Politecnico di Torino

III Facoltà di Ingegneria

Sistemi Elettronici a Basso Consumo

Laurea Magistrale in Ingegneria Elettronica

Relazione laboratori

Gruppo 16

Autori:

Angelo Balaara 257582
Leonardo Bellettini 260347
Edoardo Bollea 262968

10 luglio 2019

Indice

CAPITOLO 1

Stima di potenza: tecniche probabilistiche

1.1 Calcolo della probabilità e della *Switching Activity*: semplici porte logiche

Si sono calcolate le probabilità e le *Switching Activity* delle quattro porte logiche fondamentali supponendo gli ingressi scorrelati, indipendenti ed equiprobabili. Per il calcolo dell'attività si è utilizzata sempre la stessa formula: $A(Y) = 2P^1(1 - P^1)$. In tabella 1.1 si riportano i risultati ottenuti.

	NOT	AND	OR	XOR
$P(Y = 1)$	P = $\frac{1}{2}$	$P_a \cdot P_b = \frac{1}{4}$	$1 - (1 - P_a)(1 - P_b) = \frac{3}{4}$	$P_A(1 - P_b) + P_b(1 - P_a) = \frac{1}{2}$
A(Y)	$\frac{1}{2}$	$\frac{3}{8}$	$\frac{3}{8}$	$\frac{1}{2}$

Tabella 1.1: Probabilità e attività ideali

Dopo aver eseguito la simulazione su Modelsim, è stato possibile stimare il numero di commutazioni per ognuna delle porte della tabella 1.1: il comando *power report* consente infatti di ottenere un file contente il *toggle count* di ciascun nodo simulato. La simulazione è stata eseguita più volte in successione usando uno script *.do*, variando di volta in volta il numero di colpi di clock. I risultati sono visibili in Tabella 1.2.

Tc(CK)	Tc(INV)	Tc(AND)	Tc(OR)	Tc(XOR)
20	4	3	10	9
200	99	76	86	92
2000	994	804	710	1004
20000	9962	7415	7560	9923
200000	99971	75002	74963	99959

Tabella 1.2: *Toggle count* simulati

Dividendo il numero di transizioni dell'uscita di ciascuna porta per il numero di colpi di clock, è stato possibile ottenere una stima della *Switching Activity*:

Tc(CK)	A(INV)	A(AND)	A(OR)	A(XOR)
20	0.2	0.15	0.5	0.45
200	0.495	0.38	0.43	0.46
2000	0.497	0.402	0.335	0.502
20000	0.498	0.371	0.378	0.496
200000	0.5	0.375	0.375	0.5
A_{ideale}	0.5	0.375	0.375	0.5

Tabella 1.3: *Switching activity* simulate

Dai valori ottenuti si nota che, all'aumentare del numero di colpi di clock di simulazione, il risultato si avvicini a quello ideale mostrato nella tabella 1.1: questo è dovuto alla generazione dei valori in ingresso che è solo pseudo-casuale e quindi, se si considerano poche iterazioni dell'*LFSR*, fornisce risultati leggermente diversi.

1.2 Calcolo della probabilità e della *Switching Activity*: *Half* e *Full Adder*

Per calcolare la probabilità e la *Switching Activity* dell'*Half Adder* e del *Full Adder* si è scritta la tavola di verità di ciascuna uscita e da essa si è derivata la formula della probabilità. Si è poi utilizzata la stessa formula usata nella sezione precedente per il calcolo dell'attività.

a	b	S	C_o
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

Tabella 1.4: Tavola di verità dell'*Half Adder*

Si nota che le uscite sono rispettivamente una XOR e una AND. Si possono usare le stesse formule calcolate in precedenza.

C_{in}	a	b	S	C_o
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

Tabella 1.5: Tavola di verità del *Full Adder*

Dalla tabella 1.5 si possono ricavare le seguenti formule per la probabilità:

$$P(S=1) = P_b(1-P_a)(1-P_{C_{in}}) + P_a(1-P_b)(1-P_{C_{in}}) + P_{C_{in}}(1-P_a)(1-P_b) + P_aP_bP_{C_{in}}$$

$$P(C_o=1) = P_aP_b(1-P_{C_{in}}) + P_bP_{C_{in}}(1-P_a) + P_aP_{C_{in}}(1-P_b) + P_aP_bP_{C_{in}}$$

I risultati numerici delle formule appena ricavate sono riportati in tabella 1.6.

	Half Adder	*Full Adder*
$P(S=1)$	$\frac{1}{2}$	$\frac{1}{2}$
$P(C_o=1)$	$\frac{1}{4}$	$\frac{1}{2}$
$A(S)$	$\frac{1}{2}$	$\frac{1}{2}$
$A(C_o)$	$\frac{3}{8}$	$\frac{1}{2}$

Tabella 1.6: Probabilità e attività ideali

Al fine di rispondere in modo corretto a una richiesta sul *Ripple Carry Adder* si sono ricalcolate le probabilità e le attività suppondendo che gli ingressi fossero scorrelati ma con probabilità diversa da 0.5.
Posto quindi $P(a=1) = 0.4$, $P(b=1) = 0.6$ e $P(C_{in}=1) = 0.5$, si ha:

	Half Adder	*Full Adder*
$P(S=1)$	0.52	$\frac{1}{2}$
$P(C_o=1)$	0.24	$\frac{1}{2}$
$A(S)$	0.499	$\frac{1}{2}$
$A(C_o)$	0.3648	$\frac{1}{2}$

Tabella 1.7: Probabilità e *Switching Activity*con ingressi non equiprobabili

Il dato rilevante della tabella 1.7 è che la probabilità del *Carry Out* del *Full Adder*non cambia e rimane sempre 0.5.
Si è quindi studiato il comportamento del *Ripple Carry Adder* i cui valori di probabilità e *Switching Activity* sono riportati in tabella 1.8.

	Stadio 0	Stadio 1	Stadio 2	Stadio 3	Stadio 4	Stadio 5	Stadio 6	Stadio 7
$P(S=1)$	$\frac{1}{2}$	$\frac{1}{2}$	$\frac{1}{2}$	$\frac{1}{2}$	$\frac{1}{2}$	$\frac{1}{2}$	$\frac{1}{2}$	$\frac{1}{2}$
$A(S)$	$\frac{1}{2}$	$\frac{1}{2}$	$\frac{1}{2}$	$\frac{1}{2}$	$\frac{1}{2}$	$\frac{1}{2}$	$\frac{1}{2}$	$\frac{1}{2}$

Tabella 1.8: Probabilità e *Switching Activity* ideali per un *Ripple Carry Adder*

Essi sono tutti uguali a quelli calcolati per il singolo *Full Adder*. Le probabilità degli ingressi di ogni elemento della catena sono infatti uguali a quelle del *Full Adder* valutato singolarmente: gli ingressi sono tutti scorrelati e equiprobabili per ipotesi e il *Carry In*, essendo il *Carry Out* di un altro *Full Adder*, è anch'esso equiprobabile e scorrelato, poiché generato a partire da valori casuali.
Se si considerano ingressi non più equiprobabili il risultato resta lo stesso presentato in tabella 1.8:

non cambiando la probabilità del *Carry Out*, come visto in tabella 1.7, le condizioni dei *Full Adder* nella catena sono le stesse del caso isolato, riportando al caso appena descritto.
Si sono quindi simulati due *Ripple Carry Adder* con il file di testbench fornito. La differenza tra i due, dal punto di vista del comportamento, riguarda la propagazione del *Carry Out*: in un caso si propaga in uno step di simulazione, quindi in un tempo infinitesimo, mentre nel secondo ha un tempo fissato da DRCAC=25ps. La presenza di un ritardo dell'ordine della decina di picosendo ha reso necessaria una riduzione della risoluzione della simulazione che è stata infatti impostata a 1ps.
In tabella 1.9 sono riassunti i *Toggle Count* tramite il comando *power report* avendo impostato un numero di colpi di clock pari a 200. Si sono calcolate anche le rispettive *Switching Activity*.

		C_o	$S(7)$	$S(6)$	$S(5)$	$S(4)$	$S(3)$	$S(2)$	$S(1)$	$S(0)$
DRCAC=0	Tc	123	84	102	105	85	98	101	102	92
	A	0.615	0.42	0.51	0.525	0.425	0.49	0.505	0.51	0.46
DRCAC=25ps	Tc	123	250	242	213	199	210	201	178	92
	A	0.615	1.25	1.21	1.07	0.995	1.05	1	0.89	0.46

Tabella 1.9: *Toggle Count* e *Switching Activity* per *Ripple Carry Adder*, con e senza ritardo sul *Carry Out*

I risultati ottenuti per il primo *Ripple Carry Adder* sono coerenti con quanto ricavato analiticamente, considerando che, come visto per le semplici porte logiche, sarebbero necessari più colpi di clock per avere una stima corretta.
Il secondo *Ripple Carry Adder* presenta un numero di toggle crescente con la posizione nella catena del *Full Adder*: questo è dovuto al ritardo inserito sul *Carry Out*. L'uscita di somma di ogni *Full Adder* cambia appena cambiano i suoi ingressi, ma cambierà nuovamente quando il *Carry* si sarà propagato dall'ingresso fino a quel punto della catena, cosa che non avviene istantaneamente a causa del ritardo inserito. Questo comporta un incremento sostanziale dell'attività delle uscite del sommatore.
Dalla simulazione grafica è stato estrapolato un frammento significativo riportato in figura 1.1. In esso è facilmente visibile la differenza (ad esempio nella *Switching Activity* di S2) dovuta alle presenza del ritardo nel secondo *Ripple Carry Adder*: l'uscita presenta infatti una moltitudine di transizioni spurie, non presenti nell'altro sommatore.

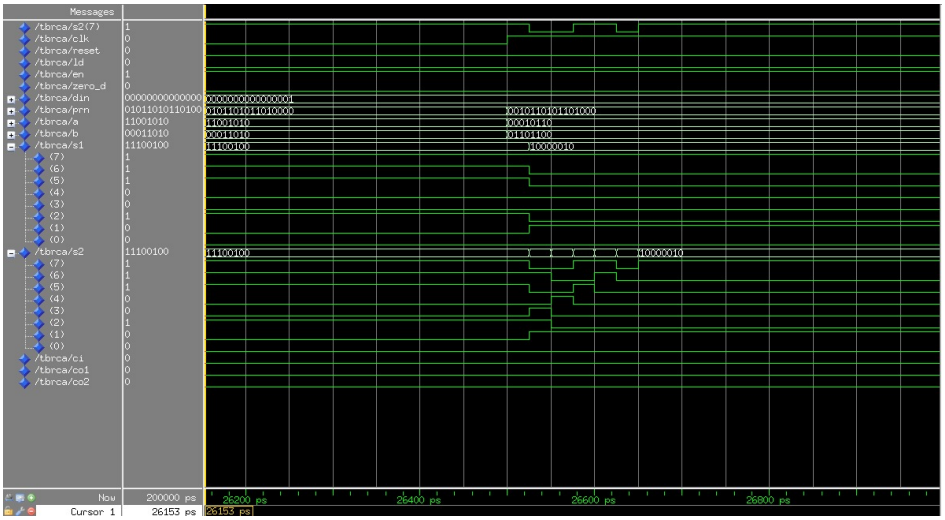

Figura 1.1: Confronto uscite di due *Ripple Carry Adder* con e senza ritardo sul *Carry Out*

Si sono quindi calcolate le *Switching Activity* totali dei due sommatori:

	$A(S)$
DRCAC=0	3.845
DRCAC=25ps	7.925

Tabella 1.10: *Switching Activity* per *Ripple Carry Adder* con e senza ritardo sul *Carry Out*

Dalla tabella 1.10 si nota che la *Switching Activity* raddoppia inserendo il ritardo.
L'*Overhead Computation* è in questo caso il numero di transizioni spurie che avvengono nel secondo *Ripple Carry Adder*. Vale quindi:

$$Overhead = Tc_{25ps} - Tc_{0ps} = 1585 - 769 = 816$$

Si è quindi cercato di capire quale fosse l'insieme di transizioni che massimizzasse la *Switching Activity* avendo ritardo sul *Carry Out*. In teoria per far sì che tutti i *Carry* si propaghino è necessario che commutino tutti da 0 a 1 o viceversa. Una combinazione per ottenere questo prevede che inizialmente tutti gli ingressi del *Ripple Carry Adder*siano a coppie di zeri e uni (ogni *Full Adder*deve avere in ingresso o due zeri o due uni). Quindi un ingresso del *Ripple Carry Adder*deve diventare il numero più grande rappresentabile con il parallelismo dato, mentre l'altro deve valere 1: in questo modo si genererà una propagazione continua del *Carry*.
Se si prende a esempio il secondo testbench fornito, questa combinazione viene inserita sui primi 6 bit di ingresso al sommatore. Modificando leggermente si è riusciti a massimizzare l'attività per una singola transizione nel sommatore: dal *power report* si ha che il bit più significativo ha 8 transizioni, a fronte di un singolo cambio di ingressi.

1.3 Sintesi e analisi di potenza di un *Ripple Carry Adder*

Dopo aver analizzato, elaborato e sintetizzato il *Ripple Carry Adder* con Synopsys, si è effettuata l'analisi di potenza del circuito.
Si è quindi cercato il percorso critico del circuito sintetizzato mediante il comando *report_timing*: si ha che il ritardo massimo è effettivamente $0.78ns$. Questo permette di impostare un clock per la simulazione del consumo del circuito avente periodo $T = 1ns$.
Una volta generato il clock con il comando *create_clock*, si è lanciato il comando *report_power*: da questo primo report si possono derivare già due conclusioni utili.
La prima riguarda il tipo di elementi circuitali che consumano potenza: nel caso del *Ripple Carry Adder* vi è il solo contributo combinatorio, ma in circuiti più complessi questa indicazione potrebbe tornare utile.
Secondariamente si può notare che la potenza dissipata è quasi tutta di tipo dinamico: la potenza statica (*Leakage Power*) è dell'ordine del μW, mentre la somma dei contributi dovuti alla carica e alla scarica delle capacità di ogni nodo (*Switching Power*) e alle correnti di cortocircuito delle varie celle (*Internal Power*) è dell'ordine della decina di μW. Inoltre dei due contributi dinamici quello dominante è quello dovuto alle correnti di cortocircuito: per poter ridurre questo effetto non si può operare a livello logico, ma è necessario passare al layout delle singole celle e ai parametri circuitali, in particolare la tensione di alimentazione. Nella prima riga della tabella 1.11 sono riportati i risultati numerici della simulazione.
Successivamente si è cercato di capire in che modo fosse distribuito il consumo all'interno del circuito: per fare questo si è usata l'opzione *-heir* con il comando *report_power*. I risultati ottenuti sono quelli mostrati in tabella 1.11.

	Switching Power	Internal Power	Leakage Power	Total Power
RCA	$9.741\mu W$	$16.743\mu W$	$953.483nW$	$27.437\mu W$
FAI 8	$0.848\mu W$	$2.154\mu W$	$119.291nW$	$3.121\mu W$
FAI 7	$1.293\mu W$	$2.150\mu W$	$118.962nW$	$3.562\mu W$
FAI 6	$1.332\mu W$	$2.191\mu W$	$119.340nW$	$3.642\mu W$
FAI 5	$1.328\mu W$	$2.171\mu W$	$119.110nW$	$3.618\mu W$
FAI 4	$1.272\mu W$	$2.101\mu W$	$119.294nW$	$3.498\mu W$
FAI 3	$1.263\mu W$	$2.090\mu W$	$118.979nW$	$3.471\mu W$
FAI 2	$1.229\mu W$	$2.002\mu W$	$119.508nW$	$3.351\mu W$
FAI 1	$1.171\mu W$	$1.884\mu W$	$118.999nW$	$3.175\mu W$

Tabella 1.11: *report_power* del *Ripple Carry Adder*e dei *Full Adder* che lo compongono

Osservando la tabella 1.11 si nota come i contributi di potenza siano gli stessi per tutti i *Full Adder* tranne l'ultimo della catena: la *Switching Power* di FAI 8 è minore di quella degli altri *Full Adder*. Questo è dovuto al fatto che non è stata definita una capacità di uscita per il *Carry Out* di tale componente: tutti gli altri *Full Adder* hanno come carico sul *Carry Out* la capacità d'ingresso dello stadio successivo, l'ultimo invece è un pin di uscita e viene caricato da Synopsis come tutte le uscite di somma. Se ne deduce che la capacità assegnata dal sintetizzatore alle usctite sia minore di quella di ingresso delle celle che compongono i *Full Adder*.
Per un'analisi più approfondita si sono osservati i consumi di ogni cella costituente il primo *Full Adder*: per fare questo si è cambiata istanza su Synopsys e si è usata l'opzione *-cell* di *report_power*. Al di fuori dei valori numerici ottenuti non vi erano informazioni aggiuntive rilevanti e si è quindi utilizzata l'opzione *-net* per analizzere vari parametri del singolo *Full Adder* e poi del *Ripple Carry Adder* completo.
L'opzione *-net* permette di conoscere i parametri dei nodi interni di un circuito sintetizzato da Synopsys: nell caso del primo *Full Adder* della catena il report è riportato in tabella 1.12

	Total Net Load	*Static Probability*	*Toggle Rate*	*Switching Power*	*Internal Power*	*Leakage Power*	*Total Dynamic Power*
n1	$4.694fF$	0.493	0.1932	$0.5488\mu W$	$0.5889\mu W$	$36.1637nW$	$1.138\mu W$
Co	$4.554fF$	0.512	0.1439	$0.3964\mu W$	$0.1377\mu W$	$14.2499nW$	$0.534\mu W$
n2	$2.010fF$	0.488	0.1439	$0.1749\mu W$	$0.3374\mu W$	$32.5747nW$	$0.512\mu W$
S	$0.310fF$	0.507	0.2735	$0.0512\mu W$	$0.8205\mu W$	$36.0111nW$	$0.872\mu W$
FA 1				$1.171\mu W$	$1.884\mu W$	$118.999nW$	$3.056\mu W$

Tabella 1.12: Power report FAI 1 nets

La prima colonna della tabella 1.12 riporta le capacità associate a ciascun nodo. Questo dimostra l'ipotesi fatta in precedenza per l'ultimo *Full Adder* della catena: infatti la capacità associata al nodo di uscita S è di un ordine di grandezza più piccola di quelle dei nodi interni. Per questo stesso motivo la potenza di *switching* associata al nodo S è molto vicina a 0 e quasi tutto il suo apporto in potenza è dovuto alla corrente di cortocircuito.
La seconda e la terza colonna contengono invece la probabilità che un nodo sia all'1 logico (*Static Probability*) e la sua *Switching Activity* (*Toggle Rate*).

