

ALGEBRA DI BOOLE E RETI COMBINATORIE

→ Sono i moltissimi fondamentali di un calcolatore;

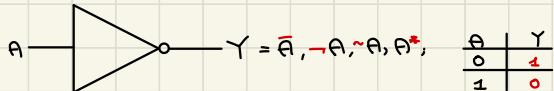
PORTE LOGICHE

→ Realizzano delle semplici operazioni che prendono uno o più INPUT e producono un OUTPUT;

Poiché gli input e gli output possono assumere sia 0 che 1 sono costituzionali delle **variabili binarie**;

PORTA NOT

→ Restituisce in output il complemento dell'input;



A	B	Y
0	0	0
0	1	0
1	0	0
1	1	1

A	B	Y
0	0	0
0	1	1
1	0	1
1	1	1

A	B	Y
0	0	0
0	1	1
1	0	1
1	1	0

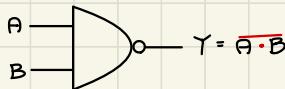
A	B	Y
0	0	0
0	1	1
1	0	1
1	1	0

A	B	Y
0	0	1
0	1	0
1	0	0
1	1	0

A	B	Y
0	0	1
0	1	0
1	0	0
1	1	0

PORTA NAND

→ $Y=1 \Leftrightarrow A \circ B$ uguali a 0;



A	B	Y
0	0	1
0	1	0
1	0	0
1	1	0

A	B	Y
0	0	1
0	1	0
1	0	0
1	1	0

CIRCUITI DIGITALI

→ Reti che elaborano segnali discreti;

→ Un circuito può essere visto come una **BLACK-BOX** con:

- Uno o più input
- Uno o più output
- Una specifica funzionale che rappresenta la relazione fra input e output.
- Una specifica temporale che descrive le istantanee che intercorrono

affinché i segnali di input si propagino nel circuito fino agli output.

La struttura interna di un circuito è composta da **Elementi e Nodi**.

Un elemento è esso stesso un circuito digitale.

Sono connessioni che

trasportano i segnali;

Vi sono 2 grandi categorie di CIRCUITI DIGITALI:

RETI COMBINATORIE

RETI SEQUENZIALI

REGOLE DI COMPOSIZIONE DI RETI COMBINATORIE:

- Ogni elemento è esso stesso una rete combinatoria;

In una rete combinatoria i valori degli output dipendono solo dai valori corrente degli input, infatti in questo caso le reti si dicono **memoryless**, cioè senza memoria;

In una rete sequenziale, invece, gli output dipendono non solo dai valori corrente degli input, ma anche dai valori precedenti. Si dice quindi che il circuito ha **memoria**;

- Ogni nodo che non è un input commette assolutamente un output di un elemento;
- Il circuito non contiene cicli, ogni cammino interno alla rete visita un nodo di più una volta;

Esercizi:

1)

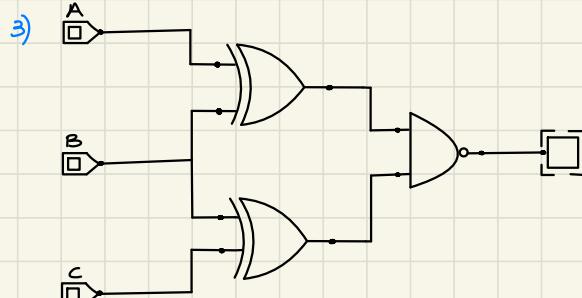
$$(\bar{A} \cdot B) + (B \oplus C)$$

A	B	C	\bar{A}	$\bar{A} \cdot B$	$B \oplus C$	$(\bar{A} \cdot B) + (B \oplus C)$
0	0	0	1	0	0	0
0	0	1	1	0	1	1
0	1	0	1	1	1	1
0	1	1	1	0	0	1
1	0	0	0	0	0	0
1	0	1	0	0	1	1
1	1	0	0	0	1	1
1	1	1	0	0	0	0

2) $(A \cdot \bar{B}) + (\bar{A} \cdot B) = \text{xor}$

A	B	\bar{A}	\bar{B}	$A \cdot \bar{B}$	$\bar{A} \cdot B$	$(A \cdot \bar{B}) + (\bar{A} \cdot B)$	xor
0	0	1	1	0	0	0	0
0	1	1	0	0	1	1	1
1	0	0	1	1	0	1	1
1	1	0	0	0	0	0	0

=	xor
	0
	1
	1
	0



Soluzione: $(A \oplus B) \cdot (B \oplus C)$

A	B	C	$A \oplus B$	$B \oplus C$	$(A \oplus B) \cdot (B \oplus C)$	$(\bar{A} \oplus B) \cdot (\bar{B} \oplus C)$
0	0	0	0	0	0	1
0	0	1	0	1	0	1
0	1	0	1	1	1	0
0	1	1	1	0	0	1
1	0	0	1	0	0	1
1	0	1	1	1	1	0
1	1	0	0	1	0	1
1	1	1	0	0	0	1

DEFINIZIONI:

- **LITERALE** \rightarrow Variabile booleana A o la sua negata \bar{A} ;
- **IMPONENTE** \rightarrow Un prodotto (AND) di literali;
- **MINTERMINE** \rightarrow Dato un insieme di variabili K, è un impONENTE che comprende tutte le variabili in K;
- **MAXTERMINE** \rightarrow È una somma di literali in cui occorrono tutte le variabili in K;

Forma SOP (Sum-of-Products)

Ogni riga delle 2ⁿ righe di una tabella di verità è contenuta da un mintermone.

I mintermimi sono enumenati riga dopo riga a partire da 0, 1 e così via.

Quindi ogni mintermone è denotato dal numero binario della configurazione di input corrispondente;

Dunque, conseguenza ad ogni tabella corrisponde una espressione booleana ottenuta sommando tutti i mintermimi per cui la valore di Y = 1;

A	B	Y	mintermone
0	0	0	$\bar{A} \bar{B}$
0	1	1	$\bar{A} B$
1	0	0	$A \bar{B}$
1	1	1	$A B$

$$Y = \bar{A} \bar{B} + AB$$

$$Y = \Sigma(1,3)$$

Esempio:

A	B	C	Y
0	0	0	1
0	0	1	0
0	1	0	0
0	1	1	0
1	0	0	1
1	0	1	1
1	1	0	0
1	1	1	0

$$Y = \bar{A} \bar{B} \bar{C} + A \bar{B} \bar{C} + A \bar{B} C$$

$$Y = \Sigma(0,4,5)$$

Forma POS (Product-of-Sums)

↓

Fornita una rappresentazione una funzione booleana è in forma POS;

Ad ogni riga di una tabella di verità corrisponde un mintermone che è uguale a 0 solo per quella riga.

A	B	Y	maxtermone
0	0	0	$A + B$
0	1	1	$A + \bar{B}$
1	0	0	$\bar{A} + B$
1	1	1	$\bar{A} + \bar{B}$

$$Y = (A + B) \cdot (\bar{A} + B)$$

$$Y = \prod(0,2)$$

Da forma normale POS di una funzione booleana si ottiene come prodotto dei maxtermi per cui la funzione ritorna 0;

ASSIOMI BOOLEANI

1 -	$B = 0$ se $B \neq 1$	$B = 1$ se $B \neq 0$	
2 -	$\bar{0} = 1$	$\bar{1} = 0$	NOT
3 -	$0 \cdot 0 = 0$	$1 + 1 = 1$	AND/OR
4 -	$1 \cdot 1 = 0$	$0 + 0 = 0$	AND/OR
5 -	$0 \cdot 1 = 1 \cdot 0 = 0$	$1 + 0 = 0 + 1 = 1$	AND/OR

TEOREMI BOOLEANI AD UNA VARIABILE

1 -	$B \cdot 1 = B$	$B + 0 = B \rightarrow$ IDENTITÀ
2 -	$B \cdot 0 = 0$	$B + 1 = 1 \rightarrow$ ELEMENTO NUOLO
3 -	$B \cdot B = B$	$B + B = B \rightarrow$ IDEMESTERIA
4 -	$\bar{\bar{B}} = B$	$\bar{B} = B \rightarrow$ INVOLUZIONE
5 -	$B \cdot \bar{B} = 0$	$B + \bar{B} = 1 \rightarrow$ COMPLEMENTO

TEOREMI BOOLEANI CON DIVERSE VARIABILI

6 -	$B \cdot C = C \cdot B$	$B + C = C + B \rightarrow$ COMMUTATIVITÀ
7 -	$(B \cdot C) \cdot D = B \cdot (C \cdot D)$	$(B+C)+D = B+(C+D) \rightarrow$ ASSOCIAZIONE
8 -	$B \cdot (C + D) = (B \cdot C) + (B \cdot D)$	$B + (C \cdot D) = (B+C) \cdot (B+D) \rightarrow$ DISTRIBUTIVITÀ
9 -	$B \cdot (B + C) = B$	$B + (B \cdot C) = B \rightarrow$ ASSORBIMENTO
10 -	$(B \cdot C) + (B \cdot \bar{C}) = B$	$(B+C) \cdot (B+\bar{C}) = B \rightarrow$ COMBINAZIONE
11 -	$(B \cdot \bar{C}) + (\bar{B} \cdot D) + (C \cdot D) = (B \cdot C) + (\bar{B} \cdot D) + (C \cdot D) =$ $(B \cdot C) + (\bar{B} \cdot D)$	$(B+C) \cdot (\bar{B}+D) \cdot (C+D) = \rightarrow$ CONSENTO

TECNICHE DI DEMOSTRAZIONE DELL'EGUAGLIANZA DI 2 ESPRESSIONI



Perfect Induction

Coé se la tabella

di verità di 2 espressioni
coincide allora le 2 espressioni
sono equivalenti;



USARE ASSIOMI E TEOREMI

PER MANIPOLARE LE ESPRESSIONI

Fino ad ottenere espressioni
uguali;

Esempio:

Perfect Induction

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

Step 1:

$B \cdot (B+C) =$

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

$B + BC =$

$B(1+C) =$

$B \cdot 1 =$

$B =$

B

Step 2:

$B \cdot (B+C) =$

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

$B + BC =$

$B(1+C) =$

$B \cdot 1 =$

$B =$

B

Step 3:

$B \cdot (B+C) =$

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

$B + BC =$

$B(1+C) =$

$B \cdot 1 =$

$B =$

B

Step 4:

$B \cdot (B+C) =$

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

$B + BC =$

$B(1+C) =$

$B \cdot 1 =$

$B =$

B

Step 5:

$B \cdot (B+C) =$

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

$B + BC =$

$B(1+C) =$

$B \cdot 1 =$

$B =$

B

Step 6:

$B \cdot (B+C) =$

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

$B + BC =$

$B(1+C) =$

$B \cdot 1 =$

$B =$

B

Step 7:

$B \cdot (B+C) =$

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

$B + BC =$

$B(1+C) =$

$B \cdot 1 =$

$B =$

B

Step 8:

$B \cdot (B+C) =$

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

$B + BC =$

$B(1+C) =$

$B \cdot 1 =$

$B =$

B

Step 9:

$B \cdot (B+C) =$

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

$B + BC =$

$B(1+C) =$

$B \cdot 1 =$

$B =$

B

Step 10:

$B \cdot (B+C) =$

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

$B + BC =$

$B(1+C) =$

$B \cdot 1 =$

$B =$

B

Step 11:

$B \cdot (B+C) =$

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

$B + BC =$

$B(1+C) =$

$B \cdot 1 =$

$B =$

B

Step 12:

$B \cdot (B+C) =$

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

$B + BC =$

$B(1+C) =$

$B \cdot 1 =$

$B =$

B

Step 13:

$B \cdot (B+C) =$

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

$B + BC =$

$B(1+C) =$

$B \cdot 1 =$

$B =$

B

Step 14:

$B \cdot (B+C) =$

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

$B + BC =$

$B(1+C) =$

$B \cdot 1 =$

$B =$

B

Step 15:

$B \cdot (B+C) =$

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

$B + BC =$

$B(1+C) =$

$B \cdot 1 =$

$B =$

B

Step 16:

$B \cdot (B+C) =$

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

$B + BC =$

$B(1+C) =$

$B \cdot 1 =$

$B =$

B

Step 17:

$B \cdot (B+C) =$

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

$B + BC =$

$B(1+C) =$

$B \cdot 1 =$

$B =$

B

Step 18:

$B \cdot (B+C) =$

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

$B + BC =$

$B(1+C) =$

$B \cdot 1 =$

$B =$

B

Step 19:

$B \cdot (B+C) =$

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

$B + BC =$

$B(1+C) =$

$B \cdot 1 =$

$B =$

B

Step 20:

$B \cdot (B+C) =$

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

$B + BC =$

$B(1+C) =$

$B \cdot 1 =$

$B =$

B

Step 21:

$B \cdot (B+C) =$

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

$B + BC =$

$B(1+C) =$

$B \cdot 1 =$

$B =$

B

Step 22:

$B \cdot (B+C) =$

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

$B + BC =$

$B(1+C) =$

$B \cdot 1 =$

$B =$

B

Step 23:

$B \cdot (B+C) =$

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

$B + BC =$

$B(1+C) =$

$B \cdot 1 =$

$B =$

B

Step 24:

$B \cdot (B+C) =$

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

$B + BC =$

$B(1+C) =$

$B \cdot 1 =$

$B =$

B

Step 25:

$B \cdot (B+C) =$

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

$B + BC =$

$B(1+C) =$

$B \cdot 1 =$

$B =$

B

Step 26:

$B \cdot (B+C) =$

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

$B + BC =$

$B(1+C) =$

$B \cdot 1 =$

$B =$

B

Step 27:

$B \cdot (B+C) =$

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

$B + BC =$

$B(1+C) =$

$B \cdot 1 =$

$B =$

B

Step 28:

$B \cdot (B+C) =$

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

$B + BC =$

$B(1+C) =$

$B \cdot 1 =$

$B =$

B

Step 29:

$B \cdot (B+C) =$

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

$B + BC =$

$B(1+C) =$

N.B.: Bisogna però dire che la tecnica del perfect induction è semplice ma "poco di intelligenza".

Inoltre ad crescere della lunghezza delle espressioni e del numero di variabili diventa sempre più difficile.

METODI DI SEMPLIFICAZIONE

• DISTRIBUTIVITÀ $\rightarrow B \cdot (C + D) = (B \cdot C) + (B \cdot D)$ / $B + (C \cdot D) = (B + C) \cdot (B + D)$

• ASSORBIMENTO $\rightarrow A + (A \cdot P) = A$

• COMBINAZIONE $\rightarrow (P \cdot \bar{A}) + (P \cdot A) = P$

• ESPANSIONE $\rightarrow P = (P \cdot \bar{A}) + (P \cdot A) \rightarrow A = A + (A \cdot P)$

• DOPPIAZIONE $\rightarrow A = A + A$

TEOREMA DI "SEMPLIFICAZIONE" $\rightarrow \overline{P \cdot A} + A = P + A$ / $P \cdot A + A = P + \bar{A}$

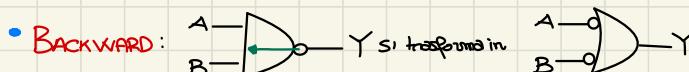
TEOREMA DI DE MORGAN

$$\overline{B_0 \cdot B_1 \cdot B_2 \dots} = \overline{B_0} + \overline{B_1} + \overline{B_2} \quad / \quad \overline{B_0 + B_1 + B_2} = \overline{B_0} \cdot \overline{B_1} \cdot \overline{B_2}$$

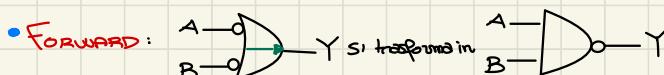
■ La negata di un prodotto è uguale alla somma delle negate;

■ La negata di una somma è uguale al prodotto delle negate;

BUBBLE PUSHING



Esempio: $\overline{A \cdot B} = \overline{A} + \overline{B}$



Esempio:

$$\overline{A} + \overline{B} = \overline{A \cdot B}$$

Esercizi

1)

a) $A \quad B \quad Y$

0	0	1
0	1	0
1	0	1
1	1	1

$A \cdot B$

b) $A \quad B \quad C \quad Y$

0	0	0	1
0	0	1	0
0	1	0	0
0	1	1	0
1	0	0	0
1	0	1	0
1	1	0	0
1	1	1	1

$\bar{A} \quad \bar{B} \quad \bar{C}$

c) $A \quad B \quad C \quad Y$

0	0	0	1
0	0	1	0
0	1	0	0
0	1	1	0
1	0	0	1
1	0	1	1
1	1	0	0
1	1	1	1

$\bar{A} \quad \bar{B} \quad \bar{C}$

SOP

$$Y = (\bar{A} \cdot \bar{B}) + (A \cdot \bar{B}) + (A \cdot B)$$

POS

$$Y = \bar{A} + B$$

SOP

$$Y = (\bar{A} \cdot \bar{B} \cdot \bar{C}) + (A \cdot B \cdot C)$$

POS

$$Y = (\bar{A} + B + C) \cdot (\bar{A} + B + \bar{C}) \cdot (\bar{A} + \bar{B} + C) \cdot (A + \bar{B} + \bar{C}) \cdot (A + \bar{B} + C) \cdot (A + B + \bar{C})$$

SOP

$$Y = (\bar{A} \cdot \bar{B} \cdot \bar{C}) + (\bar{A} \cdot B \cdot \bar{C}) + (A \cdot \bar{B} \cdot \bar{C}) + (A \cdot \bar{B} \cdot C)$$

POS

$$Y = (\bar{A} + \bar{B} + C) \cdot (\bar{A} + B + C) \cdot (A + B + \bar{C})$$

Semplificazione di c)

$$Y = \underbrace{(\bar{A}\bar{B}\bar{C})}_{\bar{AC}} + \underbrace{(\bar{A}\bar{B}C)}_{AB} + \underbrace{(A\bar{B}\bar{C})}_{AC} + \underbrace{(A\bar{B}C)}_{AC} = \bar{A}\bar{C} + A\bar{B} + AC$$

2)

	AB	00	01	11	10
CD	00	1	0	1	(1)
00	01	0	1	0	1
01	11	1	0	0	0
11	10	1	1	0	0
10	11	1	0	1	1

$$Y = \bar{B}\bar{D} + \bar{A}C + A\bar{B}\bar{C} + A\bar{C}\bar{D} + \bar{A}B\bar{D}$$

