

Appunti di Architettura degli Elaboratori

Alessandro Cheli - Prof. Marco Danelutto

A.A 2019-2020

Indice

1	Introduzione al Corso	1
2	Rappresentazioni Numeriche e Testuali	5
2.1	Aritmetica Binaria	5
2.2	Esadecimale	5
2.3	Numeri in virgola mobile	6
2.4	Codifica ASCII	7
2.5	Porte Logiche e Algebra di Boole	7
2.5.1	Algebra di Boole	11
2.5.2	Teoremi dell'Algebra Booleana	12
2.5.3	Mappe di Karnaugh	13
2.5.4	Circuito con più output	13
2.5.5	Operatori a più ingressi	13
3	Reti Logiche e Combinatorie	15
3.1	Alcuni componenti standard	15
3.2	Reti sequenziali e Automi	23
3.2.1	Realizzazione di una rete sequenziale di un Automa di Mealy con un clock e registri	25
3.2.2	Automa che riconosce "abba"	26
4	Verilog e RTL	29
4.1	Scrivere e compilare System Verilog	30
4.2	Esercizi	32

Capitolo 1

Introduzione al Corso

- Logica Booleana
- Aritmetica Binaria
- Reti Logiche
- Microarchitettura e Assembler ARM v7 e v8
- Gestione della memoria
- I/O

Strumenti Software A differenza degli A.A passati utilizzeremo Verilog e Assembler ARM. Utilizzeremo **iverilog** come compilatore Verilog e **gtkwave** come tool grafico. Un ambiente di sviluppo Verilog completo che vedremo è **Quartus**. Per la seconda parte del corso, Assembler ARM, useremo la **toolchain GNU**, in particolare:

Se non hai una macchina ARM:

- cross-compiler per compilare
- QEMU per una macchina virtuale ARM
- gdb per debugging

Se hai una macchina ARM come un Raspberry Pi:

- Toolchain GNU per compilare
- Cavo Ethernet
- Server SSH sulla macchina ARM per accesso remoto

Storia degli Elaboratori Nei corsi di Architettura degli Elaboratori negli anni 80 i processori studiati erano: il 6502 (8 bit, processore del computer Apple II, noto per essere stato costruito nel garage di Steve Jobs e Wozniak), Z80, processore a 16 bit del famoso computer ZX80 e l'Intel 8088. Tali processori raggiungevano al massimo una velocità di clock (detto molto a grandi linee, operazioni al secondo) dell'ordine di meno di una decina di MHz (Mega Hertz, milioni). I processori odierni raggiungono cicli di clock sull'ordine dei GHz (Giga Hertz, miliardi). Nel corso degli anni fino ad oggi, l'evoluzione dei processori ha seguito la **legge di Moore**. La "legge" spiega

Figura 1.1: Sinclair ZX80



che ogni 18 mesi la potenza dei processori in commercio raddoppia, perché la densità dei transistor contenuti all'interno aumenta. Negli ultimi decenni abbiamo miglioramenti architetturali come super pipeline e super scalari, ciò ha permesso di introdurre processori **multicore**, ovvero che contengono più "nuclei" interni (detti core) che elaborano le istruzioni dei processi in esecuzione in parallelo. Ad oggi il numero di core in uno smartphone raggiunge anche gli 8 core, mentre in processori per server sono stati raggiunti numeri di core anche intorno ai 64. I processori odierni utilizzano core a 64 bit, con architettura X86_64 per Desktop o ARM per dispositivi mobili. Un componente fondamentale dell'evoluzione degli elaboratori è stato anche lo sviluppo dei processori grafici (GPU) con i quali ad oggi è possibile riprodurre grafica su schermo, ambienti tridimensionali molto complessi (videogiochi) o sfruttare la loro capacità di parallelizzazione per l'uso di reti neurali nell'intelligenza artificiale.

Osserveremo i calcolatori a diversi livelli di **astrazione**

I livelli di astrazione sono:

- Applicazioni utente
- Sistema Operativo
- Architettura (ASM, ad es. x86 o ARM)
- Microarchitettura
- Logica
- Circuiti digitali
- Device
- Fisica

Ogni livello si appoggia sul livello inferiore, ovvero è costruito sui componenti offerti dal livello inferiore. Dei principi fondamentali sono: **gerarchi, modularità e regolarità**

La modularità è fondamentale per avere moduli organizzati gerarchicamente, autonomi ed indipendenti.

Set di Istruzioni Distinguiamo due set di istruzioni dei processori, **CISC** e **RISC**. Gli acronimi sono rispettivamente **Complex Instruction Set Computer** e **Reduced Instruction Set Computer**, RISC contiene i processori ARM, che studieremo in dettaglio, mentre CISC comprende i processori più comuni nei desktop (X86 e X86_64)

Capitolo 2

Rappresentazioni Numeriche e Testuali

2.1 Aritmetica Binaria

I calcolatori utilizzano valori discreti (differenze di potenziale) fra 0 e 1 per rappresentare valori numerici. Viene detta Aritmetica Binaria l'aritmetica con i numeri rappresentati in base 2.

Siamo abituati a ragionare in base 10, ad esempio il numero 413 in base 10 è

$$104 = 10^2 \cdot 1 + 10^1 \cdot 0 + 10^0 \cdot 4$$

Lo stesso numero rappresentato in base 2 (codice binario) è

$$104_{10} = 01101000_2 = 2^7 \cdot 0 + 2^6 \cdot 1 + 2^5 \cdot 1 + 2^4 \cdot 0 + 2^3 \cdot 1 + 2^2 \cdot 0 + 2^1 \cdot 0 + 2^0 \cdot 0$$

Un numero binario di 8 cifre è detto **byte**, un numero di 4 cifre è detto **nibble**. Una **parola** (**word**) è la quantità minima su cui viene rappresentato un intero in un calcolatore. Ad oggi le parole dei calcolatori sono 64 bit, alcuni calcolatori datati hanno parole da 32 bit.

La somma nell'aritmetica binaria è definita normalmente per i numeri positivi. Nei calcolatori i numeri hanno una dimensione finita (numero di bit) che indica il numero di cifre binarie con le quali è possibile rappresentare un numero. I positivi binari rappresentano numeri fino a $2^N - 1$ dove N è il numero di cifre.

Per rappresentare i numeri negativi si utilizza il metodo **segno-magnitudo** dove il bit più a sinistra rappresenta il segno (0 se il numero è positivo e 1 se è negativo). Il problema del metodo segno-magnitudo è che non rispetta la somma aritmetica. Può rappresentare numeri da $[-2^{N-1}, +2^{N-1}]$

Un metodo migliore per rappresentare i numeri negativi è il **complemento a due**. Nel complemento a due la cifra più a sinistra rappresenta sempre 2^{N-1} ma **negativo**. Il resto delle cifre sono positive e vengono sommate alla prima cifra negativa. Questo metodo rispetta la somma aritmetica. Per moltiplicare un numero per -1 si invertono le cifre binarie e si aggiunge 1 al numero. È possibile anche la sottrazione sommando un numero positivo ad uno negativo.

La somma fra due cifre può essere costruita con reti logiche. Il risultato della somma $A + B = A \oplus B$ (operatore XOR) mentre il riporto della somma $= A \wedge B$ (operatore AND)

2.2 Esadecimale

I numeri esadecimali sono numeri in base 16. Siccome non bastano le cifre decimali per rappresentare i numeri maggiori di 9 si usano le prime lettere dell'alfabeto. Una cifra esadecimale rappresenta un nibble (4 bit).

Figura 2.1: Gate XOR

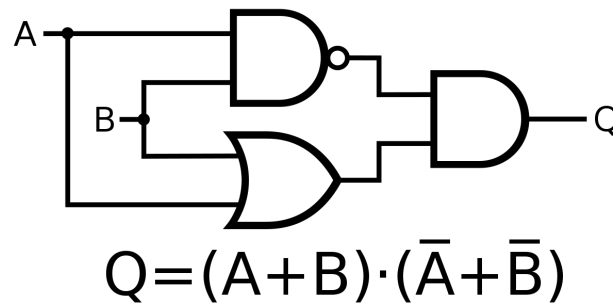
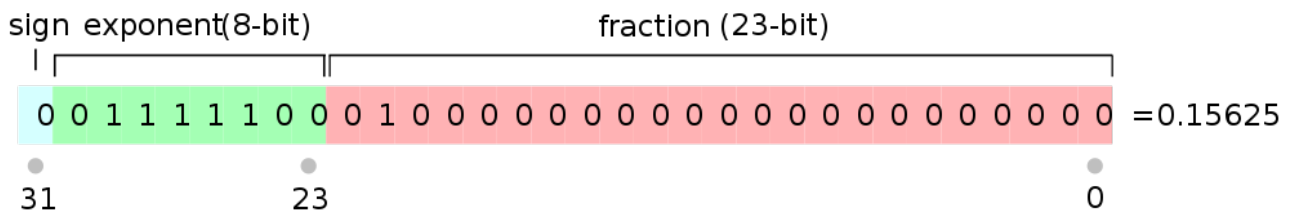


Figura 2.2: Standard IEEE 754 a 32 bit



2.3 Numeri in virgola mobile

I numeri in virgola mobile si rappresentano con lo standard IEEE 754 che definisce come si rappresentano i numeri in virgola mobile a singola precisione e doppia precisione (32 e 64 bit)

I bit del numero vengono divisi in 3 parti. Il primo bit denota il segno, la seconda parte rappresenta l'esponente e la terza parte si denota mantissa. L'esponente esprime dove la virgola verrà posizionata, come nella notazione scientifica di una calcolatrice l'esponente rappresenta 10^n dove n è l'esponente. La mantissa è un numero di base moltiplicato per 10^0 , e viene successivamente moltiplicato per l'esponente. L'esponente può essere sia positivo che negativo.

