

Appunti di Architettura degli Elaboratori

Alessandro Cheli - Prof. Marco Danelutto

A.A 2019-2020

Indice

I	1
1 Introduzione al Corso	3
2 Rappresentazioni Numeriche e Testuali	5
2.1 Aritmetica Binaria	5
2.2 Algebra di Boole, Porte Logiche e Mappe di Karnaugh	7
2.2.1 Teoremi dell'Algebra Booleana	12
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	24
4 Reti Sequenziali, Verilog e RTL	27
4.1 Scrivere e compilare System Verilog	27
4.2 Esercizi	30
4.2.1 Automa che riconosce "abba"	30
4.2.2 Automa di Moore per riconoscere "abba"	34
4.2.3 Mealy con Delay \equiv Moore	37
4.3 Componenti per il Calcolo	39
5 Memorie e Parallelismo	45
5.1 Memorie	45
5.1.1 Calcolatrice con registri di memoria	51
5.2 Parallelismo	51
5.2.1 Processore superscalare	56
5.3 Homework	59
II	65
6 Assembler ARM e microarchitettura	67
6.1 Introduzione all'Assembler ARM	67
6.2 Istruzioni Assembler ARMv7	67
6.3 Direttive, Pseudoistruzioni e Programmi	72
6.4 Homework Assembly ARM	78
6.5 Instruction Set ARMv7	82
6.6 Istruzioni in virgola mobile	84
7 Microarchitettura	85
7.1 Datapath	85
7.2 Processori Single Cycle	85

Parte I

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 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

Figura 1.1: Sinclair ZX80



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.

Definizione 1.0.1. 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)

Definizione 2.1.1. 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).

Definizione 2.1.2. 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

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]

2.2 Algebra di Boole, Porte Logiche e Mappe di Karnaugh

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 (vtaggio), 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

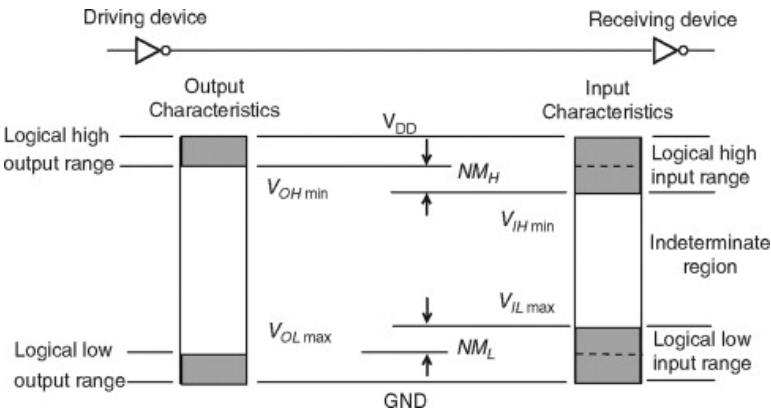


Figura 2.5: Margine di rumore nei circuiti digitali

materiali come gallio e silicio.

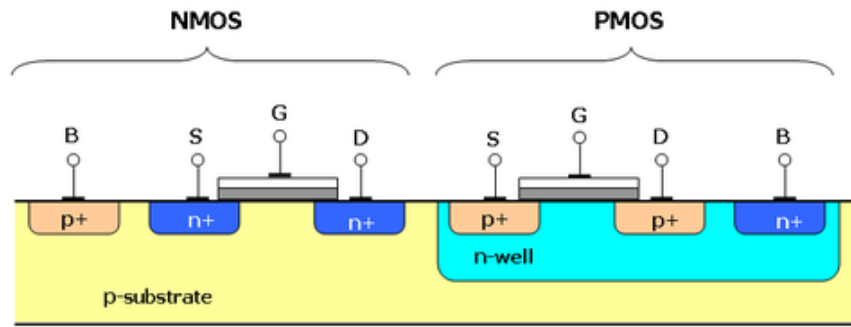


Figura 2.6: Transistor NMOS

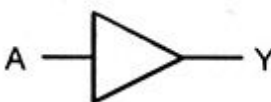
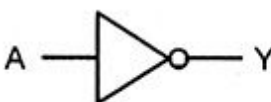
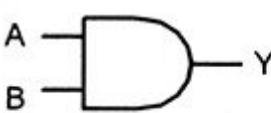
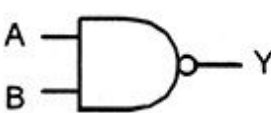

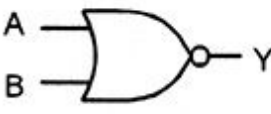


Logic function	Logic symbol	Truth table	Boolean expression															
Buffer		<table border="1" data-bbox="892 356 1043 490"><tr><td>A</td><td>Y</td></tr><tr><td>0</td><td>0</td></tr><tr><td>1</td><td>1</td></tr></table>	A	Y	0	0	1	1	$Y = A$									
A	Y																	
0	0																	
1	1																	
Inverter (NOT gate)		<table border="1" data-bbox="892 546 1043 680"><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="868 703 1067 904"><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="868 927 1067 1128"><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="868 1151 1067 1352"><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="868 1375 1067 1576"><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="868 1599 1067 1800"><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="868 1823 1067 2024"><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 2.7: Tabella delle porte logiche comuni

Inverter (Not Gate)

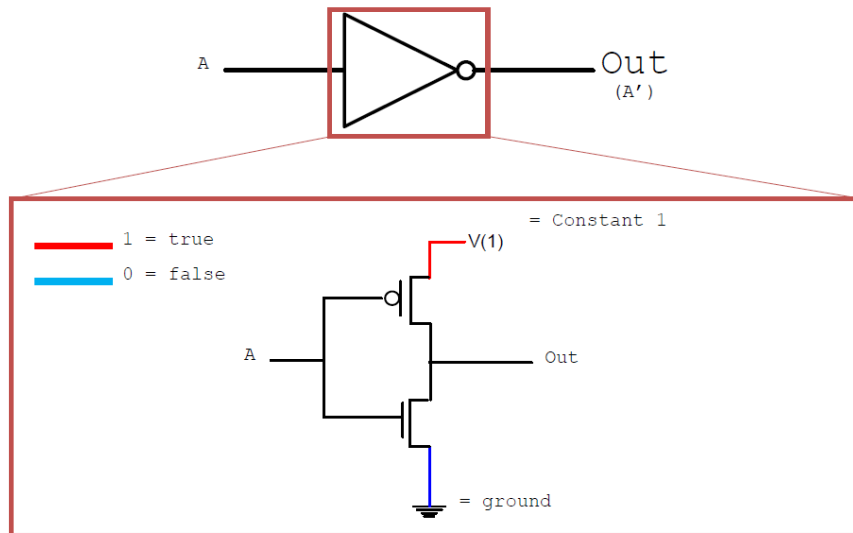


Figura 2.8: Porta NOT con transistor PMOS e NMOS

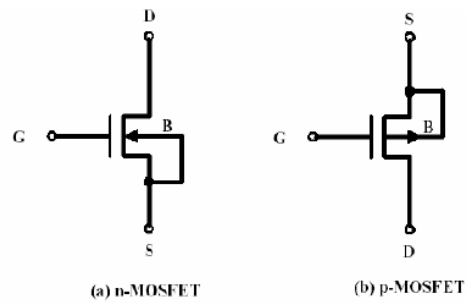
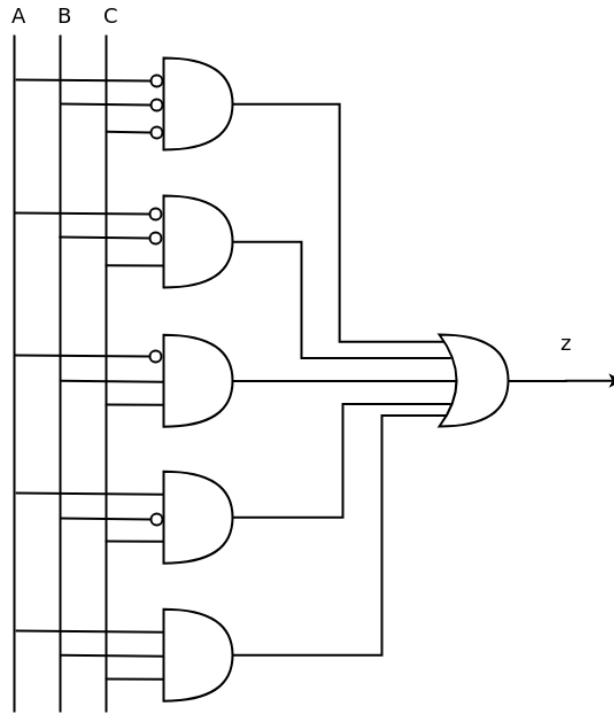


Figura 2.9: Transistor NMOS e PMOS

Definizione 2.2.1. Conversione da formula booleana a circuitoFigura 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: $z = (A + \bar{B} + C)(\bar{A}\bar{B}C)(\bar{A}\bar{B}C)$

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

Definizione 2.2.2. 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 $z = \overline{A}\overline{B} + C$

Definizione 2.2.3. 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.

Definizione 2.2.4. 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.

2.2.1 Teoremi dell'Algebra Booleana

Breve ripasso dei teoremi della Logica Booleana.

Teorema 2.2.1. Elemento Identità di prodotto e somma

$$\begin{aligned} A \cdot 1 &= A \iff A \wedge T \equiv A \\ A + 0 &= A \iff A \vee F \equiv A \end{aligned}$$

Teorema 2.2.2. Elemento assorbente

$$\begin{aligned} A \cdot 0 &= 0 \iff A \wedge F \equiv F \\ A + 1 &= 1 \iff A \vee T \equiv T \end{aligned}$$

Teorema 2.2.3. Idempotenza

$$\begin{aligned} A \cdot A &= A \iff A \wedge A \equiv A \\ A + A &= A \iff A \vee A \equiv A \end{aligned}$$

Teorema 2.2.4. Complemento

$$\begin{aligned} A \cdot \bar{A} &= 0 \iff A \wedge \neg A \equiv F \\ A + \bar{A} &= 1 \iff A \vee \neg A \equiv T \end{aligned}$$

Teorema 2.2.5. Commutatività

$$\begin{aligned} A + B &= B + A \\ A \cdot B &= B \cdot A \end{aligned}$$

Teorema 2.2.6. Associatività

$$\begin{aligned} (A \cdot B) \cdot C &= A \cdot (B \cdot C) \\ (A + B) + C &= A + (B + C) \end{aligned}$$

Teorema 2.2.7. Distributività

$$\begin{aligned} (A \cdot B) + C &= (A + C) \cdot (B + C) \\ (A + B) \cdot C &= AC + BC \end{aligned}$$

Teorema 2.2.8. DeMorgan

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

Esempio 2.2.1. Semplifichiamo la formula booleana $z = \bar{A}\bar{B}\bar{C} + \bar{A}\bar{B}C + \bar{A}BC + A\bar{B}C + ABC$

$$\begin{aligned} \begin{cases} \bar{A}\bar{B}\bar{C} + \bar{A}\bar{B}C \equiv \bar{A}\bar{B}(\bar{C} + C) \equiv \bar{A}\bar{B} \\ A\bar{B}C + ABC \equiv AC(\bar{B} + B) \equiv AC \end{cases} \\ \implies z = \bar{A}\bar{B} + \bar{A}BC + AC \end{aligned}$$

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.

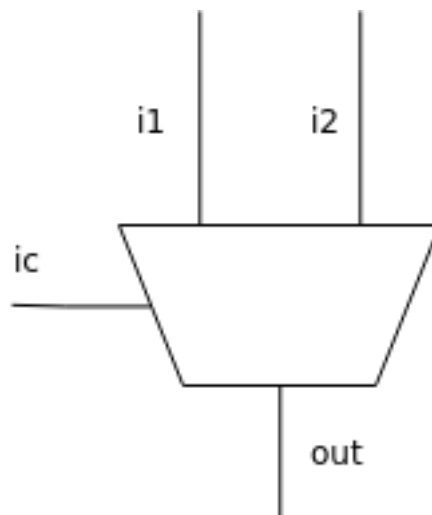


Figura 3.1: Multiplexer 2 vie 1 bit

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

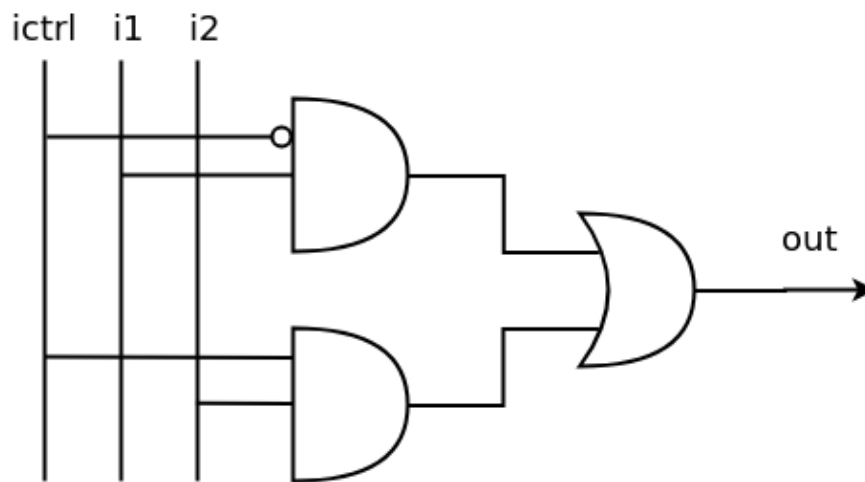


Figura 3.2: Circuito logico di un Multiplexer 2 vie 1 bit

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.

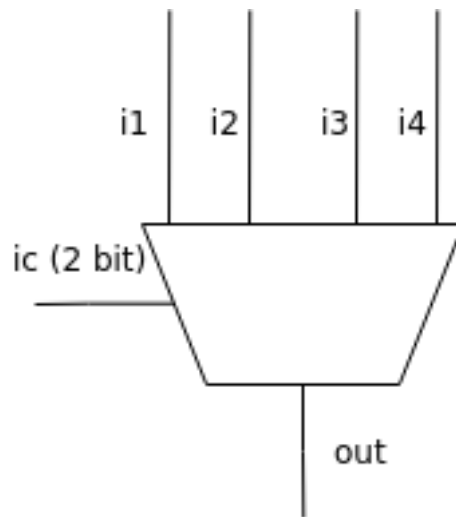


Figura 3.3: Multiplexer 4 vie 1 bit

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

$$\text{out} = (\overline{ic_1} \cdot \overline{ic_2} \cdot in_1) + (\overline{ic_1} \cdot ic_2 \cdot in_2) + (ic_1 \cdot \overline{ic_2} \cdot in_3) + (ic_1 \cdot ic_2 \cdot in_4)$$

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

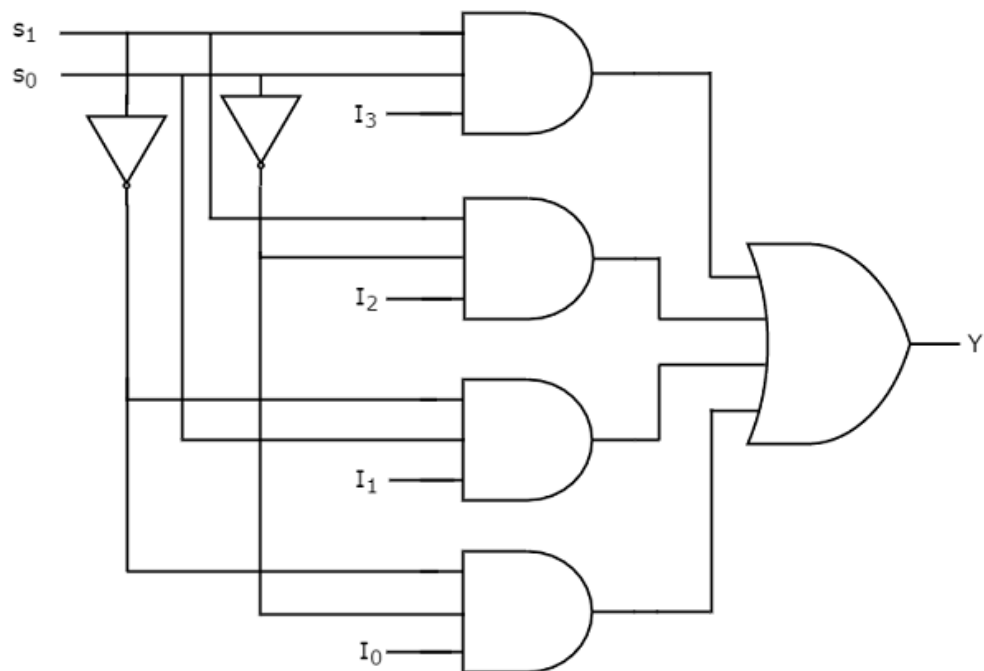


Figura 3.4: Circuito logico di un Multiplexer 4 vie 1 bit

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.

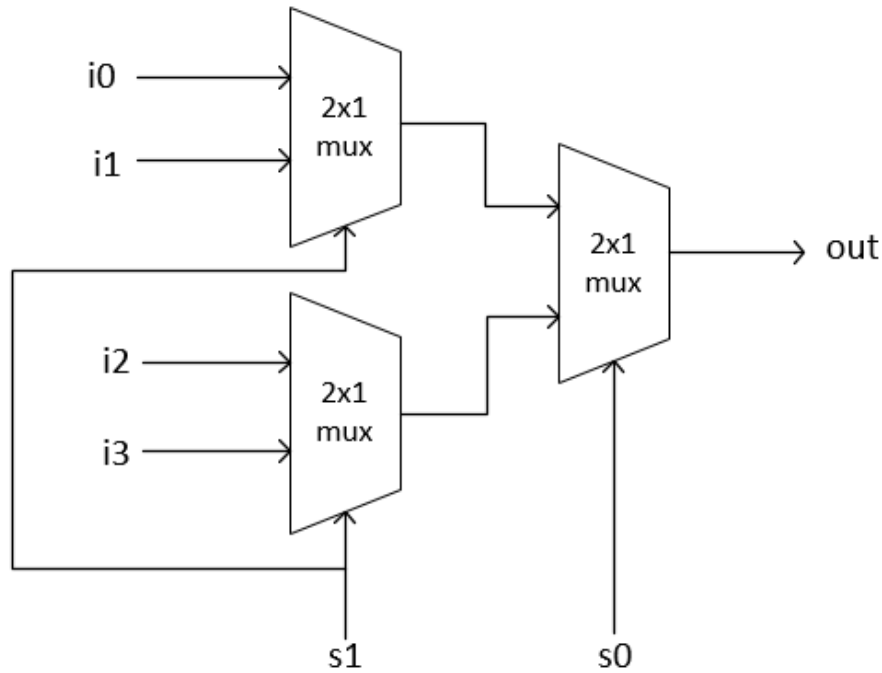


Figura 3.5: Multiplexer 4 vie 1 bit composto da due Multiplexer 2x1

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.

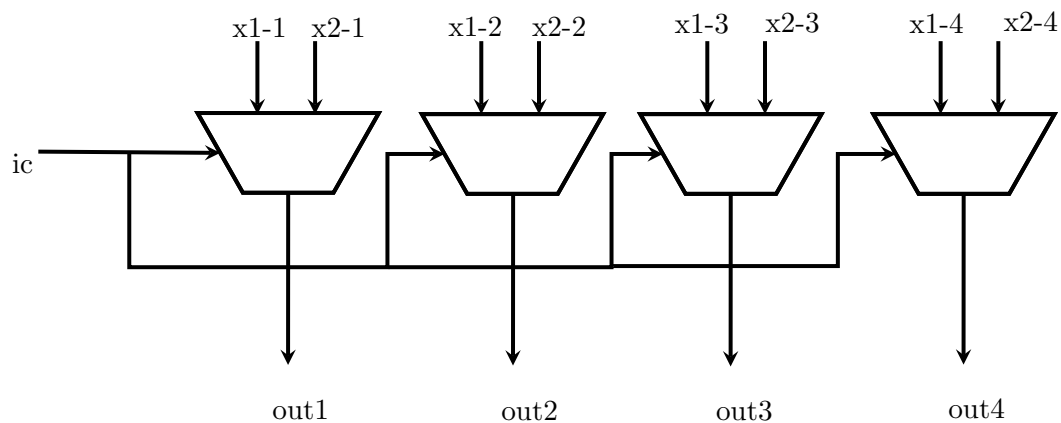


Figura 3.6: Multiplexer 2 vie da 4 bit

Sommatore di numeri Un sommatore è un componente che somma due numeri in input e restituisce in output un risultato ed un riporto. Un sommatore di due numeri che non accetta un riporto in input è detto **Half Adder** mentre un sommatore che accetta un riporto è detto Full Adder.

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 un Full Adder.

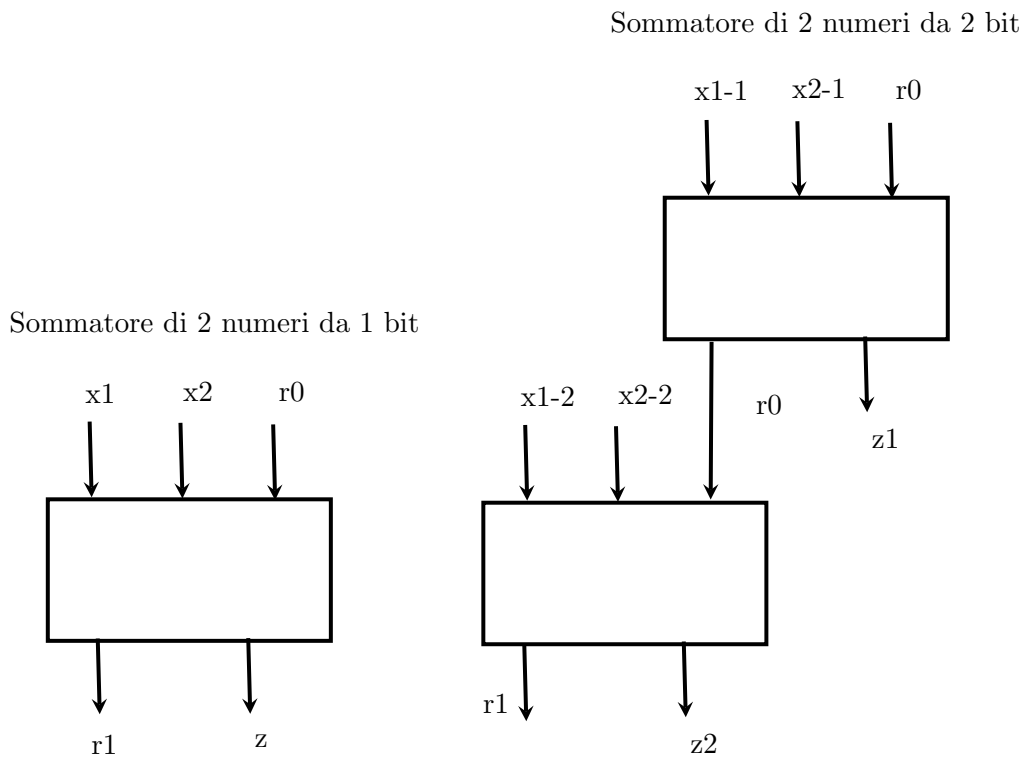


Figura 3.7: Sommatori di due numeri

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.