1.4 Un semplice *MUX*: generazione e propagazione di *glitch*

Si è inizialmente verificato che il file *.vhd* fornito effettivamente implementasse il circuito in figura 1.2.

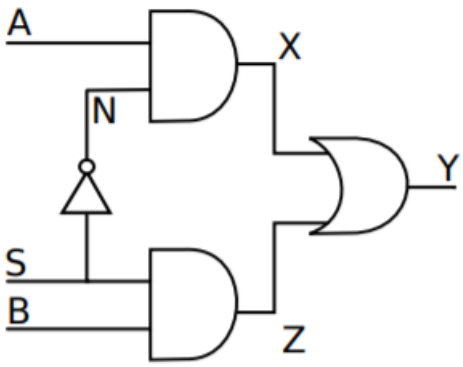

Figura 1.2: Schema di base di un *multiplexer*

Successivamente, si è simulato su Modelsim il testbench fornito, ottenendo le onde mostrate in figura 1.3.

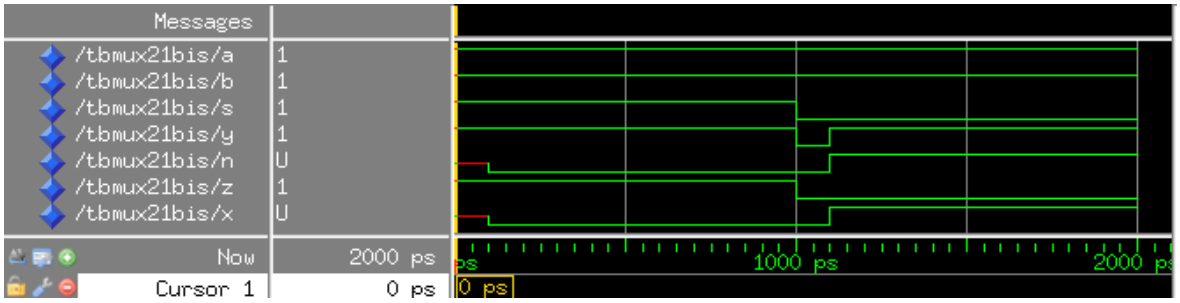

Figura 1.3: Simulazione del *MUX*

La simulazione fa variare il segnale di selezione del multiplexer: sebbene entrambi gli ingressi siano forzati a 1, per 100ps l'uscita scende a 0. Questo *glitch* è causato dal ritardo inserito nell'inverter che nega il segnale di selezione. Ritardando questo segnale si creano due percorsi combinatori con ritardo diverso: quando S scende a 0 esso si propaga istantaneamente nella porta AND connessa a B, ma, a causa del ritardo dell'inverter, anche il nodo N si trova ancora allo 0 logico. La porta OR ha quindi due zeri in ingresso e la sua uscita commuta. Quando S ha finito di propagarsi nell'inverter, tuttavia, l'uscita torna a 1.
In questo circuito l'unico modo per generare un *glitch* è far variare S, essendo esso l'unico segnale che passa per una porta con un ritardo diverso da quello infinitesimo usato dal simulatore.
Dal punto di vista del consumo energetico questo *glitch* causa la scarica e la carica di un'ipotetica capacità pilotata dall'uscita del *MUX*. Se si conoscessero la tensione di alimentazione del circuito e la dimensione della capacità di carico si potrebbe avere una stima dell'energia spesa con la seguente formula:

$$E_{tot} = E_{1\to0} + E_{0\to1} = CV_{dd}^2 + CV_{dd}^2 = 2CV_{dd}^2$$

1.5 Calcolo della probabilità e della *Switching Activity*: contatore sincrono

Si è inizialmente analizzato il circuito riportato in figura 1.4.

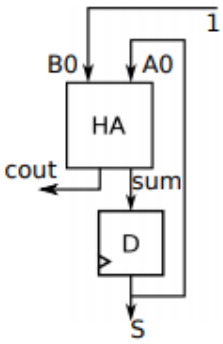

Figura 1.4: Schema dell'unità base di un contatore sincrono

Esso si comporta come un *T flip-flop* con enable: se *B0* vale 1, il circuito è abilitato e ad ogni colpo di clock il segnale di uscita *S* cambia il suo valore. Se *B0* è a 0 il circuito entra in uno stato di memoria in cui sia *cout* che *S* mantengono lo stesso valore finchè *B0* non torna nuovamente a 1. Da questo si deduce che *B0* ha la funzione di segnale di enable e che la *Switching Activity* di *S* è metà rispetto a quella del clock. Dal diagramma temporale in figura 1.5 è possibile comprendere meglio quanto appena spiegato.

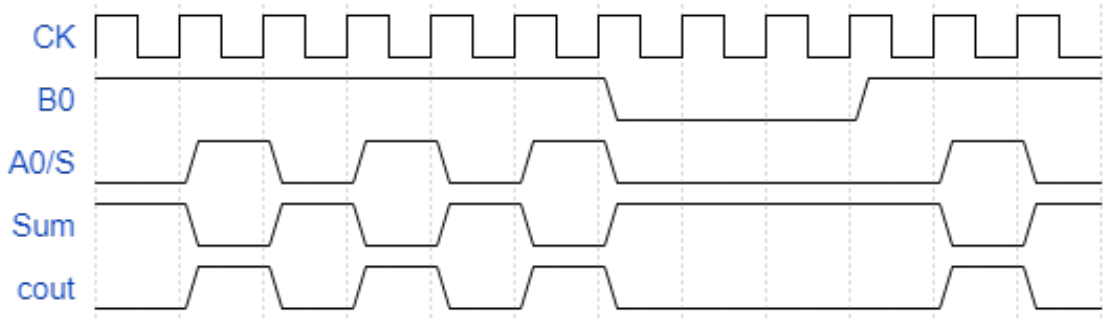

Figura 1.5: Diagramma temporale dell'unità del contatore

Si è quindi cercato di comprendere il funzionamento del circuito riportato in figura 1.6.
Come prima cosa si è tracciato il diagramma temporale (figura 1.7): ogni uscita commuta con una frequenza dimezzata rispetto alla precedente, comportamento tipico dei contatori sincroni. Si può quindi affermare che il circuito in figura 1.6 è un contatore sincrono.

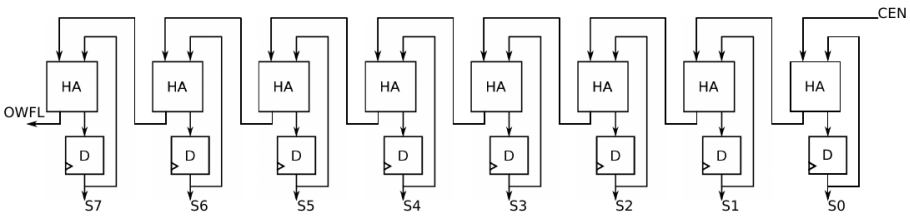

Figura 1.6: Schema contatore sincrono a 8 bit

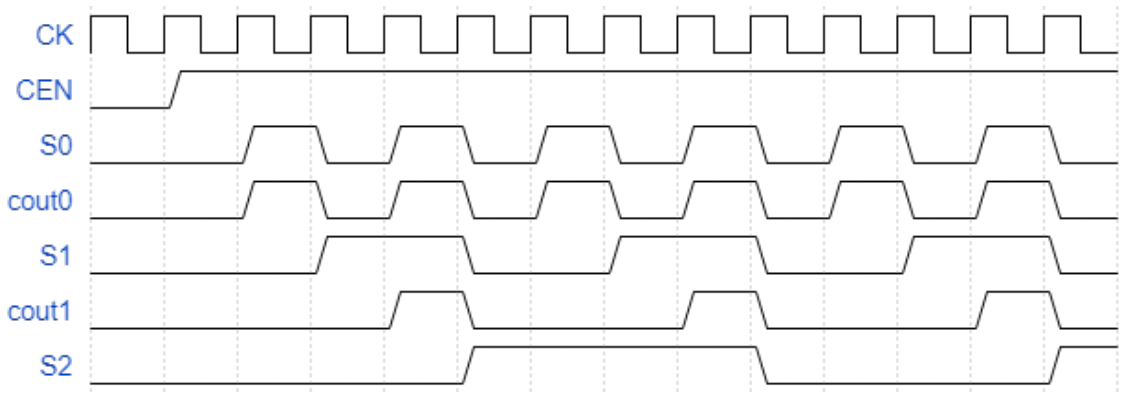

Figura 1.7: Diagramma temporale del contatore sincrono

L'ingresso *CEN* (figura 1.6) viene usato come enable dell'intero dispositivo: quando viene posto a 0, infatti, il conteggio si ferma e il contatore mantiene memorizzato il valore raggiunto in quel momento.
Il segnale *OWFL* ha, invece, la funzione di *Terminal Count*del contatore: esso sale a 1 solo quando tutti gli *Half Hadder* hanno in uscita un 1 e il segnale *CEN* è anch'esso abilitato. Da questo si deduce che la sua *Switching Activity* teorica dovrebbe essere:

$$S_{OWFL} = \frac{1}{256}$$

Si dovrebbe avere solo una transizione del *Terminal Count* per ogni conta completa, che avviene, su 8 bit, ogni 256 transizioni del primo bit del contatore.
Si è poi calcolato il numero di transizioni di ogni uscita del contatore per un ciclo completo di conta: ogni bit deve commutare la metà rispetto al al precedente. Da questa supposizione si è ricavata la seguente formula in cui N è il numero di bit e l'*LSB* è quello per $n = 0$:

$$T = 2^{N-n} \tag{1.1}$$

Si è poi passati alla simulazione su Modelsim per verificare la bontà delle supposizioni fatte.
La simulazione, con periodo $T = 2ns$, da ragione all'analisi appena fatta: in tabella 1.13 sono riportati il numero di *toggle* calcolati con la formula 1.1 e quelli forniti dal comando *power report*, che sono in perfetto accordo.

	Tc simulato	Tc ideale
CK	256	256
S0	256	256
S1	128	128
S2	64	64
S3	32	32
S4	16	16
S5	8	8
S6	4	4
S7	2	2
OWFL	16	2

Tabella 1.13: *Toggle count* simulato e ideale in uscita al contatore sincrono

Per quanto riguarda *OWFL*, si nota però come la previsione fatta in precedenza sia totalmente errata: il numero di commutazioni è infatti 16, che si traduce in un'attività molto maggiore per un segnale che in un circuito reale potrebbe essere particolarmente importante. La causa di queste transizioni spurie sono dei *glitch* causati dalla propagazione dei segnali non istantanea lungo la catena degli *Half Adder*. Per meglio spiegare questo comportamento si è riportato in tabella 1.14 il numero di transizioni dei nodi interni: *Carry Out*, ingressi e uscite dei *D flip-flop*.

	Carry Out	S(HA)	DFF
bit0	256	257	256
bit1	256	384	128
bit2	192	320	64
bit3	128	224	32
bit4	80	144	16
bit5	48	88	8
bit6	28	52	4
bit7	16	30	2

Tabella 1.14: Transizioni di Half Adder e DFF

Calcolando il rapporto tra il numero di transizioni dei nodi e quelle ideali, cioè quelle in uscita al *flip-flop*, si sono ricavate le seguenti relazioni:

$$T_c(C_o) = T_c \cdot (n+1) \tag{1.2}$$

$$T_c(S) = T_c \cdot (2n+1) \tag{1.3}$$

Queste relazioni spiegano in modo empirico quanti *glitch* vengono generati nel circuito.
Per meglio comprendere questi comportamenti si spiegherà nel dettaglio cosa succede nel secondo *Half Adder* del contatore: se si ipotizza di avere in ingresso il segnale di *Carry Out* dello stadio precedente a 1 e il valore in uscita dal *flip-flop* uguale 0, quando arriva il colpo di clock, il *flip-flop* cambierà stato, portando alla sua uscita un 1. Questo impiegherà $0.2ns$ (ritardo impostato nel file *.vhd*) a propagarsi di nuovo all'ingresso del *flip-flop*. Passati questo tempo, tuttavia, si sarà propagato il nuovo *Carry Out* del primo *Half Adder*, cosa che, dopo un altro ritardo, farà commutare nuovamente l'ingresso del *flip-flop*. Se questi ritardi non ci fossero stati l'ingresso del *flip-flop* non avrebbe dovuto commutare: vi sono quindi due transizioni spurie, visto che alla fine il nodo ritorna al valore che aveva quando è arrivato il clock. Inoltre questo succede un numero di volte pari alla posizione nella catena dell'*flip-flop*: da qui la formula 1.3.
Per quanto riguarda il *Carry Out*, e quindi anche per *OWFL*, il ragionamento è analogo, ma, date le combinazioni possibili, le transizioni spurie avvengono ogni due transizioni del *Carry Out*, giustificando la mancanza del 2 nella formula 1.2. Si è quindi pensato a cosa succederebbe al circuito riducendo il periodo del clock a $0.8ns$: poiché ogni *Half Adder* ha un ritardo tra ingressi e uscite pari a $0.2ns$, il *Carry* non riesce a propagarsi lungo tutta la catena. In particolare il funzionamento sarà garantito fino a quando verranno stimolati solo i primi 4 *Half Adder*, dopodichè i *flip-flop* in uscita campioneranno valori sbagliati e l'uscita del contatore non riporterà il valore corretto.

Simulando si ottiene esattamente questo risultato: infatti il primo errore si ha quando il contatore arriva a 15, cioè quando inizia a commutare il quarto *Half Adder*. Il contatore porta in uscita un 31, che non è il valore corretto, per il motivo descritto sopra. La simulazione è in figura 1.8.

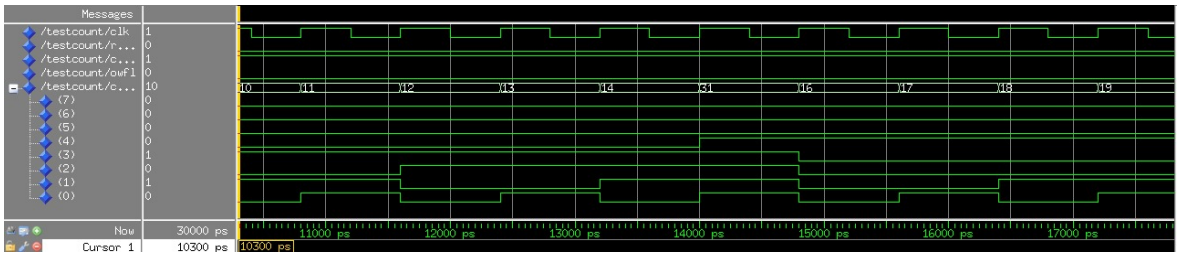

Figura 1.8: Contatore con clock a 0.8 ns e ritardo interno di 0.2 ns

CAPITOLO 2

Assegnazione degli stati in una FSM e sintesi VHDL

2.1 Assegnazione degli stati in una FSM

La progettazione della FSM è avvenuta prendendo come riferimento la codifica di *Gray*: minimizzando il numero di bit tra una transizione e l'altra si riduce la *Switching Activity* totale, che è il risultato che si vuole ottenere.
Per quanto riguarda gli stati si è scelta in modo arbitrario una qualsiasi sequenza di 5 combinazioni di 3 bit: si è prestato attenzione però al fatto che, quando la FSM opera in modo ricorsivo, il salto dall'ultimo al primo stato non comportasse un aumento di attività per nessun bit di stato.
La codifica scelta è stata:

Stato	a+b	sum+c	sum+d	sum+e	sum+f
Codifica	001	011	111	110	100

Tabella 2.1: Codifica degli stati

Contando il numero di transizioni in tabella 2.1, si vede che la *Switching Activity* è: $A(b_i) = \frac{2}{5} = 0.4$.
In modo analogo si sono trovate le sequenze delle uscite che pilotano i *multiplexer*:

Stato	S(0)S(1)	S(2)S(3)
a+b	01	10
sum+c	11	00
sum+d	11	01
sum+e	10	11
sum+f	00	11

Tabella 2.2: Codifica di S

Come si può notare dalla tabella 2.2, la codifica non è quella di *Gray*: si è cercato infatti di minimizzare il numero di transizioni bit per bit. In realtà *Gray* è stato usato su ciascuno dei selettori

del *multiplexer* poi "sfasando" le due codifiche per ottenere il risultato sperato: ogni bit cambia due volte per ciclo, se il sistema lavora in modo continuo.
Si ricava quindi che la *Switching Activity* di ogni bit di selezione è $A(S_i) = \frac{2}{5} = 0.4$.
Si sono quindi prodotti i due file VHDL richiesti. I file possono essere visionati nella sezione 2.3.
Nella tabella successiva si mostra come sono stati collegati gli ingressi ai *multiplexer* affinché il circuito eseguisse le operazioni richieste con la sequenza di selettori ricavata sopra:

multiplexer 1	ingresso
F	0
A	1
E	2
SUM	3
multiplexer 2	ingresso
C	0
D	1
B	2
SUM	3

Tabella 2.3: Connessioni ingressi dei multiplexer

Successivamente si è scritto anche un semplice testbench per valutare il corretto funzionamento della macchina e per valutare la *Switching Activity* dei selettori. Il file *.vhd* si trova sempre nella sezione 2.3.
Il risultato della simulazione è il seguente:

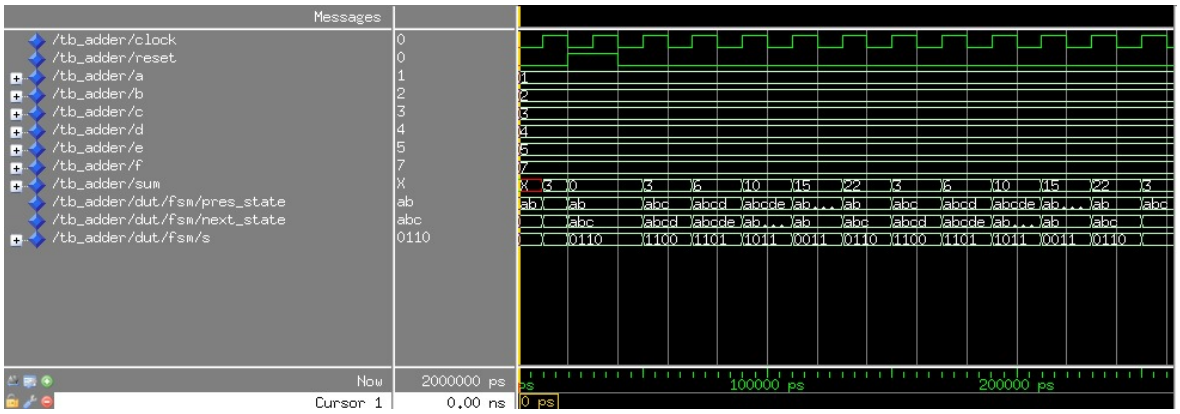

Figura 2.1: Simulazione Modelsim FSM

La macchina esegue le operazioni correttamente.