3)

$$A(B \oplus C) = 1 \quad 0 \quad A+B+C=0$$

A	B	C	D	Y
0	0	0	0	0
0	0	0	1	1
0	0	1	0	0
0	0	1	1	1
0	1	0	0	1
0	1	0	1	0
0	1	1	0	1
0	1	1	1	0
1	0	0	0	0
1	0	0	1	1
1	0	1	0	0
1	0	1	1	1
1	1	0	0	0
1	1	0	1	1
1	1	1	0	1
1	1	1	1	0

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

$$Y = \bar{A}D + BC + \bar{A}B + A\bar{B}C$$

REGOLE SCHEMATICHE DEI CIRCUITI

- Input a SINISTRA (o sopra);
- Output a DESTRA (o sotto);
- Le piste vanno da SINISTRA verso DESTRA;
- I collegamenti dritti sono i migliori;
- I fili di collegamento sempre come una aggiunzione T;
- Un punto in cui 2 fili si incontrano indica una commissione tra i 2 fili;
- I fili che si incontrano senza punto non sono commessi;

CIRCUITO A PRIORITY → Vengono utilizzati per assegnare una risorsa condivisa secondo un ordine di priorità fra chi ne fa richiesta;

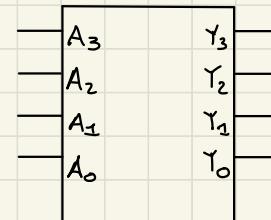


Esempio: Posso avere 4 possibili richiedenti con priorità $3 > 2 > 1 > 0$

A_0, \dots, A_3 sono le richieste della risorsa,

Y_0, \dots, Y_3 rappresentato da chi è assegnata la risorsa e

di volta in volta uno di loro sarà uguale ad 1



MAPPE DI KARNAUGH → Sono un metodo per semplificare le espressioni booleane in forma SOP;

- In realtà non introduciamo tecniche di semplificazione, ma semplicemente sono grafici, che consentono di rilevare più facilmente gli impieghi che possono essere semplificati;

→ Alla base vi è il principio: $PA + P\bar{A} = P$

→ Condizione: 1 nei quadrati adiacenti e nelle espressioni booleane includere solo i littori VARI che non hanno lo stesso condito;

A	B	C	Y
0	0	0	1
0	0	1	1
0	1	0	0
1	1	0	0
1	0	0	0
1	0	1	0
1	1	0	0
1	1	1	0

$Y = \bar{A}\bar{B}\bar{C} + \bar{A}\bar{B}C + AB\bar{C}$ → $Y = \bar{A}B$

REGOLE DI KARNAUGH

- Ogni 1 deve essere condotto almeno una volta;
- Ogni condito deve coprire una potenza di 2, cioè 1, 2, 4 quadrati in ogni direzione;
- Ogni condito deve essere il più grande possibile;
- Un condito può avere più bordi;
- Vi può essere il simbolo X (don't care) che viene considerato solo se aiuta a minimizzare l'espressione;

Esercizi (lezione 9 pag. 24):

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

$$BD + \bar{B}\bar{D} + \bar{A}\bar{B}$$

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

$$\bar{B} + CD + \bar{A}D$$

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

$$\bar{A}\bar{B}D + CD + BD + AD$$

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

Dovrebbe essere

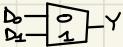
$$B + C$$

MUX → È sostanzialmente un selettore di linea;

In generale è costituito da N ingressi, 1 uscita e $\log_2(N)$

linee di selezione che indicano a quale ingresso deve corrispondere l'output;

Esempio (2:1 Mux):



S	D ₁	D ₂	Y
0	0	0	1
0	0	1	1
0	1	0	0
0	1	1	0
1	0	0	0
1	0	1	0
1	1	0	0
1	1	1	0

S	Y
0	D ₀
1	D ₁

Altri ESEMPI: lezione 9 pag. 30

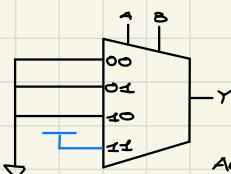
I multiplexer possono essere usati anche per sintetizzazione delle funzioni booleane;

Di fatto le linee di ingresso mi producono la TABELLA DI VERITÀ;

Esempio:

A	B	Y
0	0	0
0	1	0
1	0	0
1	1	1

$$Y = AB$$



Altri ESEMPI: lezione 9 pag. 33

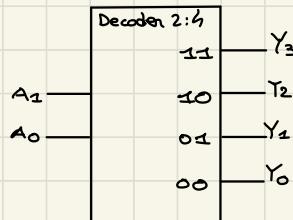
DECODER → Ha N linee di ingresso e 2^N linee di uscita;

Se nn è un numero rappresentato dagli input allora solo l'nn-esima linea di uscita è pari a 1 mentre tutte le altre sono pari a 0;

Ogni uscita rappresenta un MINTERMINO;

Esempio:

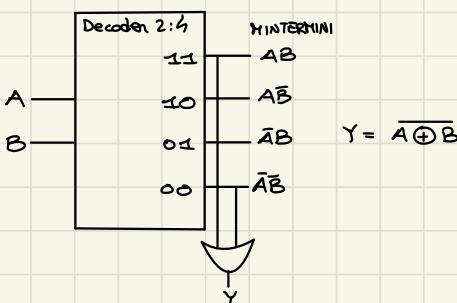
A_1	A_0	Y_3	Y_2	Y_1	Y_0
0	0	0	0	0	1
0	1	0	0	1	0
1	0	0	1	0	0
1	1	1	0	0	0



Anche i decoden possono essere usati per sintesi delle funzioni booleane;

Basta mettere in OR tutte e solo le linee di uscita che occorrono nella sigma-expresione della funzione da sintetizzare.

Esempio:



Esercizi 31/03/2023:

$$C + B\bar{C} = B + C$$

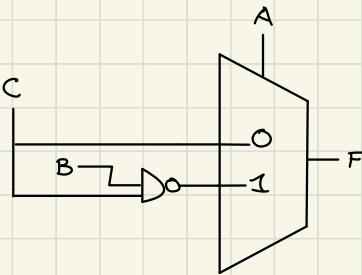
Formula da applicare → $A + \bar{A}P = A + P$;

$$1) Y = AC + \bar{A}\bar{B}C = C(A + \bar{A}\cdot\bar{B}) = C(A + \bar{B}) = AC + \bar{B}C;$$

$$2) Y = \bar{A}\bar{B} + \bar{A}B\bar{C} + (\bar{A} + C) = \bar{A}\bar{B} + \bar{A}B\bar{C} + (\bar{A} \cdot \cancel{C}) = \bar{A}\bar{B} + \bar{A}B\bar{C} + \bar{A}C = \bar{A}(\bar{B} + B\bar{C} + C) = \bar{A}(\bar{B} + B\bar{C} + B\bar{C} + C) = \\ = \bar{A}(\cancel{\bar{B} + \cancel{B}\bar{C} + \cancel{B}C}) = \bar{A}$$

3) $F = A\bar{B} + \bar{A}C + ABC$

A	B	C	\bar{A}	\bar{B}	\bar{C}	$A \cdot \bar{B}$	$\bar{A} \cdot C$	$A \cdot B \cdot C$	$\bar{A} \cdot \bar{B} + \bar{A} \cdot C + A \cdot B \cdot C$	F
0	0	0	1	1	1	0	0	0	0	0
0	0	1	1	1	0	0	1	0	1	1
0	1	0	1	0	1	0	0	0	0	0
0	1	1	1	0	0	0	1	0	1	1
1	0	0	0	1	1	1	0	0	1	1
1	0	1	0	1	0	1	0	0	1	1
1	1	0	0	0	1	0	0	1	1	1
1	1	1	0	0	0	0	0	0	0	0



4) A cosa è uguale $\bar{A}\bar{B}C + \bar{A}BC + AC$??

- $C / - A / - D / - B / - AD /$

• $\bar{A}\bar{B}C + \bar{A}BC + AC = C (\frac{\bar{A}\bar{B}}{\bar{A}} + \frac{\bar{A}B}{A} + A) = C (\frac{\bar{A} + A}{1}) = C (1) = C$

5) $\begin{array}{r} 87,75 \\ \downarrow \\ 10100111.11 \end{array}$ $\begin{array}{r} 87 \mid 2 \\ 43 \mid 2 \\ 21 \mid 2 \\ 10 \mid 2 \\ 5 \mid 2 \\ 2 \mid 2 \\ 1 \mid 2 \\ 0 \mid 2 \\ 1 \mid 0 \end{array}$ $0,75 \cdot 2 = 1,5$
 $0,5 \cdot 2 = 1$
 $87 = 10100111$

$0,75 = 11$ $10100111.11 = 1.01011111 \cdot 10^6$

• Segno : 0

• Esponente : $6 + 127 = 133 \rightarrow 10000101$

• Mantissa : 0101 1111 1000 0000 0000 0000

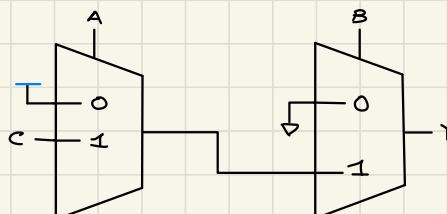
AB	CD	00	01	11	10
x	x	0	0	x	x
00	0	1	x	0	0
01	0	x	1	0	0
11	1	x	0	x	x
10	1	x	0	x	x

$$Y = \bar{B}\bar{D} + BD$$

$$\bar{B}\bar{C}\bar{D} + \bar{B}\bar{C}D + \cancel{\bar{B}C\bar{D}} + \cancel{\bar{B}CB} + \bar{B}CD + B\bar{C}D + BCD$$

$$\bar{B}\bar{C}\bar{D} + \cancel{\bar{B}C\bar{D}} + \cancel{\bar{B}CB} + \bar{B}CD + B\bar{C}D + BCD$$

A	B	C	Y
0	0	0	0
0	0	1	0
0	1	0	1
0	1	1	1
1	0	0	0
1	0	1	0
1	1	0	0
1	1	1	1



8)

A	B	C	D	F
0	0	0	0	1
0	0	0	1	0
0	0	1	0	1
0	0	1	1	0
0	1	0	0	1
0	1	0	1	0
0	1	1	0	1
0	1	1	1	0
1	0	0	0	1
1	0	0	1	0
1	0	1	0	1
1	0	1	1	0
1	1	0	0	1
1	1	0	1	0
1	1	1	0	1
1	1	1	1	0

$$SOP: \bar{A}\bar{B}\bar{C}\bar{D} + \bar{A}\bar{B}\bar{C}\bar{D} + \bar{A}\bar{B}\bar{C}\bar{D} + \bar{A}\bar{B}\bar{C}\bar{D} + A\bar{B}\bar{C}\bar{D} + A\bar{B}\bar{C}\bar{D} + A\bar{B}\bar{C}\bar{D} + A\bar{B}\bar{C}\bar{D}$$

$$POS: (\bar{A} + \bar{B} + \bar{C} + D) \cdot (\bar{A} + \bar{B} + C + D) \cdot (\bar{A} + B + \bar{C} + D) \cdot (\bar{A} + B + C + D) \cdot (A + \bar{B} + \bar{C} + D) \cdot (A + \bar{B} + C + D) \cdot$$

$$\cdot (A + B + \bar{C} + D) \cdot (A + B + C + D)$$

TIMING:

Il **DELAY** è il tempo tra la modifica dell'input e la modifica dell'output.

CIRCUITI SEQUENZIALI:

Logica Sequenziale:

Nel sistemi sequenziali l'output dipende sia dal valore corrente sia dai valori precedenti dell'input, in tal senso si dice che il sistema ha **MEMORIA**.

Lo **STATO INTERNO**, rappresenta l'informazione che mantiene lo stato di un circuito sequenziale ed è necessario per prevedere le sue comportamenti. Un **circuito sequenziale** inoltre ha una tipologia ben precisa, cioè è composto da:

- **LOGICA COMBINATORIA**: che definisce l'evoluzione del sistema;
- **BANCHI DI FLIP-FLOP**: che servono a memorizzare gli STATI DEL SISTEMA;

In fine un aspetto importante dei sistemi sequenziali è quello del **FEEDBACK**, ovvero i segnali di output vengono riportati in input;

STATE ELEMENTS:

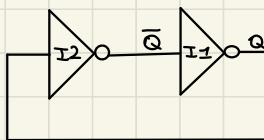
Lo STATO di un circuito influisce sull'evoluzione del sistema;

Gli **STATE ELEMENTS** sono tutte quelle componenti circolari che vengono adoperate per memorizzare lo STATO di un circuito;

Possiamo avere diversi tipi di circuiti come:

- **CIRCUITI BISTABILI**
- **SR LATCH**
- **D LATCH**
- **D FLIP-FLOP**

CIRCUITO BISTABILE:



Considera 2 casi possibili:

- $Q = 0$:

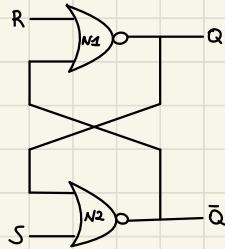
allora $Q = 0, \bar{Q} = 1$

- $Q = 1$:

allora $Q = 1, \bar{Q} = 0$

- Non ha INPUT!!
- Ha 2 output: Q, \bar{Q} ;
- Può essere utilizzato anche come **blocco di costruzione per altri STATE ELEMENTS**;
- Memorizza 1 bit nella variabile di stato Q (o \bar{Q})

SR (SET/RESET) LATCH:

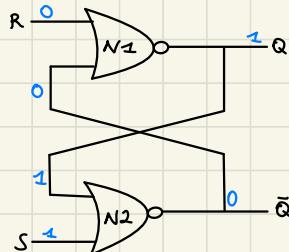


Consideriamo 4 possibili stati:

- Se $S = 1, R = 0$

allora $Q = 1 \text{ e } \bar{Q} = 0$

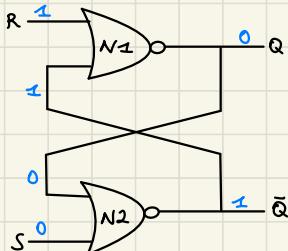
Operazione Set



- Se $S = 0, R = 1$

allora $Q = 0 \text{ e } \bar{Q} = 1$

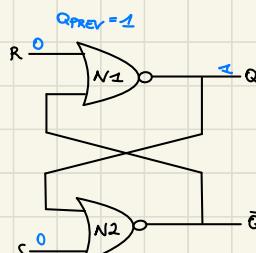
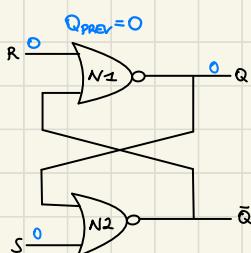
Operazione Reset



- Se $S = 0, R = 0$

allora $Q = Q_{prev}$

Memoria

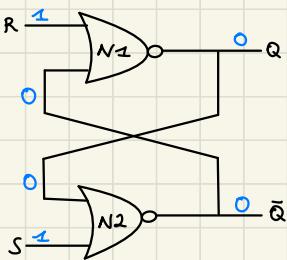


$S_2 S=1, R=1$

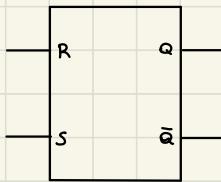
allora $Q=0, \bar{Q}=0$

Stato non valido

$Q \neq \text{NOT } \bar{Q}$



SIMBOLO PER UN SR LATCH:



SR sta per Set/Reset

memorizza un bit (a)

SET: Pone l'output a 1 ($S=1, R=0, Q=1$)

RESET: Pone l'output a 0 ($S=0, R=1, Q=0$)

MEMORIA: Mantiene memoria dell'output ($S=0, R=0, Q=Q_{\text{PREV}}$)

Occorre evitare lo stato non valido $S=R=1$

D LATCH

* 2 INPUT : CLK, D

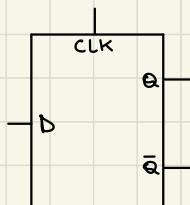
$\xrightarrow{\text{CLK}=1} D$ passa fino a Q (trasparente)

* CLK : controlla quando l'output cambia $\rightarrow \text{CLK}=0 \rightarrow Q$ mantiene il suo valore precedente (opaco)

* D (data input): controlla in che cosa l'output cambia

Occorre evitare lo stato non valido in cui $Q \neq \text{NOT } \bar{Q}$

SIMBOLO PER UN D LATCH:



D FLIP-FLOP

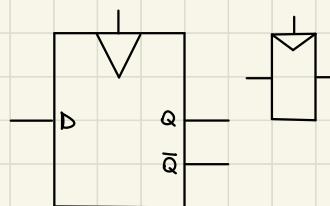
* Input: CLK, D

* Funzione:

- Quando CLK passa da 0 a 1, D passa fino a Q;
- Altri, Q mantiene il suo valore precedente;

* Q cambia solo durante la transizione di CLK da 0 a 1;

SIMBOLO DEL FLIP-FLOP



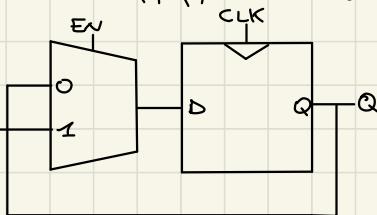
Queste tipologie di componenti sono dette **edge-triggered** perché sono pilotate con un valore minima da una transizione (di CLK).

