

Tesina di Architettura dei Sistemi di Elaborazione

Gruppo 14

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

February 12, 2019

Contents

1	Sintesi di macchine combinatorie	1
1.1	Traccia	1
1.2	Soluzione	1
1.2.1	Esercizio 1	1
1.2.2	Esercizio 2	1
1.2.3	Esercizio 3	4
1.2.4	Esercizio 4	5
1.2.5	Esercizio 5	6
1.2.6	Esercizio 6	7
2	Latch e Flip-Flop	10
2.1	Latch RS	10
2.1.1	Traccia	10
2.1.2	Soluzione	10
2.1.2.1	Implementazione	10
2.1.2.2	Simulazione	11
2.2	Latch T	12
2.2.1	Traccia	12
2.2.2	Soluzione	12
2.2.2.1	Implementazione	12
2.2.2.2	Simulazione	13
2.3	Latch JK	14
2.3.1	Traccia	14
2.3.2	Soluzione	14
2.3.2.1	Implementazione	14
2.3.2.2	Simulazione	14
2.4	Flip-flop D edge-triggered	15
2.4.1	Traccia	15
2.4.2	Soluzione	15
2.4.2.1	Implementazione	15
2.4.2.2	Simulazione	16
2.5	Flip-flop RS master-slave	16
2.5.1	Traccia	16
2.5.2	Soluzione	16
2.5.2.1	Implementazione	16
2.5.2.2	Simulazione	16

3	Display a 7 segmenti	19
3.1	Architettura	19
3.2	Clock divisor e contatore modulo 2^n	20
3.2.1	Contatore modulo 2^n	20
3.2.2	Clock divisor	20
3.3	Anodes manager	21
3.4	Cathodes manager	21
3.5	Display on-board	22

Chapter 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.

[illegible]

Tra il comando `print_stats` è possibile osservare come il 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.