Con il comando *power report* di Modelsim si è quindi valutata la *Switching Activity* di S:

	T_c	A
clk	100	-
S(0)	39	0.39
S(1)	42	0.42
S(2)	39	0.39
S(3)	41	0.41

Tabella 2.4: *Switching Activity*dei segnali si selezione

I risultati riportati in tabella 2.4 sono molti vicini a quanto ricavato analiticamente: vi è un piccolo errore dovuto alla presenza del reset all'inizio della simulazione, che è però trascurabile.

2.2 Sintesi del VHDL

Dopo una prima sintesi via interfaccia grafica seguendo i passaggi proposti, si è passati all'uso di uno script che automatizzasse tutti i passaggi. Si sono prodotti due script, uno con le impostazioni di base e uno con clock a frequenza massima consentita. Entrambi sono riportati nella sezione 2.4.
Tutta la parte dei comandi relativi all'analisi di potenza è stata ampiamente discussa nel precedente laboratorio e non verrà per tanto ritrattata. Il comando *set_max_dynamic_power* non è stato utilizzato: il manuale del programma ne sconsiglia l'uso in quanto antiquato e probabilmente già rimosso in versioni più aggiornato del software. Inoltre, provando ad usarlo, fornisce risultati totalmente incomprensibili, aumentando la potenza, invece che riducendola.
Per ogni comando vengono riportati e confrontati i risultati ottenuti con la sintesi di base e quella con clock massimo: i passaggi sono stati svolti come riportato nella traccia del laboratorio, ma si è ritenuto più efficacie presentare ciò che si è ricavato in questo modo.

2.2.1 *report_area*

Questo comando permettere di avere un'idea delle dimensioni del circuito sintetizzato dal CAD.
Vengono visualizzate informazioni sul numero di elementi del circuito, di porte e di nodi. Viene riportata anche qual è la dimensione effettiva in termini di area del circuito usando le celle della libreria fornita al sintetizzatore: il risultato ottenuto è di $295\mu m^2$, se l'unità di misura fornita per la singola cella della libreria è il micron.
Aumentando la frequenza di clock l'area aumenta a $298\mu m^2$: il sintetizzatore ha dovuto aumentare leggermente la dimensione del dispositivo per permettergli di mantenere la frequenza richiesta. In particolare l'aumento è avvenuto su celle di logica combinatoria.

2.2.2 *report_fsm*

Questo comando non da informazioni utili ai fini della potenza, ma permette di accertarsi che il sintetizzatore abbia capito il comando *attribute* di VHDL e abbia agito di conseguenza. L'assegnazione degli stati è stata fatta in modo corretto, come si può vedere dal report:

```
****************************************
Report : FSM
Design : ADDER/FSM (FSM_ADDER)
Version: F-2011.09-SP3
Date   : Wed Apr 10 13:44:47 2019
****************************************

Clock             : Unspecified
Asynchronous Reset: Unspecified

Encoding Bit Length: 3
Encoding style     : auto

State Vector: { PRES_STATE_reg[2] PRES_STATE_reg[1] PRES_STATE_reg[0] }

State Encodings and Order:

ab          : 001
abc         : 011
abcd        : 111
abcde       : 110
abcdef      : 100

1
```

2.2.3 *report_timing*

Questo comando calcola il percorso critico all'interno del circuito, ne riporta il tempo di percorrenza e lo *slack*, sia che venga o meno rispettato.
Avendo impostato inizialmente il clock a $100MHz$, si legge nel report che il percorso critico è quello che parte dall'uscita del registro contenente il bit 2 dello stato e che arriva all'ingresso dell'ultimo bit del registro della somma. Non stupisce il fatto che il punto di arrivo del percorso critico sia l'ultimo bit del sommatore, essendo un *Ripple Carry Adder*, ma guadagnamo l'informazione relativa al punto di partenza, che potrebbe tornare utile in caso di ottimizzazioni estreme.
Il ritardo trovato è di $1.91ns$, che è stato usato come clock nel secondo design.
Nella seconda sintesi il dato importante riguarda lo *slack*: si è infatti riusciti a ridurlo a 0. Questo potrebbe essere tuttavia un problema in caso si avessero ritardi non voluti e si andasse a infrangere i tempi di setup e hold.

2.2.4 *report_timing -nworst*

Questo comando mostra il ritardo dei 10 percorsi più critici. Si nota che in realtà è sempre lo stesso percorso: Il sintetizzatore fa variare i ritardi delle porte considerando corner diversi di funzionamento.

2.2.5 *Timing→Endpoint*

Timing→Endpoint genera un istogramma come quello riportato in figura 2.2.

Questo ci permette di vedere subito qual è lo *slack* peggiore, e quindi il ritardo massimo nel circuito. Inoltre ci da un'indicazione sulla distribuzione dei percorsi critici: il simulatore calcola il percorso a ritardo massimo per ogni elemento di memoria. Poi incrementa l'altezza della colonna corrispondente

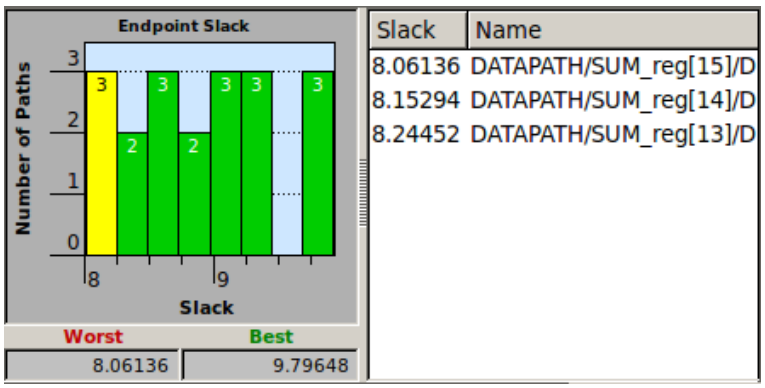

Figura 2.2: Istogramma *slack*-percorsi

allo *slack* appena ottenuto. Nel dispositivo simulato i ritardi sono distribuiti in modo abbastanza uniforme.

2.3 VHDL

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;

ENTITY FSM_ADDER IS
  PORT (CLK, RST : IN STD_LOGIC;
      S : OUT STD_LOGIC_VECTOR(3 DOWNTO 0));
END ENTITY FSM_ADDER ;

ARCHITECTURE STATE_MACHINE OF FSM_ADDER  IS

TYPE states IS (ab, abc, abcd, abcde, abcdef);

ATTRIBUTE enum_encoding : string;
ATTRIBUTE enum_encoding OF states : TYPE IS "001 011 111 110 100";

SIGNAL PRES_STATE : states;
SIGNAL NEXT_STATE : states;

BEGIN

  proc : PROCESS(CLK, RST)
    BEGIN
    IF (RST = '1') THEN
      PRES_STATE <= ab;
    ELSIF (CLK'EVENT AND CLK='1') THEN
      PRES_STATE <= NEXT_STATE;
    END IF;
  END PROCESS;

  next_state_decision : PROCESS(PRES_STATE)
    BEGIN
    CASE PRES_STATE IS
      WHEN ab =>
        NEXT_STATE <= abc;
      WHEN abc =>
        NEXT_STATE <= abcd;
      WHEN abcd =>
        NEXT_STATE <= abcde;
      WHEN abcde =>
        NEXT_STATE <= abcdef;
      WHEN abcdef =>
        NEXT_STATE <= ab;
      WHEN OTHERS =>
        NEXT_STATE <= ab;
    END CASE;
  END PROCESS;

  outs : PROCESS(PRES_STATE)
    BEGIN
    CASE PRES_STATE IS
      WHEN ab =>
        S <= "0110";
      WHEN abc =>
        S <= "1100";
      WHEN abcd =>
        S <= "1101";
      WHEN abcde =>
        S <= "1011";
      WHEN abcdef =>
        S <= "0011";
      WHEN OTHERS =>
        S <= "0110";
    END CASE;
  END PROCESS;

END ARCHITECTURE STATE_MACHINE;
```

Listing 1: FSM

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;

ENTITY ADDER IS
  PORT (A, B, C, D, E, F : IN STD_LOGIC_VECTOR(15 DOWNTO 0);
       clock, reset : IN STD_LOGIC;
       SUM : INOUT STD_LOGIC_VECTOR(15 DOWNTO 0));
END ENTITY ADDER;

ARCHITECTURE BEHAVIOURAL OF ADDER IS

COMPONENT FSM_ADDER IS
  PORT (CLK, RST : IN STD_LOGIC;
       S : OUT STD_LOGIC_VECTOR(3 DOWNTO 0));
END COMPONENT FSM_ADDER;

COMPONENT datapath_adder is
  port(   MUX00:  in std_logic_vector(15 downto 0);
    MUX01:  in std_logic_vector(15 downto 0);
    MUX02:  in std_logic_vector(15 downto 0);
    MUX03:  in std_logic_vector(15 downto 0);
    MUX10:  in std_logic_vector(15 downto 0);
    MUX11:  in std_logic_vector(15 downto 0);
    MUX12:  in std_logic_vector(15 downto 0);
    MUX13:  in std_logic_vector(15 downto 0);
    clock:   in std_logic;
    reset:  in std_logic;
    SEL00:  in std_logic;
    SEL01:  in std_logic;
    SEL10:  in std_logic;
    SEL11:  in std_logic;
    SUM:  out std_logic_vector(15 downto 0)
  );
end  COMPONENT datapath_adder;

SIGNAL S : STD_LOGIC_VECTOR (3 DOWNTO 0);

BEGIN

  DATAPATH:datapath_adder PORT MAP(
    MUX00 => F,
    MUX01 => A,
    MUX02 => E,
    MUX03 => SUM,
    MUX10 => C,
    MUX11 => D,
    MUX12 => B,
    MUX13 => SUM,
    clock => clock,
    reset => reset,
    SEL00 => S(2),
    SEL01 => S(3),
    SEL10 => S(0),
    SEL11 => S(1),
    SUM => SUM);

  FSM:FSM_ADDER PORT MAP(
    CLK => clock,
    RST => reset,
    S => S);

END ARCHITECTURE BEHAVIOURAL;
```

Listing 2: Adder, datapath e FSM

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;

ENTITY tb_ADDER IS
END ENTITY tb_ADDER;

ARCHITECTURE TB OF TB_ADDER IS

COMPONENT ADDER IS
  PORT (A, B, C, D, E, F : IN STD_LOGIC_VECTOR(15 DOWNTO 0);
       clock, reset : IN STD_LOGIC;
       SUM : INOUT STD_LOGIC_VECTOR(15 DOWNTO 0));
END COMPONENT ADDER;

SIGNAL clock, reset : std_logic:='0';
signal a,b,c,d,e,f, sum : std_logic_vector (15 downto 0);
BEGIN

clock <= not clock after 10 ns;
reset <= '1' after 20 ns, '0' after 40 ns;
a <= "0000000000000001";
b <= "0000000000000010";
c <= "0000000000000011";
d <= "0000000000000100";
e <= "0000000000000101";
f <= "0000000000000111";
dut:adder port map(
  a,b,c,d,e,f,clock,reset,sum);
end architecture;
```

Listing 3: Testbench

2.4 Script per Design Vision

```
analyze -library WORK -format vhdl {/home/lp19.16/ese2/synth/fsm_adder.vhd
↪  /home/lp19.16/ese2/synth/dpadder.vhd /home/lp19.16/ese2/synth/adder.vhd}
elaborate ADDER -architecture BEHAVIOURAL -library DEFAULT
create_clock -name "CLK" -period 10 {clock}
compile -exact_map
write -hierarchy -format ddc -output /home/lp19.16/ese2/synth/fsma-base.ddc
write -hierarchy -format vhdl -output /home/lp19.16/ese2/synth/fsma-base.vhdl
report_area > log-base/fsma-area-base.txt
report_timing > log-base/fsma-time-base.txt
report_timing -nworst 10 > log-base/fsma-10ntime-base.txt
report_power > log-base/fsma-pow-base.txt
report_power -hier > log-base/fsma-powcell-base.txt
report_power -net > log-base/fsma-tc-base.txt
current_instance FSM
report_fsm > log-base/fsma-fsm-base.txt
report_power -net -cell  > log-base/fsma-powtcfsm-base.txt
```

Listing 4: base

