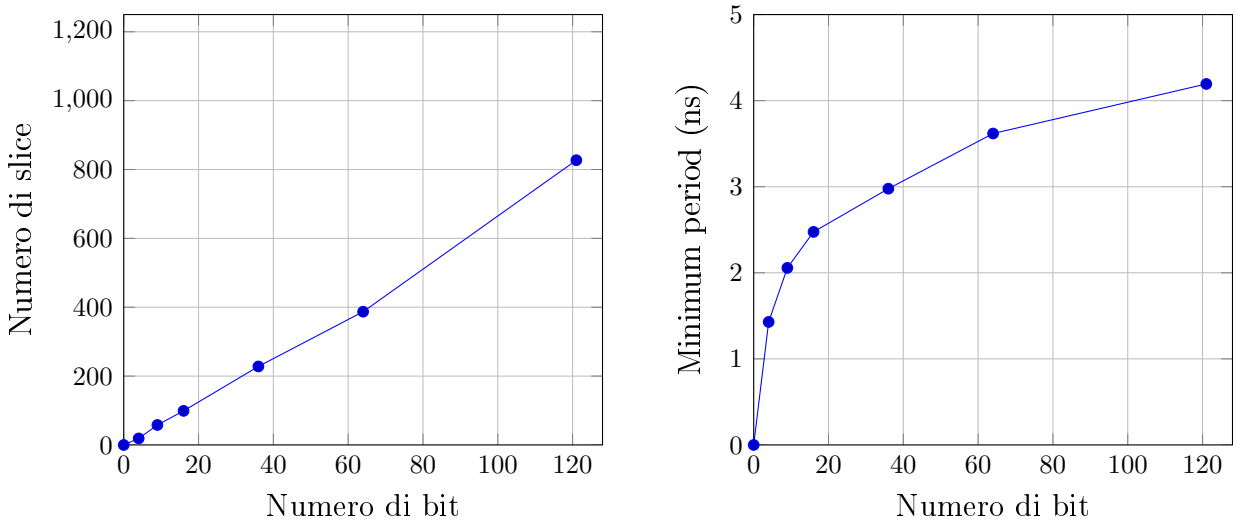

Figura 10.3: Grafici dei risultati ottenuti post-sintesi in funzione del numero di bit.

Si noti come non sono risultate particolari differenze rispetto all'utilizzo di un normale RCA: ciò è probabilmente dovuto al fatto che il tool di sintesi ottimizza il circuito e fa pieno utilizzo della matrice di interconnessione tra gli slices per migliorare le prestazioni del circuito.

Capitolo 11

Moltiplicatori

1. Realizzare in VHDL un circuito di moltiplicazione a celle MAC di N bit. La cella MAC deve contenere un Full Adder (descritto gia 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 signicativi, apportando eventuali commenti salienti.
2. (ozionale se si fa il 3) 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).
3. (opzionale se si fa il 2) 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).
4. (opzionale) Realizzare in VHDL un circuito per la moltiplicazione con la struttura di somma per righe, oppure somma per colonne oppure somma per diagonale. Discutere l'architettura al variare di N.

11.1 Moltiplicatore a celle MAC

11.1.1 Architettura

Il moltiplicatore a celle MAC è un componente in grado di moltiplicare due stringhe di bit tra loro. Data la sua architettura (mostrata in fig.11.1), tale componente è definito moltiplicatore parallelo, in quanto le somme parziali vengono calcolate contemporaneamente ad ogni passo.

Il moltiplicatore è stato realizzato mediante interconnessione di celle elementari dette celle MAC: tale componente si occupa di calcolare le somme parziali ad ogni stadio del moltiplicatore mediante l'utilizzo di:

- una porta *AND*, per effettuare il prodotto parziale $x_j y_i$;
- un *full adder*, per sommare tale prodotto con la somma parziale e il carry calcolati agli stadi precedenti.

L'architettura della cella è visibile in fig.11.1.