2

```

UC Berkeley, SIS 1.3 (compiled 20-Mar-00 at 1:56 PM)
sis> read_blif Esercitazionale_2_Y2.blif
sis> simplify
sis> write eqn
INORDER = X5 X4 X3 X2 X1 X0;
OUTORDER = Y2;
Y2 = X5*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 Esercitazionale_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*X1*X2*X1*X0 + X5*X1*X3*X2*X1*X1*X0 + X4*X3*X2*X1*X1*
X0 + X5*X3*X2*X1*X1*X0 + X4*X1*X3*X2*X1*X1*X0 + X5*X1*X4*X3*X1*X1*X0 + X5*X1*X4*X3*X1*X2*
X0 + X5*X4*X3*X1*X2*X1*X0 + X5*X4*X1*X3*X2*X1*X0 + X5*X1*X4*X1*X2*X1*X1*X0 + X5*X4*X1*X2*
X1*X0 + X4*X1*X3*X2*X1*X1*X0 + X5*X1*X3*X2*X1*X1*X0 + X5*X1*X4*X3*X1*X1*X0 + X5*X1*X4*
X2*X1*X0 + X5*X1*X4*X3*X1*X0 + X5*X1*X4*X1*X3*X2*X1 + X5*X4*X1*X3*X1*X2*X1;
sis> read_blif Esercitazionale_2_Y0.blif
sis> simplify
sis> write eqn
INORDER = X5 X4 X3 X2 X1 X0;
OUTORDER = Y0;
Y0 = X5*X1*X4*X1*X3*X2*X1*X1*X0 + X5*X4*X1*X3*X2*X1*X1*X0 + X5*X1*X4*X3*X1*X2*X1*X1*
X0 + X5*X4*X3*X1*X2*X1*X1*X0 + X5*X1*X4*X1*X3*X2*X1*X1*X0 + X5*X4*X1*X3*X2*X1*X1*X0 + X5*
X4*X3*X2*X1*X1*X0 + X5*X4*X3*X2*X1*X1*X0 + X5*X1*X4*X1*X3*X1*X2*X1*X1*X0 + X5*X4*X1*X3*
X2*X1*X1*X0 + X5*X1*X4*X3*X1*X2*X1*X1*X0 + X5*X1*X4*X3*X1*X2*X1*X1*X0 + X5*X1*X4*X3*
X1*X2*X1*X1*X0 + X5*X4*X1*X3*X2*X1*X1*X0 + X5*X1*X4*X3*X2*X1*X1*X0 + X5*X4*X3*X2*X1*X1*
X0 + X5*X1*X4*X1*X3*X2*X1*X1*X0 + X5*X4*X1*X3*X2*X1*X1*X0 + X5*X1*X4*X3*X2*X1*X1*X0 +
X5*X4*X3*X2*X1*X1*X0 + X5*X1*X4*X1*X3*X2*X1*X1*X0 + X5*X4*X1*X3*X2*X1*X1*X0 + X5*X1*X4*X3*
X2*X1*X1*X0 + X5*X4*X1*X3*X2*X1*X1*X0 + X5*X1*X4*X1*X3*X2*X1*X1*X0 + X5*X4*X1*X3*X2*X1*X1*
X0 + X5*X4*X3*X2*X1*X1*X0;
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	✓
37:	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	✓

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
>>> before removing serial inverters <<<
# of outputs: 3
total gate area: 244.00
maximum arrival time: (16.41,16.41)
maximum po slack: (-13.21,-13.21)
minimum po slack: (-16.41,-16.41)
total neg slack: (-43.76,-43.76)
# of failing outputs: 3
>>> before removing parallel inverters <<<
# of outputs: 3
total gate area: 196.00
maximum arrival time: (15.51,15.51)
maximum po slack: (-12.31,-12.31)
minimum po slack: (-15.51,-15.51)
total neg slack: (-41.13,-41.13)
# of failing outputs: 3
# of outputs: 3
total gate area: 196.00
maximum arrival time: (15.51,15.51)
maximum po slack: (-12.31,-12.31)
minimum po slack: (-15.51,-15.51)
total neg slack: (-41.13,-41.13)
# of failing outputs: 3
sis> print_map_stats
Total Area = 196.00
Gate Count = 72
Buffer Count = 0
Inverter Count = 6
Most Negative Slack = -15.51
Sum of Negative Slacks = -41.13
Number of Critical PO = 3

sis> map -W -m 0 -s
>>> before removing serial inverters <<<
# of outputs: 3
total gate area: 161.00
maximum arrival time: (21.50,21.50)
maximum po slack: (-14.30,-14.30)
minimum po slack: (-21.50,-21.50)
total neg slack: (-53.20,-53.20)
# of failing outputs: 3
>>> before removing parallel inverters <<<
# of outputs: 3
total gate area: 161.00
maximum arrival time: (21.50,21.50)
maximum po slack: (-14.30,-14.30)
minimum po slack: (-21.50,-21.50)
total neg slack: (-53.20,-53.20)
# of failing outputs: 3
# of outputs: 3
total gate area: 159.00
maximum arrival time: (21.30,21.30)
maximum po slack: (-14.30,-14.30)
minimum po slack: (-21.30,-21.30)
total neg slack: (-53.00,-53.00)
# of failing outputs: 3
sis> print_map_stats
Total Area = 159.00
Gate Count = 52
Buffer Count = 0
Inverter Count = 9
Most Negative Slack = -21.30
Sum of Negative Slacks = -53.00
Number of Critical PO = 3

sis> map -W -m 0.5 -s
>>> before removing serial inverters <<<
# of outputs: 3
total gate area: 223.00
maximum arrival time: (19.10,19.10)
maximum po slack: (-15.50,-15.50)
minimum po slack: (-19.10,-19.10)
total neg slack: (-50.10,-50.10)
# of failing outputs: 3
>>> before removing parallel inverters <<<
# of outputs: 3
total gate area: 169.00
maximum arrival time: (17.90,17.90)
maximum po slack: (-14.30,-14.30)
minimum po slack: (-17.90,-17.90)
total neg slack: (-46.50,-46.50)
# of failing outputs: 3
# of outputs: 3
total gate area: 169.00
maximum arrival time: (17.90,17.90)
maximum po slack: (-14.30,-14.30)
minimum po slack: (-17.90,-17.90)
total neg slack: (-46.50,-46.50)
# of failing outputs: 3
sis> print_map_stats
Total Area = 169.00
Gate Count = 58
Buffer Count = 0
Inverter Count = 10
Most Negative Slack = -17.90
Sum of Negative Slacks = -46.50
Number of Critical PO = 3

```

Figure 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*!X3*!X2*[10] + X4*!X3*[9]*[10] + !Y2*!Y0*[8] + !Y2*[3]*[6] + !Y2*[10] + !Y2*[9];
Y0 = !X5*[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>
    