```
analyze -library WORK -format vhdl {/home/lp19.16/ese2/synth/fsm_adder.vhd
↪  /home/lp19.16/ese2/synth/dpadder.vhd /home/lp19.16/ese2/synth/adder.vhd}
elaborate ADDER -architecture BEHAVIOURAL -library DEFAULT
create_clock -name "CLK" -period 1.91 {clock}
compile -exact_map
write -hierarchy -format ddc -output /home/lp19.16/ese2/synth/fsma-constr.ddc
write -hierarchy -format vhdl -output /home/lp19.16/ese2/synth/fsma-constr.vhdl
report_area > log-constr/fsma-area-constr.txt
report_fsm > log-constr/fsma-fsm-constr.txt
report_timing > log-constr/fsma-time-constr.txt
report_power > log-constr/fsma-pow-constr.txt
report_power -hier > log-constr/fsma-powcell-constr.txt
report_power -net > log-constr/fsma-tc-constr.txt
current_instance FSM
report_power -net -cell  > log-constr/fsma-powtcfsm-constr.txt
```

Listing 5: massima frequenza

CAPITOLO 3

Clock Gating, *pipelining* e parallelizzazione

3.1 Un primo approccio al *Clock Gating*

Si considerino due registri, L1 e L2, collegati in serie in modo che l'uscita di L1 sia anche l'ingresso di L2. Se si collegasse al registro L1 il clock e al registro L2 un segnale generato dall'AND tra il clock e un ENABLE, otterremmo un clock gating e ci aspetteremmo, in presenza di ENABLE attivo, il dato in ingresso ad L1 ritardato di due colpi di clock all'uscita (D3) di L2. Dalla simulazione di Modelsim visibile in figura 3.1 si ottiene un risultato diverso da quello previsto per via del fatto che, in assenza di un ritardo esplicitato, Modelsim simula il circuito come se la porta AND avesse un ritardo infinitesimo. Questo modo di operazione del programma di simulazione porta ad un campionamento simultaneo tra L1 e L2 ottenendo in D3 lo stesso dato presente contemporaneamente in D2.

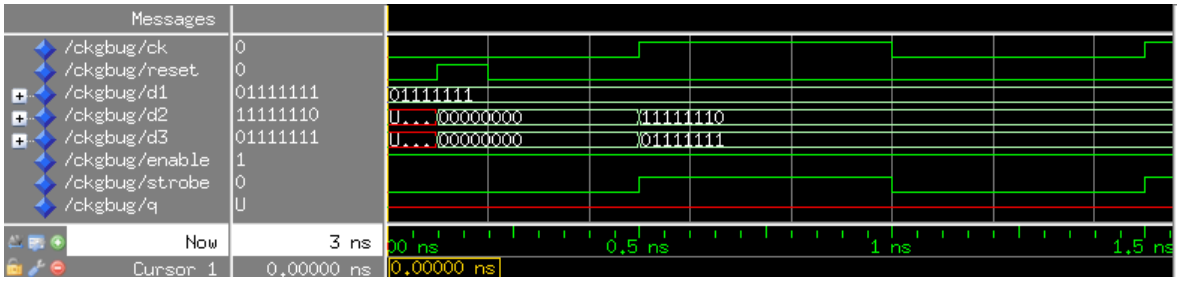

Figura 3.1: Simulazione senza ritardi

Se adesso aggiungiamo un ritardo sull'uscita di L1 di, per esempio, 0.1ps, otteniamo la waveform presente in figura 3.2 che rappresenta ció che ci saremmo aspettati giá precedentemente.

Aggiungendo, infine, un ritardo di 0.2ps anche sulla porta AND, riotteniamo quello che abbiamo giá visto in figura 3.1.

3.2 *Clock Gating* in un circuito complesso

Si è quindi studiato il comportamento del circuito visibile in figura 3.3 per poter poi applicare la tecnica del *Clock Gating*.
Il dispositivo è un comparatore che fornisce in uscita il numero più grande tra quelli salvati nei registri *A* e *B*. Al segnale di reset si suppone che tutti i registri vengano inizializzati a 0. Abilitando successivamente i due segnali di *INCA* e *INCB*, è possibile incrementare il valore contenuto nei registri: quando

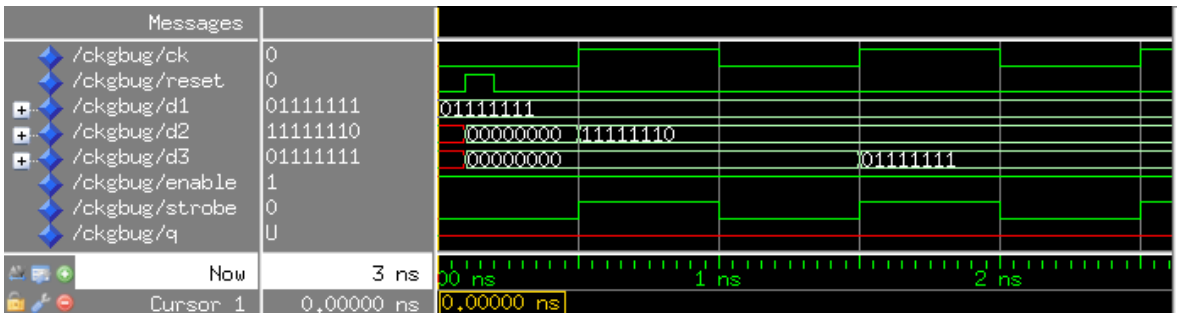

Figura 3.2: Simulazione con ritardo su L1

uno dei due numeri diventa maggiore dell'altro, il comparatore modifica il selettore del *multiplexer* fornendo al registro di uscita il nuovo valore da campionare e portare in uscita.

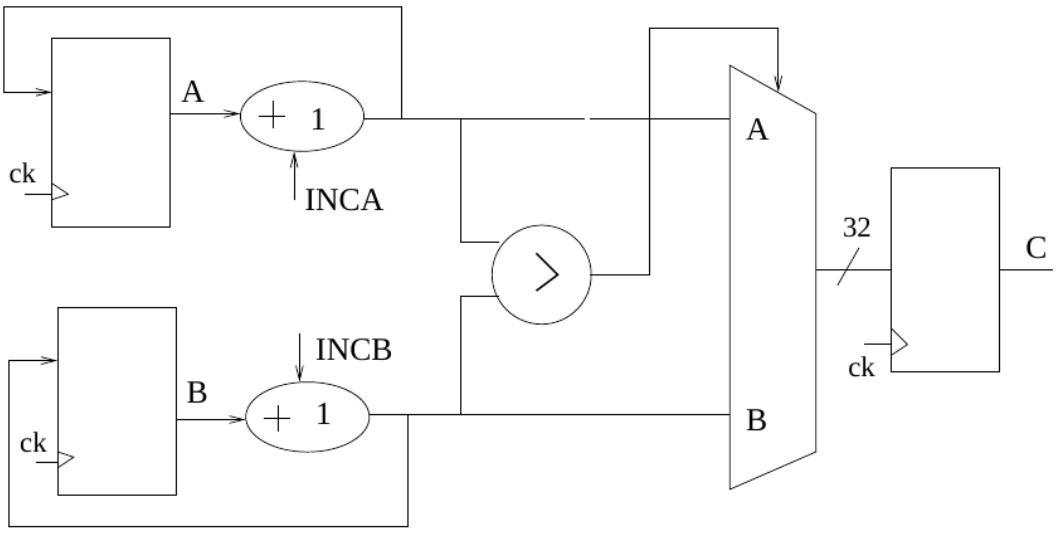

Figura 3.3: Circuito complesso su cui applicare il *Clock Gating*

Per verificare che il circuito funzionasse come ipotizzato, lo si è simulato su Modelsim. In figura 3.4 è mostrato il risultato della simulazione, il cui comportamento è congruo con le ipotesi fatte.

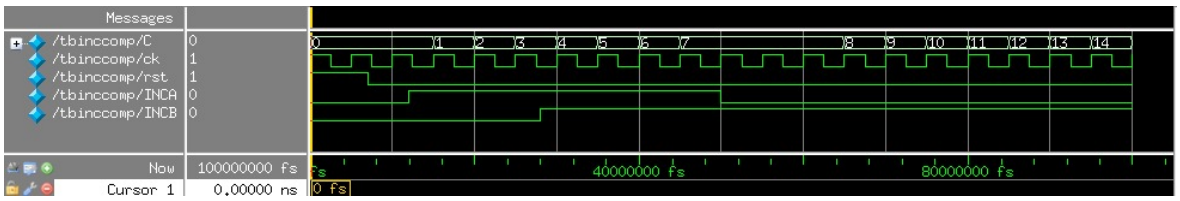

Figura 3.4: Simulazione Modelsim del circuito complesso

Successivamente si è applicata la tecnica del *Clock Gating*. Poiché sono presenti tre elementi di memoria, si sono inseriti tre *latch* e tre porte AND aggiuntive. Per il registri contenti A e B è stato usato come segnale di abilitazione del clock il corrispondente segnale di incremento.
Per il registro di uscita, poiché deve essere abilitato quando almeno una delle due parti del circuito

cambia il suo stato, si è usato come *enable* l'OR logico dei due segnali di incremento.
Il circuito così ottenuto è mostrato in figura 3.5.

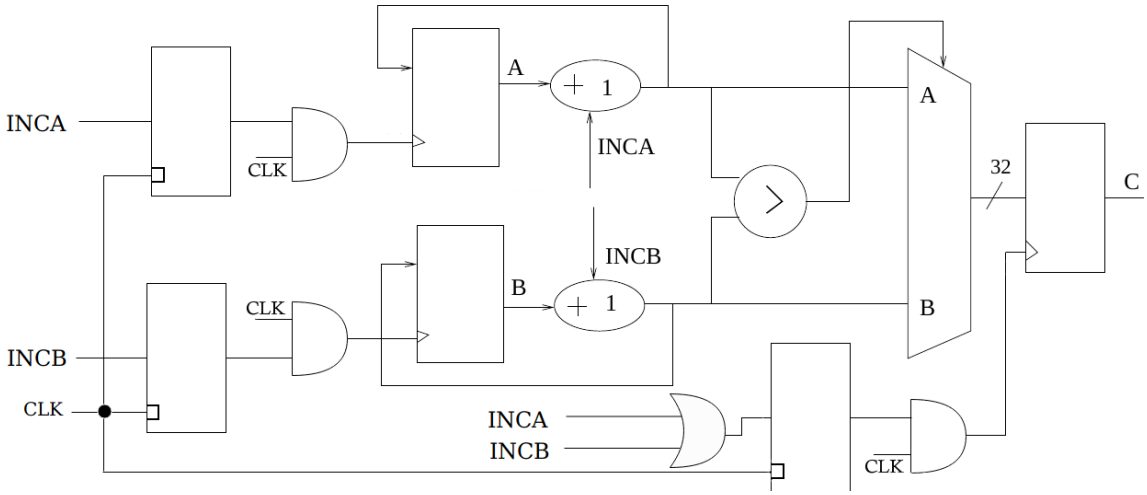

Figura 3.5: Circuito complesso con *Clock Gating*

Si è passati quindi alla sintesi con Synopsys e all'analisi di potenza.
Come richiesto si è inizialmente sintetizzato il circuito senza comunicare al programma di applicare la tecnica del *Clock Gating*.
I risultati ottenuti sono riportati in tabella 3.1.

Internal Power	*Switching Power*	*Leakage Power*	*Total Power*
$38.7514\mu W$	$3.578\mu W$	$3.796\mu W$	$46.1263\mu W$

Tabella 3.1: *report_power* del circuito senza *Clock Gating*

Questi risultati sono tuttavia distanti dal reale consumo del circuito: utilizzando l'opzione *-net* si è osservato che la probabilità di tutti i segnali è assegnata arbitrariamente dal sintetizzatore a 0.5. Si è quindi modificata la probabilità e la *Switching Activity* dei segnali di clock e reset, aumentando l'attività del clock e riducendo quella del reset.
L'analisi di potenza risultante è riporta in tabella 3.2.

Internal Power	*Switching Power*	*Leakage Power*	*Total Power*
$27.636\mu W$	$4.7219\mu W$	$4.180\mu W$	$36.538\mu W$

Tabella 3.2: *report_power* del circuito senza *Clock Gating* usando il comando *set_switching_activity* su RST e CLK

Come previsto il consumo si è ridotto. Sebbene sia aumentata la potenza dovuta alla carica e alla scarica delle capacità, a causa del forte aumento di attività del segnale di clock, l'aver azzerato la *Switching Activity* del segnale di reset riduce il numero di commutazioni del circuito. Precedentemente il reset aveva il 50% di possibilità di portare il circuito allo stato iniziale, facendo commutare tutto il circuito. Questo spiega la riduzione della potenza interna.
Inoltre il sintetizzatore deve aver modificato il *layout* del circuito, poichè la potenza statica è aumentata, seppur di poco.
Infine si è modificata anche la probabilità dei segnali di incremento INCA e INCB, supponendo che

commutino meno spesso della metà delle volte e che sia molto più probabile che il loro valore sia 0 piuttosto che 1. Si ottiene quanto riportato in tabella 3.3.

Internal Power	*Switching Power*	*Leakage Power*	*Total Power*	*Cells*
$21.5273\mu W$	$1.1314\mu W$	$4.0921\mu W$	$26.7508\mu W$	74

Tabella 3.3: *report_power* del circuito senza *Clock Gating* usando il comando *set_switching_activity* su RST, CLK, INCA e INCB

La potenza dissipata si è ridotta ulteriormente: questa volta la riduzione sulla potenza dovuta alle commutazioni che impatta maggiormente, risultando ridotta globalmente di circa $10\mu W$, 6 risparmiati in termini di corrente di corto circuito e 4 in termini di capacità caricate e scaricate.
Si è quindi comunicato al sintetizzatore di applicare il *Clock Gating* usando l'opzione *-gate_clock* lanciando il comando *compile*. Il primo risultato, senza alcuna modifica alle probabilità e attività dei segnali, è riportato in tabella 3.4.

Internal Power	*Switching Power*	*Leakage Power*	*Total Power*
$32.9555\mu W$	$6.4769\mu W$	$3.9340\mu W$	$43.3664\mu W$

Tabella 3.4: *report_power* del circuito con *Clock Gating*

La potenza totale è leggermente diminuita rispetto al caso senza *Clock Gating*: la potenza di *switching* e quella di *leakage* sono aumentate a causa dell'inserimento dei blocchi di *Clock Gating*, ma si è ridotta la corrente di corto circuito totale. Poiché i segnali sono ancora tutti equiprobabili non si ha alcun vantaggio effettivo nell'applicare il *Clock Gating*: il clock commuta poco rispetto a quanto dovrebbe fare e i segnali di controllo del *gating* commutano troppo spesso, rendendo la tecnica totalmente inefficace.
Si sono quindi modificate le probabilità di clock e reset per cercare di avere un risultato più coerente con quanto atteso, ottenendo i risultati contenuti in tabella 3.5.

Internal Power	*Switching Power*	*Leakage Power*	*Total Power*
$26.507\mu W$	$7.5627\mu W$	$4.3465\mu W$	$38.4163\mu W$

Tabella 3.5: *report_power* del circuito con *Clock Gating* usando il comando *set_switching_activity* su RST e CLK

Valgono gli stessi ragionamenti fatti per il caso senza *Clock Gating*: la potenza si riduce, ma di poco e il *Clock Gating* continua a non apportare significativi vantaggi, sempre a causa dell'eccessiva attività dei segnali di *enable*.
Modificando le probabilità e le attività di *INCA* e *INCB*, si ottiene quanto segue:

Internal Power	*Switching Power*	*Leakage Power*	*Total Power*	*Cells*
$13.3592\mu W$	$1.8292\mu W$	$4.257\mu W$	$19.4454\mu W$	78

Tabella 3.6: *report_power* del circuito con *Clock Gating* usando il comando *set_switching_activity* su RST, CLK, INCA e INCB

I risultati della simulazione sono concordi con quanto supposto: si ha un vantaggio circa del 25% rispetto al caso senza *Clock Gating*. Sebbene la potenza dovuta alle commutazioni delle capacità e la potenza di *leakage* siano aumentate, poiché vi sono più elementi nel circuito (il sintetizzatore ha utilizzato 4 *gate* in più per poter applicare il *Clock Gating*), vi è un enorme risparmio nella potenza interna, che rende il *Clock Gating* efficace.

3.2.1 Ancora più *Clock Gating*?

Il *Clock Gating* sintetizzato automaticamente da Synopsys viene applicato solo ai registri di ingresso e non a quello di uscita: questo è dovuto a come viene interpretata la descrizione *VHDL* comportamentale dal programma. Si sono quindi osservate quali fossero le differenze tra i registri di ingresso e quello di uscita a livello *VHDL*, per poterle uniformare e far comprendere al sintetizzatore dove applicare il *Clock Gating*. Si è scoperto che se esiste un segnale che ha la funzione di *enable* per un dato registro, esso viene sintetizzato come un elemento di *Clock Gating*, se richiesto al sintetizzatore. Si è modificato il file *.vhd* aggiungendo un segnale (*INCA OR INCB*) e lo si è posto come *enable* sul registro di uscita.
Si è passati alla fase di sintesi: le considerazioni sono le stesse fatte per il caso con *Clock Gating* solo sui registri di ingresso. La differenza principale è sita nel numero di celle sintetizzati, che porta ad un aumento generale di tutte le potenze di *switching* e *leakage*.
In tabella 3.7 sono riportati tutte le analisi di potenza.

	Internal Power	*Switching Power*	*Leakage Power*	*Total Power*	*Cells*
Default E_{SW}	$32.9285\mu W$	$8.634\mu W$	$4.0098\mu W$	$45.5723\mu W$	
E_{SW} su RST e CLK	$27.5938\mu W$	$9.6973\mu W$	$4.4268\mu W$	$41.7180\mu W$	
E_{SW} su tutto	$10.0862\mu W$	$2.4722\mu W$	$4.343\mu W$	$16.9015\mu W$	80

Tabella 3.7: *report_power* del circuito con *Clock Gating* applicato a tutto il circuito

Si ha un ulteriore guadagno rispetto al caso precedente, sempre grazie alla riduzione sulla potenza di corto circuito.
Come previsto inoltre il numero di celle è aumentato passando da 78 a 80: queste celle aggiuntive non sono tuttavia dei *latch* e delle porte AND, come ci si sarebbe aspettato, ma sono dei blocchi specifici presi dalla libreria del sintetizzatore che hanno il solo scopo di applicare il *Clock Gating*.

3.2.2 Un metodo automatico per annotare le attività

In questa sezione del laboratorio si è impiegato un metodo per calcolare, in modo automatico, il consumo di potenza. Questo consiste nell'assegnare delle *Switching Activity* opportune a ciascuna porta per avere delle simulazioni più realistiche della potenza. Fino ad ora questo processo è stato fatto tramite degli appositi comandi, i quali permettevano di impostare manualmente, segnale per segnale, l'attività che più si riteneva adeguata. Questa operazione può essere sostituita da un sistema di annotazione automatica tramite il file chiamato *SAIF*. Il processo di generazione del file *SAIF* parte dalla descrizione *RTL* del dispositivo, il quale viene preso dal sintetizzatore che lo analizza, lo elabora e genera un file *SDF*. Il file *SDF* contiene tutte le informazioni sui ritardi degli elementi del circuito. Esso viene poi letto dal simulatore logico per poter effettuare una simulazione più realistica del circuito. Da questa simulazione vengono estratti tutti i valori delle *Switching Activity* dei vari nodi, che vengono salvati in un file *VCD*, ed infine convertiti nel file *SAIF* che viene elaborato da Synopsys per restituire un stima della potenza dinamica più coerente.

3.3 *Pipelining* e parallelizzazione

Dati i parametri tecnologici con cui si è implementato il circuito del punto precedente, si sono ricavati i dati riportati in tabella 3.8.

Originale	
Percorso critico	$140ns$
Frequenza di clock	$\frac{1}{MaxDelay} = 7.14MHz$
V_{DD}	$1V$
Potenza dissipata	$P_{nom} \cdot \frac{f_{max}}{f_{nom}} = 10.73\mu W \cdot \frac{7.14MHz}{5MHz} = 15.32\mu W$
Area	$1.747mm^2$

Tabella 3.8: Soluzione originale @ 7.14MHz

Si è supposto, viste le dimensioni del circuito, che la tecnologia adottata fosse micrometrica: questo permette di ipotizzare che la dissipazione di potenza sia dovuta solamente al contributo dinamico, trascurando quindi le correnti di *leakage*. Questo consente di utilizzare la seguente formula:

$$P \propto f \cdot V_{DD}^2$$

Si è quindi cercato di parallelizzare il circuito (figura 3.6): sebbene si siano calcolati tutti i parametri richiesti, si è subito notato un problema nella parallelizzazione. Il dispositivo presenta una retroazione sugli elementi di memoria che, funzionando alternativamente, non conterranno il valore corretto.

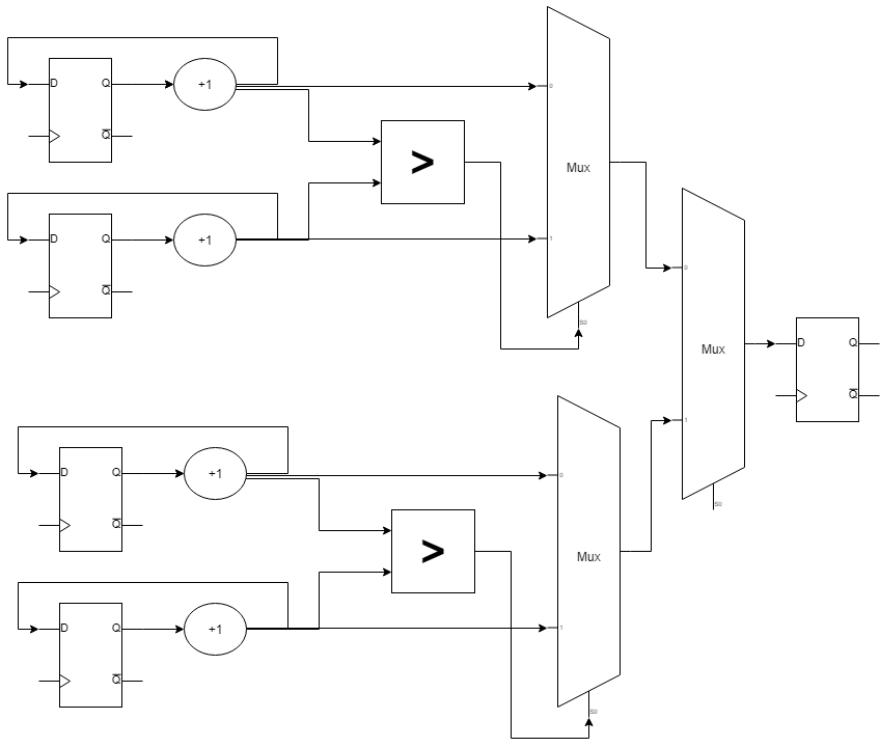

Figura 3.6: Parallelized circuit

Il ritardo massimo viene incrementato a causa dell'inserimento del *multiplexer* di uscita: tuttavia la frequenza di lavoro del registro di uscita può rimanere la stessa, poiché si terrà in conto dell'aumento del ritardo durante il dimensionamento della tensione di alimentazione.
Per calcolare la potenza è necessario prima trovare la tensione di alimentazione minima che il circuito può ora sostenere. Note le formule fornite nella traccia del laboratorio si ha:

$$T(u) = \frac{2}{f} = 280ns$$

Questo sarà il tempo che ogni segnale avrà nel circuito per potersi propagare: l'aumento del ritardo nel percorso critico non va a inficiare sul circuito in quanto il periodo del clock in questione è molto maggiore di quel tempo critico.

$$u = \frac{0.25\frac{T(u)}{T_{nom}}}{\frac{T(u)}{T_{nom}} - 0.75} = \frac{0.25\frac{280ns}{152ns}}{\frac{280ns}{152ns} - 0.75} = 0.4217$$

$$P = 2P_{cir} + P_{mux} + P_{regC} = 2(10.13\mu W \cdot \frac{3.57MHz}{5MHz} \cdot u^2) + (1.67\mu W + 0.6\mu W)\frac{7.14MHz}{5MHz} = 5.815\mu W$$

Si ottengono quindi i risultati in tabella 3.9.

Parallelizzazione	
Percorso critico	$152ns$
Frequenza di clock	$3.57MHz$
V_{DD}	$422mV$
Pootenza dissipata	$5.29\mu W$
Area	$3.292mm^2$
Overhead	$1.545mm^2$

Tabella 3.9: Soluzione con un livello di parallelizzazione

Si nota come la tecnica sia efficiente riducendo la potenza di un fattore 3 a discapito quasi di un *overhead* sull'area pari quasi all'area totale del circuito senza parallelizzazione.
Si è quindi passati alla tecnica del *pipelining* (figura 3.7): si è scelto di spezzare il circuito in modo che i percorsi combinatori fossero il più simili possibile. Si sono quindi aggiunti dei registi tra i due incrementatori e il comparatore: in particolare si è ritenuto opportuno porre i registri dopo la ramificazione della retroazione per evitare problemi di incoerenza tra i dati.

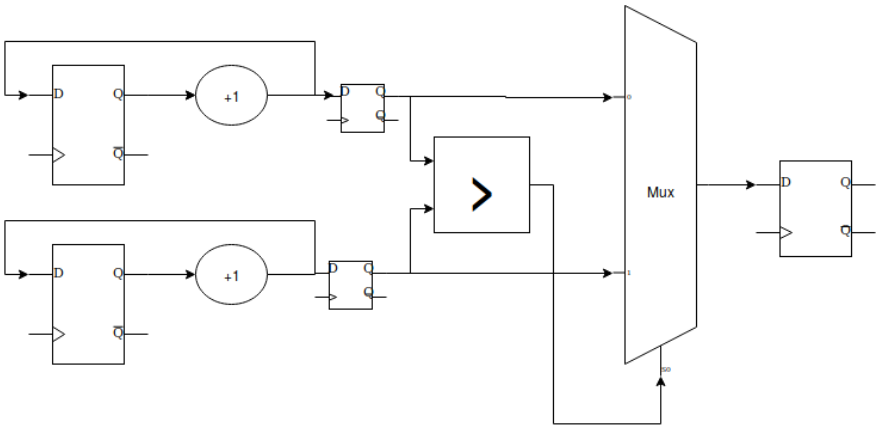

Figura 3.7: Circuito con *pipelining*

Per il calcolo dei vari parametri si sono seguiti gli stessi ragionamenti, ma questa volta il percorso critico è più breve: mantenendo la frequenza costante si può abbassare la tensione di alimentazione.

$$T(u) = 140ns$$

$$T_{nom} = 100ns$$

$$u = \frac{0.25\frac{T(u)}{T_{nom}}}{\frac{T(u)}{T_{nom}} - 0.75} = \frac{0.25\frac{140ns}{100ns}}{\frac{140ns}{100ns} - 0.75} = 0.5385$$

1 livello di *pipe*	
Percorso critico	$100ns$
Frequenza di clock	$7.14MHz$
V_{DD}	$539mV$
Potenza dissipata	$4.939\mu W$
Area	$2.385mm$
Overhead	$611mm^2$

Tabella 3.10: Soluzione con un livello di *pipe*

Anche questa volta il guadagno è netto: oltre a una riduzione ancora più accentuata sui consumi, anche l'*overhead* dell'area si è ridotto rispetto al caso precedente.
Si è quindi aumentato il numero di livelli di *pipe* e di parallelizzazione come richiesto.
Per quanto riguarda il *pipelining* l'unica possibilità per aumentare le prestazioni è spezzare un'altra volta il circuito: il percorso critico comprende sia un *multiplexer* sia un comparatore, spezzandolo è quindi possibile ridurre il percorso critico, potendo quindi ridurre ancora la tensione di alimentazione.

$$T(u) = 140ns$$

$$T_{nom} = 86ns$$

$$u = \frac{0.25\frac{T(u)}{T_{nom}}}{\frac{T(u)}{T_{nom}} - 0.75} = \frac{0.25\frac{140ns}{86ns}}{\frac{140ns}{86ns} - 0.75} = 0.4636$$

2 livelli di *pipe*	
Percorso critico	$86ns$
Frequenza di clock	$7.14MHz$
V_{DD}	$434mV$
Potenza dissipata	$4.2135\mu W$
Area	$3.342mm^2$
Overhead	$1.595mm^2$

Tabella 3.11: Soluzione con due livelli di *pipe*

La potenza si è nuovamente ridotta, ma l'*overhead* è più che raddoppiato. Questa opzione è da considerarsi ottima solo se l'area non è effettivamente un problema ai fini del progetto.

Si è infine cercata una soluzione ibrida (figura 3.8) che cercasse di minimizzare il più possibile il consumo di potenza: si è parallelizzato il circuito lasciando i due stadi di *pipe* ottenendo i valori in tabella 3.12.

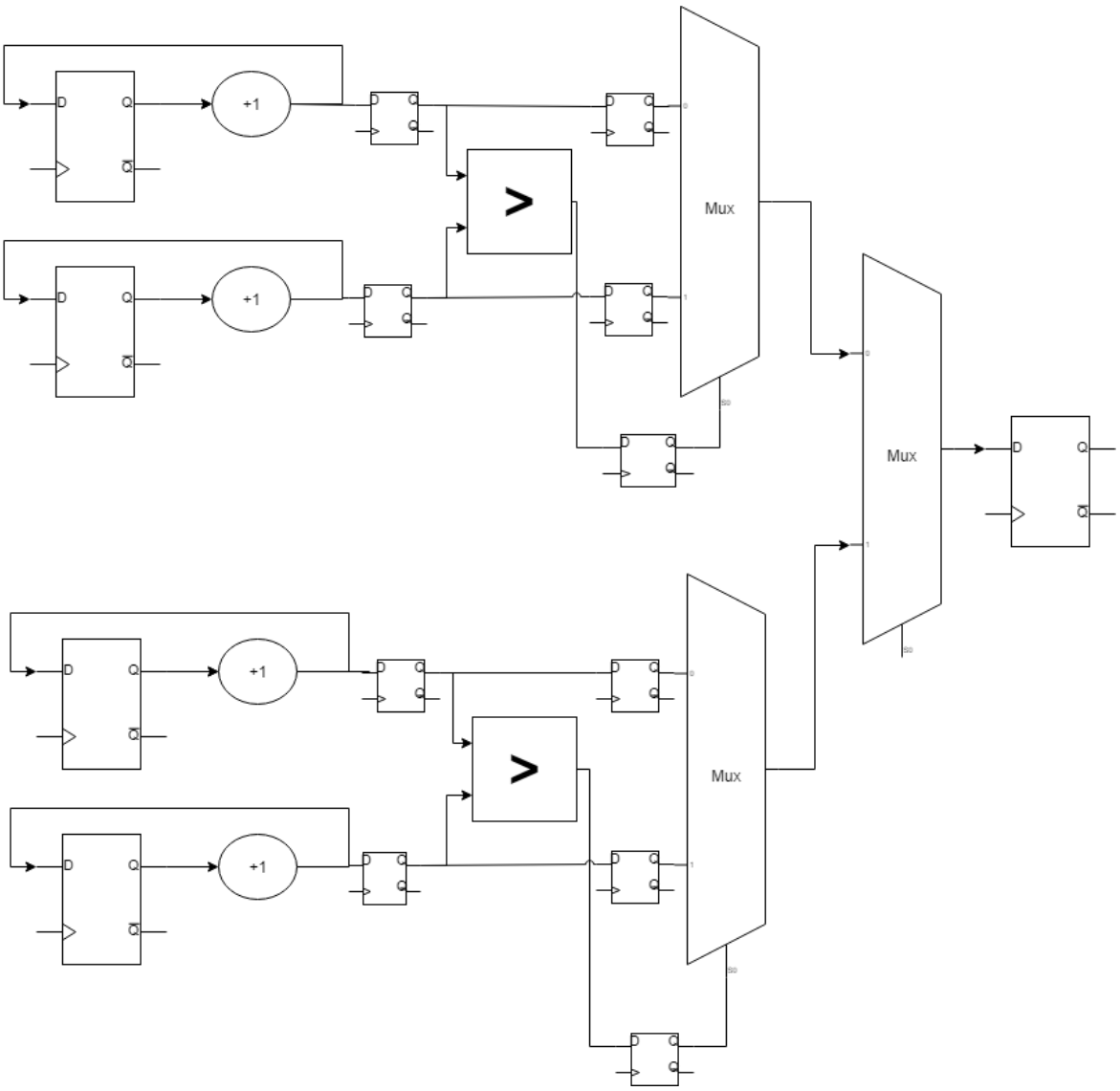

Figura 3.8: Circuito con *pipelining* e parallelizzazione

$$T(u) = 280ns$$

$$T_{nom} = 86ns$$

$$u = \frac{0.25\frac{T(u)}{T_{nom}}}{\frac{T(u)}{T_{nom}} - 0.75} = \frac{0.25\frac{280ns}{86ns}}{\frac{280ns}{86ns} - 0.75} = 0.3248$$

Pipelining e parallelizzazione	
Percorso critico	$86ns$
Frequenza di clock	$7.14MHz$ e $3.57MHz$
V_{DD}	$325mV$
Potenza dissipata	$5.2207\mu W$
Area	$6.482mm^2$
Overhead	$4.735mm^2$

Tabella 3.12: Soluzione con *pipelining* e parallelizzazione

Aumentando il livello di parallelizzazione del circuito si riesce a ridurre ulteriormente la tensione di alimentazione. Questa tecnica, tuttavia, comporta un aumento significativo dell'area del circuito e il rischio che, siccome si è supposta la tensione di soglia dei transistor pari a $\frac{1}{4}$ di quella di alimentazione originaria, non si riesca a portare i transistor in zona lineare. Per questi motivi e per il fatto che una parallelizzazione superiore a 2 livelli necessiterebbe di un segnale di clock supplementare, questa tecnica non è stata ritenuta abbastanza vantaggiosa.
Aumentare il livello di *pipelining* non porterebbe ad alcun vantaggio dato che, con 2 livelli, il percorso critico risulta già essere il comparatore stesso, che non è modificabile.

3.3.1 Parallelizzazione corretta

Osservando la waveform del circuito parallelizzato si nota come non si abbia più la coerenza con i dati del circuito originale. Questo è dovuto al fatto che i due rami della parallelizzazione campionano con fasi differenti del clock e potrebbe avvenire che un dato disponibile per uno dei due non lo sia per l'altro in fase di campionamento e si crei un'incoerenza tra i dati salvati negli elementi di memoria di un ramo e quelli salvati negli elementi di memoria dell'altro. Una possibile soluzione a questo problema è quella di parallelizzare solo la parte finale del circuito, escludendo, cioè, la parte con la retroazione e gli incrementatori. Il circuito finale è visibile in figura 3.9.

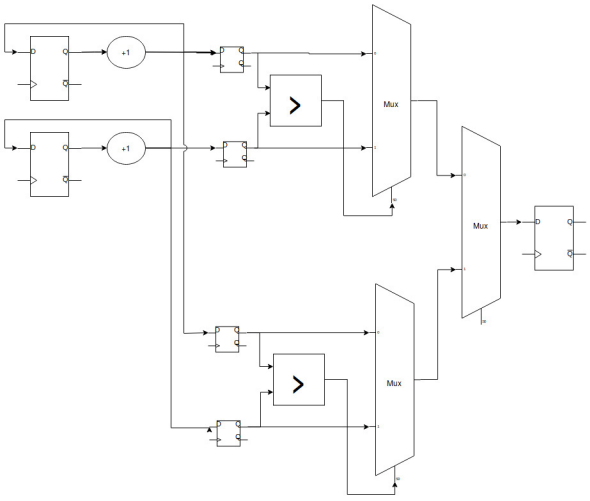

Figura 3.9: Circuito parallelizzato funzionante

3.4 *.vhd*

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_signed.all;

entity inccomp is
Port(  C: out std_logic_vector(7 downto 0);
  ck: in std_logic;
  rst: in std_logic;
  INCA: in std_logic;
  INCB: in std_logic);
end inccomp;

architecture behavioral of inccomp is

signal syncha, synchb : std_logic_vector(7 downto 0);
signal ENC : std_logic;

begin
  ENC <= INCA or INCB;
  p1: process(ck,rst)
  variable tmpa, tmpb : std_logic_vector(7 downto 0);
  begin
    if rst='1' then
      syncha <= (others => '0');
    synchb <= (others => '0');
    C <= (others => '0');
    elsif ck'event and ck='1' then
      tmpa:= syncha;
    tmpb:= synchb;
      if INCA='1' then
      syncha <= syncha+1;
      tmpa:= tmpa+1;
    end if;
    if INCB='1' then
      synchb <= synchb+1;
      tmpb:= tmpb+1;
    end if;
    if ENC = '1' then
      if ((tmpa)>(tmpb)) then
        C <= tmpa;
      else
        C <= tmpb;
      end if;
    end if;
    end if;
  end process;
end behavioral;
```