FLIP-FLOP "ENABLE"

★ Input: CLK, D, EN

★ L'input enable (EN) stabilisce quando un nuovo valore di D è memorizzato

- $EN = 1 \rightarrow D$ passa fino a Q (clock: $0 \rightarrow 1$);
- $EN = 0 \rightarrow$ il flip-flop mantiene il suo stato precedente;

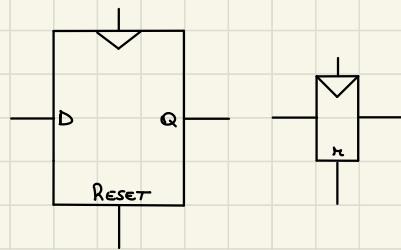


FLIP-FLOP "RESETTABILISI" LEZIONE 11 pag. 10

★ Input: CLK, D, Reset

★ $RESET = 1 : Q = 0$;

★ $RESET = 0 : \text{il flip-flop si comporta "normalmente" come un D flip-flop;}$

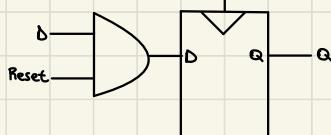


Vi sono 2 tipi di flip-flop "resettabili":

• **SINCRONI:** Il Reset è pilotato dal clock;

• **ASINCRONI:** Il Reset avviene non appena $RESET = 1$

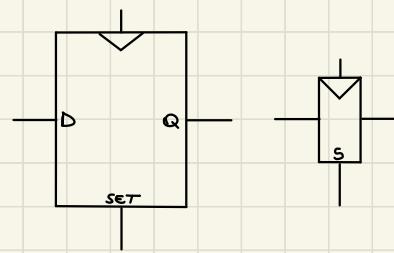
• **Flip-Flop Sincroni:**



• Per i Flip-Flop Asincroni occorre modificare le circuitali interne del flip-flop;

FLIP-FLOP "SETTABILIO"

LEZIONE 11 pag. 12



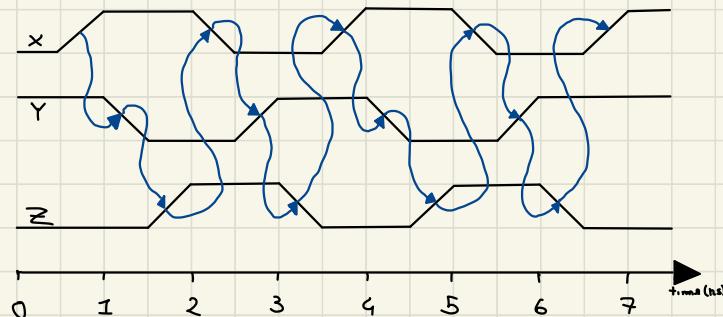
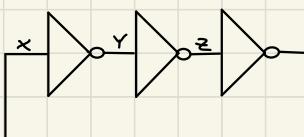
* Input: CLK, D, SET

* $SET = 1 : Q = 0 ;$

* $SET = 0 :$ il flip-flop si comporta "normalmente" come un D flip-flop;

(vedere esercizi lez. 14 pag. 13)

CRITICITÀ NELLA LOGICA SEQUENZIALE



Questi circuiti vengono detti "astabili" poiché hanno un comportamento oscillante, le periodi di oscillazione dipende dai ritardi degli invertori.

Idealmente è di 6 ns, tuttavia può variare a causa delle differenze nella manifattura e della temperatura;

In casi più complessi che comprendono l'uso di più porte AND, NOT, OR, il comportamento di una rete ASINCRONA può dipendere fortemente dai ritardi accumulati sui singoli cammini;

Nei CIRCUITI ASINCRONI, l'output è retroazionato in modo diretto;

LOGICHE SEQUENZIALI SINCRONE

I circuiti asincroni presentano delle criticità a volte difficilmente analizzabili, in quanto, dipendono dalla struttura fisica dei componenti, per questo si cerca di evitare la retroazione dell'output diretta, interponendo un registro nel ciclo di retroazione;

In generale, un CIRCUITO SEQUENZIALE SINCRONO ha un insieme finito di STATI $\{S_0, \dots, S_{k-1}\}$

le design di logiche sequenziali sincrone è composto da:

- **REGISTRI**, che determinano lo STATO S_0, \dots, S_{k-1} del sistema;
- I cambiamenti di stato sono determinati dalle transizioni del clock;

Inoltre nego le seguenti Regole di composizione:

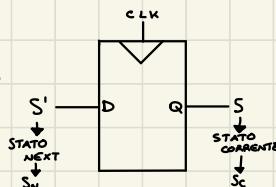
- Ogni componente è un **registro** o un **circuito combinatorio**;
- Almeno un componente è un **registro**;
- Tutti i registri sono sincronizzati con un unico clock;
- Ogni ciclo contiene almeno un **registro**;

Due tipici circuiti sequenziali sincroni sono:

- **Finite State Machines (FSMs)**;
- **Pipelines**;

Un flip-flop D è il più semplice circuito sequenziale sincrono

- $Q = S_c \rightarrow$ STATO CORRENTE
- $D = S_n \rightarrow$ STATO NEXT



FINITE STATE MACHINES (FSM)

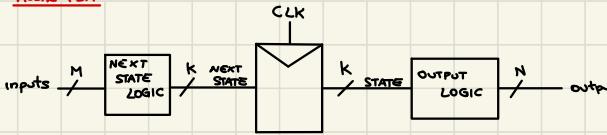
- ★ **STATE REGISTER**
 - Memorizziamo lo stato corrente;
 - Concetto di prossimo stato;
 - "Computa il prossimo stato"; (1)
- ★ **Logica Combinatoria**
 - "Computa" gli output; (2)

★ S_n dipende sia da **INPUT** che dello **STATO CORRENTE** $\rightarrow S_n = g(\text{input}, \text{stato corrente})$;

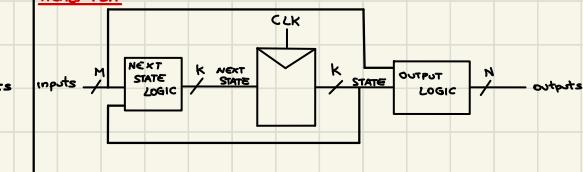
★ Esistono 2 tipi di **FSM** a seconda della logica di output:

- **Moore FSM** \rightarrow output: $f(\text{stato corrente})$;
- **Mealy FSM** \rightarrow output: $f(\text{input}, \text{stato corrente})$;

MOORE FSM



MEALY FSM



Esempio Semaforo:

SENZORI: T_A, T_B (TRUE, quando c'è traffico)

INPUTS: CLK, RESET, T_A, T_B

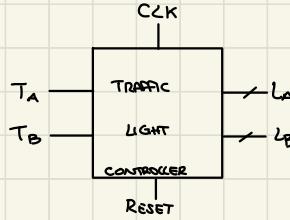
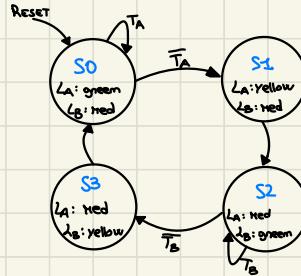
Usc: L_A, L_B

OUTPUTS: L_A, L_B

DIAGRAMMA DI TRANSIZIONE (MOORE FSH).

- STATI: etichettati con gli outputs;

- TRANSIZIONI: etichettate con gli inputs;



FSM STATE TRANSITION TABLE

CURRENT STATE	INPUTS		NEXT STATE
	T_A	T_B	
S0, S0	0	x	S1'
0, 0	0	x	0
0, 0	1	x	0, 0
0, 1	x	x	1, 0
1, 0	x	0	1, 1
1, 0	x	1	1, 0
1, 1	x	x	0, 0

FSM ENCODED STATE TRANSITION TABLE

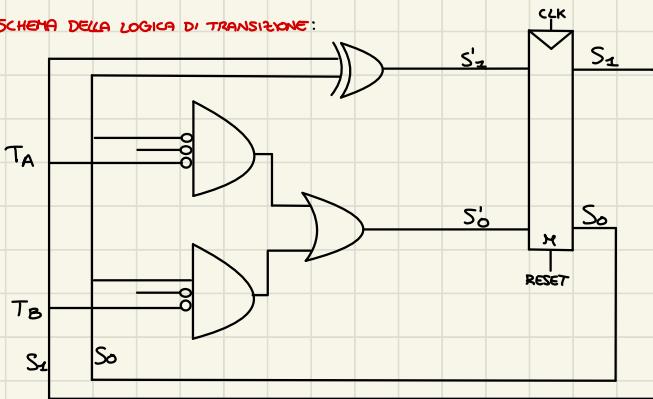
CURRENT STATE	INPUTS	NEXT STATE
S1, S0	T_A T_B	S'_1 S'_0
0, 0	0 x	0 1
0, 0	1 x	0 0
0, 1	x x	1 0
1, 0	x 0	1 1
1, 0	x 1	1 0
1, 1	x x	0 0

STATE	ENCODING
S0	00
S1	01
S2	10
S3	11

$$S'_1 = \overline{S_2}S_0 + S_2\overline{S_0}\overline{T_B} + S_2\overline{S_0}T_B = S_1 \oplus S_0$$

$$S'_0 = \overline{S_2}\overline{S_0}\overline{T_A} + S_2\overline{S_0}\overline{T_B}$$

SCHEMA DELLA LOGICA DI TRANSIZIONE:



ENCODING DEGLI STATI → ENCODING BINARIO: per 4 stati, 00, 01, 10, 11;

- un bit per stato;

ENCODING ONE-HOT:

- Solo un bit HIGH alla volta;

- per 4 stati, 0001, 0010, 0100, 1000;

- Richiede più flip-flops;

- Spesso la logica combinatoria associata è più semplice;

PROGETTARE UNA FSM:

- 1) IDENTIFICARE GLI INPUT E OUTPUT;
 - 2) ABBRACCARE UNO STATE TRANSITION DIAGRAM;
 - 3) SCRIVERE LA STATE TRANSITION TABLE;
 - 4) SELEZIONARE UN ENCODING DEGLI STATI;
 - 5) MACCHINA DI MOORE/MEALY:
 - Rischierare la STATE TRANSITION TABLE con l'encoding degli stati,
 - Scrivere l'OUTPUT TABLE;
 - 6) SCRIVERE LE EQUAZIONI BOOLEANE RELATIVE ALLA LOGICA DI PROSSIMO STATO E ALLA LOGICA DI OUTPUT;
 - 7) MINIMIZZARE LE EQUAZIONI;
 - 8) FARE UNO SCHEMA DEL CIRCUITO;
- (Esercizi ed esempi: lezione 13 pag. 18)

FATTORIZZAZIONE DI FSM:

Fattorizzazione, consiste nel suddividere una FSM complessa in FSM più piccole che interagiscono tra di loro;

PARALLELISMO: Esistono 2 tipi di parallelismo:

SPATIALE **TEMPORALE**

- Duplicare l'hardware per eseguire più task;
- le task è suddiviso in più fasi;
- le diverse fasi sono eseguite in pipelining contemporaneamente;

LATENZA E THROUHPUT:

- **TOKEN**: Gruppo di input da processare per ottenere un output significativo;
- **LATENCY**: Tempo che occorre ad un token per essere processato e produrre un output;
- **THROUGHPUT**: Numero di output prodotti per unità di tempo;

N.B.: le PARALLELISMO incrementa il THROUGHPUT;

Esempio di Parallelismo:

Bon vede fare delle torte; Latency: $(5 + 15) \text{ min} = 20 \text{ min} = \frac{1}{3} \text{ h}$;

5 minuti per preparare una torta; Throughput: $\frac{1}{\text{latency}} = \frac{1}{\frac{1}{3}} = 1 \cdot 3 = 3 \text{ torte}$;

15 minuti per cucinarla;

parallelismo spaziale: Bon chiude ed ERSA di aiutando, usando anche le sue forze;

Latency: $\frac{4}{3} \text{ h}$ Throughput: $2 \cdot \frac{1}{\frac{4}{3}} = 2 \cdot 1 \cdot \frac{3}{4} = 6 \text{ torte}$;

parallelismo temporale: 2 fasi: preparare e cucire; mentre una tonta è in forno, Ben me prepara un'altra;

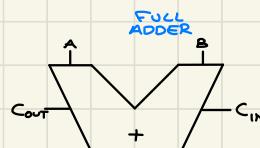
latency: $\frac{1}{3} h$ Throughput: $\frac{4}{65} \cdot 60 \cong 3,7$ tonte/h

CIRCUITI ARITMETICI E MEMORIE:

1-BIT ADDERS:



		Cout S	
A	B	S	
0	0	0	$S = A \oplus B$
0	1	0	$Cout = A \cdot B$
1	0	0	
1	1	1	



$$S = A \oplus B \oplus C$$

$$C_{out} = AB + A C_{in} + B C_{in}$$

C _{in}	A	B	Cout S
0	0	0	0 0
0	0	1	0 1
0	1	0	0 1
0	1	1	1 0
1	0	0	0 1
1	0	1	1 0
1	1	0	1 0
1	1	1	1 1

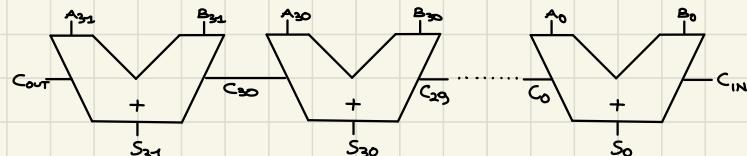
MULTIBIT ADDERS (CPAs)

Tipologie CARRY PROPAGATE ADDERS.

- Ripple-Carry (lento)
- Carry-Lookhead (veloce)

RIPPLE-CARRY ADDER:

- 1-bit adder concatenati insieme
- Il ritardo si propaga lungo la catena
- Svantaggio: LENTO



Il RITARDO di un Ripple-Carry è $T_{Ripple} = N \cdot t_{FA}$, dove N è il numero di 1-bit full adders e t_{FA} è il ritardo di un singolo 1-bit full adder.

CARRY-LOOKAHEAD ADDER:

Il bit i -esimo produce un ritardo (carry out) per generazione o per propagazione di un ritardo del bit $(i-1)$ -esimo (carry in).

- GENERA: Il bit i -esimo genera un carry out se $A_i \cdot B_i$ sono entrambi 1. $G_i = A_i \cdot B_i$
- PROPAGA: Il bit i -esimo propaga un carry or carry out se $A_i \oplus B_i$ sono 1. $P_i = A_i \oplus B_i$
- CARRY OUT: Il carry i (C_i) è dato da: $C_i = A_i \cdot B_i + (A_i \oplus B_i) C_{i-1} = G_i + P_i C_{i-1}$

BLOCK Propagate AND Generate

Ora dobbiamo estendere le funzioni PROPAGATE e GENERATE a blocchi di K-bits, ovvero:

- calcolare se un blocco di K-bit ($i, i+1, \dots, i+k-1$) PROPAGA il carry out del blocco precedente;
- calcolare se un blocco di K-bit ($i, i+1, \dots, i+k$) GENERA un carry out;

In generale:

$$P_{i:j} = P_i \cdot P_{i-1} \cdot P_{i-2} \cdots P_j$$

$$G_{i:j} = G_i + P_i \cdot (G_{i-1} + P_{i-1} \cdots G_j) \cdots)$$

$$C_i = G_{i:j} + P_{i:j} \cdot C_{j-1}$$

RITARDO DEL CARRY - LOOKAHEAD ADDER:

Per un N-bit CARRY-LOOKAHEAD ADDER con blocchi di K-bit:

$$t_{CLA} = t_{PG} + t_{PG-block} + \left(\frac{N}{K}-1\right) \cdot t_{AND/OR} + K \cdot t_{FA} \text{ dove:}$$

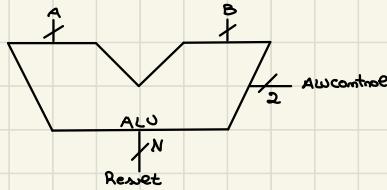
- t_{PG} è il ritardo per generare P_i, G_i ;
- $t_{PG-block}$ è il ritardo per generare $P_{i:j}, G_{i:j}$;
- $t_{AND/OR}$ è il ritardo delle porte AND/OR, che si propaga da C_{IN} a C_{OUT} lungo $\frac{N}{K}-1$ blocchi;

ALU: ARITHMETIC LOGIC UNIT

Le operazioni che tipicamente l'ALU esegue sono:

- ADDIZIONE
- SOTTRAZIONE
- AND
- OR

ALU Control 1:0		Function
00		ADD
01		SUBTRACT
10		AND
11		OR



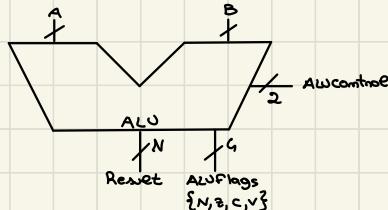
Esempio:

$$A + B \rightarrow \text{ALUcontrol} = 00$$

$$\text{Result} = A + B$$

FLAG	DESCRIZIONE
N	Risultato è Negativo
Z	Risultato è zero
C	L'Adder produce Carry Out
V	Adder overflowed

ALU con Flags di stato



SHIFTERS:

Si dividono in:

- **LOCAL SHIFTER**: trasla i bit a sinistra o a destra e riempie gli spazi vuoti con degli 0;

Esempio:

- ~~1101 >> 2 = 00110~~
- ~~21001 << 2 = 00100~~

- **ARITHMETIC SHIFTER**: si comporta come le logiche shifter a sinistra, invece nella traduzione a destra mantiene gli spazi vuoti con il bit più significativo; (utile per re segno)

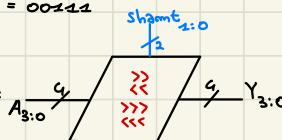
Esempio:

- ~~1100 >> 2 = 11110~~ \Rightarrow Equivale a: $A \ggg N = A / 2^N$
- ~~21001 << 2 = 00100~~ \Rightarrow Equivale a: $A \lll N = A \times 2^N$

- **ROTATOR**: ruota i bit in centro, i bit che escono da un lato mantengono dorsi' altro;

Esempio:

- ~~11001 RDR 2 = 01110~~
- ~~11001 ROL 2 = 00111~~

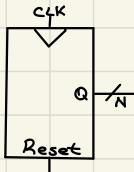


COUNTERS.

Incrementano ad ogni clock, vengono usati per eseguire i cicli sui numeri, infatti vengono spesso usati

o come analogi digitali o come program counter.

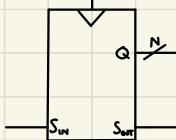
Il simbolo è.



SHIFT REGISTERS:

Trasferire di un bit ad ogni clock, ritornando in uscita un Sout. Solitamente convertono un **input seriale** (S_{in}) in un **output parallelo** $Q_{0:N-1}$.