```

Figure 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.

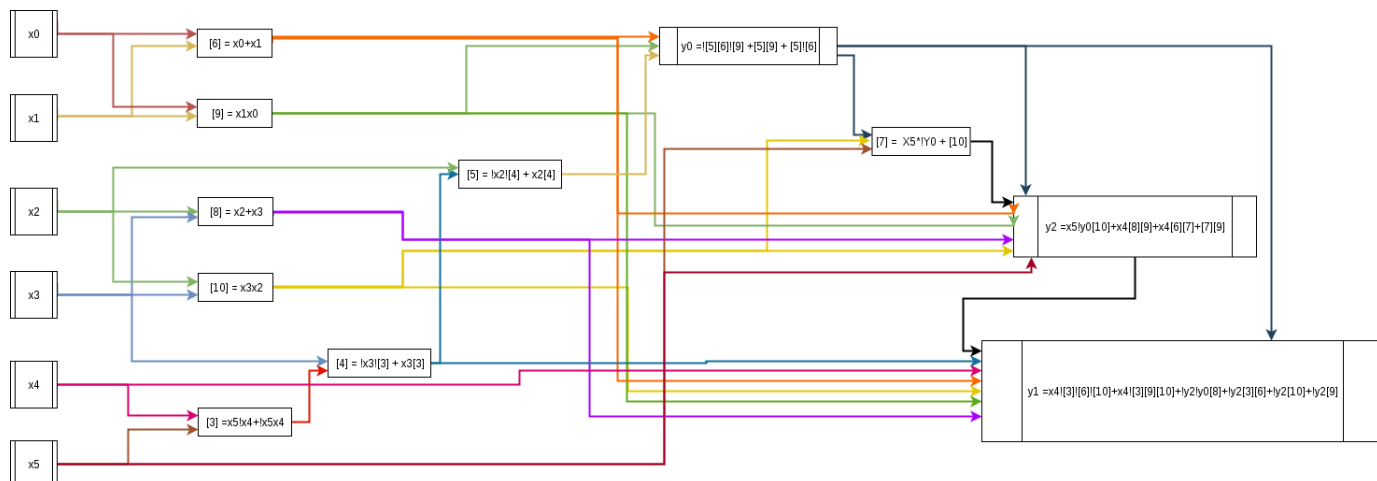


Figure 1.6: Grafo della funzione minimizzata tramite *rugged.script*.

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.

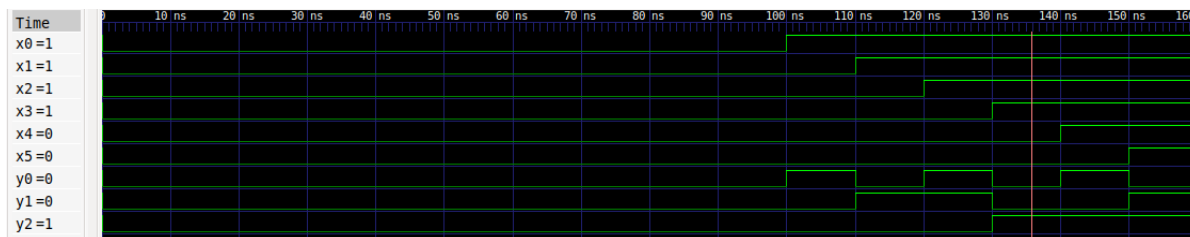


Figure 1.7: Simulazione della macchina M in *gtkwave*.

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, mentre 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.

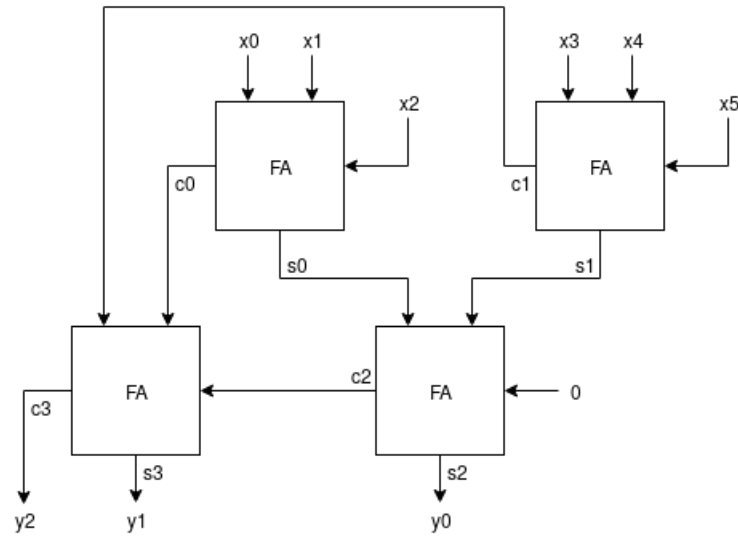


Figure 1.8: Schema della macchina M a partire da blocchi full-adder.

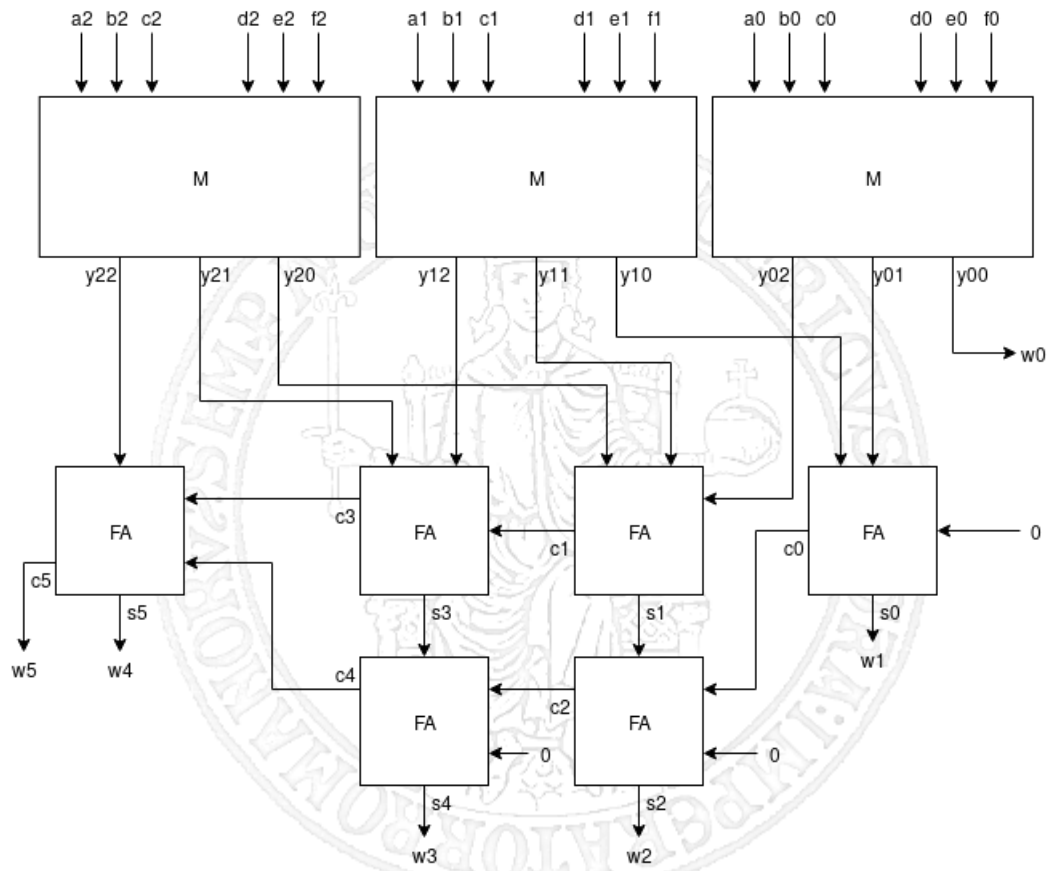


Figure 1.9: Schema della macchina S.

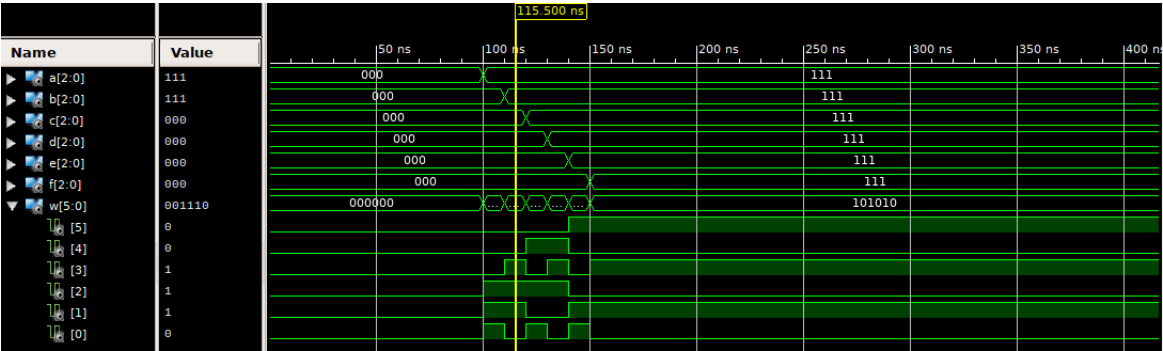
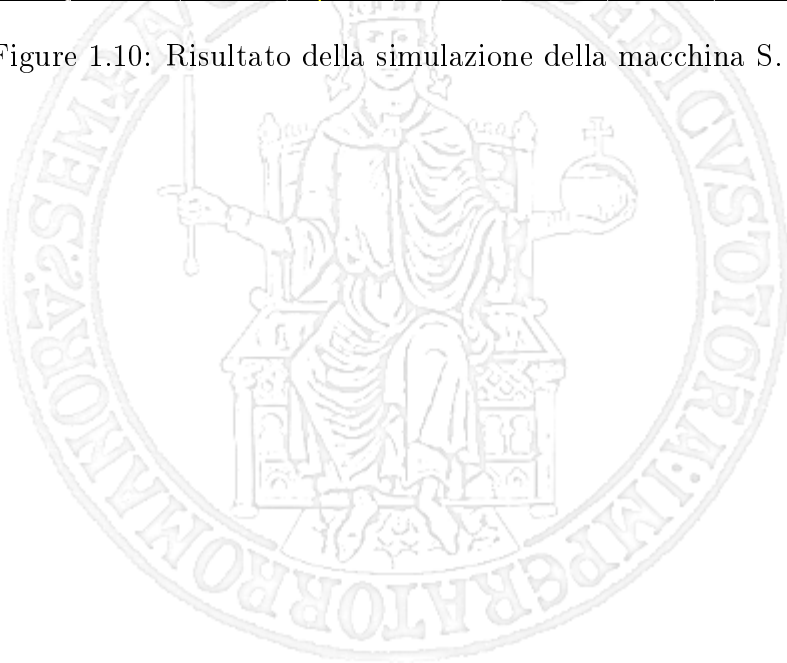


Figure 1.10: Risultato della simulazione della macchina S.



Chapter 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 collegando opportunamente le porte NOR e AND, queste ultime necessarie per l'abilitazione. Lo schema di collegamento è riportato in fig.2.1.

