

## VALORE MASSIMO RAPPRESENTABILE

Riferendosi alla base 10, il numero massimo rappresentabile con  $m$  cifre è:

$$v_{\max} = 10^m - 1$$

Quindi la formula generale data una **BASE**  $b$  e  $m$  **CIFRE** è:

$$v_{\max} = b^m - 1$$

OSS

$\lceil m \rceil$  = primo intero maggiore di  $m$

$\lfloor m \rfloor$  = primo intero minore di  $m$

## CIFRE NECESSARIE PER RAPPRESENTARE $m$

Sappiamo che data una base  $b$  ed un numero di cifre  $m$  il più grande numero rappresentabile è  $b^m - 1$ .

Sappiamo quindi che il nostro  $m$  sarà minore o uguale di questa quantità.

Scrivendo questa eguaglianza possiamo dire che:

$$m = \lceil \log_b(m+1) \rceil$$

$m$  = cifre    $b$  = base    $m$  = numero

## CAMBIO DI BASE

Dato  $m$  in base  $a$ , cambiiamo la base in  $b$ .

Dobbiamo dividere  $m$  per  $b$  finché il quoziente non è 0.

La sequenza dei **RESTI** sarà la codifica di  $m$  in base  $b$

Ej.

$251_{10}$  in base 3

$$251/3 = 83 + 2 \text{ HENDO SIGNIFICATIVA}$$

$$83/3 = 27 + 2$$

$$27/3 = 9 + 0$$

$$251_{10} = 100022_3$$

$$9/3 = 3 + 0$$

$$3/3 = 1 + 0$$

$$1/3 = 0 + 1 \text{ PIU' SIGNIFICATIVA}$$

## BASE NON DECIMALE

Se vogliamo codificare  $333_7$  in base 9, potremo dividere il problema in due passi:

① Codifico  $333_7$  in base 10

$$333_7 = 3 \cdot 7^2 + 3 \cdot 7 + 3 \cdot 7^0 = 171$$

② Codifico  $171_{10}$  in base 9

$$171/9 = 19 + 0$$

$$19/9 = 2 + 1$$

$$171_{10} = 210_9$$

$$2/9 = 0 + 2$$

### CODIFICA BINARIA E ESADECIMALE

I componenti digitali operano in codice binario

la codifica binaria porta risultati estremamente lunghi,

introduciamo quindi la base **ESADECIMALE**

le cifre vanno da "0 a 9" e da "a ad f"

Si come son 16 caratteri rappresentabili,  $16 = 2^4$  posiamo

codificare ogni cifra a 4 bit, viceversa 4 bit del sistema binario

corrispondono a una cifra in base esadecimale

$$0 = 0000_2$$

$$6 = 0110_2$$

$$C = 1100_2$$

$$1 = 0001_2$$

$$7 = 0111_2$$

$$d = 1101_2$$

$$2 = 0010_2$$

$$8 = 1000_2$$

$$e = 1110_2$$

$$3 = 0011_2$$

$$9 = 1001_2$$

$$f = 1111_2$$

$$4 = 0100_2$$

$$a = 1010_2$$

$$5 = 0101_2$$

$$b = 1011_2$$

Se abbiamo numero in base 16 da convertire in binario  
basterà semplicemente codificare le singole cifre:

$$4c9f_{16} = 0100\ 1100\ 1001\ 1111_2$$

OSS

Comunemente per indicare un numero in base 16 si antepona "0x"

$$f95_{16} = 0x f95$$

Se abbiamo un numero in base binaria eseguiamo la codifica  
inversa, aggiungendo gli opportuni 0 non significativi

$$10011_2 = 0001\ 0011_2 = 0x 13$$

### NUMERI CON PAROLE DI LUNGHEZZA FISSA

I registri dei calcolatori moderni sono a 32 o 64 bit,  
quindi (nel caso di un'architettura a 64 bit) ogni numero  
sarà rappresentato dello stesso numero di cifre (64) e quindi avranno  
numerosi 0 non significativi.

## OVERFLOW

In un'architettura a 64 bit, il più grande rappresentabile è  $2^{64}-1$ . Se in seguito ad un'operazione otteniamo un numero maggiore di  $2^{64}-1$  parleremo di un **OVERFLOW**

Ese.

$$1024^7 = (2^{10})^5 = 2^{70} > 2^{64}-1$$

## SOMMA e MOLTIPLICAZIONE BINARIA

La somma binaria è analoga a quella decimale:

Ese.

$$\begin{array}{r} 1 \ 1 \\ 1 0 \ 1 \ 1 \\ 0 \ 0 \ 1 \ 1 \\ \hline 1 \ 1 \ 1 \ 0 \end{array}$$

avremo un overflow se alla fine avremo un riporto.

Anche la moltiplicazione risulta analoga

## RAPPRESENTAZIONE DI INTERI

Rappresentiamo adesso numeri mediante parole a lunghezza fissa, senza dimenticarci del segno. Le principali rappresentazioni sono:

## RAPPRESENTAZIONE CON SEGNO

Supponiamo lunghezza  $m$ . La cifra più significativa indica il **SEGNO**:

$0=+$ ,  $1=-1$ . Le restanti  $m-1$  cifre indicheranno il valore assoluto.

Il range di numeri rappresentabili in base  $b$  sarà:

$$-(b^{m-1} - 1) \leq n \leq b^{m-1} - 1$$

Lo **SVANTAGGIO** sta nell' eseguire addizioni e sottrazioni,  
dovendo pur forza controllare prima il segno.

## ESERCIZI

$$0110011_2 = 1+2+16+32 = 51_{10}$$

$$10101000_2 = 4+8+32+128 = 172_{10}$$

$$431204_5 = 4+2 \cdot 5^2 + 5^3 + 3 \cdot 5^4 + 4 \cdot 5^5 = 14554_{10}$$

$$1938_{10} = 1938 \text{ mod } 16 = 121 + 2 = 792_{16}$$

$$121 \text{ mod } 16 = 7 + 9$$

$$7 \text{ mod } 16 = 0 + 7$$

$$10010_2 = 0001 \ 0010 = 12_{16}$$

### COMPLEMENTO A 2

La rappresentazione è analoga a quella **UNSIGNED** ma il bit più significativo corrisponde a  $-2^{m-1}$  anziché  $2^{m-1}$

$$\begin{array}{r} 10011100 \\ \Rightarrow \end{array} \begin{array}{r} -128 \ 00168400 \\ + + + + + + \end{array} = -100$$

OSS

- Lo 0 ha un'unica rappresentazione

0 0 0 0 0 .. 0

- Il **MINIMO** rappresentabile è  $-2^{m-1}$

1 0 0 0 0 .. 0

● Il **MASSIMO** rappresentabile è  $2^{m-1} - 1$

0 1 1 1 1 ... 1

● Il range di rappresentazione è  $[-2^{m-1}, 2^{m-1} - 1]$

● Encundo  $-2^{m-1}$  il valore assoluto di "puso" maggiore,  
se un numero inizia con 1 sarà sempre **NEGATIVO**

## SOMMA COMPLEMENTO A 2

La somma avviene come per i numeri unsigned:

$$-2 + 1 = 1110 + 0001 = 1111 = -1$$

$$-7 + 7 = 1001 + 0111 = 0000 \text{ e } 1 = 0$$

in questo caso il resto 1 non indica un **OVERFLOW**

## COMPLEMENTO DI UN NUMERO

Per **COMPLEMENTARE** (invertire) un numero occorre invertire ogni cifra  
e sommare 1:

$$2 = 0010 \rightarrow -2 = 1101 + 0001 = 1110$$

$$7 = 0111 \rightarrow -7 = 1000 + 0001 = 1001$$

OSS

NON vale per  $-2^{m-1}$  il cui complemento non e' rappresentabile

### OVE RFLLOW COMPLEMENTO A 2

Sommare un positivo e un negativo non genera mai un OVERFLOW

L'OVERFLOW avviene SOMMANDO due numeri CONCORDI la cui somma ha segno OPPOSTO

### ESTENSIONE DI UN NUMERO

Se vogliamo estendere la rappresentazione di un numero ad una con più bit, basta riprodurre la cifra più significativa:

$$4 = 0100 = 0000 \ 0100$$

4 bit                    8 bit

$$-5 = 1101 = 1111 \ 1101$$

4 bit                    8 bit

### BINARY CODED DECIMAL

La codifica BCD aiuta a codificare dal decimale al binario.

Siccome dobbiamo rappresentare i simboli ("0", ..., "9"), usiamo 4 bit.

Avendo però 4 bit a disposizione avremo  $2^4 = 16$  combinazioni per rappresentare 10 simboli, risulta quindi essere ridondante.

Diciamo che è una codifica per **GIUSTAPPOSIZIONE**:

$$23_{10} = \begin{matrix} 0010 & 0011 \\ \text{bcd} \end{matrix}$$

OSS

- $23_{10} = 0010 \ 0011_{\text{bcd}} \neq 00100011_2 = 35_{10}$
- La codifica BCD viene usata solo per semplificare la visualizzazione di numeri binari. In quanto tale, **NON** esiste un'aritmetica BCD (vengono bomialmente codificate le somme)

### CAMBI DI BASE DECIMALE

Eseguiamo il cambio base di un numero  $x$  da una base "a" a "b"

Separiamo parte intera e decimale (abbiamo già visto per la parte intera)  
Per la parte **FRAZIONARIA** eseguiamo il procedimento inverso:

- Moltiplichiamo la parte frazionaria di " $x$ " per "b"  
La parte INTERA del risultato sarà un numero da 0 a  $b-1$  che **CONVERTITO** in base "b" costituisce la prima cifra frazionaria.

- Iteriamo quest'operazione sul risultato finché la parte frazionaria non è 0 oppure abbiamo raggiunto la precisione desiderata

E.

Convertiamo in binario  $0,625_{10}$

$$0,625 \cdot 2 = 1,250 \quad i=1 \quad f=0,250$$

$$0,250 \cdot 2 = 0,500 \quad i=0 \quad f=0,500$$

$$0,500 \cdot 2 = 1,000 \quad i=1 \quad f=0,000$$

$$0,625 = 0,101_2$$

Convertiamo  $0,65_8$  in base 7 fino alla 3 cifra significativa

- Convertiamo prima in decimale

$$0,6 \cdot 8^{-1} + 5 \cdot 8^{-2} = 0,828125_{10}$$

$$0,828125 \cdot 7 = 5, 796875$$

$$0,796875 \cdot 7 = 5, 578125$$

$$0,578125 \cdot 7 = 4, 046875$$

$$0,658_8 = 0,554_7$$

## RAPPRESENTAZIONE CON VIRGOLA FISSA

Rappresentiamo ora positivi frazionari di lunghezza m.

Nella rappresentazione in virgola fissa si suddividono gli m bit in 2 sottopartite

- I primi h bit sono per la parte intera
- I rimanenti K bit rappresentano la parte frazionaria

E.

$$7,25_{10} = 0111,0100 \quad 4 \text{ bit e } 4 \text{ bit}$$

$$7_{10} = 0111_2$$

$$0,25 \cdot 2 = 0,5$$

$$0,5 \cdot 2 = 1,0$$

## RAPPRESENTAZIONE A VIRGOLA MOBILE

A differenza della rappresentazione a virgola fissa, con questa non definiamo a priori la lunghezza della parte intera e frazionaria

I numeri in base b sono così rappresentati:

$$x = (-1)^s m b^e$$

- s determina il **SEGNO** ( $0+$ ,  $1-$ )

- m è la **MANTISSA**

- e è l' **ESPOLENTE**

Ogni numero reale viene quindi rappresentato dalla tripla  $(s, m, e)$

Siccome i modi per rappresentare un numero sarebbero infiniti  
poniamo  $1 \leq m < b$  (**RAPPRESENTAZIONE SCIENTIFICA**)

In binario quindi  $m = 1, xyz$  ( $b=2$ )

Inoltre, essendo l'1 sempre presente, lo daremo per sottinteso.

## STANDARD IEEE 754

Viene utilizzata come rappresentazione delle variabili a virgola FISSA (tipi FLOAT o DOUBLE).

Utilizza 4 diversi formati a seconda della PRECISIONE (singola a 32 bit o doppia a 64 bit, equivalenti alle variabili float o double) e del TIPO (semplice o esteso).

In particolare :

- SINGOLA precisione (32 bit totali)

1 bit per il SEGNO, 23 per la MANTISSA e 8 per l'ESPOENTE

- DOPPIA precisione (64 bit totali)

1 bit per il SEGNO, 52 per la MANTISSA e 11 per l'ESPOENTE

### OSS

Con il complemento a 2 l'ordine fra positivi e negativi viene invertito (i negativi iniziano con 1 e saranno ovviamente maggiori), per ovviare a questo problema lo Standard IEEE utilizza la RAPPRESENTAZIONE POLARIZZATA

## RAPPRESENTAZIONE POLARIZZATA

Dati  $K$  bit per l'espONENTE, il più grande espONENTE rappresentabile (in complemento a 2) è  $P = 2^{K-1} - 1$  (Per offset **OFFSET**)

In rappresentazione polarizzata, un valore  $e$  (compreso tra 0 e  $2^k - 1$ ) codifica l'**ESPONENTE**  $e' = e - P$

E.s.

Se ho 3 bit per espONENTE,  $P = 2^{3-1} - 1 = 3$

$$e=0 \rightarrow e' = 0 - 3$$

$$e=1 \rightarrow e' = 1 - 3 = -2$$

Così facendo gli espONENTI sono codificati in ordine crescente.

## DETtagli dello standard IEEE

- Per la mantissa si adotta la rappresentazione scientifica
- Alla tripla  $(s, m, e)$  è associato il numero  $x = (-1)^s \cdot 1, m \cdot 2^{e-P}$
- IEE 754 precisione **SINGOLA** (8 bit per espONENTE), l'**OFFSET** è  $P = 2^{8-1} - 1 = 127$
- IEE 754 precisione **DOPPIA** (11 bit per espONENTE), l'**OFFSET** è  $P = 2^{11-1} - 1 = 1023$

## ECCEZIONI DELLO STANDARD IEEE

Lo 0 rappresenta un' eccezione (perché per come è definito  $x$ , non potremo mai avere 0)

precisione singola

	$e=0 \ (00000000_2)$	$e=1, \dots, 254$	$e=255 \ (11111111_2)$
$m=0$	$(-1)^s \times 0$	$(-1)^s \times 1,0 \times 2^{e-127}$	$(-1)^s \infty$
$m \neq 0$	$(-1)^s \times 0, m \times 2^{-126}$	$(-1)^s \times 1, m \times 2^{e-127}$	NaN (indefinito)

- Per  $m=0$  ed  $e=0$  otteniamo due possibili rappresentazioni dello 0 (segno positivo e negativo)
- Per  $m=0$  ed  $e=255$  abbiamo  $\pm \infty$

E

Rappresentiamo 1021 in STANDARD IEEE ( $s, m, e$ )

$(0; 111\ 1111\ 0100\ 0000\ 0000\ 0000; 1000\ 1000)$

- $s=0$  perché positivo

- Rappresentazione binaria di 1021 è:

$$1\ 111\ 111\ 01 = 1,111\ 111\ 01 \cdot 2^9 \quad (\text{notazione scientifica})$$

$m = 111\ 111\ 01\ 00\ 0000\ 0000\ 0000 \quad (\text{m ha 23 bit})$

**RICORDA:** la mantissa considera solo i decimali

- Avendo 8 bit per l'esponente,  $P = 2^{8-1} - 1 = 127$

L'esponente e si ricava dalla formula inversa

$$e = e' + P = 9 + 127 = 136 \quad (9 \text{ per } 2^9)$$

che convertito in binario sarà  $136_{10} = 1000\ 1000_2$

## OPERAZIONI IN VIRGOLA MOBILE

Consideriamo due numeri  $(s_1, m_1, e_1)$  e  $(s_2, m_2, e_2)$

### MOLTIPLICAZIONE

Il prodotto tra due numeri ha per risultato la tripla tale che:

- $s = 0$  se  $s_1 = s_2$ , altrimenti  $s = 1$
- $e = e_1 + e_2$
- $m = m_1 \circ m_2$

Di solito dopo l'operazione è necessaria la normalizzazione del risultato.

Analogamente vale per la **DIVISIONE**

### ADDITIONE e SOTTRAZIONE

Risultano essere molto più complesse in quanto prima di eseguire l'operazione bisogna rendere uguali gli **ESPOVENTI**.

Questa operazione può portare a una **PERDITA di CIFRE SIGNIFICATIVE** per via dello scompenso delle mantisse

### PERDITA DI CIFRE SIGNIFICATIVE

Prenotiamo in considerazione due numeri decimali con mantissa a 4 cifre e esponente a una cifra:

$$m_1 = 1,3435 \cdot 10^3$$

$$m_2 = 1,9970 \cdot 10^5$$

per effettuare la somma partiamo l'1 e, da 3 a 5, dividendo poi la mantissa per 100 affinché l'uguaglianza resti valida

$$m_1 = 0,01345 \cdot 10^5$$

poiché le cifre per la mantissa sono 4, dobbiamo effettuare un **ARROTONDAMENTO** per difetto:

$$m_1 = 0,0134 \cdot 10^5$$

A questo punto la somma sarà  $2,0104 \cdot 10^5$  (anziché  $2,010435 \cdot 10^5$ )

### CODIFICA CARATTERI ALFANUMERICI

I processori moderni associano ad ogni **CARATTORE** un **NUMERO**

Un testo o data dalla **GIUSTAPPOSIZIONE** della codifica

dei singoli carattere. Le codifiche più comuni sono:

### ASCII

Codice a 7 bit: 95 caratteri stampabili e 33 di controllo

### OSS

In realtà l'ASCII sfruttava 8 bit, l'ottavo era detto di **CONTROLLO** in quanto serviva a verificare che durante una trasmissione, per fenomeni esterni, il codice non fosse stato modificato.