Listing 6: VHDL corretto per la totale implementazione del clock gating da parte di synopsys

CAPITOLO 4

Codifica del bus

4.1 Simulazione

In questo capitolo si analizzeranno come varie codifiche dei bus possono variare anche significativamente la potenza del circuito. In particolare sono prese in considerazione le seguenti codifiche:

- non-encoded bus
- bus-invert
- transition-based
- gray
- T0

4.1.1 Non-encoded

Si è partiti dall'analisi di un bus senza codifiche di trasmissione. Il testbench di questo circuito sfrutta due algoritmi differenti per la generazione degli indirizzi e dei dati. In particolare i dati sono letti da un file di valori generati casualmente mentre gli indirizzi sono incrementati fino all'indirizzo 63 a partire da 0, effettuano un salto casuale, ricominciano ad incrementare fino a tornare a 63 e così via. La waveform dei dati è visibile in figura 4.1.

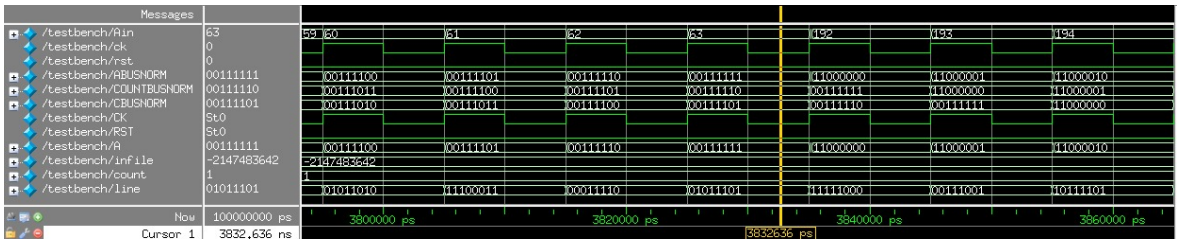

Figura 4.1: Waveform indirizzi non codificati

Dall'analisi della *Switching Activity* degli indirizzi e dei dati ottenuta con Modelsim, si ottengono i valori in tabella 4.1 e 4.2. Si può osservare come quella dei dati sia quasi identica per ogni bit vista la loro casualità mentre quella degli indirizzi sia circa dimezzata andando versi i bit più significativi.

bit7	bit6	bit5	bit4	bit3	bit2	bit1	bit0
97	118	312	624	1249	2499	4999	9999

Tabella 4.1: *Switching Activity* indirizzi non codificati

bit7	bit6	bit5	bit4	bit3	bit2	bit1	bit0
4974	5021	4987	4972	4959	4965	5023	5060

Tabella 4.2: *Switching Activity* dati non codificati

Per i dati la codifica transition-based è stata ritenuta come quella meno adatta per il fatto che, data la casualità dei dati, non c'è una prevalenza abbastanza rilevante degli zeri rispetto agli uni.

4.1.2 Bus-invert, Transition-based, Gray

Bus-invert

Ora si analizza il vantaggio dell'applicazione di una codifica di tipo bus-invert che, però, implica l'invio di un bit aggiuntivo per la segnalazione o meno di un'inversione sui dati inviati. Applicando questa tecnica ai dati e agli indirizzi si ottiene la *Switching Activity* in tabella 4.3 e 4.4 rispettivamente.

bit8	bit7	bit6	bit5	bit4	bit3	bit2	bit1	bit0
3650	3576	3611	3617	3640	3613	3601	3639	3722

Tabella 4.3: *Switching Activity* dati codifica Bus Invert

bit8	bit7	bit6	bit5	bit4	bit3	bit2	bit1	bit0
624	527	506	312	0	625	1875	4375	9375

Tabella 4.4: *Switching Activity* indirizzi codifica Bus Invert

Transition-based

La codifica transition-based è molto vantaggiosa, come detto in precedenza, in presenza di una probabilità quasi pari a 1 di avere zeri sul bus (bassissima probabilità di avere uni). Questa codifica, infatti, produce un 1 in presenza di una transizione (positiva o negativa) dell'ingresso e uno 0 negli altri casi. Una possibile implementazione è visibile nel circuito in figura 4.2.