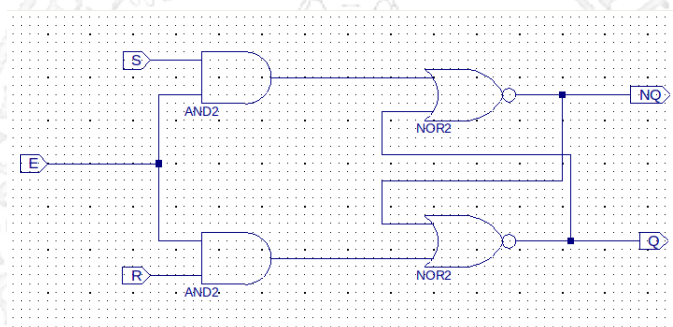


Figure 2.1: Schematico del latch RS abilitato.

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 correttamente in quanto per come è definito NQ deve valere che “ $Q \text{ XOR } NQ = 1$ ”. 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 circuito¹ 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 oscillazioni 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.

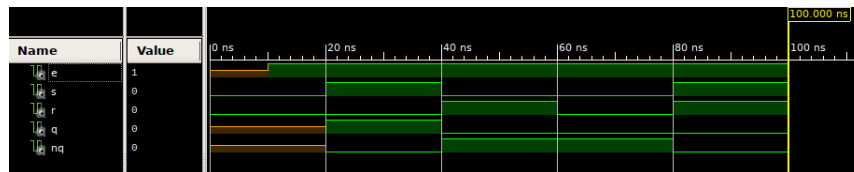


Figure 2.2: Simulazione behavioural del latch RS.

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.

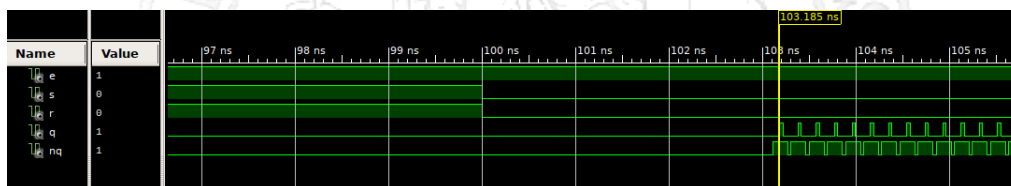


Figure 2.3: Simulazione post-map del latch RS.

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

¹ritardi 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 oscillazione di Q e NQ sono praticamente uguali. 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 utilizzando 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.