Succivamente i macchinari divennero più affidabili e fu creato l'**EXTENDED ASCII** a 8 bit, con simboli grafici e accenti

### UNICODE

Codice a 21 bit (1 milione di simboli). Attualmente sono definiti solo 128.000 caratteri

### UTF-8

E' una codifica UNICODE **VARIABILE** (da 1 a 4 byte)

E' retrocompatibile con ASCII ed e' consigliata per XML e HTML

## CODICE ASCII

I primi 32 numeri sono di controllo

- **NULL** indica la fine di una stringa
- **CARRIAGE RETURN** indica il fine riga

## ASCII TABLE

Decimal	Hex	Char	Decimal	Hex	Char	Decimal	Hex	Char	Decimal	Hex	Char
0	0	[NULL]	32	20	[SPACE]	64	40	@	96	60	`
1	1	[START OF HEADING]	33	21	!	65	41	A	97	61	a
2	2	[START OF TEXT]	34	22	"	66	42	B	98	62	b
3	3	[END OF TEXT]	35	23	#	67	43	C	99	63	c
4	4	[END OF TRANSMISSION]	36	24	\$	68	44	D	100	64	d
5	5	[ENQUIRY]	37	25	%	69	45	E	101	65	e
6	6	[ACKNOWLEDGE]	38	26	&						
7	7	[BELL]	39	27	,						
8	8	[BACKSPACE]	40	28	(	72	48	H	104	68	h
9	9	[HORIZONTAL TAB]	41	29	)	73	49	I	105	69	i
10	A	[LINE FEED]	42	2A	*	74	4A	J	106	6A	j
11	B	[VERTICAL TAB]	43	2B	+	75	4B	K	107	6B	k
12	C	[FORM FEED]	44	2C	,	76	4C	L	108	6C	l
13	D	[CARRIAGE RETURN]	45	2D	-	77	4D	M	109	6D	m
14	E	[SHIFT OUT]	46	2E	.	78	4E	N	110	6E	n
15	F	[SHIFT IN]	47	2F	/	79	4F	O	111	6F	o
16	10	[DATA LINK ESCAPE]	48	30	0	80	50	P	112	70	p
17	11	[DEVICE CONTROL 1]	49	31	1	81	51	Q	113	71	q
18	12	[DEVICE CONTROL 2]	50	32	2	82	52	R	114	72	r
19	13	[DEVICE CONTROL 3]	51	33	3	83	53	S	115	73	s
20	14	[DEVICE CONTROL 4]	52	34	4	84	54	T	116	74	t
21	15	[NEGATIVE ACKNOWLEDGE]				5	85	U	117	75	u
22	16	[SYNCHRONOUS IDLE]				6	86	V	118	76	v
23	17	[END OF TRANS. BLOCK]	55	37	7	87	57	W	119	77	w
24	18	[CANCEL]	56	38	8	88	58	X	120	78	x
25	19	[END OF MEDIUM]	57	39	9	89	59	Y	121	79	y
26	1A	[SUBSTITUTE]	58	3A	:	90	5A	Z	122	7A	z
27	1B	[ESCAPE]	59	3B	:	91	5B	[	123	7B	{
28	1C	[FILE SEPARATOR]	60	3C	<	92	5C	\	124	7C	
29	1D	[GROUP SEPARATOR]	61	3D	=	93	5D	]	125	7D	}
30	1E	[RECORD SEPARATOR]	62	3E	>	94	5E	^	126	7E	~
31	1F	[UNIT SEPARATOR]	63	3F	?	95	5F	-	127	7F	[DEL]

## CODICE UTF-8

La codifica UTF-8 codifica l'unicode, in cui il primo byte indica la LUNGHEZZA della lunghezza

Primo byte	Byte totali	Bit a disposizione del carattere
0xxx xxxx	1	7
110x xxxx	2	11
1110 xxxx	3	16
1111 Oxxx	4	21

Tutti i byte successivi hanno nella sequenza  $10xx xxxx$

Ese

Consideriamo "€", codice Unicode U+20AC

È un codice di 16 bit, quindi richiede 3 byte in UTF-8 (perché  
dobbiamo inserire anche i "bit fissi" quindi 2 byte non bastano)

$0x 20AC = 0010 0000 1010 1100$

Codifica UTF-8

$1110 0010 | 1000 0010 | 1010 1100$   
  
3byte

I bit ROSSI sono quelli fissi (vedi tabella), indipendentemente dal carattere che si va a codificare

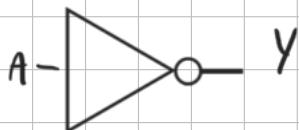
# ALGEBRA DI BOOL

## PORTE LOGICHE

Sono i **COMPONENTI DIGITALI** che realizzano le operazioni, prendendo dati in input e restituendo dati output, costituiti da **VARIABILI BOOLEANE** (rappresentate da 0 e 1)

## PORTE NOT

La porta **NOT** restituisce in output il **COMPLEMENTO** di A

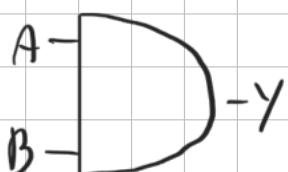


A	y
0	1
1	0

$$y = \bar{A} \text{ (NOT A)}$$

## PORTE AND

La porta **AND** restituisce in output la **CONGIUNZIONE** degli input



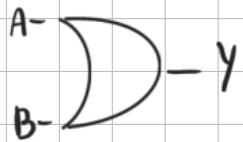
A	B	y
0	0	0
0	1	0
1	0	0
1	1	1

$$y = AB$$

- $y=1$  se e solo se A e B sono uguali a 1

## PORTE OR

La porta OR restituisce in output la DISGIUNZIONE degli input



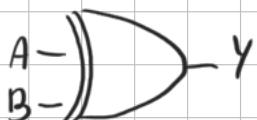
A	B	y
0	0	0
0	1	1
1	0	1
1	1	1

$y = A + B$

- $y = 1$  se e solo se  $A = 1 \vee B = 1$

## PORTE XOR

- $y = 1$  se e solo se  $A = 1 \vee B = 1$



A	B	y
0	0	0
0	1	1
1	0	1
1	1	0

$y = A \oplus B$

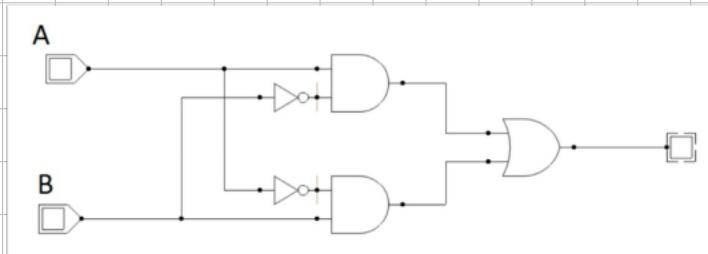
## OSS

La porta XOR può essere ottenuta mediante porte OR, AND e NOT

- XOR:  $y = 1 \Leftrightarrow A \neq B$

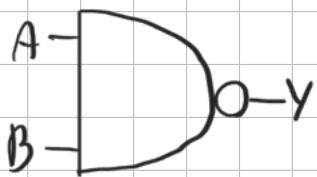
- $A \neq B \Leftrightarrow (A = 1 \wedge B = 0) \vee (A = 0 \wedge B = 1)$

- $y = (A \bar{B}) + (\bar{A} B)$



## PORTE NAND

- $Y=1$  se e solo se  $A \circ B$  sono diversi da 1

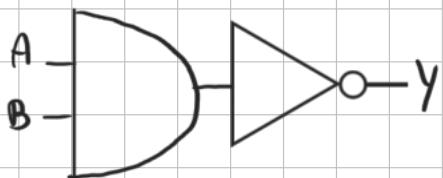


A	B	Y
0	0	1
0	1	1
1	0	1
1	1	0

$y = \overline{AB}$

OSS

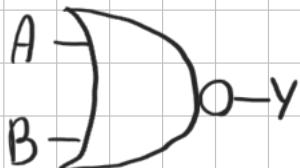
La porta NAND può essere ottenuta mediante porte AND e NOT



A	B	$\overline{A}$	$\overline{B}$	$\overline{AB}$
0	0	1	1	1
0	1	1	0	1
1	0	0	1	1
1	1	0	0	0

## PORTE NOR

- $Y=1$  se e solo se  $A \neq 1$  e  $B \neq 1$

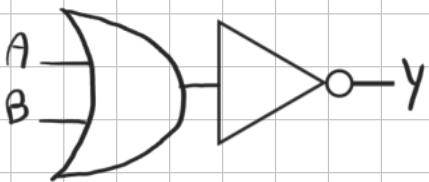


A	B	Y
0	0	1
0	1	0
1	0	0
1	1	0

$y = \overline{A+B}$

OSS

La porta NOR può essere ottenuta mediante porte OR e NOT



A	B	$A+B$	$\overline{A+B}$
0	0	0	1
0	1	1	0
1	0	1	0
1	1	1	0

## PORTE XNOR

- $Y=1$  se e solo se  $A=B$



A	B	$\overline{A \oplus B}$
0	0	1
0	1	0
1	0	0
1	1	1

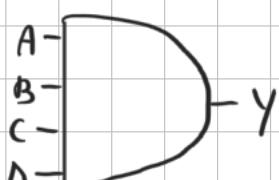
OSS

Lo XNOR equivale alla negazione dello XOR ( $A \oplus B$ )

## PORTE LOGICHE CON PIU' LINEE DI INPUT

Le porte possono avere anche piu' linee di input:

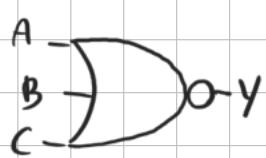
AND4



$$Y = ABCD$$

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

NOR 3



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

A	B	C	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	0

## 0,1 POTENZIALE ELETTRICO

Gli 0 e 1 letti dalla macchina corrispondono ovviamente a delle grandezze fisiche, in particolare al **POTENZIALE ELETTRICO**

- 0 = 0V (GRD)

- 1 =  $V_{DD}$   $V_{DD} \approx 1,5V$  (prima era 5V)

OSS

Siccome il Potenziale elettrico è soggetto a del rumore, esistono delle soglie di tolleranza

## FUNZIONI BOOLEANE

Possiamo vedere le porte logiche come delle **FUNZIONI BOOLEANE**

che hanno come dominio e codominio  $\{0,1\}$

$$f: \{0,1\}^n \rightarrow \{0,1\}$$

*n variabili*

Quante funzioni booleane di  $n$  variabili esistono?

Immaginiamo che ogni combinazione di variabili sia una parola di lunghezza  $n$  ( $n$  variabili) con 2 caratteri  $(0,1)$ .

Avremo  $2^n$  parole (quindi combinazioni)

Possiamo vedere le funzioni come parole di lunghezza  $2^n$  (le nostre combinazioni) con 2 caratteri, quindi  $2^{2^n}$  funzioni

## FUNZIONI BOOLEANE A 2 VARIABILI

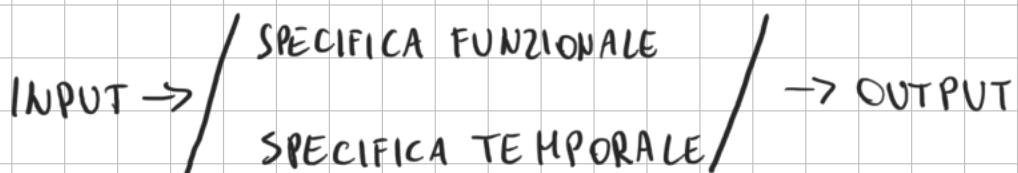
Se abbiamo 2 variabili avremo  $2^{2^2} = 16$  funzioni (XOR, AND, OR...)

## CIRCUITI DIGITALI

E' una **RETE** che elabora segnali discreti (rappresentati da variabili booleane).

Possiamo vedere un **CIRCUITO** come una **BLACK BOX** con:

- Uno o più INPUT
- Uno o più OUTPUT
- Una **SPECIFICA FUNZIONALE** che rappresenta la relazione tra INPUT e OUTPUT (espressioni booleane)
- Una **SPECIFICA TEMPORALE** che descrive il ritardo che intercorre affinché i segnali di INPUT si propaghino nel circuito fino agli output



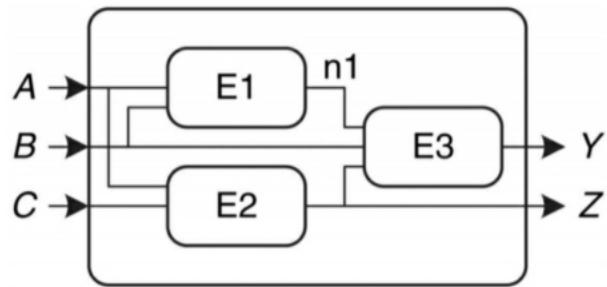
## STRUTTURA INTERNA DEI CIRCUITI DIGITALI

La **STRUTTURA INTERNA** dei circuiti digitali è composta da **NODI** ed **ELEMENTI**:

- Un **ELEMENTO** stesso è un circuito digitale
- Un **NODO** è una connessione che trasporta il segnale (filo elettrico)

Ne esistono di 3 tipi :

- modo **INPUT** riceve segnale dall'esterno
- modo **OUTPUT** porta il segnale all'esterno
- modo **INTERNO** connette due elementi



I circuiti digitali si dividono in **RETI COMBINATORIE** e **SEQUENZIALI**

### RETI COMBINATORIE

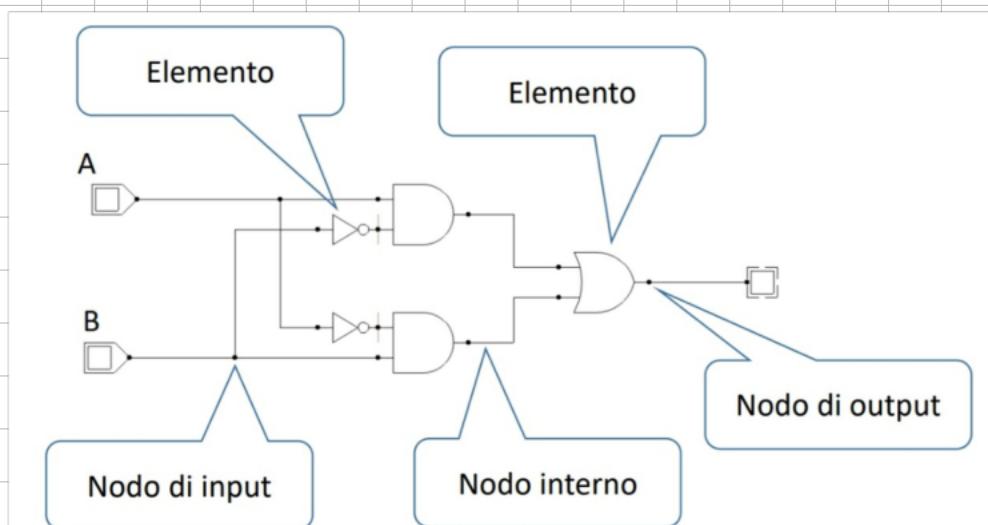
In una rete **COMBINATORIA** i valori di INPUT e OUTPUT

dipendono esclusivamente dal valore corrente degli INPUT.

Per questo si dicono **MEMORYLESS**

Le **PORTE LOGICHE** rappresentano un esempio di **RETE COMBINATORIA**

Ecco un altro esempio



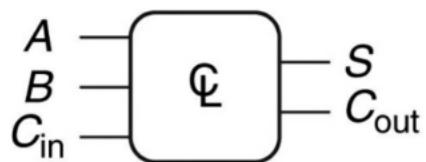
## RETI SEQUENZIALI

Nelle reti sequenziali i valori di output non dipendono solo dai valori correnti di input, bensì anche dai valori precedenti, hanno

### MEMORIA

## FULL ADDER

Per generalizzare la struttura interna di una RETE COMBINATORIALE possiamo così descriverla



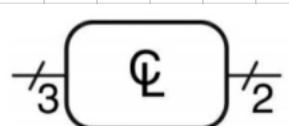
### FULL ADDER

$$S = A \oplus B \oplus C_{in}$$
$$C_{out} = AB + AC_{in} + BC_{in}$$

$C_{in}$  e  $C_{out}$  rappresentano eventuali RIPORTI delle operazioni

## INPUT e OUTPUT MULTIPLI

Se non abbiamo intenzione a specificare le variabili, possiamo così generalizzare



Ottenevi così un adder con due input multipli.



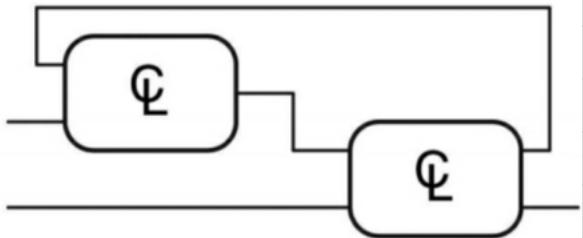
OSS

Il n° di input della seconda rete dev'essere uguale al numero di OUTPUT

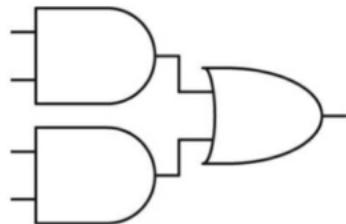
### REGOLE DI COMPOSIZIONE DI RETI COMBINATORIE

E' possibile combinare più reti combinatorie affinché si crei un'ulteriore rete combinatoria più grande, rispettando 3 regole:

- Ogni ELEMENTO è una RETE COMBINATORIA
- Ogni nodo che non è INPUT connette ESATTAMENTE un OUTPUT di ogni elemento
- IL circuito NON presenta CICLI



NO



SI

## TABELLE DI VERITA' ED ESPRESSIONI BOOLEANE

le **SPECIFICHE FUNZIONALI** sono espresse da tabelle di verità ed espressioni booleane, questo significa che c'è una forte correlazione tra le due cose, in particolare è possibile passare da una tabella di verità a un'espressione booleana e viceversa.

## ESPRESSIONI → TABELLE

Questa conversione è molto banale in quanto basta calcolare tutte le singole tavole di verità.

E.s.

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

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

• Accumula gli 0 (AND)

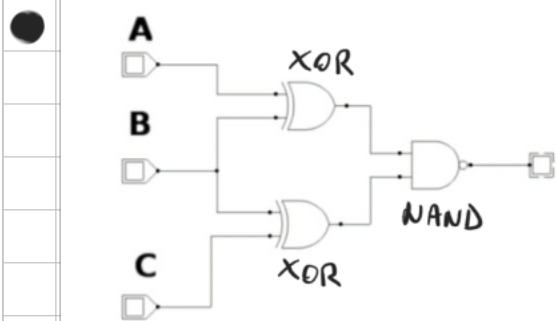
+ Accumula gli 1 (OR)

# ESERCIZI

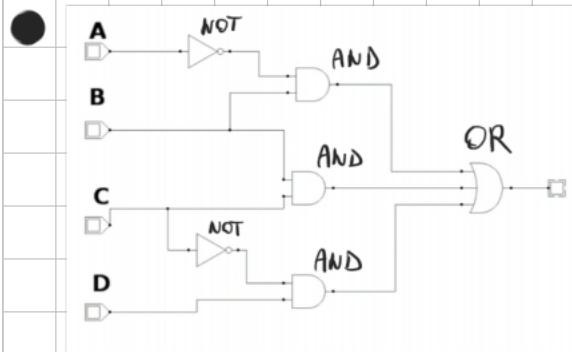
•  $(A\bar{B}) + (\bar{A}B)$

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

A	B	$A \oplus B$
0	0	0
0	1	1
1	0	1
1	1	0



A	B	C	$A \oplus B$	$B \oplus C$	$(A \oplus B) \oplus (B \oplus C)$
0	0	0	0	0	1
0	0	1	0	1	1
0	1	0	1	1	0
0	1	1	1	0	1
1	0	0	1	0	1
1	0	1	1	1	0
1	1	0	0	1	1
1	1	1	0	0	1



A	B	C	D	$\bar{A}$	$\bar{C}$	$\bar{A}B$	$BC$	$\bar{C}D$	$\bar{A}B + BC + CD$
0	0	0	0	1	1	0	0	0	0
0	0	0	1	1	1	0	0	1	1
0	0	1	0	1	0	0	0	0	0
0	0	1	1	1	0	0	0	0	0
0	1	0	0	1	1	1	0	0	1
0	1	0	1	1	1	1	0	1	0
0	1	1	0	1	0	1	1	0	1
0	1	1	1	1	0	1	0	1	1
1	0	0	0	0	1	0	0	0	0
1	0	0	1	0	1	0	0	1	1
1	0	1	0	0	0	0	0	0	0
1	0	1	1	0	0	0	0	0	0
1	1	0	0	0	1	0	0	0	0
1	1	0	1	0	1	0	0	1	1
1	1	1	0	0	0	0	1	0	1
1	1	1	1	0	0	0	1	0	1

## TABELLE → ESPRESSIONI

La formalizzazione di una tabella di verità si divide in:

### FORMA SOP

Vediamo alcune definizioni:

- Una variabile booleana A o la sua negata sono dette **LITERALI**
- Un prodotto (AND) di literali è detto **IMPONENTE**
  - $\bar{A}B, \bar{A}\bar{B}\bar{C}, B$  sono tutti implicanti
- Dato un insieme K di variabili booleane, un **HINTERMINE** di K è un impONENTE che comprende (positive e negative) **TUTTE** le variabili di K
  - Se  $K = \{A, B, C\}$  allora  $\bar{A}\bar{B}\bar{C}$  è un mintermine di K, mentre  $\bar{A}B$  è un impONENTE ma non un mintermine
- Analogamente un **MAXTERMINE** di K è una somma in cui occorrono **TUTTI** i termini di K
  - $A + B + C$  è maxtermine di  $K = \{A, B, C\}$
- Esiste un **ORDINE DI PRECEDENZA** che è:  
NOT → AND → OR
  - $\bar{A}\bar{B}\bar{C} + C\bar{B} = (\bar{A}\bar{B}\bar{C}) + C\bar{B}$

## OSS

- Ogni MINTERMINE di un dato K corrisponde a una specifica configurazione INPUT
- $0 = \bar{A}$      $1 = A$

Ogni minitermine viene enumerato per riga ( $m_1, m_2, \dots$ )

Per ottenere la FORMA NORMALE SOP, prendiamo tutti i minitermini associati a 1 e li SOMMIAMO tra di loro.

IN	OUT	MINTERMINI	FORMA SOP			
A	B	Y	$\bar{A}\bar{B}$	$\bar{A}\bar{B}$	$\bar{A}B$	$\bar{A}\bar{B} + \bar{A}\bar{B} + AB$
0	0	1	1	0	0	1
0	1	0	0	0	0	0
1	0	1	0	1	0	1
1	1	1	0	0	1	1

$$Y = \sum (0,45) \text{ notazione forma SOP}$$

I minitermini 1 che abbiamo individuato daranno 1 SOLO con la configurazione (riga) che abbiamo individuato.