Il simbolo è:



SHIFT REGISTER CON LOAD PARALLELO:

Quando **Load = 1**, funziona come un normale registro a N-bit, invece quando **Load = 0**, agisce da shift register, quindi può operare sia da **convertitore SERIALE-PARALLELO** sia da **convertitore PARALLELO-SERIALE**

MEMORY ARRAYS:

Vengono utilizzati per memorizzare efficacemente una grossa quantità di dati.

Vi sono 3 tipologie:

- **DYNAMIC RANDOM ACCESS MEMORY (DRAM)**
- **STATIC RANDOM ACCESS MEMORY (SRAM)**
- **READ ONLY MEMORY (ROM)**

Ogni dato di N-bit viene letto/scritto su un unico indirizzo a N-bit;

I **memory arrays**, sono array bidimensionali di celle di memoria, in cui ogni cella memorizza un bit;

Così N bit di indirizzo e M bit dato:

- 2^N sono le righe;
- M sono le colonne;
- **LUNGHEZZA o PROFONDITÀ** = Numero di righe (pande);
- **LARGHEZZA** = Numero di colonne (lunghezza di una parola);
- **DIMENSIONE ARRAY** = LUNGHEZZA \times LARGHEZZA = $2^N \times M$;

WORDLINE:

- Agisce come un **ENABLE**;
- Seleziona una riga nella memoria;
- Corrisponde ad un unico indirizzo;
- Solo una wordline per volta è attivata;

TIPI DI MEMORIA

- * **RANDOM ACCESS MEMORY (RAM)**: **VOLATILE**;
- * **READ ONLY MEMORY (ROM)**: **NON VOLATILE**;

RAM:

La **RAM** è la memoria primaria di un computer, quest'ultima svolge le operazioni di lettura e scrittura in modo molto più veloce rispetto alla **ROM**, però è una memoria **volatile**, ovvero i dati vengono persi quando il computer si spegne;

Questa memoria viene detta **Random Access Memory** perché se tempo di accesso ad una parola è lo stesso per ogni indirizzo;

ROM:

Questa è una memoria **non volatile**, ovvero mantiene i dati anche quando il computer è spento.

L'operazione di lettura è molto veloce, ma l'operazione di scrittura è lenta oppure non è possibile scrivere.

ROM sta per **read only memory** perché inizialmente questo tipo di memoria veniva "scritta" baciando dei fusibili;

Quindi una volta configurata non è più manipolabile;

Esempi di ROM: Flash memory, Bios, ecc....

TIPI DI RAM:

* **DRAM** → DYNAMIC RANDOM ACCESS MEMORY

* **SRAM** → STATIC RANDOM ACCESS MEMORY

Si differenziano per le componenti usate per memorizzare i dati, infatti:

- La **DRAM** usa delle capacità;
- La **SRAM** usa invertitori;

DRAM:

I bit data sono immagazzinati in delle capacità;

Questa memoria è **dinamica**, perché il valore di un bit deve essere riscritto periodicamente;

DRAM vs SRAM:

- La DRAM è più lenta;
- Scrivere e refresh inducono ad un maggiore consumo di energia.
- DRAM sono più economiche;

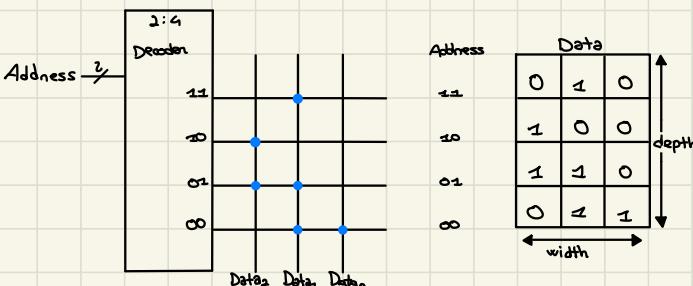
DDR SDRAM:

DDR SDRAM sta per: DOUBLE DATA RATE SYNCHRONOUS DYNAMIC RANDOM ACCESS MEMORY

In questa memoria, le operazioni di lettura e scrittura sono sincronizzate da un clock;

Le trasmissioni dei dati avvengono sia nel **Rising-edge** del clock ($0 \rightarrow 1$) che nel **Falling-edge** ($1 \rightarrow 0$)

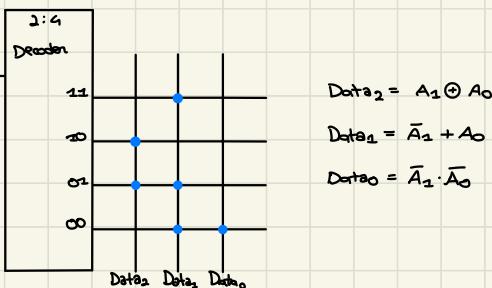
ROM STORAGE



ROM con porte logiche



ROM logic



$$\text{Data}_2 = A_1 \oplus A_0$$

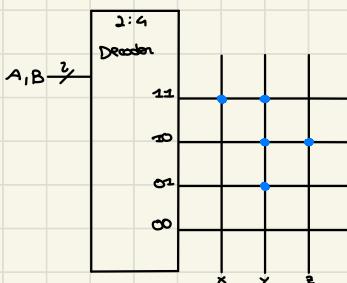
$$\text{Data}_1 = \bar{A}_1 + A_0$$

$$\text{Data}_0 = \bar{A}_1 \cdot \bar{A}_0$$

Esempio:

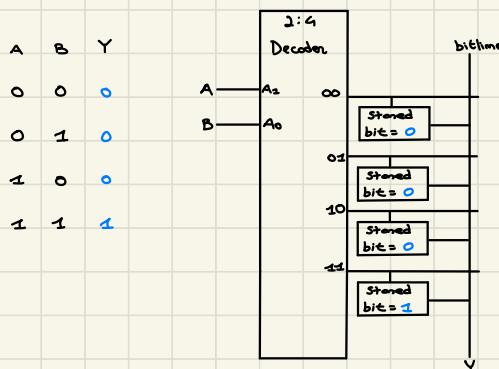
Usare un $2^2 \times 3$ -bit ROM per implementazione funzioni booleane:

- X : A · B
- Y : A + B
- Z : A · \bar{B}



LOOKUP TABLES (LUT'S)

Sono chiamate **lookup tables** perché si "memorano a guardie" gli output per ogni combinazione di input;



LOGIC ARRAYS

- PLAs (Programmable logic arrays)

AND array, registro da un OR array;

real circuiti combinatorici (SOP);

le connessioni interne sono fisse;

■ FPGAs (Field programmable gate arrays)

- Array elementi logici (LE);
 - Circuiti combinatori e sequenziali;
 - Programmable internal connections;

ARCHITETTURA ARH

Architettura: descrive le operazioni del computer:

- Come una programmazione vede a basso livello un computer;
 - Definisce un insieme di istruzioni e di registri che fungono da operandi per tali istruzioni;

Istruzioni ARM

- **ASSEMBLY LANGUAGE:** Formato HUMAN-READABLE;
 - **MACHINE LANGUAGE:** Formato COMPUTER-READABLE;

Principi di progettazione alla base

- la negoziazione favorisce la semplicità;
 - rendere veloci le cose frequenti;
 - più piccolo è, più veloce è;
 - un buon disegnatore richiede buoni compromessi.

ADDIZIONE:

Codice C	ARM Assembly Code
$a = b + c$	ADD a, b, c ADD \Rightarrow indica l'operazione da eseguire b, c \Rightarrow operandi d'ingresso; a \Rightarrow contenuto d'uscita.

SOTTRAZIONE:

Codice C	ARM Assembly Code
$a = b - c$	Sub r, b, c Sub → indica l'operazione da eseguire b, c → operandi d'ingresso; r → memoria d'uscita.

PRINCIPIO DI PROGETTAZIONE 1

La megafonità supporta la semplicità di progettazione

- Formato di istruzioni coerente;
 - Stesso numero di operandi (2 sorgenti e 1 destinazione);
 - Facilità di codifica e gestione in hardware;

SLIDE 12 sovietata prende le due righe dopo i 3 principi

PRINCIPIO DI PROGETTAZIONE 2:

Rendere veloce e poco costoso;

≡ ARM include solo istruzioni semplici e di uso comune;

≡ Gli hardware per la decodifica e l'esecuzione di istruzioni sono mantenuti semplici, piccole e veloci;

≡ Le istruzioni più complesse sono eseguite utilizzando istruzioni più semplici;

≡ ARM è un Reduced Instruction Set Computer, con un piccolo numero di semplici istruzioni;
RISC

≡ Altre architetture, come Intel x86 sono Complex Instruction Set Computer;
CISC

PRINCIPIO DI PROGETTAZIONE 3:

Più piccolo è, più veloce è;

≡ ARM include solo un numero limitato di registri;

Istruzioni multiple

Codice C ARM Assembly Code

$a = b + c - d;$ ADD t, b, c ; t = b+c;
 SUB a, t, d ; a = t - d;

Le posizioni riserve in un computer sono:

- I REGISTRI;
- LE COSTANTI anche chiamate IMMEDIATES;
- LA MEMORIA;

I REGISTRI:

ARM ha 16 registri (R0...R15) a 32 bit che sono equivalenti tra di loro, però dal punto di vista logico vengono usati per scopi specifici:

Nome	Utilizzo
R0	Argument / return value / temporary variable
R1 - R3	Argument / temporary variables
R4 - R12	Saved variables
R12	Temporary variable
R13 (SP)	Stack Pointer
R14 (LR)	Link Register
R15 (PC)	Program Counter

ADDITIONE con i registri:

Codice C ARM Assembly Code

$$a = b + c \quad R0 = a, R1 = b, R2 = c;$$

```
ADD R0, R1, R2;
```

LE COSTANTI / IMMEDIATES:

Alcune istruzioni come ADD e SUB, possono usare costanti/immediates; le valori sono immediatamente disponibili dalle istruzioni;

Esempio:

Codice C ARM Assembly Code

$$R0 = a, R1 = b;$$

$$a = a + 4; \quad ADD R0, R0, #4;$$

$$b = a - 12; \quad SUB R1, R0, #12;$$

Generazione di Costanti:

È possibile definire le costanti con l'istruzione MOV:

Codice C ARM Assembly Code

$$R0 = a, R1 = b;$$

$$\text{int } a = 23; \quad \text{MOV R0, } \#23$$

$$\text{int } b = 0x4B; \quad \text{MOV R1, } \#0x4B$$

Le costanti così generate hanno una precisione di massimo di 8 bits;

N.B.:

MOV può anche essere usato per spostare il contenuto di un registro in un altro registro:

`MOV R7, R9;`

LA MEMORIA:

Trasporto dati per stoccare solo in 16 registri;

Mermorizzazione più dati in memoria;

La memoria è grande però è lenta;

Le variabili di uso comune restano comunque nei registri;

La LETTURA della MEMORIA prende se nome di LOAD;

COMANDO: Load register (LDR);

FORMATO: LDR R0, [R1, #12]

INDIRIZZO: R1 + 12

In R0 vi entrerà il contenuto presente nell'indirizzo (R1 + 12)

Esempio (lezione 17 pag. 31):

- INDIRIZZO: $(R2 + 8) = 8$

- R3 = 0x02EE28C7

ARM Assembly Code:

MOV R2, #0

LDR R3, [R2, #8]

La SCRITTURA DELL'MEMORIA prende il nome di STORES

COMANDO: Store register (STR)

FORMATO: STR R7, [RS, #0x54]

Esempio (lezione 17 pag. 30)

- INDIRIZZO: $4 \times 2 - 1 = 8 - 1 = 8 - 1 = 0x54$

ARM Assembly Code:

MOV R5, #0

R7, [RS, #0x54]

Come si posso memorizzare i byte dell'interno della parola??

LITTLE-ENDIAN

I numeri dei byte

iniziano alla fine

(meno significativa);

BIG-ENDIAN

I numeri dei byte

iniziano dall'estremità

più grande (più

significativa);

BLOCCHI PREDEFINITI DI PROGRAMMAZIONE:

* ISTRUZIONI PER L'ELABORAZIONE DATI;

* ESECUZIONE CONDIZIONALE;

* BRANCHES;

* COSTRUTTI DI ALTO LIVELLO;

* ISTRUZIONI PER L'ELABORAZIONE DATI:

= OPERAZIONI LOGICHE;

= SHIFTS / ROTATE;

= MOLTIPLICAZIONE;

OPERAZIONI LOGICHE:

- AND
- ORR (OR) → Operano tutte bit a bit
- EOR (XOR) → Prima sorgente un registro, seconda registro/immediato
- BIC (Bit Clean)
- MVN (Move and Not)

• AND o BIC → Sono utile per mascherare i bit;

• ORR → È utile per combinare i campi di bit;

• MOV e ORR → Generare costanti.

↓
Codice C

ARM Assembly Code

```
int a = 0x7EDC8765;      MOV R0, #0x7E000000
                          ORR R0, R0, #0xDC0000
                          ORR R0, R0, #0x8700
                          ORR R0, R0, #0x65
```

SHIFT/ROTATE

- **LSL**: logical shift left → LSL R0, R2, #5 ; R0 = R2 << 5;
- **LSR**: logical shift right → LSL R3, R2, #31; R3 = R2 >> 31;
- **ASR**: arithmetic shift right → ASR R9, R11, R4; R9 = R11 >> R4;
- **RRR**: rotate right → ROR R8, R2, #3 ; R8 = R2 ROR 3;

MOLTIPLICAZIONE

- **MUL**: 32×32 multipl. percorso, 32-bit → MUL R1, R2, R3; R1 = $(R2 \times R3)_{32:0}$
- **UMULL**: 32×32 multipl. unsigned long, 64-bit → UMULL R1, R2, R3, R4 {R1, R2} = R2 \times R3 (R2, R3 unsigned)
- **SMULL**: 32×32 multipl. signed long, 64-bit → SMULL R1, R2, R3, R4 {R1, R2} = R3 \times R4 (R3, R4 signed)

ESECUZIONE CONDIZIONALE

Non sempre si desidera eseguire il codice in sequenza

Ad esempio istruzioni come **if/else**, **cicli**, **while**, ecc... eseguono il codice solo se la condizione è VERA;

Oppure vi è le **branching** che mette ad un'altra parte del codice se la condizione è VERA;

ARM include delle **flag di condizione** che possono essere:

- impostate da un'istruzione;
- utilizzate per eseguire condizionalmente un'istruzione;

FLAG:

FLAG	NAME	DESCRIZIONE
N	Negative	Il risultato dell'istruzione è NEGATIVO
Z	Zero	Il risultato dell'istruzione è ZERO
C	Carry	L'istruzione causa un CARRY OUT
V	Overflow	L'istruzione causa un OVERFLOW

Impostazione dei flag:

METODO 1: Confronta le istruzioni (CMP);

Esempio: CMP R5, R6 → Non salva il risultato;

↓

Esegue
R5-R6

↓

Imposta i flag. Se il risultato: - è 0 → Z=1; - è Negativo → N=1;
- Causa un carry out → C=1; - Causa un overflow → V=1;

METODO 2: Aggiungi l'istruzione mnemonica con lo S;

Esempio: ADDS R1, R2, R3 → Salva il risultato in R1;

↓
Esegue
R2+R3

↓
Imposta i flag. Se il risultato: - è 0 → Z=1; - è Negativo → N=1;
- Causa un carry out → C=1; - Causa un overflow → V=1;

CONDIZIONI MNEMONICHE

Le istruzioni possono essere eseguite condizionalmente in base ai flag.

Le condizioni di esecuzione sono codificate come **condizioni mnemoniche** aggiunte alle istruzioni;

Esempio:

CMP R1, R2

SUBNE R3, R5, R6 → Verrà eseguita solo se R1 ≠ R2

Condizioni mnemoniche

CONDIZIONI MNEMONICHE

cond	Mnemonic	Name	CondEx
0000	EQ	Equal	Z
0001	NE	Not equal	\bar{Z}
0010	CS / HS	Carry set / Unsigned higher or same	C
0011	CC / LO	Carry clear / Unsigned lower	\bar{C}
0100	MI	Minus / Negative	N
0101	PL	Plus / Positive of zero	\bar{N}
0110	VS	Overflow / Overflow set	V
0111	VC	No overflow / Overflow clear	\bar{V}
1000	HI	Unsigned higher	$\bar{Z}C$
1001	LS	Unsigned lower or same	$Z \text{ OR } \bar{C}$
1010	GE	Signed greater than or equal	$\bar{N} \oplus V$
1011	LT	Signed less than	$N \oplus V$
1100	GT	Signed greater than	$\bar{Z}(\bar{N} \oplus V)$
1101	LE	Signed less than or equal	$Z \text{ OR } (N \oplus V)$
1110	AL (or none)	Always / unconditional	ignored

★ BRANCHES

I branch consentono l'esecuzione di istruzioni fuori dalla sequenza;

Abbiamo 2 tipi di branch:

- **BRANCH (B)** → Passa ad un'altra istruzione; → Esempio di Branch INCONDIZIONALE: `MOV R2, #17` ; $R2 = 17$
 - **BRANCH AND LINK (BL)** → Ordine a passare ad un'altra istruzione quest'ultimo copia l'indirizzo dell'istruzione successiva in **LR**, **LINK REGISTER**; **Registri dei collegamenti**
- Entrambi possono essere condizionati o incondizionati;

★ COSTRUTTI DI ACRO LINELLO

+ ISTRUZIONE IF :

Codice C

ARM Assembly Code

```

;R0 = f, R1 = g, R2 = h, R3 = i, R4 = j
if (i == 5)
    CMP R3, R4
    f = g + h;
    BNE L1
    ADD R0, R1, R2
L1:
    f = f - i;
    SUB R0, R0, R3

```

N.B.: Il codice assembly effettua il test opposto ($i \neq 5$) rispetto a quello di alto livello ($i == 5$)

+ ISTRUZIONE IF/ELSE:

Codice C

ARM Assembly Code

```

;R0 = f, R1 = g, R2 = h, R3 = i, R4 = j
if (i == 5)
    CMP R3, R4
    f = g + h;
    BNE L1
    ADD R0, R1, R2
    B L2
else
    L1: SUB R0, R0, R3
    f = f - i;
    L2: ...