Nello standard a 32 bit la sezione esponente ha 8 bit di lunghezza. Un numero ad 8 bit può rappresentare numeri da 0 a 255, per ottenere gli esponenti negativi nello standard dei numeri a virgola mobile il numero a 8 bit rappresenta invece numeri da -127 a +128

Somma dei numeri a virgola mobile Per sommare i numeri a virgola mobile il primo passo è allineare le mantisse, significa osservare gli esponenti e spostarli fino a che le cifre non sono sommabili in colonna. Il secondo passo consiste nel sommare e il terzo passo nel normalizzare la somma. Nei processori la somma floating point viene eseguita in dei moduli appositi che in input ricevono due o più numeri floating point ed eseguono in dei sotto-moduli i tre passaggi della somma in un tempo $1/3t$ dove t è il tempo totale per eseguire una somma. I tre passaggi della

Figura 2.3: IEEE 754 a 64 bit

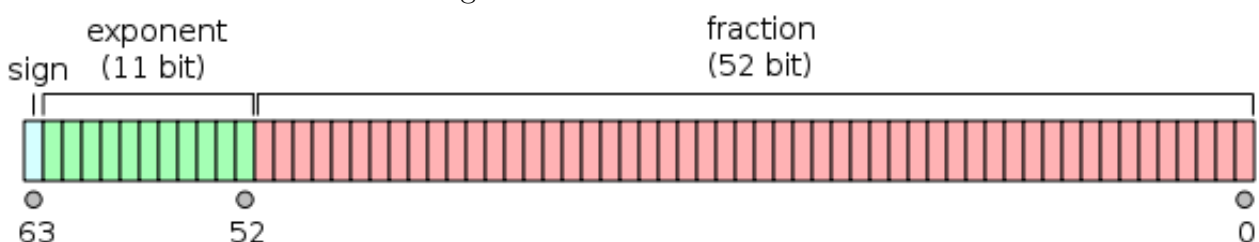


Figura 2.4: Tabella ASCII

ASCII TABLE

Decimal	Hex	Char	Decimal	Hex	Char	Decimal	Hex	Char	Decimal	Hex	Char
0	0	[NULL]	32	20	[SPACE]	64	40	@	96	60	`
1	1	[START OF HEADING]	33	21	!	65	41	A	97	61	a
2	2	[START OF TEXT]	34	22	"	66	42	B	98	62	b
3	3	[END OF TEXT]	35	23	#	67	43	C	99	63	c
4	4	[END OF TRANSMISSION]	36	24	\$	68	44	D	100	64	d
5	5	[ENQUIRY]	37	25	%	69	45	E	101	65	e
6	6	[ACKNOWLEDGE]	38	26	&	70	46	F	102	66	f
7	7	[BELL]	39	27	'	71	47	G	103	67	g
8	8	[BACKSPACE]	40	28	(72	48	H	104	68	h
9	9	[HORIZONTAL TAB]	41	29)	73	49	I	105	69	i
10	A	[LINE FEED]	42	2A	*	74	4A	J	106	6A	j
11	B	[VERTICAL TAB]	43	2B	+	75	4B	K	107	6B	k
12	C	[FORM FEED]	44	2C	,	76	4C	L	108	6C	l
13	D	[CARRIAGE RETURN]	45	2D	-	77	4D	M	109	6D	m
14	E	[SHIFT OUT]	46	2E	.	78	4E	N	110	6E	n
15	F	[SHIFT IN]	47	2F	/	79	4F	O	111	6F	o
16	10	[DATA LINK ESCAPE]	48	30	0	80	50	P	112	70	p
17	11	[DEVICE CONTROL 1]	49	31	1	81	51	Q	113	71	q
18	12	[DEVICE CONTROL 2]	50	32	2	82	52	R	114	72	r
19	13	[DEVICE CONTROL 3]	51	33	3	83	53	S	115	73	s
20	14	[DEVICE CONTROL 4]	52	34	4	84	54	T	116	74	t
21	15	[NEGATIVE ACKNOWLEDGE]	53	35	5	85	55	U	117	75	u
22	16	[SYNCHRONOUS IDLE]	54	36	6	86	56	V	118	76	v
23	17	[ENG OF TRANS. BLOCK]	55	37	7	87	57	W	119	77	w
24	18	[CANCEL]	56	38	8	88	58	X	120	78	x
25	19	[END OF MEDIUM]	57	39	9	89	59	Y	121	79	y
26	1A	[SUBSTITUTE]	58	3A	:	90	5A	Z	122	7A	z
27	1B	[ESCAPE]	59	3B	;	91	5B	[123	7B	{
28	1C	[FILE SEPARATOR]	60	3C	<	92	5C	\	124	7C	
29	1D	[GROUP SEPARATOR]	61	3D	=	93	5D]	125	7D	}
30	1E	[RECORD SEPARATOR]	62	3E	>	94	5E	^	126	7E	~
31	1F	[UNIT SEPARATOR]	63	3F	?	95	5F	_	127	7F	[DEL]

somma possono essere sequenzializzati così che una volta che ogni sotto-modulo ha completato il passo, può ricevere subito l'input successivo (la somma di due numeri FP impiegherà $t + 1/3$ invece che $2t$)

Estensioni vettoriali Alcuni processori permettono di eseguire operazioni contemporaneamente su un registro dividendolo in sottoregistri più piccoli.

2.4 Codifica ASCII

La codifica ASCII è una tabella di codifica di caratteri testuali con interi da 0 a 255 (8 bit). La codifica ASCII estesa è a 16 bit e comprende diversi caratteri non latini.

2.5 Porte Logiche e Algebra di Boole

I circuiti digitali vengono realizzati utilizzando componenti chiamati **porte logiche**. Sono realizzate con componenti fisici come transistor e resistenze, ma nella progettazione dei circuiti digitali le porte logiche vengono schematizzate con i simboli riportati nella Figura 3.1 per semplificare la progettazione **astruendo** il livello di complessità della circuiteria analogica. Solamente con la porta NAND si possono realizzare tutte le altre porte (NAND è funzionalmente completo), ma le porte in generale si costruiscono singolarmente con componenti appositi. Esse implementano la **logica booleana** che conseguentemente permette di realizzare operazioni di **aritmetica binaria** per costruire unità di calcolo in componenti elettronici e processori.

I componenti elettronici molto piccoli sono sensibili al **rumore**, per ovviare al problema i valori discreti (0 e 1) nei circuiti digitali non seguono un cambiamento istantaneo di differenza di potenziale (voltage), ma ammettono un margine per ridurre i problemi causati dal rumore.

I componenti (transistor) con cui si costruiscono porte logiche e circuiti sono realizzati con materiali semiconduttori, che possono essere di diversi tipi. Vedremo il tipo NMOS. Un transistor è composto da materiali come gallio e silicio.

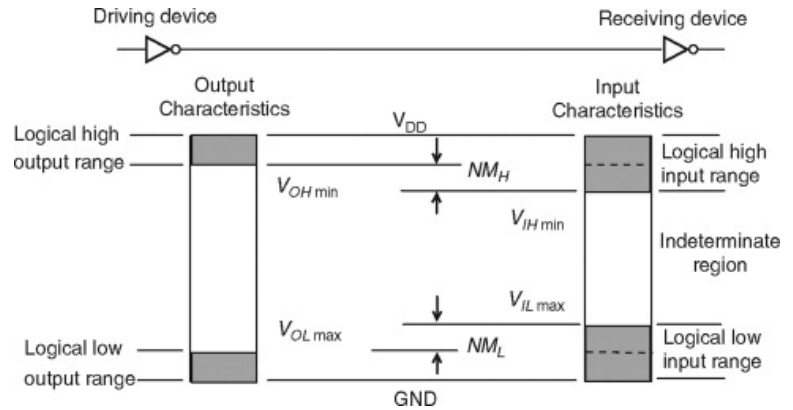


Figura 2.5: Margine di rumore nei circuiti digitali

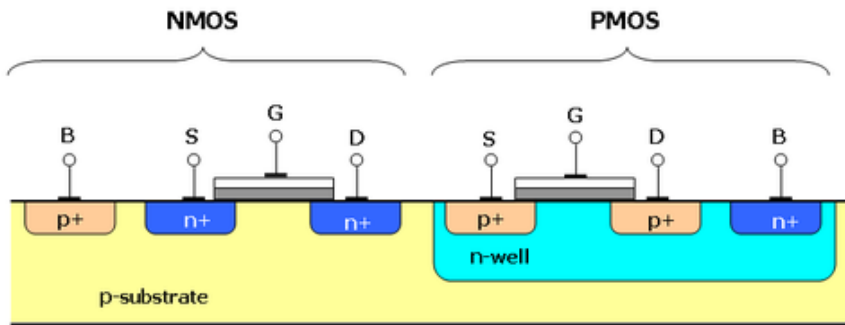


Figura 2.6: Transistor NMOS

Figura 2.7: Tabella delle porte logiche comuni

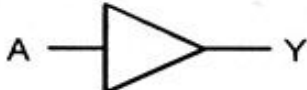
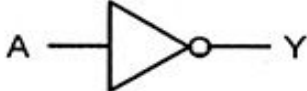






Logic function	Logic symbol	Truth table	Boolean expression															
Buffer		<table border="1" data-bbox="903 418 1067 542"><tr><td>A</td><td>Y</td></tr><tr><td>0</td><td>0</td></tr><tr><td>1</td><td>1</td></tr></table>	A	Y	0	0	1	1	$Y = A$									
A	Y																	
0	0																	
1	1																	
Inverter (NOT gate)		<table border="1" data-bbox="903 598 1067 721"><tr><td>A</td><td>Y</td></tr><tr><td>0</td><td>1</td></tr><tr><td>1</td><td>0</td></tr></table>	A	Y	0	1	1	0	$Y = \bar{A}$									
A	Y																	
0	1																	
1	0																	
2-input AND gate		<table border="1" data-bbox="871 766 1096 945"><tr><td>A</td><td>B</td><td>Y</td></tr><tr><td>0</td><td>0</td><td>0</td></tr><tr><td>0</td><td>1</td><td>0</td></tr><tr><td>1</td><td>0</td><td>0</td></tr><tr><td>1</td><td>1</td><td>1</td></tr></table>	A	B	Y	0	0	0	0	1	0	1	0	0	1	1	1	$Y = A \bullet B$
A	B	Y																
0	0	0																
0	1	0																
1	0	0																
1	1	1																
2-input NAND gate		<table border="1" data-bbox="871 990 1096 1169"><tr><td>A</td><td>B</td><td>Y</td></tr><tr><td>0</td><td>0</td><td>1</td></tr><tr><td>0</td><td>1</td><td>1</td></tr><tr><td>1</td><td>0</td><td>1</td></tr><tr><td>1</td><td>1</td><td>0</td></tr></table>	A	B	Y	0	0	1	0	1	1	1	0	1	1	1	0	$Y = \overline{A \bullet B}$
A	B	Y																
0	0	1																
0	1	1																
1	0	1																
1	1	0																
2-input OR gate		<table border="1" data-bbox="871 1214 1096 1393"><tr><td>A</td><td>B</td><td>Y</td></tr><tr><td>0</td><td>0</td><td>0</td></tr><tr><td>0</td><td>1</td><td>1</td></tr><tr><td>1</td><td>0</td><td>1</td></tr><tr><td>1</td><td>1</td><td>1</td></tr></table>	A	B	Y	0	0	0	0	1	1	1	0	1	1	1	1	$Y = A + B$
A	B	Y																
0	0	0																
0	1	1																
1	0	1																
1	1	1																
2-input NOR gate		<table border="1" data-bbox="871 1438 1096 1617"><tr><td>A</td><td>B</td><td>Y</td></tr><tr><td>0</td><td>0</td><td>1</td></tr><tr><td>0</td><td>1</td><td>0</td></tr><tr><td>1</td><td>0</td><td>0</td></tr><tr><td>1</td><td>1</td><td>0</td></tr></table>	A	B	Y	0	0	1	0	1	0	1	0	0	1	1	0	$Y = \overline{A + B}$
A	B	Y																
0	0	1																
0	1	0																
1	0	0																
1	1	0																
2-input EX-OR gate		<table border="1" data-bbox="871 1662 1096 1841"><tr><td>A</td><td>B</td><td>Y</td></tr><tr><td>0</td><td>0</td><td>0</td></tr><tr><td>0</td><td>1</td><td>1</td></tr><tr><td>1</td><td>0</td><td>1</td></tr><tr><td>1</td><td>1</td><td>0</td></tr></table>	A	B	Y	0	0	0	0	1	1	1	0	1	1	1	0	$Y = A \oplus B$
A	B	Y																
0	0	0																
0	1	1																
1	0	1																
1	1	0																
2-input EX-NOR gate		<table border="1" data-bbox="871 1886 1096 2065"><tr><td>A</td><td>B</td><td>Y</td></tr><tr><td>0</td><td>0</td><td>1</td></tr><tr><td>0</td><td>1</td><td>0</td></tr><tr><td>1</td><td>0</td><td>0</td></tr><tr><td>1</td><td>1</td><td>1</td></tr></table>	A	B	Y	0	0	1	0	1	0	1	0	0	1	1	1	$Y = \overline{A \oplus B}$
A	B	Y																
0	0	1																
0	1	0																
1	0	0																
1	1	1																