## OSS

- accumula gli 0 ed associa agli input il MINIMO tra loro
- + accumula gli 1 ed associa agli input il MASSIMO tra loro

## FORMA POS

La forma POS è detta anche DUALE

Ad ogni riga corrisponde un MAXTERMINE

OSS

Nella forma POS  $0=A$   $1=\bar{A}$

Anche i maxtermini vengono numerati per riga ( $M_1, M_2, \dots$ )

La FORMA NORMALE POS di una funzione booleana

si ottiene come PRODOTTO dei MAXTERMINI

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

Prendiamo i maxtermini = 0 e

ne faccio il PRODOTTO

$Y = \prod(0, 2)$  notazione forma POS

## FORME SOP E POS

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

POS		
A	B	Y
$A+B$		
0	0	0
0	1	1
1	0	1
1	1	1

Prenderemo come espressione per il nostro circuito la più ECONOMICA

Quindi sceglieremo la forma SOP se ci sono pochi 1, altrimenti la forma POS (in questo caso la forma SOP)

## ASSIOMI DELL' ALGEBRA DI BOOLE

	Axiom	Dual	Name
A1	$B = 0$ if $B \neq 1$	$A1'$ $B = 1$ if $B \neq 0$	Binary field
A2	$\bar{0} = 1$	$A2'$ $\bar{1} = 0$	NOT
A3	$0 \bullet 0 = 0$	$A3'$ $1 + 1 = 1$	AND/OR
A4	$1 \bullet 1 = 1$	$A4'$ $0 + 0 = 0$	AND/OR
A5	$0 \bullet 1 = 1 \bullet 0 = 0$	$A5'$ $1 + 0 = 0 + 1 = 1$	AND/OR

① I valori di una variabile booleana sono solo 0 e 1

② Definizione dell' operatore NOT

I restanti definiscono gli operatori AND e OR

## TEOREMI A UNA VARIABILE

	Theorem	Dual	Name
T1	$B \bullet 1 = B$	$T1'$ $B + 0 = B$	Identity
T2	$B \bullet 0 = 0$	$T2'$ $B + 1 = 1$	Null Element
T3	$B \bullet B = B$	$T3'$ $B + B = B$	Idempotency
T4		$\overline{\overline{B}} = B$	Involution
T5	$B \bullet \overline{B} = 0$	$T5'$ $B + \overline{B} = 1$	Complements

## TEOREMI A PIU' VARIABILI

Number	Theorem	Name
T6	$B \bullet C = C \bullet B$	Commutativity
T7	$(B \bullet C) \bullet D = B \bullet (C \bullet D)$	Associativity
T8	$B \bullet (C + D) = (B \bullet C) + (B \bullet D)$	Distributivity
T9	$B \bullet (B + C) = B$	Covering
T10	$(B \bullet C) + (B \bullet \bar{C}) = B$	Combining
T11	$(B \bullet C) + (\bar{B} \bullet D) + (C \bullet D) = (B \bullet C) + (\bar{B} \bullet D)$	Consensus

OSS

Per il PRINCIPIO DI DUALITA', i teoremi sono analoghi ai teoremi stessi scambiando  $+$  con  $\bullet$  e  $1$  con  $0$

## DIMOSTRAZIONI DEI TEOREMI

Per dimostrare questi teoremi possiamo usufruire di 3 tecniche dimostrative diverse:

### PERFECT INDUCTION

Studiamo le tabelle di verita' e verifichiamo che coincidono

### ASSIOMI E TEOREMI

Possiamo sfruttare assiomi e teoremi precedentemente dimostrati

### CASI RILEVANTI

Sfruttando la struttura delle due tabelle, andiamo a verificare solo i casi rilevanti

## TEOREMA DI DE MORGAN

La **NEGATA** di un **PRODOTTO** è uguale alla **SOMMA** delle negate (e viceversa)

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

Il Teorema di De Morgan ci consente quindi di convertire una porta **OR** in una porta **AND**

## ALTRI TEOREMI

- Sia  $E$  un' espressione booleana che contiene una sott'espressione  $E_1$ . Se  $E_1=E_2$ , allora  $E=E'$  dove  $E'$  è ottenuto sostituendo a  $E_1, E_2$ .

$E_1$

$$BC + \overline{BD} + CD = BC + \overline{BD}$$

$$(BC + BD + CD)(FD + A) = (BC + BD)(FD + A)$$

- Sia  $E_1=E_2$  allora  $E'_1=E'_2$  dove questi sono ottenuti sostituendo una variabile booleana  $A$  con qualsiasi espressione  $E$

$E_1$ .

$$\overline{AB} + AB = A$$

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

- Se  $E = E'$  allora  $\bar{E} = \bar{E}'$

### DE MORGAN E DUALITÀ

Il **PRINCIPIO DI DUALITÀ** è una diretta conseguenza dei teoremi di **DE MORGAN**

Ese.

Dimostriamo che  $BC = CB \Rightarrow B+C = C+B$

$$BC = CB \rightarrow \overline{BC} = \overline{CB} \rightarrow \overline{B}\overline{C} = \overline{C}\overline{B} \rightarrow \overline{\overline{B}+\overline{C}} = \overline{\overline{C}+\overline{B}} \rightarrow B+C = C+B$$

### DE MORGAN E FORME POS/SOP

Grazie ai teoremi di De Morgan possiamo passare da una generica espressione booleana in forma SOP/POS :

① Applichiamo De Morgan per "spingere" la negazione della struttura della formula

② Se voglio una forma SOP applico la **DISTRIBUTIVITÀ** dell' AND sull' OR

Se voglio una forma POS applico la **DISTRIBUTIVITÀ** dell' OR sull' AND

E.

$$Y = \overline{(\overline{A}C\overline{E} + \overline{D}) + B} = (\overline{\overline{A}C\overline{E}} + \overline{D}) \cdot \overline{B} = (\overline{\overline{A}C} \cdot \overline{\overline{E}}) \cdot \overline{B} = ((\overline{\overline{A}\overline{C}} + \overline{\overline{E}}) \cdot D) \cdot \overline{B}$$

$$= ((AC + \overline{E}) \cdot D) \cdot \overline{B} = (ACD + \overline{ED}) \cdot \overline{B} = ACD\overline{B} + \overline{ED}\overline{B}$$

SOP  $\Leftarrow$  POS

Sempre grazie ai teoremi di De Morgan possiamo ricavare dalla forma SOP la corrispettiva forma POS (e viceversa)

E.

Possiamo dalla forma SOP alla forma POS

A	B	Y	$\overline{Y}$	$\overline{A}\overline{B}$	$\overline{A}B$
0	0	0	1	1	0
0	1	0	1	0	1
1	0	0	1	0	0
1	1	1	0	0	0

① Nego l' output

② li ricaviamo la forma SOP della negata

③ Nego ulteriormente la forma SOP

④ Applico De Morgan

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

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

COMPLETEZZA DEGLI OPERATORI

Abbiamo visto come ogni funzione booleana puo' essere espressa tramite forma SOP/POS con l'utilizzo delle porte OR, AND e NOT.

L'insieme  $\{+, \cdot, ?\}$  è detto un **SET COMPLETO**

Definiremo **SET MINIMALE** un insieme di operazioni che non contiene "sottinsiemi" di operazioni.

De Morgan mostra come attraverso  $\{+, ?\}$  possiamo ottenere  $\cdot$ , quindi l'insieme  $\{+, \cdot, ?\}$  non è **MINIMALE** mentre  $\{+, ?\}$  e  $\{\cdot, ?\}$  sì

### OPERATORI INSIEMISTICI

●  $T = \text{INSIEME UNIVERSO}$

●  $Z^T = \{A \mid A \subseteq T\}$  INSIEME DELLA PARTI DI  $T$

●  $\cap = \text{AND}$

●  $\cup = \text{OR}$

●  $\neg T = \text{NOT}$

●  $1 = T$

●  $0 = \emptyset$

●  $\sim = \text{COMPLEMENTO}$

## SEMPLIFICAZIONE DELLE ESPRESSIONI

Semplificiamo le espressioni per creare circuiti più economici

E.

$$Y = A(AB + ABC)$$

DISTRIBUTIVITÀ

$$= A(AB(1+C))$$

NULL ELEMENT

$$= A(AB(1))$$

IDENTITÀ

$$= A(AB)$$

ASSOCIAZIVITÀ

$$= (AA)B$$

IDEMPOTENZA

$$= AB$$

## TEOREMI COMUNI PER SEMPLIFICARE

● DISTRIBUTIVITÀ

● COVERING  $(A+AP = A)$

● COMBINING  $(PA + P\bar{A} = P)$

● EXPANSION  $(\text{inverso di covering e combining})$

● DUPLICATION  $(A = A+A)$

● "SIMPLIFICATION"  $(P\bar{A} + A = P+A \quad / \quad PA + \bar{A} = P+\bar{A})$

## SCHEMI CIRCUITALI SOP

Per disegnare uno schema circuitale SOP basta:

- Disegnare una linea di input per ogni variabile positiva

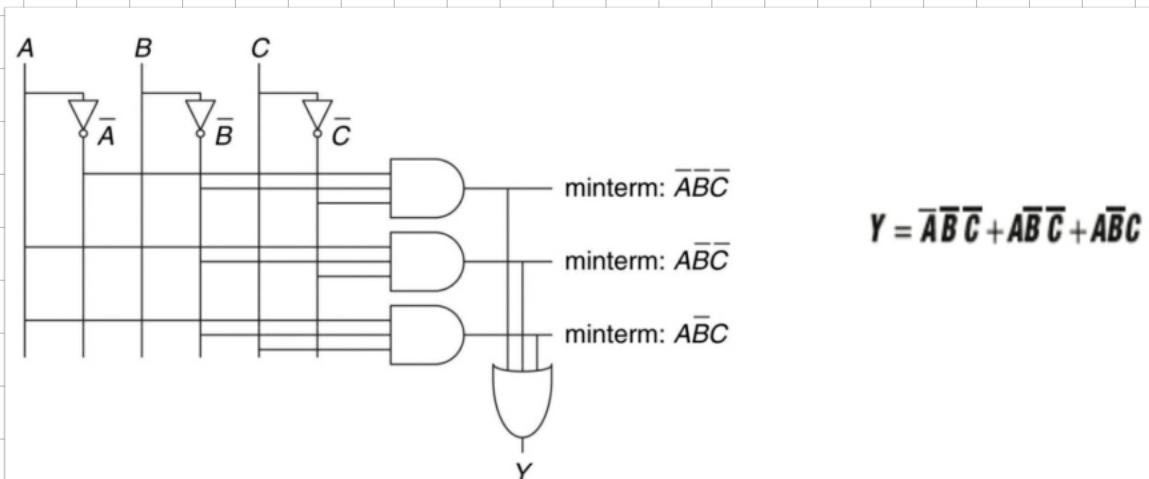
Aggiungere una linea con un not per ogni variabile che appare negativa

- Per ogni mintermine disegnare una porta AND e ovviamente collegare i rispettivi littorali

- Collegare tutte le uscite delle porte AND ad un unico OR

### OSS

Per questo la forma SOP viene detta logica a due livelli AND-OR



## SEMPLIFICARE LE FORME SOP

Per semplificare una forma SOP possiamo applicare il **COMBINING**

$(P\bar{B} + P{B} = P)$  ad ogni **IMPONENTE**

Un implicant è detto **PRIMO** se non può essere combinato con altri implicanti per ridurne i letterali

Un'espressione SOP è detta **MINIMALE** se tutti i suoi letterali sono primi (non è possibile **MINIMIZZARE**)

Ese.

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

Abbiamo 2 implicanti e 3 letterali, proviamo diversamente:

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

Abbiamo 2 implicanti e 2 letterali

### SEMPLIFICARE FORME POS

Per semplificare una forma POS possiamo applicare il **COMBINING**

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

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

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

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

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

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

## CIRCUITO A PRIORITÀ

Sono circuiti nei quali viene assegnata una risorsa condivisa secondo un ordine di priorità.

Gli **INPUT** indicano le **RICHIESTE DI RISORSA**

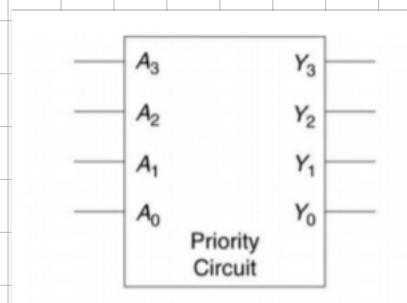
Gli **OUTPUT** indicano gli **ASSEGNOTARI** della risorsa, questi:

sono 1 se la risorsa è stata assegnata a quell' output, altrimenti 0

Ese.

Priorità:  $3 > 2 > 1 > 0$

$A_3$	$A_2$	$A_1$	$A_0$	$Y_3$	$Y_2$	$Y_1$	$Y_0$
0	0	0	0	0	0	0	0
0	0	0	1	0	0	0	1
0	0	1	0	0	0	1	0
0	0	1	1	0	0	1	0
0	1	0	0	0	1	0	0
0	1	0	1	0	1	0	0
0	1	1	0	0	1	0	0
0	1	1	1	0	1	0	0
1	0	0	0	1	0	0	0
1	0	0	1	1	0	0	0
1	0	1	0	1	0	0	0
1	0	1	1	1	0	0	0
1	1	0	0	1	0	0	0
1	1	0	1	1	0	0	0
1	1	1	0	1	0	0	0
1	1	1	1	1	0	0	0



## DON'T CARES

Possiamo semplificare la tabella conoscendo la priorità:

$A_3$	$A_2$	$A_1$	$A_0$	$Y_3$	$Y_2$	$Y_1$	$Y_0$
0	0	0	0	0	0	0	0
0	0	0	1	0	0	0	1
0	0	1	X	0	0	1	0
0	1	X	X	0	1	0	0
1	X	X	X	1	0	0	0

## ESERCIZI

①  $A \oplus A = 0$

$$A \oplus A = A\bar{A} + \bar{A}A = A(\bar{A} + \bar{A}) = A(\overline{A+A}) = A\bar{A} = 0$$

COMPLEMENTO

●  $A \oplus 0 = a$

$$A \oplus 0 = A \cdot 1 + \bar{A} \cdot 0 = 1$$

●  $A \oplus 1 = \sim A$

$$A \oplus 1 = A \cdot 0 + \bar{A} \cdot 1 = \bar{A} = \sim A$$

●  $A \oplus \sim A = 1$

$$A \oplus \bar{A} = A\bar{A} + \bar{A}\bar{A} = A + \bar{A} = 1$$

●  $A \oplus B = B \oplus A$

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

COMMUTATIVITÀ

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

(2.1)

a)  $\bar{A}\bar{B} + A\bar{B} + AB$

b)  $\bar{A}\bar{B}\bar{C} + ABC$

c)  $\bar{A}\bar{B}\bar{C} + \bar{A}B\bar{C} + A\bar{B}\bar{C} + A\bar{B}C + ABC$

d)  $\bar{A}\bar{B}\bar{C}\bar{D} + \bar{A}\bar{B}\bar{C}D + \bar{A}\bar{B}C\bar{D} + \bar{A}\bar{B}CD + A\bar{B}\bar{C}\bar{D} + A\bar{B}\bar{C}D + ABC\bar{D}$

e)  $\bar{A}\bar{B}\bar{C}\bar{D} + \bar{A}\bar{B}CD + \bar{A}B\bar{C}\bar{D} + \bar{A}B\bar{C}D + A\bar{B}\bar{C}\bar{D} + A\bar{B}\bar{C}D + AB\bar{C}\bar{D} + ABC\bar{D}$

(12) a)  $\bar{A}B + A\bar{B} + AB$

b)  $\bar{A}\bar{B}C + \bar{A}B\bar{C} + \bar{A}BC + A\bar{B}\bar{C} + AB\bar{C}$

c)  $\bar{A}\bar{B}C + AB\bar{C} + ABC$

d)  $\bar{A}\bar{B}\bar{C}\bar{D} + \bar{A}\bar{B}C\bar{D} + \bar{A}\bar{B}CD + \bar{A}B\bar{C}\bar{D} + \bar{A}BC\bar{D} + A\bar{B}\bar{C}\bar{D} + A\bar{B}C\bar{D}$

e)  $\bar{A}\bar{B}CD + \bar{A}B\bar{C}\bar{D} + \bar{A}BC\bar{D} + A\bar{B}\bar{C}\bar{D} + A\bar{B}\bar{C}D + A\bar{B}\bar{C}\bar{D} + A\bar{B}CD$

(2.3) a)  $A + \bar{B}$

b)  $(A+B+\bar{C})(A+\bar{B}+C)(A+\bar{B}+\bar{C})(\bar{A}+B+C)(\bar{A}+B+\bar{C})(\bar{A}+\bar{B}+C)$

c)  $(A+B+\bar{C})(A+\bar{B}+\bar{C})(\bar{A}+\bar{B}+C)$

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

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

(2.4) a)  $A + B$

b)  $(A+B+C)(\bar{A}+B+\bar{C})(\bar{A}+\bar{B}+\bar{C})$

c)  $(A+B+C)(A+\bar{B}+C)(A+\bar{B}+\bar{C})(\bar{A}+B+C)(\bar{A}+\bar{B}+\bar{C})$

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

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

$$\textcircled{3} \quad A \oplus B = A\bar{B} + \bar{A}B = \overline{\overline{A}\bar{B} + \bar{A}B} = (\bar{A}B)(A\bar{B}) = (\bar{A} + \bar{B})(B + A)(\bar{A}\bar{B}) = \\ (\bar{A} + \bar{A})(\bar{B})(B + B)(A) = (\bar{A} + \bar{B})(\bar{A} + \bar{B})(B + A)(B + A) = (\bar{A} + \bar{B})(A + B)$$

④ Dobbiamo dimostrare che con la porta NAND possiamo esprimere le porte AND, NOT e OR.

$$A \text{NAND } A = \text{NOT } A$$

$$\overline{AA} = \bar{A} \quad A \rightarrow \boxed{D} \circ -\bar{A}$$

$$A \text{NAND } B = \text{NOT}(A \text{ AND } B)$$

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

$$\begin{matrix} A \\ B \end{matrix} \rightarrow \boxed{D} \circ \rightarrow \boxed{D} \circ \begin{bmatrix} A \\ B \end{bmatrix}$$



## VANTAGGI DELLE SOP/POS

Il grande vantaggio delle porte SOP, POS è dato dal bassissimo **DELAY** (tempo che intercorre tra tra la prima variabile che cambia valore e l'ultima variabile che cambia valore)

Questo è dovuto al fatto che si sviluppa solo su due livelli.

## LIMITI DELLE FORME SOP/POS

Alcune funzioni booleane in forma SOP richiedono numerivoli porte:

Se il numero di 0 e di 1 è lo stesso avremo  $2^{m-1}$  mintermini / maxtermini Es.

Consideriamo ad esempio uno XOR a più variabili

XOR8 richiede 128 AND8 ( $2^{8-1}$ ) e un OR128 (un ingresso per ogni AND)

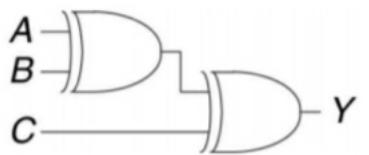
## LOGICHE MULTILIVELLO

Per ridurre il numero di porte logiche possiamo ricorrere ad una

### LOGICA MULTILIVELLO

E.

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



### BUBBLE PUSHING

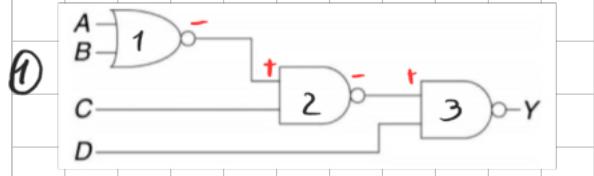
Circuiti che sfruttano la logica multilivello risultano essere ricche di negazioni annidate, questo ci complica il lavoro di deduzione dell'espressione booleana dato il circuito.

Possiamo però applicare De Morgan alle varie porte, tramite il

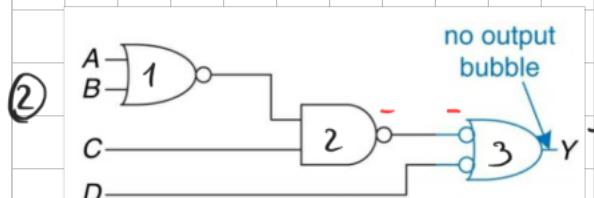
### BUBBLE PUSHING :

Partendo dall'OUTPUT si applicano inversamente le leggi di De Morgan in modo che INPUT e OUTPUT di ogni modo siano concordi tra loro.

E.s.



Applico De Morgan su 3 ( $\overline{xy} = \overline{x} + \overline{y}$ )

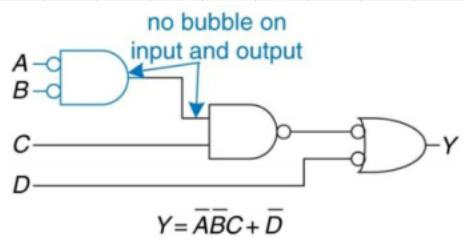


Ora OUT2 e' concorde a IN3

Tuttavia IN2 e OUT1 sono discordi

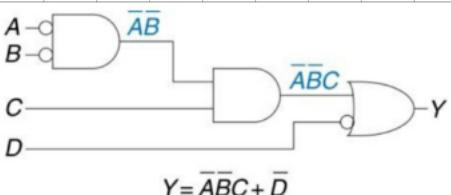
ma non posso applicare De Morgan a 2 (prenderei la concordanza con 2), quindi lo faccio su 1

③



A questo punto possiamo semplificare le doppie negazioni ( $\bar{\bar{x}}=x$ ) e scrivere la SOP

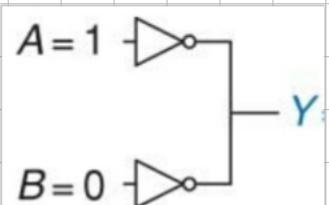
④



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

### VALORI ILLEGALI

Prendiamo in considerazione il seguente circuito:



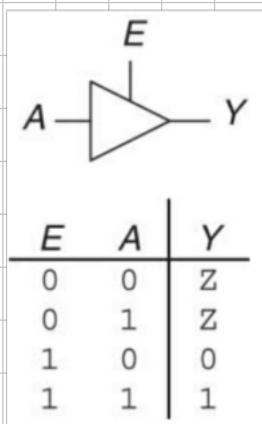
L'output Y dovrebbe essere al tempo stesso 1 e 0.