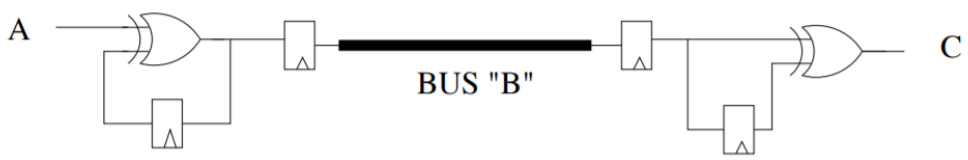

Figura 4.2: Circuito codifica transition-based

Applicando questa tecnica ai dati e agli indirizzi dati si ottiene la *Switching Activity* in tabella 4.5 e 4.6 rispettivamente.

bit7	bit6	bit5	bit4	bit3	bit2	bit1	bit0
4816	3776	4992	4992	5000	5000	5000	5000

Tabella 4.5: *Switching Activity* dati codifica transition-based

bit7	bit6	bit5	bit4	bit3	bit2	bit1	bit0
5003	4946	4938	4997	5125	4989	5000	4994

Tabella 4.6: *Switching Activity* indirizzi codifica transition-based

Gray

La codifica Gray è molto vantaggiosa quando si rimane in sequenza in quanto è necessaria la variazione di un solo bit. Il circuito in figura 4.3 mostra una possibile implementazione di un codificatore/decodificatore Gray.

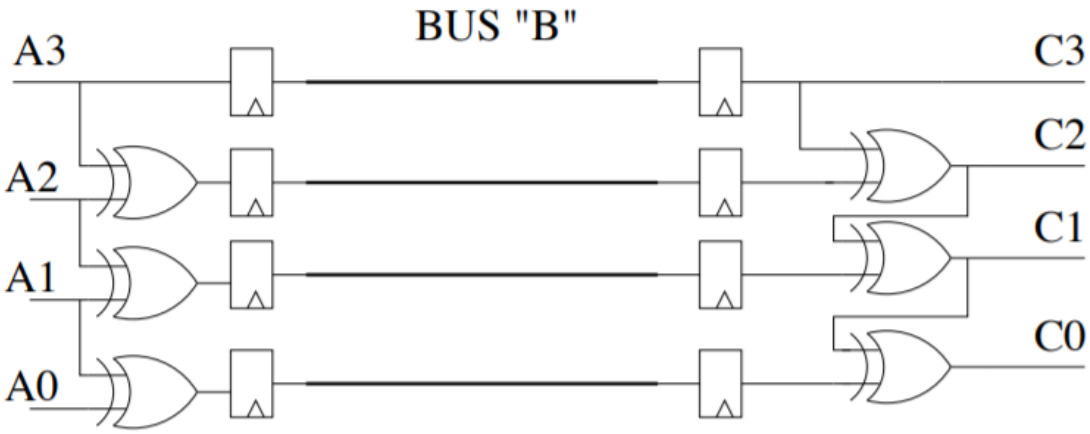

Figura 4.3: Circuito codifica Gray

Il circuito è stato testato con alcuni ingressi come visibile in tabella 4.7.

A	B	C
0010	0011	0010
0011	0010	0011
0100	0110	0100

Tabella 4.7: Test circuito Gray

Applicando questa tecnica ai dati e agli indirizzi dati si ottiene la *Switching Activity* in tabella 4.8 e 4.9 rispettivamente.

bit7	bit6	bit5	bit4	bit3	bit2	bit1	bit0
4974	5087	4952	4981	4937	5056	4962	5075

Tabella 4.8: *Switching Activity* dati codifica Gray

bit7	bit6	bit5	bit4	bit3	bit2	bit1	bit0
97	55	194	312	625	1250	2500	5000

Tabella 4.9: *Switching Activity* indirizzi codifica Gray

Confronto tra i risultati

Dall'analisi dei risultati delle tabelle 4.1, 4.4, 4.6 e 4.9, si nota come la migliore codifica per gli indirizzi sia quella di Gray in quanto permette una riduzione di circa il 50% della *Switching Activity* del primo bit rispetto al caso in cui non venga applicata alcuna codifica. Per quanto riguarda la *Switching Activity* dei dati nelle tabelle 4.2, 4.3, 4.5 e 4.8, la codifica Bus Invert risulta quella che garantisce una *Switching Activity* minore (3700 toggle in media) anche se a scapito dell'aggiunta di un segnale nel bus (il bit 8).

4.1.3 Codifica T0

Di grande importanza è anche la codifica T0. Questa è molto vantaggiosa se si va in sequenza incrementando via via un valore; infatti, tramite l'utilizzo di un bit in più, è possibile segnalare se si è in sequenza (bit a 0) lasciando invariati tutti gli altri bit o se si desidera saltare (bit a 1). In figura 4.4 è visibile più chiaramente come funziona l'algoritmo.

$$\left(B^{(t)}, INC^{(t)}\right) = \begin{cases} \left(B^{(t-1),1}\right) & if \quad b^{(t)} = b^{(t-1)} + 1 \\ \left(b^{(t)}, 0\right) & otherwise \end{cases}$$

Figura 4.4: Implementazione codifica T0

A questo punto si è scritto un file *.vhd* che implementasse tale codifica (sezione 4.3).
Dalla sua analisi con Synopsys si sono ottenute le *Switching Activity* visibili in tabella 4.10 e 4.11 per dati e indirizzi.

bit8	bit7	bit6	bit5	bit4	bit3	bit2	bit1	bit0
73	4974	5021	4987	4972	4957	4963	5001	5028

Tabella 4.10: *Switching Activity* dati codifica T0

bit8	bit7	bit6	bit5	bit4	bit3	bit2	bit1	bit0
118	29	24	0	0	0	0	0	0

Tabella 4.11: *Switching Activity* indirizzi codifica T0

Si nota come, per quanto riguarda gli indirizzi, l'utilizzo di tale codifica comporti un vantaggio notevole annullando la *Switching Activity* per i bit meno significativi e riducendola per quelli più significativi. Questo risultato era facilmente prevedibile data la dinamica degli indirizzi che prevede di andare in sequenza per la maggior parte del tempo.

4.2 Sintesi

Sfruttando la back annotation si è valutato il consumo energetico delle varie codifiche sopra analizzate ottenendo i risultati visibili in tabella 4.12. Come si può notare, l'analisi è stata fatta per varie capacità (1fF, 10fF, 50fF e 100fF) delle linee e sia nel caso degli indirizzi che in quello dei dati.

POWER	Indirizzi				Dati			
uW	1f	10f	50f	100f	1f	10f	50f	100f
BUSNORMAL	22.05	25.22	37.79	55.65	18.25	23.03	42.04	67.73
BUSINVERT	14.11	19.1	30.12	44.89	15.68	18.01	33.1	45.95
TRANSITION-BASED	18.52	21.7	35.68	36.6	23.76	21.3	40.95	66.01
GRAY	14	9.76	15.66	25.2	18.3	9.91	19.88	31.79
T0	39.07	38.25	38.13	38.07	39.01	38.25	38.13	38.08

Tabella 4.12: Confronto tra consumi totali con e senza codifica su indirizzi (sinistra) e dati (destra)

Osservando i valori ottenuti, si nota come con le codifiche bus-invert, transition-based e Gray si sia ottenuto un miglioramento in termini di consumo nel caso degli indirizzi anche se l'ultima codifica citata si è mostrata la migliore. Per quanto riguarda la trasmissione dei dati, solo le codifiche bus-invert e Gray hanno mostrato un miglioramento e, per capacità della linea superiori ai 10fF, la codifica di Gray è risultata la migliore. Secondo le nostre previsioni, la codifica T0 sarebbe dovuta essere quella con l'efficienza migliore data la sua bassa *Switching Activity* ma, dai risultati ottenuti con l'analisi di potenza, è risultata la peggiore. Questo è dovuto al fatto che il circuito di codifica e decodifica ha un consumo molto elevato, tanto da rendere l'utilizzo di tale codifica svantaggioso.

4.3 *.vhd*

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_unsigned.all;
use IEEE.numeric_std.all;
use work.all;

entity busT0 is
port (  ck : in std_logic;
  rst : in std_logic;
  A : in std_logic_vector(7 downto 0);
  B : out std_logic_vector(8 downto 0);
  C : out std_logic_vector(7 downto 0));
end busT0;

architecture behavioral of busT0 is

signal T0 : std_logic;
signal count : std_logic_vector(7 downto 0);
signal Aold, Abus, buss : std_logic_vector(7 downto 0);

begin

  penc: process(ck,rst)
  begin
    if rst='1' then
    Abus <= (others => '0');
    T0 <= '0';
    Aold <= (others => '0');
    buss <= (others => '0');
    elsif ck'event and ck='1' then
    if A = Aold then
      T0 <= '0';
      buss <= Abus;
    else
      T0 <= '1';
      Abus <= A;
      buss <= A;
      end if;
    Aold <= std_logic_vector(unsigned(A)+"00000001");
    end if;
  end process;

  pdec: process(ck,rst)
  begin
    if rst='1' then
      count <= (others => '0');
    elsif ck'event and ck='1' then
      if T0 = '0' then
      count <= std_logic_vector(unsigned(count)+"00000001");
      else
      count <= buss;
      end if;
    end if;
  end process;
  C <= count;
  B(7 downto 0) <= buss;
  B(8) <= T0;