Gli output dell'encoder a 2 bit saranno

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

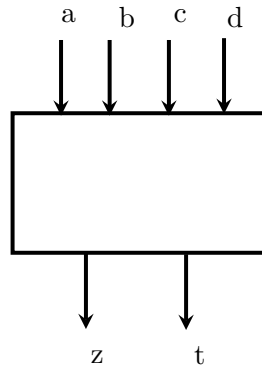


Figura 3.8: Encoder di un numero da 2 bit.

$$t = \overline{a}bcd + a\overline{b}cd$$

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

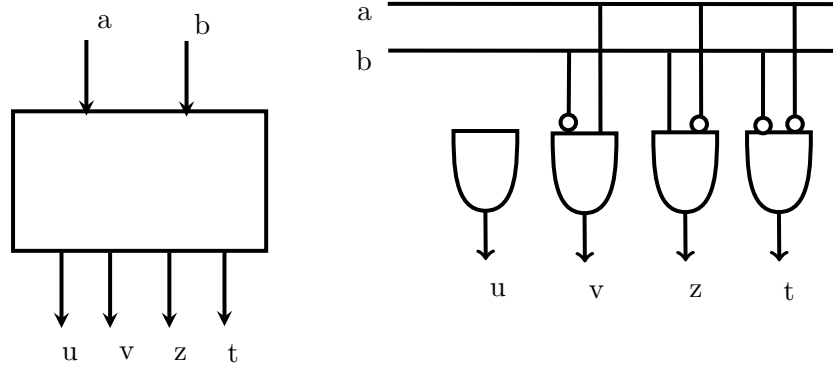


Figura 3.9: Decoder di un numero da 2 bit.

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.

$$z = ab + \overline{ab}$$

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.

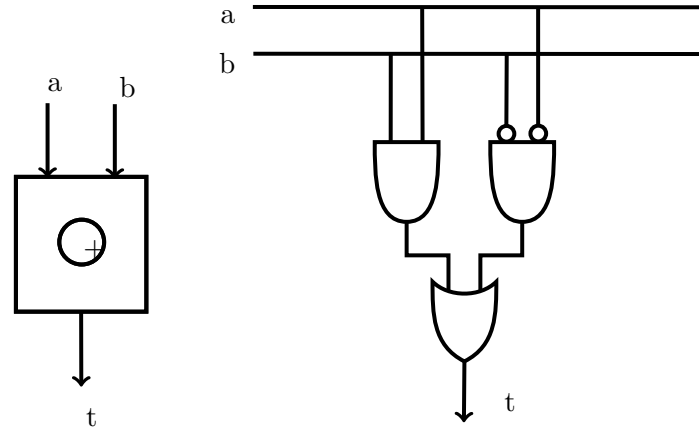


Figura 3.10: Confrontatore da 1 bit

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.

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.

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.

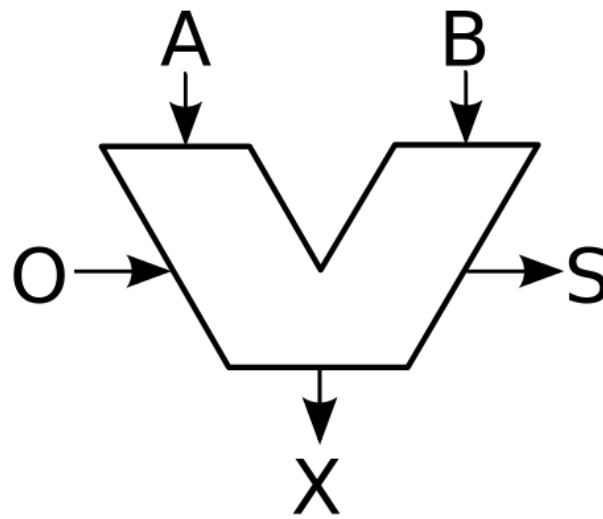


Figura 3.11: Simbolo di una ALU

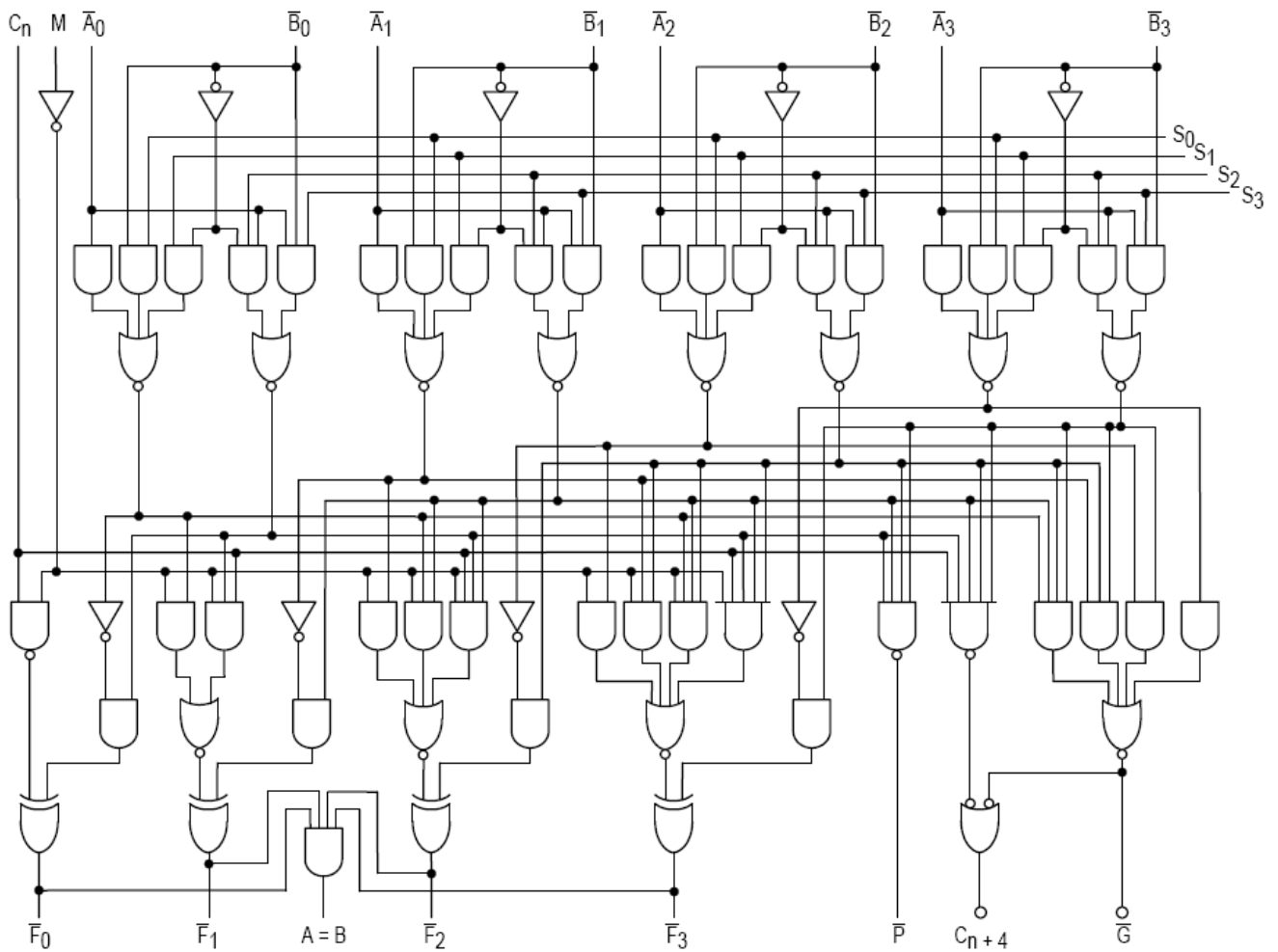


Figura 3.12: Schematica del circuito integrato della ALU 74181, la prima ALU completa su un singolo chip.

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.

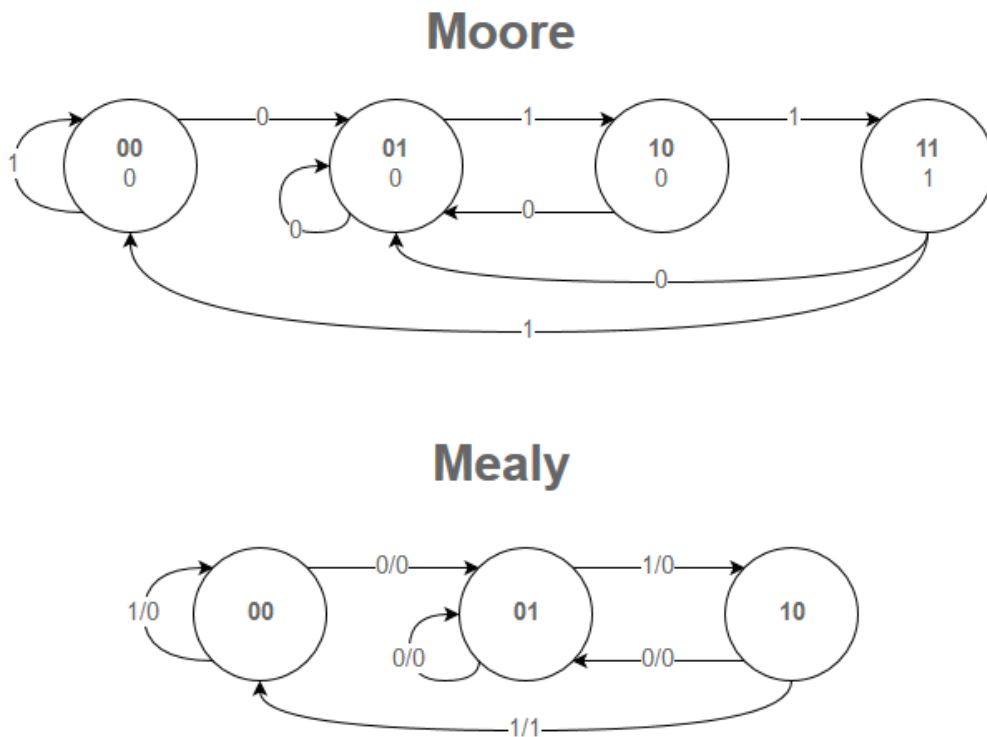


Figura 3.13: Automi di Moore e Automi di Mealy

Gli automi di Moore hanno sempre un numero di stati **maggiore o uguale** agli 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 latch possono essere di tipo SR, D, T e JK . Un **flip-flop** è un latch che accetta in input un segnale di clock e i due bit S, R (Set e Reset)

SR Latch 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.

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

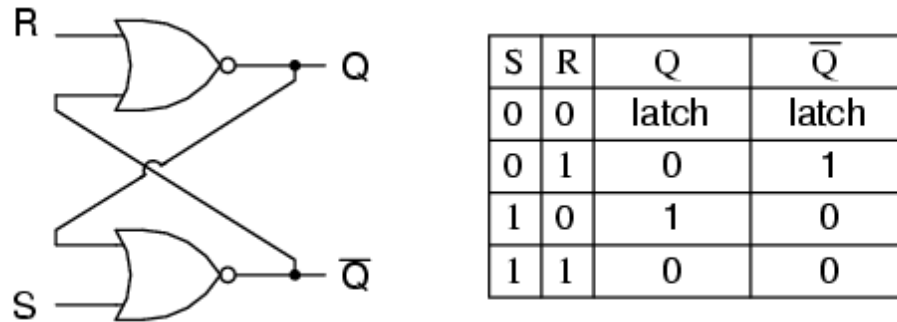


Figura 3.14: Tabella di Verità e circuito di un SR Latch

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.

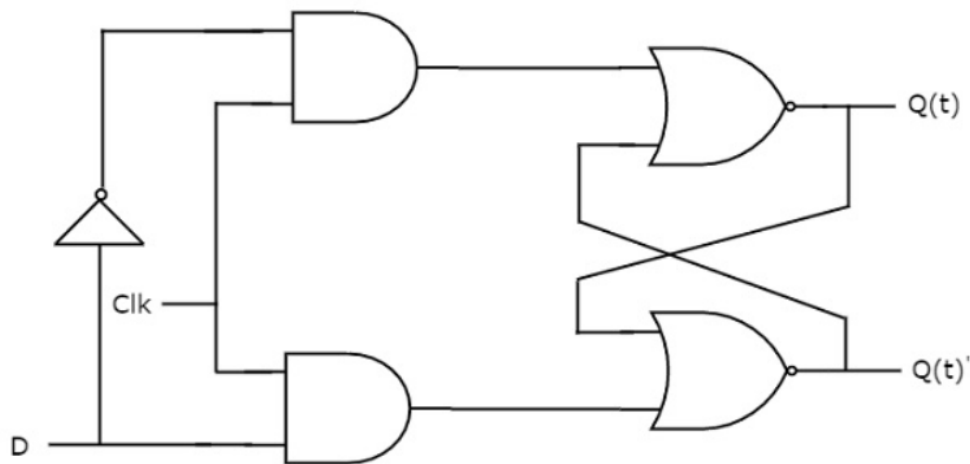
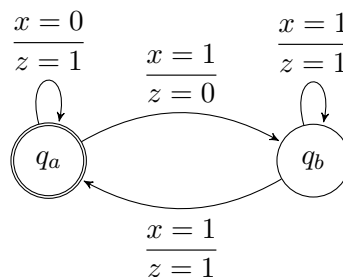


Figura 3.15: Un componente D-latch

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

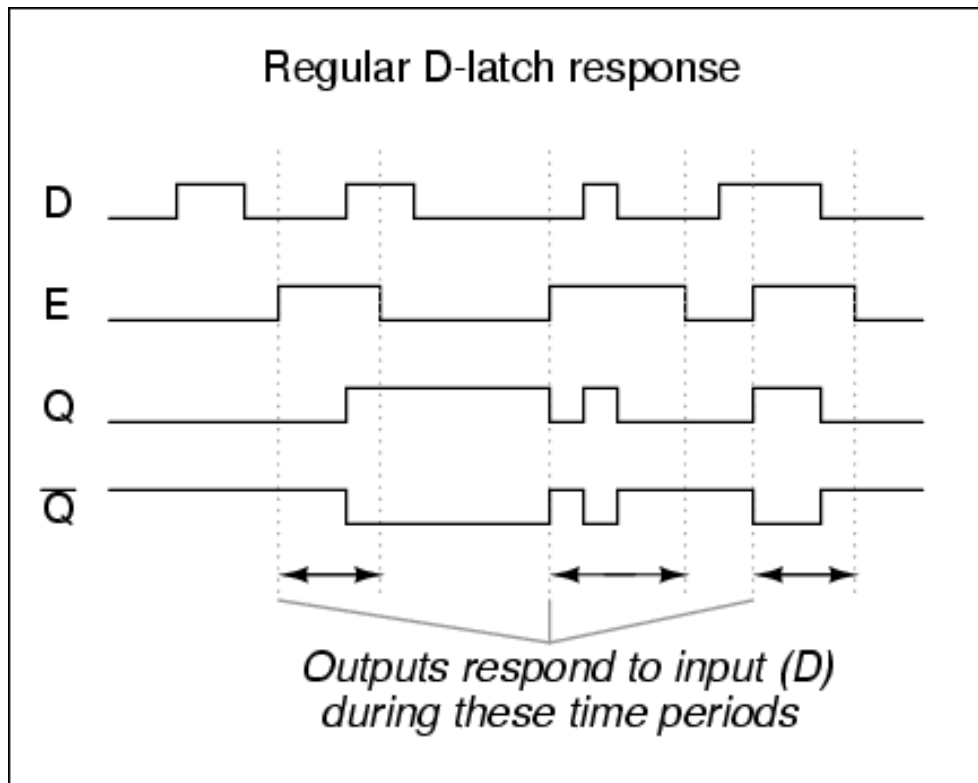


Figura 3.16: Diagramma temporale di un D-latch

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

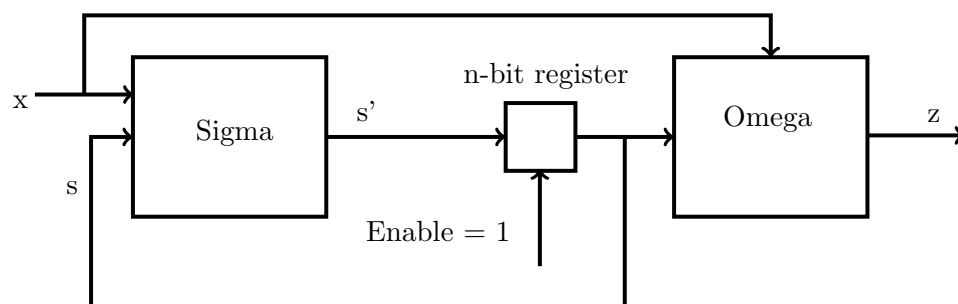


Figura 3.18: Rete sequenziale dell'automa di Mealy in figura precedente

Capitolo 4

Reti Sequenziali, 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 eseguire una funzione booleana di 3-5 ingressi con 1 uscita, implementare un bit di memoria ed effettuare routing.

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

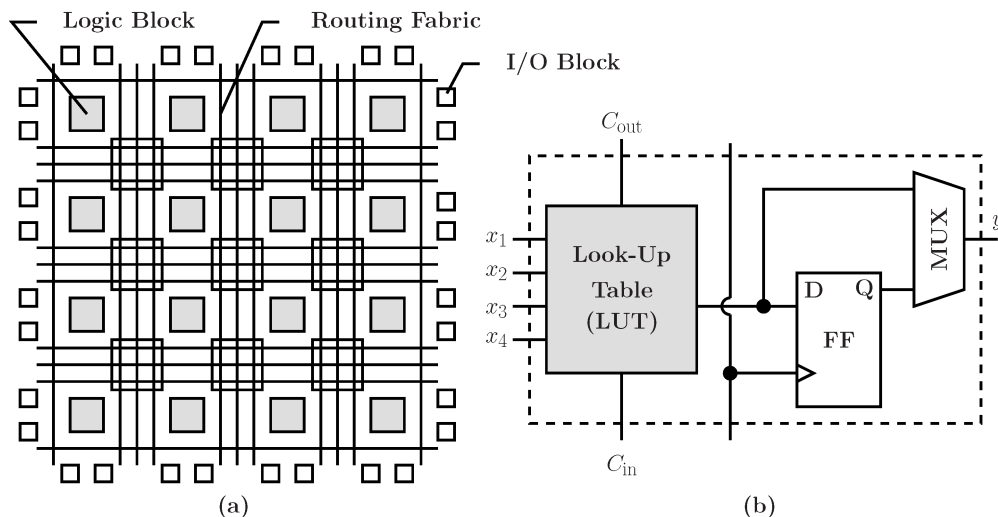


Figura 4.1: Schema FPGA

4.1 Scrivere e compilare System Verilog

Creiamo un multiplexer da due bit con System Verilog, compiliamo e visualizziamo con **GTKWave**

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

```



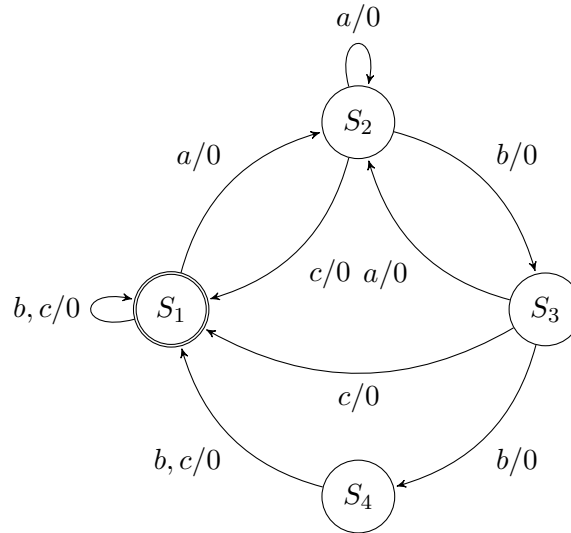
```
3    assign
4        z = ((~ic) & x) | (ic & y);
5
6    endmodule
```

4.2 Esercizi

4.2.1 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 di una rete sequenziale di un automa di mealy. 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 4.2: 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 4.1: 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 4.2: 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:

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

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

Nella componente **sigma** in Verilog avremo l'assegnamento ad un nuovo stato **news**:

```
1 assign
2     news[1] = s[2] & ~x[1] & x[2];
3
4 assign
5     news[2] = ~s[1] & ~x[1] & ~x[2]
6             | s[2] & ~x[1] & ~x[2]
7             | ~s[1] & s[2] & ~x[1];
```

La formula per l'uscita sarà $z = s_1\overline{s_2}\overline{x_1}x_2$. La componente **omega** effettuerà l'assegnamento

```
1 assign
2     z = s[1] & ~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.

Abbiamo 3 moduli. Uno per il registro, uno per il modulo ω e uno per il modulo σ . Scriviamolo in Verilog.

Note sulla sintassi e semantica Verilog La sintassi **[a:b]** denota un array indicizzato dal numero **a** al numero **b**. Utilizziamo gli array per rappresentare valori a più bit. In questo caso, il registro è generalizzato per un parametro **N** e può ad esempio essere inizializzato con **registro #(4) nome(...)** dove 4 è il parametro **N** e corrisponde al numero di bit del registro.

La negazione con **!** nega un singolo valore, mentre la notazione **~** viene detta negazione *bit wise* e nega tutti i valori di una sequenza di bit.