Da un punto di vista fisico abbiamo una dissipazione di energia che porta alla fusione del circuito. Questa configurazione è detta **ILLEGALE**

### TRISTATE BUFFER

Un modo oltre ad assumere i valori 0/1 può entrare in uno stato **Z** (alta impedenza), in cui non vi è passaggio di corrente.

Per fare ciò usiamo i **TRISTATE BUFFER**, utilizzati principalmente per isolare una parte di circuito dal resto.



$E=0 \Rightarrow$  circuito **APERTO** (non passa corrente)

$E=1 \Rightarrow$  circuito **CHIUSO** (passa la corrente d. A)

E è detto **ENABLER**

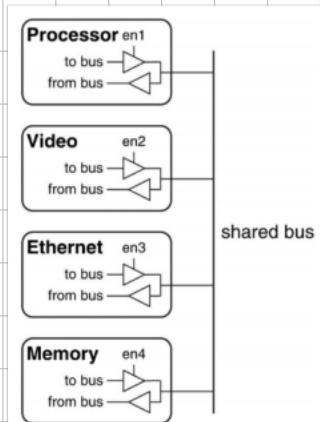
## BUS CONDIVISI

I Tristate Buffers vengono usati nei **BUS** che connettono i vari chip

Ovviamente il **BUS** può ricevere un solo input per evitare stati

illegali, quindi i componenti possono 1 solo segnale alla volta.

Gli altri componenti sono **DISCONNESSI** con un Tristate Buffer



## MAPPE DI KARNAUGH

Le MAPPE DI KARNAUGH sono un metodo grafico di semplificazione di una SOP.

Tecnicamente non introducono nuove tecniche, sfrutta il principio

$$P = P_A + P_{\bar{A}}$$

A	B	C	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

Disegniamo la K-map di quest'espressione booleana mettendolo sulle righe i due possibili valori di C, e nelle colonne avremo le possibili coppie AB

Y	AB	GREY CODE			
		00	01	11	10
C	0	1	0	0	0
	1	1	0	0	0

Gli input di AB NON sono in ordine crescente in quanto sono disposti in modo tale che non cambino assieme entrambi i valori

Y	AB	GREY CODE			
		00	01	11	10
C	0	$\bar{ABC}$	$\bar{ABC}$	$ABC$	$ABC$
	1	$\bar{ABC}$	$\bar{ABC}$	$ABC$	$\bar{ABC}$

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

Ad ogni input ormai corrisponde un mintermone, in particolare, gli 1 che appartengono alla stessa BOLLA saranno sicuramente un mintermone in comune (utile per la semplificazione)

## MINIMIZZAZIONE IN K-MAPS

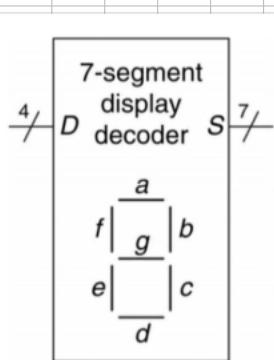
- Usare il minimo numero di **BOLLE** per ricoprire tutti gli 1
- Tutte le caselle sotto a una bolla devono contenere un 1
- Ogni bolla deve essere il più grande possibile
- Una bolla può "trasbordare" i bordi dello schema (forma **TOROIDALE**)
- Una casella può essere inclusa in più bolle se permette un numero inferiore di bolla
- Ogni bolla copre  $2^n$  caselle

## OSS

Più grande è una **BOLLA**, più piccola sarà il **LITERALE**, perché posso mettere in evidenza

## DISPLAY A 7 SEGMENTI

E' un dispositivo a 4 ingressi e 7 uscite



Con questo dispositivo possiamo mostrare dei numeri.

Ad esempio per lo 0 avremo in INPUT 0000 e in OUTPUT 1 per ogni lettera tranne g=0

(13)

$$a) Y = AC + \overline{A}\overline{B}C = C(A + \overline{A}\overline{B}) = C(A + A\overline{B} + \overline{A}\overline{B}) = C(A + \overline{B}(A + \overline{A})) = C(A + \overline{B})$$

$$b) Y = \overline{A}\overline{B} + \overline{A}BC + \overline{(A+C)} = \overline{A}(\overline{B} + BC) + (\overline{A}C) = \overline{A}(\overline{B} + B(C + C)) = \overline{A}(\overline{B} + C)$$

$$c) Y = \overline{A}\overline{B}\overline{C}\overline{D} + A\overline{B}\overline{C} + A\overline{B}C\overline{D} + ABD + \overline{A}\overline{B}C\overline{D} + B\overline{C}D + \overline{A}$$

$E + EF = E$  POSSIAMO APPLICARLO AD  $\overline{A}$

$$= \overline{A} + A\overline{B}\overline{C} + A\overline{B}C\overline{D} + ABD + B\overline{C}D \quad \text{ESPANSIONE DI } \overline{A}$$

$$= \underline{\overline{A} + \overline{A}\overline{B}\overline{C}} + A\overline{B}\overline{C} + A\overline{B}C\overline{D} + ABD + B\overline{C}D \quad \text{RACCOLGO } \overline{B}\overline{C}$$

$$= \overline{A} + \overline{B}\overline{C} + A\overline{B}C\overline{D} + ABD + B\overline{C}D$$

(14)

$$a) Y = \overline{ABC} + \overline{ABC} = \overline{AB}$$

$$b) Y = \overline{\overline{ABC} + A\overline{B}} = \overline{A + \overline{B} + \overline{C} + A\overline{B}} = \overline{A + \overline{B} + \overline{C}} = \overline{ABC}$$

$$c) Y = ABC\overline{D} + \overline{ABC}D + (\overline{A + B + C + D})$$

$$Y = ABC\overline{D} + \overline{ABC}D(A + B + C + D) = ABC\overline{D} + \overline{ABC}D = ABC\overline{D} + \overline{ABC} + \overline{D}$$

$$= \overline{D}(ABC) + \overline{ABC} = \overline{D} + \overline{ABC} = \overline{ABC}D$$

SIMPLIFICATION

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

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

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

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

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

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

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

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







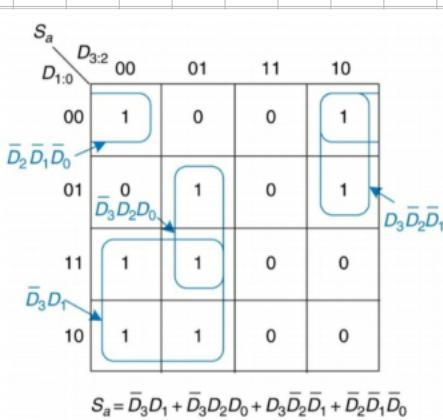
$D_{3:0}$	$S_a$	$S_b$	$S_c$	$S_d$	$S_e$	$S_f$	$S_g$
0000	1	1	1	1	1	1	0
0001	0	1	1	0	0	0	0
0010	1	1	0	1	1	0	1
0011	1	1	1	1	0	0	1
0100	0	1	1	0	0	1	1
0101	1	0	1	1	0	1	1
0110	1	0	1	1	1	1	1
0111	1	1	1	0	0	0	0
1000	1	1	1	1	1	1	1
1001	1	1	1	0	0	1	1
others	0	/0	0	0	0	0	0

OSS

Per rappresentare 10 cifre sul display, I 4 segnali di input risultano essere **RIDONDANTI**

Possiamo rappresentare le configurazioni su un K-map.

Vediamo ad esempio Sa



## DON'T CARE IN K-MAPS

Anche nelle k-maps abbiamo dei valori che non influiscono sulla minimizzazione dell'espressione.

A questi valori X possiamo assegnare i valori che più ci fanno comodo per minimizzazione

E.s.

- Nel display a 7 segmenti abbiamo 5 configurazioni don't care (quelle che rappresenterebbero i numeri da 10 a 15)

$S_a$	$D_{3:2}$	00	01	11	10
$D_{1:0}$	00	1	0	X	1
01	0	1	X	1	
11	1	1	X	X	
10	1	1	X	X	

$S_a = D_3 + D_2 D_0 + \bar{D}_2 \bar{D}_0 + D_1$

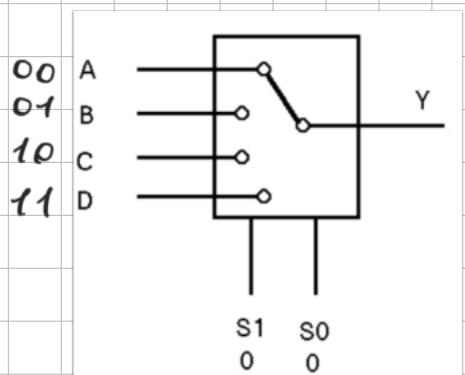
AB	00	01	11	10
CD	00	0	0	1
00	1	0	0	1
01	0	1	1	1
11	0	1	1	1
10	1	0	0	1

Scriviamo i littorali costanti in ogni bolla

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

## MULTIPLEXER

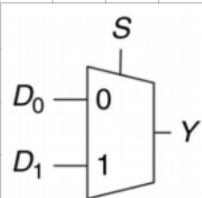
Sono dei selettori di linea



In generale è costituito da  $2^n$  ingressi, 1 uscita e  $\log_2 2^m$  linee di selezione che indicano a quale ingresso deve corrispondere l'output.

$S=00$  chiama A,  $S=01$  chiama B e così via

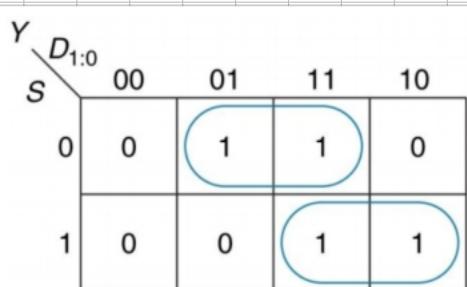
Può essere rappresentato attraverso K-maps



S	D <sub>1</sub>	D <sub>0</sub>	Y
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	0
1	1	0	1
1	1	1	1

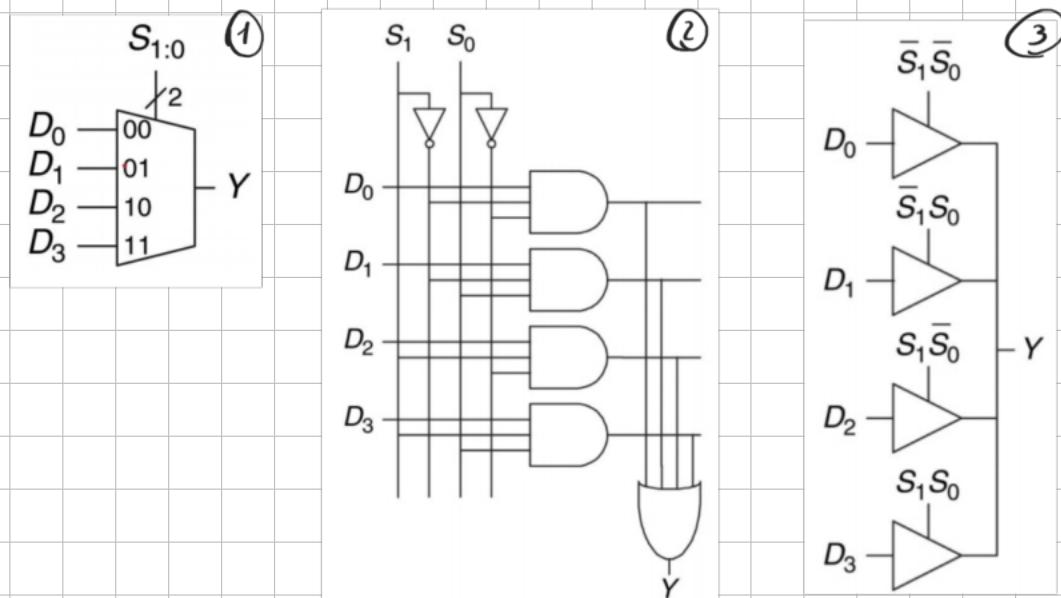
$S=0$  chiama D<sub>0</sub>

$S=1$  chiama D<sub>1</sub>

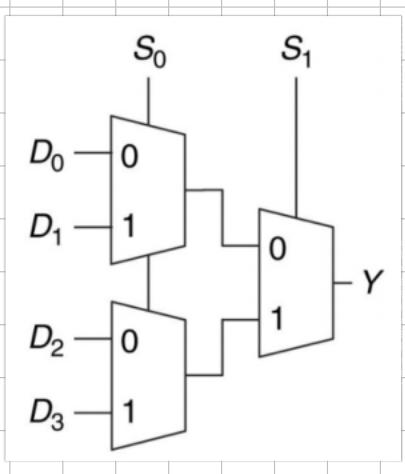


$$Y = D_0 \bar{S} + D_1 S$$

## MULTIPLEXER 4:1



Come vediamo da ①, Se  $S_0 = 0$  e  $S_1 = 0$  (00) avremo l'output di  $D_0$ , Se  $S_0 = 0$  e  $S_1 = 1$  (01) avremo l'output di  $D_1$  e così via. Si può verificare che se sostituisco 0 e 1 nel circuito ② avremo tutti 0 tranne la porta indicata da  $S_0$  e  $S_1$  ( $00 = D_0$ ,  $01 = D_1 \dots$ ). Analogamente in ③ ogni tristate gestisce singolarmente l'input.



Un altro modo ancora

può essere quello di collegare 2 multiplex 2:1 a cascata. Il risultato va letto partendo da Y (come se  $S_0$  e  $S_1$  fossero scambiati)

## FUNZIONI BOOLEANE e MUX

Possiamo quindi utilizzare i Multiplex per sintetizzare funzioni booleane.

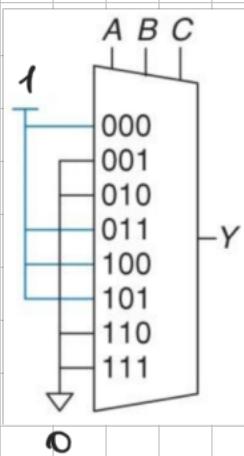
Per sintetizzare una funzione di  $n$  variabili avremo un mux di  $2^n$  righe.

Le variabili saranno i **SELETTORI**, l'output darà una determinata configurazione, ripeterà la tabella di verità.

E.s.

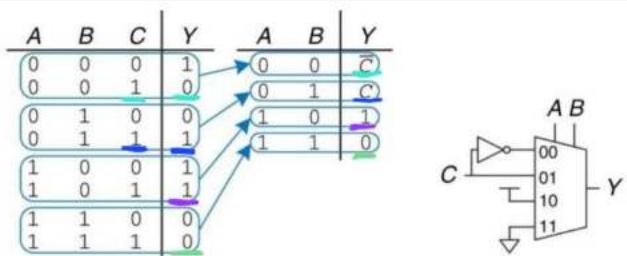
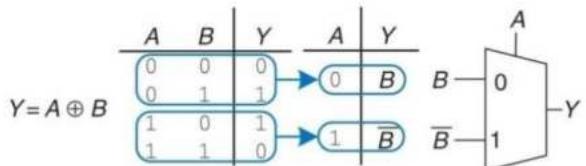
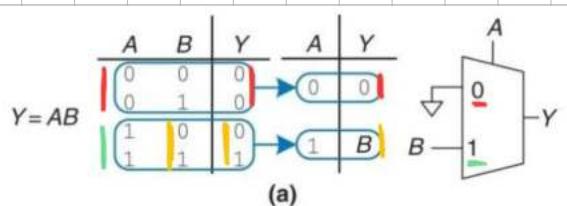
Prenotiamo  $Y = A\bar{B} + \bar{B}\bar{C} + \bar{A}\bar{B}C$ . Avremo  $2^3$  righe

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



Possiamo in alternativa usare un mux a  $2^{n-1}$  ingressi le prime  $n-1$  variabili saranno i selettori, le **LINEE DI INGRESSO** possono essere 0,1 oppure l'ultima variabile positiva e negata ( $A, \bar{A}$ )

E.



OSS

Nel mux, la freccia  $\nabla$  indica GROUND

di potenza, con valore logico 0, mentre invece  $\top$  indica il  $V_{DD}$  con valore logico 1.

## SINTETIZZARE IN MUX 4:1

Abbiamo questa tavola di verità da sintetizzare in un mux 4:1

ABCD	Y
0000	0
0001	0
0010	0
0011	1
0100	1
0101	1
0110	0
0111	0
1000	1
1001	1
1010	0
1011	0
1100	1
1101	1
1110	1
1111	1

quel è l'espressione booleana  
associata all'output

Il nostro obiettivo è capire per ogni valori di AB,

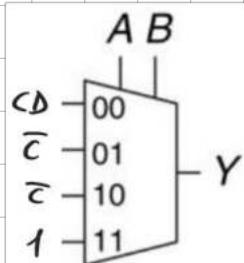
ABCD	Y
0000	0
0001	0
0010	0
0011	1
0100	1
0101	1
0110	0
0111	0
1000	1
1001	1
1010	0
1011	0
1100	1
1101	1
1110	1
1111	1

$$= CD$$

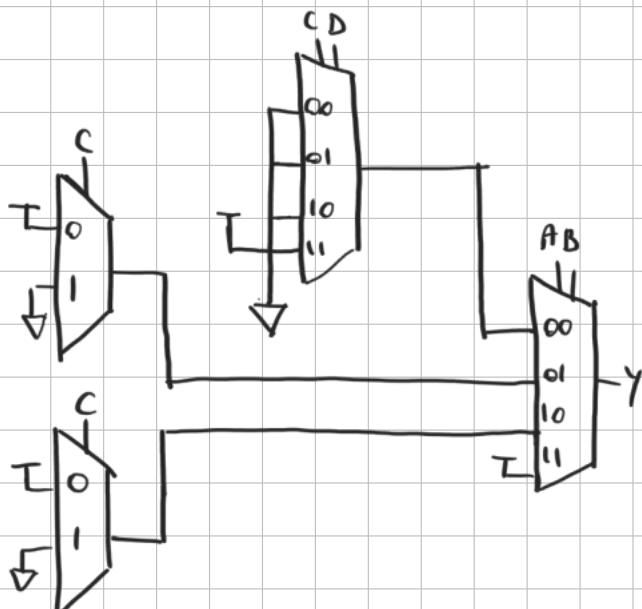
$$= \bar{C}$$

$$= \bar{C}$$

$$= 1$$



Trasformiamo questo mux in un mux 4:1

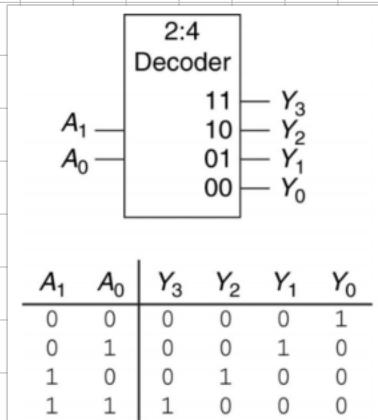


## DECODER

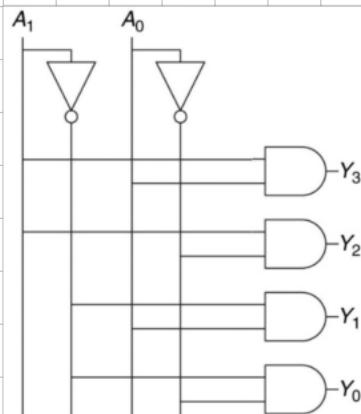
E' un dispositivo a N linee d'ingresso e  $2^m$  linee d' uscita.

Le configurazioni di input di un decoder sono dette **INDIRIZZI**.