end behavioral;
```

Listing 7: Implementazione codifica T0 in VHD

CAPITOLO 5

Leakage: uso di Spice per la caratterizzazione delle celle e carta&penna per l'organizzazione della memoria

5.1 Caratterizzazione della libreria di una porta

In questo capitolo caratterizzeremo la libreria di una porta NAND (figura 5.1) sfruttando un software Spice-like.

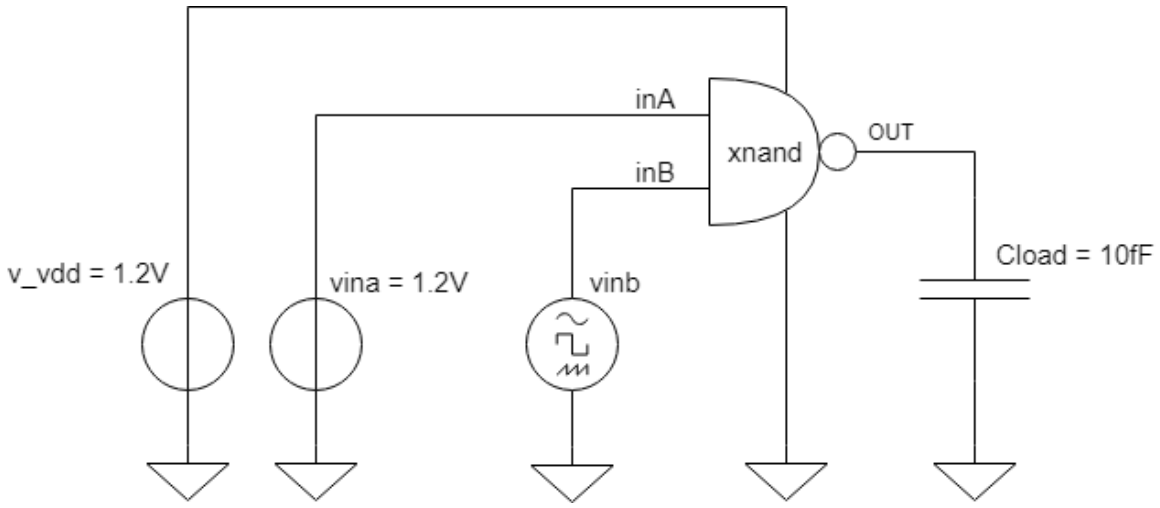

Figura 5.1: Schema porta NAND libreria

L'ingresso vinb è generato sfruttando una *PWL (Piecewise Linear Function)* che genera segnali dalla forma arbitraria indicando, istante per istante, il valore dell'ampiezza del segnale. Sfruttando il comando *.measure* è possibile misurare il tempo che intercorre tra il momento in cui un segnale assume un valore in un determinato nodo e quando ne assume un altro arbitrariamente scelto. Ai fini della simulazione è stato necessario aggiungere il comando: "*.measure tran nanddelayLH TRIG V(inB) VAL='alim*0.5' FALL=1 TARG V(out) VAL='alim*0.5' RISE=1*" per visualizzare anche

il tempo di propagazione "low to high". Dalla simulazione con Spice, sono stati ottenuti i seguenti tempi per la simulazione:

Rise time NAND	89.303 ps
Fall time NAND	74.319 ps
Delay HL NAND	48.993 ps
Delay LH NAND	56.490 ps

Tabella 5.1: Simulazione in Spice porta NAND

e una total power dissipation pari a 6.79 nW. I valori in tabella 5.1 rappresentano i tempi di rise, falling, propagazione "high to load" e propagazione "low to high". Si sono ora cercati gli stessi risultati in maniera grafica usando il software EZwave (figura 5.2) e i cursori, ottenendo la tabella 5.2.

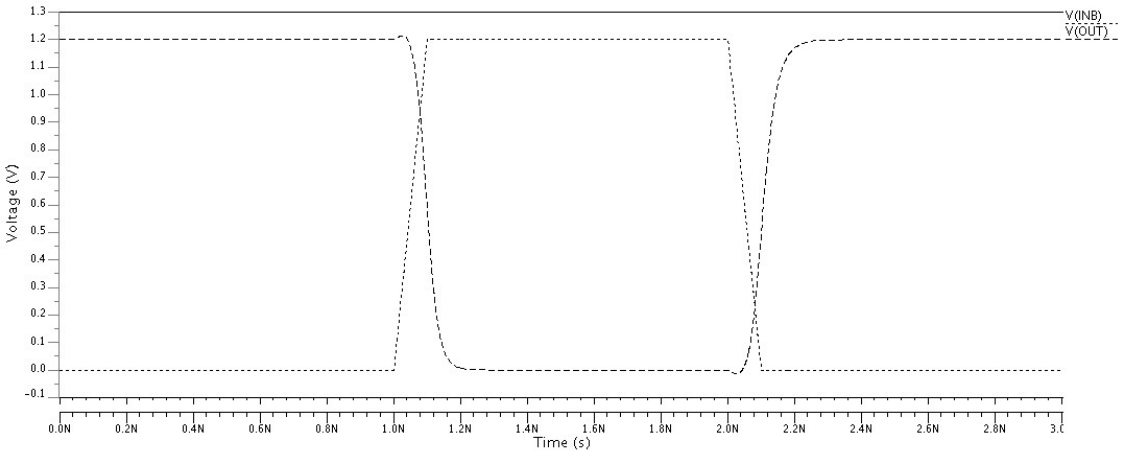

Figura 5.2: Waveform porta NAND EZwave

RNAND	90 ps
FNAND	74 ps
NANDDELAYHL	49 ps
NANDDELAYLH	57 ps

Tabella 5.2: Simulazione in EZwave porta NAND

5.1.1 Misurazione della tensione di soglia

Sfruttando il comando *.print* di Spice è stato possibile ottenere i valori delle tensioni di soglia dei transitor che compongono la porta NAND (tabella 5.3). Le tensioni di soglia sono circa 1/4 di quelle di alimentazione come ci aspetteremmo. Ad ogni modo la tensione di soglia del secondo NMOS risulta diversa da quella del primo per via dell'effetto body che vi agisce.

NMOS 0	313.71 mV
NMOS 1	272.41 mV
PMOS 0	-247.12 mV
PMOS 1	-247.12 mV

Tabella 5.3: Tensione di soglia porta NAND

5.2 Caratterizzazione di una porta al variare delle capacità di uscita

Adesso si analizzerà una porta NAND con un carico in uscita variabile. Questo è stato fatto creando un vettore con i vari valori assunti nel tempo dalla capacità di uscita. A questo punto si sono ottenute le seguenti misure:

Output loads	Fall Time	Prop. delay HL	I_{GND}	I_{VDD}	I_{LOAD}
5 aF	68.511 ps	20.524 ps	76.53 uA	-47.912 uA	-126.03 nA
50 aF	68.830 ps	20.851 ps	77.056 uA	-47.680 uA	-1.2504 uA
500 aF	72.112 ps	23.980 ps	81.964 uA	-45.637 uA	-16.637 uA
5 pF	98.194 ps	48.55 ps	116.64 uA	-34.365 uA	-75.275 uA
50 pF	296.52 ps	180.16 ps	240.77 uA	-11.039 uA	-230.42 uA

Tabella 5.4: Misurazioni NAND con carico variabile falling

Output loads	Rise time	Prop. delay LH	I_{GND}	I_{VDD}	I_{LOAD}
5 aF	77.226 ps	37.948 ps	45.879 uA	-70.006 uA	151.10 nA
50 aF	67.604 ps	38.286 ps	45.692 uA	-70.423 uA	1.1429 uA
500 aF	71.228 ps	41.521 ps	44.055 uA	-74.383 uA	10.641 uA
5 pF	102.81 ps	66.657 ps	34.693 uA	-102.77 uA	67.574 uA
50 pF	363.91 ps	209.00 ps	12.571 uA	-209.19 uA	199.72 uA

Tabella 5.5: Misurazioni NAND con carico variabile rising

Dalle tabelle 5.4 e 5.5 si nota come le correnti in gioco aumentino all'aumentare del carico in uscita. I picchi si hanno durante le commutazioni dell'uscita, in particolare $I_{GND,F}$ è la corrente che va a GND durante la transizione da alto a basso in uscita, $I_{GND,R}$ durante la transizione da basso ad alto in uscita e $I_{VCC,F/S}$ l'analogo per le correnti a VCC. La potenza dissipata rimane invariata rispetto al caso precedente ed è insensibile al carico di uscita. Gli andamenti delle tensioni e correnti sono visibili in figura 5.3.

5.2.1 Tensione di soglia porta NAND

Le tensioni di soglia dei transistor che compongono la NAND rimangono uguali al caso precedente in quanto sono indipendenti dalla capacità di uscita.

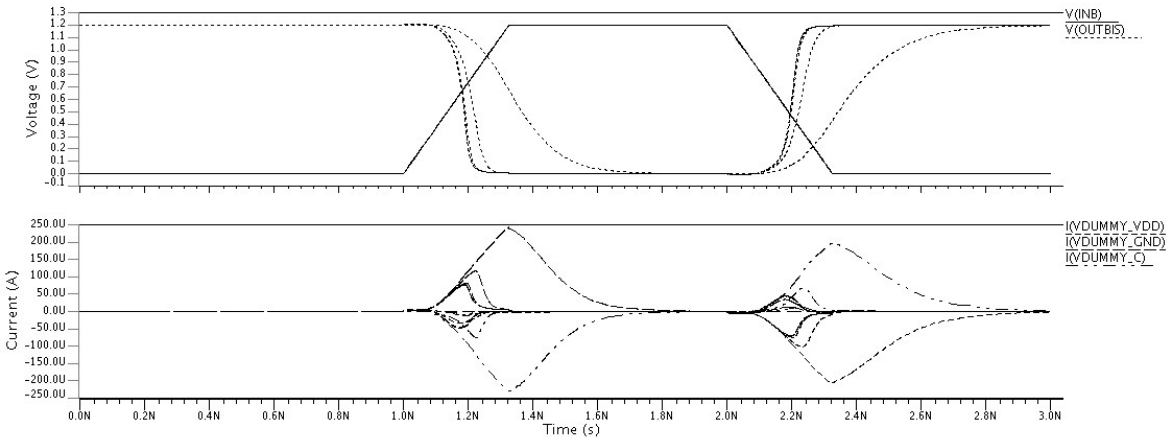

Figura 5.3: Plot V(inb) e V(outbis) e correnti

5.3 Comparazione di porte con dimensioni differenti

Ora si confrontano due porte NAND *high speed* di cui una ottimizzata per pilotare capacità fino a 0.16 fF (X1) e una fino a 1.28 fF (X8). Entrambe sono state testate con una capacità di 60 aF e una di 60fF. I risultati sono visibili in tabella 5.6 e 5.7. La porta X1 risulta più lenta (si notino Rise Time, Fall Time e Propagation Delay) ma presenta una dissipazione minore (tutte le correnti sono più piccole). Facendo un rapporto tra le correnti dissipate si vede che questo è circa 8, come ci si aspetterebbe dato che il rapporto tra le capacità dei gate di X1 e X8 è tale valore.

Output loads	Fall Time	Prop. delay HL	I_{GND}	I_{VDD}	I_{LOAD}
NAND-X1					
60 aF	68.976 ps	20.923 ps	77.172 uA	-47.631 uA	-1.4979 uA
60 fF	341.77 ps	203.71 ps	245.47 uA	-9.4773 uA	-236.49 uA
NAND-X8					
60 aF	63.845 ps	17.465 ps	637.45 uA	-410.35 uA	-1.709 uA
60 fF	106.87 ps	57.44 ps	1.069 mA	-258.52 uA	-812.00 uA

Tabella 5.6: Misurazioni NAND falling HS

Output loads	Rise Time	Prop. delay LH	I_{GND}	I_{VDD}	I_{LOAD}
NAND-X1					
60 aF	67.692 ps	38.361 ps	45.653 uA	-70.514 uA	1.3693 uA
60 fF	425.04 ps	236.61 ps	10.965 uA	-214.72 uA	206.27 uA
NAND-X8					
60 aF	62.271 ps	31.258 ps	387.92 uA	-552.88 uA	1.546 uA
60 fF	113.07 ps	72.256 ps	264.86 uA	-628.61 uA	726.78 uA

Tabella 5.7: Misurazioni NAND rising HS

5.3.1 VT

Le tensioni di soglia della NAND debole sono uguali al caso precedente.
Le tensioni di soglia della NAND forte sono riportate in tabella 5.8.

#MOS	1/3/5/7	2/4/6/8
NMOS	318.93mV	277.63mV
POMS	-241.64mV	-241.64mV

Tabella 5.8: Tensioni di soglia NAND-X8

Le tensioni di soglia sono le stesse del caso precedente: la differenza è che per ogni transistor della porta X1 vi sono ora più transistor in parallelo. Questo comporta la presenza di più tensioni di soglia ma, visto che il posizionamento, e quindi la polarizzazione, dei transistor è la stessa, le tensioni di soglia non cambiano.

5.4 Confronto di porte High Speed e Low Leakage

Analizziamo ora delle porte NAND *low leakage*:

Output loads	Fall Time	Prop. delay HL	I_{GND}	I_{VDD}	I_{LOAD}
NAND-X1					
60 aF	63.646 ps	31.612 ps	51.878 uA	-19.758 uA	-13.351 uA
60 fF	406.23 ps	257.67 ps	194.22 uA	-2.4801 uA	-186.52 uA
NAND-X8					
60 aF	57.371 ps	26.768 ps	404.32 uA	-165.52 uA	-1.5185 uA
60 fF	112.39 ps	78.324 ps	867.10 uA	-70.31 uA	-688.63 uA

Tabella 5.9: Misurazioni NAND falling LL

Output loads	Rise Time	Prop. delay LH	I_{GND}	I_{VDD}	I_{LOAD}
NAND-X1					
60 aF	71.556 ps	52.657 ps	16.961 uA	-42.315 uA	1.0745 uA
60 fF	592.23 ps	331.22 ps	3.6258 uA	-151.76 uA	144.72 uA
NAND-X8					
60 aF	63.48 ps	44.524 ps	132.79 uA	-303.01 uA	1.2191 uA
60 fF	137.67 ps	102.77 ps	68.306 uA	-683.65 uA	543.25 uA

Tabella 5.10: Misurazioni NAND rising LL

Come facilmente intuibile, queste ultime porte NAND *low leakage* risultano più lente (150%) rispetto alle *high speed* viste in precedenza ma presentano anche consumi minori (-150%).

5.4.1 VT

Come facilmente intuibile, anche in questo caso, le tensioni di soglia delle NAND *low leakage* risultano superiori di quelle *high speed*.

NMOS	413.33mV	380.62mV
PMOS	-367.12mV	-367.12mV

Tabella 5.11: NAND-X1

#MOS	1/3/5/7	2/4/6/8
NMOS	418.93mV	386.21mV
PMOS	-367.64mV	-367.64mV

Tabella 5.12: Tensioni di soglia NAND-X8 *low leakage*

5.5 Dipendenza dalla temperatura

Si è quindi analizzato il comportamento della potenza e delle tensioni di soglia al variare della temperatura.

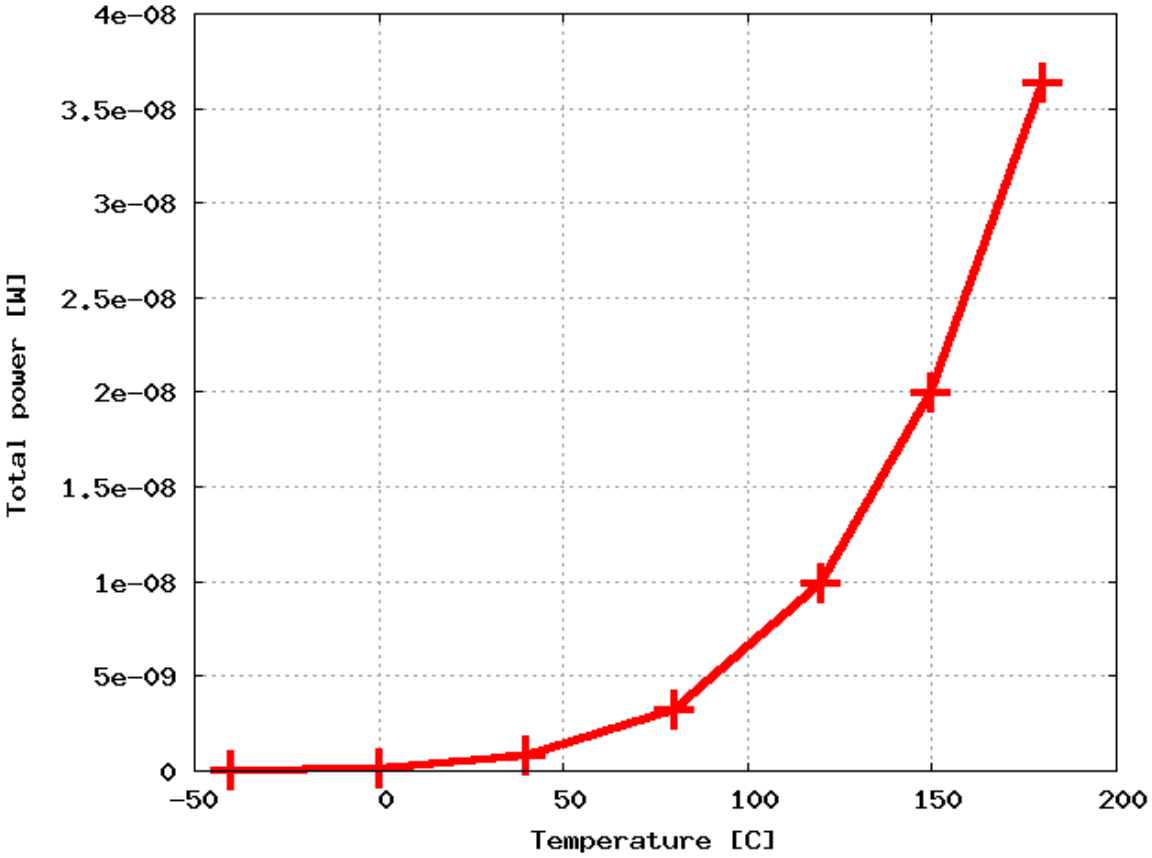

Figura 5.4: Potenza porta NANDLL al variarare della temperatura

Dalla figura 5.4 è possibile vedere come la potenza dissipata aumenti con l'aumento della temperatura: in particolare l'andamento è più che lineare, probabilmente vicino a una parabola. Questo aumento di consumi è ragionevolmente dovuto all'aumento dell'agitazione termica, che va ad incrementare il *leakage* del circuito.

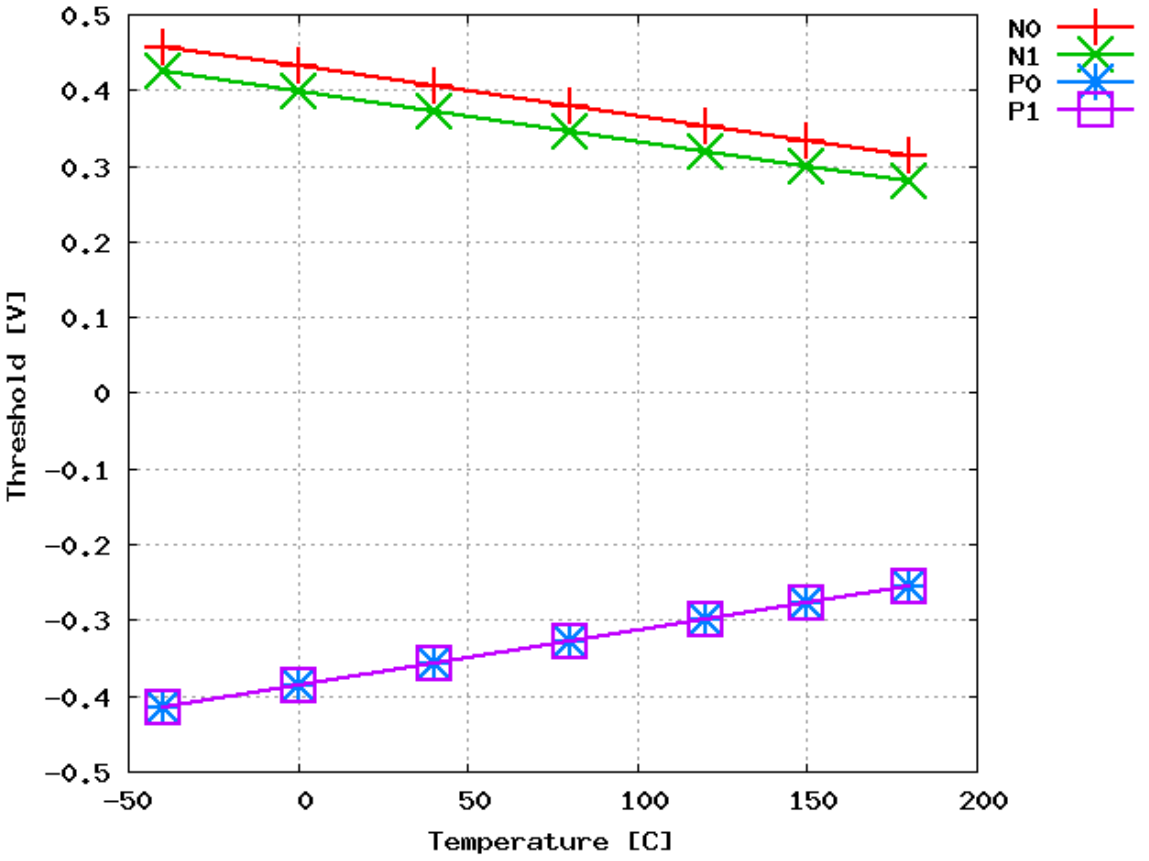

Figura 5.5: Tensioni di soglia NANDLL al variarare della temperatura

La figura 5.5 mostra invece l'andamento delle tensioni di soglia con la temperatura: vi è una decrescita lineare, uguale per tutti i transistori. Questo è sempre dovuto all'agitazione termica che, aumentando l'energia totale dei portatori, rende più semplice la formazione del canale. Questo spiega anche perchè la potenza, come visto, aumenti: il ridursi della tensione di soglia amplifica infatti gli effetti del *leakage* sottosoglia.

5.6 Analisi dei consumi di una memoria

In questa sezione del laboratorio è stato richiesto di analizzare il consumo di potenza di una memoria SRAM da 8192 bit con parallelismo di 16 bit. Si avevano a disposizione i datasheet di memorie di diverse grandezze con diversi parallelismi: da 512 a 8192 bit da 4 fino a 32 bit di parallelismo. In particolare, dalle suddette documentazioni, si sono ricavati i contributi di corrente (in stand-by e durante i cicli di lettura e scrittura), l'area e la capacità dei pin.
In merito alla corrente, questa dipendeva dalle modalità operative: fast, typical e slow nelle quali il consumo via via decresceva. Essendo che la progettazione che si vuole fare è mirata al risparmio energetico, si è deciso di prendere in considerazione la modalità slow.
Dopo aver valutato diverse configurazioni aggregando insieme memorie di diverse grandezze, si è visto come più si aumenta l'aggregazione di memorie via via più piccole, più l'area occupata e il consumo di potenza aumentava (senza, tuttavia, considerare le interconnessioni che avrebbero portato ragionevolmente ad un valore peggiore). La soluzione migliore è stata individuata, perciò, nella SRAM 8192_16_16: questa garantisce il minor consumo nelle tre modalità di funzionamento (stand-by, read,

write) e la minor area.
Ad ogni modo è bene notare che, usando un'unica grande memoria, si perde la modularità che permette, nel caso di letture o scritture di pochi dati, di attivare un sottoinsieme dei banchi di memoria disponibili lasciando in stand-by gli altri. Con quest'ultima soluzione, infatti, il consumo sarebbe pari alla somma dei consumi dei soli banchi selezionati (ipotizzando di trascurare il consumo in stand-by), minore, cioè, del consumo della memoria SRAM 8192_16_16.

CAPITOLO 6

Verifica funzionale

6.1 Verifica VHDL

6.1.1 Un RCA dato

In questa sezione si testerà il progetto di un RCA per trovare eventuali errori. Dal testbench si nota come gli ingressi del dispositivo in test siano generati casualmente in modo che venga rappresentata una situazione il più possibile attinente a quella reale (raramente il RCA dovrà sommare solo numeri consecutivi). Usando il comando *assert* di VHDL per confrontare i risultati della rete RCA con quelli attesi, si è notato che, dopo un numero di cicli del testbench, il risultato era sbagliato. Lo stesso non si otteneva con un numero basso di cicli per via del fatto che la probabilità di trovare un errore non era molto alta e 5 cicli non erano abbastanza per verificare che il circuito funzionasse correttamente. Il tempo richiesto per ottenere l'errore, infatti, è stato di 30ns e dipende dai numeri generati casualmente: cambiando il *seed* si sono ottenuti tempi diversi.
Sfruttando la simulazione di Modelsim è stato possibile trovare e correggere l'errore (dovuto al *carry in* dell'ultimo FA che non era collegato all'ultimo *carry out* ma al penultimo). Aumentando il parallelismo a 64 bit e a 128 bit non è stato notato un incremento del tempo necessario per trovare il primo errore e, quindi, il tempo richiesto è stato sempre di 30 ns.

6.1.2 Un caso più complesso

Per l'analisi del sistema incrementatore e comparatore si è scelto un approccio diverso rispetto a quanto proposto. Dopo aver testato inizialmente il circuito con un testbench a ingressi totalmente casuali, ci si è resi conto che, in una normale situazione di funzionamento, è molto difficile che i segnali varino con uno schema così vario. Rifacendosi anche al laboratorio 3, in cui un circuito funzionalmente identico era stato testato e in cui erano state assegnate delle probabilità ai vari segnali a seconda della loro funzione, si è deciso di testare il circuito tenendo in conto il suo funzionamento.
L'idea è quella di identificare gli stati in cui il circuito può trovarsi e andare a sondarli direttamente per vedere se portano a degli errori. Questo è possibile in quanto il circuito è abbastanza semplice dal punto di vista funzionale. Tuttavia, anche in un caso ancora più complesso, sarebbe sufficiente suddividere il test in più parti, analizzando prima i blocchi singolarmente e salendo man mano di gerarchia per trovare i bachi.
Si sono quindi individuate le principali modalità di funzionamento del circuito: il primo modo consiste nel far incrementare un sommatore, abilitandolo finché non va in *overflow*; il secondo nel caricamento di un numero di partenza da cui iniziare a incrementare, senza però abilitare la somma per verificare che effettivamente il circuito non incrementi quando non deve; l'ultima modalità prevede di continuare

a caricare numeri alternati nei due registri per verificare che il comparatore e il multiplexer di uscita funzionino in modo corretto.
Si è quindi scritto un testbench che permettesse di testare il circuito nelle situazioni appena descritte: non si è scritta la parte relativa al comparatore, in quanto un suo malfunzionamento verrebbe individuato in una delle quattro prove precedenti.
Si è simulato il primo circuito testando il primo sommatore in modalità incrementale e il secondo in modalità senza somma. Si trova subito un errore nel primo sommatore: tutti i primi bit di ogni gruppo di quattro in uscita del sommatore sono sempre uguali a 1, come si può vedere dalla sezione di simulazione in figura 6.1.

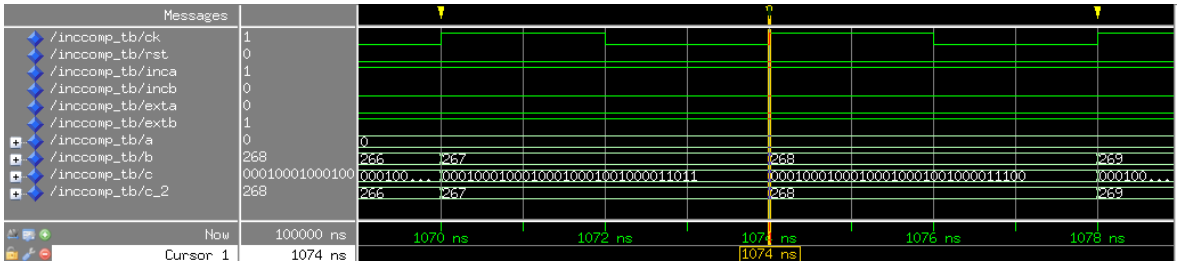

Figura 6.1: Sezione della simulazione di verifica della prima versione del comparatore incrementale

Da questo comportamento si deduce che in qualche modo il *Carry In* di ogni sommatore a 4 bit è impostato di default uguale a 1. L'errore è stato quindi cercato nella rete di generazione del riporto, ma senza successo. Essendo il dispositivo però un *Carry Select Adder* si è pensato che l'errore potesse essere nella selezione del risultato con o senza il *Carry In*. Andando a controllare si è scoperto che l'errore era proprio un'inversione degli ingressi del multiplexer di selezione dei risultati delle somme a 4 bit.
Ci si è però resi conto che il simulatore sovrascrive ogni volta tutti i componenti in quanto sia il sommatore 1 che il sommatore 2 hanno gli stessi nomi. Infatti il baco, che era presente solo nel sommatore 2, è stato riscontrato anche nel sommatore 1, a causa dell'ordine di compilazione. Per questa ragione non si sono testati i restanti circuiti, in quanto esiste la possibilità di non individuare il bug, poiché magari presente nella versione che non viene effettivamente simulata, ma sovrascritta.
Il procedimento resta tuttavia identico nello svolgimento: si simula prima un sommatore in una modalità, poi l'altro e via dicendo, controllando se i risultati rimangono coerenti di volta in volta con l'architettura di riferimento.

6.1.3 *Finite State Machine*

Per l'analisi della macchina a stati che descrive il contatore si è utilizzato nuovamente un set casuale di ingressi.
Questo è stato sufficiente per individuare l'errore nella prima versione sbagliata del dispositivo, che saltava uno stato, passando da up-4 direttamente ad up-6, tuttavia, per gli altri casi, questo si è rivelato fallimentare: impostando un clock molto piccolo ($2ns$ di periodo) e simulando per 200 ms, non si sono riscontrati errori di alcun tipo. Si è cercato di modificare la probabilità degli ingressi, lasciando sempre attivo il segnale di conta, o sempre disattivo il reset, ma nuovamente non si è trovato alcun baco.
Si sono quindi analizzati i file *.vhd* per cercare di comprendere dove potesse essere l'errore, nel tentativo di simularlo e verificare un effettivo malfunzionamento.
Si nota che la seconda versione presenta un *if* aggiuntivo all'interno del process del reset: quando il

sistema si resetta ed è nello stato down-2, invece di tornare in idle, ritorna ad up-1. Dalla simulazione in figura 6.2 risulta evidente lo sfasamento che si ottiene a quel punto tra la versione con bachi e quella funzionante.

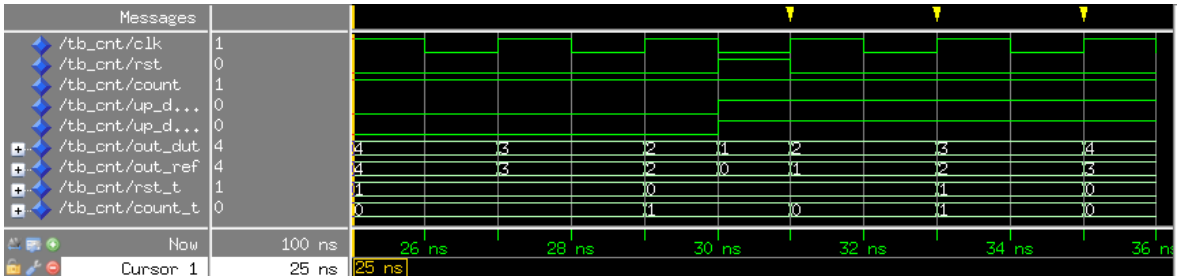

Figura 6.2: Errore nella seconda versione del contatore

La terza versione ha invece un *process* che entra in gioco quando il dispositivo riceve il segnale di clock mentre il segnale di conta è disabilitato. Si è nuovamente fatta una simulazione ad hoc per controllare se l'errore fosse effettivamente quello. Dalla figura 6.3 si nota come, ad un certo punto mentre i contatori sono fermi, il secondo segnale di up_down abbia un'inversione non prevista.

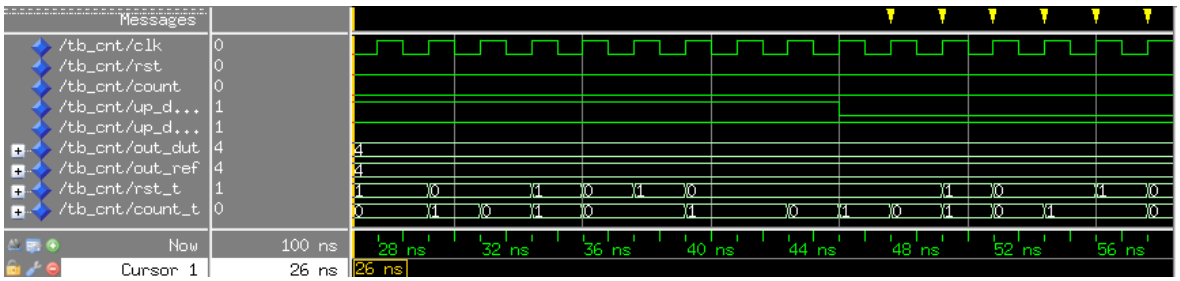

Figura 6.3: Errore nella terza versione del contatore

L'ultima versione, poiché le altre tre presentano bachi, è corretta. Comunque, sia simulando con valori casuali, che controllando il file *.vhd*, non si sono riscontrate anomalie.

6.2 *.vhd* e *.do*