Figura 2.8: Porta NOT con transistor PMOS e NMOS

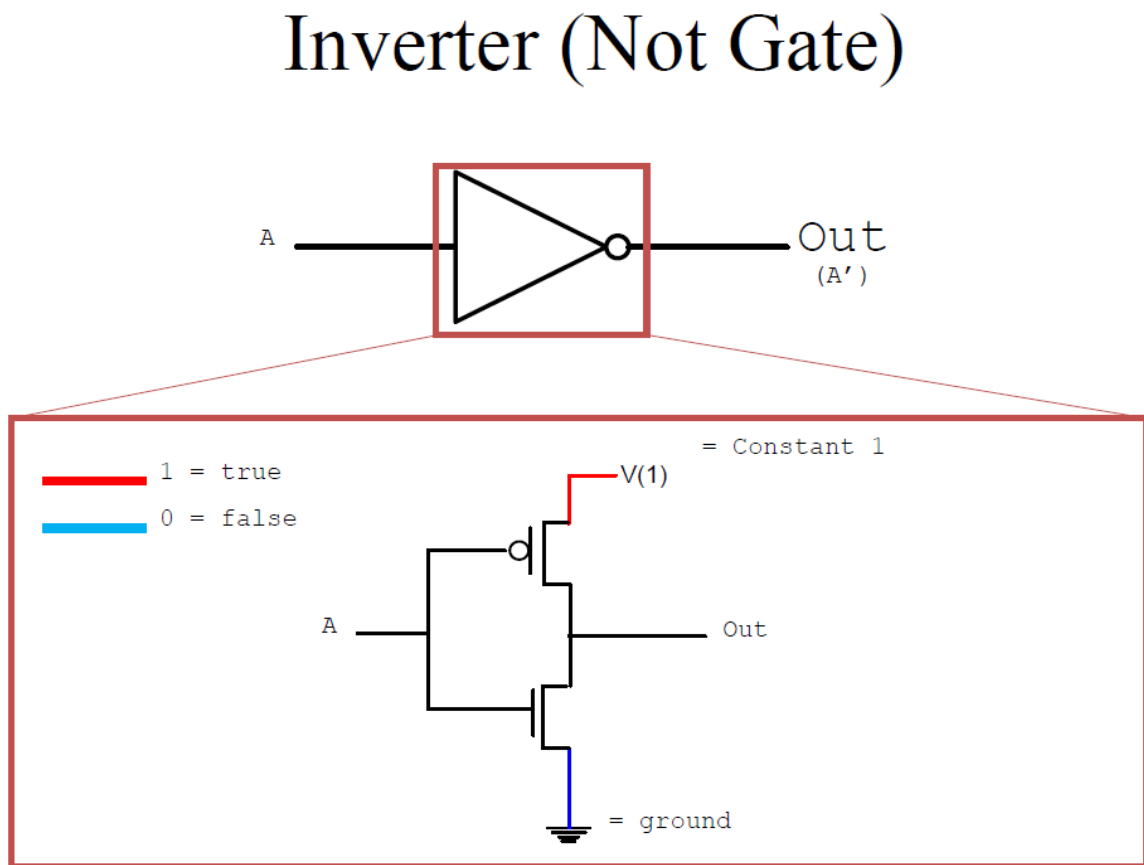
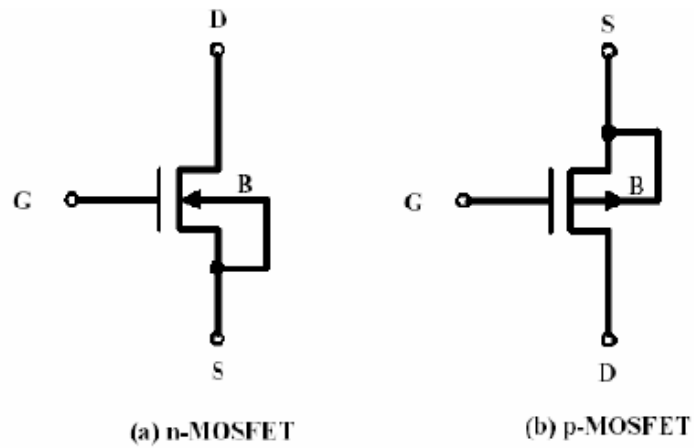


Figura 2.9: Transistor NMOS e PMOS



2.5.1 Algebra di Boole

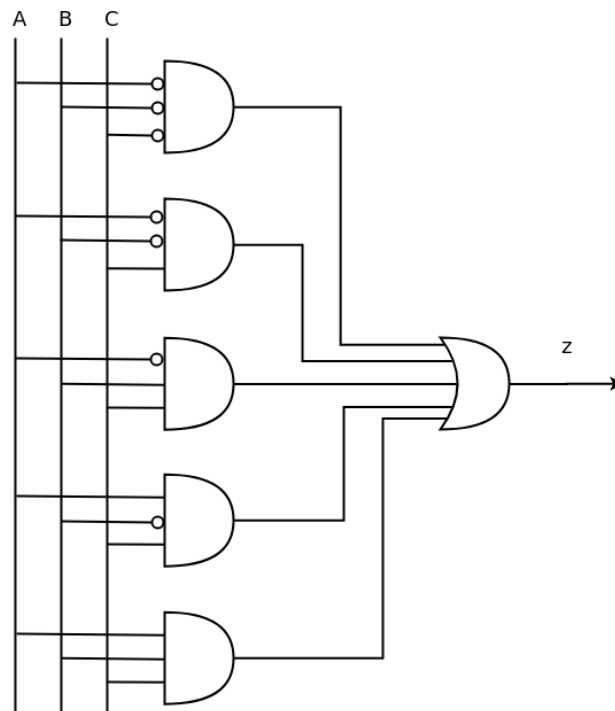


Figura 2.10: Conversione di z da formula booleana a circuito logico

Funzione	Notazione Usata	Notazione Logica
NOT(A)	\bar{A}	$\neg A$
AND(A,B)	$A \cdot B$	$A \wedge B$
OR(A,B)	$A + B$	$A \vee B$

Tabella 2.1: Notazione usata per l'Algebra di Boole

Prendiamo ad esempio un'espressione booleana in forma canonica in **somma di prodotti**:

$$z = \bar{A}\bar{B}\bar{C} + \bar{A}\bar{B}C + \bar{A}BC + A\bar{B}C + ABC$$

A	B	C	z
0	0	0	1
0	0	1	1
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	0
1	1	1	1

Tabella 2.2: Tabella di verità di z

z si può anche esprimere come prodotto di somme: $z = (A + \bar{B} + C)(\bar{A}BC)(\bar{A}\bar{B}C)$

2.5.2 Teoremi dell'Algebra Booleana

Breve ripasso dei teoremi della Logica Booleana.

Elemento Identità di prodotto e somma

$$A \cdot 1 = A \iff A \wedge T \equiv A$$

$$A + 0 = A \iff A \vee F \equiv A$$

Elemento assorbente

$$A \cdot 0 = 0 \iff A \wedge F \equiv F$$

$$A + 1 = 1 \iff A \vee T \equiv T$$

Idempotenza

$$A \cdot A = A \iff A \wedge A \equiv A$$

$$A + A = A \iff A \vee A \equiv A$$

Complemento

$$A \cdot \bar{A} = 0 \iff A \wedge \neg A \equiv F$$

$$A + \bar{A} = 1 \iff A \vee \neg A \equiv T$$

Commutatività

$$A + B = B + A$$

$$A \cdot B = B \cdot A$$

Associatività

$$(A \cdot B) \cdot C = A \cdot (B \cdot C)$$

$$(A + B) + C = A + (B + C)$$

Distributività

$$(A \cdot B) + C = (A + C) \cdot (B + C)$$

$$(A + B) \cdot C = AC + BC$$

DeMorgan

$$\overline{(A + B)} = \bar{A} \cdot \bar{B} \iff \neg(A \vee B) \equiv \neg A \wedge \neg B$$

$$\overline{(A \cdot B)} = \bar{A} + \bar{B} \iff \neg(A \wedge B) \equiv \neg A \vee \neg B$$

Esempio Semplifichiamo la formula booleana $z = \bar{A}\bar{B}\bar{C} + \bar{A}\bar{B}C + \bar{A}BC + A\bar{B}C + ABC$

$$\begin{cases} \bar{A}\bar{B}\bar{C} + \bar{A}\bar{B}C \equiv \bar{A}\bar{B}(\bar{C} + C) \equiv \bar{A}\bar{B} \\ \bar{A}\bar{B}C + A\bar{B}C \equiv \bar{B}C(\bar{A} + A) \equiv \bar{B}C \end{cases} \quad (2.1)$$

$$\implies z = \bar{A}\bar{B} + \bar{B}C + AC \quad (2.2)$$

Le leggi della logica Booleana ci permettono di semplificare molto i componenti realizzati con porte logiche.