```

+ CICLO WHILE:

CODICE C :

```
int pow = 1, x = 0;          ; R0 = pow, R1 = x
while(pow != 128){           MOV R0, #1
    MOV R1, #0
    pow = pow * 2;          WHILE
    x = x + 1;              CMP R0, #128
}
                                BEQ DONE
    LSL R0, R0, #1
    ADD R1, R1, #1
    B WHILE
}
                                DONE
```

+ CICLO FOR:

for (inizializzazione; condizione; operazione di loop)

istruzioni;

- **INIZIALIZZAZIONE** → Eseguita prima che inizi il loop;
- **CONDIZIONE** → condizione di continuazione che è verificata all'inizio di ogni iterazione;
- **OPERAZIONE DI LOOP** → eseguita alla fine di ogni iterazione;
- **ISTRUZIONI** → eseguite ad ogni iterazione, finché la condizione è verificata;

CODICE C :

ARM ASSEMBLY CODE:

```
int sum = 0          ; R0 = i, R1 = sum
for(i = 1; i != 10; i = i + 1)  MOV R0, #1
                                MOV R1, #0
    sum = sum + i;          ADD R1, R1, R0
}
                                ADD R0, R0, #1
                                B FOR → Se arriva qui, ripete il loop;
}
                                DONE
```

+ ARRAYS:

Ci permettono di accedere a grandi quantità di dati simili;

Sono principalmente composti :

- deve **INDICE** che ci permette di accedere ad ogni elemento;
- deve **DIMENSIONE** che indica il numero di elementi;
- deve **INDIRIZZO BASE**;

CODICE C :

ARM ASSEMBLY CODE :

```
int array[5];          R0 = array base address;           *2: LDR R1, [R0, #4] → R1 = array[1]
array[0] = array[0]*8; MOV R0, #0x60000000 → R0 = 0x60000000    LSL R1, R1, 3 → R1 = R1 << 3 = R1*8
array[1] = array[1]*8; LDR R1, [R0] → R1 = array[0]           STR R1, [R0, #4] → array[1] = R1
                                                               2SL R1, R1, 3 → R1 = R1 << 3 = R1*8
                                                               STR R1, [R0] → array[0] = R1
                                                               *2
```

Arrary come in loop:

CODICE C :

ARM ASSEMBLY CODE :

```
int array[200], i;      , R0 = array base address, R1 = i   *2: FDR
for (i = 0, i >= 0, i = i+1) MOV R0, #0x60000000
array[i] = array[i]*8;  MOV R1, #199
                                                               *2
                                                               LDR R2, [R0, R1, LSL #2] → R2 = array(i)
                                                               LSL R2, R2, #3 → R2 = R2 << 3 = R2*8
                                                               STR R2, [R0, R1, LSL #2] → array(i) = R2
                                                               SUBS R2, R2, #1 → i = i - 1 e imposta i flag
                                                               BPL FOR → se (i >= 0) ripeti le code
```

INDICIZZAZIONE

ARM fornisce 3 tipi di indicizzazione:

■ **OFFSET** : l'indirizzo è calcolato sommando o sottraendo un offset al contenuto del registro di base;

Esempio:

LDR R1, [R2, #4] ; R1 = mem[R2 + 4]

■ **PRE-INDEX** : l'indirizzo viene calcolato come nel caso dell'offset e viene scritto nel registro di base;

Esempio:

LDR R3, [R5, #16]! ; R3 = mem[R5 + 16] , R5 = R5 + 16

■ **POST-INDEX** : l'indirizzo corrisponde al contenuto del registro di base;

l'offset viene aggiunto o sottratto al contenuto del registro di base e riscritto nel registro di base;

Esempio:

LDR R8, [R4], #8 , R8 = mem[R4] , R4 = R4 + 8

+ CHIAMATE DI FUNZIONI:

Sono composte:

- dae **CALLER**: cioè la funzione chiamante, che passa gli argomenti al **callee** ed esegue un salto su quest'ultimo;
- dae **CALEE**: ovvero la funzione chiamata, che ritorna il risultato del **caller** e ritorna nel punto di chiamata del **caller**.

CODICE C ARM Assembly Code

```
int main() {
    simple();
    a = b + c;
}

void simple() {
    SIMPLE:
    MOV PC, LR
    return;
}
```

ISTRUZIONI ARM per chiamata di funzione:

- **CHIAMATA A FUNZIONE**: Branch and Link (BL)
- **RETURN da funzione**: ripristina un PC ie valore del Link register (MOV PC, LR)
- **ARGOMENTI**: R0 - R3
- **VALORE DI RITORNO**: R0

LO STACK DELLE FUNZIONI

Lo stack costituisce la memoria usata per contenere temporaneamente le variazioni delle variabili.

È organizzato come una **PLA Last-In-First-Out (LIFO)** ed è composto da uno **STACK POINTER (SP)** che punta al top dello stack;

Sopra stack è possibile effettuare 2 operazioni principali:

- **PUSH** → che consente di salvare in memoria più registri e aggiorna consistentemente lo stack → Esempio: PUSH {R4, R8, R9}
- **POP** → che ripristina uno o più registri e incrementa consistentemente i valori dello stack → Esempio: POP {R4, R8, R9}

Esempi:

```
MOV R0, #0
MOV R1, #0
MOV R3, #0
CMP R0, #5 loop
BGE exit
MOV R3, R0
CMP R3, #0 ine
BLE next
ADD R1, R1, R0
SUB R3, R3, #1
B ine
ADD R0, R0, #1 next
B loop
exit
```

```
int i = 0; sum = 0;
int m = 0, R3 = m;
for( i = 0, i < 5, i++ ) {
    m = i;
    while( m > 0 ) {
        sum += i;
        m--;
    }
}
```

FATTORIALE

```
int fattoriale(int m) {
    if (n <= 1)
        return 1;
    else
        return (m * fattoriale(n-1));
}
```

, R1 = m

{ supponiamo che R0 contenga

fattoriale ie valore di input a R0

PUSH {R1, LR} contiene anche i.e.

MOV R1, R0; valore restituito }

CMP R0, #1

BGT fattoriale-next

MOV R0, #1

B fattoriale-next

fattoriale-next

SUB R0, R0, #1

BL fattoriale

MUL R0, R0, R1

fattoriale-next

POP {R1, PC}

■ REGOLA DI SALVATAGGIO DEL CHIAMANTE:

Prima di una chiamata a funzione, il chiamante deve salvare ogni registro non preservato che intende usare dopo la chiamata e successivamente deve ripristinarlo prima di poterlo usare nuovamente;

■ REGOLA DI SALVATAGGIO DEL CHIAMATO:

prima di modificare un registro preservato, il chiamato deve riservare il contenuto e ripristinare prima di tornare al chiamante;

CHIAMATA DI UNA FUNZIONE RICORSIVA

Abbiamo una funzione main legata che:

- chiama se stessa;
- Si comporta da CHIAMATO a CHIAMANTE;
- Salva sia registri preservati che NON;

In sintesi:

IL CALLER:

- ★ Mette gli argomenti in R0 - R3;
- ★ Salva tutti i registri necessari;
- ★ Funzioni di chiamata : **BL CALLEE**
- ★ Ripristina i registri
- ★ Cerca se riservato in R0;

IL CALLEE:

- ★ Salva i registri che potrebbero essere disturbati (R4-R7);
- ★ Esegue la funzione;
- ★ Mette a riservato un R0;
- ★ Ripristina i registri;
- ★ Restituisce: **MOV PC, LR**

Esercizio:

```
int f( int m, int k){  
    int b;  
    b = k+2;  
    if(n==0) b = 10;  
    else b = b + (n*m) + f(m-1, k+1);  
    return b*k;  
}
```

; R4 = b, R1 = k; R0 = m **ELSE:**

2 PUSH {R4, LR}
ADD R4, R1, #2
CMP R0, #0
BNE ELSE
MOV R4, #10
B DONE

DONE:

PUSH {R0, R1} MUL R0, R4, R1
SUB R0, R0, #1 POP {R4, LR}
ADD R1, R1, #2 MOV PC, LR
BL f
MOV R2, R0
POP {R0, R1}
MUL R3, R0, R0
ADD R2, R2, R3
ADD R4, R2, R4

PRINCIPIO DI PROGETTAZIONE 4.

Un buon progetto richiede buoni compromessi;

- ⇒ Formati di istruzioni multipli consentano flessibilità;
- ⇒ Il numero di formati di istruzioni resta piccolo per ragione anche i principi di progettazione 1 e 3, ovvero la negoziazione supporta la semplicità di progettazione e inoltre più piccolo è più veloce è;

LINGUAGGIO MACCHINA



FORMATO DELLE ISTRUZIONI PER L'ELABORAZIONE DEI DATI

★ OPERANDI :

- Rn → Registro delle prime fonti;
- $Snc2$ → Seconda fonte - può essere un registro o un immediato;
- Rd → Registro di Destinazione;

★ CAMPI DI CONTROLLO :

- cmd → Specifica Condizione di Esecuzione;
- op → Il codice dell'operazione;
- $funct$ → La funzione/operazione da eseguire;

CAMPI DI CONTROLLO DELL'ELABORAZIONE DATI :

- $op = 00_2$ per le istruzioni di elaborazione dati;

- $funct$ è composto da:

- ↳ cmd , che specifica l'istruzione corretta per l'elaborazione dei dati. Ad esempio : $cmd = 0100_2$ è usato per ADD e $cmd = 0010_2$ è usato per SUB;
- ↳ I -bit può essere → $I = 1$ → $Snc2$ è un IMMEDIATO;
- ↳ $I = 0$ → $Snc2$ è un REGISTRO;

S-bit: è 1 se imposta i flag di condizionale;

Ma può essere → **S=1** → SUB R0, R5, R7

→ **S=0** → ADDS R8, R2, R4 oppure CMP R3, #10

VARIAZIONI DI SRC2:

Src2 può essere:

→ Un IMMEDIATO



- L'immmediato è codificato come:

- imm8 → Immmediato di 8 bit non segno;

- not → Valore di notazione a 4 bit

- La costante a 32-bit è:

imm8 ROR (not × 2)

N.B.:

ROR by x = ROL by (32 - x)

Ex. ROR by 30 = ROL by $(32 - 30)^2$

Esempio:

ADD R0, R1, #42

Equivalenti a:

cond = 1110₂ Src2 = è un immediato quindi I = 1

op = 00₂ Rd = 0, Rm = 1

cmd = 0100₂ imm8 = 42, not = 0
(A00)

1	1	1	0	0	0	1	0	1	0	0	0	4	2
cond	op	I	cmd	S	Rm	Rd	not	sh	Rm				

1	1	1	0	0	0	1	0	1	0	0	0001	0000	0000	0	0	1	0	1	0	1	0
---	---	---	---	---	---	---	---	---	---	---	------	------	------	---	---	---	---	---	---	---	---

In esadecimale è uguale a: 0xE281002A



- Rm → Il secondo operando sorgente;

- Shamt5 → La quantità di RH che viene shiftata

- Sh → Il tipo di shift (ex. >>, <<, >>>, ROR)

Esempio:

ADD RS, R6, R7 Equivalenti a:

cmd = 1110₂ Rd = 5

op = 00₂ Rm = 6

cmd = 0100₂ Rm = 7

Src2 = è un shamt = 0

1	1	1	0	00	0	0	1	0	0	0	0	6	5	0	0	0	7
cond	op	I	cmd	S	Rm	Rd	shamt5	sh	O	Rm							

1	1	1	0	00	0	0	1	0	0	0	0110	0101	00000	00	0	0111
E	0			8			6	5	0	0						

Registro quindi I = 0 Sh = 0 In esadecimale è uguale a: 0xE0865007



Esempio:

EOR R8, R9, R10, ROR R12

operation = R8 = R9 XOR (R10 ROR R12)

cond = 1110₂ Rd = 8

op = 00₂ Rm = 9

cmd = 000L₂ Rmn = 10

Src2 = imm Rs = 12 Im esadecimale è uguale a: 0xE0298C7A

registro quindi: I = 0; sh = 11₂ (ROR)

CODIFICA DEI VARI SHIFT :

Shift Type Sh

- LSL → 00₂
- LSR → 01₂
- ASR → 10₂
- ROR → 11₂

FORMATO DELLE ISTRUZIONI DI MEMORIA

CODIFICA: LDR, STR, LDRB, STRB

- op = 01₂ per istruzioni suella memoria
- Rn = base register
- Rd = destination(load), source(store)
- Src2 = offset : register or immediate
- funct = 6 bit di controllo

OPZIONI DI OFFSET

ADDRESS = BASE ADDRESS + OFFSET

Esempio:

LDR R1, [R2, #4]

Base Address = R2, offset = 4

Address = (R2 + 4)

- BASE ADDRESS è sempre un

un registro;

- L'OFFSET può essere:

- un IMMEDIATO;

- un REGISTRO;

- un REGISTRO SHIFTATO;

MODALITÀ DI INDICIZZAZIONE

MODE	ADDRESS	BASE REG. UPDATE
Offset	Registro di Base + Offset	Non cambia
Preindex	Registro di Base + Offset	Registro di base + Offset
Postindex	Registro di Base	Registro di base + Offset

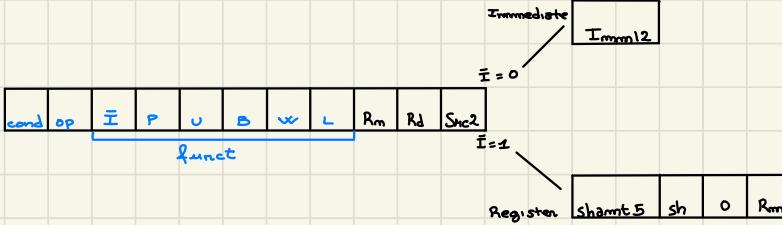
Esempi:

- ✗ **OFFSET**: LDR R1, [R2, #4] ; R1 = mem[R2+4]
- ✗ **PREINDEX**: LDR R3, [R5, #-26]. ; R3 = mem[R5 + 16] R5 = R5 + 16
- ✗ **POSTINDEX**: LDR R8, [R4], #8 ; R8 = mem[R4] R4 = R4 + 8

FORMATO DELLE ISTRUZIONI DI MEMORIA:

- Funct → è composta da:

- **I** → Immediate bit
- **P** → PostIndex
- **U** → Add
- **B** → Byte
- **W** → Writeback
- **L** → Load



- Tipi di operazione:

L	B	Istruzione
0	0	STR
0	1	STRB
1	0	LDR
1	1	LDRB

- Modalità di Indicizzazione:

P	W	Modalità di Indicizzazione
0	0	PostIndex
0	1	Non Supportato
1	0	Offset
1	1	PreIndex

Esempio con un immediato:

STR R11, [R5], #-26

operazione = mem[R5] <= R11 ; R5 = R5 - 26, funct: Rd = 11

cond = 1110₂; I = 0, P = 0, U = 0, Rm = 5

op = 0112; B = 0, W = 0, L = 0; Imm12 = 26

1	1	1	0 ₂	0	1 ₂	0	0	0	0	0 ₂	5	1 ₂	26
---	---	---	----------------	---	----------------	---	---	---	---	----------------	---	----------------	----

1	1	1	0	0	1	0	0	0	0	0	0101	1011	0000 0001 1010
---	---	---	---	---	---	---	---	---	---	---	------	------	----------------

In esadecimale equivale a: **0xE405B01A**

Esempio con un registro:

LDR R3, [R4, R5]

operation = R3 <= mem[R4+R5]; funct.

Rd = 3; shamt5 = 0;

cond = 1110₂;

I = 1, P = 1, U = 1, Rm = 4; Sh = 0;

op = 01₂;

B = 0, W = 0, L = 1; Rm = 5;

1	1	1	0 ₂	0	1 ₂	1	1	1	0	0	1	4	3	0	0	0	5
---	---	---	----------------	---	----------------	---	---	---	---	---	---	---	---	---	---	---	---

1	1	1	0	0	1	1	1	0	0	1	0100	0011	0000	00	0	0101
---	---	---	---	---	---	---	---	---	---	---	------	------	------	----	---	------

In esadecimale equivale a: **0xE79C3005**

BRANCH

ISTRUZIONI DI BRANCHING

Codifica B and BL :

Le ISTRUZIONI DI BRANCHING permettono di cambiare le valori del PROGRAM COUNTER. In

ARM vi sono 2 tipi di branch: SIMPLE BRANCH (B) & BRANCH AND LINK (BL).

Come anche altre istruzioni ARM, i BRANCH possono essere CONDIZIONATI o INCODIZIONATI. Le istruzioni di branching utilizzano un unico operando costante di:

- op = 10₂
- imm24 = immediato da 24 bit
- funct = 1L₂: L=1 per BL, L=0 per B 24 bit, imm24. Esse hanno un campo cond di 4 bit e un campo op di 2 bit, i cui valori è 10₂. Il campo funct ha solo 2 bit. Il bit più significativo è sempre 1 per B-branch. Il bit meno significativo L, indica il tipo di BRANCH:
 - 1 per BL
 - 0 per B;

cond	op	1L	imm24
funct			

Le istruzioni di branching utilizzano un unico operando costante di 24 bit, imm24. Esse hanno un campo cond di 4 bit e un campo op di 2 bit, i cui valori è 10₂. Il campo funct ha solo 2 bit. Il bit più significativo è sempre 1 per B-branch. Il bit meno significativo L, indica il tipo di BRANCH:

- 1 per BL
- 0 per B;

I 24 bit, imm24 specifica la posizione dell'istruzione relativamente all'indirizzo PC+8.

L'istruzione BL (Branch and Link) è usata per la chiamata di una subroutine,

Quest'ultima:

- Salva l'indirizzo di ritorno (R15) in R14;
- Il ritorno della routine si effettua copiando R14 in R15.

Il registro R14 detto **LINK REGISTER** ha la funzione di subroutine.

In esso viene salvato l'indirizzo di ritorno quando viene eseguita l'istruzione BL.