```
vcom -93 -work ./work ./v1/src/utils/constants.vhd
vcom -93 -work ./work ./v1/src/P4_1/carry_generator.vhd
vcom -93 -work ./work ./v1/src/P4_1/carry_logic_network.vhd
vcom -93 -work ./work ./v1/src/P4_1/carry_select_block.vhd
vcom -93 -work ./work ./v1/src/P4_1/g_block.vhd
#vcom -93 -work ./work ./v1/src/P4_1/p4_adder.vhd
vcom -93 -work ./work ./v1/src/P4_1/p4_adder_1.vhd
vcom -93 -work ./work ./v1/src/P4_1/pg_block.vhd
vcom -93 -work ./work ./v1/src/P4_1/pg_network.vhd
vcom -93 -work ./work ./v1/src/P4_1/pg_network_block.vhd
vcom -93 -work ./work ./v1/src/P4_1/sum_generator.vhd
vcom -93 -work ./work ./v1/src/P4_2/carry_generator.vhd
vcom -93 -work ./work ./v1/src/P4_2/carry_logic_network.vhd
vcom -93 -work ./work ./v1/src/P4_2/carry_select_block.vhd
vcom -93 -work ./work ./v1/src/P4_2/g_block.vhd
#vcom -93 -work ./work ./v1/src/P4_2/p4_adder.vhd
vcom -93 -work ./work ./v1/src/P4_2/p4_adder_2.vhd
vcom -93 -work ./work ./v1/src/P4_2/pg_block.vhd
vcom -93 -work ./work ./v1/src/P4_2/pg_network.vhd
vcom -93 -work ./work ./v1/src/P4_2/pg_network_block.vhd
vcom -93 -work ./work ./v1/src/P4_2/sum_generator.vhd
vcom -93 -work ./work ./v1/src/utils/andgate2.vhd
vcom -93 -work ./work ./v1/src/utils/comparator.vhd
vcom -93 -work ./work ./v1/src/utils/fa.vhd
vcom -93 -work ./work ./v1/src/utils/mux21_generic.vhd
vcom -93 -work ./work ./v1/src/utils/notgate.vhd
vcom -93 -work ./work ./v1/src/utils/orgate2.vhd
vcom -93 -work ./work ./v1/src/utils/rca_gen.vhd
vcom -93 -work ./work ./v1/src/utils/RegisterN.vhd
vcom -93 -work ./work ./v1/src/inccomp.vhd

vcom -93 -work ./work ./vREF/src/utils/constants.vhd
vcom -93 -work ./work ./vREF/src/P4_1/carry_generator.vhd
vcom -93 -work ./work ./vREF/src/P4_1/carry_logic_network.vhd
vcom -93 -work ./work ./vREF/src/P4_1/carry_select_block.vhd
vcom -93 -work ./work ./vREF/src/P4_1/g_block.vhd
vcom -93 -work ./work ./vREF/src/P4_1/p4_adder_1.vhd
vcom -93 -work ./work ./vREF/src/P4_1/pg_block.vhd
vcom -93 -work ./work ./vREF/src/P4_1/pg_network.vhd
vcom -93 -work ./work ./vREF/src/P4_1/pg_network_block.vhd
vcom -93 -work ./work ./vREF/src/P4_1/sum_generator.vhd
vcom -93 -work ./work ./vREF/src/P4_2/carry_generator.vhd
vcom -93 -work ./work ./vREF/src/P4_2/carry_logic_network.vhd
vcom -93 -work ./work ./vREF/src/P4_2/carry_select_block.vhd
vcom -93 -work ./work ./vREF/src/P4_2/g_block.vhd
vcom -93 -work ./work ./vREF/src/P4_2/p4_adder_2.vhd
vcom -93 -work ./work ./vREF/src/P4_2/pg_block.vhd
vcom -93 -work ./work ./vREF/src/P4_2/pg_network.vhd
vcom -93 -work ./work ./vREF/src/P4_2/pg_network_block.vhd
vcom -93 -work ./work ./vREF/src/P4_2/sum_generator.vhd
vcom -93 -work ./work ./vREF/src/utils/andgate2.vhd
vcom -93 -work ./work ./vREF/src/utils/comparator.vhd
vcom -93 -work ./work ./vREF/src/utils/fa.vhd
vcom -93 -work ./work ./vREF/src/utils/mux21_generic.vhd
vcom -93 -work ./work ./vREF/src/utils/notgate.vhd
vcom -93 -work ./work ./vREF/src/utils/orgate2.vhd
vcom -93 -work ./work ./vREF/src/utils/rca_gen.vhd
vcom -93 -work ./work ./vREF/src/utils/RegisterN.vhd
vcom -93 -work ./work ./vREF/src/inccomp.vhd

vcom -93 -work ./work tb_inccomp.vhd

vsim -t ns -novopt work.inccomp_tb

set NumericStdNoWarnings 1
run 0 ps
set NumericStdNoWarnings 0

add wave -radix decimal *
run 100 us
```

Listing 8: File .do per la simulazione del comparatore con incrementatore

```
library ieee ;
use ieee.std_logic_1164.all ;
use ieee.numeric_std.all ;

entity inccomp_tb is
  GENERIC (
            N_BIT   : integer := 32
          );
end entity ;

architecture arch of inccomp_tb is

component inccomp
  generic  (
     N_BIT : integer := 32
   );
  port(
    CK    : in   std_logic;
    RST    : in   std_logic;
    INCA  : in   std_logic;
    INCB  : in   std_logic;
    EXTA   : in   std_logic;
    EXTB   : in   std_logic;
    A     : in   signed (N_BIT-1 downto 0);
    B     : in   signed (N_BIT-1 downto 0);
    C     : out  signed (N_BIT-1 downto 0)
  );
end component;

--REF
component inccomp_2
  generic  (
     N_BIT : integer := 32
   );
  port(
    CK    : in   std_logic;
    RST    : in   std_logic;
    INCA  : in   std_logic;
    INCB  : in   std_logic;
    EXTA   : in   std_logic;
    EXTB   : in   std_logic;
    A     : in   signed (N_BIT-1 downto 0);
    B     : in   signed (N_BIT-1 downto 0);
    C     : out  signed (N_BIT-1 downto 0)
  );
end component;

signal CK :  std_logic:='0';
signal RST: std_logic:='0';
signal INCA,INCB : std_logic;
signal EXTA,EXTB : std_logic;
signal A,B : signed (N_BIT-1 downto 0):=(others=>'0');
signal C,C_2 : signed (N_BIT-1 downto 0);

begin

  --clock generation
  CK <= not CK after 2 ns;
  RST <= '1' after 3 ns, '0' after 6 ns;

  --ADDER 1 INC, ADDER 2 NULL
  INCA <= '1';
  INCB <= '0';
  EXTA <= '0'; --incremental mode
  EXTB <= '1';
  --A <= (others => '0');
  B_generation:process(CK,RST)
    begin
      if RST = '1' then
        B <= (others => '0');
      elsif CK='1' and CK'event then
        B <= B + 1;
      end if;
  end process B_generation;

  bug_report: process(CK)
    begin
      if CK='1' AND CK'EVENT then
        assert ( C = C_2) report "There is a bug." severity warning;
      end if;
  end process bug_report;

  DUT : inccomp
    generic map (
      N_BIT => N_BIT)
    port map    (
      CK    =>  CK,
      RST    =>  RST,
      INCA  =>  INCA,
      INCB  =>  INCB,
      EXTA   =>  EXTA,
      EXTB   =>  EXTB,
      A     =>  A,
      B     =>  B,
      C     =>  C);

  REF : inccomp_2
    generic map (
      N_BIT => N_BIT)
    port map    (
      CK    =>  CK,
      RST    =>  RST,
      INCA  =>  INCA,
      INCB  =>  INCB,
      EXTA   =>  EXTA,
      EXTB   =>  EXTB,
      A     =>  A,
      B     =>  B,
      C     =>  C_2);

end architecture ;
```

Listing 9: Testbench per il comparatore con incrementatore

```
vcom -93 -work ./work ./v3/counter.vhd
vcom -93 -work ./work ./vREF/counter_ref.vhd
vcom -93 -work ./work tb_cnt.vhd
vsim -t ns -novopt work.tb_cnt
set NumericStdNoWarnings 1
run 0 ps
set NumericStdNoWarnings 0
add wave -radix decimal *
run 100 ns
```

Listing 10: File .do per la simulazione del contatore

```
library ieee ;
use ieee.std_logic_1164.all ;
use ieee.numeric_std.all ;
use ieee.math_real.all ; -- for UNIFORM, TRUNC

entity tb_cnt is
end entity ;

architecture behaviour of tb_cnt is

component counter
  port
  (
    CLK         : in  std_logic;
    RST         : in   std_logic;
    COUNT       : in   std_logic;
    DATA_OUT  : out  unsigned (8-1 downto 0);
    UP_DN       : out  std_logic
  );
end component;

--REF
component counter_ref
  port
  (
    CLK         : in  std_logic;
    RST         : in   std_logic;
    COUNT       : in   std_logic;
    DATA_OUT  : out  unsigned (8-1 downto 0);
    UP_DN       : out  std_logic
  );
end component;

signal CLK :  std_logic:='0';
signal RST: std_logic:='0';
signal COUNT: std_logic:='0';
signal UP_DN_DUT,UP_DN_REF : std_logic;
signal OUT_DUT, OUT_REF : unsigned (8-1 downto 0):=(others=>'0');
signal RST_t, COUNT_t : std_logic_vector(1 downto 0);

begin

  --cloCLK and reset generation
  CLK <= not CLK after 1 ns;
  --RST <= '1' after 3 ns, '0' after 6 ns,'1' after 30 ns, '0' after 31 ns; -- v2 test
  --COUNT <= '1';
  RST <= '1' after 3 ns, '0' after 6 ns; -- v3 test
  COUNT <= '1' after 3 ns, '0' after 14 ns, '1' after 70 ns;
  random: process (CLK)
    variable seed1, seed2: positive;

    variable rand_1: real;
    variable rand_2: real;

    variable int_rand_1: integer;
    variable int_rand_2: integer;

  begin
    if CLK='1' AND CLK'EVENT then
      -- Random number generation
      UNIFORM(seed1, seed2, rand_1);
      UNIFORM(seed1, seed2, rand_2);

      int_rand_1 := INTEGER(ROUND(rand_1));
      int_rand_2 := INTEGER(ROUND(rand_2));
      RST_t <= std_logic_vector(to_signed(int_rand_1,2));
      COUNT_t <= std_logic_vector(to_signed(int_rand_2,2));
      --RST <= RST_t(0);
      --COUNT <= COUNT_t(0);
    end if;

  end process random;

  bug_report: process(CLK)
    begin
      if CLK='1' AND CLK'EVENT then
        assert ( OUT_DUT = OUT_REF) report "There is a bug [OUTPUT MISMATCH]" severity warning;
      end if;
      if CLK='1' AND CLK'EVENT then
        assert ( UP_DN_DUT = UP_DN_REF) report "There is a bug [UP/DOWN MISMATCH]" severity warning;
      end if;
  end process bug_report;

  DUT : counter
    port map    (
      CLK     =>  CLK,
      RST     =>  RST,
      COUNT   =>   COUNT,
      DATA_OUT=>  OUT_DUT,
      UP_DN   =>  UP_DN_DUT);

  REF : counter_ref
    port map    (
      CLK     =>  CLK,
      RST     =>  RST,
      COUNT   =>   COUNT,
      DATA_OUT=>  OUT_REF,
      UP_DN   =>  UP_DN_REF);

end architecture behaviour;
```

Listing 11: Testbench per il comparatore con incrementatore