2.5.3 Mappe di Karnaugh

Una mappa di Karnaugh o K-Map è un metodo di semplificare un'espressione booleana. I valori sono trasferiti da una tabella di verità ad una mappa bidimensionale:

Approfondisci su Wikipedia

Prendiamo una formula a quattro variabili $f(A, B, C, D)$, una mappa di Karnaugh può essere:

AB\CD	00	01	11	10
00	1	1	1	1
01	1	1	0	0
11	0	1	0	0
10	0	0	1	1

Tabella 2.3: Tabella di verità della mappa di Karnaugh di f

Facciamo ad esempio la mappa di Karnaugh di $z = f(A, B, C)$ vista nella sezione precedente:

A\BC	00	01	11	10
0	1	1	1	0
1	0	1	1	0

Tabella 2.4: Tabella di verità della mappa di Karnaugh di z

Possiamo riconoscere un'implicante nella seconda e terza colonna che corrisponde esattamente a C . Osserviamo un'altra implicante nella prima riga, prima e seconda colonna che corrisponde esattamente a $\overline{A}\overline{B}$. Possiamo poi sommare le implicanti per ottenere una formula equivalente a quella di partenza, ciò implica che $z = \overline{A}\overline{B} + C$

2.5.4 Circuito con più output

Se abbiamo una tabella di verità di un circuito con n input e 2^n righe, con più output z_1, \dots, z_k , gli output si suddividono in k tabelle con un solo output (z_k) e 2^n righe di input.

2.5.5 Operatori a più ingressi

Gli operatori a più ingressi AND, OR, etc..., se presentano più di due ingressi si rappresentano con una rete logica che sfrutta la proprietà associativa degli operatori logici. Ciò comporta un limite massimo di ingressi perché viene introdotto un ritardo di stabilizzazione determinato e piccolo. Ad esempio, un AND a 4 ingressi sarà rappresentato come $z = x_1 \cdot x_2 \cdot x_3 \cdot x_4 = (x_1 \cdot x_2) \cdot (x_3 \cdot x_4)$. Perciò gli operatori associativi a più ingressi si rappresentano come un albero k-ario di porte logiche. Il numero di livelli di porte sarà $\log_k(n)$ dove k è l'arietà delle singole porte ed è n il numero di ingressi nel circuito.

Capitolo 3

Reti Logiche e Combinatorie

3.1 Alcuni componenti standard

Le tabelle di verità che vediamo si convertono in reti logiche. Una tabella di verità con n ingressi ha 2^n righe e corrisponde ad un componente logico di un circuito con n input. Vediamo alcuni oggetti utili. Ricordiamo che i passaggi per definire un componente in forma di rete logica sono:

1. Definizione della funzione con tabella di verità
2. Conversione della funzione normalizzata in somma di prodotti
3. Conversione a circuito con porte AND/OR/NOT

Multiplexer (k commutatore) Un multiplexer o commutatore è un circuito, ad esempio, con due 2 ingressi, con un ingresso aggiuntivo chiamato di *controllo* che permette di alternare quale sarà l'input che verrà copiato in output.

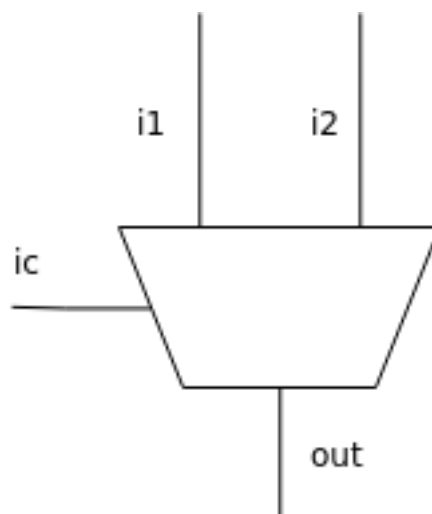


Figura 3.1: Multiplexer 2 vie 1 bit

ictrl	in_1	in_2	out
0	0	0	0
0	0	1	0
0	1	0	1
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	0
1	1	1	1

Tabella 3.1: Tabella di verità di un multiplexer a due vie da un bit

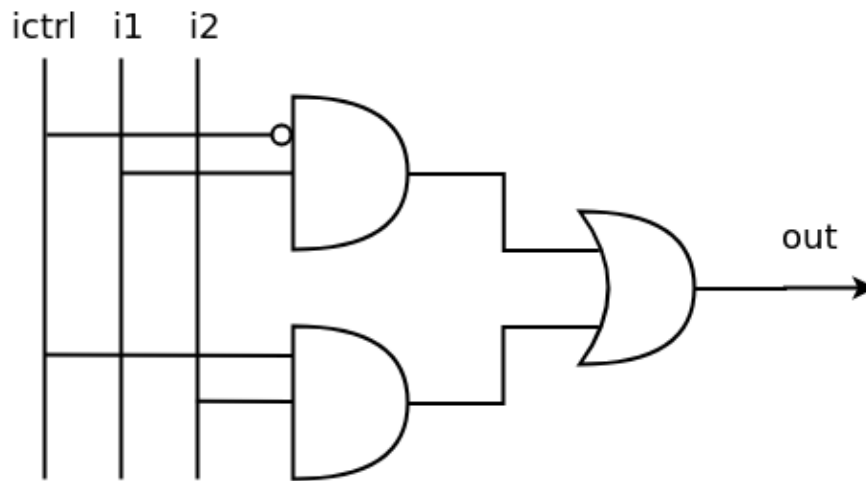


Figura 3.2: Circuito logico di un Multiplexer 2 vie 1 bit

La funzione è definita come $out = \bar{ic} \cdot in_1 \cdot \bar{in}_2 + \bar{ic} \cdot in_1 \cdot in_2 + ic \cdot \bar{in}_1 \cdot in_2 + ic \cdot in_1 \cdot in_2$ e si può ridurre in $out = \bar{ic} \cdot in_1 + ic \cdot in_2$. Le tabelle di verità semplificate ci permettono di dedurre direttamente la formula ridotta. Ad esempio, la tabella seguente corrisponde a quella antecedente.

ictrl	in1	in2	out
0	0	-	0
0	1	-	1
1	-	0	0
1	-	1	1

Tabella 3.2: Tabella di verità semplificata di un multiplexer a due vie da un bit

Multiplexer a 4 Input Un multiplexer a 4 input ha bisogno di due bit di controllo. Avendo 6 ingressi, con porte da 8 ingressi massimo $\Rightarrow \lceil \log_8(6) \rceil$ livelli di porte.

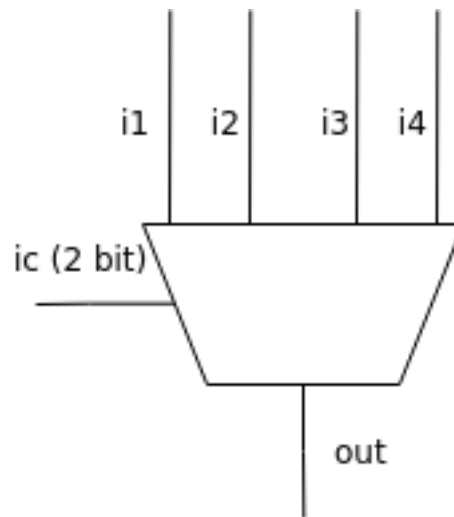


Figura 3.3: Multiplexer 4 vie 1 bit

ic_1	ic_2	in_1	in_2	in_3	in_4	out
0	0	1	-	-	-	1
0	1	-	1	-	-	1
1	0	-	-	1	-	1
1	1	-	-	-	1	1

Tabella 3.3: Tabella di verità semplificata di un multiplexer a quattro vie da un bit

La formula di verità corrispondente sarà

$$\text{out} = (\bar{ic}_1 \cdot \bar{ic}_2 \cdot in_1) + (\bar{ic}_1 \cdot ic_2 \cdot in_2) + (ic_1 \cdot \bar{ic}_2 \cdot in_3) + (ic_1 \cdot ic_2 \cdot in_4)$$

I livelli delle porte AND saranno $\log_2(n + 1)$

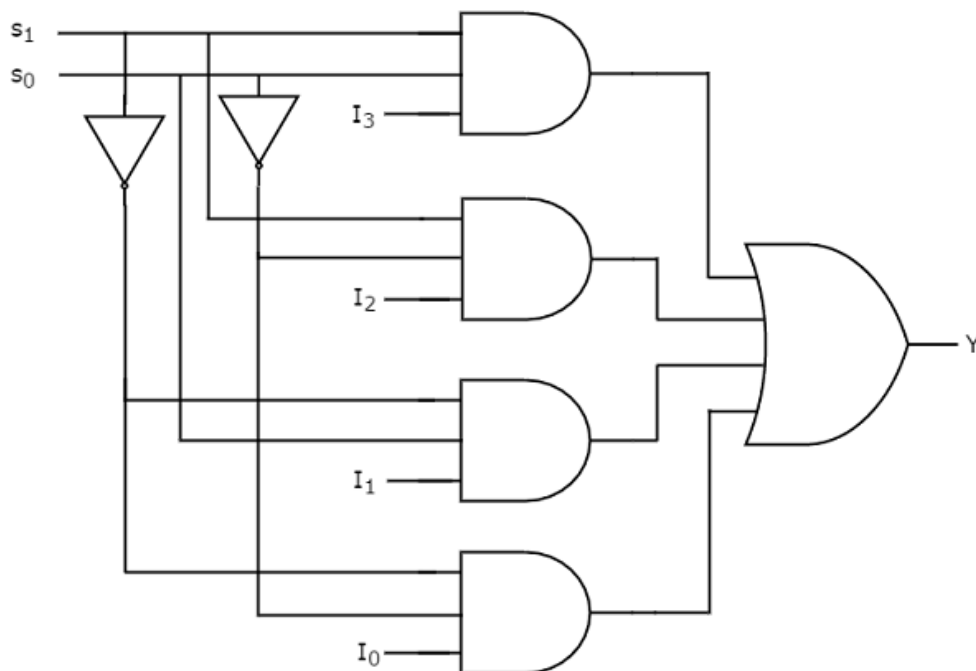


Figura 3.4: Circuito logico di un Multiplexer 4 vie 1 bit

Commutatori (multiplexer) composti Un commutatore multiplo da 1 bit con un numero di vie y , con $y = 2^x \wedge x \in \mathbb{N}$ e $y > 2$ si può costruire a partire da un albero con $\log_2 y$ livelli di multiplexer da 2 vie a 1 bit. Dove ogni bit di controllo del multiplexer complessivo controlla un livello singolo dell'albero.