CODIFICA DELL'INDIRIZZO DI DESTINAZIONE DEL BRANCH (BTA):

L'INDIRIZZO DI DESTINAZIONE DEL BRANCH passa al successivo Program Counter (PC) quando il branch viene preso;

L'indirizzo di destinazione del branch è relativo all'attuale PC + 8;

L'immediato di 24 bit codifica l'indirizzo di destinazione del branch;

Imm₂₄ è uguale al numero di parole per cui BTA è distante dal PC+8;

Esempio:

0x A0 BLT THERE ← PC PC = 0xA0;
0x A6 ADD R0, R1, R2
0x A8 SUB R0, R0, R3 ← PC + 8 PC + 8 = 0xA8;
0x AC ADD SP, SP, #8
0x B0 MOV PC, LR THERE è 3 istruzioni
0x B4 THERE SUB R0, R0, #1 ← BTA distante da PC + 8
0x B8 BL TEST Quando imm₂₄ = 3;

1	0	1	s ₂	1	0 ₂	1	0 ₂	3
cond		op		funct		imm ₂₄		

1	0	1	1	1	0	1	0	00	00	00	00	00	00	00	00	00	00	11
---	---	---	---	---	---	---	---	----	----	----	----	----	----	----	----	----	----	----

In esadecimale equivale a : 0xBA00003

INTERPRETAZIONE DEL CODICE MACCHINA

- Parte con op : tells how to parse op ??

- op = 00 (Elaborazione Dati)
- op = 01 (Memoria)
- op = 10 (Branch)

- I-bit : tells how to parse s₁s₂ ??

- Istruzioni per l'elaborazione dati.

Se I-bit = 0, i.e. bit 4 determina se s₁s₂

è un registro - shiftato di un registro; → bit 4 = 1

è un registro; → bit 4 = 0

- Istruzioni di memoria.

Esaminiamo i bit di funzione per la modalità di indirizzamento, le istruzioni e l'aggiunta o sottrazione di offset

Esempio:

0xE0475001

$$op = 00_2$$

$I-bit = 0 \rightarrow$ src2 è un registro

$G-bit = 0 \rightarrow$ src2 è un registro

$$cmd = 0010_2 (\text{SUB})$$

$$Rm = 7, Rd = 5, Rm = 1$$

$$shamt5 = 0, sh = 0$$

cond	op	I	cmd	S	Rm	Rd	shamt5	sh	Rm
1 1 1	00 ₂	00 ₃	0 2	0	7	5	0	0	0 1

1	1	1	0	00	0	0010	0	0111 0101	00000 00	0	0001

Im istruzione equivale a: SUB R5, R7, R1

Esempio:

0xE5949010

cond	op	I	P	U	B	W	L	Rm	Rd	imm12
1 1 1 0	0 0	1	0	1	1	0	0	1	0100	1001 0000 0001 0000

$$cond = 1110_2$$

op = (01)₂ → istruzioni per la memoria

$$I = 0, P = 1, U = 1,$$

$$B = 0, W = 0, L = 1;$$

Funct = Visto che $L=1 \wedge B=0$ è uguale a LDR; mentre $P=1 \wedge W=0$ quando abbiamo un offset indexing.

Inoltre vediamo che, $I=0$ quando abbiamo un IMMEDIATO e $U=1$ quando l'offset si somma;

$$Rm = 4, Rd = 3, imm12 = 16$$

Quindi possiamo dire che l'istruzione di questo ESADECIMALE è: LDR R3, [R4, #16]

MODALITÀ DI INDIRIZZAMENTO

• REGISTER ADDRESSING :

- Operandi di origine e di destinazione trovati nei registri;
- Usato dalle istruzioni per l'elaborazione dei dati;
- Abbiamo 3 sottomodalità.

✗ Solo Registri → Ex. ADD R0, R2, R7

✗ Immediato shiftato di un registro → Ex. ORR R5, R1, R3 LSL #4

✗ Registro shiftato di un altro registro → Ex. SUB R2, R9, R0, ASR R1

● IMMEDIATE ADDRESSING:

- Operandi di origine e di destinazione trovati nei registri o negli immediati → Ex. ADD R5, R1, #15
- Utilizza l'elaborazione dati nel formato com $I = 1$
- L'immediato è codificato come:
 - ✗ Immediato a 8 bit (imm8)
 - ✗ Rotazione a 4 bit (rot)

- L'immediato a 32 bit = imm8 ROR (rot $\times 2$)

● BASE ADDRESSING:

- L'indirizzo dell'operando è: BASE REGISTER + OFFSET
- L'OFFSET può essere:
 - ✗ Un immediato a 12 bit (imm12) → Ex. LDR R0, [R8, # -14] ($R0 = \text{mem}[R8 - 14]$)
 - ✗ Un registro → Ex. LDR R1, [R7, R3] ($R1 = \text{mem}[R7 + R3]$)
 - ✗ Un registro shiftato di un immediato → Ex. STR R5, [R3, R2, LSL #4] ($R5 = \text{mem}[R3 + (R2 \ll 4)]$)

● PC-RELATIVE:

- Usato per i branch;
- Formato delle istruzioni dei branch:
 - ✗ Gli operandi sono il PROGRAM COUNTER (PC) e un IMMEDIATO DI 24 bit con segno (imm24);
 - ✗ Cambio del PC;
 - ✗ Il nuovo PC è relativo a quello vecchio;
 - ✗ Imm24 indica il numero di parole compresa tra PC e PC+8;

POTENZA DI UN PROGRAMMA MEMORIZZATO

- Istruzioni a dati a 32 bit: vengono memorizzati in memoria;
- Per eseguire un nuovo programma: 1) Non è richiesto un ricollegamento; 2) Memorizzata semplicemente il nuovo programma in memoria;
- Esecuzione del programma: 1) il processore recupera le istruzioni in sequenza dalla memoria; 2) il processore esegue l'operazione specificata;

MICROARCHITETTURA ARM

Una **ARCHITETTURA** → Rappresenta la struttura di un calcolatore dal punto di vista di un programmatore;

→ È definita dai set di istruzioni e operandi che costituiscono il **Lingaggio macchina**, a cui corrisponde poi un **Lingaggio ASSEMBLY**;

→ Esistono molte architetture differenti, come ad esempio: **ARM, X86**;

Una **MICROARCHITETTURA** → definisce la disposizione specifica dei registri, delle ALU, delle macchine a stati finiti, delle memorie e degli altri blocchi necessari per implementare un'architettura;

→ Può realizzare la medesima architettura di un'altra microarchitettura, ma con scelte diverse e prestazioni differenti, come ad esempio INTEL e AMD che realizzano diverse microarchitetture ma con la stessa architettura (x86);

Cos' è un PROCESSORE?

Un **PROCESSORE** è un **automa** a stati finiti, che esegue **istruzioni** presenti in memoria;

Lo **STATO** del sistema è definito sia dai valori contenuti nelle **locazioni di memoria** e sia da valori contenuti in alcuni **registri** del processore stesso;

Ogni **istruzione** che viene eseguita definisce .

- In che modo lo stato deve cambiare;
- Quale istruzione deve essere eseguita in seguito;

Una microarchitettura può essere vista come un **automa**;

VARIAZIONI DI STATO

I registri, la memoria dati e la memoria istruzioni operano mediante **logica combinatoria**.

Tutti i componenti effettuano la scrittura nel fronte alto del clock, così che lo stato del sistema cambia solo al fronte del clock;

Gli indirizzi, i dati e le segnali di unità esecutiva, devono essere impostati prima del fronte del clock;

Sia gli elementi di stato, che il microprocessore sono costituiti da logica combinatoria e da componenti **asincronizzate** dal clock, quindi l'intero sistema è **SINCRONO** e può essere visto come una complessa macchina a stati finiti;

MICROARCHITETTURE A CICLO SINGOLO

In queste microarchitetture, un'istruzione viene eseguita in un singolo ciclo;

VANTAGGI:

- Sempre da comprendere;
- Unità di controllo molto semplice;
- **NON** richiede stati man architettonici;

SVANTAGGI:

- Il tempo è pari a quello dell'istruzione più lenta;
- Memoria dati e memoria istruzioni separate;

MICROARCHITETTURE A CICLO MULTIPLO

In queste microarchitetture, un'istruzione viene eseguita in più cicli brevi;

VANTAGGI:

- Più usa le componenti;
- Durata delle istruzioni;
- Non richiede la separazione delle memorie;

SVANTAGGI:

- Richiede stati man architettonici;
- Esegue un'istruzione per volta;

MICROARCHITETTURE CON PIPELINE

In queste microarchitetture viene utilizzato il **pipelining**, che si applica ad processoare a ciclo singolo in quanto ne migliora le performance;

VANTAGGI:

- Esegue più istruzioni contemporaneamente;
- Si può accedere a dati e registri contemporaneamente;

SVANTAGGI:

- Logica di controllo più complessa;
- Richiede registri di pipeline;

MISURA DELLE PRESTAZIONI

Ci sono molti modi per misurare le performance di un processore;

Una misura affidabile consiste nel valutare le prestazioni di tempo rispetto all'esecuzione di un insieme fissato di programmi, che prende le nome di **BENCHMARK**.

Il tempo di esecuzione è calcolato come: $\left(\frac{\text{numero di istruzioni}}{\text{cicli di istruzione}} \right) \left(\frac{\text{secondi}}{\text{ciclo}} \right)$

Ricordiamo che, un'istruzione ha un ciclo di vita che

è composto da 4 FASI PRINCIPALI ovvero:

FETCH → DECODE → EXECUTE → STORE

PROGETTAZIONE

Una microstruttura è composta di 2 componenti molto importanti che interagiscono tra di loro, ovvero:

→ IL DATAPATH :

- ⇒ Determina il flusso dei dati durante l'esecuzione di un'istruzione;
- ⇒ Opera su piste dati e contiene strutture quali memoria, registro, AW e multiplexer;
- ⇒ Usiamo un DATAPATH a 32 bit;

→ L' UNITÀ DI CONTROLLO :

- ⇒ Riceve l'istruzione corrente dal DATAPATH e dice ai DATAPATH come eseguire tale istruzione;
- ⇒ Produce i valori di selezione dei multiplexer, i segnali di abilitazione alla scrittura dei registri e della memoria;

Il miglior modo di procedere nel processo di PROGETTAZIONE consiste nel considerare prima gli elementi di stato e poi aggiungere la LOGICA COMBINATORIA.

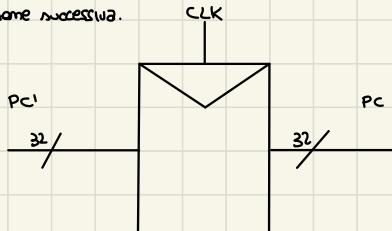
Suddividiamo la memoria in 5 ELEMENTI DI STATO:

- IL PROGRAM COUNTER;
- I REGISTRI (file register);
- IL REGISTRO DI STATO (status register)
- LE ISTRUZIONI DI MEMORIA (instruction memory)
- I DATI IN MEMORIA (data memory)

PROGRAM COUNTER (PC)

Sebbene esso logicamente fa parte del file registro, il PC viene letto e scritto ad ogni ciclo indipendentemente dagli altri registri ed è quindi implementato come se fosse un registro a 32 bit autonomo.

Ricordiamo che la sua uscita (PC), indica l'indirizzo dell'istruzione corrente, mentre invece il suo ingresso (PC') indica l'indirizzo dell'istruzione successiva.



FILE REGISTER

Consiste in 16 registri (compreso PC) da 32 bit (R_0, \dots, R_{15}),

Dal punto di vista logico i registri non sono EQUIVALENTI, infatti a livello strutturale svolgono funzioni diverse.

In particolare:

R_{13} : Viene detto **STACK POINTER**;

R_{14} : Viene detto **LINK REGISTER**;

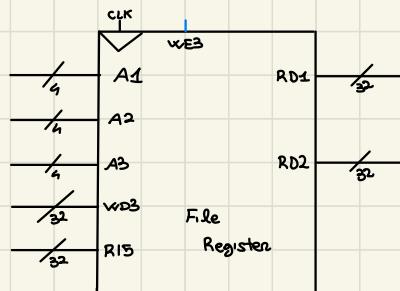
R_{15} : Provvisoriamente detto **PROGRAM COUNTER**;

Ie **FILE REGISTER** ha 2 porte per la lettura A_1 e A_2 ed una porta per la scrittura A_3 .

Ciascuna porta di lettura ha un input di 4 bit, che specificano uno dei registri come operando che viene poi letto nei registri RD_1 e RD_2 .

La porta di scrittura ha:

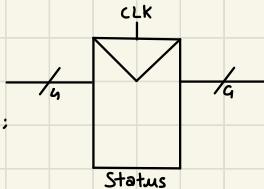
- Un indirizzo di 4 bit A_3 (dove scrivere);
- Un elemento di 32 bit WD_3 (cosa scrivere);
- Un segnale di WRITE ENABLE (WE_3);



Se $WE = 1$, e dato WD_3 , viene scritto nel registro specificato da A_3 ;

STATUS REGISTER

Memorizza i flags **N, Z, C, V** forniti dalle ALU, al fine di eseguire istruzioni condizionate;



INSTRUCTION MEMORY

Essa ha una singola porta di lettura A , che indica l'indirizzo di un'istruzione che verrà letta nel registro RD ;

DATA MEMORY

I Data Memory ha una singola porta di lettura/scrittura;

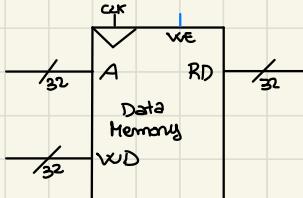
Quando il segnale WE (Write Enable) è attivo, essa scrive

il dato WD è attivo, essa scrive il dato nella cella puntata

da A durante il fronte alto del clock;



Quando, invece, il segnale **WE** è 0, essa legge i dati dalla cella più antica dell'indirizzo **A** durante il fronte alto del clock e li pone nel registro **RD**.



DATAPATH LDR

Ricordiamo che il Program Counter, contiene l'indirizzo dell'istruzione da eseguire;

Il primo passo è quello di leggere l'istruzione della memoria istruzioni, per cui il PC viene collegato all'indirizzo di ingresso della memoria di istruzioni;

L'istruzione a 32 bit viene letta ed è rappresentata dall'etichetta **Instn.**

Il passo successivo è quello di leggere il registro contenente l'indirizzo di base.

L'indirizzo di questo registro è specificato nel campo **Rm** dell'istruzione;

Questi bit vengono collegati all'ingresso di una delle porte del FILE REGISTER (**A1**). Infine il FILE REGISTER legge il valore di registro in **RD1**.

L'istruzione **LDR** richiede anche un **offset**, il quale è memorizzato nell'istruzione stessa e corrisponde a **S12**.

L'OFFSET è un valore nemmeno negativo, quindi deve essere esteso a 32 bit;

Il valore a 32 bit (**ExtImm**) è tale che i primi 20 bit sono uguali a 0 e gli altri 12 bit sono uguali a **S12**;

Inoltre il processore deve aggiungere l'indirizzo di base all'offset per trovare l'indirizzo di memoria a cui leggere;

La somma viene effettuata dall'**ALU**.

Quest'ultima riceve 2 operandi: (**srcA** & **srcB**) e grazie all'**ALUControl** sa che operazione effettuare, infatti la **SOMMA** è indicata con 00, se il bit **J = 1**, invece, la **SOTTRAZIONE** è indicata con 01.

L'**ALU** genera un valore a 32 bit detto **ALUReset**, che viene inviato alla memoria dati come indirizzo di lettura;

I dati vengono poi letti dalla memoria dati, se bns **ReadData** e poi scritti nel Registro di Destinazione alla fine del ciclo;

Il registro di destinazione per l'istruzione **LDR** è specificato nel campo **RD** dell'istruzione che è collegato all'indirizzo di ingresso **A3**.

Contemporaneamente il processore deve calcolare l'indirizzo della successiva istruzione **PC'**.

Pensiamo ricordando che le istruzioni sono a 32 (4 byte) quindi l'istruzione successiva si trova a **PC + 4**.

Si utilizza una **sommazione** per incrementare il **PC** di 4. Il nuovo indirizzo viene scritto nel contenitore di programmazione nel successivo fronte di salita del clock.

Infine, l'istruzione successiva, potrebbe essere letta anche dalla memoria e quindi corrispondere al contenuto di **ReadData** per questo motivo si potrebbe usare un **MUX** che permette di selezionare tra:

- * 0 - **PC + 4**
- * 1 - **ReadData**

Il segnale di controllo associato al multiplexer è **PCSrc**.

DATAPATH STR

L'istruzione **STR** scrive una parola di 32 bit contenuta in un registro nella memoria centrale. In modo in cui questa operazione viene effettuata dipende dalla politica di indirizzamento specificata.

Come **LDR**, **STR** legge un indirizzo di base data, porta **A1** del FILE REGISTER e completa il immediato.

Il registro è specificato nel campo **Rd** dell'istruzione e collegato alla porta **A2** del FILE REGISTER.

Il valore del registro viene letto sulla porta **RD2**, che è collegata alla porta dati di scrittura (**WD**) della memoria dati;

L'abilitazione del segnale di scrittura **WE** è controllata da **Memwrite**, che se è 1 i dati devono essere scritti in memoria;

Il segnale **AwControl** deve essere impostato a 00 per sommare l'indirizzo di base e l'offset.

Il segnale **Regwrite** è impostato a 0, perdendo cioè dove essere scritto nel FILE REGISTER.

DATAPATH ADD, SUB, AND, ORR

Estendiamo le DATAPATH per le istruzioni per l'elaborazione dati: **ADD, SUB, AND, ORR** utilizzando la modalità di indirizzamento immediato.

In tal caso, le istruzioni hanno come operandi, un **registro** ed una **costante**. L'**AW** esegue l'operazione e le risposte vengono scritte in un terzo registro.

Esse differiscono solo nella specifica operazione eseguita dall'**AW**. Quindi possono essere implementate tutte con lo stesso hardware utilizzando diversi segnali: **AWControl**.

I valori per **AWControl** sono:

L'**AW** imposta anche dei bit in **AWFlags**:

- ▶ ADD - 00;
- ▶ SUB - 01;
- ▶ AND - 10;
- ▶ ORR - 11;
- ▶ Zero;
- ▶ Negativo;
- ▶ Carry;
- ▶ overflow;

Le istruzioni per l'elaborazione dati utilizzano costanti di 8 bit per cui il blocco **Extend** riceve in input un segnale di controllo

ImmSrc:

- ▶ $\text{ImmSrc} = 0 \rightarrow \text{ExtImm}$ è esteso da **Imm8**;
- ▶ $\text{ImmSrc} = 1 \rightarrow \text{ExtImm}$ è esteso da **Imm12**;

Inoltre dobbiamo evidenziare anche il fatto che il File Register può ricevere l'input sia dalla memoria dati che dall'ALU;

Di conseguenza aggiungeremo un altro multiplexer che permette di selezionare la sorgente tra ReadData e ALUResult.