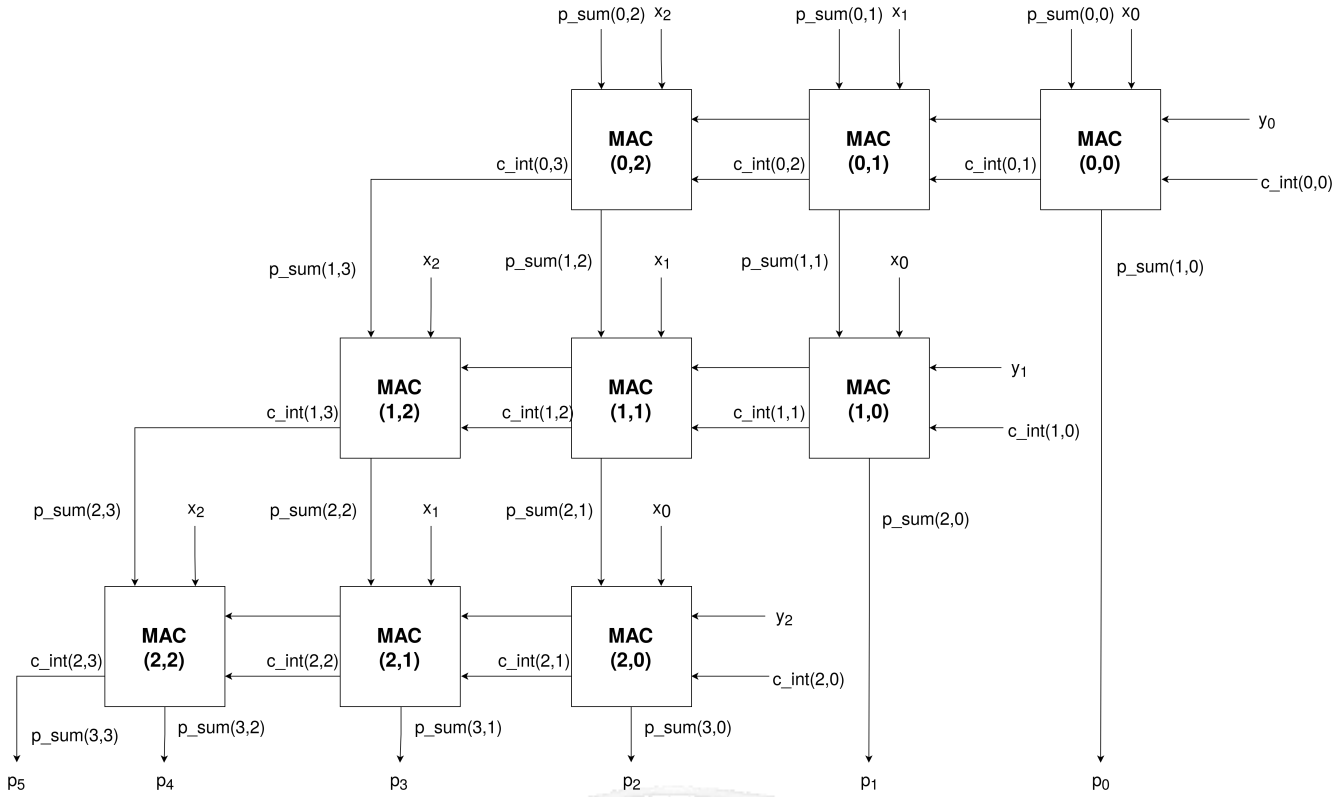

Figura 11.1: Architettura del moltiplicatore MAC.

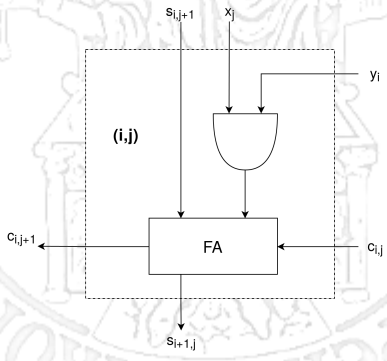

Figura 11.2: Architettura della cella MAC.

11.1.2 Implementazione

11.1.2.1 Cella MAC

La singola cella MAC è stata realizzata mediante descrizione dataflow:

```
architecture dataflow of mac_cell is
[...]
begin
  prodotto_xy <= X and Y;
  full_adder_inst : full_adder port map (
    X => prodotto_xy,
    Y => S_in,
    CIN => C_in,
    S => S_out,
    C => C_out
  );
end dataflow;
```

Codice Componente 11.1: Implementazione di una cella MAC.