Normalmente, in un blocco **assign** assegno un valore ad una variabile booleana istantaneamente. Posso invece aggiungere un delay inserendo **#x** di fronte all'identificatore della variabile, dove **x** è un numero. Ad esempio:

```
1 assign
2     #1 z = (ic == 0 ? x : y)
```

Listing 4.5: Registro a N bit

```
1 module registro(output [1:N]z, input clock, input enable, input [1:N]inval);
2
3     parameter N = 2;
4
5     reg [1:N] stato;
6
7     initial
8         begin
9             stato = 0;
10        end
11
12    always @(posedge clock)
13        begin
```

```

14         if(enable)
15             stato <= inval;
16         end
17
18     assign
19         z = stato;
20
21 endmodule

```

Listing 4.6: Modulo di σ o funzione di cambio di stato

```

1 module sigma(output [1:2]news, input [1:2]s, input [1:2]x);
2
3     assign
4         news[1] = s[2] & ~x[1] & x[2];
5
6     assign
7         news[2] = ~s[1] & ~x[1] & ~x[2]
8             | s[2] & ~x[1] & ~x[2]
9             | ~s[1] & s[2] & ~x[1];
10
11 endmodule

```

Listing 4.7: Modulo di ω

```

1 module omega(output z, input [1:2]s, input [1:2]x);
2
3     assign
4         z = s[1] & ~s[2] & ~x[1] & ~x[2];
5
6 endmodule

```

Listing 4.8: Modulo della rete sequenziale

```

1 module m1(output z, input [1:2]x, input clock);
2
3     wire [1:2]outreg;
4     wire [1:2]inreg;
5
6     registro s(outreg, clock, 1'b1, inreg);
7     sigma sm(inreg, outreg, x);
8     omega om(z, outreg, x);
9
10 endmodule

```

Listing 4.9: Modulo della rete sequenziale (con Sincronizzazione)

```

1 module m1(output z, input [1:2]x, input clock);
2
3     wire [1:2] outreg;
4     wire [1:2] inreg;
5     wire [1:2] xsy;
6
7     registro #(2) r(xsy, clock, 1'b1, x);
8
9     registro #(2) s(outreg, clock, 1'b1, inreg);

```

```

10    sigma sm(inreg, outreg, xsy);
11    omega om(z, outreg, xsy);
12
13
14    endmodule

```

Listing 4.10: Modulo di test della rete sequenziale

```

1  module testm1();
2
3      reg [1:2] x;
4      reg      clock;
5      wire z;
6
7      m1 m(z, x, clock);
8
9      initial
10         clock = 0;
11
12      always
13         begin
14             #2 clock = 1;
15             #1 clock = 0;
16         end
17
18      initial
19         begin
20             $dumpfile("test-mealy.vcd");
21             $dumpvars;
22
23             x = 2'b00;
24             #4 x = 2'b01;
25             #5 x = 2'b01;
26             #1 x = 2'b00;
27
28
29             #10 $finish;
30         end
31    endmodule // testm1

```

Listing 4.11: Makefile

```

SOURCES = test-m1.v m1.v sigma.v omega.v reg.v
CC = iverilog
VIEW = gtkwave
EXECS = a.out Mealy.out

mealy: $(SOURCES)
    $(CC) $(SOURCES) -o Mealy.out

clean:
    rm -f $(EXECS)

run:    mealy
        ./Mealy.out

```

```
$(VIEW) test-mealy.vcd
```

Per sincronizzare il modulo della rete sequenziale dell'automa di Mealy mettiamo un registro di fra l'input x e le reti sequenziali σ e ω .

Figura 4.3: Diagramma dei tempi della Rete Sequenziale dell'automa di Mealy per riconoscere "abba" **senza sincronizzazione**

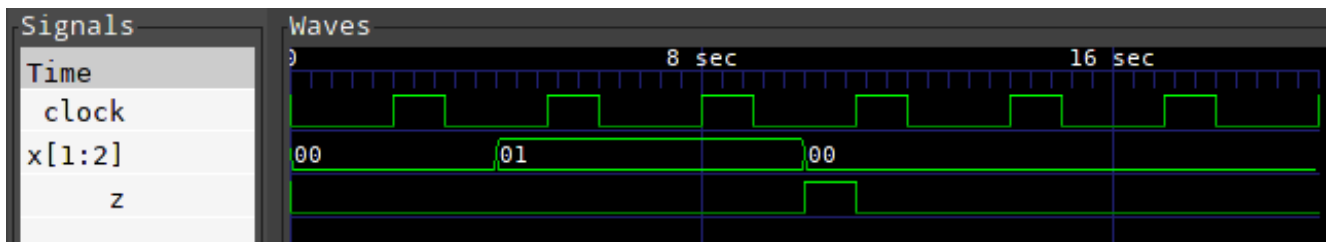
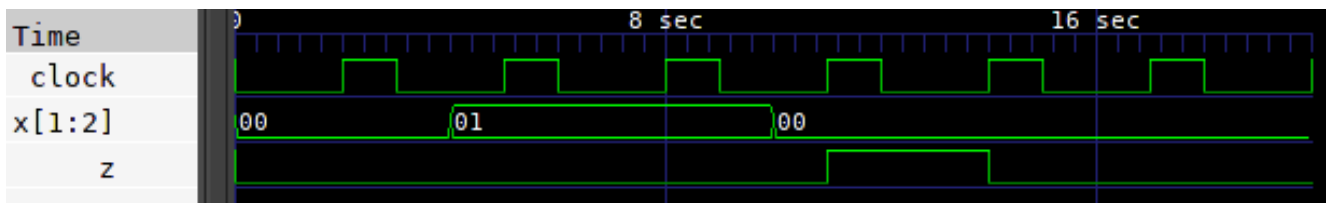


Figura 4.4: Diagramma dei tempi della Rete Sequenziale dell'automa di Mealy per riconoscere "abba" **con sincronizzazione**



4.2.2 Automa di Moore per riconoscere "abba"

Vediamo adesso un automa di Moore per riconoscere la stringa "abba" nell'alfabeto $\{a, b, c\}$. La differenza sta nel fatto che i singoli nodi non hanno accesso all'input x . Teniamo nota che la differenza fra un automa di Mealy di un automa di Moore, consiste che nell'automa di Moore la funzione ω non accetta riceve l'input

Figura 4.5: Automa di Moore che riconosce "abba"

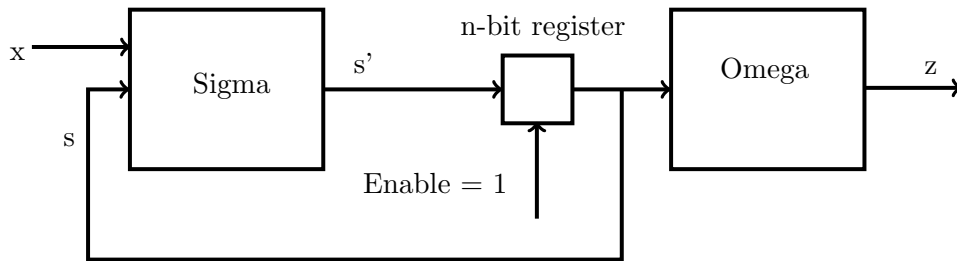
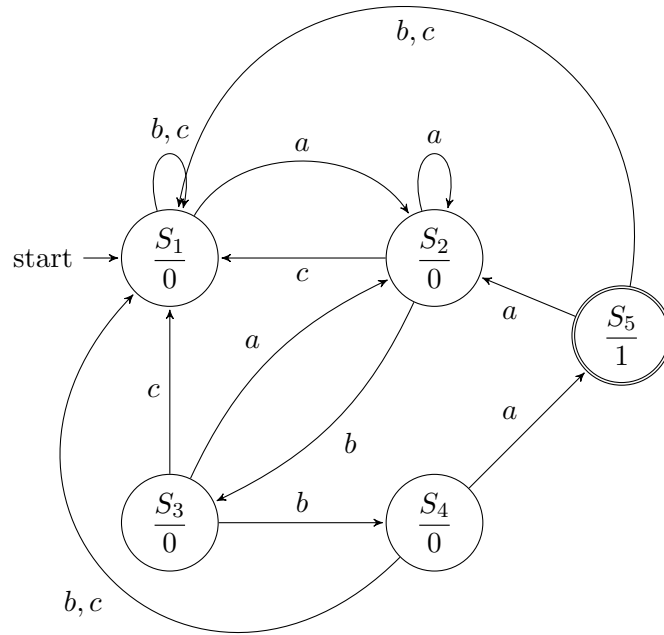


Figura 4.6: Rete Sequenziale dell'automa di Moore per riconoscere "abba"

Listing 4.12: Modulo di σ o funzione di cambio di stato (Moore)

```

1 module sigma(output [1:3]t, input [1:3]s, input [1:2]x);
2   // s[1] & s[2] & s[3] & x[1] & x[2]
3   assign
4     t[1] = ~s[1] & s[2] & s[3] & ~x[1] & ~x[2];
5   assign
6     t[2] = ~s[1] & ~s[2] & s[3] & ~x[1] & x[2] |
7           ~s[1] & s[2] & ~x[1] & x[2];
8   assign
9     t[3] = ~s[1] & ~s[2] & ~x[1] & ~x[2] |
10          ~s[1] & s[2] & ~s[3] & ~x[1] |
11          s[1] & ~s[2] & ~s[3] & ~x[1] & x[2];
12 endmodule

```

Listing 4.13: Modulo di ω (Moore)

```

1 module omega(output z, input [1:3]s, input [1:2]x);
2   // z = 1 solo se ci troviamo nello stato 5 (100)
3   assign
4     z = s[1]& ~s[2] &~s[3];
5
6
7 endmodule

```

Listing 4.14: Modulo della rete sequenziale (Moore)

```

1 module m1(output z, input [1:2]x , input clock);
2
3   wire [1:3]sinp;
4   wire [1:3]sout;
5
6   sigma sm(sinp, sout, x);
7   omega om(z, sout, x);
8   registro #(3) rs(sout, clock, 1'b1, sinp);
9
10
11 endmodule

```

Listing 4.15: Modulo di test della rete sequenziale (Moore)

```

1 module testm1();
2
3   reg [1:2] x;
4   reg      clock;
5   wire z;
6
7   m1 m(z, x, clock);
8
9   initial
10     clock = 0;
11
12   always
13     begin
14       #2 clock = 1;
15       #1 clock = 0;
16     end
17
18   initial
19     begin
20       $dumpfile("test-moore.vcd");
21       $dumpvars;
22
23       x = 2'b00;
24       #4 x = 2'b01;
25       #5 x = 2'b01;
26       #1 x = 2'b00;
27
28
29       #10 $finish;
30     end
31 endmodule // testm1

```


4.2.3 Mealy con Delay \equiv Moore

Vedremo come in termini di esecuzione, introducendo dei delay nell'automa di Mealy otteniamo un comportamento analogo all'automa di Moore.

Listing 4.16: Modulo di σ o funzione di cambio di stato (Mealy con Delay)

```

1 module sigma(output [1:2]news, input [1:2]s, input [1:2]x);
2
3   assign
4     #1 news[1] = s[2] & ~x[1] & x[2];
5
6   assign
7     #2 news[2] = ~s[1] & ~x[1] & ~x[2]
8               | s[2] & ~x[1] & ~x[2]
9               | ~s[1] & s[2] & ~x[1];
10
11 endmodule

```

Listing 4.17: Modulo di ω (Mealy con Delay)

```

1 module omega(output z, input [1:2]s, input [1:2]x);
2
3   assign
4     #1 z = s[1] & ~s[2] & ~x[1] & ~x[2];
5
6 endmodule

```

Listing 4.18: Registro a N bit con delay

```

1 module registro(output [1:N]stato, input clock, input enable, input [1:N]inval);
2
3   parameter N = 2;
4
5   reg [1:N] st;
6
7   initial
8     begin
9       st = 0;
10    end
11
12   always @(posedge clock)
13     begin
14       if(enable)
15         st <= inval;
16     end
17
18   assign
19     #1 stato = st;
20
21 endmodule

```

Listing 4.19: Modulo della rete sequenziale (Mealy con Delay)

```

1 module m1(output z, input [1:2]x, input clock);
2
3   wire [1:2]outreg;

```

```

4    wire [1:2]inreg;
5
6    registro s(outreg, clock, 1'b1, inreg);
7    sigma sm(inreg, outreg, x);
8    omega om(z, outreg, x);
9
10   endmodule

```

Listing 4.20: Modulo di test della rete sequenziale (Mealy con Delay)

```

1  module testm1();
2
3      reg [1:2] x;
4      reg      clock;
5      wire z;
6
7      m1 m(z, x, clock);
8
9      initial
10         clock = 0;
11
12         // ciclo di clock tau = 4 (leggermente piu' grande del necessario (3))
13         always
14             begin
15                 // funz
16                 #3 clock = 1;
17                 // non funz
18                 // #1 clock = 1;
19
20                 #1 clock = 0;
21             end
22
23         initial
24             begin
25                 $dumpfile("test-mealy-delay.vcd");
26                 $dumpvars;
27
28                 x = 2'b00;
29                 #4 x = 2'b01;
30                 // #4 fa aba e non da' mai 1
31                 // #8 fa abba e mi da' l'uno
32                 // se metto il clock a 2 (1+1) non funziona piu'
33                 #8 x = 2'b01;
34                 // col clock a 1+1 non funz
35                 // #5 x = 2'b01;
36
37                 #1 x = 2'b00;
38
39
40                 #10 $finish;
41             end
42     endmodule // testm1

```

4.3 Componenti per il Calcolo

Nel capitolo precedente abbiamo visto gli **adder**, che accettano due bit e restituiscono bit sommato e riporto, e abbiamo visto i **full adder**, che accettano in input anche un riporto per poter essere composti a cascata. Tutti i tempi di stabilizzazione dei full adder si sommano l'uno con l'altro. Ciò è chiaramente un problema perché se dobbiamo sommare due numeri da 32 bit dobbiamo aspettare la stabilizzazione di 32 sommatore. Oppure, sommando due numeri da 4 bit avremo $4 \cdot 2\Delta t = 8\Delta t$. Per ovviare al problema

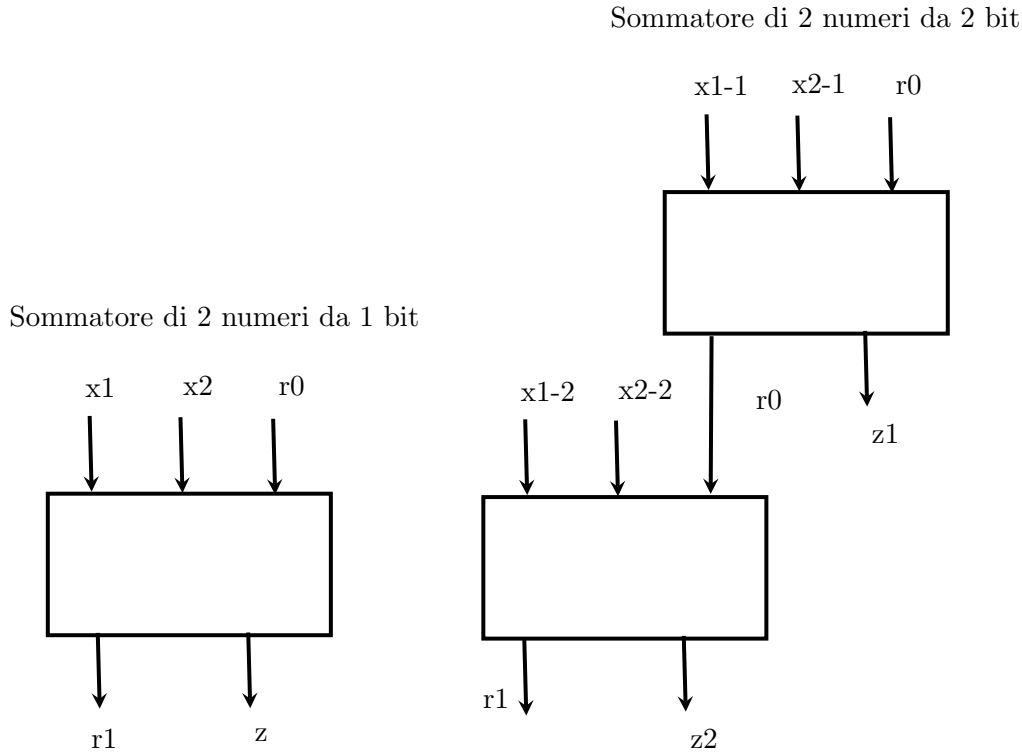


Figura 4.7: Sommatore di due numeri

possiamo realizzare $n + 1$ tabelle di verità (ognuna per ogni bit di input, più il riporto che indica se è avvenuto un overflow). Le tabelle di verità avranno 2^{2n} righe. Ad esempio, per sommare due numeri da 4 bit avremo 1 livello AND e 3 livelli OR perché usiamo porte che accettano al massimo 8 entrate. Ciò impiegherà (per numeri da 4 bit) $4\Delta t$ a differenza della somma con Full Adder a cascata che impiega $8\Delta t$.

Se volessi sommare numeri da 8 bit, la situazione varia leggermente. Servono 2 livelli di AND, e per esprimere la tabella di verità servono al minimo 2^{15} righe. I livelli di OR saranno quindi $\lceil \frac{\log_2 2^{15}}{\log_2 2^3} \rceil = \lceil \frac{15}{3} \rceil$.

ALU con sottrattore Possiamo realizzare la sottrazione in questo modo (funziona anche per i numeri negativi se utilizziamo il complemento a due):

1. Vogliamo sottrarre un numero $B = 4 \equiv 00000100$ ad un altro numero $A = 12 \equiv 00001100$:
 $A - B = 8 \equiv 00001000$
2. Invertiamo B e aggiungiamo 1: $(\bar{B} + 1) \equiv 11111100$
3. Otteniamo che $A - B \equiv A + (\bar{B} + 1) \equiv 000001000$ (non va considerato l'overflow)

Un possibile (ma errato) circuito che realizza sottrazione e addizione può essere il seguente:

Si può semplificare molto il circuito utilizzando un ALU per la somma che accetta un riporto iniziale come i full adder a cascata.

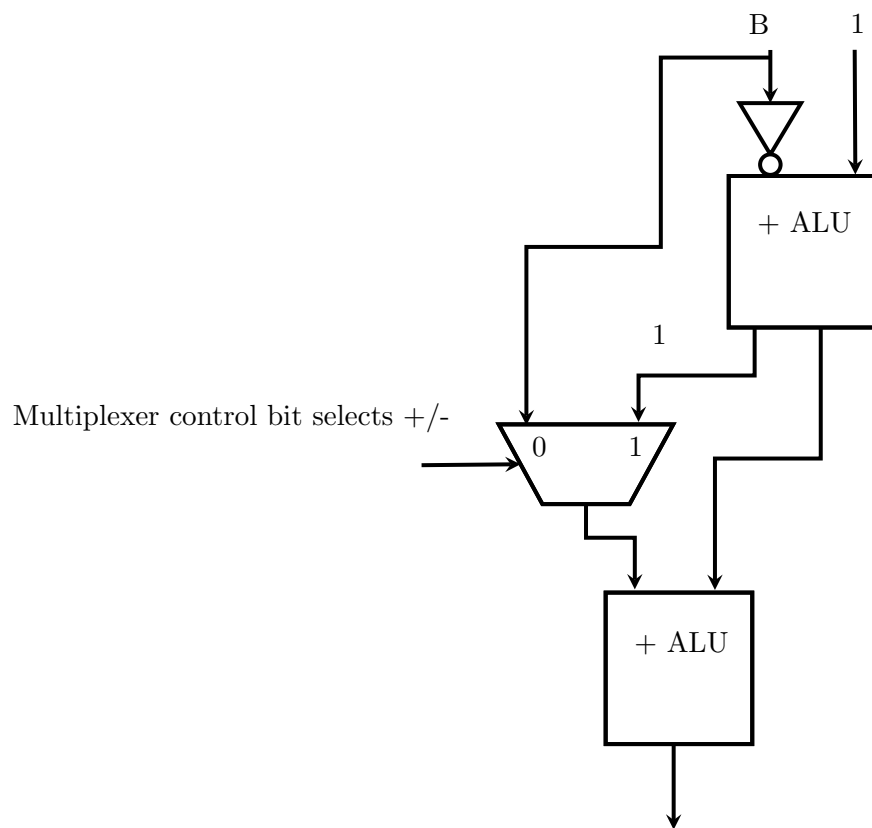


Figura 4.8: ALU per addizione e sottrazione

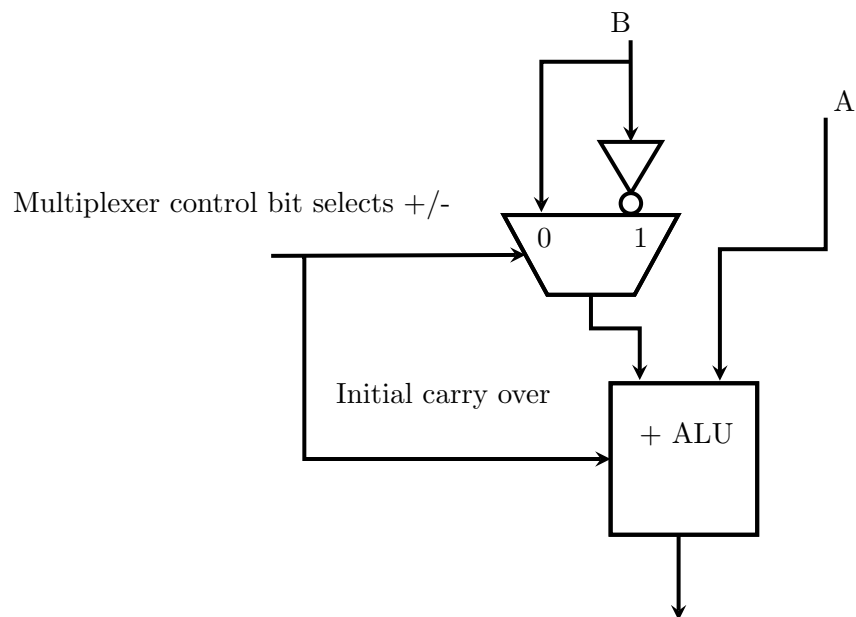


Figura 4.9: ALU per addizione e sottrazione semplificata

Uscite aggiuntive delle ALU o flag Le ALU, oltre all'uscita del risultato possono avere delle uscite aggiuntive, dette **flag**. I flag più comuni sono il flag **Zero**, il flag **Negative** ed il flag **Carry**.

Il flag **Z** (Zero) è 1 se tutti i bit del risultato sono 0. Ovvero $Z = \forall i. (out_i == 0)$. Si può realizzare inserendo nella alu un gate NOR che riceve in entrata tutti i bit dell'output.

Il flag **N** (Negative) è semplicemente il bit più significativo (a sinistra) se utilizziamo la notazione a complemento a due.

Il flag **C** (Carry) è usato per indicare se un riporto è stato generato nell'output dal bit più significativo. Permette di realizzare ALU composte da altre ALU, con dimensioni di parole in bit maggiori di quella della singola unità aritmetica. Il bit di carry della ALU che calcola la sezione LOW della parola viene inserito come riporto iniziale della ALU che calcola la parte HIGH della parola.

				1	0	1	1
			×	1	1	0	1
		(1)		1	0	1	1
	(1)	0		0	0	0	
(1)	1	0		1	1		
1	0	1		1			
1	0	0	0	1	1	1	1

Tabella 4.3: Moltiplicazione binaria $11 \times 13 = 143$

Moltiplicatore La moltiplicazione binaria avviene in maniera analoga alla normale moltiplicazione in colonna. Ciò ci permette di visualizzare rapidamente un possibile circuito realizzato da moltiplicazione e somma dei risultati. È facile osservare che la moltiplicazione fra due singole cifre binarie è banalmente il gate AND. Per realizzare un circuito moltiplicatore sarà necessario quindi una "matrice" di gate AND e dei sommatore in fondo. Per l'ultimo ed il primo bit della somma saranno necessari solamente degli Half Adder, perché non accettano un riporto in input.