L'uscita del multiplexer è indicata con Reset;

Il multiplexer richiede un segnale di controllo, ovvero MemtoReg:

- MemtoReg = 0 → input preso da ALUResult;
- MemtoReg = 1 → input preso da ReadData;

Le istruzioni di data processing con indirizzamento da registro ricevono la loro seconda fonte da Rm specificato nella istruzione. Di conseguenza aggiungeremo un multiplexer sugli ingressi del file register. In base al valore del segnale di controllo RegSrc, RA2 può essere selezionato fra:

- Rd → per STR;
- Rm → per le istruzioni di elaborazione dati con indirizzamento da registro;

Infine, aggiungeremo un ulteriore multiplexer sugli ingressi delle ALU per selezionare questo secondo registro sorgente.

In base al valore del segnale di controllo ALUSrc, la seconda sorgente sarà selezionata fra:

- ExtImm per istruzioni che utilizzano le costanti;
- Due file register per istruzioni per l'elaborazione dati con indirizzamento da registro;

DATAPATH ISTRUZIONI DI BRANCHING le istruzioni di branching vedere pag. 41

Dato che PC + 8 è letto dalla prima parte del file register è necessario un multiplexer per selezionare R15 come ingresso di RA1. Il multiplexer è controllato dal segnale RegSrc, il cui valore è preso dai specifici bit della istruzione (Imstr_{19:16}) per la maggior parte delle istruzioni ed è impostata a 15 per le istruzioni di branch.

MemtoReg è impostato a 0 e PCSrc è impostato a 1 per selezionare il nuovo PC da ALUResult.

UNITÀ DI CONTROLLO

L'unità di controllo calcola i segnali di controllo in base a:

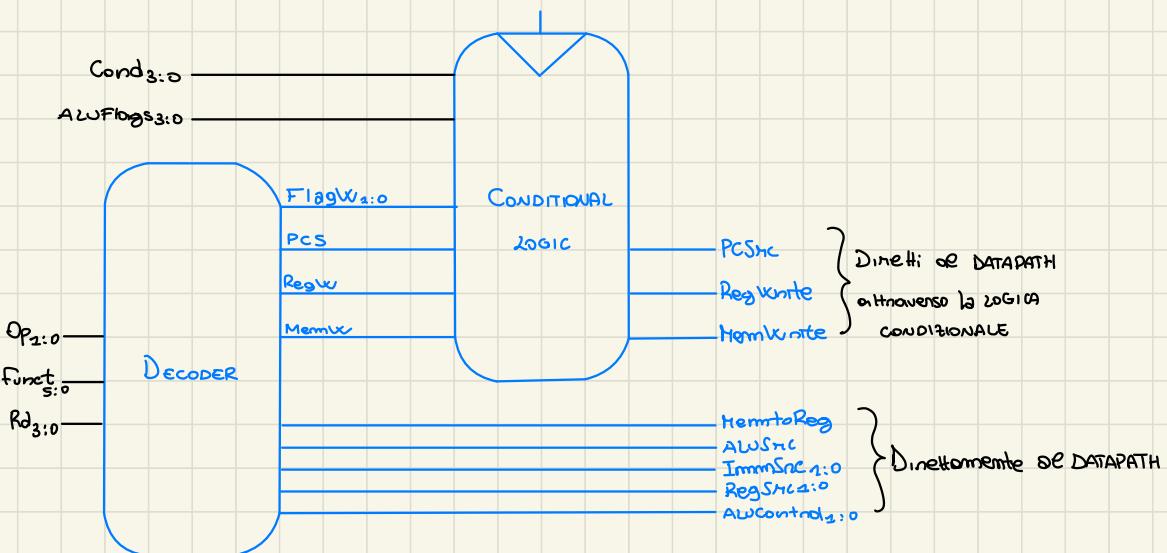
- I campi cond, op e funct dell'istruzione;
- I FLAG;
- Se il registro di destinazione è il PROGRAM COUNTER;

L'unità di controllo si può dividere in 2 parti principali:

- Il **DECODER**, che genera i segnali di controllo sulla base dei campi dell'istruzione.
- La **LOGICA CONDIZIONALE**, che gestisce i flag di stato e li aggiorna quando l'istruzione deve essere eseguita su condizione.

Il **DECODER** è composto da:

- + Un **DECODIFICATORE PRINCIPALE**, che produce la maggior parte dei segnali di controllo;
- + Un **DECODER ALU**, che utilizza il campo Funct per determinare il tipo di istruzione per l'elaborazione dati;
- + La **LOGICA DI CONTROLLO DEL PC**, che determina se il PC deve essere aggiornato a causa di un branch o di una scrittura in R15.



FlagW_{1:0}: Flag Write Signal, indica quando le ALUFlags devono essere aggiornate avendo quando un'un'istruzione S=1;

ADD, SUB: aggiorniamo tutti i flag (**NZCV**); **ADD, ORR:** aggiorniamo solo N e Z;

Quindi sono necessari 2 bit:

- $\text{FlagW}_1 = 1 \rightarrow \text{NZ}$
- $\text{FlagW}_0 = 1 \rightarrow \text{CV}$

DECODER

IE DECODER:

- Determina il tipo di istruzione: Elaborazione dati con registro o costante, STR, LDR o B
- Produce segnali di controllo per le DATAPATH. Infatti alcuni segnali vengono inviati direttamente ai DATAPATH, come MemtoReg, ALUOp, ImmSinc_{4:0} e RegSinc_{4:0}.
- Genera i segnali che abilitano la scrittura (MemW e RegW), i quali devono passare attraverso la LOGICA CONDIZIONALE prima di diventare segnali DATAPATH (Memwrite e Regwrite)
- Genera i segnali Branch e ALUOp, utilizzati rispettivamente per indicare l'istruzione B o il tipo di istruzione dell'elaborazione dati.

La logica per le DECODER principale può essere svolta dalla tabella di verità;

LOGICA CONDIZIONALE

I segnali che abilitano la scrittura (MemW and RegW) e l'appagamento dei flag (FlagWrite) e del PC (PCS) devono passare attraverso la logica condizionale prima di diventare operativi.

ANALISI DELLE PRESTAZIONI

Ogni istruzione nel processo a ciclo singolo impiega un ciclo di clock, quindi il CPI è 1;

I CRITICAL PATH per l'istruzione LDR sono:

- t_{pcq_pc} → caricamento di un nuovo indirizzo (PC) due fronte di salita del clock;
- t_{mem} → lettura dell'istruzione in memoria;
- t_{dec} → il decoder principale calcola RegSinc0 e i file register legge questo registro;
- $\max(t_{mux} + t_{RFread}, t_{ext} + t_{mux})$ → Mentre i file register viene letto, il campo costante viene esteso e viene selezionata da multiplexer per determinare SincB.
- t_{ALU} → L'ALU somma SincA e SincB per trovare l'indirizzo effettivo;
- t_{rmem} → La memoria di dati legge da questo indirizzo;
- t_{mux} → I file multiplexer MemtoReg seleziona ReadData;
- $t_{RFsetup}$ → Viene impostato il segnale Reset e il risetato viene scritto nei file register.

Il tempo totale è dato dalla somma dei puntuali:

$$T_{C_2} = t_{pcq_pc} + t_{mem} + \max(t_{mux} + t_{RFread}, t_{mux} + t_{ext}) + t_{ALU} + t_{rmem} + t_{mux} + t_{RFsetup};$$

Nella maggior parte delle implementazioni, c'è ALU, la MEMORIA e le FILE REGISTER sono più lenti di altri blocchi combinatori, quindi il tempo di ciclo può essere semplificato come:

$$T_{C_1} = t_{PC \rightarrow PC} + 2t_{mem} + t_{dec} + t_{RF\text{read}} + t_{ALU} + 2t_{max} + t_{RF\text{setup}}$$

LIMITI DEL CICLO SINGOLO

Le architetture a ciclo singolo hanno 3 limiti principali, ovvero:

1. **SEPARAZIONE DELLE MEMORIE** → la memoria istruzioni e la memoria dati devono essere separate dato che i dati e le istruzioni devono essere gestite nello stesso ciclo;
2. **INEFFICIENZA TEMPORALE** → il ciclo di clock deve avere una lunga pari al tempo impiegato da un'istruzione più lenta, sprecando tempo per le istruzioni più veloci;
3. **DUPPLICAZIONE DEGLI COMPONENTI** → lo stesso componente non può essere utilizzato per scopi distinti;

VANTAGGI DEL CICLO MULTIPLO

Le architetture a ciclo multiplo risolvono tali problemi, partizionando un'istruzione in più passi, ciascuno dei quali viene eseguito in un ciclo di clock differente;

- ▶ È possibile utilizzare una sola memoria comune sia per le istruzioni che per i dati. Inoltre, un'istruzione viene letta in un ciclo, mentre i dati vengono letti o scritti in memoria in un ciclo differente;
- ▶ Istruzioni meno complesse richiedono un minor numero di cicli di clock, evitando sprechi di tempo;
- ▶ È possibile utilizzare una unica ALU sia per gestire il PROGRAM COUNTER che per eseguire le istruzioni, purché però tali operazioni siano effettuate in cicli di clock differenti;

DATAPATH

Ie **DATAPATH** è sviluppato in modo incrementale.

Al fine di facilitare la comprensione delle architetture di un processore ARM, consideriamo un limitato set di istruzioni:

- Le istruzioni di elaborazione dati: **ADD**, **SUB**, **AND** e **ORR** (con registro e modalità di indirizzamento diretto e **register shift**);
- Le istruzioni di memoria: **LDR**, **STR** (diretta o con offset positivo);
- Le istruzioni di salto (branch): **B**;

ELEMENTI DI STATO A PIÙ CICLI

Sostituiamo la memoria delle istruzioni e la memoria dei dati, con un'unica memoria che comprende entrambe.

DATAPATH LDR

Ie **PC** contiene l'indirizzo dell'istruzione da eseguire. Il primo passo è quello di leggere questa istruzione dalla memoria istruzioni, per cui il **PC** viene collegato all'indirizzo di ingresso della memoria;

L'istruzione a **32 bit** viene letta e memorizzata in un registro **IR** (**Instruction Register**). Quest'ultimo riceve un segnale **IRwrite** che indica quando caricare l'istruzione.

Il passo successivo è quello di leggere il registro corrente contenente l'indirizzo di base. Questo registro è specificato nel campo **Rm** dell'istruzione. I 4 bit vengono collegati all'ingresso di una delle porte del FILE REGISTER (**A1**). IL FILE REGISTER legge il valore di registro in **RD1** e lo memorizza in **A**.

L'istruzione **LDR** richiede anche un **OFFSET** (12 bit), ie quale è memorizzato nell'istruzione stessa;

L'offset è un valore norma negra, quindi deve essere esteso a 32 bit.

Il valore a **32 bit** (**ExtImm**) è tale che $\text{ExtImm}_{31:12} = 0 \wedge \text{ExtImm}_{11:0} = \text{Imstr}_{11:0}$

ExtImm estende a 32 bit costanti a 8, 12 e 24 bit. Non viene memorizzato in un registro perché dipende solo dall' **istruzione**.

Il processore deve aggiungere l'**indirizzo di base** all'offset per trovare l'indirizzo di memoria a cui seguire. La somma viene effettuata da un **ALU**. Quest'ultima riceve 2 operandi (srcA e srcB). srcA proviene dal **FILE REGISTER**, mentre, invece, srcB proviene da ExtImm . Inoltre, il segnale a 2 bit **ALUControl** specifica l'operazione, cioè l'**ADDITIONE** è **00** e la **SOTTRAZIONE** è **01**.

l' **ALU** genera un valore a 32 bit, **ALUReset**, che viene memorizzato in un registro **ALUOut**.

L'indirizzo calcolato dall'**ALU** deve essere inviato alla porta **A** della memoria.

Però serve un multiplexer per distinguere l'accesso con **ALUOut** o **PC**. Il multiplexer è controllato dal segnale **AdrSel**. I dati vengono letti dalla memoria dati sul bus **ReadData** e poi vengono memorizzati in un registro chiamato **Data**. Inoltre i dati appena letti devono essere scritti nel registro **Rd** specificato nello stesso **istruzione**. Piuttosto che collegare direttamente **Data** alla porta **WD3**, si consideri che anche se rispettato dese' **ALU** permette essere scritto in **Rd**, per questo si utilizza un multiplexer che seleziona tra **ALUOut** e **Data**. Il segnale di **Regwrite** deve essere impostato a **1**, per permettere la scrittura nel registro.

Un ulteriore compito dell'**ALU** è l'**incremento del PC**, operazione che prima veniva fatta da un'altra ALU. Si aggiunge poi un multiplexer sul primo ingresso dell'**ALU**, che permette di scegliere tra le contenute del registro **A** e il **PC**. Se secondo ingresso dell'**ALU** aggiungiamo un multiplexer che permette di selezionare o **ExtImm** o la costante **G**.

Si consideri infine, che il contenuto del registro **R15** corrisponde a **PC+8**. Durante il passo di fetch, se **PC** è stato aggiornato a **PC+4**, per cui sommare 4 al nuovo contenuto di **PC** produce **PC+8** che viene memorizzato in **R15**. Per scrivere **PC+8** in **R15** c'è bisogno che il risultato dell'**ALU** sia collegato a tale registro. A tal fine collegheremo **ALUReset** con uno dei 3 ingressi del multiplexer.

DATAPATH ELABORAZIONE DATI

Per le istruzioni dell'elaborazione dati con COSTANTE (**ADD**, **SUB**, **AND**, **OR**) il datapath legge il primo operando specificato da **Rm** estende la costante da 8 a 32 bit, esegue l'operazione mediante l'**ALU** e scrive il risultato in un registro del **FILE REGISTER**.

L'operazione da effettuare è specificata dal segnale **ALUControl**, mentre gli **AluFlags** permettono di aggiornare il registro di stato.

Per le istruzioni dell' elaborazione dati con REGISTRO (**ADD**, **SUB**, **AND**, **OR**) ie datapath legge il secondo operando specificato da **Rm** presente nell' istruzione.

Inseriamo un multiplexer per selezionare tale campo nella porta **A2** del File REGISTER. Il multiplexer è controllato dal segnale **RegSrc**.

DATAPATH BRANCH

Per le istruzioni di branch, ie datapath legge PC+8 e una costante a 26 bit, che viene estesa a 32 bit. La somma di questi 2 valori è addizionata al PC.

Si ricorda, inoltre, che il registro **R15** contiene il valore PC+8 e deve essere letto per tornare da un salto. È sufficiente aggiungere un multiplexer per selezionare **R15** come input sulla porta **A1**. Il multiplexer è controllato dal segnale **RegSrc**.

UNITÀ DI CONTROLLO

Come nel processore a ciclo singolo, c'è un'unità di controllo, generando i segnali di controllo in base ai campi **cond**, **op** e **funct** dell' istruzione, ai **Reg** e al fatto che il registro di destinazione sia o meno il PC.

Anche in questo caso l' unità di controllo è suddivisa in **DECODIFICATORE** e **LOGICA CONDIZIONALE**.

DATAFLOW DI UN'ISTRUZIONE

L' unità di controllo produce segnali di attivazione per tutto il datapath.

Uno stato dell' automa che implementa il **main decoder** non è altro che lo stato dei segnali in un determinato momento.

Per avere una visione più chiara di quali sono gli stati e di come vengono effettuate le transizioni da uno stato all' altro è utile considerare il **DATA FLOW** del processore.

ANALISI DELLE PRESTAZIONI

In un processore a ciclo multiplio, il tempo impiegato per eseguire un' istruzione dipende dal numero di cicli di clock di cui necessita e dalla durata di un singolo ciclo di clock.

Il numero di cicli di clock necessario ad eseguire le diverse istruzioni è di:

- BRANCH → 3 cicli;
- DATA PROCESSING → 4 cicli;
- MEMORY STORE → 4 cicli;
- MEMORY LOAD → 5 cicli;

I percorsi CRITICI nel datapath che richiedono maggior tempo, sono 2:

- ✗ Dal PC, attraverso i multiplexer SmCA, attraverso l'ALU, attraverso i multiplexer ResSet, attraverso la porta RIS, fino al registro A;
- ✗ Da ALUOut, attraverso il registro ResSet, attraverso i multiplexer Adr, attraverso la memoria (mem), fino al registro Data;

$$T_{C_2} = t_{PCQ-PC} + 2t_{MUX} + \max [t_{ALU} + t_{MUX}, t_{MEM}] + t_{Setup};$$

PROCESSORE ARM CON PIPELINE

- 1) PARALLELISMO TEMPORALE;
- 2) DIVIDE IL PROCESSORE A CICLO SINGOLO IN 5 FASI:

- FETCH;
- DECODE;
- EXECUTE;
- MEMORY;
- WRITEBACK;

- 3) AGGIUNGE REGISTRI DI PIPELINE TRA LE FASI;

MEMORIE

Nell'architettura Von Neumann il canale di comunicazione tra la CPU e la MEMORIA è il punto critico del sistema;

La tecnologia consente di realizzare CPU sempre più veloci e memorie sempre più grandi, ma la velocità di accesso delle memorie non cresce così rapidamente come quella della CPU.

Storicamente le CPU sono sempre state più veloci delle MEMORIE.

Al giorno d'oggi siamo in grado di produrre delle memorie veloci quanto una CPU, ma:

- ✖ queste memorie hanno un costo elevatissimo;
- ✖ le memorie dovrebbero essere piazzate in gran parte sugli stessi chip della CPU, il che non è possibile;

Per ovviare a questo problema, gli ingegneri ricorrono ad uno schema chiamato **A GERARCHIA DI MEMORIA** in cui si combineranno:

- Una quantità molto piccola di memoria estremamente veloce;
- Una quantità molto grande di memoria lenta;

Inquadramento del Problema

La maggior parte del tempo di esecuzione di una CPU è solitamente impegnato da procedure in cui vengono eseguite ripetutamente le stesse istruzioni. In realtà, non è importante conoscere lo schema della sequenza di istruzioni, il punto è che molte istruzioni in sede ben localizzata del programma vengono eseguite ripetutamente in un determinato periodo e si accede al resto del programma a stento;

Questa proprietà viene detta **LOCALITÀ DEI RIFERIMENTI**. Si manifesta in 2 modi:

◆ **LOCALITÀ TEMPORALE**, che rappresenta la probabilità che l'istruzione eseguita di recente venga eseguita di nuovo entro breve tempo;

◆ **LOCALITÀ SPAZIALE**, che invece, rappresenta la probabilità che istruzioni vicine ad un'istruzione eseguita di recente, siano anche esse eseguite nel prossimo futuro.