L'implementazione completa è consultabile qui: mac_cell.vhd

11.1.2.2 Moltiplicatore

L'implementazione del moltiplicatore è stata realizzata mediante descrizione structural: sono state generate dapprima generate le celle MAC, collegate secondo il grafico visto precedentemente, e poi dalla matrice delle somme parziali sono stati ricavati i valori corrispondenti al risultato della moltiplicazione P. In particolare, i primi M bit sono ricavabili dalla prima colonna della matrice delle somme parziali, mentre i restanti N bit sono ricavabili dall'ultima riga della matrice, come mostrato qui:

```
architecture structural of mac_multiplier is
begin
  mac_rows : for i in 0 to M-1 generate
    mac_columns : for j in 0 to N-1 generate
      mac_cell_inst : mac_cell port map (
        X => X(j),
        Y => Y(i),
        C_in => carry_int (i,j),
        S_in => partial_sum(i,j+1),
        C_out   => carry_int(i,j+1),
        S_out   => partial_sum(i+1,j)
      );
    end generate;
    partial_sum (i+1,N) <= carry_int(i,N);
    end generate;
  result_M : for i in 0 to M-1 generate
      P(i) <= partial_sum(i+1,0);
  end generate;
  result_N : for i in 1 to N generate
```

```
20      P(i+M-1) <= partial_sum(M,i);
21    end generate;
22  end structural;
```

Codice Componente 11.2: Implementazione del moltiplicatore a celle MAC.

L'implementazione completa è consultabile qui: mac_multiplier.vhd

11.1.3 Simulazione e sintesi

Per tale componente è stata effettuata una simulazione behavioural, durante la quale sono stati fatti variare i due operandi da moltiplicare. I risultati ottenuti sono osservabili in fig.11.3.

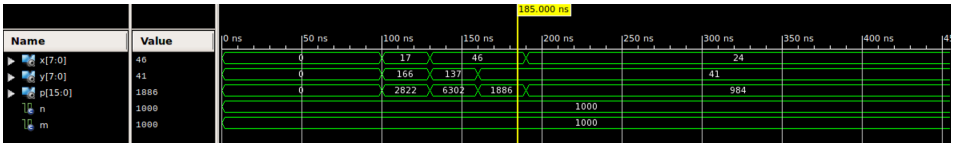

Figura 11.3: Simulazione behavioural del moltiplicatore MAC.

11.2 Moltiplicatore di Robertson

11.2.1 Architettura

Per poter realizzare un moltiplicatore di Robertson in grado di implementare l'algoritmo di Robertson si è fatto riferimento al modello PO/PC utilizzando i seguenti componenti:

- Unità di controllo
- Contatore
- Registro M (per contenere il moltiplicando X)
- Registri A e Q (per contenere rispettivamente i risultati parziali e il moltiplicatore Y)
- Un flip flop F
- Un addizionatore parallelo: Carry Select add/sub
- Porte XOR
- Un multiplexer

Lo schema strutturale è riportato in fig.11.4:

Essendo il moltiplicatore di Robertson una macchina sequenziale, sono stati dapprima definiti gli stati della macchina per la realizzazione della parte di controllo. L'automa a stati finiti è raffigurato in fig.11.5 e prevede i seguenti stati:

- Idle: la macchina permane in questo stato finchè non giunge un segnale di *start;*

- Init: in questo stato vengono inizializzati i registri e si resetta il contatore;
- Choice: in questo stato si sceglie l'operazione da fare in base al valore di *counter_hit, x_sign* e *q0*;
- Right_Shift: in questo stato viene effettuato lo shift dei registri A e Q e viene salvato il valore di *x_sign*;
- Add_Sub: in questo stato si effettua un'operazione di somma tra A (registro di accumulazione) ed M; nel caso in cui il moltiplicando X sia negativo e *counter_hit* sia alto (fine dell'operazione di moltiplicazione) si effettuerà una sottrazione A-M (operazione di correzione).