A seconda dell' indirizzo si attiverà un determinato output

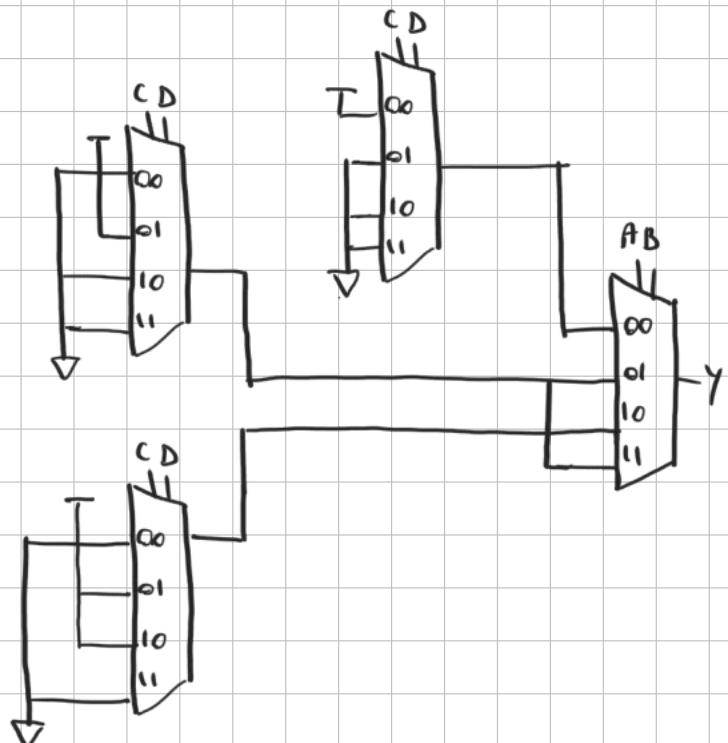
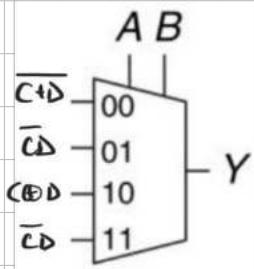


Gli output saranno tutti 0 tranne l' output individuato dall' input

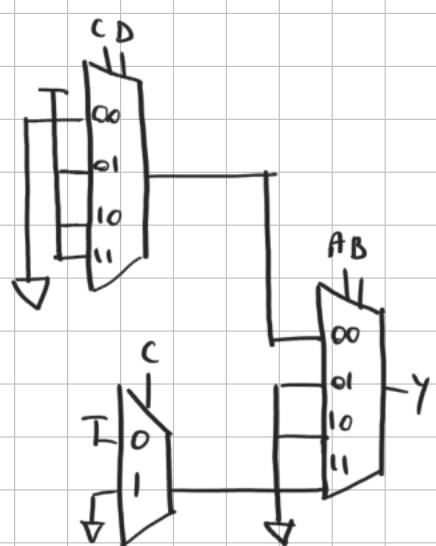
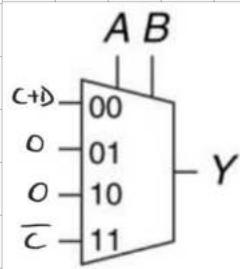


# ESERCIZI

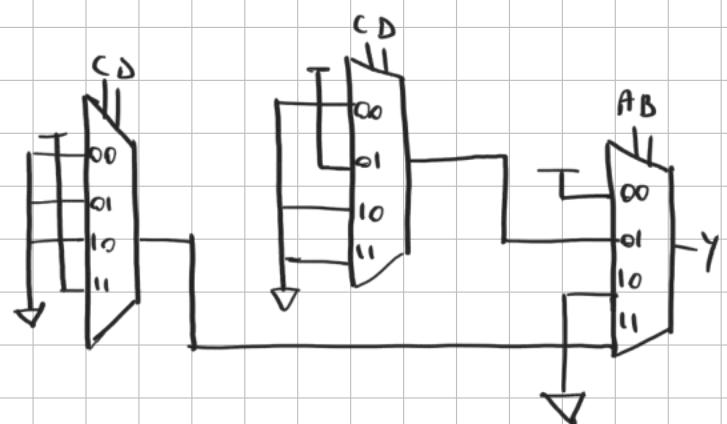
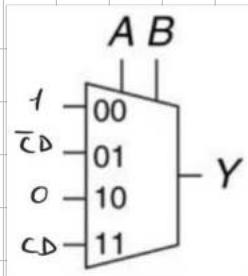
ABCD	Y
0000	1
0001	0
0010	0
0011	0
0100	0
0101	1
0110	0
0111	0
1000	0
1001	1
1010	1
1011	0
1100	0
1101	1
1110	0
1111	0



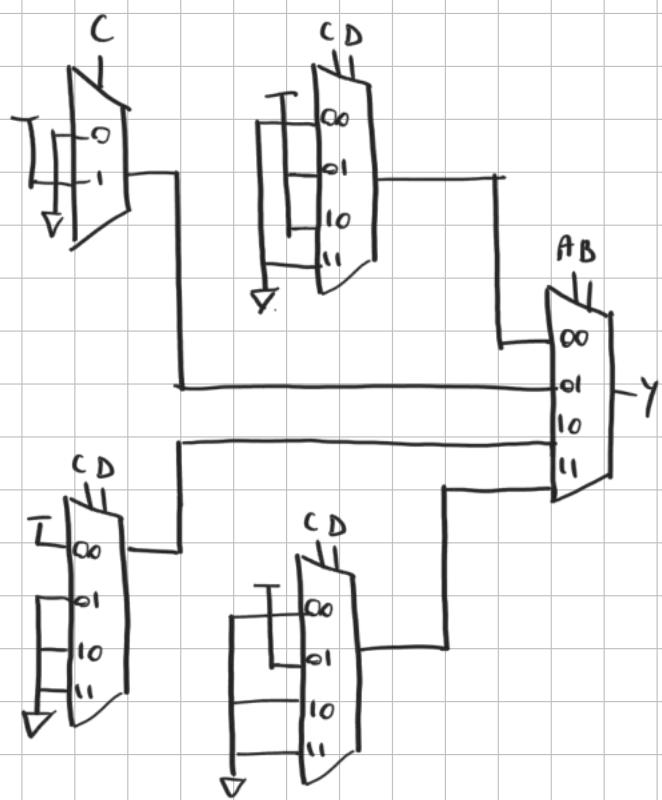
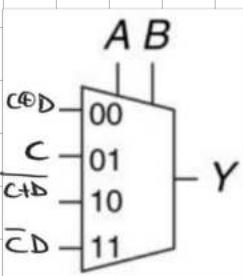
ABCD	Y
0000	0
0001	1
0010	1
0011	1
0100	0
0101	0
0110	0
0111	0
1000	0
1001	0
1010	0
1011	0
1100	1
1101	1
1110	0
1111	0



ABCD	Y
0000	1
0001	1
0010	1
0011	1
0100	0
0101	1
0110	0
0111	0
1000	0
1001	0
1010	0
1011	0
1100	0
1101	0
1110	0
1111	1



ABCD	Y
0000	0
0001	1
0010	1
0011	0
0100	0
0101	0
0110	1
0111	1
1000	1
1001	0
1010	0
1011	0
1100	0
1101	1
1110	0
1111	0







## ESE'RCI 121

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

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

AB	CD	00	01	11	10
00	X			X	
01	1	1	X	X	
11	1			1	
10	X			1	

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

AB	CD	00	01	11	10
00	X			X	
01	1		X	X	
11	1			1	
10	1	X		1	

$$Y = \bar{B}$$

- $(A+C)(\bar{A}+B)(B+C) = (A+C)(\bar{A}+B)$  CONSENSUS

- Indicare il numero di funzioni booleane di 4 variabili tali che  $f(0, B, C, D) = 0$

$2^8$  ( $8$  output "don't cares")

- $f(A, B, C, D) = 1$  con  $A > D$

$2^{16}/2^4$  ( $4$  casi limitanti)

- Ridurre in forma minima SOP l'espressione:

$$\overline{(AC)} \oplus (B+D) + A\bar{C}\bar{D}$$

RICORDA

$$E \oplus F = E\bar{F} + \bar{E}F$$

$$\overline{E \oplus F} = \overline{E}\bar{F} + \overline{\bar{E}}F$$

$$(AC)(B+D) + \overline{(AC)}(\overline{B+D}) + A\bar{C}\bar{D} = (AC)(B+D) + (\overline{A+C})(\overline{B}\bar{D}) + A\bar{C}\bar{D}$$

$$= \underline{AB\bar{C}} + \underline{A\bar{C}D} + \underline{\overline{A}\overline{B}\bar{D}} + \underline{\overline{B}C\bar{D}} + \underline{A\bar{C}\bar{D}}$$
 A questo punto facciamo la K-map

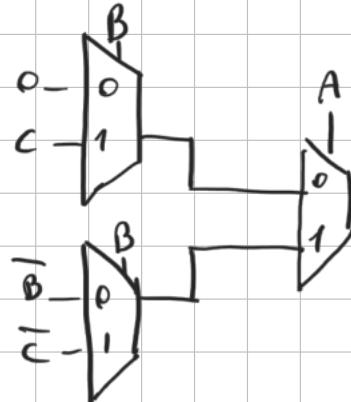
	AB	CD	00	01	11	10
CD	00	1	0	1	1	
00	0	0	1	1		
01	0	0	0	0		
11	1	0	0	1		
10	1	0	0	1		

$$= \overline{B}\bar{D} + A\bar{C} \quad \text{forma minima SOP}$$

- Usando Multiplexer 2:1 e i segnali di ingresso  $A, B, C, \bar{A}, \bar{B}, \bar{C}, 0, 1$  disegnare il circuito di  $(A \oplus C)(A + B)$

Fixo A (abbiamo 3 variabili, con multiplexer 2:1)

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



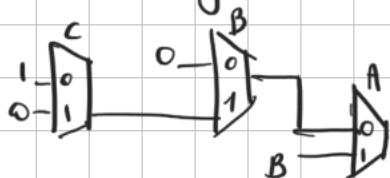
- Ingresso  $A, B, C, 0, 1$ , 2 MUX 2:1,  $\bar{B} + \bar{C}$  + AB

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

ATTENZIONE all'ordine in cui sceglieremo i selettori

$\bar{B} + \bar{C}$  Non avendo l'ingresso di  $\bar{C}$  dovremo costruirici il mux per  $\bar{C}$

Potremmo svolgerlo così:



Ma abbiamo solo 2 MUX

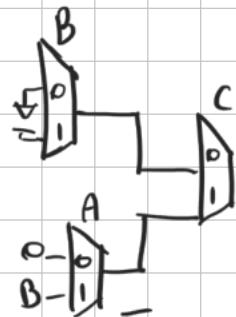
Proviamo a cambiare i selettori, studiando la tavola di verità

Vediamo ad esempio al variare di C come varia l'output

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

Per  $C=0, Y=B$

Per  $C=1, Y=AB$



↗ In questo caso il selettore è  
indifferente

## CIRCUITO SEQUENZIALE

Ricordiamo che in un CIRCUITO SEQUENZIALE l' output tiene conto della STORIA (valori precedenti) della "memoria" del sistema. Quest' informazione è rappresentata dallo STATO INTERNO, necessario per gli output futuri.

Lo stato viene salvato in componenti dette LATCHES e FLIP-FLOPS.

I circuiti sequenziali (SINCRONO) hanno due proprietà:

- LOGICA COMBINATORIA
- REGISTRI (banchi di flip-flops) che memorizzano gli stati interni

Ma l' aspetto più importante è la RETROAZIONE del circuito, ovvero i segnali da OUTPUT sono riportati in INPUT

## STATE ELEMENTS

Sono detti STATE ELEMENTS le componenti circolari utilizzate per memorizzare lo STATO del circuito.

Si suddividono in:

- CIRCUITI BISTABILI
- SR LATCH

## ● D LATCH

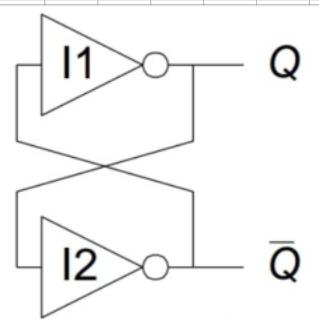
## ● D FLIP-FLOP

### CIRCUITO BISTABILE

Il CIRCUITO BISTABILE è costituito da 2 porte NOT retroazionate (l'output di una è l'input dell'altra).

Ha 2 caratteristiche:

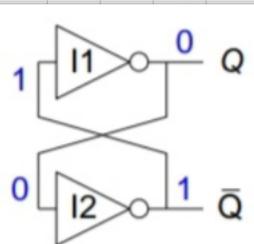
- Ci sono sempre due output
- Non ci sono input



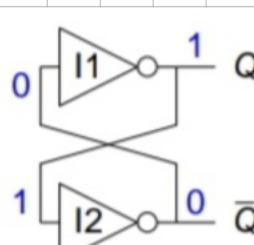
E' detto BISTABILE perché i due output saranno sempre 0 e 1 (o viceversa)

I due possibili casi sono:

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



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



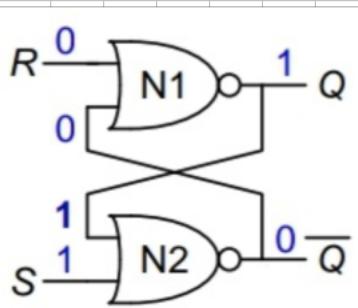
I valori  $L1$  e  $L2$  sono fissati all'avvio della macchina e **NON** possono essere modificati, questo rappresenta una limitazione.

### SR LATCH

E' un circuito composto da 2 porte **NOR** innestate con 2 input (**S** ed **R**) e 2 output.

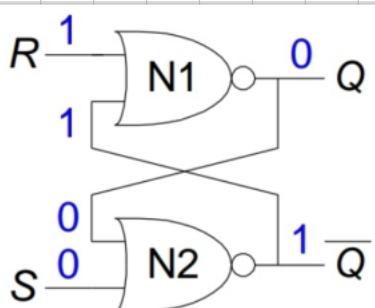
I 4 possibili casi sono:

- $S=1, R=0 \rightarrow Q=1, \bar{Q}=0$



Venne detta configurazione di **SET**

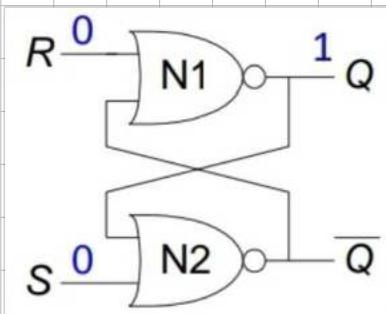
- $S=0, R=1 \rightarrow Q=0, \bar{Q}=1$



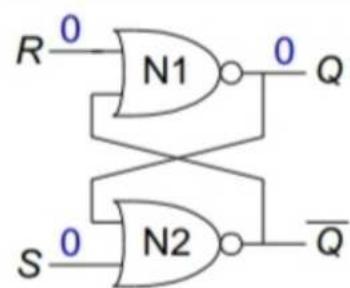
Venne detta configurazione di **RESET**

- $S=0, R=0 \rightarrow Q = Q_{PREV}$

$$Q_{PREV} = 1$$



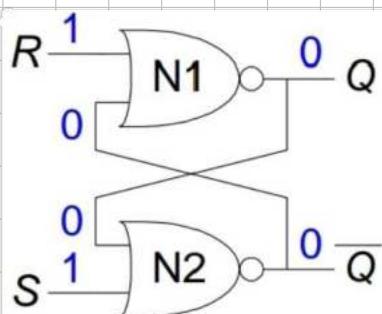
$$Q_{PREV} = 0$$



In questo caso il valore di  $Q$  dipende dal valore di  $Q$  precedentemente, e in particolare lo mantiene.

Viene detta configurazione di **MEMORIA**

- $S=1, R=1 \rightarrow Q = \bar{Q} = 0$



Questa configurazione **NON** è valida  
e va evitata, perché  $Q$  e  $\bar{Q}$  non possono  
assumere lo stesso valore.

A livello circuitale, lo stato passa da  $S=1, R=1$  a  $S=0, R=1$   
e così via, oscillando tra valori pur non definiti

Ricapitolando, l'SR LATCH memorizza un bit ( $Q$ ),

► Set pome l' output a 1 ( $S=1, R=0$ )

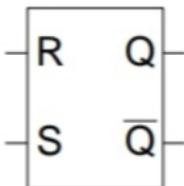
► Reset lo pome a 0 ( $S=0, R=1$ )

► Memoria memorizza l' output ( $S=0, R=0$ )

►  $S=R=1$  non è valido

Puo' essere così rappresentato:

SR Latch  
Symbol



## D LATCH

Il D LATCH è un' estensione dell' SR LATCH

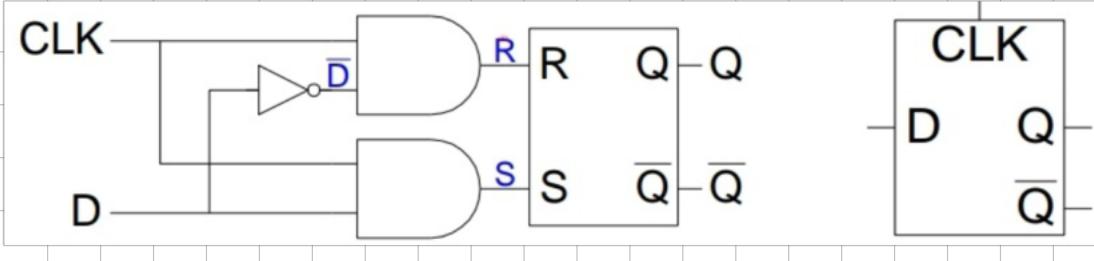
Ha 2 input:

- CLK (clock) che controlla quando cambia l' input
- D rappresenta il vero e proprio input da passare in output

Abbiamo 2 casi:

- $CLK=1 \rightarrow D$  arriva a Q (**TRASPARENTE**)
- $CLK=0 \rightarrow Q$  mantiene il nuovo valore precedente (**OPACO**)

Va **EVITATO** lo stato in cui  $Q \neq \text{NOT } \bar{Q}$



Si presenta come un SR LATCH espanso.