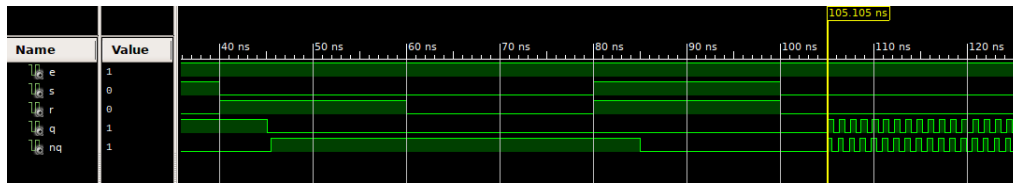


Figure 2.4: Simulazione post-route del latch RS su board Nexys 4 DDR.

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 mostra quello che accade nel caso reale.

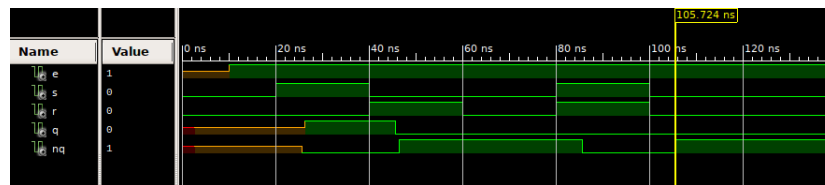


Figure 2.5: Simulazione post-route del latch RS su board Basys 2.

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.

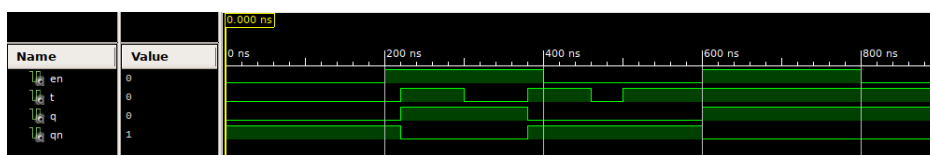


Figure 2.6: Simulazione behavioural del latch T abilitato.

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.

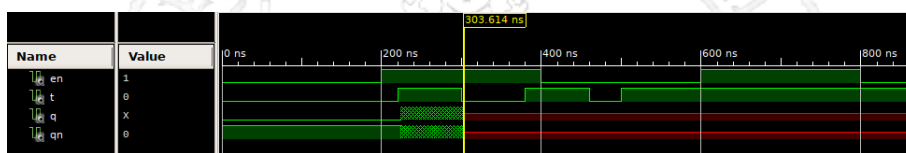


Figure 2.7: Simulazione post-map del latch T abilitato.

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.

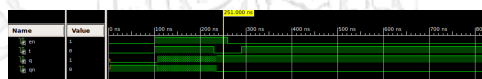


Figure 2.8: Simulazione post-route del latch T abilitato.

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 viene portato 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.