11.2.2 Implementazione

Per l'implementazione di tale componente si è utilizzata una descrizione di tipo structural, collegando opportunamene i componenti descritti precedentemente. In particolare si riporta l'interfaccia del componente moltiplicatore:

```
entity robertson_multiplier is
  GENERIC (N : INTEGER :=  8);
  PORT (
    X : in STD_LOGIC_VECTOR (N-1 downto 0);
    Y : in STD_LOGIC_VECTOR (N-1 downto 0);
    start : in STD_LOGIC;
    clock : in STD_LOGIC;
    reset_n : in STD_LOGIC;
    stop : out STD_LOGIC;
    Z : out STD_LOGIC_VECTOR ((2*N)-1 downto 0));
end robertson_multiplier;
```

Codice Componente 11.3: Implementazione del moltiplicatore di Robertson.

L'implementazione completa del moltiplicatore è consultabile qui: robertson_multiplier.vhd

In particolare, per quanto riguarda la parte di controllo, questa è stata realizzata mediante una control_unit implementata come FSM in descrizione behavioural, seguendo l'automa a stati finiti visto precedentemente. Di seguito è riportato il comportamento della macchina nello stato di *choice*:

```
when choice =>
  if counter_hit = '0' then     -- se operazione non terminata
           if current_multiplicand = '0' then    -- se q0=0
      nxt <= right_shift;      -- solo shift
    else                                  -- se q0=1
      nxt <= add_sub;          -- addizione
    end if;
  else                         -- se operazione terminata
    if x_sign = '0' then     -- se moltiplicando positivo
    nxt <= idle;             -- finito, torno in idle
```

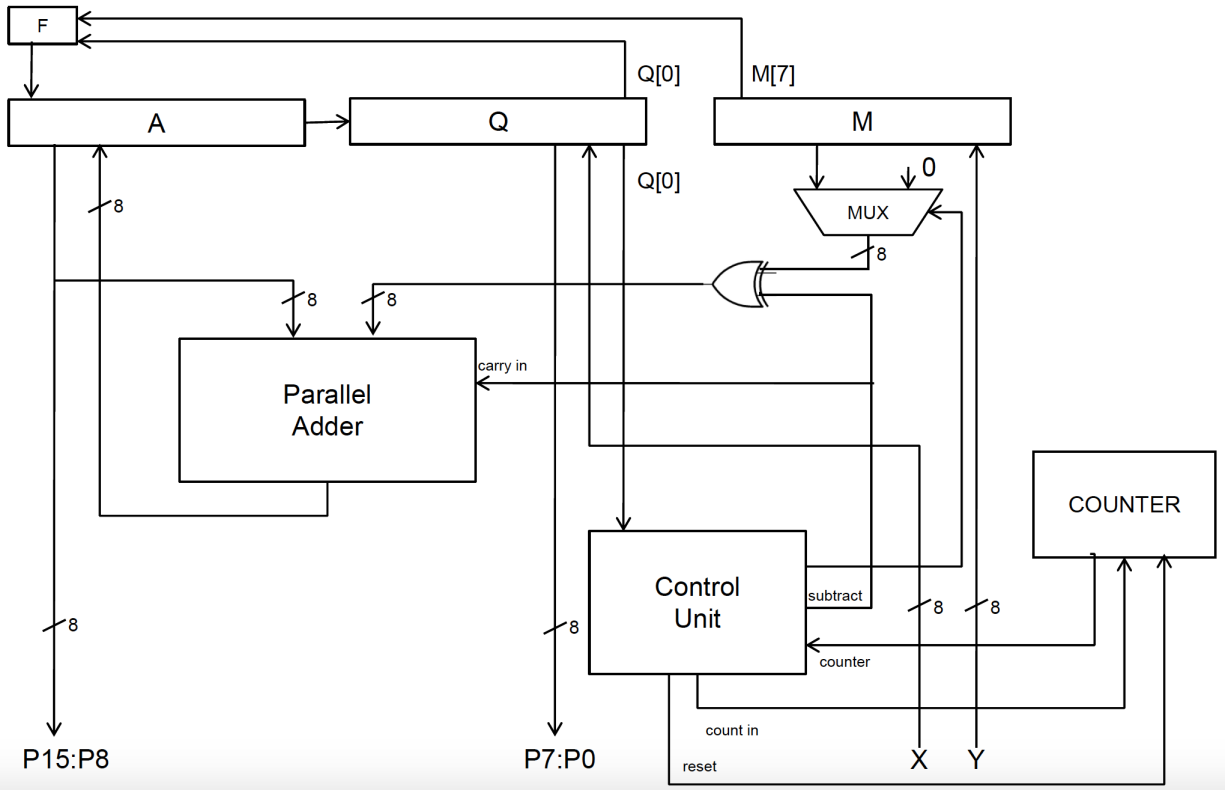

Figura 11.4: Architettura del moltiplicatore di Robertson.