Listing 4.21: Half Adder in Verilog

```

1 module ha(output c, output z, input x, input y);
2
3     assign
4         c = x&y;
5
6     assign
7         z = ~x&y | x&~y;
8
9 endmodule //fa

```

Listing 4.22: Full Adder in Verilog

```

1 module fa(output c, output z, input x, input y, input r);
2
3     assign
4         c = ~x&y&r | x&~y&r | x&y&~r | x&y&r;
5
6     assign
7         z = ~x&~y&r | ~x&y&~r | x&~y&~r | x&y&r;
8
9 endmodule //fa

```

Listing 4.23: mul4bit.v

```

1 module mul(output [7:0]z, // 8 bit number result

```

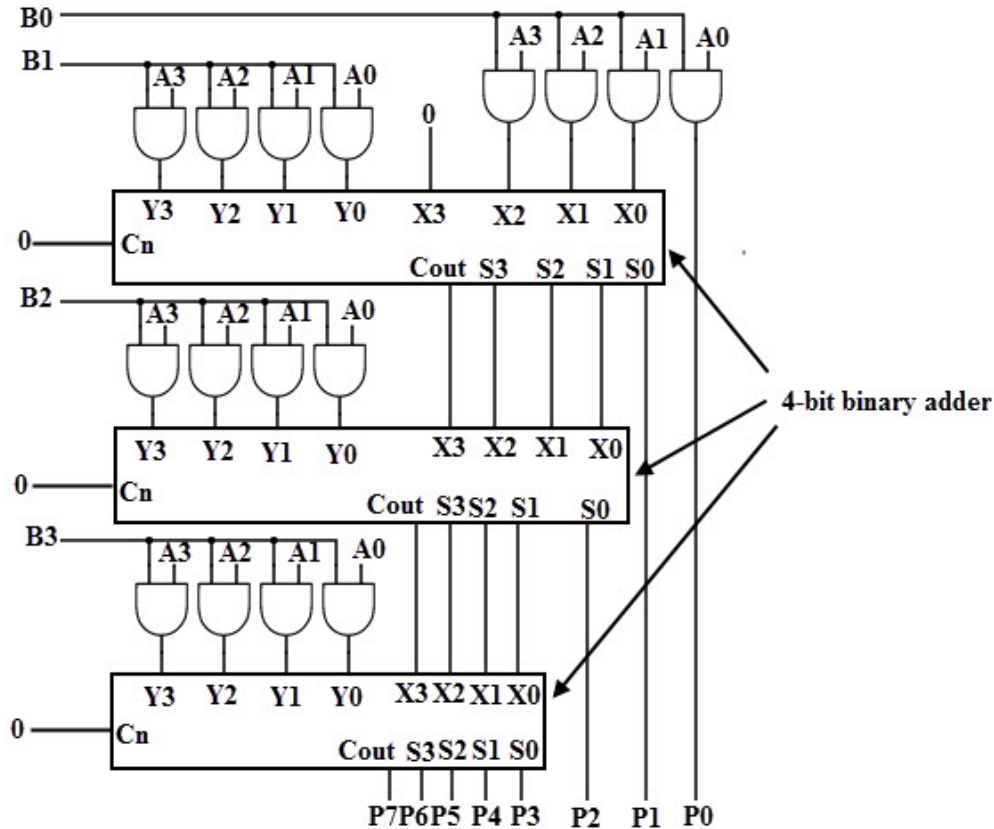


Figura 4.10: Circuito di un moltiplicatore interno ad una ALU di due numeri da 4 bit A e B

```

2  input [3:0] x, // 2 4-bit numbers input
3  input [3:0] y;
4
5  wire [11:0] c; // carry wire
6  wire [6:1] w; // adder result wire
7
8  assign
9      z[0] = x[0]&y[0]; // first bit of output (least significant)
10     // is just AND between the first bits of the two inputs
11
12     // first row of adders
13     //carryout  output  addend1      addend2      carryin
14     ha h1(c[0],    z[1],  x[1]&y[0],  x[0]&y[1]); // second bit of output
15     fa f2(c[1],    w[1],  x[2]&y[0],  x[1]&y[1], c[0]);
16     fa f3(c[2],    w[2],  x[3]&y[0],  x[2]&y[1], c[1]);
17     ha h4(c[3],    w[3],  x[3]&y[1],  c[2]);
18
19     // second row of adders
20     ha h5(c[4],    z[2],  w[1],      x[0]&y[2]); // third bit of output
21     fa f6(c[5],    w[4],  w[2],      x[1]&y[2], c[0]);
22     fa f7(c[6],    w[5],  w[3],      x[2]&y[2], c[1]);
23     fa f8(c[7],    w[6],  c[3],      x[3]&y[2], c[6]);
24
25     // third row of adders
26     ha h9(c[8],    z[3],  w[4],      x[0]&y[3]); // third bit of output
27     fa f10(c[9],   z[4],  w[5],      x[1]&y[3], c[8]);
28     fa f11(c[10],  z[5],  w[6],      x[2]&y[3], c[9]);

```

```
29     fa f12(z[7],    z[6],    c[7],    x[3]&y[3], c[10]);
30     // on last bit, carry out is the last bit of output
31
32 endmodule //mul4bit
```


Capitolo 5

Memorie e Parallelismo

5.1 Memorie

Un componente fondamentale di un processore è la **memoria**. Si implementa utilizzando dei registri da n bit (dette parole). Vogliamo realizzare una memoria il cui accesso sia simile ad un array nel linguaggio C. Ovvero vogliamo avere un vettore di registri a cui possiamo accedere con un offset intero. Data una memoria A si accederà alla parola k –esima tramite $A[k]$. Per implementare tale selezione si può utilizzare un multiplexer che riceve in input le uscite dei singoli registri. I singoli registri ricevono tutti un segnale di clock in input, e permettono la scrittura attraverso un demultiplexer (decoder) che dato un indirizzo invia un 1 all'enable del registro selezionato. Si può mettere in AND il segnale enable del decoder insieme ad un input enable generale.

L'implementazione reale delle memorie (in particolare memorie flash), avviene ponendo delle linee dette "di parola" (orizzontali) e linee dette "bit line" (verticali) in una matrice. All'incrocio delle linee vi è un condensatore. Il condensatore mantiene una carica per un breve periodo di tempo. Ho modo di controllare se nel condensatore alla posizione xy era "ricordato" un valore 1, mandando della corrente lungo la "linea di parola" x . Se il condensatore era carico esso si scarica lungo la linea "bit line" y . Inviando della corrente sulla linea di parola x leggeremo quindi dalle bit line (in alcuni casi indicate come source lines in output) la parola x . Prima di un condensatore, si inserisce un transistor nella giunzione fra WL e BL. Utilizzando un condensatore, nelle RAM dinamiche (Dynamic Random Access Memory) possono insorgere problemi dovuti ai tempi di carica e scarica del condensatore. Una volta letto un valore su una Bit Line, il condensatore si scarica e "perde" il valore precedentemente "ricordato".

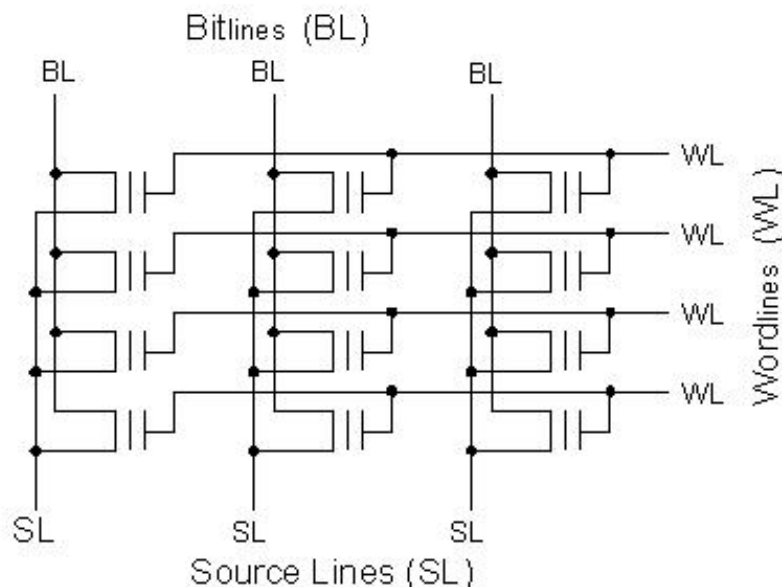


Figura 5.1: Memoria Flash

Per selezionare la Word Line utilizziamo sempre un demultiplexer (decoder) che riceve un indirizzo. Nelle ram DDR comuni sono presenti spesso 8 o 9 chip. Ad esempio, su uno stick di ram DDR3 da 1GB (1 Giga Byte) sono presenti 8 memorie da 1Gb (1 Giga Bit), e generalmente anche una nona memoria che ricorda la "parità" per il controllo degli errori. Le RAM dinamiche perdono il loro contenuto se non sono alimentate. Visto il comportamento dei condensatori (la loro carica si perde lentamente) le DRAM hanno bisogno di un **refresh** periodico. Le DRAM sono molto economiche (1 transistor per 1 bit) ma possono risultare più lente rispetto alle alternative SRAM. Le memorie SRAM (Static RAM) sono meno economiche ma più rapide rispetto alle DRAM. Utilizzano flip-flop invece dei condensatori e per ogni bit utilizzano 6 transistor. Le memorie all'interno del processore (calcolatore) sono memorie statiche (realizzate con flip-flop) molto rapide. Un meccanismo sia a livello hardware che a livello di S.O. permette la suddivisione dei dati processati fra vari livelli di cache, dalla memoria statica del processore alla più capiente ma lenta DRAM.

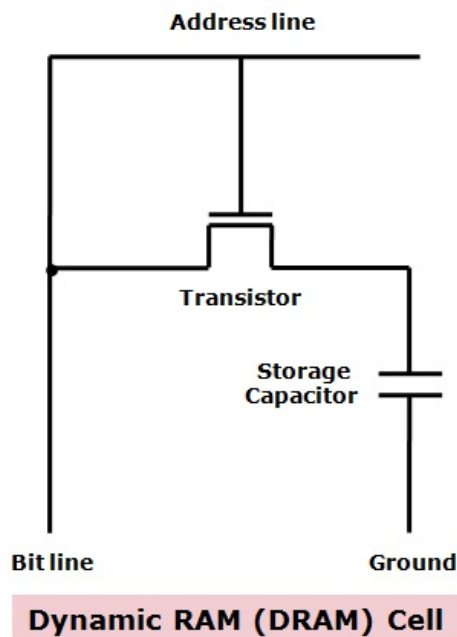


Figura 5.2: Schema di una cella di una memoria DRAM

Definizione 5.1.1. RAM, ROM, PROM e EEPROM: L'acronimo RAM (Random Access Memory) indica in generale delle memorie sulle quali possiamo leggere e scrivere. Le memorie ROM sono memorie a sola lettura (Read Only Memory). Le memorie ROM non hanno condensatori nelle giunzioni della matrice, ma a momento di produzione vengono "incisi" dei collegamenti o isolamenti nelle giunzioni fra Word Line e Bit Line. Le memorie **PROM** utilizzano dei fusibili nelle giunzioni fra WL e BL per permettere agli acquirenti di scrivere nella memoria una volta sola. Di fabbrica tutti i bit di una PROM sono impostati a 1. Bruciando i fusibili nelle giunzioni si interrompono i collegamenti realizzando una vera e propria scrittura permanente.

Un altro tipo di memorie sono le **EEPROM** (Electrically Erasable ROM). Utilizzano dei transistor particolari che possono essere "reimpostati" permettendo di cancellare e riscrivere la memoria con una corrente maggiore.

Nella seconda parte del corso vedremo dettagliatamente Assembly ARM. Le istruzioni Assembly eseguono delle operazioni sui registri interni alla CPU. Un esempio di istruzione banale è la somma di numeri fra registri. `ADD R1 R2 R3`. Nell'Assembly ARM la lettura avviene da destra a sinistra, ed il risultato viene memorizzato nel registro più a sinistra, in questo caso R1.

Memorie Multi Porta e Tempi di Accesso Le memorie multiporta permettono la lettura parallela di due o più registri. Invece di un singolo multiplexer per leggere un registro, alle uscite dei registri è collegato un altro multiplexer che ricevendo un indirizzo diverso dal primo multiplexer, permette di leggere

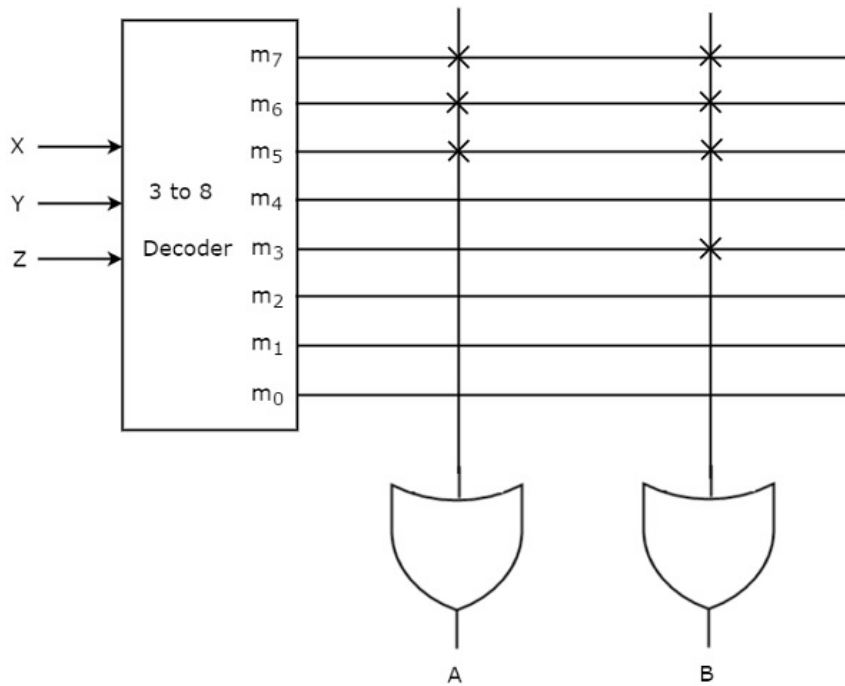


Figura 5.3: Esempio di memoria PROM

un altro valore dalla memoria, abilitando alla lettura parallela di due registri contemporaneamente. È fondamentale avere una memoria con almeno due porte per eseguire istruzioni su due registri alla volta. Indichiamo i tempi di accesso alla memoria con T_a . Ad oggi siamo in un range di tempi inferiori ad un nanosecondo per i tempi di accesso ai flip-flop (poche decine di picosecondi), sull'ordine dei nanosecondi per le SRAM (cache) e sull'ordine dei microsecondi per le DRAM.

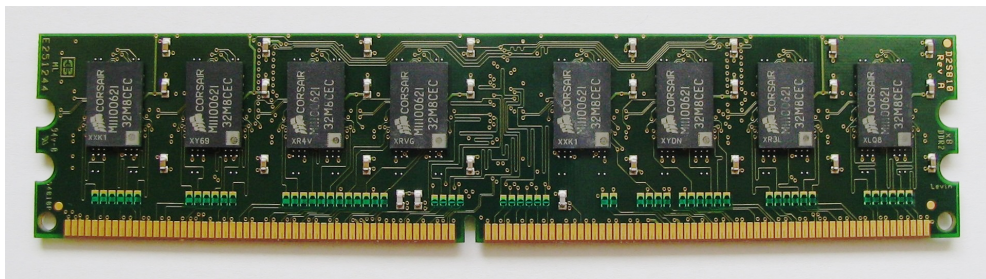


Figura 5.4: Modulo RAM Corsair DDR2-533

Blocco memoria Introduciamo un blocco per la memoria da utilizzare nei diagrammi dei circuiti. La memoria riceve in input un dato x da scrivere nell'indirizzo indicato dall'input di controllo **Address**. Se il segnale **enable** è HIGH allora avviene la scrittura di x all'interno dell'indirizzo puntato. Altrimenti, se **enable** è LOW avviene la lettura. Assumiamo che l'uscita sia stabile per il ciclo di clock

LUT (Lookup Table) Una rete combinatoria che riceve n bit in input può essere implementata con una memoria ROM. Data una tabella di verità di una rete combinatoria, si può portare la tabella di verità in forma vettoriale per creare una LUT. Una LUT o **Lookup Table** è in generale un array che rimpiazza una computazione, possibilmente complessa, con un'operazione di accesso, nel caso delle reti combinatorie con una memoria ROM. I bit in input della tabella di verità vengono portati a bit rappresentanti l'indirizzo di accesso della memoria, e i risultati della tabella di verità saranno memorizzati in un vettore di 2^n elementi (se la tabella di verità ha un solo output). Per quanto alcune reti combinatorie possono risultare più complesse a livello di circuito se implementate con una LUT, un vantaggio non ba-

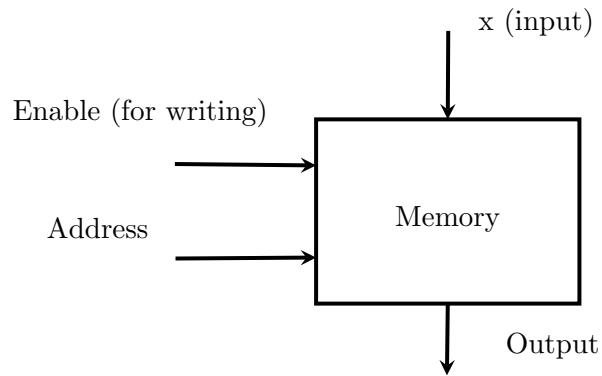


Figura 5.5: Blocco Memoria

nale delle Lookup Tables è che possono essere riprogrammate se viene utilizzata una memoria EEPROM. Ciò svolge un ruolo fondamentale negli FPGA. Le LUT possono essere utilizzate in parallelo.

Esempio 5.1.1. Rete Sequenziale di un contatore Se volessimo realizzare un contatore, con le sole operazioni di incremento e decremento con un ASF, dovremmo avere un numero di stati pari alla capacità massima del contatore. Per realizzare un contatore sono necessari un registro da n bit, (prendiamo ad esempio 8 bit) ed una ALU che fa due sole operazioni (+1 e -1), quindi con un solo bit di controllo. Scegliamo se aumentare o decrementare il contatore impostando il bit di controllo della ALU, e scegliamo se scrivere nel prossimo ciclo di clock impostando ad HIGH il bit Enable (Event) del registro. Essendo una rete sequenziale, la funzione σ è implementata dalla ALU, e la funzione ω è semplicemente l'identità. È una rete di Moore poiché la funzione identità non dipende dagli ingressi (Inc/Dec ed Event).

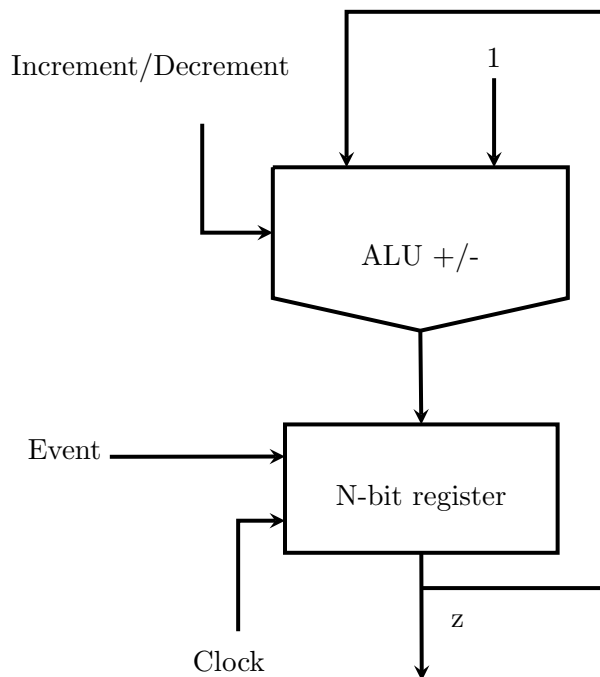


Figura 5.6: Rete sequenziale di un contatore

Esempio 5.1.2. Semplice Calcolatrice Vogliamo realizzare una calcolatrice molto semplice da 32 bit per eseguire addizione, sottrazione, moltiplicazione e divisione. La calcolatrice riceverà in input due numeri A, B e restituirà in output il risultato, che potrà essere re-inserito nei due registri contenenti A e B . Essa è una rete sequenziale che possiamo inserire in un componente che avrà in input i due numeri A, B , la selezione dei due multiplexer per gli input, segnali di enable per la scrittura nel registro A e nel registro B e due bit di controllo per selezionare l'operazione. In output avrà il numero in uscita e 4

bit di flag (Overflow, Carry, Negative e Zero). Definiamo questa parte del calcolatore "parte operativa". Abbiamo bisogno anche di una "parte di controllo" che dati due input si occuperà di controllare le entrate di controllo della parte operativa della calcolatrice.