```

1 architecture behavioral of latch_jk is
2   signal Qtemp: std_logic := '0';
3   begin
4     p: process(J,K) is
5       begin
6         if (J='1' and K='0') then
7           Qtemp<='1';
8         else
9           if (J='0' and K='1') then
10            Qtemp<='0';
11          else
12            if (J='1' and K='1') then
13              Qtemp<= not Qtemp;
14            end if;
15          end if;
16        end if;
17      end process;
18      Q <= Qtemp;
19      Qnot <= not Qtemp;
20 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.

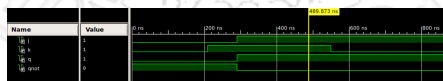


Figure 2.9: Simulazione behavioural del latch JK.

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.

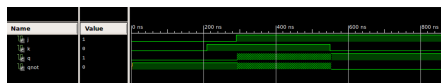


Figure 2.10: Simulazione post-map del latch JK.

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.

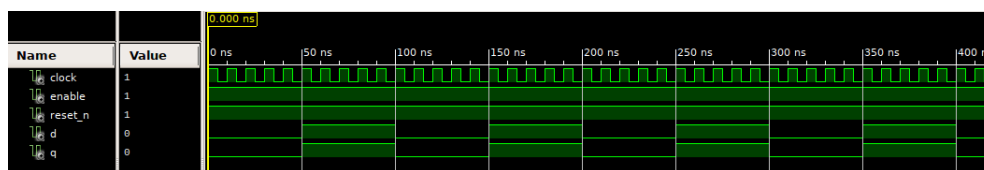


Figure 2.11: Simulazione behavioural del flip-flop RS master-slave.