$CLK$	$D$	$\bar{D}$	$S$	$R$	$Q$	$\bar{Q}$
0	X	X	0	0	$Q_{prev}$	$\bar{Q}_{prev}$
1	0	1	0	1	0	1
1	1	0	1	0	1	0

**SR LATCH**

OSS

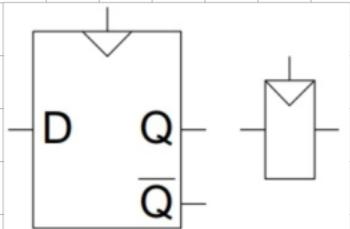
Grazie al D LATCH non avremo mai lo stato illegale  $S=1, R=1$

## D FLIP-FLOP

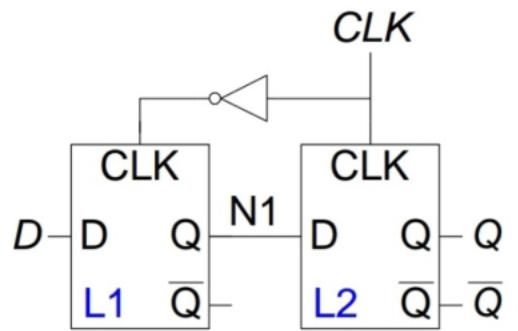
Anch'esso ha come input solo CLK e D, ma funziona così:

- Quando CLK passa da 0 a 1, D arriva a Q
- Altrimenti, Q mantiene il suo valore precedente

Siccome Q cambia solo al variare di CLK, e' pilotato da una **TRANSIZIONE** ed e' perciò detto **EDGE-TRIGGERED**



Il circuito è così composto:



L1 e L2 sono due D-LATCH, controllati da CLK

I due casi saranno:

- $CLK = 0$

► L1 è TRASPARENTE

► L2 è OPACO

► D arriva a N1

- $CLK = 1$

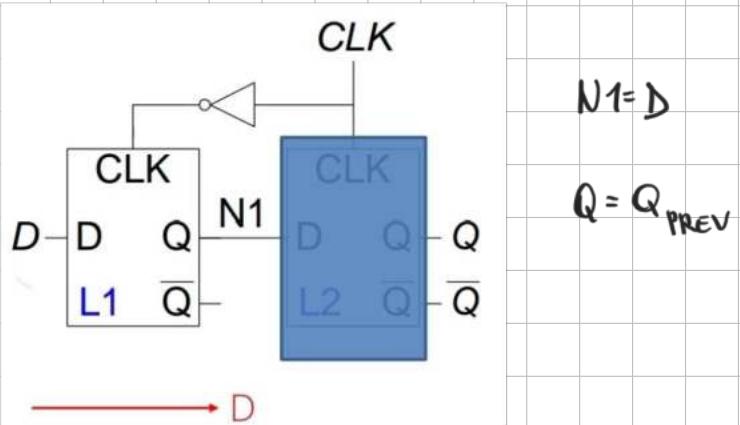
► L1 è OPACO

► L2 è TRASPARENTE

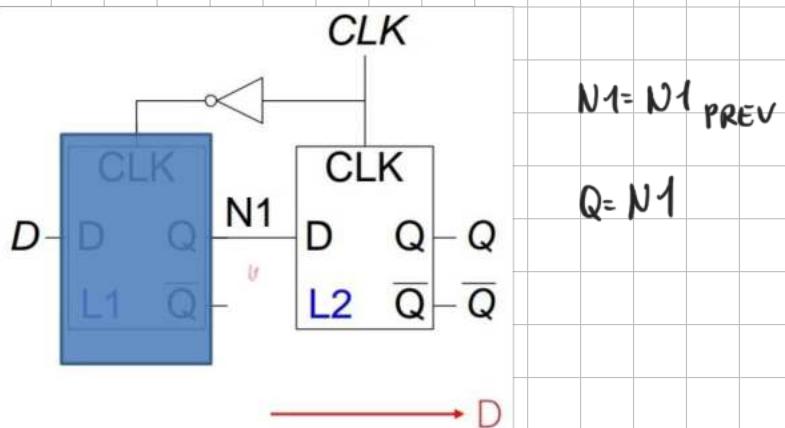
► D arriva a Q

Quindi D arriva a Q SOLO dalla Transizione di CLK  
da 0 a 1

Con  $CLK=0$  abbiamo

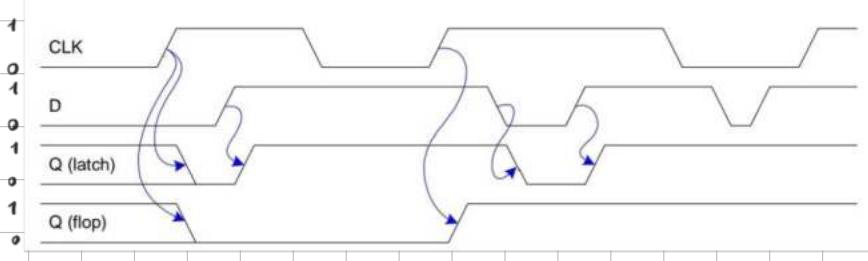
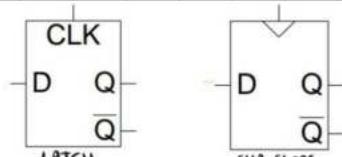


Nel momento in cui  $CLK$  passa da 0 a 1 avremo



Anche se variano D, L1 e' ora OPACO quindi non cambia N1

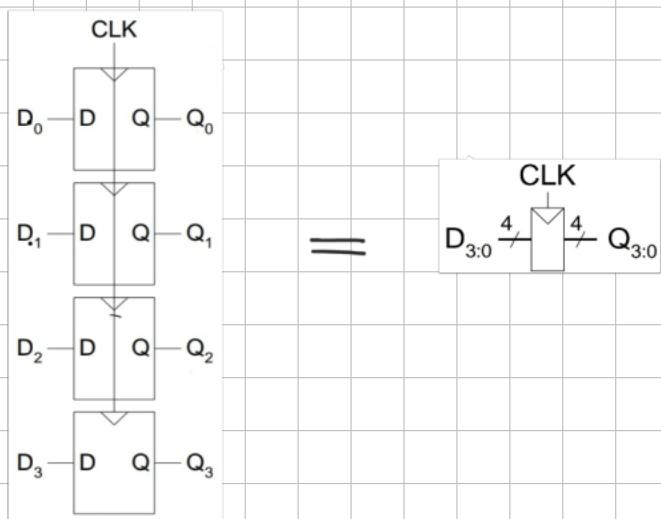
## D LATCH e D FLIP FLOPS



Questa è una rappresentazione grafica  
di come funzionano i due circuiti, considerando anche i  
ritardi del segnale

## REGISTRI

I REGISTRI sono SERIE di FLIP-FLOPS in parallelo



I registri memorizzano parole di 4 bit ( $Q_{0:3}$ ) e il valore  
di questa viene settata SOLO al battere del clock

## FLIP-FLOP "ENABLED"

Sono D FLIP FLOPS con un terzo input EN.

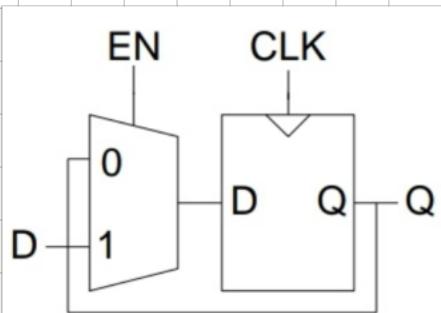
Questo (enabler) stabilisce quando il valore di D dev'essere  
memorizzato. Quindi i due casi sono:

●  $EN = 1$

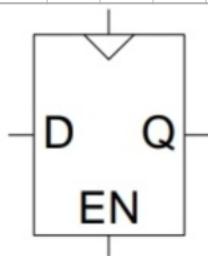
D passa fino a Q ( $clk \ 0 \rightarrow 1$ )

●  $EN = 0$

Il flip-flop mantiene il suo stato precedente



Di solito viene così rappresentato:

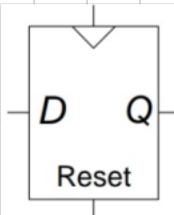


### FLIP-FLOPS "RESETABILI"

Sono D FLIP-FLOPS che hanno un terzo input RESET

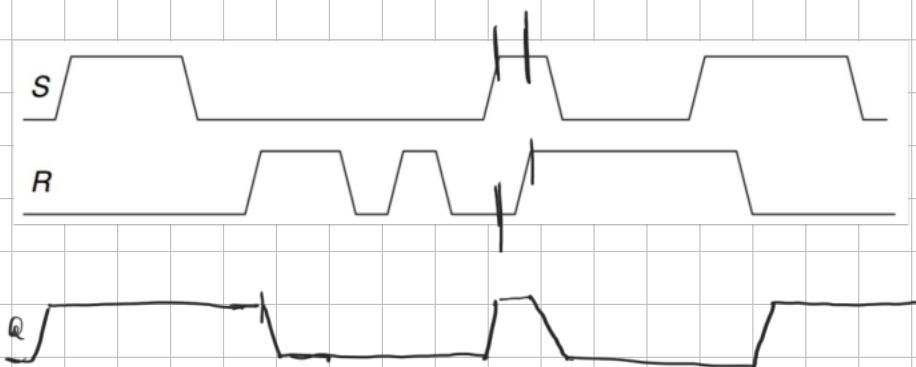
●  $RESET = 1 \rightarrow Q = 0$

●  $RESET = 0 \rightarrow$  FLIP-FLOPS normale

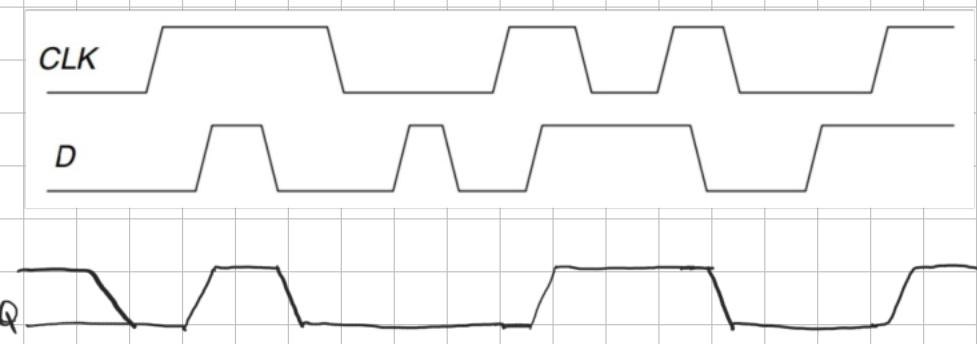


# ESERCIZI

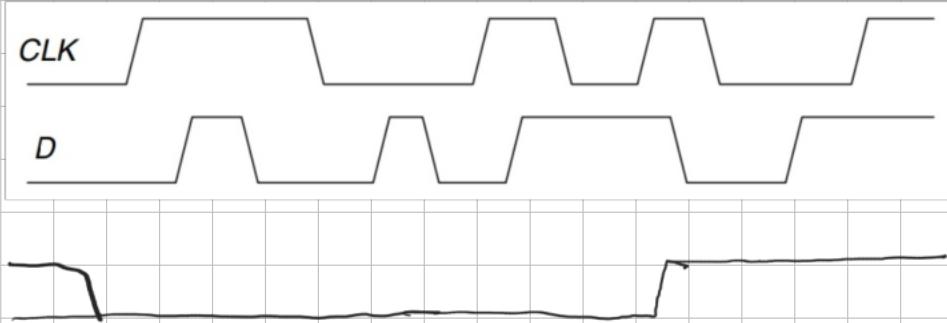
3.1)



3.3)

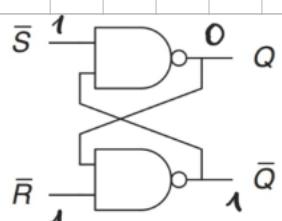


3.5)



3.7) E' una LOGICA SEQUENZIALE

S	R	Q
1	1	X
0	1	0
1	0	1
0	0	$Q_{PREV}$



3.8)

C D R S

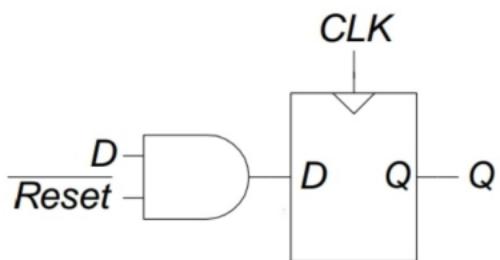






Ne esistono di 2 tipi:

- **SINCRONI**: il reset è pilotato dal clock



- **ASINCRONI**: il reset avviene non appena  $\text{RESET} = 1$

(hanno un circuito che va modificato dall'interno)

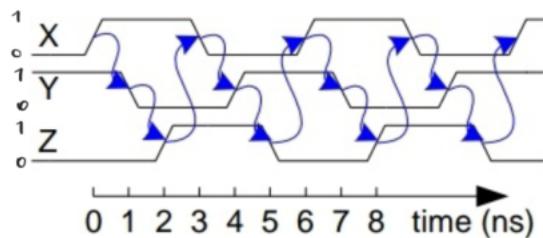
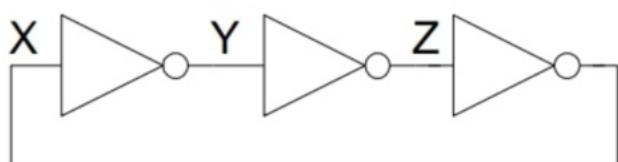
### FLIP-FLOPS "SETTABILI"

Sono D FLIP-FLOPS che hanno un terzo input **SET**

- $\text{SET} = 1 \rightarrow Q = 1$
- $\text{SET} = 0 \rightarrow$  FLIP-FLOP normale

### CRITICITÀ DELLA LOGICA SEQUENZIALI

Vediamo un tipo di circuito detto **ASTABILE**



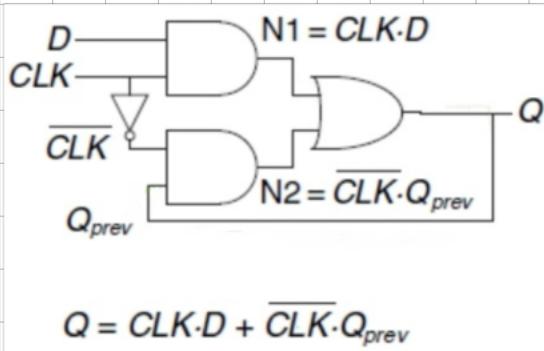
Come si puo' facilmente evincere dallo schema del comportamento, questo e' particolarmente oscillante, con un ritardo dovuto agli inverter (porta not)

Questo tipo di circuito in cui l'output e' RETROAZIONATO DIRETTAMENTE e' detto ASINCRONO

OSS

Nel caso di reti asincrone complesse (con l'uso di porte AND, NOT, OR) il comportamento puo' dipendere fortemente dai ritardi accumulati dalle singole porte.

E.s.

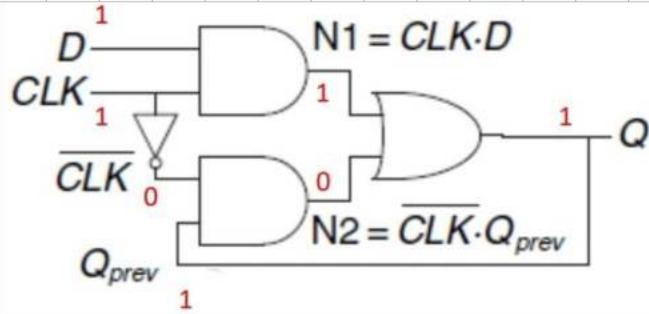


CLK	D	$Q_{\text{prev}}$	Q
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	0
1	1	0	1
1	1	1	1

$$Q = Q_{\text{prev}}$$

$$Q = D$$

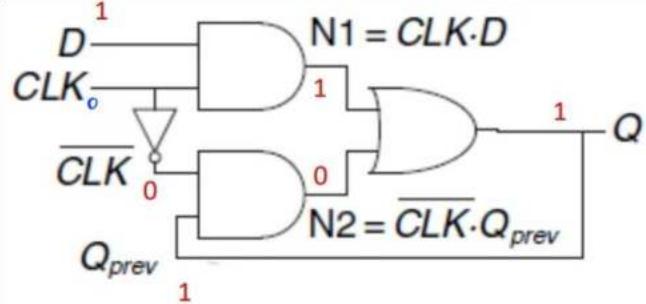
Sembra che il circuito rappresenti un DLATCH (CLK=0 aperto, CLK=1 trasparente). In realtà non e' cosi' per i ritardi:



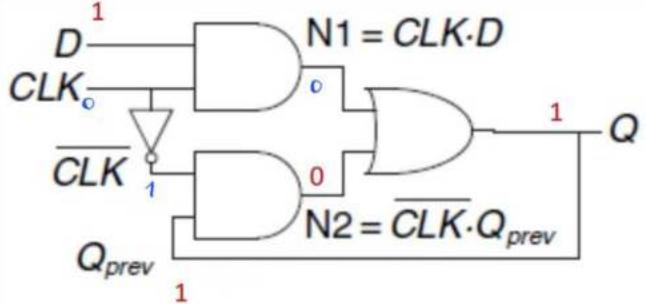
$$\begin{aligned} &D = 1 \\ &\Rightarrow Q = 1 \\ &CLK = 1 \end{aligned}$$

•  $t_0$

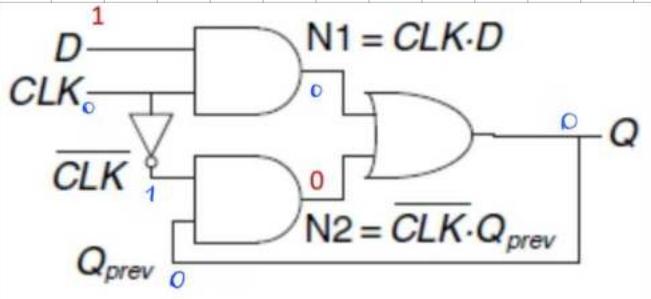
$CLK \ 1 \rightarrow 0$



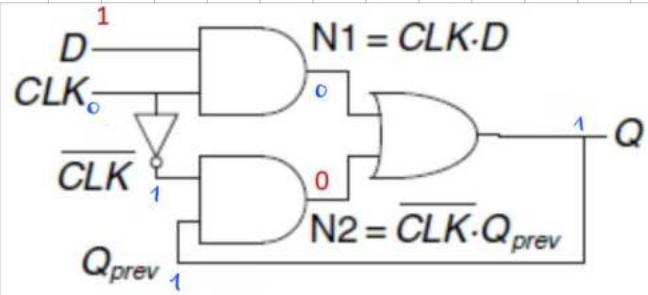
•  $t_1$



•  $t_2$



•  $t_3 (=t_1)$



A questo punto seguirebbe  $t_2$  poi  $t_3$  e così via.

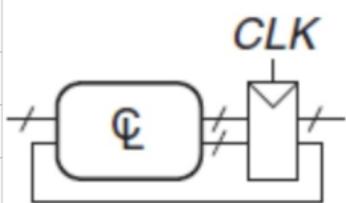
Il valore di  $Q$  oscilla tra 0 e 1





## LOGICHE SEQUENZIALI SINCRONE

Per non incontrare in criticità dovute al ritardo del circuito, si cerca di evitare di **RETROAZIONARE** direttamente l'output. Si interpone quindi un **REGISTRO** nel ciclo di retroazione. Se il clock è più lento del ritardo accumulato, il registro permette al sistema di essere sincronizzato col clock, costituendo un **CIRCUITO SINCRONO**.



## STATO DELLE LOGICHE SEQUENZIALI SINCRONE

Se una logica combinatoria gestisce l'output in funzione unicamente dell'input ( $OUT = f(INPUT)$ ) nelle **LOGICHE SEQUENZIALI SINCRONE** l'output varia in funzione anche dello **STATO CORRENTE** come abbiamo visto. Lo stato **SUCCESSIONE** a sua volta varierà in funzione dell'**INPUT** e dello stato corrente

$$OUT = f(INPUT, S_c)$$

$$S_s = g(INPUT, S_c)$$

## DESIGN DELLE L.S.S.

I circuiti delle logiche sequenziali sincrone sono così composti:

- Ogni cammino **CICLICO** ha un suo **REGISTRO**
- I registri determinano lo **STATO** del **SISTEMA**
- I cambiamenti di **STATO** sono determinati quindi dalle **TRANSIZIONI** del **CLOCK** ( $0 \rightarrow 1$ ), il sistema risulta quindi essere sincronizzato al clock

Il sistema ha delle sue regole di composizione:

- Il sistema è composto unicamente da **REGISTRI** e **CIRCUITI COMBINATORI**
- C'è almeno un registro
- Tutti i registri sono sincronizzati a un unico clock
- Ogni ciclo contiene quindi almeno un registro

Possiamo suddividere i circuiti sequenziali sincroni in:

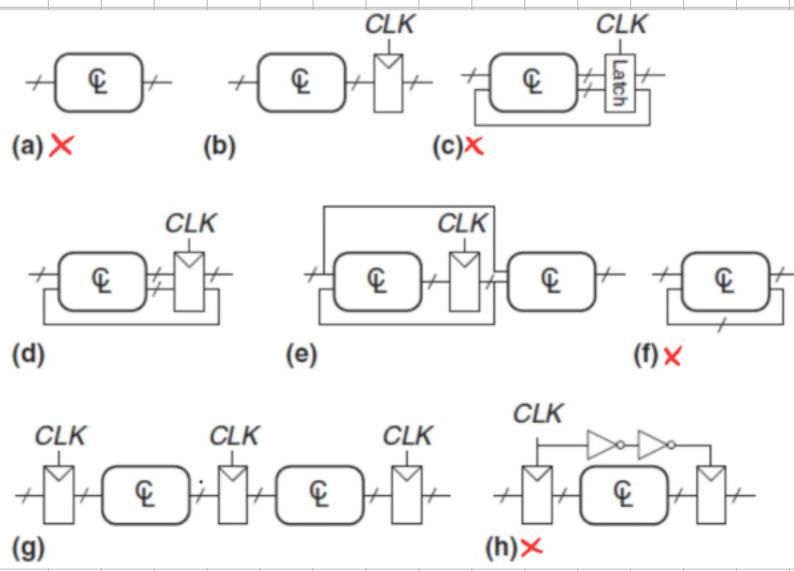
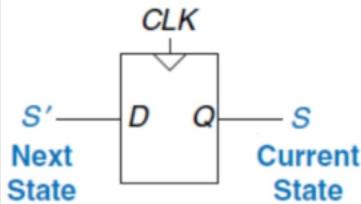
- Finite State Machines (FSM<sub>s</sub>)
- Pipelines

E.

- Un D FLIP-FLOP è il più banale circuito sequenziale sincrono

$$Q = S_c$$

$$D = S_N$$



a) non ha il registro

c) la retroazione è effettuata attraverso un D LATCH e non un registro

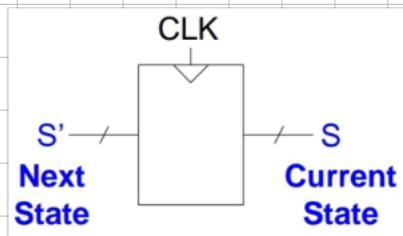
f) la retroazione è diretta (asincrono)

h) vi sono 2 porte NOT che generano ritardo, quindi i due registri non avranno lo stesso clock

## FINITE STATE MACHINES

E' costituita da:

- **STATE REGISTER**: memorizzano lo stato corrente e caricano il successivo al battere del clock



- **LOGICHE COMBINATORIE**: queste definiscono le funzioni booleane per l'output (in funzione di input e stato corrente) e per lo stato successivo (in funzione di input e stato corrente)



## MEALY e MOORE FSM

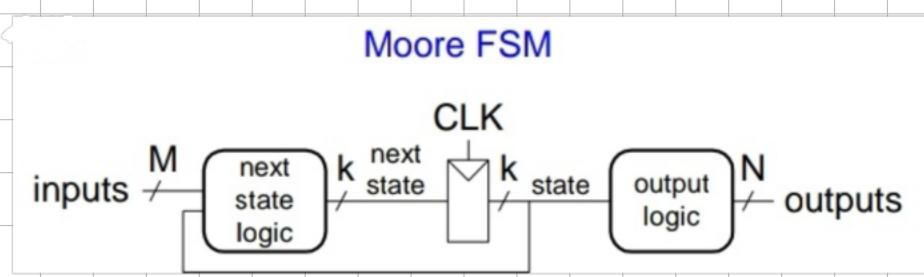
Gli FSM si dividono in due tipi: **MEALY** e **MOORE**.