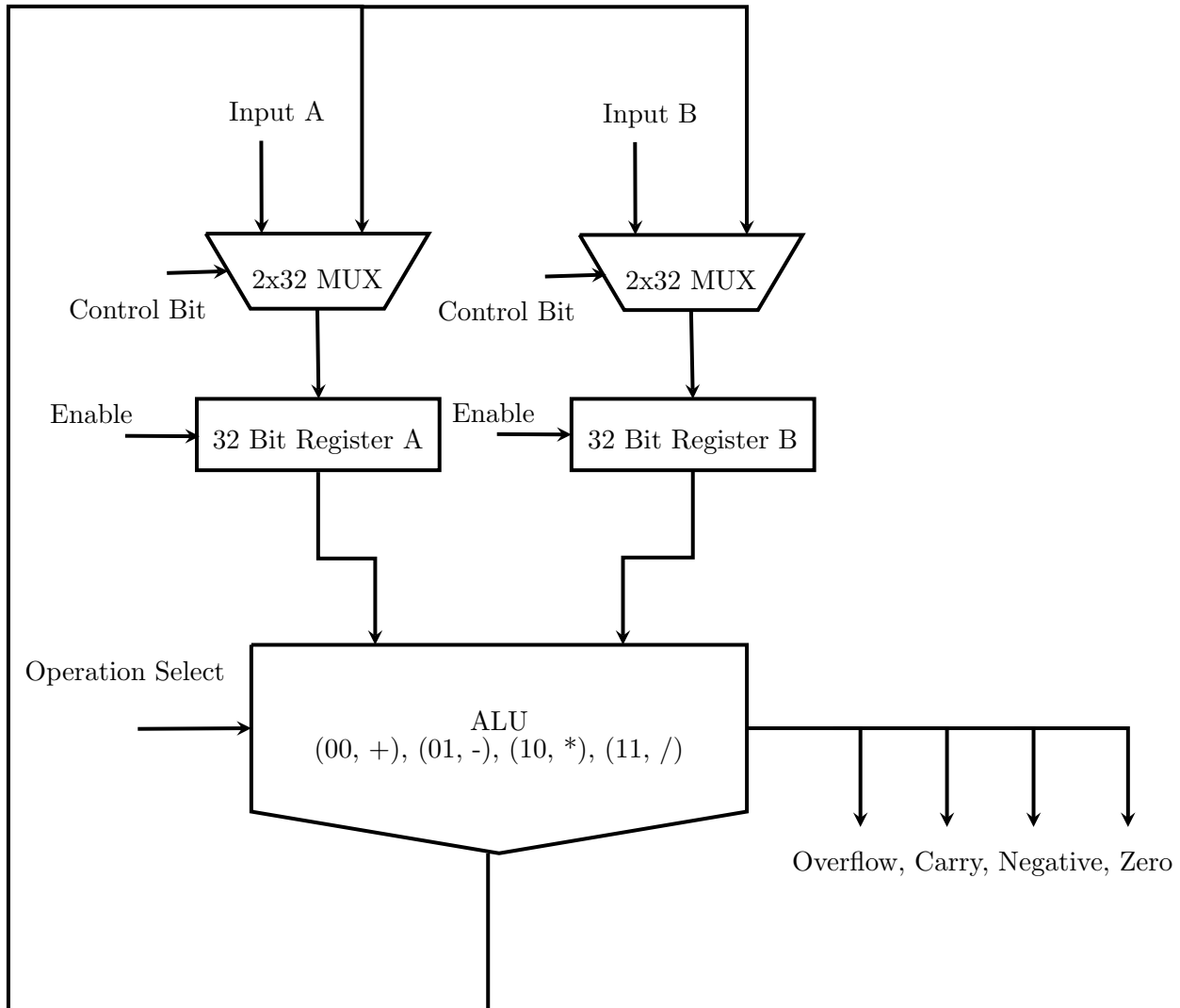


Figura 5.7: Semplice Calcolatore (Omesso clock per semplicità del disegno, è presente)

Definizione 5.1.2. if-then-else: Supponiamo di voler realizzare un importante costrutto per la programmazione (*if-then-else*) utilizzando solo componenti standard. Abbiamo due reti combinatorie f, g , una parola memorizzata in un registro A e un registro B in cui sarà memorizzato il risultato. Vogliamo costruire una rete per valutare l'espressione:

```

if(expr) then
    g(A) -> B
else
    f(A) -> B
  
```

Definizione 5.1.3. while: Se volessimo invece implementare un ciclo while della forma:

```

while(expr) do
    f(A) -> A
  
```

Possiamo realizzarlo con la seguente rete:

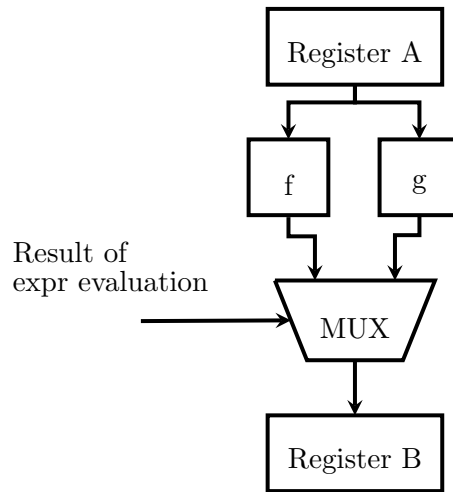


Figura 5.8: Rete che implementa if-then-else

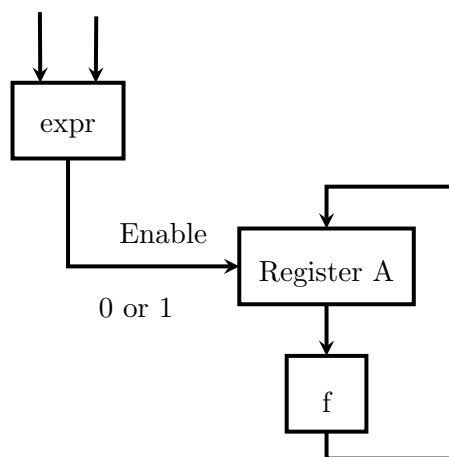


Figura 5.9: Rete che implementa while

In maniera simile è possibile realizzare anche un costrutto switch-case. Icarus Verilog, per trasformare programmi Verilog di moduli behavioural in netlist di componenti di reti logiche utilizza meccanismi simili.

Esercizio 5.1.3. Confronto maggiore

Vediamo ora diversi metodi per realizzare una rete combinatoria che controlla se un numero è maggiore di un altro, utile per realizzare operazioni condizionali.

Listing 5.1: Esercizio di confronto maggiore in Verilog (approccio algoritmico)

```

1 module maggiore(output reg gt, input [N-1:0] x, input [N-1:0] y)
2     parameter N = 2;
3     always @(x or y)
4         begin
5             if(x[1] > y[1])
6                 gt <= 1;
7             else
8                 begin
9                     if(y[1] > x[1])
10                        gt <= 0;
11                    else
12                        begin
13                            if(x[0] > y[0])
14                                gt <= 1;
15                            else
16                                gt <= 0;
17                        end
18                    end
19                end
20 endmodule //maggiore

```

5.1.1 Calcolatrice con registri di memoria

Realizziamo adesso una calcolatrice con sottrazione e addizione che permette di leggere due numeri in input e fare operazioni utilizzando anche numeri precedentemente memorizzati in degli altri registri. Abbiamo bisogno di risorse di calcolo (una ALU) e registri o memorie. Abbiamo bisogno di una memoria multiporta particolare per realizzare correttamente il circuito. Questa memoria accetta 3 indirizzi, i primi due sono per la lettura (ha due parole in output) e il terzo indirizzo è per la scrittura. È una rete sequenziale, per questo motivo abbiamo bisogno di un meccanismo per evitare i problemi di temporizzazione. Essendo una rete sequenziale il tempo di esecuzione sarà $\tau = \delta + \max\{t_\sigma, t_\omega\}$. Dovremmo prendere

un tempo del ciclo di clock $\tau = \delta + \max \left\{ \begin{matrix} t_{\text{MUX}} + t_{\text{ALU}}, \\ t_{\text{ALU}} + t_{\text{MUX}} \end{matrix} \right\}$ dove $t_\sigma = t_{\text{MUX}} + t_{\text{ALU}}$ e $t_\omega = t_{\text{ALU}} + t_{\text{MUX}}$

5.2 Parallelismo

Definizione 5.2.1. Supponiamo di avere una serie di libri L_1, L_2, L_3, L_4 da tradurre. Vogliamo ridurre il tempo totale di traduzione di tutti i libri. Se a tradurre un libro impiego un tempo da t_0 a t_1 , suddividendo la traduzione a metà fra due persone, impiegheremo $t_{1/2}$ a tradurre il libro. L'obiettivo del parallelismo è ridurre la **latenza**. Se ho un insieme di libri, la latenza L di un libro è il tempo necessario per tradurlo (il tempo a cui ho terminato la traduzione, meno il tempo in cui ho iniziato la traduzione). Se ho m task da completare, il **tempo di completamento** $T_c = m \cdot L$ è il tempo totale necessario a completare tutti i task m . Con il parallelismo vogliamo diminuire il tempo di completamento. Con **tempo di servizio** intendiamo l'intervallo medio di tempo fra la "consegna" di due risultati successivi. Definiamo anche **throughput** come il numero di calcoli eseguiti per unità di tempo.

Data una lista di istruzioni $i_0, i_1, i_2, i_3, \dots$ il nostro processore esegue (in pseudocodice):

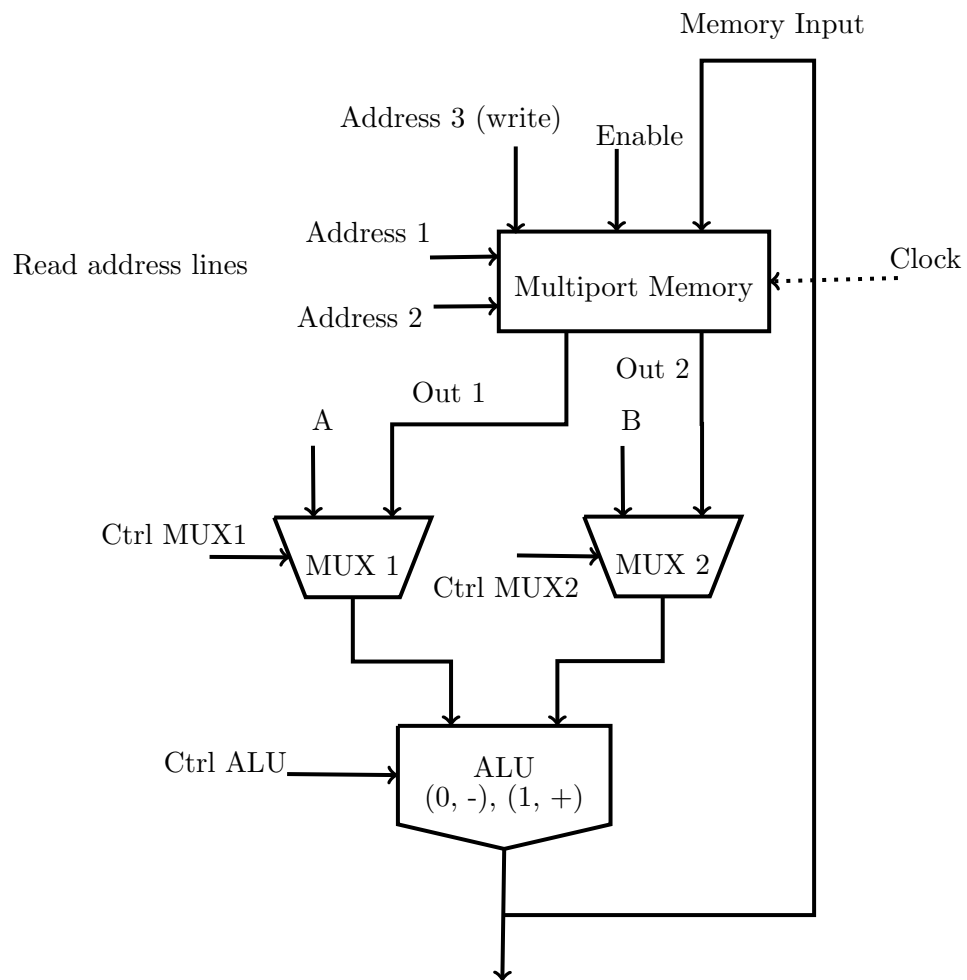


Figura 5.10: Calcolatrice con memoria.


```

while(true) {
  leggi un istruzione (ASM)
  interpreta
  esegui
}

```

Definizione 5.2.2. Tempo ideale: Dato n = numero di worker (o grado di parallelismo) e T_{seq} = il tempo sequenziale, il **tempo ideale** $T_{\text{id}}(n) = \frac{T_{\text{seq}}}{n}$.

Definizione 5.2.3. Speedup: Definiamo anche lo **speedup** come $\frac{T_{\text{seq}}}{T(n)}$, ovvero il miglior tempo sequenziale fratto il tempo parallelo con grado di parallelismo n .

Definizione 5.2.4. Scalabilità: $\text{Scalab}(n) = \frac{T(1)}{T(n)} \leq n$

Definizione 5.2.5. Efficienza: $\text{Efficienza}(n) = \frac{T_{\text{id}}}{T(n)} = \frac{T_{\text{seq}}/n}{T(n)} = \frac{T_{\text{seq}}}{n \cdot T(n)} \leq 1$

Definizione 5.2.6. Parallelismo temporale o Pipelining: Il processore deve sempre eseguire dei task (istruzioni) sequenzialmente per eseguire un programma, e si può diminuire la latenza dei singoli task per ridurre il tempo di esecuzione del programma.

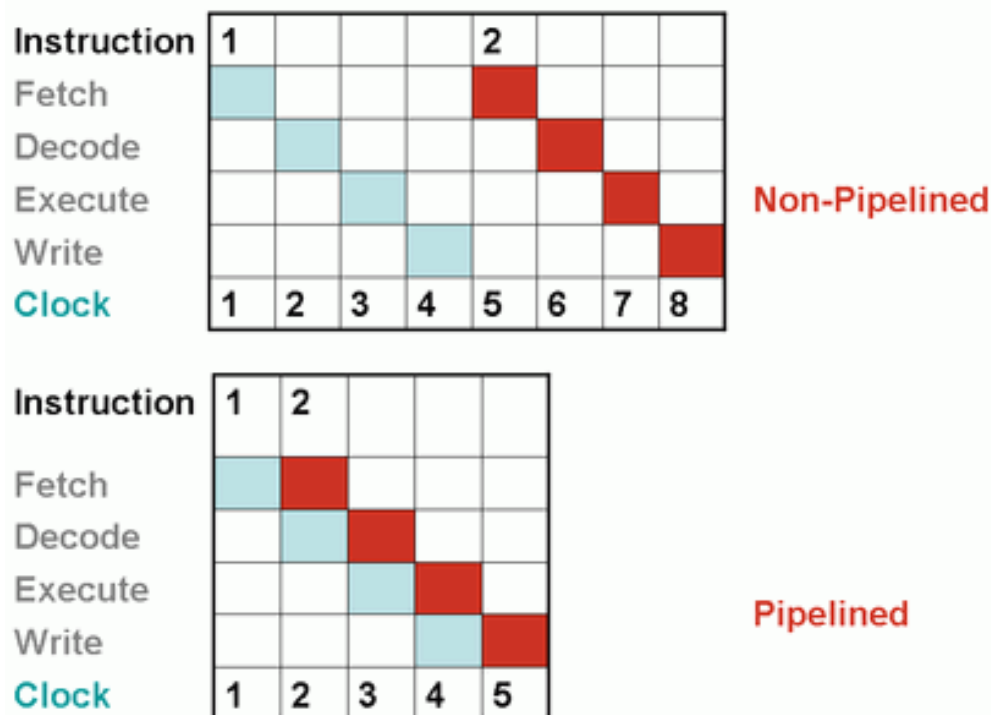


Figura 5.11: Diagramma temporale del pipelining di una CPU.

Avremo che T_c = tempo iniziale di "riempimento" della pipeline + tempo ($\propto (m \cdot$ tempo delle singole istruzioni))

In alternativa, il nostro processore può assegnare un "ciclo" ad ogni istruzione

```

// Per istruzione 0
while(true) {
  leggi, decodifica, esegui
}

```

```
// Per istruzione 1
while(true) {
leggi, decodifica, esegui
}
```

```
// Per istruzione 2...
while(true) {
leggi, decodifica, esegui
}
```

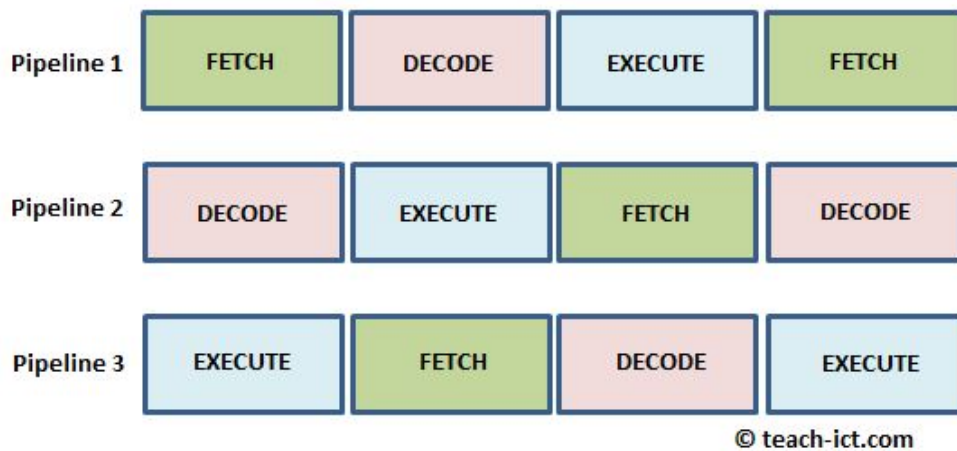


Figura 5.12: Pipelining con più worker (parallelismo temporale).

Usando il **parallelismo temporale**, ad ogni periodo L otterremo 3 risultati. Ciò riduce il tempo di servizio a $\frac{1}{3}L$. Le istruzioni dei task sono comunque eseguite sequenzialmente su worker separati.

Definizione 5.2.7. Parallelismo spaziale: Nel **parallelismo spaziale**, suddividiamo le istruzioni di ogni task su worker diversi. Significa che i worker possono eseguire in parallelo le istruzioni di un singolo task. Il parallelismo spaziale serve a diminuire la latenza, il parallelismo temporale serve invece ad aumentare il throughput.

Pipeline (stream parallel) Avendo 3 worker ed eseguendo 3 istruzioni a cascata che impiegano rispettivamente $2t, 3t, 1t$, se eseguiamo dei task successivamente, le istruzioni dei task successivi avverranno con un delay dovuto alla differenza del tempo di esecuzione. Dopo un numero di istruzioni le istruzioni si sincronizzeranno. Se i worker impiegano rispettivamente dei tempi t_1, t_2, t_3 , il tempo di completamento di m istruzioni sarà $T_c = \sum t_i + m \cdot \max\{t_1, t_2, t_3\}$. Ciò crea una situazione non ottimale

n = numero di stadi della pipeline

$$T_s = \max\{T_{S_i}\}$$

$\max(\text{speedup}(n)) = \#$ di stadi della pipeline

$$T_c(n) = \sum_i L_i + n - 1T_s$$

Map e Farm Farm è di tipo *stream parallel* mentre map è di tipo *data parallel*. Nel caso del **Map** (e anche del Farm), se le istruzioni impiegano un tempo t che viene precisamente suddiviso in $t/2$ allora il problema non sussiste. Se invece vengono suddivise in tempi diversi si creano spazi temporali in cui non

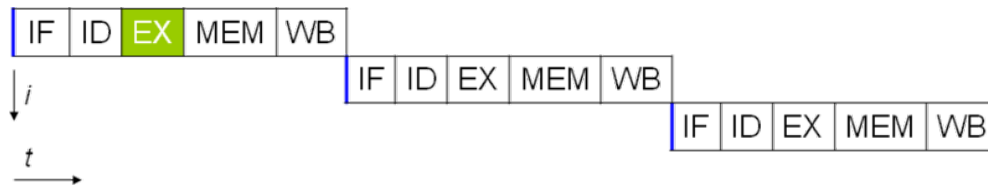


Figura 5.13: Processore senza pipeline (sottoscalare)

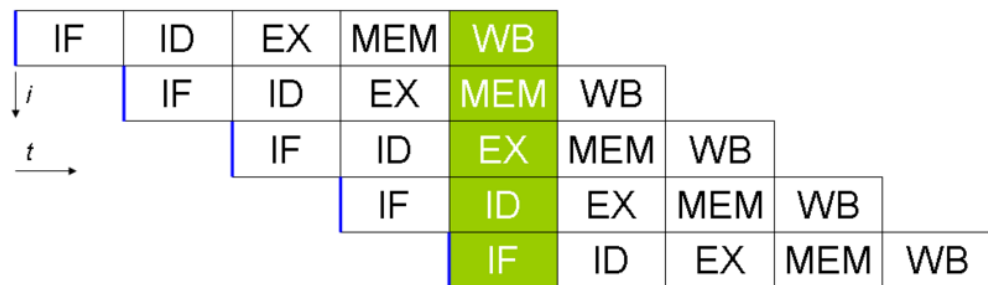


Figura 5.14: Processore con pipeline a cinque stadi (scalare)

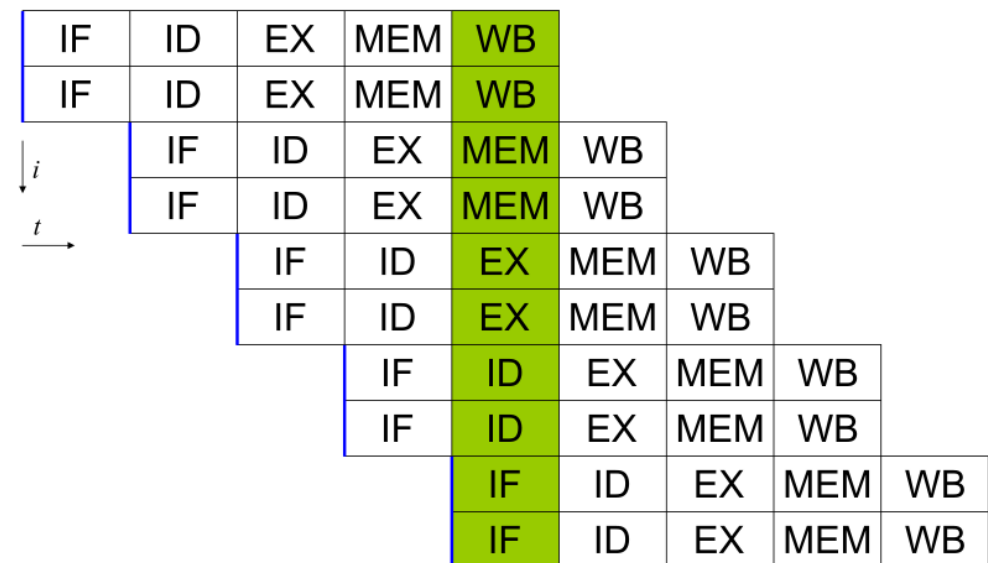


Figura 5.15: Processore con pipeline superscalare

è possibile sfruttare tutta la potenza di calcolo. Nella map posso suddividere un dato in arrivo e fare del calcolo in parallelo su di esso (*data parallelism*). Nella farm e nella pipeline i dati in arrivo possono essere passati a diversi worker (verranno schedulati) che eseguono computazioni in parallelo (*stream parallelism*). I dati non possono essere suddivisi nello *stream parallelism* come nel *data parallelism*.