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.

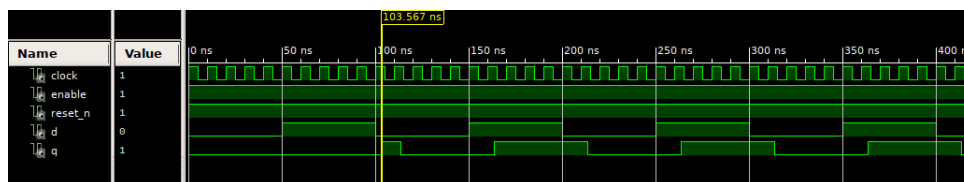


Figure 2.12: Simulazione post-map del flip-flop RS master-slave.

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).



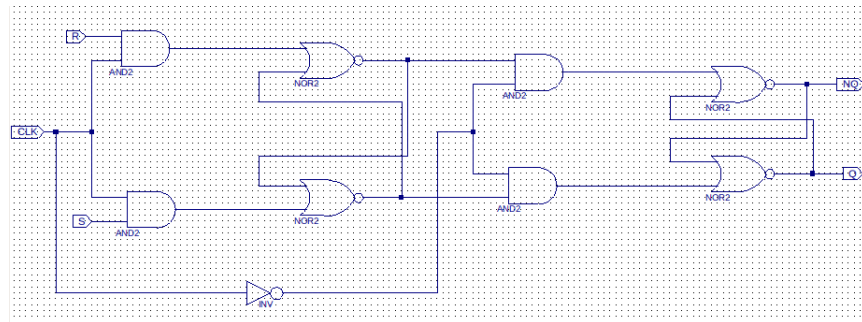


Figure 2.13: Schematico del flip-flop RS master-slave.

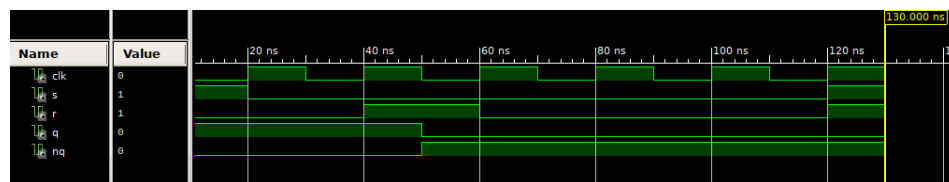


Figure 2.14: Simulazione behavioural del flip-flop RS master-slave.

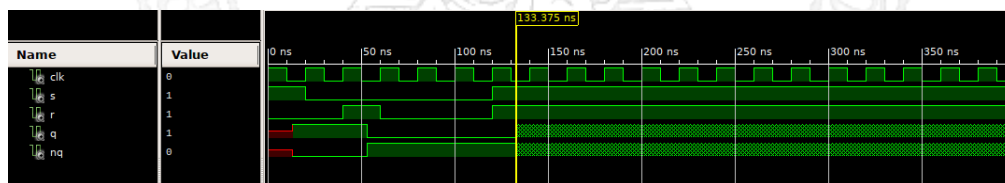


Figure 2.15: Simulazione post-map del flip-flop RS master-slave.

Chapter 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

L'architettura del componente è mostrata in fig.3.1. 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 pressione pulsante);
- *values* - segnale di 16 bit per detereminare il valore da visualizzare sul display;
- *dots* - segnale di abilitazione su 4 bit per l'abilitazione dei punti decimali sul display;
- *enable_digit* - segnale di abilitazione su 4 bit per l'abilitazione degli 4 anodi corrispondenti alle 4 cifre sul display (logica 1-attivo);

I segnali in uscita saranno invece:

- *anodes* - segnale per l'abilitazione delle 4 cifre del display (0-attivo);
- *cathodes* - segnale per l'abilitazione dei segmenti di ogni cifra (0-attivo);

Il componente è stato realizzato tramite descrizione schematica: in particolare, i componenti realizzati ed utilizzati sono:

- *clock_divisor* - divisore di frequenza per il clock;
- *counter_mod2n* - contatore modulo 2^n , con $n=2$, per la selezione della cifra da attivare;
- *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`.