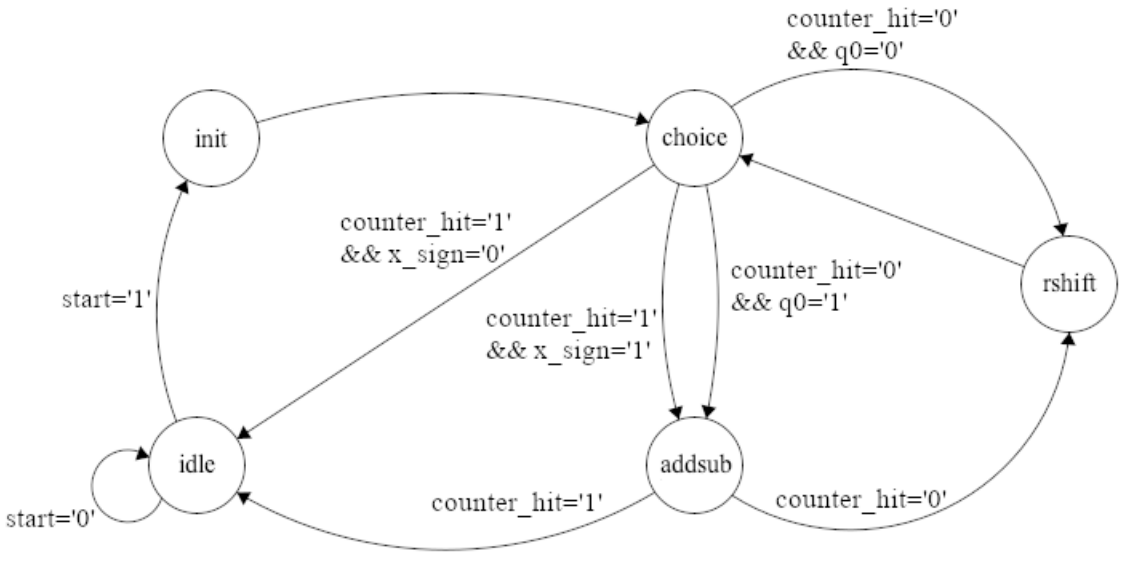

Figura 11.5: Automa a stati finiti (moltiplicatore di Robertson).

```
10    else                          -- se moltiplicando negativo
11      nxt <= add_sub;           -- operazione di correzione
12      end if;
13    end if;
```

Codice Componente 11.4: Implementazione dello stato di choice del moltiplicatore di Robertson.

L'intera implementazione dell'unita di controllo del moltiplicatore di Robertson è consultabile qui: robertson_control_unit.vhd

11.3 Moltiplicatore di Booth

11.4 Ripple carry multiplier (somma di righe)

11.4.1 Architettura

Il ripple carry multiplier è un componente che effettua il prodotto tra due stringhe di N e M bit rispettivamente tramite somma delle righe della matrice dei prodotti parziali. L'architettura del componente è visibile in fig.11.6).

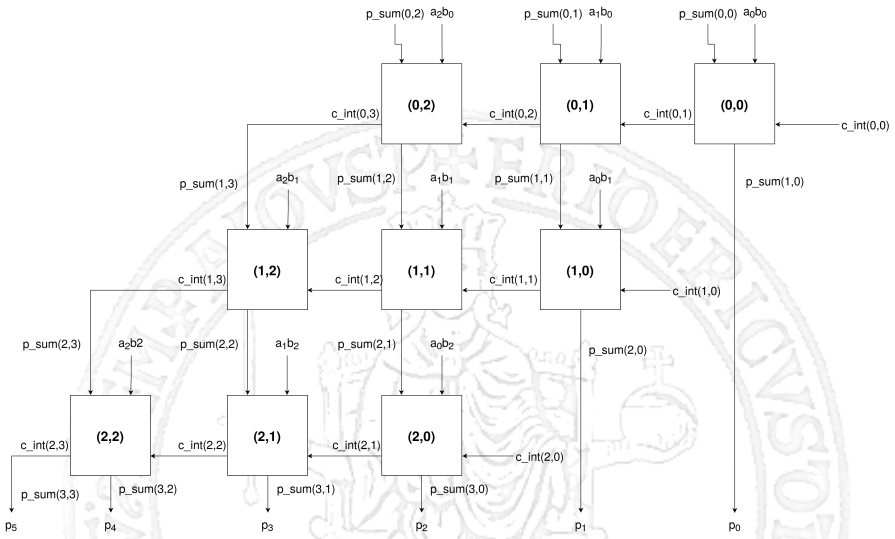

Figura 11.6: Architettura del Ripple Carry Multiplier.