Tempi della Farm

$$nw = \text{grado di parallelismo}$$

$$T_w = \text{il tempo che 1 worker impiega per calcolare 1 risultato}$$

$$T_s = \max \left\{ T_{\text{sched}}, \frac{T_w}{nw}, T_{\text{coll}} \right\}$$

Tempi della Map

$$T_c = T_{\text{split}} + \frac{m}{nw} T_f + T_{\text{merge}}$$

$$T_s = \max \left\{ T_{\text{split}}, T_{\text{merge}}, \frac{m}{nw} t_f \right\}$$

5.2.1 Processore superscalare

Scriviamo un loop di esecuzione del processore in maniera più accurata.

```
while(true) {
I = fetch(PC) // PC = program counter
decode(I)
exec(I)
|--> update(PC)
}
```

Composizione nel parallelismo Possiamo comporre i diversi metodi di parallelismo. Ad esempio si può realizzare una pipeline che contiene una farm. Prendiamo ad esempio

$$\text{pipe}(S_1, \text{farm}(S_2), S_3)$$

Un possibile schema dei worker della pipeline è

$$\rightarrow f \rightarrow \begin{cases} (\text{farm}) \\ g \\ g \end{cases} \rightarrow h$$

Calcoliamo i tempi

$$T_s = \max \{ T_{S_1}, T_{S_2}, T_{S_3} \}$$

$$T_s = \max \left\{ T_f, \max \left\{ T_{\text{sched}}, \frac{T_g}{nw}, T_{\text{coll}} \right\}, T_h \right\}$$

$$= \max \left\{ T_f, T_{\text{sched}}, \frac{T_g}{nw}, T_{\text{coll}}, T_h \right\}$$

Definizione 5.2.8. Bottleneck o Colli di Bottiglia

Supponiamo di avere due computazioni composte $f \circ g$. Se f impiega $5t$ e g impiega $2t$, eseguire in successione $\rightarrow f \rightarrow g \rightarrow$ può causare dei bottleneck (rallentamenti). Se f è data parallel si può suddividere la sua computazione in parti uguali (ad esempio due) e poi passare il risultato a g . In alternativa si può dividere f in 5 parti uguali e ad ogni parte calcolare la composizione sequenziale (in una farm) in modo che $T_s = \max \left\{ T_{\text{sched}}, \frac{T_w}{nw}, T_{\text{coll}} \right\} \approx \frac{T_w}{nw} \approx \frac{7t}{5} \approx 1$

Esempio 5.2.1. Pipeline in Verilog

Vediamo adesso una pipeline per parallelizzare l'operazione di moltiplicazione per 2 composta due volte.

Listing 5.2: Modulo moltiplicazione per 2

```

1 module per2(output [N-1:0]out, input [N-1:0]in);
2
3     parameter N = 8;
4
5     assign
6         #2 out = in*2;
7
8 endmodule // per2

```

Listing 5.3: Modulo moltiplicazione per 4

```

1 module per4(output [N-1:0]out, input [N-1:0]in, input clock);
2
3     parameter N = 8;
4
5     wire [N-1:0] pass;
6     reg [N-1:0]  ingresso;
7
8     per2 p1(pass, ingresso);
9     per2 p2(out, pass);
10
11     always @(posedge clock)
12         ingresso = in;
13
14 endmodule // per2

```

Listing 5.4: Programma di test

```

1 module test();
2
3     parameter N = 8;
4
5     reg [N-1:0] in;
6     reg         clock;
7
8     wire [N-1:0] out;
9     integer     i;
10
11     per4 p4(out,in,clock);
12
13     always
14     begin
15         #3 clock = 1;
16         #1 clock = 0;
17     end
18
19     initial
20     begin
21         clock = 0;
22         in = 0;

```

```

23
24     $dumpfile("test.vcd");
25     $dumpvars;
26
27     #3 in=0;
28
29     for(i=1; i<9; i=i+1)
30         #4 in = i;
31
32     #10 $finish;
33
34     end
35 endmodule // test

```

Listing 5.5: Modulo moltiplicazione per 2 (pipelined)

```

1 module per2(output [N-1:0]out, input [N-1:0]in);
2
3     parameter N = 8;
4
5     assign
6         #2 out = in*2;
7
8 endmodule // per2

```

Listing 5.6: Modulo moltiplicazione per 4 (pipelined)

```

1 module per4(output [N-1:0]out, input [N-1:0]in, input clock);
2
3     parameter N = 8;
4
5     wire [N-1:0] pass;
6     stadio s1(pass, in, clock);
7     stadio s2(out, pass, clock);
8
9 endmodule // per2

```

Listing 5.7: Programma di test (pipelined)

```

1 module test();
2
3     parameter N = 8;
4
5     reg [N-1:0] in;
6     reg        clock;
7
8     wire [N-1:0] out;
9     integer    i;
10
11     per4 p4(out,in,clock);
12
13     always
14     begin
15         #1 clock = 1;
16         #1 clock = 0;

```

```

17     end
18
19     initial
20     begin
21         clock = 0;
22         in = 0;
23
24         $dumpfile("test.vcd");
25         $dumpvars;
26
27         #1 in = 0;
28
29         for(i=1; i<9; i=i+1)
30             #2 in = i;
31
32         #10 $finish;
33
34     end
35 endmodule // test

```

5.3 Homework

Esercizio 5.3.1. Si forniscano le espressioni booleane che calcolano: Se una configurazione di 8 bit è una potenza di due: È necessario soltanto uno XOR dei singoli bit $b_0 \oplus b_1 \oplus b_2 \oplus b_3 \oplus b_4 \oplus b_5 \oplus b_6 \oplus b_7$.

Fornire le espressioni booleane che calcolano l'indice del bit più significativo a 1 in una configurazione di 8 bit:

x_7	x_6	x_5	x_4	x_3	x_2	x_1	x_0	z_3	z_2	z_1	z_0
1	-	-	-	-	-	-	-	0	1	1	1
0	1	-	-	-	-	-	-	0	1	1	0
0	0	1	-	-	-	-	-	0	1	0	1
0	0	0	1	-	-	-	-	0	1	0	0
0	0	0	0	1	-	-	-	0	0	1	1
0	0	0	0	0	1	-	-	0	0	1	0
0	0	0	0	0	0	1	-	0	0	0	1
0	0	0	0	0	0	0	1	0	0	0	0
0	0	0	0	0	0	0	0	1	1	1	1

Tabella 5.1: Tabella di verità di una rete combinatoria che restituisce l'indice del bit più significativo a 1 di una configurazione di 8 bit. L'indice inizia da 1, il valore in output 0 viene usato se il numero in input sono tutti 0

$$\begin{aligned}
 z_3 &= \overline{x_7 x_6 x_5 x_4 x_3 x_2 x_1 x_0} \\
 z_2 &= x_7 + \overline{x_7} x_6 + \overline{x_7} \overline{x_6} x_5 + \overline{x_7} \overline{x_6} \overline{x_5} x_4 + \overline{x_7} \overline{x_6} \overline{x_5} \overline{x_4} x_3 x_2 x_1 x_0 \\
 z_1 &= x_7 + \overline{x_7} x_6 + \overline{x_7} \overline{x_6} x_5 x_4 x_3 + \overline{x_7} \overline{x_6} \overline{x_5} x_4 x_3 x_2 + \overline{x_7} \overline{x_6} \overline{x_5} \overline{x_4} x_3 x_2 x_1 x_0 \\
 z_0 &= x_7 + \overline{x_7} \overline{x_6} x_5 + \overline{x_7} \overline{x_6} \overline{x_5} x_4 x_3 + \overline{x_7} \overline{x_6} \overline{x_5} \overline{x_4} x_3 x_2 x_1 + \overline{x_7} \overline{x_6} \overline{x_5} \overline{x_4} \overline{x_3} x_2 x_1 x_0
 \end{aligned}$$

Esercizio 5.3.2. Delle reti combinatorie definite dalle espressioni booleane dell'esercizio precedente si fornisca l'implementazione come module Verilog insieme ad un programma testbench che ne dimostri il corretto funzionamento.

Per calcolare se un numero è potenza di due non utilizziamo il bitwise xor, ma mettiamo in bitwise AND il numero in input e il numero in input a cui sottraiamo 1. Se il numero in input è una potenza

di due, quando gli viene sottratto 1 rimarranno soltanto 1 a destra della posizione dell'1 nel numero in input. Se l'AND è 0 allora il numero è potenza di due.

Listing 5.8: Modulo che calcola se una configurazione di bit è potenza di 2

```
1 module potenzadidue(output z, input [0:N-1]x);
2     parameter N = 8;
3
4     assign
5         z = (x & (x-1)) == 0 ;
6 endmodule //potenzadidue
```

Listing 5.9: Modulo di test della rete combinatoria che calcola se una configurazione di bit è potenza di 2

```
1 module test();
2     reg [0:7] x;
3     wire z;
4
5     potenzadidue p (z, x);
6
7     initial
8         begin
9             $dumpfile("test.vcd");
10            $dumpvars;
11
12            x = 8'b10000000; // z = 1
13            #1 x = 8'b01000000; // z = 1
14            #1 x = 8'b00100000; // z = 1
15            #1 x = 8'b00010000; // z = 1
16            #1 x = 8'b00001000; // z = 1
17            #1 x = 8'b00000100; // z = 1
18            #1 x = 8'b00000010; // z = 1
19            #1 x = 8'b00000001; // z = 1
20            #1 x = 8'b01010010; // z = 0
21            #1 x = 8'b01011010; // z = 0
22            #1 x = 8'b01000001; // z = 0
23            #1 x = 8'b11010000; // z = 0
24            #1 x = 8'b10110110; // z = 0
25        end
26 endmodule //test
```

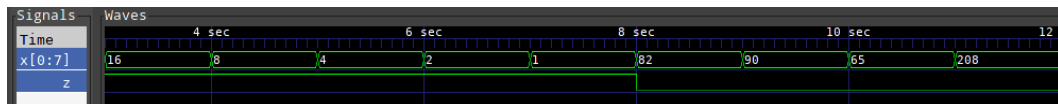


Figura 5.16: Visualizzazione dell'esecuzione del modulo che calcola se un numero da 8 bit è potenza di 2

Per implementare una rete combinatoria che restituisce l'indice del bit più significativo si può utilizzare un semplice ciclo for per realizzare un modulo behavioral. La funzione built-in `$clog2()` calcola il logaritmo in base 2 di un numero. La utilizziamo per definire la dimensione dell'output del modulo, dato che il numero di bit n del valore in input è rappresentabile con un numero di $\log_2(n)$ bit.

Listing 5.10: Modulo della rete combinatoria che calcola l'indice del bit ad 1 più significativo in un numero da N bit

```
1 module index (
```



```

2     input [N-1:0] x,
3     output logic [$clog2(N):0] z
4 );
5
6 parameter N = 8;
7 reg [$clog2(N):0] i;
8
9 always @(x)
10     if(x == 0)
11         z <= ~0; // set all output bits to 1
12     else
13         for (i = 0; i < N; i++) begin
14             if(x[i])
15                 z <= i;
16         end
17
18
19 endmodule // index

```

Listing 5.11: Modulo di test del modulo che calcola l'indice del bit ad 1 più significativo in un numero da N bit

```

1 module test();
2     reg [7:0] x;
3     wire [$clog2(7):0] z;
4
5     index p (x, z);
6
7     initial
8         begin
9             $dumpfile("test.vcd");
10            $dumpvars;
11
12            x = 8'b10000000;
13            #1 x = 8'b01000000;
14            #1 x = 8'b00100000;
15            #1 x = 8'b00010000;
16            #1 x = 8'b00001000;
17            #1 x = 8'b00000100;
18            #1 x = 8'b00000010;
19            #1 x = 8'b00000001;
20            #1 x = 8'b00000000;
21            #1 x = 8'b01010010;
22            #1 x = 8'b01011010;
23            #1 x = 8'b01000001;
24            #1 x = 8'b11010000;
25            #1 x = 8'b10110110;
26            #1 x = 8'b00000000;
27        end
28 endmodule // test

```

Esercizio 5.3.3. Definire un automa che processa una coppia di variabili da k bit e che conta, modulo 2, il numero delle volte che le due variabili in ingresso sono uguali.

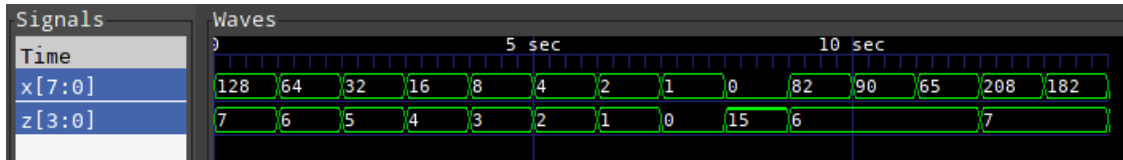
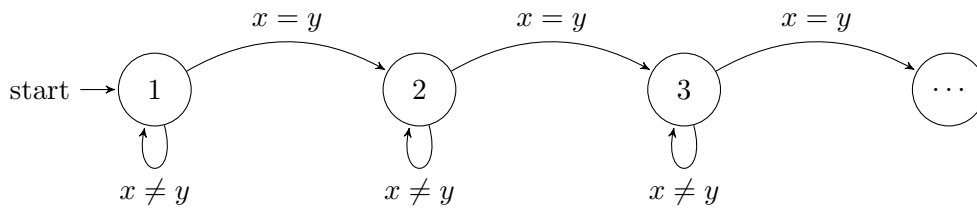


Figura 5.17: Visualizzazione dell'esecuzione del modulo di test del modulo che calcola l'indice del bit ad 1 più significativo in un numero da N bit

Figura 5.18: Automa di Moore che conta quante volte gli input sono uguali. Gli stati possono essere teoricamente infiniti, l'output è uguale al numero di stato, realisticamente gli stati possibili saranno 2^n dove n è il numero di bit nel registro



Per semplicità di implementazione in Verilog decidiamo di utilizzare un contatore come registro. (L'alternativa potrebbe essere includere una ALU nel modulo di σ che legge il valore del registro di stato e lo incrementa di uno nel caso lo stato debba essere incrementato). La funzione σ calcolerà $x \bmod 2$ e $y \bmod 2$, eseguirà un confronto e nel caso siano uguali invierà il risultato del confronto alla linea INCREMENT del contatore, così che al prossimo ciclo di clock il contatore delle volte che x e y sono uguali aumenti. La funzione ω è la funzione identità.

Esercizio 5.3.4. Implementare l'automa dell'esercizio precedente in Verilog.

Listing 5.12: Modulo contatore

```

1 module counter(
2     input inc,
3     input clock,
4     output [0:N-1] z
5 );
6     parameter N = 8;
7
8     reg [0:N-1] value;
9
10    initial
11        value = 0;
12
13    always @(posedge clock) begin
14        value = value + inc;
15    end
16
17    assign
18        z = value;
19
20 endmodule // counter
  
```

Listing 5.13: Modulo di σ che controlla se x e y sono uguali modulo 2 e invia il bit di increment al contatore

```

1 module sigma (
  
```

```

2     input [0:M-1] x,
3     input [0:M-1] y,
4     output inc
5 );
6     parameter M = 8;
7
8     assign
9         inc = x[M-1] == y[M-1];
10 endmodule // sigma

```

Listing 5.14: Modulo di test della rete combinatoria dell'automa che conta il numero di volte che due numeri in input sono uguali modulo 2

```

1 module test();
2
3     reg [0:7] x;
4     reg [0:7] y;
5     wire [0:7] z;
6     reg clock;
7
8     automata a(x, y, clock, z);
9
10    initial
11        clock = 0;
12
13    always
14        begin
15            #2 clock = 1;
16            #1 clock = 0;
17        end
18
19    initial
20        begin
21            $dumpfile("test.vcd");
22            $dumpvars;
23
24            x = 3;
25            y = 5;
26            #3 x = 2;
27            y = 3;
28            #3 x = 3;
29            #3 x = 1;
30            #3 x = 2;
31            y = 2;
32
33            #10 $finish;
34        end
35 endmodule // test

```

Esercizio 5.3.5. Progettare, utilizzando componenti standard una rete sequenziale che possa essere utilizzata per implementare operazioni di push e pop su uno stack di massimo 8 posizioni.

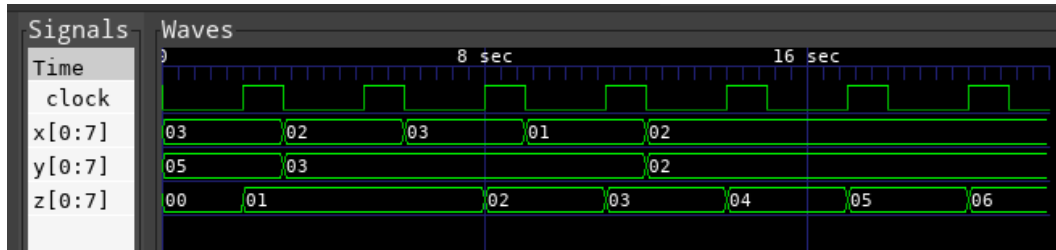


Figura 5.19: Visualizzazione dell'esecuzione del modulo di test dell'automa che conta quante volte due numeri sono uguali modulo 2

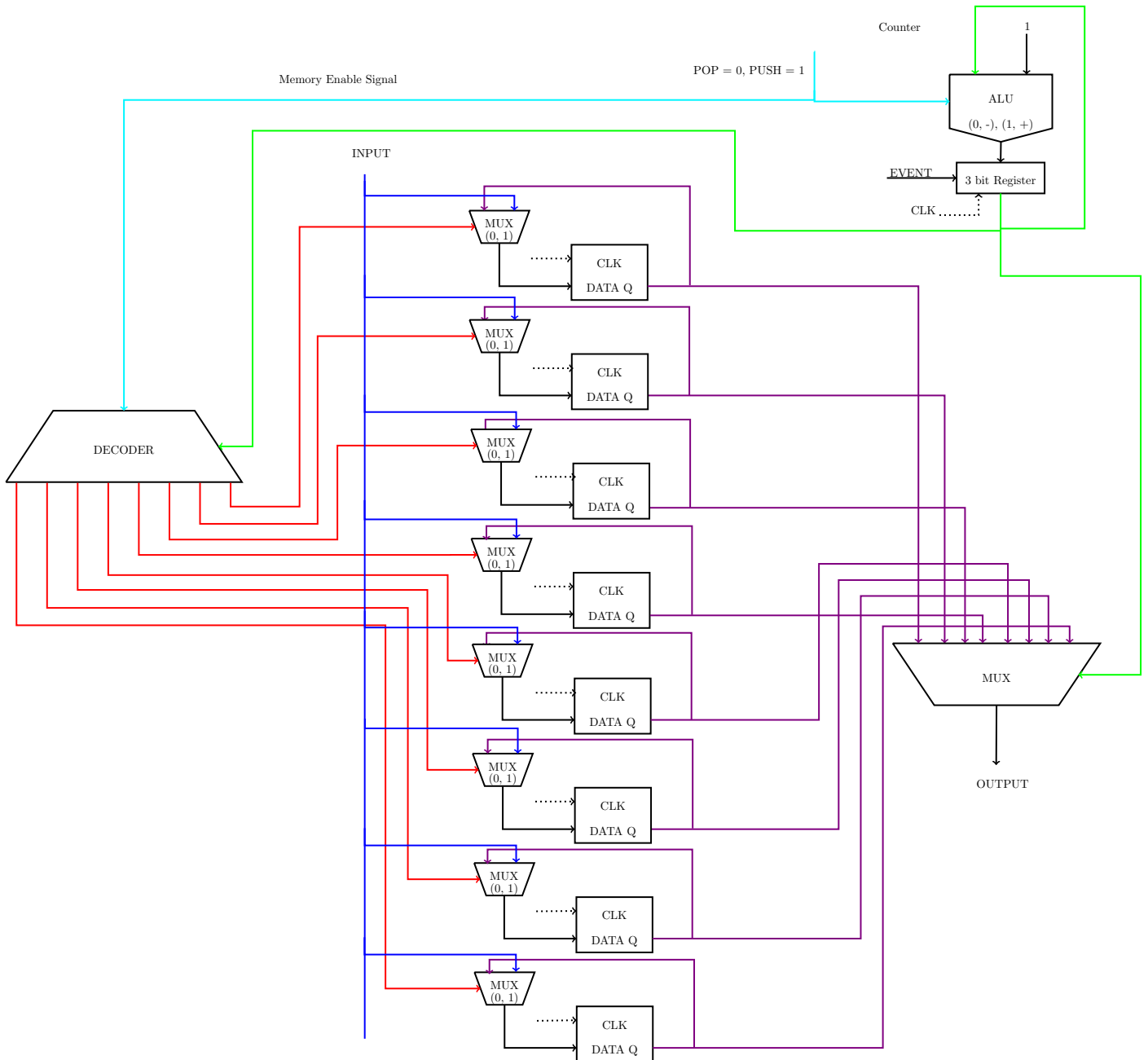


Figura 5.20: Diagramma di circuito con componenti standard che realizza uno stack di 8 posizioni. Il numero di bit dei registri è stato omissso volutamente.

Parte II

Capitolo 6

Assembler ARM e microarchitettura

6.1 Introduzione all'Assembler ARM

Abbiamo visto un ciclo che viene eseguito dal processore.

```
while(true) {  
    preleva un istruzione (ASM)  
    interpreta  
    esegui  
}
```

Definizione 6.1.1. Prelevare un istruzione significa estrarre dalla memoria una parola (configurazione di 0 e 1) contenente un istruzione **Assembly**. L'Assembly, spesso abbreviato **asm** è un linguaggio di programmazione a basso livello con una forte corrispondenza fra il linguaggio e le istruzioni in linguaggio macchina di una particolare architettura. Vedremo il linguaggio Assembler per ARM, architettura acronimo di Advanced Risc Machines, dove Risc sta per Reduced Instruction Set Computer. Risc utilizza molte meno istruzioni di CISC, più comune nelle architetture X86 e X86_64. I principi di Risc sono:

1. La regolarità \implies semplicità
2. Supporto al caso più frequente
3. "Piccolo è bello"
4. Un buon risultato è frutto di un buon compromesso

Il linguaggio asm viene compilato a linguaggio macchina. asm è un linguaggio mnemonico più vicino possibile alle istruzioni macchina. Una volta compilato e caricato in memoria, le istruzioni di un programma Assembler risiedono in dei registri.

Definizione 6.1.2. Nell'architettura **Von Neumann** è presente un **Program Counter** (abbreviato con pc), un contatore (registro). Nel caso dell'architettura ARMv7 (32 bit) il pc è un registro generale in cui è abilitata la scrittura. In ARMv8 (64 bit) non è più possibile utilizzare il pc come un registro generale. Il contenuto del pc indica l'indirizzo di memoria dove risiede l'indirizzo dell'istruzione corrente eseguita dal ciclo del processore, e dev'essere aggiornato per contenere l'indirizzo della prossima istruzione alla fine del ciclo.

Definizione 6.1.3. Tipi di istruzioni Assembler: Vedremo istruzioni **aritmetico-logiche**, istruzioni di **load/store**, istruzioni di **controllo di flusso** ed alcune istruzioni **speciali**.

6.2 Istruzioni Assembler ARMv7

Nota. Il linguaggio Assembler ARM è radicalmente diverso dai linguaggi Assembler della famiglia X86! Vi sono diversi tipi di istruzioni ARM asm, vediamo i tipi di operandi (parametri) che le istruzioni primitive ricevono.

Definizione 6.2.1. Sintassi delle istruzioni ARMv7:

Nota. Le istruzioni sono case-insensitive.