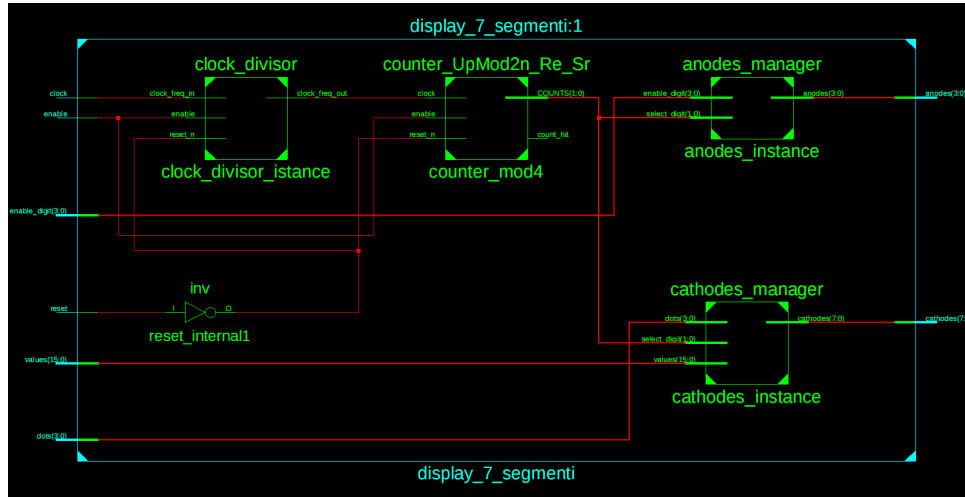


Figure 3.1: Schematico del latch RS abilitato.

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 $\text{clock_frequency_in} / \text{clock_frequency_out} - 1$). 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 (a partire dal modello Nexys 4), 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 mantenere attivi solo gli anodi abilitati. 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 0 in uscita (caso non previsto dal normale funzionamento degli anodi) posso riconoscere subito la presenza di comportamenti imprevisti nel componente.

```

1 architecture dataflow of anodes_manager is
2   signal anodes_switching : STD_LOGIC_VECTOR (3 downto 0) := (others =>
3     '0');
4   begin
5     anodes <= not anodes_switching OR not enable_digit;
6     with select_digit select anodes_switching <=
7       x"1"    when "00",
8       x"2"    when "01",
9       x"4"    when "10",
10      x"8"    when "11",
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 0, poichè i catodi sono pilotati da segnali 0-attivi. Il componente prende in ingresso:

- il valore *counter* fornito dal contatore (come l'anodes manager);
- il segnale *values* (16 bit) per determinare il valore di ogni cifra e dunque i segmenti da accendere;

- il segnale *dots* (4 bit) per determinare quali dei 4 punti decimali accendere.

In uscita abbiamo un segnale 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 della variabile *nibble*¹ corrispondenti alla giusta *digit*, ossia 4 bit di values corrispondenti alla cifra selezionata;
- *decoder* - in base alla *digit* presente nel *nibble*, imposta *cathodes_for_digit* al valore necessario per accendere i segmenti corretti. Tali valori sono espressi come costanti e ricavabili dal reference manual.

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 si aggiunge *cathodes_for_digit*.

```
1 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 on-board

Osserviamo il funzionamento del display a 7 segmenti su board. Per poter caricare l'architettura descritta sulla board Nexys 4, si è realizzata un'entità di più alto di livello denominata *display_onBoard*. Inoltre, è stato necessario l'utilizzo del file Nexys4DDR_master.ucf per effettuare il mapping tra i componenti fisici della board e i segnali di ingresso/uscita. In particolare, i valori di *cathodes* e *anodes* corrispondono alle prime 4 digit del display presente sulla board, mentre le digit rimanenti sono state disattivate manualmente. Per quanto concerne gli ingressi, invece, i valori del segnale *values* vengono determinati tramite l'utilizzo degli switch presenti sulla board.

```
1 architecture structural of display_onBoard is
2   [...]
3   alias anodes_0_3 is anodes (3 downto 0); -- i bit da 3 a 0 di values
4   alias anodes_4_7 is anodes (7 downto 4); -- i bit da 4 a 7 di values
5   begin
6     display: display_7_segmenti port map (
7       clock => clock,
8       reset => reset,
```

¹Un nibble è una stringa di 4 bit.

```
9      values => values,  
10     dots => dots,  
11     enable_digit => enable_digit,  
12     anodes => anodes_0_3,  
13     cathodes => cathodes );  
14     anodes_4_7 <= (others => '1');  
15 end architectural;
```

Codice Componente 3.3: Determinazione del valore di cathods..

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