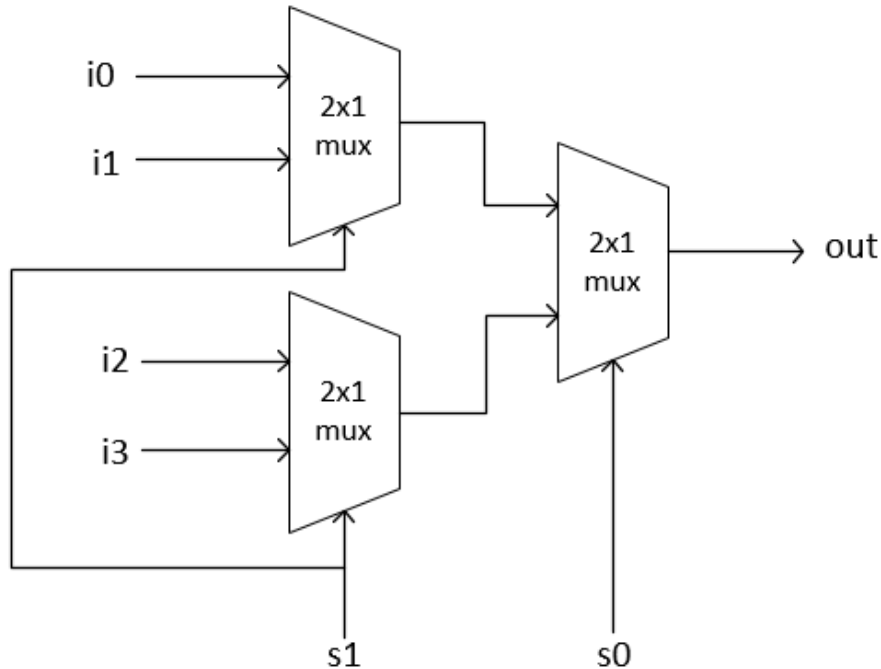


Figura 3.5: Multiplexer 4 vie 1 bit composto da due Multiplexer 2x1

Multiplexer a 2 vie da k bit Un commutatore a 2 vie da k bit si costruisce con k commutatori a 2 vie ad 1 bit. Dove ogni commutatore 2x1 accetta in input le corrispondenti vie di ogni bit, gli output saranno i k bit corrispondenti ad ogni Multiplexer.

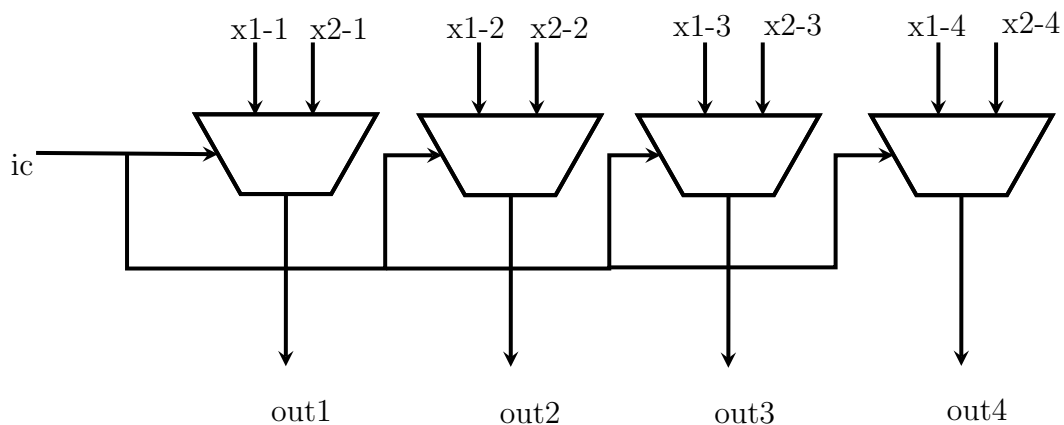


Figura 3.6: Multiplexer 2 vie da 4 bit

Sommatore di numeri Un sommatore è un componente che somma due numeri in input e restituisce in output un risultato ed un riporto.

x_1	x_2	r_0	z_1	r_1
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	1	0
1	0	0	1	0
1	1	0	0	1
1	0	1	0	1
1	1	1	1	1

Tabella 3.4: Tabella di verità di una somma di due numeri da un bit.

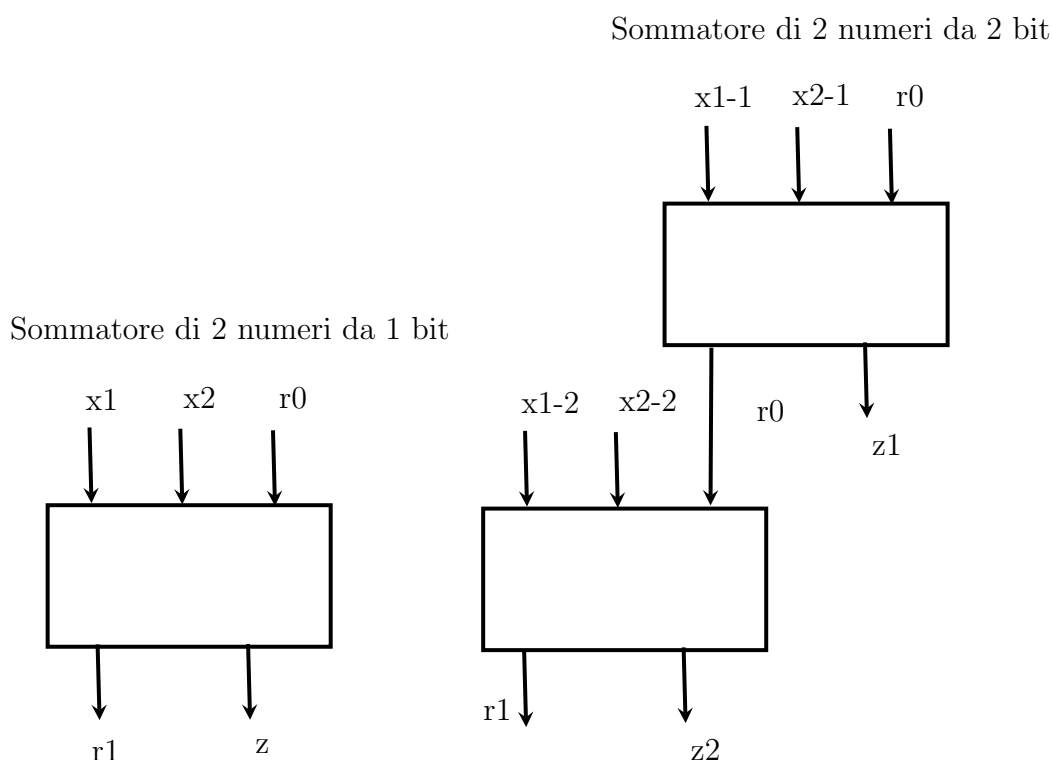


Figura 3.7: Sommatori di due numeri

Analogamente ai commutatori si possono costruire sommatore di 2 numeri a più bit a partire da sommatore di 2 numeri da 1 bit.

Esercizio Realizzare la tabella di verità e circuito di un demultiplexer a 2 vie da 1 bit.

Encoder Un encoder è un circuito che converte 2^n linee in input in un codice di n bit.

a	b	c	d	z	t
0	0	0	1	0	0
0	0	1	0	0	1
0	1	0	0	1	0
1	0	0	0	1	1

Tabella 3.5: Tabella di verità di encoder da 2 bit.

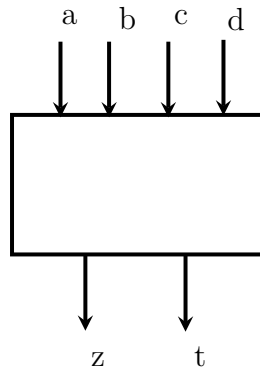


Figura 3.8: Encoder di un numero da 2 bit.

Gli output dell'encoder a 2 bit saranno

$$z = \bar{a}bcd + a\bar{b}cd = \bar{c}d(\bar{a}b + a\bar{b})$$

$$t = \bar{a}b\bar{c}d + a\bar{b}\bar{c}d$$

Decoder Un decoder esegue l'operazione opposta.

a	b	u	v	z	t
0	0	0	0	0	1
0	1	0	0	1	0
1	0	0	1	0	0
1	1	1	0	0	0

Tabella 3.6: Tabella di verità di un decoder da 2 bit.

Gli output saranno $u = ab, v = a\bar{b}, z = \bar{a}b, t = \bar{a}\bar{b}$

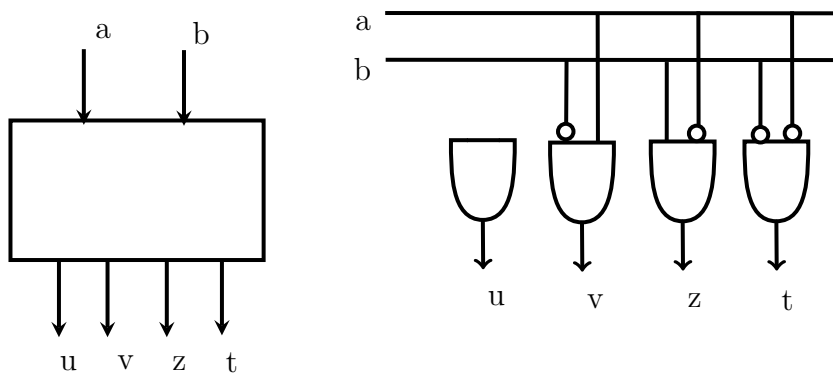


Figura 3.9: Decoder di un numero da 2 bit.

Confrontatore Un confrontatore (comparator) confronta due configurazioni di bit e controlla che siano uguali. Un confrontatore da 1 bit è esattamente un gate NOT XOR.

$$z = ab + \bar{a}\bar{b}$$

a	b	t
0	0	1
0	1	0
1	0	0
1	1	1

Tabella 3.7: Tabella di verità di un confrontatore da 1 bit.