Si noti come tale architettura può essere realizzata mediante l'utilizzo di M ripple carry adder in cascata, che andranno a sommare le N righe della matrice dei prodotti parziali con le somme parziali calcolate riga per riga. Tale schema può essere ulteriormente raffinato eliminando il primo (fig. 11.7). Si noti come sia possibile rimuovere dallo schema il primo RCA, che dovrebbe in teoria sommare la prima riga della matrice dei prodotti parziali con le somme parziali iniziali (pari a zero), semplicemente sommando direttamente le prime due righe della matrice.

11.4.2 Implementazione

L'implementazione del moltiplicatore è stata realizzata mediante descrizione structural: dapprima si è generata la matrice dei prodotti parziali, e poi si sono generati i RCA nel seguente modo:

```
gen_rcmultiplier: for i in 0 to N-1 generate
    gen_first_sum: if i=0 generate
      partial_sum(i+1)(M-1 downto 0) <= partial_prod(i)(M-1 downto 0);
    end generate;

    gen_rca: if i>0 generate
      rca: rippleCarry_adder PORT MAP(
        X => partial_prod(i)(M-1 downto 0),
        Y => partial_sum(i)(M downto 1),
        c_in =>'0',
        s => partial_sum(i+1)(M-1 downto 0),
        c_out => partial_sum(i+1)(M)
        );
    end generate;
end generate
```

Codice Componente 11.5: Implementazione del Ripple Carry Multiplier.

Si noti come si è evitato di generare il primo RCA portando direttamente la prima riga della matrice dei prodotti parziali nella matrice delle somme parziali. Infine, si è prelevato il risultato come specificato nel grafico: i primi M bit sono ricavati dalla prima colonna della matrice delle somme parziali, mentre i restanti N bit sono ricavati dall'ultima riga della matrice. L'implementazione completa del moltiplicatore è consultabile qui: ripple_carry_multiplier.vhd

11.4.3 Simulazione e sintesi

Per tale componente è stata effettuata una simulazione behavioural, durante la quale sono stati fatti variare i due operandi da moltiplicare. I risultati ottenuti sono osservabili in fig.11.8.

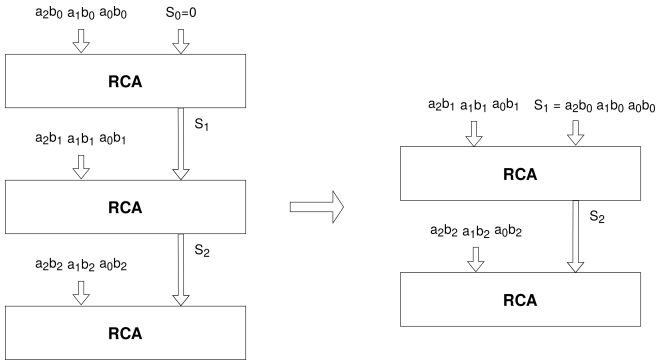

Figura 11.7: Ripple Carry Multiplier realizzato tramite RCA.

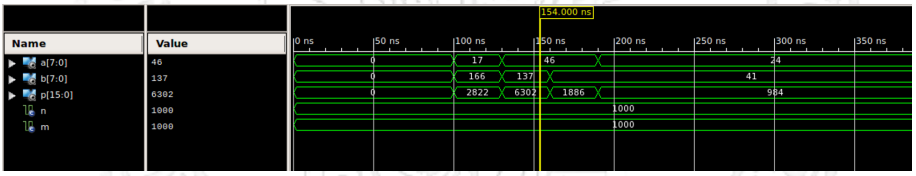

Figura 11.8: Simulazione behavioural del moltiplicatore RCM.