1. Istruzioni a singolo operando (MOV, MOVN)

`<opcode>{<cond>}{<flags>} Rd, <Op2>`

2. Istruzioni che non producono un risultato (CMP, CMN, TEQ, TST)

`<opcode>{<cond>} Rn, <Op2>`

3. Altre istruzioni (AND,EOR,SUB,RSB,ADD,ADC,SBC,RSC,ORR,BIC)

`<opcode>{<cond>}{<flags>} Rd, Rn, <Op2>`

- `<opcode>` è l'istruzione da eseguire
- `cond` Un codice condizionale di due lettere (opzionale).
- `flags` Flag opzionali aggiuntivi fra cui `S`, che è implicito per le istruzioni `CMP`, `CMN`, `TEQ`, `TST`.
- `OP2` Un secondo operando flessibile (opzionale), ha la forma `Rm,<shift>` oppure `<#imm>`
- `Rd` Il registro di destinazione dell'operazione
- `Rn,Rm` Registri sorgente.
- `<#imm>` Un valore immediato (se rappresentabile).
- `<shift>` Un operazione di shift della forma `<shiftname> <register>` oppure della forma `<shiftname> <#imm>`, oppure soltanto `RRX`
- `<shiftname>` Opcode di un istruzione di shift (`ASL`, `LSL`, `LSR`, `ASR`, `ROR`)
- `@ commento` Si possono inserire commenti a fine riga.

<i>cond: condition code</i>			
code	meaning	flags tested	4 bit cond field
AL or omitted	always	(ignored)	1110
EQ	equal	$Z = 1$	0000
NE	not equal	$Z = 0$	0001
CS	carry set (same as HS)	$C = 1$	0010
CC	carry clear (same as LO)	$C = 0$	0011
MI	minus	$N = 1$	0100
PL	positive or zero	$N = 0$	0101
VS	overflow	$V = 1$	0110
VC	no overflow	$V = 0$	0111
HS	unsigned higher or same	$C = 1$	0010
LO	unsigned lower	$C = 0$	0011
HI	unsigned higher	$C = 1 \wedge Z = 0$	1000
LS	unsigned lower or same	$C = 0 \vee Z = 1$	1001
GE	signed greater than or equal	$N = V$	1010
LT	signed less than	$N \neq V$	1011
GT	signed greater than	$Z = 0 \wedge N = V$	1100
LE	signed less than or equal	$Z = 1 \vee N \neq V$	1101
NV	never		1111

Tabella 6.1: Tabella dei codici condizionali ARMv7

Definizione 6.2.2. Registri ARMv7: Nelle cpu ARMv7 sono presenti soltanto 16 registri da 32 bit, della forma `rn` dove $0 \leq n < 16$.

- `r0`: Parametri temporanei o valori di ritorno. (Standard per i valori di ritorno di funzioni)
- `r1 - r3`: Argomenti, valori temporanei.
- `r4 - r12`: Valori temporanei.
- `r13`: Stack Pointer (SP).
- `r14`: Link register.
- `r15`: Program counter.

Definizione 6.2.3. Aliasing dei registri

Si possono usare degli alias per riferirsi ai registri utilizzando la direttiva `.req`. Ad esempio, inserendo

```
MYName .req r5
```

Ci permette di utilizzare l'alias `MYNAME` al posto di `r5` nel codice che segue.

Definizione 6.2.4. Operandi Immediati: Operandi che non risiedono in dei registri ma sono costanti presenti all'interno della configurazione della parola, ad esempio `#1` è costante per il numero 1. `#0x1234` corrisponde in esadecimale al numero 4661

Definizione 6.2.5. Operandi e istruzioni di Memoria: Gli operandi di memoria sono della forma:

```
@ access memory by offset without altering rx
[rx, #imm]
[rx, rn]
[rx, rn, <shift>]
@ update rx by value then access memory
[rx, #imm]!
[rx, rn]!
[rx, rn, <shift>]!
@ access memory at rx then update rx by offset
[rx], #imm
[rx], rn
[rx], rn, <shift>
```

Tale operando va a leggere il contenuto della memoria all'indirizzo puntato dal registro `Rx`, a cui va aggiunto l'offset.

L'istruzione per caricare una parola dalla memoria in un registro è

```
@ with pre-index r2 is unaltered
ldr r0, [r2, #0]
@ with post-increment r2 is incremented by 4 after load
ldr r0, [r2], #4
@ from a label
.data
x: .word 123
ldr r0, =x
@ treat x's content as an address and load it
ldr r0, [r0]
@ load a single byte with #1 byte post increment
ldrb r2, [r0], #1
```

Tale istruzione caricherà nel registro **r0** il contenuto dell'indirizzo puntato dal contenuto di **r2** a cui va aggiunto 0. L'offset è utile per accedere a strutture dati (memorizzate sequenzialmente) come **struct** e **array**, presenti nel linguaggio C.

Per caricare in un registro un operando immediato o il valore di un altro registro si usa

```
mov rd, <src>
@ <src> is either a register or an immediate value
```

Per salvare il valore di un registro in memoria si usa

```
str
```

Nota. La memoria di ARM, come microarchitettura, è indirizzata a byte (gli indirizzi corrispondono a singoli byte). Si può caricare un valore dalla memoria in un registro con

```
mov r2, #0x400
ldr r0, [r2, #0]
```

All'interno del registro **r0** saranno caricati i 4 byte a partire dall'indirizzo 1024 (400 in esadecimale).

Definizione 6.2.6. Big Endian e Little Endian: Se vogliamo caricare un numero da 4 byte (32 bit) in un registro dalla memoria, può sorgere il dubbio, in quale ordine vengono letti i byte? Nel metodo **Big Endian** carica partendo dal byte più significativo (quello più a sinistra), fino a quello meno significativo. Nel metodo **Little Endian**, il primo byte caricato è quello meno significativo (quello più a destra del registro/numero), mentre l'ultimo caricato è il byte più significativo. ARMv7 supporta entrambi i metodi di **endianness**.

Definizione 6.2.7. Istruzioni Logiche Bitwise

Realizza l'and bit per bit dei registri **r1** (maschera) e **r2** (sorgente), memorizzando in **rd**

```
and rd, r1, r2
```

Realizza l'OR bit per bit dei registri **r1** e **r2**, memorizzando in **rd**

```
orr rd, r1, r2
```

Realizza l'Exclusive OR bit per bit dei registri **r1** e **r2**, memorizzando in **rd**

```
eor rd, r1, r2
```

Realizza l'Exclusive OR bit per bit dei registri **r1** e **r2**, memorizzando in **rd**

```
eor rd, r1, r2
```

Imposta a 0 i bit di **r2** laddove nella posizione corrispondente della maschera **r1** vi sia un 1, memorizzando il risultato in **rd**

```
bic rd, r1, r2
```

Definizione 6.2.8. Bit di flag in ARMv7: In ARMv7 sono presenti 4 bit di flag di condizione (che possono essere alterati dopo l'esecuzione di un istruzione). Il flag **Z** indica se il risultato di un operazione è 0. Il flag **N** indica se il risultato dell'operazione è negativo. Il flag **C** indica se è presente un carry (riporto) sull'ultima cifra. Il flag **V** (overflow) indica se l'operazione ha dato overflow. I flag sono memorizzati in un registro detto "parola di stato", che non è accessibile come gli altri registri general purpose. Vi è presente anche un bit che indica l'endianness delle operazioni di memoria. Solo alcune istruzioni impostano di default i set di condizione (come **CMP**). Per forzare un'istruzione ad impostare i bit di flag di condizione si può passare il flag aggiuntivo **S**.

Definizione 6.2.9. Istruzioni Aritmetiche

Realizza la somma dei registri **r1** e **r2**, memorizzando il risultato in **rd**

```
add rd, r1, r2
```

Realizza la somma dei registri **r1** e **r2 con riporto**, memorizzando il risultato in **rd**

```
adc rd, r1, r2
```

Realizza la sottrazione dei registri **r1 - r2**, memorizzando il risultato in **rd**

```
sub rd, r1, r2
```

Realizza la sottrazione dei registri **r1 - r2 con riporto**, memorizzando il risultato in **rd**

```
sbc rd, r1, r2
```

Compare accetta solamente due operandi, realizza la sottrazione dei registri **r1 - r2** ed **imposta i bit di flag**. Viene utilizzato per realizzare esecuzione condizionale nei programmi, laddove le istruzioni successive utilizzino un codice condizionale.

```
cmp r1, r2
```

Sottrazione "al contrario" (Reverse Subtraction), esegue l'operazione **r2 - r1** e memorizza il risultato in **rd**.

```
sbc rd, r1, r2
```

Definizione 6.2.10. Istruzioni di Shift (logico-aritmetiche)

Nota. I registri passati per parametro alle istruzioni di shift non sono registri di destinazione ma contengono il numero parametro dello shift (di quante posizioni deve essere effettuato). Vedi il paragrafo sulla sintassi delle istruzioni Assembler.

Bitshift di **num** posizioni a sinistra. Semanticamente identico all'istruzione **ASL**. Equivalente alla moltiplicazione per la **num**-esima potenza di 2.

```
@ come operando
lsl #imm
lsl rn
@ come istruzione
lsl rd, rn, #imm
```

Bitshift di **num** posizioni a destra. Equivalente alla divisione per la **num**-esima potenza di 2.

```
@ come operando
lsr #imm
lsr rn
@ come istruzione
lsr rd, rn, #imm
```

Bitshift aritmetico di **num** posizioni a destra (preserva il bit di segno). Equivalente alla divisione per la **num**-esima potenza di 2.

```
@ come operando
asr #imm
asr rn
@ come istruzione
asr rd, rn, #imm
```

Rotate Right: esegue uno shift bitwise a destra e re-inserisce il bit shiftato (meno significativo), nella posizione del bit più significativo (più a sinistra).

```
@ come operando
ror #imm
ror rn
@ come istruzione
ror rd, rn, #imm
```

Esempio 6.2.1. Program Counter:

```
mov r0, #15
add r0, r0, r0
```

Il program counter (pc) prima dell'esecuzione dell'istruzione MOV indicherà l'indirizzo di memoria dove è contenuta la parola dell'istruzione. Alla fine dell'esecuzione di MOV sarà incrementato di 4 posizioni (4 byte = 32 bit, dimensione di una parola).

Definizione 6.2.11. Istruzioni di salto

L'istruzione branch "salta" all'istruzione contenuta in pc a cui viene sommato un valore immediato. Più semplicemente può saltare ad un'istruzione contenuta in un'etichetta. Sono di fondamentale utilizzo i flag condizionali.

```
B #imm          @ offset #imm is relative to pc
B label         @ branch to label
```

Salva il contenuto di pc in LR (registro 14) e "salta" all'istruzione contenuta in pc + #num. Sono di fondamentale utilizzo i flag condizionali.

```
BL #imm         @ offset #imm is relative to pc
BL label        @ branch to label
```

Si utilizza per realizzare l'equivalente di procedure e funzioni in asm.

6.3 Direttive, Pseudoistruzioni e Programmi

Definizione 6.3.1. Direttive Le direttive non sono istruzioni asm, ma sono istruzioni per il compilatore (toolchain GNU) che compileranno a sezioni del programma. La sintassi delle direttive inizia con un simbolo "punto" .

Definizione 6.3.2. Direttive di sezione del codice

Inizia una nuova sezione di codice o dati.

```
.section <section_name> {, "<flags>"}
```

Le sezioni <section_name> nell'assembler ARM gnu possono essere:

- .text una sezione di codice
- .data una sezione di dati inizializzati
- .bss una sezione di dati non inizializzati
- .rodata inizializza una sezione di dati (read only?)

Definizione 6.3.3. Direttive di inizializzazione di dati

Inserisce una lista di valori di parola da 32 bit nell'assembly.

```
.word 123
.word 1,2,3,4
```

Inserisce una lista di byte nell'assembly.

```
.byte <byte1> {, <byte2>, ...}
```

Inserisce una stringa di caratteri ASCII nell'assembly.

```
.ascii "<string>"
```

Come .ascii, inserisce una stringa di caratteri ASCII nell'assembly, ma seguita da un byte 0.

```
.asciz "<string>"
```

Riserva number_of_bytes byte in memoria, sono riempiti con byte 0.

```
.space <number_of_bytes>
```

Definizione 6.3.4. Chiamate al Sistema Operativo Come da C, possiamo fare chiamate al Sistema Operativo in Assembly. Si possono chiamare funzioni di librerie, che comprendono funzioni per i processi e funzioni per l'IO.

Realizza una chiamata di sistema, il cui codice è contenuto in `r7`. Le due istruzioni sono equivalenti.

```
svc 0    @ "supervisor call"
swi 0    @ "software interrupt"
```

Esempio 6.3.1. `write(fd, buffer, num_byte)`; è una funzione C che stampa `num_byte` del `buffer` all'interno di `fd`, ovvero un *file descriptor*. La funzione della *stdlib* C `printf` che utilizza internamente `write`. I file descriptor standard in sistemi Unix sono

- 0: standard input o `stdin`
- 1: standard output o `stdout`
- 2: standard error o `stderr`

Il numero della syscall `write` è 4. Il numero della syscall `exit` è 1.

Definizione 6.3.5. Label I label identificano sezioni di codice.

```
testlabel:
<instructions>
...
```

Per accedere all'indirizzo di memoria di un label `esempio` contenuto nella sezione `.data` si usa l'operando `=esempio`. Per i label nella sezione `.text` non va utilizzato il simbolo prefisso `=`.

Esempio 6.3.2. Hello World in Assembler ARM

Listing 6.1: Hello World in Assembler ARM

```
1  .data                                @ initialized data section
2  message:                            @ label "message"
3      .asciz "hello_world\n"          @ allocate a string ended with a 0 byte
4  len = .-message                     @ create a len label containing length of "message"
5
6  .text                                @ section containing the code to run
7  .global main                        @ give the label "_start" external linkage
8  main:                               @ main program
9      mov r0, #1                      @ tell SYS_WRITE to use #1 (stdout) as fd
10     ldr r1, =message                 @ load message label from memory into SYS_WRITE
11                                     @ buffer. an =x operand returns x's mem address
12     ldr r2, =len                    @ same as previous instruction but load "how many
13                                     @ bytes to write" as SYS_WRITE third argument
14     mov r7, #4                      @ SYS_WRITE code is 4.
15     swi 0                           @ software interrupt
16
17     mov r7, #1                      @ SYS_EXIT code is 1. r7 is where svc and swi take
18                                     @ SYSCALL codes
19     swi 0
```

Definizione 6.3.6. If-then-else in asm ARM La direttiva `.if` rende il blocco di codice seguente condizionale. Analogamente esiste la direttiva `.else`. Una direttiva condizionale `.if` si termina con la direttiva `.endif`.

```

1 .if <logical_expression>
2     <do_something>
3 .else
4     <do_something_else>
5 .endif

```

Vediamo come viene compilata ad Assembly una direttiva `if`, utilizzeremo quasi sempre le direttive e non condizioni testate manualmente.

```

1 @ asm equivalent to C: if(x == y) {x++; y=2x} else x--;
2     cmp r1, r2      @ x and y are in r1 and r2, test condition
3     bne else
4 then:
5     add r1, r1, #1   @ x++
6     add r2, r1, r1   @ y=2x
7     b cont
8 else:
9     sub r1, r1, #1
10 cont:
11     \dots           @ continue program

```

Definizione 6.3.7. Ciclo for in asm ARM

```

1 @ asm equivalent to C: for(int i=0; i<4; i++)
2     mov r0, #0      @ initialize counter
3 for:
4     cmp #4, r0      @ test condition, sets flags
5     beq endfor
6     add r0, r0, #1   @ step increment
7 endfor:
8     ...             @ for is over, continue program

```

Definizione 6.3.8. Switch-case in asm ARM

```

switch(x) {
    case 1: {x = 100; break;}
    case 2: {x = 200; break;}
    default: x = 300;
}

```

Realizziamo l'equivalente di questo snippet C in asm ARM

```

1 case1:
2     cmp r0, #1
3     bne case2
4     mov r0, #100
5     b cont
6 case2:
7     cmp r0, #2
8     bne default
9     mov r0, #200
10    b cont
11 default:
12    mov r0, #300
13 cont:
14    ... @ continue program

```

Esempio 6.3.3. Allocare un vettore di interi

```

1 .data
2 a:      .word 1,2,3,4    @ array
3 n =     (.-a)/4
4 @ the number of bytes in the arr
5 @ divided by the number of bytes (at compile time)
6 @ in a word (4 bit integers)
7 @ gives us the len of array

```

Esempio 6.3.4. Accedere ad un vettore

```

for(int i = 0; i < 5; i++) {
    y = x[i];
}

```

Realizziamo l'equivalente di questo snippet C in asm ARM

```

1 mov r3, #0                @ i = 0
2 loop:
3     ldr r2, [r1, r3]       @ array's address is store in r1, use r3 as offset
4                             @ ... do something
5     add r3, r3, #4         @ i++ (memory offset is in bytes)
6     cmp r3, #5 * #4       @ test i < 5 (mult by 4 bytes)
7     bne loop

```

Oppure, semplificando utilizzando un post-indice per ldr:

```

1 .data
2 array: .word 4,3,2,1    @ the array
3 length : (.-array) / 4 @ length
4
5 .text
6 .global main
7     ldr r5, =length
8     ldr r6, =array
9 loop:
10    ldr r2, [r6], #4      @ load and add offset to r1 at the same time
11    subs r5, r5, #1
12    bne loop

```

Esempio 6.3.5. Accedere ad un vettore, 2

```

for(int i = 0; i < 5; i++)
    x[i]++;

```

Realizziamo l'equivalente di questo snippet C in asm ARM

```

1 .data
2 array: .word 4,3,2,1    @ the array
3 length : (.-array) / 4 @ length
4
5 .text
6 .global main
7     ldr r5, =length
8     ldr r6, =array
9 loop:
10    ldr r2, [r6]

```

```

11      add r2, r2, #1      @ x[i]++;
12      str r2, [r6]        @ store r2
13      add r6, r6, #4      @ increment offset
14      subs r5, r5, #1
15      bne loop

```

Esempio 6.3.6. Realizzare funzioni e chiamate di funzione Per realizzare una funzione in asm ARM si definisce una sezione di codice con un label. Per "chiamare" la funzione si utilizza l'istruzione `bl nomefunzione` (*branch and link*) che eseguirà un *branch* all'istruzione contenuta nel label e salverà l'indirizzo dell'istruzione corrente all'interno del registro `lr` o `r14` (*link register*). Per ritornare dall'esecuzione della funzione al codice chiamante si usa l'istruzione `bx lr` per ritornare all'istruzione salvata nel *link register*. Si può anche effettuare il return con una condizione utilizzando ad esempio `moveq pc,lr` per impostare il registro *program counter* a `lr` se l'istruzione precedente ha impostato il flag *Zero* a 1.

Si possono usare i registri da `r0` a `r4` (escluso) per i parametri di una funzione. `r0` è anche il valore di return. Il valore di ritorno della funzione va memorizzato in `r0` prima del ritorno da essa. Dal quarto parametro in poi, i parametri vanno salvati sullo stack con `push`. Le funzioni della standard library C consumano direttamente i parametri necessari sullo stack.

Definizione 6.3.9. Utilizzare lo Stack Lo stack è una sezione di memoria del programma/processo, utilizzata con costrutti ed istruzioni particolari per essere equivalente alla struttura dati *stack* comune in programmazione. Viene allocato quando viene creato il processo. Utilizziamo lo stack per memorizzare valori temporanei come le variabili locali di una funzione, o valori per i quali non vi è abbastanza spazio nei registri utilizzabili per i valori temporanei (ad esempio, gli argomenti dal quinto in poi in una chiamata a `printf`). Per interagire con lo stack utilizziamo le istruzioni `push` e `pop`, alias ad altre istruzioni di memoria per semplificarne l'accesso. Lo stack *cresce* quando inseriamo una parola (32 bit) all'interno di esso, alla locazione contenuta nel registro `r13` o `sp` (*stack pointer*).

Esempio 6.3.7. Esempio di utilizzo dello stack

```

1 main:
2      mov     r0, #2      @ set up r0
3      push   {r0}        @ save r0 onto the stack
4      mov     r0, #3      @ overwrite r0
5      pop     {r0}        @ restore r0 to it's initial state
6      mov     r7, #1      @ finish the program
7      svc     0

```

Esempio 6.3.8. Chiamare funzioni C Per chiamare funzioni della standard library C o di una libreria C si può utilizzare la direttiva `.extern` per includere la funzione nel programma asm. Si possono chiamare le funzioni rese visibili con `bl`, come una normale funzione asm.

Esempio 6.3.9. Hello world con funzioni della stdlib C

```

1 @ Make the glibc symbols visible.
2 .extern exit, puts
3 .data
4      msg: .asciz "hello_world"
5 .text
6 .global main
7 main:
8      @r0 is the first argument.
9      ldr r0, =msg
10     bl puts
11     mov r0, #0
12     bl exit

```


Esempio 6.3.10. Programma per stampare un array con printf

Listing 6.2: printarray.s

```

1  .data
2  fmt: .asciz "The number is: %d\n"
3  array: .word 4,3,2,1    @ the array
4  .equ n, (. - array) / 4    @ the number of bytes in the arr
5                               @ divided by the number of bytes
6                               @ in a word (4 bit integers)
7                               @ gives us the length of array
8
9  .text
10 .global main
11 .extern printf
12
13 LENGTH .req r7
14 OFFSET .req r6
15 STRING .req r0
16 NUMBER .req r1
17
18 @ print a vector in order
19 main:
20     push {ip, lr}
21     ldr LENGTH, =n
22     ldr OFFSET, =array
23     b loop
24 loop:
25     ldr NUMBER, [OFFSET], #4
26     ldr STRING, =fmt    @ load the string to print
27     bl printf
28     subs LENGTH, LENGTH, #1
29     bne loop
30     pop {ip, pc}
31     mov r7, #1    @ exit
32     svc 0

```

Esempio 6.3.11. Equivalente della funzione main e argomenti a linea di comando All'avvio della sezione main del codice ARM asm GNU, in `r0` sarà contenuto il numero di argomenti equivalente ad `argc` e in `r1` sarà contenuto l'indirizzo di memoria dove l'equivalente di `argv` è contenuto.

“

Definizione 6.3.10. Modi operativi del processore. I processori, fra cui quelli ARM, hanno diversi modi di operazione. Nei processori in generale sono presenti i modi **user** e **kernel** (*privileged*). In **user** mode il codice in esecuzione non ha accesso diretto all'hardware o alla memoria, in **kernel** mode il codice in esecuzione ha accesso senza restrizioni alle periferiche e alla memoria.

In ARM sono presenti i modi:

- **user:** esecuzione senza accesso all'hardware e alle periferiche. Ho accesso ad uno spazio di memoria ma non alla memoria completa
- **fast interrupt**
- **interrupt**
- **supervisor:** accesso senza restrizioni al sistema, corrisponde all'esecuzione di istruzioni `svc`.

- abort
- system
- undefined

Nel modo utente sono presenti i 16 registri comuni, quando si passa a modi operativi diversi alcuni registri vengono duplicati, ad esempio quando passo al modo **fast interrupt** vengono duplicati i registri da **r0** a **r8** per non interagire con quanto sta accadendo in modalità utente. In modalità **supervisor** vengono duplicati i registri **r13** e **r14**.