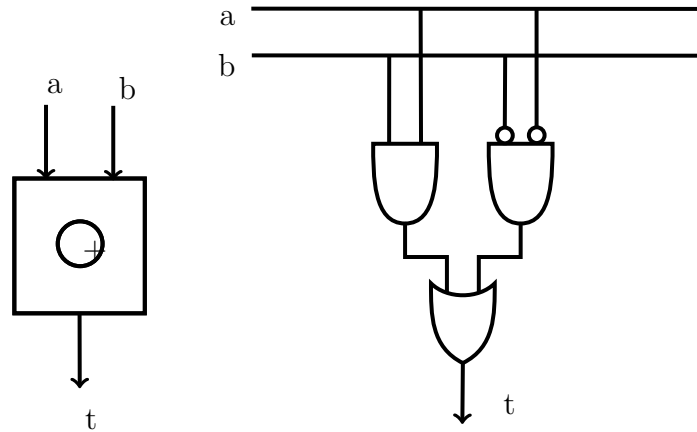


Figura 3.10: Confrontatore da 1 bit

Per realizzare un confrontatore da più bit si mettono in parallelo più confrontatori da un solo bit e si uniscono gli output con un albero di AND.

Un confrontatore a 2 bit impiegherà tempo $2\Delta t + \Delta t$

Per confrontare valori da n bit si impiegherà però un tempo di $\lceil \log_k n \rceil (\Delta t \cdot \text{ogni livello AND})$

Confronto maggiore In maniera simile ad un confrontatore si può realizzare un componente che controlla se un numero di n bit è maggiore di un altro.

x_0	x_1	y_0	y_1	z
0	0	-	-	0
0	1	0	0	1
1	-	0	-	1
1	1	-	0	1

Tabella 3.8: Tabella di verità di un confronto maggiore da 2 bit.

$y_1 y_0 \backslash x_1 x_0$	00	01	11	10
00	0	1	1	1
01	0	0	1	1
11	0	0	0	0
10	0	0	1	0

Tabella 3.9: Mappa di Karnaugh di un confrontatore dell'operatore maggiore da 2 bit.

La formula booleana ottenuta è $z = x_0 \overline{y_1} \overline{y_0} + x_1 \overline{y_1} + x_1 x_0 \overline{y_0}$

ALU Una ALU, ovvero Arithmetic Logic Unit è un componente che in genere, in input accetta due parole x, y entrambe da n bit, e può fare due o più operazioni. L'operazione viene selezionata attraverso uno o più bit di controllo. In output ci sarà un uscita da n bit. Ad esempio, prendiamo una ALU a 4 bit con somma/sottrazione, che accetterà un solo bit di controllo.

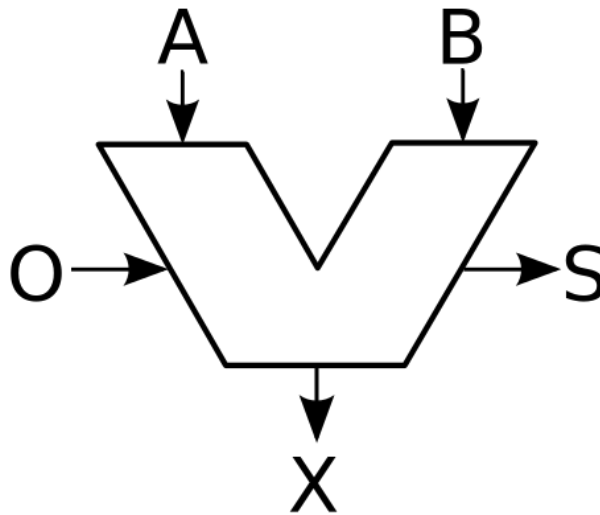


Figura 3.11: Simbolo di una ALU

I livelli di porte AND saranno $\lceil \log_k 8 \rceil$ che verranno sommati ai livelli di porte OR che saranno $\log_k(128)$. Le ALU si distinguono fra intere e a floating point. Per semplicità vedremo le ALU intere con somma e moltiplicazione.

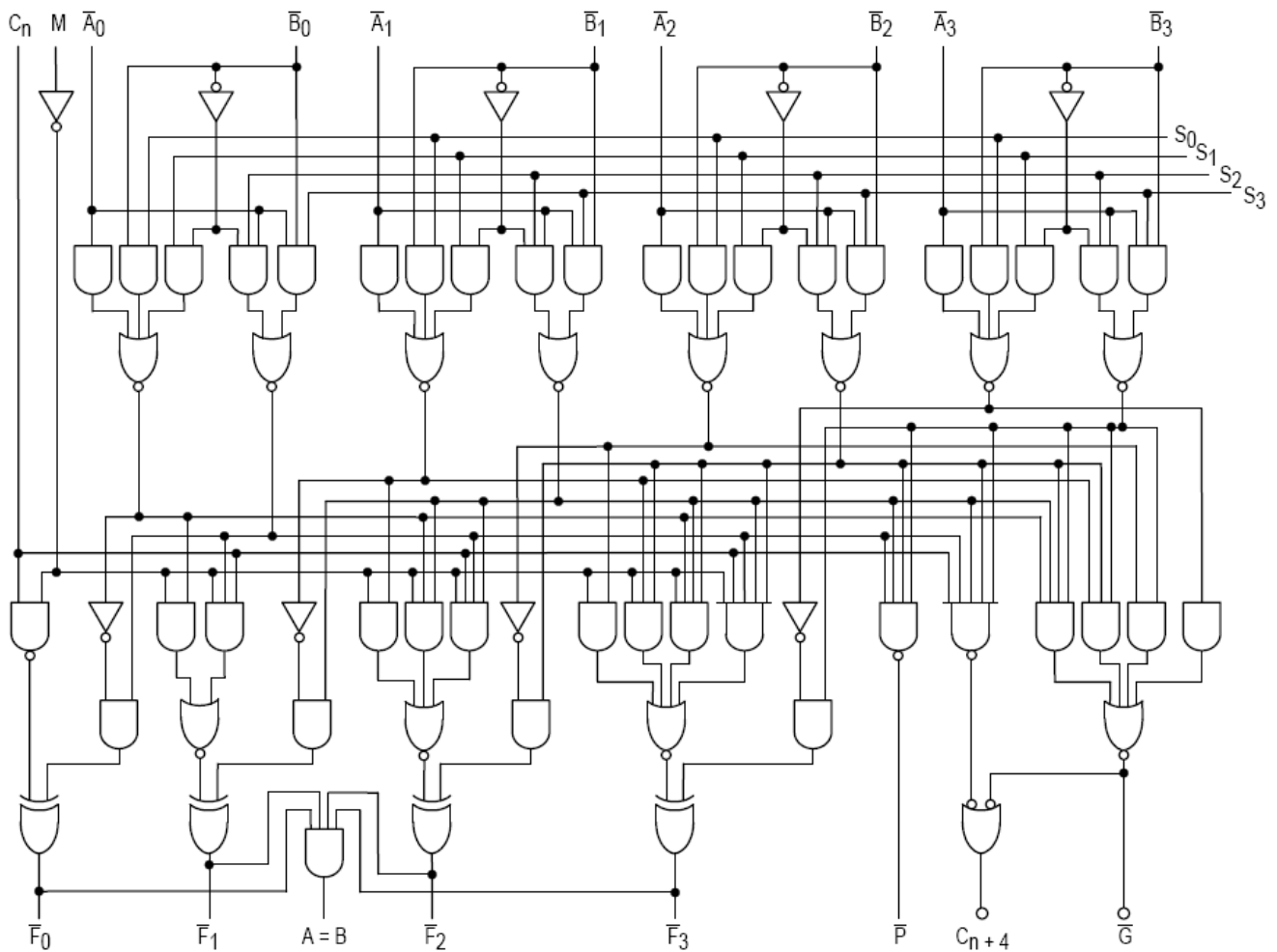


Figura 3.12: Schematica del circuito integrato della ALU 74181, la prima ALU completa su un singolo chip.

Multiply and Add Un componente Multiply and Add (MUL&ADD) è un componente ALU che realizza l'operazione matematica $\sum_i x_i y_i$, ovvero prende gli ingressi x_i, y_i , li moltiplica e li accumula sommando.

Ritardi di propagazione Sappiamo che il cambiamento fra HIGH e LOW (0 e 1) nei circuiti digitali non è istantaneo. A volte, il ritardo di propagazione dei transistor può causare dei glitch (fluttuazioni) che potrebbero causare una lettura incorretta dell'output di un circuito. Per ovviare a questo problema si introduce un componente chiamato **clock**, che scandisce un ciclo preciso per permettere ai valori di propagarsi nei circuiti e leggere l'output preciso alla fine della propagazione Δt .

3.2 Reti sequenziali e Automi

Abbiamo visto nel corso di programmazione 1 gli automi a stati finiti (Finite State Machines o FSM). Per ora, abbiamo visto circuiti logici che implementano funzioni senza stato. Per implementare lo stato possiamo utilizzare gli automi a stati finiti. In un automa di Moore gli output sono determinati solo dallo stato corrente, mentre negli automi di Mealy gli output sono determinati sia dallo stato corrente che dagli input correnti.

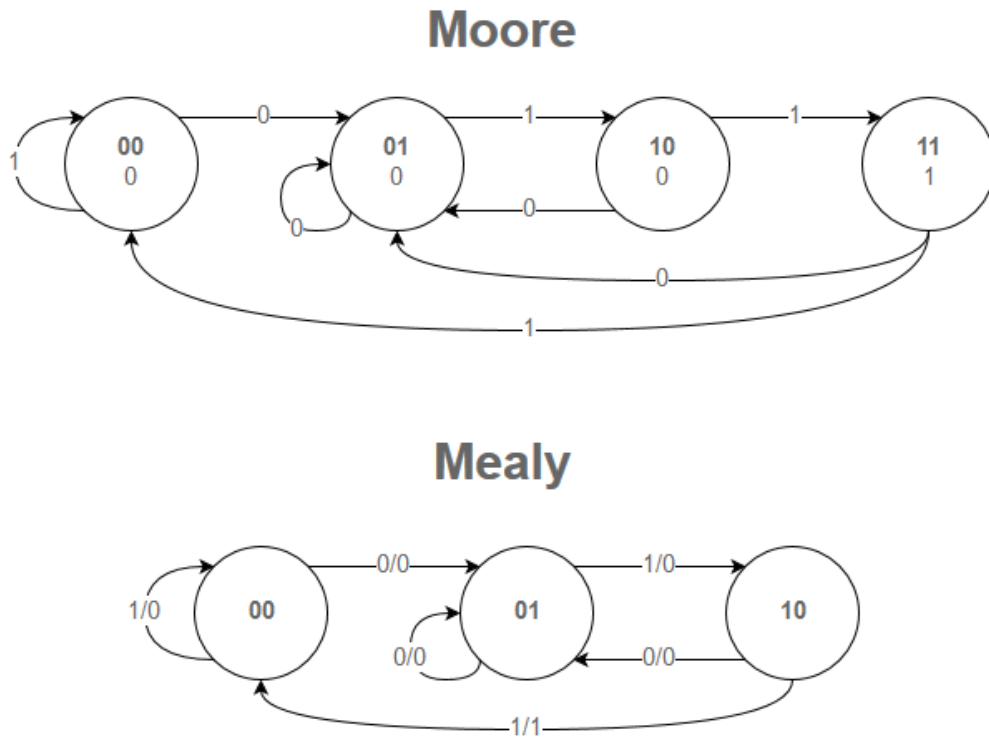


Figura 3.13: Automi di Moore e Automi di Mealy

Per portare un Automa di Mealy ad una rete sequenziale abbiamo bisogno di tre blocchi di una rete combinatoria. Uno che dallo stato precedente s e dall'input x dia in output l'uscita z , e un altro che ricevendo x, s in input restituisca s^1 (lo stato del prossimo nodo del grafo) e un **SR Latch** (o **flip-flop**). I flip-flop possono essere di tipo SR, D, T e JK .