CACHE

- 1 - Livello più alto nella gerarchia della memoria;
- 2 - È veloce;
- 3 - L'ideale è quello di fornire la maggior parte dei dati al processore;
- 4 - Solitamente contiene i dati a cui si è avuto accesso recentemente;

Urtà di avere una cache memory

La velocità con cui la memoria risponde alle richieste di istruzioni e dati della CPU, ha un peso importante sulle prestazioni di un sistema.

Infatti, se tra la memoria principale e la CPU si potesse inserire una memoria molto veloce, contenenti le parti di un programma e i dati che interessano l'elaborazione, il tempo totale di esecuzione verrebbe ridotto di molto.

Questa è esattamente la funzione della **cache memory**.

Si tratta quindi di un elemento che inserito tra la CPU e la memoria principale impedisce alla CPU di vedere i tempi di risposta reali della memoria;

La struttura della CPU, non viene influenzata dalla presenza o meno di una CACHE MEMORY.

PRINCIPI DI FUNZIONAMENTO

Le operazioni svolte da una memoria cache sono molto semplici. I circuiti di controllo della memoria cache sono progettati per sfruttare al meglio le proprietà della LOCALITÀ DEI RIFERIMENTI.

■ L'aspetto TEMPORALE suggerisce di portare un elemento nella cache quando viene richiesto per la prima volta, in modo tale che rimanga a disposizione nel caso di una nuova richiesta.

■ L'aspetto SPAZIALE suggerisce che, invece, di spostare un elemento alla volta dalla memoria principale alla cache, è conveniente portare un insieme di elementi che risiedono in indirizzi adiacenti.

Un altro termine utilizzato di frequente per indicare un blocco della cache è la **linea di cache**;

DIMENSIONE DELLA CACHE

Affinché la cache svolga efficacemente il suo compito deve avere tempi di accesso molto più brevi di quelli della memoria principale e ciò però impone che sia piccola e che quindi potrà contenere solo una frazione ridotta delle istruzioni ed dei dati della memoria principale;

TECNICHE DI GESTIONE

La corrispondenza tra i blocchi della memoria principale e quelli della cache è specificata dalla funzione di **posizionamento (mapping)**.

Quando la cache è piena e si fa riferimento ad una parola della memoria che non è presente nella cache, l'hardware di controllo della cache deve decidere quale blocco della cache debba essere rimosso per far spazio al nuovo blocco, che contiene la parola a cui si fa riferimento.

L'insieme delle regole in base alle quali viene fatta questa scelta costituisce **L'ALGORITMO DI SOSTITUZIONE**.

La struttura dei sistemi è di per sé organizzata in modo che la presenza di una cache non influenti il funzionamento della CPU. Quest'ultima, infatti, nell'esecuzione del programma effettua le richieste di lettura e scrittura utilizzando gli indirizzi delle locazioni nella memoria principale.

Sarà poi la logica di controllo della CACHE che si farà carico di determinare se la parola è presente o meno nella cache. Se è presente, viene effettuata l'operazione di lettura o scrittura della locazione di memoria appropriato. In questo caso si dice che l'accesso in LETTURA o SCRITTURA ha avuto successo.

Possibili Gestioni di un "cache hit"

Se è' accesso alla cache ha avuto successo bisogna distinguere 2 tipi di situazioni:

- **OPERAZIONE DI LETTURA**, in cui la memoria principale non viene coinvolta;

- **OPERAZIONE DI SCRITTURA**, in cui il sistema può procedere in 2 modi:

• **WRITE-THROUGH** → La locazione della cache e quella della memoria principale vengono entrambe aggiornate.

• **WRITE-BACK** → È più semplice, ma implica inutile operazioni di scrittura nella memoria

principale, se una ponda viene aggiornata più volte durante il periodo in cui è nella cache.

- WRITE-BACK → Aggiorna soltanto la locazione della memoria cache, segnalando come aggiornata con un bit di modifica o dirty.

Anche se protocollo write-back può causare inutile scrittura nella memoria principale, visto che quando si procede con la scrittura nella memoria principale di un blocco, tutte le ponde del blocco vengono scritte, anche se solo una delle ponde del blocco della cache è stata modificata.

GESTIONE DI UN "MISS READ"

Quando la ponda indirizzata durante un'operazione di lettura non è presente nella cache si dice che l'accesso in lettura è fallito (read miss).

Il blocco di ponde contenente la ponda richiesta deve essere copiato dalla memoria principale alla cache.

A questo punto ci sono le possibilità:

- Dopo aver caricato l'intero blocco nella cache, la ponda richiesta viene inviata alla CPU;
- È possibile inviare immediatamente la ponda alla CPU, non appena la si legge dalla memoria principale;

Quest'ultimo approccio, chiamato LOAD-THROUGH o anche EARLY RESTART, riduce il tempo di attesa della CPU, a discapito di una maggiore complessità del circuito di controllo della cache;

GESTIONE DI UN "WRITE MISS"

Durante un'operazione di scrittura, se la ponda indirizzata non è quella nella cache, si dice che l'accesso in scrittura è fallito (write miss).

Anche in questo caso ci sono 2 opzioni:

- + Se si utilizza un protocollo write-through, le informazioni vengono scritte direttamente nella memoria principale;
- + Se, invece, si utilizza un protocollo write-back, il blocco, contenente la ponda indirizzata viene prima caricato nella cache, poi viene sovrascritto con le nuove informazioni;

PRESTAZIONI

$$\text{Hit Rate} = \frac{\# \text{hits}}{\# \text{memory accesses}} = 1 - \text{Miss Rate}$$

$$\text{Miss Rate} = \frac{\# \text{misses}}{\# \text{memory accesses}} = 1 - \text{Hit Rate}$$

TEMPO MEDIO DI ACCESSO ALLA MEMORIA (AMAT): tempo medio che il processore impiega per accedere ai dati;

$$\text{AMAT} = t_{\text{cache}} + \text{MR}_{\text{cache}} [t_{\text{MM}} + \text{MR}_{\text{MM}} (t_{\text{VM}})]$$

Esempio:

2.000 load e stores

1.250 presenti nella cache

se resto è fornito in altri livelli nella gerarchia della memoria

$$\text{Hit Rate} = 1250 / 2000 = 0.625$$

$$\text{Miss Rate} = 750 / 2000 = 0.375 = 1 - \text{Hit Rate}$$

Ora assumiamo che un processore ha 2 livelli di memoria: cache e memoria principale

$$t_{\text{CACHE}} = 1 \text{ cycle} \quad t_{\text{MM}} = 100 \text{ cycles}$$

$$\text{AMAT} = ?? \rightarrow t_{\text{CACHE}} + \text{MR}_{\text{cache}} (t_{\text{MM}}) = [1 + 0.375 \cdot (100)] \text{ cycles} = 38.5 \text{ cycles}$$

TERMINOLOGIA CACHE

- ◆ **CAPACITÀ (C)** → Numero di byte nella cache;
- ◆ **DIMENSIONE DEL BLOCCO (B)** → Grandezza di un blocco, ovvero numero di byte che sono trasmessi nella cache per volta;
- ◆ **NUMERO DI BLOCCHI (B = C/b)** → Numero di blocchi nella cache;
- ◆ **GRADO DI ASSOCIAZIVITÀ (N)** → Numero di blocchi in un SET;
- ◆ **NUMERO DI SETS (S = B/N)** → Ogni indirizzo di memoria è mappato in esattamente un set della cache;

TIPOLOGIE DI MEMORIE CACHE

La memoria cache è organizzata in S set, ogni indirizzo di memoria mappa in esattamente un unico set. Diverse categorie di memoria cache dipendono dal numero di blocchi che un set contiene:

- + **MAPPATO DIRETTAMENTE**: 1 blocco per ogni set;
- + **N-WAY SET ASSOCIATIVO**: N blocchi per set;
- + **COMPLETAMENTE ASSOCIATIVO**: Tutti blocchi di una cache sono in un unico set

BIT DI VALIDITÀ

Un ulteriore bit di controllo, chiamato **BIT DI VALIDITÀ**, è necessario per ogni blocco; questo bit indica se il blocco contiene o meno DATI VALIDI.

Non deve però essere confuso con il **BIT DI MODIFICA**, che invece indica se il blocco è stato modificato o meno durante il periodo in cui è stato nella cache, nerne soltanto nei sistemi che non fanno uso del metodo **write-through**.

Bisogna ricordare che i **bit di validità** dei blocchi che si trovano nella cache vengono tutti posti a 0 nell'istante iniziale di accensione del sistema e in seguito ogni volta che nella memoria principale vengono caricati programmi e dati nuovi;

IL DIRECT MEMORY ACCESS (DMA)

I trasferimenti da disco alla memoria principale vengono effettuati mediante un meccanismo detto **DMA**, ovvero **Direct Memory Access**. In questo meccanismo la CPU non interviene, perché è la logica di controllo del disco che gestisce le linee di indirizzo e le linee dati, fornendo i segnali di controllo necessari.

In un'operazione di questo genere vengono trasferite grandi quantità di istruzioni e dati organizzati in entità dette "**pagine**" ciascuna delle quali, di solito, contiene migliaia di **locazioni** e quindi centinaia di **blocchi**. Normalmente, le pagine di programma e quelle di dati vanno nella memoria centrale senza coinvolgere la cache;

GESTIONE DEL BIT DI VALIDITÀ

Ie bit di validità di un certo blocco della cache viene posto a "1" la prima volta che esso è chiamato a contenere istruzioni o dati trasferiti dalla memoria principale; Per ogni voce che le locazioni di un blocco della memoria principale vengono interessate da un trasferimento di nuove istruzioni o nuovi dati da disco, viene effettuato un controllo per determinare se qualcuno dei blocchi che stanno per essere sovrascritti fossero o meno presenti nella cache;

Se nella cache c'è una copia del blocco, e suo bit di validità viene impostato a "0", in tal modo si garantisce che nella cache non ci siano dati obsoleti e inoltre quando il bit di validità "0", il blocco si candida immediatamente ad essere sovrascritto;

Svuotamento della Cache (Flush)

Lo svuotamento della cache (flush), forza i dati con il bit di modifica attivo ad essere ricopiati nella memoria principale prima che venga effettuato il trasferimento tramite DMA.

COERENZA DELLA CACHE

Il sistema operativo è in grado di fare tutte le operazioni in modo molto efficiente, senza peggiorare le prestazioni. Questa necessità di garantire che 2 diverse entità utilizzino la stessa copia dei dati viene chiamata problema della **COERENZA DELLA CACHE**

ALGORITMO DI SOSTITUZIONE

In una cache ad indirizzamento diretto, la posizione di ogni blocco è predefinita, quindi non esiste alcuna strategia di sostituzione.

Nelle memorie **cache associative** e **set-associative** esiste invece una certa flessibilità; infatti, quando un nuovo blocco deve essere posto nella cache e tutte le posizioni che potrebbe occupare contengono dati validi, la controllore della cache deve decidere quale blocco, tra quelli esistenti, sovrascrivere, e questa è una scelta importante, perché la scelta potrebbe essere determinante per le prestazioni del sistema;

In generale l'obiettivo è quello di mantenere nella cache quei blocchi che hanno una maggiore possibilità di essere nuovamente utilizzati nel prossimo futuro, ma non è facile prevedere i blocchi a cui si farà riferimento;

Una possibile strategia è di tenere presente che la **località dei riferimenti** suggerisce che i blocchi a cui c'è stato accesso di recente hanno elevata probabilità di essere utilizzati nuovamente in breve tempo.

SOSTITUZIONE CON ALGORITMO LRU

Quando bisogna eliminare dalla cache un blocco, ha senso **risarcire** quello a cui non si accede da più tempo. Tale blocco prende le forme di blocco **utilizzato meno di recente** (**Least Recently Used, LRU**) e la tecnica utilizzata prende le forme di **ALGORITMO DI SOSTITUZIONE LRU**.

Per realizzare questa tecnica, il controllore della cache deve mantenere traccia di tutti gli accessi ai blocchi durante l'elaborazione. Il blocco LRU di una memoria cache **SET-ASSOCIAUTA** con insiemi di 4 blocchi può essere indicato da un contatore di 2 bit associato a ciascun blocco.

GESTIONE DEL CONTATORE DI ACCESSI

Abbiamo 2 casi possibili:

▲ Quando si ha un successo nell'accesso al blocco (**HIT**):

In questo caso il contatore di quel blocco viene posto a 0. I contatori con valori originalmente inferiori a quelli del blocco a cui si accede, vengono incrementati di uno, mentre tutti gli altri rimangono invariati.

▲ Quando, invece, l'accesso fallisce (**MISS**):

In questo caso abbiamo 2 possibilità:

- L'insieme **NON È PIENO**, il contatore associato al nuovo blocco caricato dalla memoria principale viene posto a 0 e le valori di tutti gli altri contatori viene incrementato ad 1.
- L'insieme **È PIENO**, si rimuove il blocco, il cui contatore ha valore 3 e si pone il nuovo blocco al suo posto, mettendo il contatore a 0. Gli altri 3 contatori dell'insieme vengono incrementati di 1.

ALTERNATIVE ALL' ALGORITMO LRU

L'algoritmo LRU è stato utilizzato ampiamente, in numerosi schemi di accesso, però in alcuni situazioni, come nel caso di accessi sequenziali ad una vettore di elementi, che è leggermente grande per poter stare tutto nella cache, le prestazioni sono molto scadenti;

Sono stati proposti molti altri algoritmi di sostituzione, che richiedono talvolta anche notevoli complicazioni hardware, ma l'algoritmo è più semplice e spesso anche più efficace, consiste nello scegliere in modo completamente casuale i blocchi da sostituire.

TIPI DI ERRORI

1. **OBLIGATORIO**: Primo accesso ai dati;
2. **CAPACITÀ**: Cache troppo piccola per contenere tutti i dati d'interesse;
3. **CONFLITTO**: I dati d'interesse vengono mappati alla stessa posizione nella cache;

PENALITÀ MANCATA: Tempo necessario per recuperare un blocco dal livello inferiore della gerarchia;

TENDENZE DEL MISS RATE

- Le cache più grandi riducono i miss di capacità;
- Una maggiore assocattività riduce i miss di conflitto;
- I blocchi più grandi riducono gli miss obbligatori;
- I blocchi più grandi aumentano gli miss di conflitto;

Data una cache di dimensioni fissate, all'aumentare del block size il numero di set diminuisce ($S \times N \times B = C$), quindi aumentano i miss di conflitto.

CACHE MULTILIVELLO

- Le cache più grandi hanno tassi di errori inferiori e tempo di accesso più lunghi;
- Espande la gerarchia della memoria a più livelli di cache:

- Livello 1: Piccolo e VELOCE;
- Livello 2: GRANDE e LENTO;

P.s.: I computer moderni dispongono di cache 1st, 2nd, 3rd

Vi possono essere 2 tipologie di cache multi-nivello:

INCLUSIVA:

1. L2 contiene i dati presenti in L1;
2. L2 contiene sia i dati contenuti in L1, sia quelli che erano presenti in precedenza che sono stati sovrascritti tramite l'algoritmo di sostituzione COPY BACK;
3. Per mantenere la coerenza L1 ha una politica di write-through verso L2 e così L2 verso L3;
4. Miss in L1 e hit in L2: il blocco relativo viene copiato da L2 a L1;
5. Miss in L1 e L2: il blocco dalla memoria principale viene copiato sia in L2 che in L1;

ESCLUSIVA:

1. L2 contiene solo i dati di COPY BACK da L1;
2. La memoria complessiva della cache è data dalla somma delle capacità dei vari livelli;
3. Miss in L1 e hit in L2: le linee di cache di L1 e L2 vengono scambiati fra loro, cioè la linea di cache di L1 viene rimemorizzata in L2 e viceversa;
4. Miss in L1 e L2: il dato letto dalla memoria è rimemorizzato direttamente in L1 e la linea di cache rimpiazzata di L1 è trasferita in L2 rimpiazzando un'altra linea di cache;

CACHE CONDIVISE

Nelle architetture multi-core L2 e L3 possono essere condivisi fra più core di uno stesso processore, questo consente:

★ Un uso più efficiente del livello condiviso: Se un core è INATTIVO, allora è altro può utilizzarne per se il livello condiviso;

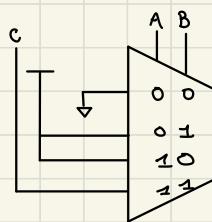
★ Nella programmazione multi-threading consente di utilizzare i dati condivisi in maniera più efficiente: Un core può eseguire un pre-/post-processing di dati forniti da un altro core.

★ Riduce il front-side bus traffic: Un unico livello condiviso che si interfaccia con la memoria principale;

Es. 5 della prova

$$Y = A(\bar{B}\bar{C} + C) + \bar{A}B + BC = \\ = A\bar{B}\bar{C} + AC + AB + BC$$

A	B	C	Y
0	0	0	0
0	0	1	0
0	1	0	1
0	1	1	1
1	0	0	1
1	0	1	1
1	1	0	0
1	1	1	1



Es 3 della prova

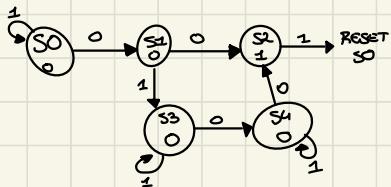
T_{BONIA} 1

Ex. $\overline{11100} \quad \overline{110101011}$

T_{BONC} 0

000 01 00 0000 100

STEGNE 1 MACCHINA DI MOORE :



Traccia 4 Gennaio 2022

N° 6

DXA - obbligatorio Vedere a casa !!