Si tiene traccia del modo corrente in una **parola di stato** del processore nell'unità di controllo. Si abbrevia con *CPSR* (Current Program Status Register).

Definizione 6.3.11. Cosa avviene al boot di un processore ARM Per primo passo imposto PC a 0 ed entro nella modalità CPU **supervisor**. In tale modalità inizio ad eseguire le istruzioni alla locazione il **bootloader** carica il sistema operativo da disco e cambia la modalità di esecuzione a **user**.

6.4 Homework Assembly ARM

Esercizio 6.4.1. Programma che calcola il prefisso di un vettore

Listing 6.3: Programma che calcola il prefisso di un vettore

```

1  .data
2  fmt: .asciz "The number is: %d\n"
3  array: .word 4,3,2,1    @ the array
4  .equ n, (. - array)/4    @ the number of bytes in the arr
5                          @ divided by the number of bytes
6                          @ in a word (4 bit integers)
7                          @ gives us the length of array
8  prefix: .word 0,0,0,0
9
10 .text
11 .global main
12 .extern printf
13
14 LENGTH .req r5
15 OFFSET .req r6
16 PREFOFF .req r10
17 STRING .req r0
18 NUMBER .req r9
19 SUM .req r11
20
21 @ print a vector in order
22 main:
23     push {ip, lr}
24     ldr LENGTH, =n
25     ldr OFFSET, =array
26     ldr PREFOFF, =prefix
27     mov SUM, #0
28     b loop
29 loop:
30     ldr NUMBER, [OFFSET], #4
31     ldr STRING, =fmt    @ load the string to print
32     add SUM, SUM, NUMBER

```

```

33     str SUM, [PREFOFF], #4
34     mov r1, SUM
35     bl printf
36     subs LENGTH,LENGTH,#1
37     bne loop
38     pop {ip, pc}
39     mov r7, #1      @ exit
40     svc 0

```

Esercizio 6.4.2. Programma che calcola la divisione con resto di due interi

Listing 6.4: Programma che calcola la divisione con resto di due interi

```

1  .data
2  fmt:      .asciz "dividend:_%d,_divisor:_%d,_quotient:_%d,_remainder:_%d\n"
3  .text
4  .global main
5  .extern printf
6  main:
7      mov     r1, #128      @ dividend
8      mov     r2, #3        @ divisor
9      mov     r3, #0        @ quotient
10     mov     r4, #0        @ remainder
11     mov     r5, r1        @ down-step from dividend
12     mov     r6, #0        @ dividend*quotient
13 loop:
14     cmp     r5, r2        @ if remainder is < than divisor
15     mulle   r6, r2, r3    @ calculate remainder
16     suble   r4, r1, r6
17     ble     print
18     add     r3, r3, #1
19     sub     r5, r2
20     b loop
21 print:
22     ldr     r0, =fmt
23     push    {r4}
24     bl      printf
25     mov     r7, #1
26     svc     0

```

Esercizio 6.4.3. Programma che calcola la moltiplicazione di due interi

Listing 6.5: Programma che calcola la moltiplicazione di due interi

```

1  .data
2  fmt:      .asciz "x:_%d,_y:_%d,_x*y:_%d"
3  .text
4  .global main
5  .extern printf
6  main:
7      mov     r1, #128      @ x
8      mov     r2, #3        @ y
9      mov     r3, #0        @ result
10     mov     r4, r2        @ counter
11     bl      multiply

```

```

12     b        print
13 multiply:
14     cmp      r4, #0
15     moveq    pc, lr        @ return
16     addgt    r3, r1        @ y > 0
17     subgt    r4, #1
18     sublt    r3, r1        @ y < 0
19     addlt    r4, #1
20     b        multiply
21 print:
22     ldr      r0, =fmt
23     push     {r4}
24     bl       printf
25     mov      r7, #1
26     svc      0

```

Esercizio 6.4.4. Funzione che calcola il prodotto scalare di due vettori

Listing 6.6: Funzione che calcola il prodotto scalare di due vettori

```

1
2 .data
3 fmt:      .asciz  "The scalar product is %d\n"
4 x:        .word  1,0,3,4
5 .equ n,    (-x) / 4
6 y:        .word  0,1,-3,1
7 .text
8 .global main
9 .extern printf
10 main:
11     ldr      r0, =n        @ length
12     ldr      r1, =x
13     ldr      r2, =y
14     bl       scalar
15     mov      r1, r0
16     ldr      r0, =fmt
17     bl       printf
18     mov      r7, #1        @ exit syscall
19     svc      0
20 scalar:
21     mov      r4, #0        @ accumulator
22 loop:
23     ldr      r5, [r1], #4
24     ldr      r6, [r2], #4
25     mla      r4, r5, r6, r4 @ r = x*y+r
26     subs     r0, r0, #1
27     bne      loop
28     mov      r0, r4        @ return accumulator
29     mov      pc, lr

```

Esercizio 6.4.5. Funzione che conta le occorrenze di un carattere in una stringa

Listing 6.7: Funzione che conta le occorrenze di un carattere in una stringa

```

1 .data

```

```

2  str:      .asciz "a_quick_brown_fox_jumps_over_the_lazy_dog"
3  .equ      n, (.-str)
4  fmt:      .asciz "the_character_%c_appears_%d_times_in_the_string\n"
5  .text
6  .global  main
7  main:
8      ldr    r0, =str
9      mov    r1, #'o'
10     bl     countocc
11     mov    r2, r0          @ print
12     ldr    r0, =fmt
13     bl     printf         @ exit
14     mov    r7, #1
15     svc    0
16 countocc:
17     mov    r5, #0          @ counter
18 loop:
19     ldrb   r4, [r0], #1    @ load char
20     cmp    r4, #0
21     moveq  r0, r5
22     moveq  pc, lr
23     cmp    r4, r1
24     addeq  r5, r5, #1
25     b      loop

```

Esercizio 6.4.6. Funzione che calcola l'elevamento a potenza di un intero, con base ed esponente positivi

Listing 6.8: Funzione che calcola l'elevamento a potenza di un intero con base ed esponente positivi

```

1  .data
2  fmt:      .asciz  "%d_to_the_power_of_%d_is_%d\n"
3  .text
4  .global  main
5  .extern  printf
6  main:
7      push   {ip, lr}
8      mov    r0, #5          @ base
9      mov    r1, #4          @ power counter
10     mov    r2, r0          @ multiplier
11     push   {r1, r2}        @ store counter
12     bl     recpow
13     pop    {r1, r2}        @ restore counter
14     mov    r7, r1          @ swap r1, r2
15     mov    r1, r2
16     mov    r2, r7
17     mov    r3, r0
18     ldr    r0, =fmt
19     bl     printf
20     pop    {ip, pc}
21     mov    r7, #1
22     svc    0
23 recpow:
24     cmp    r1, #0

```

```

25     bgt     else
26     mov     r0, #1
27     mov     pc, lr
28 else:
29     push    {r0, lr}           @ calculate recursively
30     sub     r1, r1, #1
31     bl      recpow
32     pop     {r2, lr}
33     mul     r0, r2, r0
34     mov     pc, lr

```

6.5 Instruction Set ARMv7

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
Cond	0	0	I	Opcode				S	Rn				Rd				Operand 2												<i>Data Processing / PSR Transfer</i>			
Cond	0	0	0	0	0	0	A	S	Rd				Rn				Rs		1	0	0	1	Rm				<i>Multiply</i>					
Cond	0	0	0	0	1	U	A	S	RdHi				RdLo				Rn		1	0	0	1	Rm				<i>Multiply Long</i>					
Cond	0	0	0	1	0	B	0	0	Rn				Rd				0	0	0	0	1	0	0	1	Rm				<i>Single Data Swap</i>			
Cond	0	0	0	1	0	0	1	0	1	1	1	1	1	1	1	1	1	1	1	1	0	0	0	1	Rn				<i>Branch and Exchange</i>			
Cond	0	0	0	P	U	0	W	L	Rn				Rd				0	0	0	0	1	S	H	1	Rm				<i>Halfword Data Transfer: register offset</i>			
Cond	0	0	0	P	U	1	W	L	Rn				Rd				Offset		1	S	H	1	Offset				<i>Halfword Data Transfer: immediate offset</i>					
Cond	0	1	I	P	U	B	W	L	Rn				Rd				Offset												<i>Single Data Transfer</i>			
Cond	0	1	1																										1			<i>Undefined</i>
Cond	1	0	0	P	U	S	W	L	Rn				Register List																<i>Block Data Transfer</i>			
Cond	1	0	1	L	Offset																											<i>Branch</i>
Cond	1	1	0	P	U	N	W	L	Rn				CRd				CP#		Offset								<i>Coprocessor Data Transfer</i>					
Cond	1	1	1	0	CP Opc				CRn				CRd				CP#		CP	0	CRm				<i>Coprocessor Data Operation</i>							
Cond	1	1	1	0	CP Opc				L	CRn				Rd				CP#		CP	1	CRm				<i>Coprocessor Register Transfer</i>						
Cond	1	1	1	1	Ignored by processor																											<i>Software Interrupt</i>
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	

Figura 6.1: Instruction Set ARMv7

Definizione 6.5.1. Rappresentazione di un'istruzione di Data Processing in una parola da 32 bit

Le istruzioni assembler vengono rappresentate in una parola da 32 bit.

La prima sezione di 4 bit della parola viene usata per i codici di condizionali (*condition codes*). Vedi la tabella 6.1

La seconda sezione di 2 bit contiene il tipo di istruzione:

- 00: istruzione operativa (*Data Processing*)
- 01: istruzione di memoria (*load, store, etc.*)
- 10: istruzione di salto (*branch, branch and link*)
- 11: Data transfer, non vedremo queste istruzioni.

La prossima sezione, denominata *FUNCT* è di 6 bit e determina l'istruzione da eseguire e gli operandi nel caso ci troviamo in un'operazione di *Data Processing*. Il primo bit di *FUNCT*, detto *I* determina il tipo dell'ultimo operando (operand 2). Se il bit *I* è a 0 allora l'operando 2 è un registro, è un immediato altrimenti. I 4 bit centrali sono detti *Operation Code* o **OPCODE** e determinano l'operazione da eseguire. Ad esempio, le istruzioni di tipo *Data Processing* (Vedi figura 6.1) avranno i seguenti *OPCODE*

opcode: Data Processing Operation Codes			
Assembler Mnemonic	OPCODE	Action	
AND	0000	op1 AND op2	
EOR	0001	op1 EOR op2	
SUB	0010	op1 - op2	
RSB	0011	op2 - op1	
ADD	0100	op1 + op2	
ADC	0101	op1 + op2 + carry	
SBC	0110	op1 - op2 + carry - 1	
RSC	0111	op2 - op1 + carry - 1	
TST	1000	like AND but result is not written	
TEQ	1001	like EOR but result is not written	
CMP	1010	like SUB but result is not written	
CMN	1011	like ADD but result is not written	
ORR	1100	op1 OR op2	
MOV	1101	move op2 in the register op1	
BIC	1110	op1 AND NOT op2 (bit clear)	
MVN	1111	move NOT op2 in the register op1	

Tabella 6.2: Tabella degli OPCODE delle principali operazioni di data processing

L'ultimo bit rimanente nella sezione *FUNCT* è detto *SET*. Se impostato ad 1 allora verranno impostati i bit di flag alla fine dell'esecuzione dell'istruzione. Ne seguono due sezioni da 4 bit che indicano il registro di destinazione *rd* e il registro sorgente *rx*. I rimanenti 12 bit vengono utilizzati per il secondo operando. Se il bit *I* è ad 1, il secondo operando conterrà 4 bit di rotazione e 8 di valore immediato. Altrimenti, se *I* è a 0, i 12 bit del secondo operando vengono interpretati nel seguente modo. Abbiamo due casi, dove il registro è shiftato di una costante, ed un secondo caso dove il registro è shiftato di un altro registro. Nel primo caso abbiamo 5 bit per il valore dello shift, 2 bit che indicano il tipo di shift (*LSL, LSR, ASR, ROR*), 1 bit che indica se il valore dello shift è contenuto in una costante (0) o in un registro (1). I rimanenti 4 bit indicano il registro su cui applicare lo shift. Nel secondo caso (il valore di shift è contenuto in un registro) abbiamo 4 bit per indicare il registro da cui leggere il valore di shift, un bit sempre a 0, il tipo di shift come nel primo caso e 1 bit per il tipo di shift (in questo caso impostato ad 1), i rimanenti 4 bit, come prima, indicano il registro su cui applicare lo shift.

Definizione 6.5.2. Rappresentazione di un istruzione di memoria

Nelle istruzioni di memoria (*load e store*) i primi 4 bit sono sempre di codice condizionale (vedi Tabella 6.1). I seguenti 2 bit che determinano il tipo di operazione saranno 01. Il primo bit dei 6 bit *FUNCT*, ovvero il bit *I* che indica se l'ultimo operando è un valore immediato è negato a differenza delle istruzioni operative. Seguono 5 bit di controllo, rispettivamente

Seguono poi i due registri (4 bit), destinazione e base. L'ultimo operando comprende i 12 bit rimanenti e indica un valore immediato o un ulteriore registro.

<i>opcode</i> : Memory Data Transfer Instruction Control Codes		
bit	meaning	example
P	if set, use pre-index	ldr r0, [r1, #4]
U	if set, use negative offset	ldr r0, [r1, -r2]
B	if set, load a single byte instead of 4	ldrb r0, [r1]
W	if set, write back the index+offset in the index register	
	used combined with P, such that	
PW = 00	Use post-index	ldr r0, [r1], #4
PW = 01	Not used	
PW = 10	Use offset normally	
PW = 11	Use pre-index	ldr r0, [r1, #4]!
L	if set, load, otherwise store	ldr r0, [r1] str r0, [r1]

Tabella 6.3: Bit di controllo di un'istruzione di memoria

Definizione 6.5.3. Struttura delle istruzioni di salto Nelle istruzioni di salto i primi 4 bit sono sempre di condizione, i bit di operazione sono settati a 01. Seguono due bit che indicano il tipo di salto, vedremo solo le configurazioni 10 (*branch*) e 11 (*branch and link*). Nei rimanenti 24 bit è contenuto il valore immediato di offset (positivo o negativo) da sommare al valore corrente del registro *Program Counter*.

Definizione 6.5.4. Istruzioni thumb Nella parola di stato *CPSR* è presente un bit che indica se il processore sta operando in modalità *thumb*, realizzata da ARM per realizzare le stesse istruzioni utilizzando meno bit, e di conseguenza risparmiando memoria ed energia. Nella modalità *thumb* sono disponibili soltanto 8 registri generali, con in più SP e PC. In questo modo, servono solo 3 bit per indicare un registro. Per le istruzioni operative, se il registro di destinazione e il registro sorgente sono lo stesso, l'istruzione viene codificata indicando solo il registro destinazione, risparmiando 3 bit. La lunghezza degli operandi immediati viene ridotta se possibile. In modalità *thumb*, l'esecuzione condizionale è possibile solo per le istruzioni di salto

6.6 Istruzioni in virgola mobile

Definizione 6.6.1. Registri in virgola mobile Mentre i registri **r0-r15** si usano per mantenere valori interi in complemento a 2 da 32 bit, esistono 32 registri da 32 bit per i valori in virgola mobile (*float*), **s0-s31**. Possono essere usati due a due (**s0-s1**, **s2-s3**) come 16 registri a doppia precisione (*double*), **d0-d15**.

Definizione 6.6.2. Istruzioni su registri virgola mobile Per operare sui registri in virgola mobile, esistono i corrispondenti delle istruzioni operative regolari (sui valori interi), a cui va prefisso il carattere *v*. Ad esempio: **vabs**, **vadd**, **vmov**, **vsub**, **vcmp**, ... Si specifica anche la lunghezza del valore sul quale si desidera operare, ad esempio **vadd.f32 s0, s1**.

Vale anche per le istruzioni di memoria: **VLDR**, **VSTR**

Capitolo 7

Microarchitettura

7.1 Datapath

Il ciclo di esecuzione di un processore è

```
while(true) {  
    Instruction = fetch(PC) // PC = program counter  
    decode(Instruction)  
    exec(Instruction)  
    update(PC)  
}
```

Vedremo come implementare un piccolo processore che esegue un sottoinsieme delle istruzioni ARM, suddiviso in due oggetti, la parte di controllo e il **datapath** (parte operativa).

I processori possono essere di diversi tipi:

- **Single cycle:** esegue un singolo ciclo fetch-decode-execute per ogni ciclo di clock (tutto viene eseguito nell'intervallo di tempo in cui il segnale di clock è basso).
- **Multi cycle:** può eseguire un'istruzione in più cicli di clock, in genere, un ciclo per il fetch-decode, un ciclo per l'exec e un ciclo per il writeback (risposta)
- **Pipeline:**

Nota. **Prestazioni**

La misura *CPI* (Clock per Instruction) misura quanti cicli di clock τ sono necessari per eseguire un'istruzione. Da tale misura possiamo dedurre, per ogni processore, il tempo necessario per eseguire un programma di N istruzioni. Esso impiegherà $N \cdot CPI \cdot \tau$

7.2 Processori Single Cycle

Per realizzare un processore Single Cycle dobbiamo capire quali componenti (reti logiche e sequenziali) sono necessari per realizzare il datapath. Possiamo inferire da i componenti necessari per mantenere lo stato del processore (Registri e memoria) e l'insieme *ISTR* di istruzioni che vogliamo implementare.

Vediamo i **componenti di stato**, il primo componente da utilizzare è una memoria per le istruzioni che riceverà in input un indirizzo e restituirà in output l'**istruzione corrente**.

Il secondo componente necessario è una memoria dati (una RAM statica) che contiene la memoria sulla quale possiamo fare operazioni di *load* e *store*. Ha bisogno di un solo indirizzo di memoria per la lettura e la scrittura, un input di clock, un input di *write enable*, un indirizzo in input, un valore in input e un valore in output.

Il terzo componente necessario è una memoria multiporta statica che contiene lo stato dei registri. Riceverà in input 3 indirizzi (2 in lettura ed 1 in scrittura), un segnale di clock, uno di write enable, un valore in input e due in output. Se il segnale *write enable* è LOW, l'indirizzo in scrittura viene utilizzato

per la lettura. Sarà necessario un registro separato per mantenere il program counter, che riceverà sempre clock, write enable, input e restituirà il suo contenuto in output.

Definizione 7.2.1. Implementare un'istruzione LDR (Load register)

Vediamo come implementare un'istruzione LDR con offset immediato (pre-indice), prendiamo ad esempio

```
ldr r0, [r1, #4]
```

Avremo il registro pc che punterà all'istruzione ldr. La memoria istruzioni conterrà l'istruzione all'indirizzo puntato da pc. Caricheremo r1 dal primo indirizzo di lettura della *memoria registri* (nel suo output 1), a cui sommeremo nella ALU l'offset costante #4. Caricheremo dall'indirizzo sommato un valore dalla *memoria dati*, che andrà in input alla memoria registri e verrà scritto all'indirizzo di scrittura, in questo caso r0. Se volessimo realizzare un offset variabile (un registro), come ad esempio

```
ldr r0, [r1, r2]
```

Dovremmo utilizzare anche il secondo input/output di lettura della memoria registri, (il registro r2) come operando di somma della ALU. Ciò ci fa notare che è necessario un multiplexer fra *out2* della memoria registri e l'operando immediato per realizzare correttamente l'offset.

Definizione 7.2.2. Implementare un'istruzione STR (Store register)

Implementare un'istruzione di *store* è simile all'implementazione di un'istruzione di *load*. Il segnale *write enable* della memoria registri sarà low. Leggerò dal primo output della memoria registri l'indirizzo in cui memorizzerò il valore, dal secondo output della memoria registri il valore da memorizzare e opzionalmente, un registro di offset dal terzo output. Il segnale *write enable* della memoria dati sarà HIGH.

Definizione 7.2.3. Implementare istruzioni di salto

Per implementare le istruzioni di salto dobbiamo sommare un immediato all'indirizzo corrente contenuto nel program counter, ed inserirlo di nuovo all'interno del program counter. Abbiamo bisogno di un multiplexer posto fra l'uscita della memoria dati e l'uscita della ALU posta dopo gli output della memoria registri. Collegheremo l'uscita di tale multiplexer all'ingresso del program counter, dove scriveremo il valore dell'istruzione dopo il salto.

Alla fine di un'istruzione *non di salto* il program counter viene incrementato di 4 posizioni attraverso una ALU. Ciò ci dice che è necessario avere un altro multiplexer in ingresso al program counter.

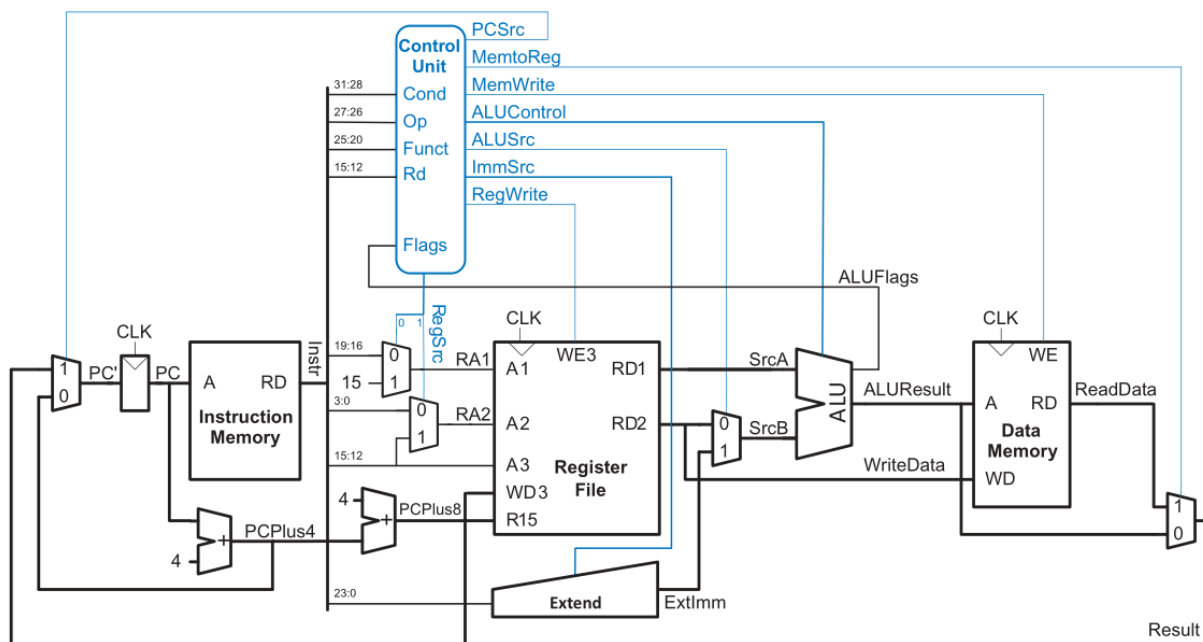


Figura 7.1: Processore a ciclo singolo completo