SR Latch o Flip Flop Un componente SR Latch è un componente che implementa un bit di memoria. Impostando l'input $S = 1$ (bit di set) e $R = 0$ (bit di reset) l'output sarà $Q = 1, \bar{Q} = 0$ mentre impostando gli input $S = 0, R = 1$ l'output è $Q = 0, \bar{Q} = 1$. Il fulcro della funzionalità di un SR Latch è che se entrambi gli input $S = 0, R = 0$ allora l'output è in uno stato detto **latch**, ovvero che "ricorda" lo stato precedente.

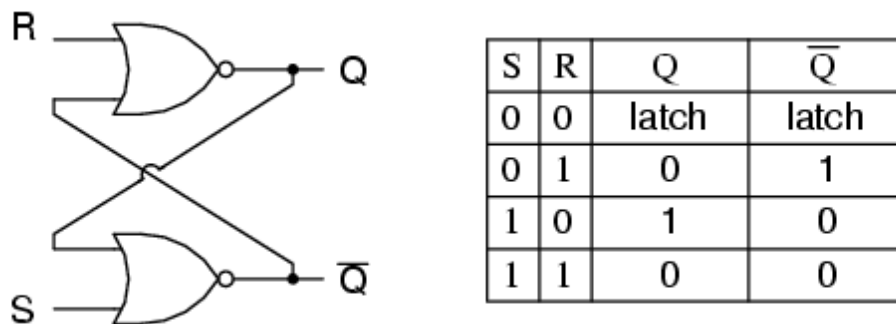


Figura 3.14: Tabella di Verità e circuito di un SR Latch

Clock e D-latch Introduciamo un componente fondamentale, il **clock**, che introduce una dimensione temporale nelle reti combinatorie scandendo un **ciclo**. Un ciclo di clock ha lunghezza τ . Possiamo così estendere un SR Latch rendendolo un **D-Latch**, che riceve in input soltanto il

ciclo di clock e un bit D , ovvero il bit che va "ricordato". Per evitare **glitch** possiamo scrivere sul D-Latch soltanto quando il segnale di clock è nello stato HIGH.

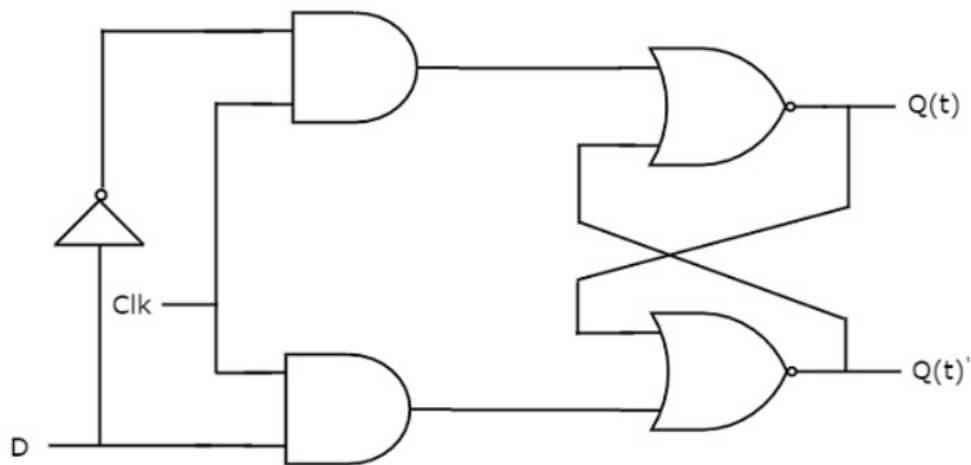


Figura 3.15: Un componente D-latch

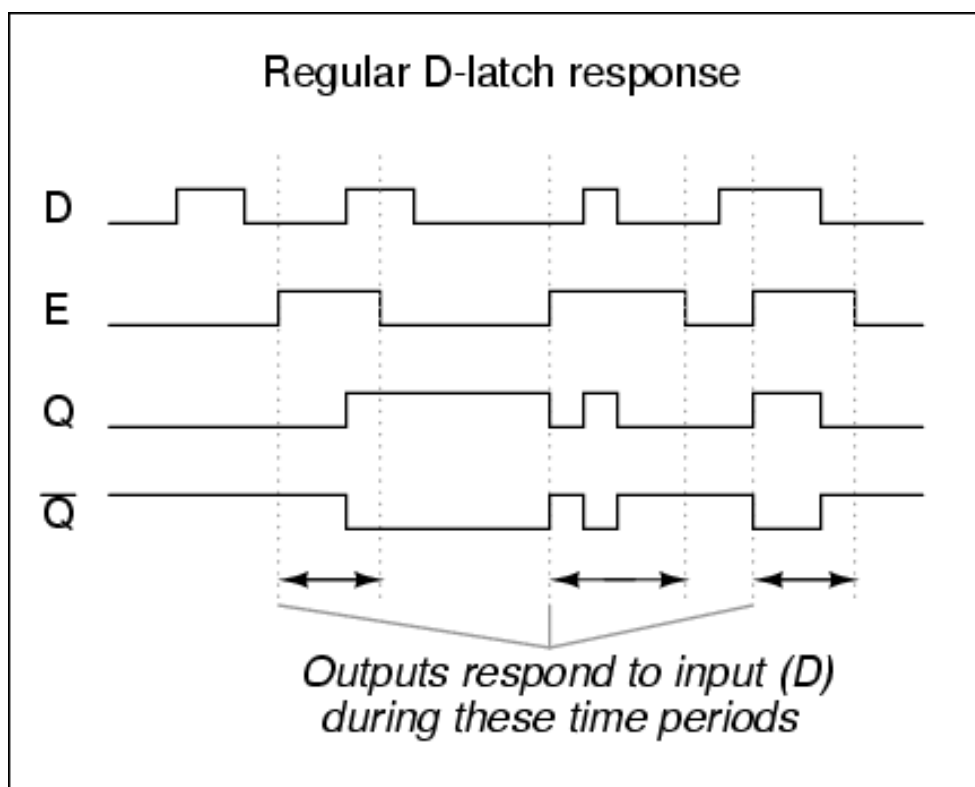
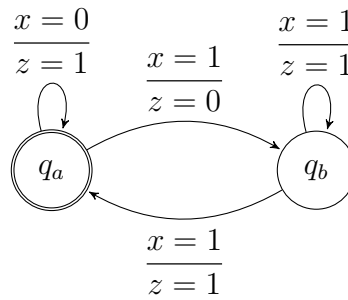


Figura 3.16: Diagramma temporale di un D-latch

3.2.1 Realizzazione di una rete sequenziale di un Automa di Mealy con un clock e registri

Consideriamo, ad esempio, un automa che calcola la parità del numero di 1 in una sequenza di bit. Ricevendo in input una sequenza di bit 1,0,0,0,1,0,1 ad esempio restituirebbe in output 0,1,1,1,1,0,0 (l'output diventa high)

Figura 3.17: Automa di parità di una sequenza di bit



Per trasformare l'automa di Mealy in una rete sequenziale avremo una rete combinatoria che calcola lo stato successivo, un registro che mantiene lo stato e una rete combinatoria che calcola l'uscita. Siccome l'automa di Mealy usa l'input x per decidere le transizioni, l'input x dovrà essere passato ad entrambe le reti combinatorie dei nodi del grafo.

Chiamiamo la prima rete combinatoria σ e la seconda ω

x	s	s'
0	0	0
1	0	1
0	1	1
1	1	0

Tabella 3.10: Tabella di verità della rete combinatoria σ

x	s	z
0	0	1
1	0	0
0	1	0
1	1	1

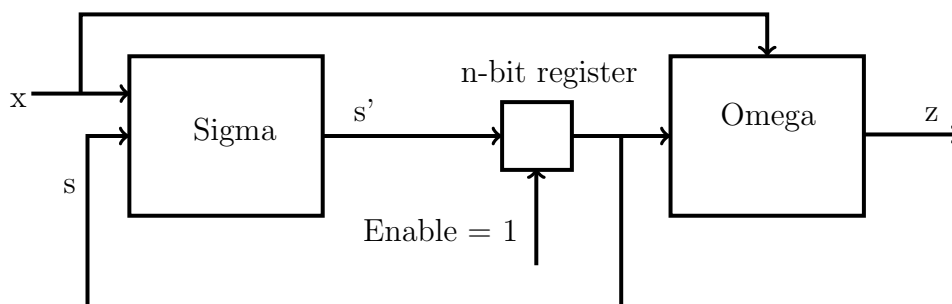
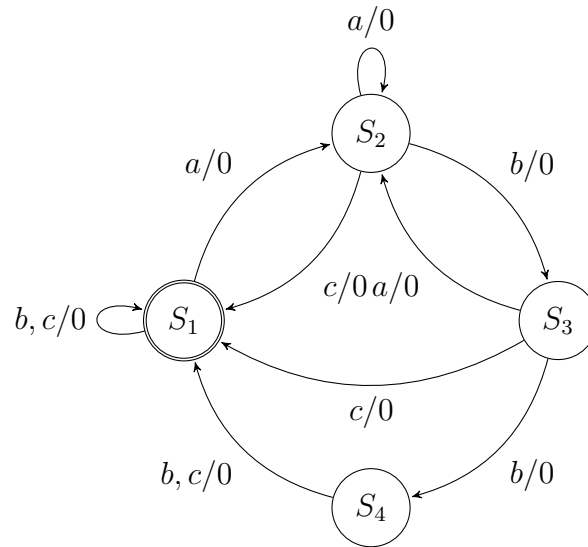
Tabella 3.11: Tabella di verità della rete combinatoria ω 