In entrambi i casi l'output successivo dipende dal corrente

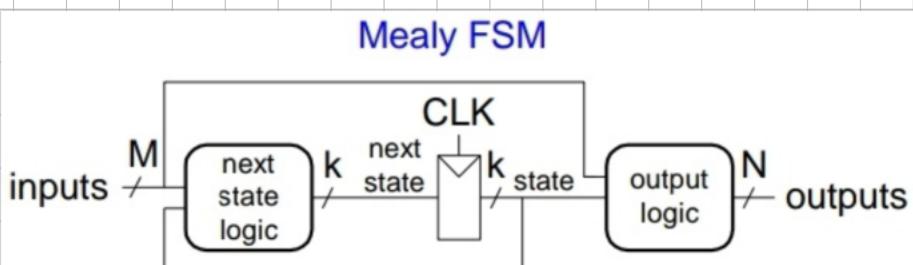
$$S_N = g(\text{INPUT}, S_C)$$

La differenza sta nell' OUTPUT:

● MOORE FSM:  $OUTPUT = f(S_c)$

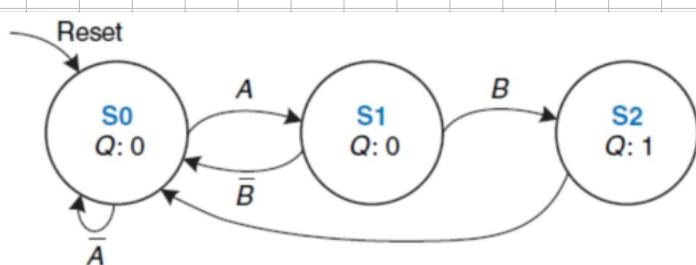


● MEALY FSM:  $OUTPUT = f(INPUT, S_c)$



E

● Esempio di FSH MOORE



l' output Q dipende solo dallo stato corrente,  
quindi ogni stato determina un output

► Nello stato S<sub>0</sub>, se l'input A vale 0, si mantiene lo stato,  
altrimenti passa in S<sub>1</sub>.

► Nello stato  $S_1$ , al variare dell' input B cambierà

lo stato: se  $B=1$  passa in  $S_2$ , altrimenti resta in  $S_1$

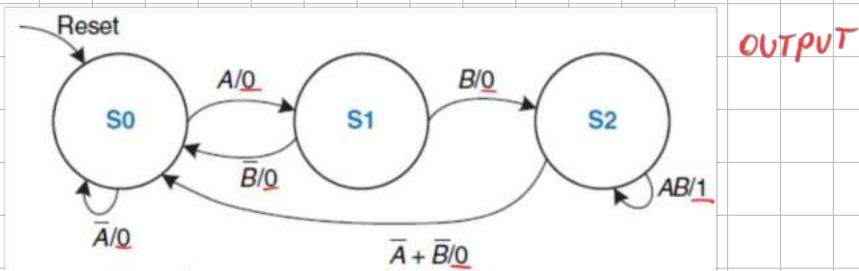
► Nello stato  $S_2$  al battere del clock passerà sempre in  $S_0$

	$S_0$	$S_0$	$S_0$	$S_1$	$S_0$	$S_0$	$S_0$	$S_1$	$S_2$	$S_0$
A	0	0	1	1	0	0	1	1	0	1
B	1	0	1	0	1	0	0	1	1	0
Q	0	0	0	0	0	0	0	0	1	0

### RICORDA

I valori degli input cambiano al battere del clock

### Esempio di FSM MEALY



L'output in questo caso dipende sia dallo stato da cui parte, sia dall'input, per questo viene indicato sulle frecce di transizione

	$S_0$	$S_0$	$S_0$	$S_1$	$S_2$	$S_2$	$S_0$	$S_1$	$S_2$	$S_0$
A	0	0	1	1	1	0	1	1	0	1
B	1	0	1	1	1	1	0	1	1	0
Q	0	0	0	0	1	0	0	0	0	0

OSS

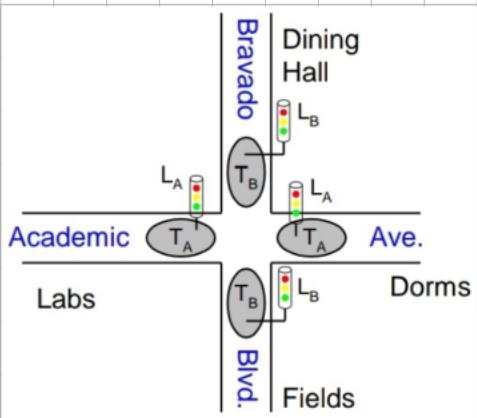
E' fondamentale che non ci siano configurazioni equivalenti a definire gli stati. Ad esempio se il passaggio a  $S_1$  e' dato da  $A+B$  e il passaggio a  $S_2$  e' dato da  $AB$ , nel caso di una configurazione  $A=1$  e  $B=1$  non sappiamo a che stato passare.

Per evitare che ciò accada le configurazioni di uno stato devono soddisfare 2 proprietà:

$$\blacktriangleright E_i \cdot E_j = 0$$

$$\blacktriangleright E_1 + E_2 + E_3 = 1 \quad (\text{esempio con 3 variabili})$$

● Vediamo adesso il caso di un sistema di semafori

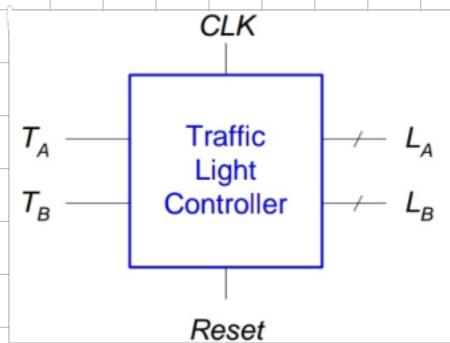


I sensori  $T_A$  e  $T_B$  indicano quando c'è traffico

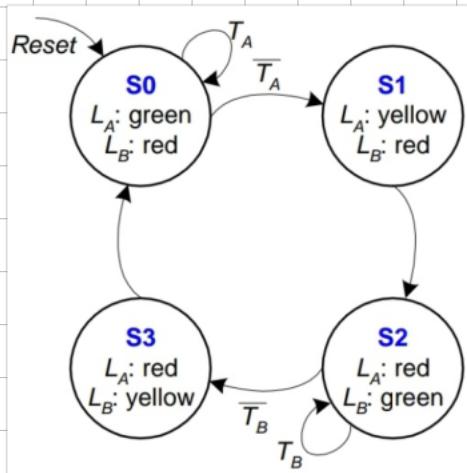
I semafori saranno:

INPUT: CLK, RESET,  $T_A$ ,  $T_B$

OUTPUT:  $L_A$ ,  $L_B$



Vediamo il **DIAGRAMMA DI TRANSIZIONE**



RESET riporta lo stato a  $S_0$ , che si ripete fintanto che  $T_A = 1$ .

Quando  $T_A = 0$  passa allo stato  $S_1$ , che al clock successivo

passera' inevitabilmente a  $S_2$ . Analogamente accadrà con  $T_B$

Alessio dobbiamo però capire come realizzare un circuito concreto di questo diagramma. Partiamo dalla **TABELLA DI TRANSIZIONE**

Current State	Inputs		Next State
	$T_A$	$T_B$	
$S$			$S'$
S0	0	X	S1
S0	1	X	S0
S1	X	X	S2
S2	X	0	S3
S2	X	1	S2
S3	X	X	S0

I nostri stati però sono ancora figurati, dobbiamo **CODIFICARLI**.

Avevamo 4 stati, ci servono 2 bit. Ecco una possibile soluzione

State	Encoding
S0	00
S1	01
S2	10
S3	11

A questo punto codifichiamo la tabella di transizione

Current State	Inputs		Next State			
	$S_1$	$S_0$	$T_A$	$T_B$	$S'_1$	$S'_0$
0	0	0	0	X	0	1
0	0	0	1	X	0	0
0	1	X	X	X	1	0
1	0	X	0	0	1	1
1	0	X	1	0	1	0
1	1	X	X	X	0	0

A questo punto possiamo scrivere il **NEXT STATE LOGIC**

ovvero l' espressione booleana che espriue i due next state (come se fossero degli output) in funzione dello stato corrente

e degli input. In questo caso il risultato sarà:

$$\bullet S'_1 = S_1 \oplus S_0$$

$$\bullet S'_0 = \overline{S_1} \overline{S_0} T_A + S_1 \overline{S_0} T_B$$

Di conseguenza anche l'output dei semafori va codificato.

Se come i possibili output sono 3, useremo 2 bit anche qui

Output	Encoding
green	00
yellow	01
red	10

Ora guardando il diagramma di transizione, possiamo codificare gli output

Current State	Outputs					
	$S_1$	$S_0$	$L_{A1}$	$L_{A0}$	$L_{B1}$	$L_{B0}$
$S_0$	0	0	0	0	1	0
$S_1$	0	1	0	1	1	0
$S_2$	1	0	1	0	0	0
$S_3$	1	1	1	0	0	1

$L_A$        $L_B$

E quindi possiamo a sua volta scrivere l'espressione booleana dell'output

$$L_{A1} = S_1$$

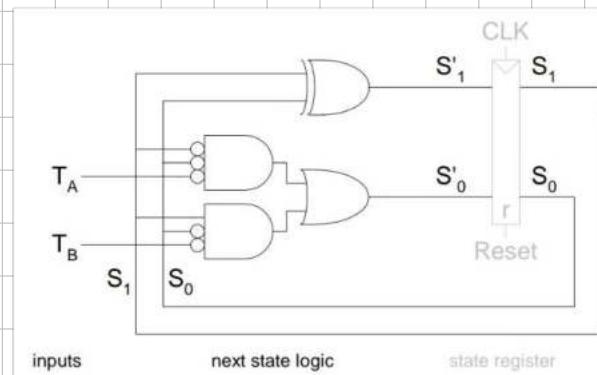
$$L_{A0} = \overline{S_1} S_0$$

$$L_{B1} = \overline{S_1}$$

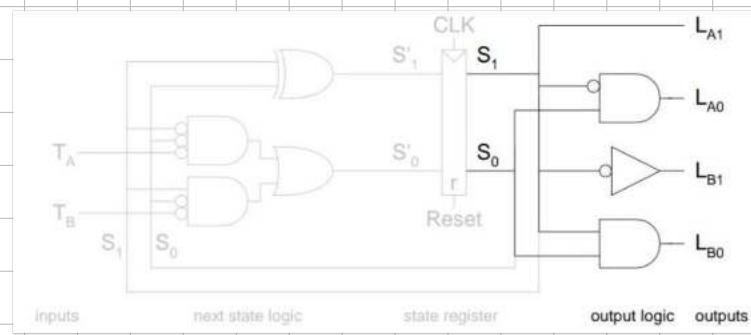
$$L_{B0} = S_1 S_0$$

In definitiva, possiamo definire il circuito della FSM che definisce questo semaforo.

Lo schema della **LOGICA DI TRANSIZIONE** sarà:



Lo schema della **LOGICA DI OUTPUT** sarà:



## ENCODING DEGLI STATI

Di solito gli stati vengano **CODIFICATI IN BINARIO** (0,1).

Un' alternativa è l'**ENCODING ONE-HOT** secondo cui:

- ogni bit indica uno stato
- per ogni stato solo un bit è uguale a uno

E. (4 stati)

0001, 0010, 0100, 1000

- Richiede più flip flops (4) per ogni stato rispetto all'encoding binario (2)
- Spesso la logica combinatoria associata è più semplice

## MOORE e MEALY FMS a confronto

Immaginiamo di dover costruire un automa che dato un tape

10011000101101

Restituisca 1 quando si ripete la sequenza 01

● MOORE

► OUTPUT 1

10011000101101

legge 2 cifre alla volta

## ► OUTPUT 0

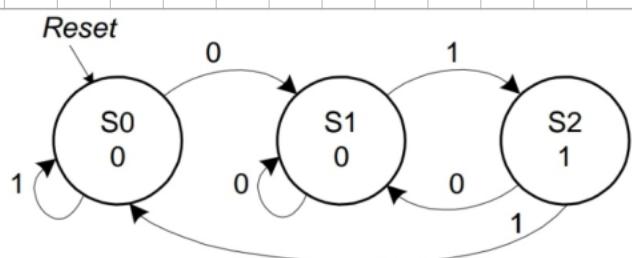
10011000101101

legge 2 cifre alla volta

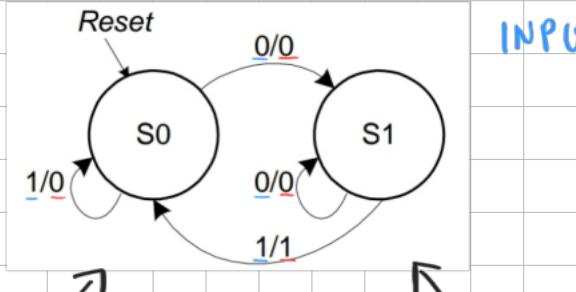
## ► OUTPUT 1

10011000101101

legge 1 cifra alla volta



## ● MEALY



INPUT / OUTPUT

↗  
Stato nel quale  
la distanza e' 1 dal  
prossimo output

↖  
Stato in cui potenzialmente  
possiamo avere OUT 1

Mealy ci permette di avere un numero minore di STATI

Vediamo adesso le tabelle di transizione e quindi i circuiti

# ● MOORE

Current State		Inputs	Next State	
$S_1$	$S_0$	$A$	$S'_1$	$S'_0$
0	0	0	0	1
0	0	1	0	0
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	0	0

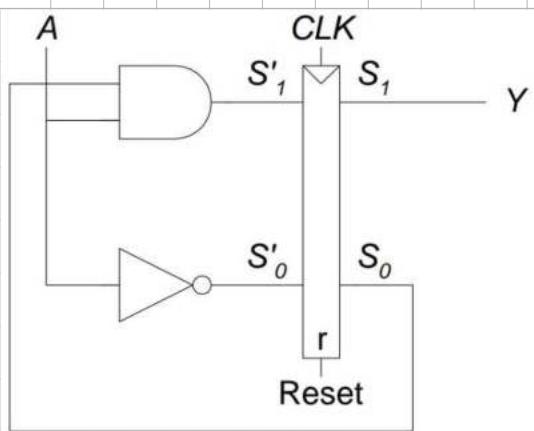
State	Encoding
$S_0$	00
$S_1$	01
$S_2$	10

$$S'_1 = S_0 A$$

$$S'_0 = \overline{A}$$

Current State		Output
$S_1$	$S_0$	$Y$
0	0	0
0	1	0
1	0	1

$$Y = S_1$$



CIRCUITO DELLA FSM MOORE

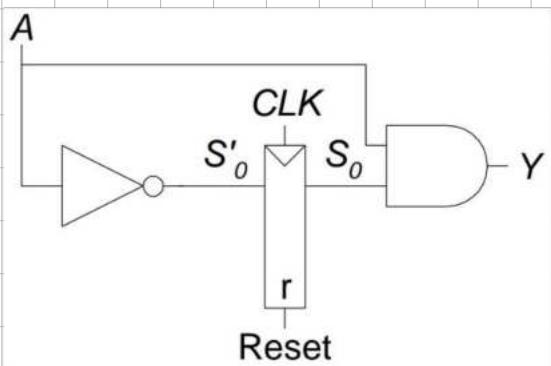
## ● MEALY

Current State	Input	Next State	Output
S	A	S'	Y
0	0	1	0
0	1	0	0
1	0	1	0
1	1	0	1

State	Encoding
S0	0
S1	1

$$S' = \bar{A}$$

$$Y = SA$$



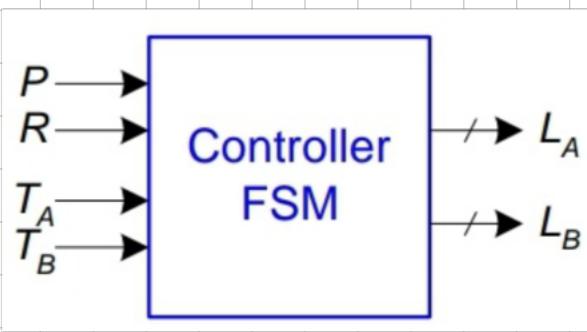
OSS

L'automa di MOORE ha registri più grandi  
e circuiti più complessi, mentre l'FSM MEALY ha il  
problema di avere l'output leggermente sfasato essendo che  
Y non è regolato da un clock (non genera grandi problemi)

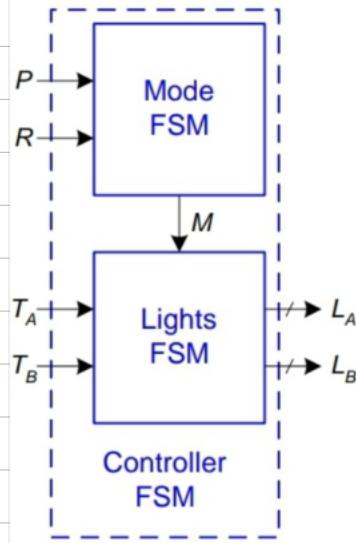
## FATTORIZZAZIONE DI FSM

FATTORIZZARE una FSM significa suddividerla in FSM più piccole

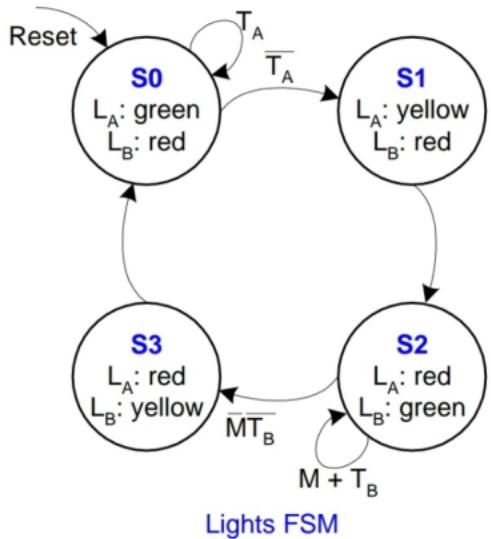
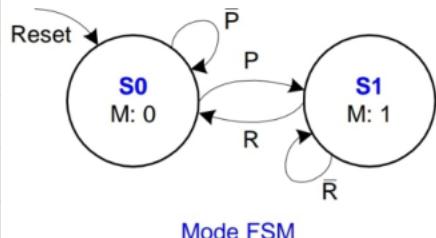
Immaginiamo di introdurre due input  $P$  ed  $R$  al caso del semaforo tali che quando  $P=1$  una strada è sempre verde mentre quando  $R=1 \rightarrow P=0$ .



Questo è il nuovo controller a 4 input, proviamo a fattorizzarlo



I diagrammi di transizione saranno:



La nostra FSH MODE influenza solo lo stato S2 perch' e' quello che vogliamo mantenere con  $P=0$

### PROGETTARE UNA FMS

Per progettare una FMS dobbiamo:

- Identificare **INPUT** e **OUTPUT**
- Abbozzare un **DIAGRAMMA DI TRANSIZIONE**
- Scrivere la **TABELLA DI TRANSIZIONE**
- Selezionare un **ENCODING** per gli stati (binario)
- Sviluppare l'automa di **HEALY/MOORE**:
  - Riscrivere la Tabella di transizione codificata
  - Scrivere la **TABELLA DI OUTPUT**
  - Scrivere le **EQUAZIONI BOOLEANE** per le logiche di **NEXT STATE**

e di **OUTPUT**

- Minimizzare le espressioni (eventualmente con K-maps)
- Sviluppare il **CIRCUITO**

## PARALLELISMO

**PARALLELISMO** significa eseguire contemporaneamente più tasks.

Esistono 2 tipi di parallelismo:

**SPAZIALE**: Duplicare l'hardware per eseguire più tasks assieme

**TEMPORALE**: le task vengono suddivise in più fasi eseguite in pipelining (un circuito combinatorio particolare)

## LATENZA e THROUHPUT

**TOKEN**: Gruppo di input da processare per ottenere un output significativo

**LATENCY**: Tempo che occorre al Token per essere processato e produrre un output

**THROUHPUT**: Numero di output prodotti per unità di tempo

Il parallelismo aumenta il throughput

E.