Figura 3.18: Rete sequenziale dell'automa di Mealy in figura precedente

3.2.2 Automa che riconosce "abba"

Realizziamo un automa di Mealy che riconosce le stringhe "abba" da un insieme $\{a, b, c\}$. La rete sequenziale dell'automa si realizzerà con i componenti visti in figura 3.18. Consideriamo la rappresentazione binaria dell'alfabeto con $a = 00, b = 01, c = 11$. Per gli stati usiamo la codifica $S_1 = 00, S_2 = 01, S_3 = 11, S_4 = 10$

Figura 3.19: Automa di Mealy che riconosce "abba"



	s_1	s_2	x_1	x_2	z
Stato S_1	0	0	-	-	0
Stato S_2	0	1	-	-	0
Stato S_3	1	1	-	-	0
Stato S_4	1	0	0	0	1
	1	0	0	1	0
	1	0	1	1	0

Tabella 3.12: Tabella di verità dell'output della rete sequenziale dell'automa "abba" (ω)

	s_1	s_2	x_1	x_2	s'_1	s'_2
Stato S_1	0	0	0	0	0	1
	0	0	0	1	0	0
	0	0	1	1	0	0
Stato S_2	0	1	0	0	0	1
	0	1	0	1	1	1
	0	1	1	1	0	0
Stato S_3	1	1	0	0	0	1
	1	1	0	1	1	0
	1	1	1	1	0	0
Stato S_4	1	0	0	0	0	0
	1	0	0	1	0	0
	1	0	1	1	0	0

Tabella 3.13: Tabella di verità del cambio di stato dell'automa per riconoscere "abba" (σ)

Avremo che la formula booleana sarà per il primo e secondo bit di stato:

$$s'_1 = \overline{s_1}s_2\overline{x_1}x_2 + s_1s_2\overline{x_1}x_2 = s_2\overline{x_1}x_2$$

$$s'_2 = \overline{s_1}\overline{s_2} + \overline{s_1}s_2\overline{x_1} + s_2\overline{x_2}$$

La formula per l'uscita sarà $z = s_1 \overline{s_2 x_1 x_2}$. Ci rimane da definire il registro da 2 bit per definire tutta la rete sequenziale. Abbiamo supposto che la rete sequenziale funzioni ricevendo un segnale di clock ad intervalli regolari. La rete di output ω è definita utilizzando soltanto AND ed impiegherà Δt per eseguire l'operazione. La funzione di cambio di stato σ è definita invece con 1 AND e 1 OR ed impiegherà $2\Delta t$ per l'operazione. Il ciclo di clock dev'essere almeno $\tau = \delta + \max\{t_\sigma, t_\omega\}$ dove δ è la durata del segnale HIGH nel clock.

Capitolo 4

Verilog e RTL

Gli RTL, o Register Transfer Language, permettono di descrivere cosa succede a livello di circuito fra registri. Vengono utilizzati per descrivere l'hardware. Vedremo il linguaggio **Verilog**. Gli RTL permettono di descrivere e comporre dei moduli. Il libro di testo propone il dialetto **System Verilog** che mette a disposizione due metodi per descrivere i moduli. Un metodo è il metodo *constructive*, noi vedremo il metodo *behavioral* dove ad esempio un Multiplexer da 2 vie 1 bit è descritto da:

```
1      z = (ic == 0 ? x : y)
```

Verilog è un linguaggio compilato. Un file System Verilog compilato produce una traccia di esecuzione e un eseguibile che simula il comportamento dei moduli. Viene detta **simulazione**.

Un programma Verilog può anche essere dato in input a un programma detto **synthetizer**, che produce una **netlist**, ovvero una lista dei componenti e dei collegamenti per realizzare il modulo fisicamente. Un altro modo per realizzare la sintesi è utilizzare un **FPGA**, o Field-programmable gate array. Un FPGA è un circuito integrato composto da una matrice di celle, e una singola cella può:

1. Eseguire una funzione booleana di 3-5 ingressi con 1 uscita
2. Implementare un bit di memoria
3. Routing

Un FPGA moderno comprende, oltre a delle celle, delle righe che contengono diversi componenti come delle ALU.

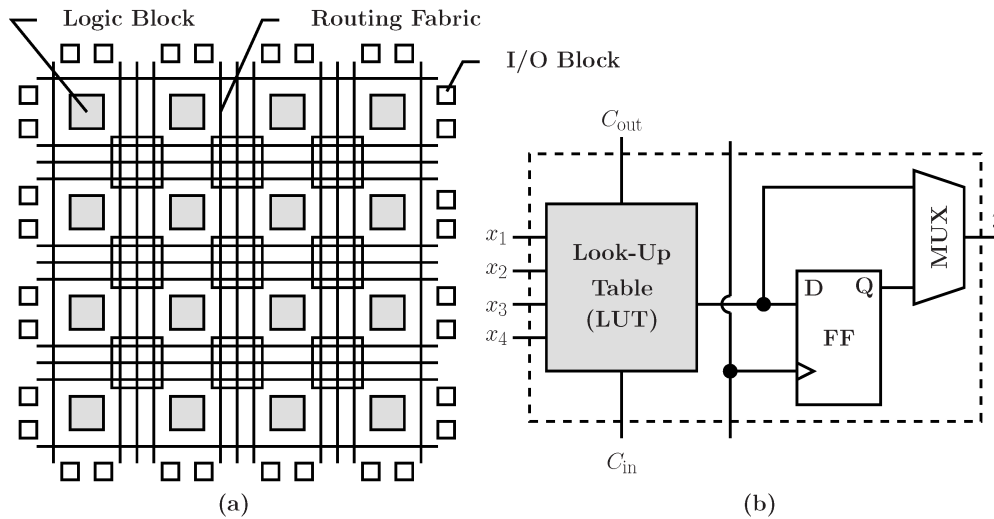


Figura 4.1: Schema FPGA

4.1 Scrivere e compilare System Verilog

Creiamo un multiplexer da due bit con System Verilog, compiliamo e visualizziamo con **GTK-Wave**

Listing 4.1: mux.sv

```
1 module mux (output logic z, input logic x, y, ic );
2     assign z = (ic == 0 ? x : y);
3 endmodule
```

Listing 4.2: test_mux.sv

```
1 module test_mux();
2
3     reg my_x, my_y, my_ic;
4     wire my_z;
5
6     mux mymux(my_z, my_x, my_y, my_ic);
7
8     initial
9         begin
10             // Dump log to a file
11             $dumpfile("provamux.vcd");
12             $dumpvars;
13             my_x = 0;
14             my_y = 0;
15             my_ic = 1;
16
17             #10
18                 my_x = 1;
19
20             $finish;
21         end
22 endmodule // test_mux
```

Per compilare, eseguiamo da terminale

```
iverilog -g2005-sv nome_sorgente.sv -o nome_eseguibile
```

Quindi, per compilare entrambi i file e caricarli in GTKWave:

```
iverilog -g2005-sv test_mux.sv mux.sv -o test_mux
# Eseguiamo la simulazione
./test_mux
# Viene creato il file provamux.vcd, carichiamolo in GTKWave
gtkwave provamux.vcd &
```

Listing 4.3: Multiplexer da 4 vie ad 1 bit

```
1 module mux4(output logic z,
2     input logic [3:0] x,
3     input logic [0:1] ic);
4
5     logic k12k3;
6     logic k22k3;
7
8     mux k1(k12k3,x[0],x[1],ic[0]);
9     mux k2(k22k3,x[2],x[3],ic[0]);
10    mux k3(z,k12k3,k22k3,ic[1]);
11 endmodule
```

Listing 4.4: Multiplexer di variabili booleane

```
1 module mux(output logic z, input logic x, y, ic);
2
3     assign
4         z = ((~ic) & x) | (ic & y);
5
6 endmodule
```

4.2 Esercizi

Automa di Mealy per riconoscere "abba" Abbiamo visto nella sezione precedente un automa di Mealy a stati finiti per riconoscere una stringa "abba" in una stringa nell'alfabeto $\{a, b, c\}$. Abbiamo 3 moduli. Uno per il registro, uno per il modulo ω e uno per il modulo σ . Scriviamolo in Verilog.

Listing 4.5: Registro a N bit

```

1 // to allocate a register module in a program or another module:
2 // register #(1) state
3 // register is the type, #(p1, p2, p3) are the parameters in order
4 // and state is the variable name.
5 module register(output reg [1:N]state, input clock, input enable, input [1:N]inval);
6                                     // reg keyword denotes a flip-flop
7     parameter N = 2;
8
9     // initial runs a block when the module is created (initialized)
10    initial
11        begin
12            state = 0;
13        end
14
15    // always, similar to 'while true' runs a block when
16    // posedge means when the clock becomes positive
17    always @(posedge clock)
18        begin
19            // <= is the asynchronous assignment operator
20            // while = is sequential, <= tries to run subsequent
21            // assignments at the same time. it is recommended
22            // to use asynchronous assignment inside of always blocks
23            if(enable) state <= inval;
24        end
25 endmodule //register

```

Listing 4.6: Modulo di σ

```

1 // State change function
2 module sigma(input [1:2]s, output [1:2]news, input [1:2]x);
3
4 endmodule

```

Listing 4.7: Modulo di ω

```

1 // Output module of the Mealy automata finding "abba" in strings of {a,b,c}
2 // The syntax [1:2] denotes an array indexed from 1 to 2
3 module omega(output z, input [1:2]s, input [1:2]x)
4
5     // Continuous assignment, z should continuously be equal to the equation
6     assign
7         z = (s[1] && ~s[2] && ~x[1] && ~x[2])
8
9 endmodule //omega

```

Listing 4.8: Modulo della rete sequenziale

```
1  'include "register.v"
2  'include "sigma.v"
3  'include "omega.v"
4
5  // Module for the Mealy automata recognising an "abba" string in an {a,b,c} alphabet
6  module abba(output z, input[1:2]x, input clock);
7      wire [1:2] outreg;
8      wire [1:2] inreg;
9
10     register #(2) s(outreg, clock, 1'b1, inreg);
11     sigma sm(inreg, outreg, x);
12     omega om(z, outreg, x);
13 endmodule
```