Vogliamo preparare una torta, ci vogliono 5 minuti a prepararla e 15 minuti a cucinarla

$$\text{LATENCY} = 5 + 10 = 20 \text{ min} = 1/3 \text{ h}$$

$$\text{THROUGHPUT} = \frac{1}{\text{LATENCY}} = 3 \text{ torte/h}$$

Parallelismo spaziale: usiamo un secondo forno per cucinare

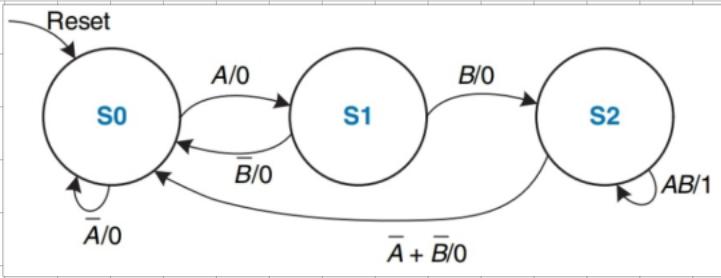
Parallelismo temporale: 2 fasi, preparare e cucinare. Mentre una torta è in forno preparo la prossima torta

Possiamo verificare che con il parallelismo spaziale avremo 6 torte dopo 1 ora, mentre con il temporale solo 3,7 (senza però un secondo forno)

ESERCIZI 3,23 / 3,31

SLIDE 76

3.23)



S	A	B	S'
0	0	x	1
0	1	x	0
1	x	0	2
1	x	1	0
2	0	0	2
2			0

S	E
0	00
1	01
2	10

S, S <sub>0</sub>	AB	00	01	11	10
		01	01	10	
		00	00	00	00
		10	00	10	

$$\bar{S}_1 \bar{S}_0 B +$$

S <sub>1</sub> S <sub>0</sub>	A	B	S' <sub>1</sub> S' <sub>0</sub>	OUT
00	1	x	01	0
00	0	x	00	0
01	x	1	10	0
01	x	0	00	0
10	1	1	00	1
10	1	0	00	0
10	0	1	00	0
10	0	0	10	0

S <sub>1</sub> '	S, S <sub>0</sub>	AB	00	01	11	10
			00	00	00	00
			01	01	10	
			11	x	x	x

$$= \bar{S}_1 \bar{S}_0 B + S_1 AB$$

S <sub>0</sub> '	S, S <sub>0</sub>	AB	00	01	11	10
			00	00	11	
			01	00	00	
			10	00	00	

$$= \bar{S}_1 \bar{S}_0 A$$

Q	AB	S, S <sub>0</sub>	00	01	11	10
			00			
			01			
			11	x	x	x

$$= S_1 AB$$

$$S_0' = \bar{S}_1 \bar{S}_0 A$$

$$S_1 = S_0 B + S_1 AB$$

$$Q = S_1 AB$$

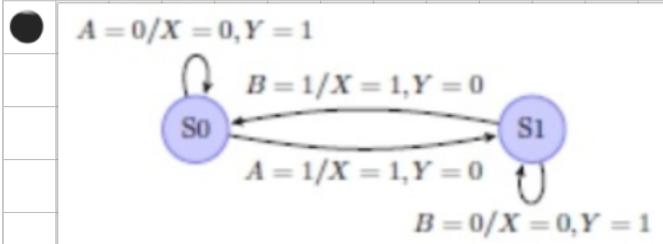








## ESERCIZI



S	E
0	0
1	1

S	A	B	S'	X	Y
0	0	•	0	0	1
0	1	•	1	1	0
1	•	0	1	0	1
1	•	1	0	1	0

$$S' = \bar{S}A + SB$$

$$X = \bar{S}A + SB$$

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

S	AB	00	01	11	10
0	00	0	0	1	1
1	11	0	0	1	1

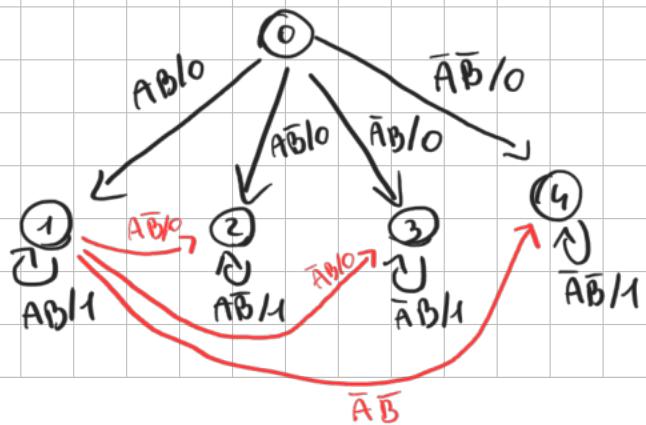
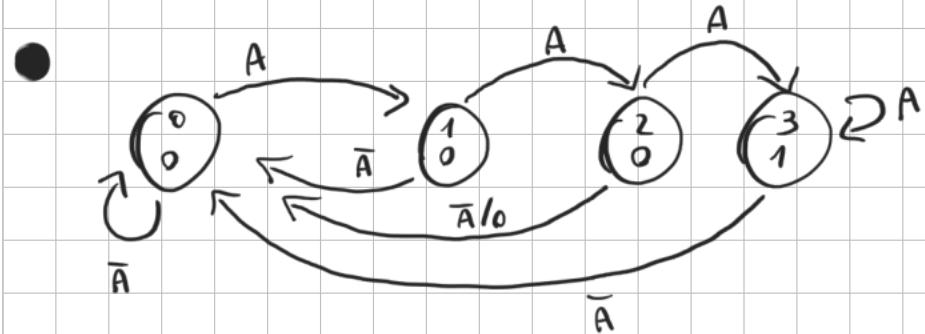
$$= S'$$

S	AB	00	01	11	10
0	00	0	0	1	1
1	01	0	1	1	0

$$= X$$

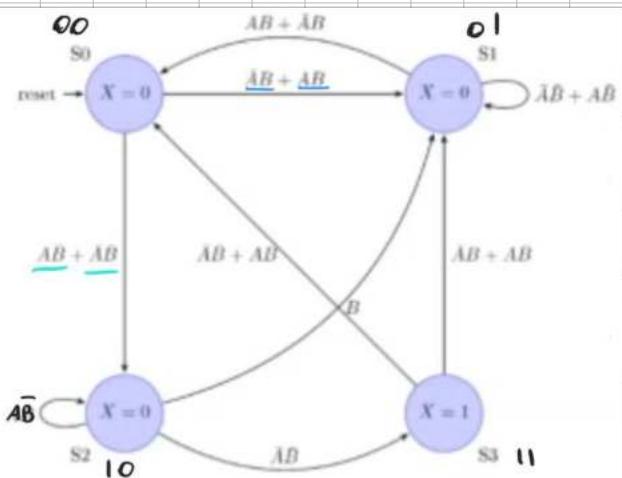
S	AB	00	01	11	10
0	11	0	0	0	0
1	10	1	0	0	1

$$= Y$$



Così per tutti gli stati:

$S$     $S_1$ ,  $S_0$   
 0   0 0  
 1   0 1  
 2   1 0  
 3   1 1



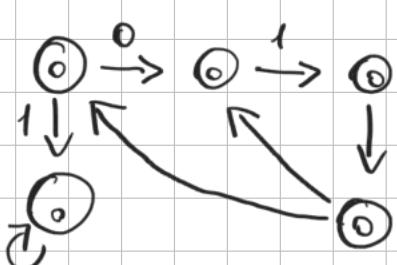
$S_1^1$	$S_0, S_1$	$A, B$	00	01	11	10
			1	0	0	1
			00	01	11	10
			01	0	0	0
			11	0	0	0
			10	1	0	1

$$S_1^1 = \overline{S_0} \overline{B}$$

$S_0^1$	$S_0, S_1$	$A, B$	00	01	11	10
			0	1	1	0
			00	01	11	10
			01	1	0	1
			11	0	1	0
			10	1	1	1

$$S_0^1 =$$

$$S_0^1 = \overline{S_0} B + \underline{S_0} \overline{A} \overline{B} + \overline{S_1} \overline{S_0} \overline{A} + S_1 \overline{A} B + \underline{\overline{S_1} S_0} B$$
✓
✓









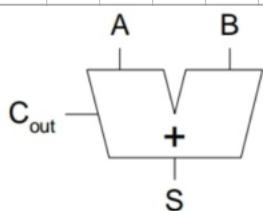
## 1-BIT ADDERS

Sono circuiti che, presi due bit, li sommano tra di loro

Si dividono in **HALF** e **FULL** adders

### HALF ADDERS

Si occupano di sommare due soli bit con riporto  $C_{out}$



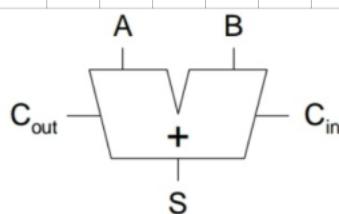
A	B	$C_{out}$	S
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

$$S = A \oplus B$$

$$C_{out} = AB$$

### FULL ADDERS

Teniamo conto di un secondo riporto  $C_{in}$  (che vedremo più avanti)



$C_{in}$	A	B	$C_{out}$	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

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

$$C_{out} = AB + AC_{in} + BC_{in}$$

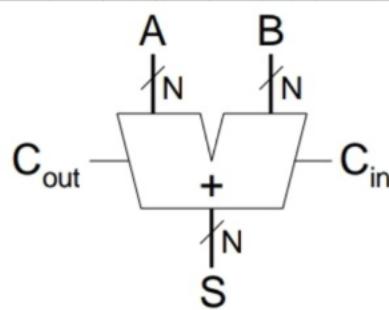
Se gli 1 in input sono **PARI** l'output sarà 0, altrimenti 1

Se abbiamo 2 o più "1" in input, avremo un riporto  $C_{out}$

## MULTIBIT ADDERS CPA<sub>S</sub>

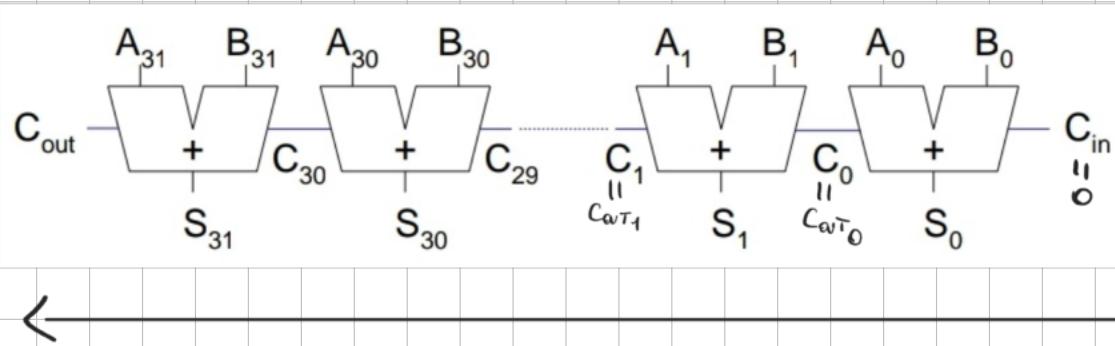
Possono sommare parole con più bits. Si suddividono in:

**RIPPLE-CARRY** e **CARRY-LOOKHEAD** (prestazioni migliori ma richiede un hardware più potente).



A e B sono parole a N bit

## RIPPLE-CARRY ADDER



È una serie di **FULL-ADDER** dalla cifra **meno** significativa alla più significativa

In questo caso sommiamo parole a 32 bit

Il  $C_{in}$  parte da 0, porta il resto delle due cifre sommate e diventerà il  $C_{out}$  di quell'adder per poi diventare il  $C_{in}$  dell'adder successivo e così via

Il ritardo, sia  $T_{FA}$  il ritardo di un singolo adder, e  
 $N T_{FA}$  dove  $n$  è la lunghezza delle parole.

### CARRY-LOOK-AHEAD ADDER

Il concetto di questo circuito è quello di non calcolare il riporto di ogni singolo adder, bensì calcolarlo in blocchi.

Per fare ciò dobbiamo suddividere le cifre  $A_i, B_i$  in "categorie".

$C_{in}$	$A_i$	$B_i$	$C_{out}$
0	0	0	0
1			0
0	0	1	0
1			1
0	1	0	0
1			1
0	1	1	1
1			1

La configurazione 1-1 è detta GENERATE perché genera un riporto a prescindere dal  $C_{in}$

$$G_i = A_i B_i$$

Le configurazioni 1-0 / 0-1 sono dette PROPAGATE perché appunto propagano il resto  $c_{in}$  in  $C_{out}$

$$P_i = A_i + B_i$$

Quindi il valore di  $C_{out} = G_i + P_i C_{i-1}$

GENERATE → PROPAGATE ↑

## BLOCK PROPAGATE/GENERATE

Vediamo come si comportano queste funzioni a BLOCCI di bits

$C_{IN}$	$C_{out}$
0 0 1 0 1	1 1 1 1 1 0 0 0 0 0 0 0 0 0
0 0 1 0	1 0 0 1 0 1 0 1 0 1 0 0 0 1
0 0 1 0	0 1 1 0 1 1 0 0 0 0 0 0 0 0
0 1 0 1	0 0 0 0 0 0 0 0 0 1 0 0 0 1
<u>BLOCCO</u>	
0 0 1 0 0 0 0 0 0	1 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 1 0	1 0 0 1 0 1 0 1 0 1 0 0 0 1
0 0 1 0	0 1 1 0 0 0 1 0 0 0 0 0 0 0
0 1 0 0	1 1 1 1 1 0 0 0 1 0 0 0 1

Il resto si propaga perché tutti e due i blocchi sono composti

SOLO da configurazioni PROPAGATE

①	0 0 1 0 0 0 1 1 1 1 0 0 0 0 0 0 0 0
②	0 0 1 0 1 1 1 0 0 1 0 0 0 0 0 0 0 0
①	0 0 1 0 1 0 0 1 1 0 1 0 1 0 0 0 0 1
②	0 0 1 0 0 0 1 0 1 0 1 1 0 1 0 0 0 0
①	0 0 1 0 0 0 1 0 0 0 0 0 0 1 0 0 0 1
②	0 1 0 0 1 1 1 1 1 0 0 0 1 0 0 0 1

Nel caso ① la configurazione 0-0 elimina il riporto, nel

caso ② la configurazione GENERATE 1-1 genera un riporto.

Le formule per esprimere queste regole sono:

(Esempio per parole a 5 bit)

$$P_{3:0} = P_3 P_2 P_1 P_0$$

$$G_{3:0} = G_3 + P_3 G_2 + P_3 P_2 G_1 + P_3 P_2 P_1 G_0$$

In generale

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

$$G_{i:j} = G_i + P_i (G_{i-1} + P_{i-1} (G_{i-2} + P_{i-2} (\dots G_j) \dots))$$

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

E.s.

Sommiamo le parole  $0x52A703C1$  e  $0xA05C11E3$

$$0x52A703C1 = 0101 \quad 0010 \quad 1010 \quad 0111 \quad 0000 \quad 0011 \quad 1100 \quad 0001$$

$$0xA05C11E3 = 1010 \quad 0000 \quad 0101 \quad 1100 \quad 0001 \quad 0001 \quad 1110 \quad 0011$$

$$G_{3:0} = 0$$

$$P_{3:0} = 0$$

Gli step quindi sono

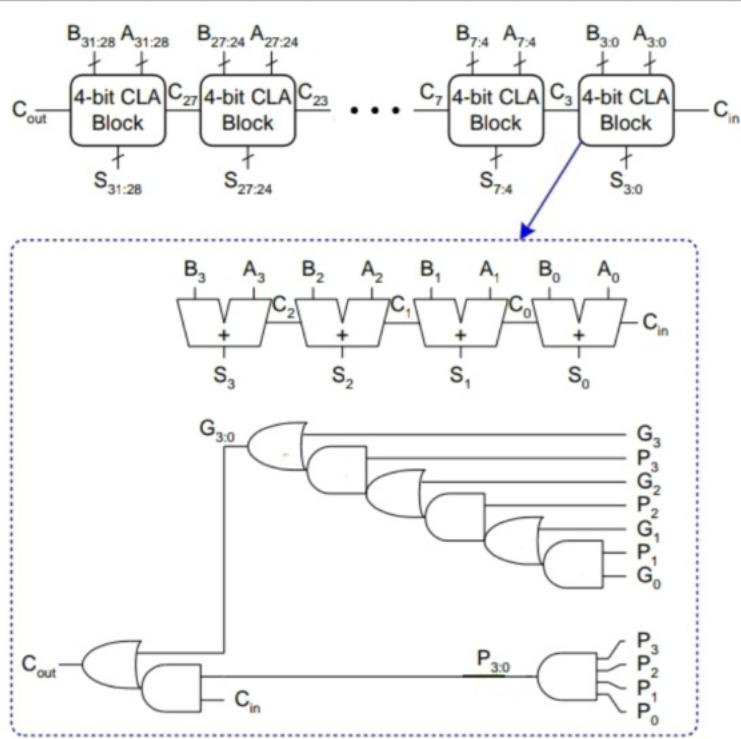
① Calcolo di  $G_i$  e  $P_i$

② Calcolo  $G$  e  $P$  per i blocchi

③  $C_{1N}$  si propaga mediante propagati generate dai vari blocchi

## 32-BIT CLA CON BLOCCI 4-BIT

Vediamo quindi com'è fatto un circuito CLA a blocchi di 4-bit



Il circuito è la traduzione circuitale dell'espressione booleana per il calcolo dei  $G$  e  $P$

## RITARDO DEI CLA

Sia

$t_{pg}$ : il ritardo per generare gli  $P_i$  e  $G_i$

$t_{pg\text{ block}}$ : il ritardo per generare gli  $P_{i:j}$  e  $G_{i:j}$

$t_{AND\_OR}$ : è generato dalle porte AND/OR che si propagano

da  $C_{IN}$  a  $C_{OUT}$  che si propaga a sua volta lungo i  
 $N/K - 1$  blocchi (K numero di bits per blocco)

La formula generale è:

$$t_{CLA} = t_{pg} + t_{pg\ block} + (N/K - 1)t_{AND-OR} + Kt_{FA}$$

OSS

Conviene usare un CLA anziché un CRA per parole con più di 16 bits

E.

Vediamo il ritardo su parole a 32-bit considerando che  
una porta logica ha un ritardo di 100 ps mentre  
un full adder ha un ritardo di 300 ps

$$t_{ripple} = Nt_{FA} = 32(300) = 9,6 \text{ ms}$$

$$t_{CLA} = 100 + 600 + 7 \cdot 200 + 4(300) = 3,3 \text{ ms}$$

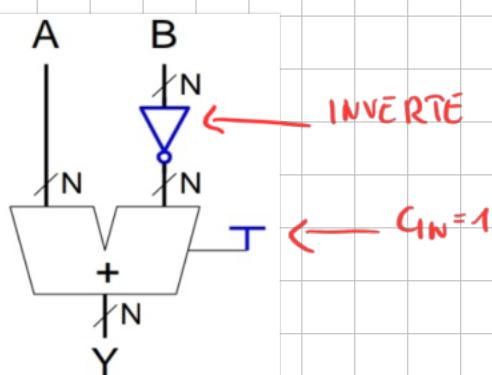
Il CLA è più veloce perché calcola il ritardo dei blocchi in  
parallelo

## SOTTRATTORE

Nella rappresentazione a complemento a 2 per complementare un numero bisogna invertire ogni cifra e sommare 1:

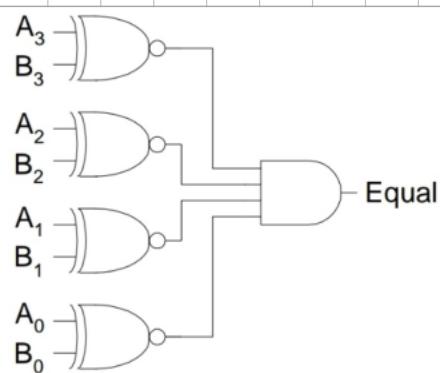
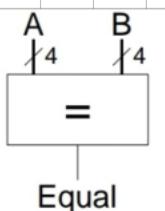
$$2 = 0010_2 \rightarrow -2 = 1101_2 + 0001 = 1110_2$$

Possiamo effettuare quest'operazione in un **FULL-ADDER** considerando l'1 da sommare, come primo  $C_{IN}$



## COMPARATORE

E' un dispositivo che prese due parole restituisce 1 solo se sono uguali

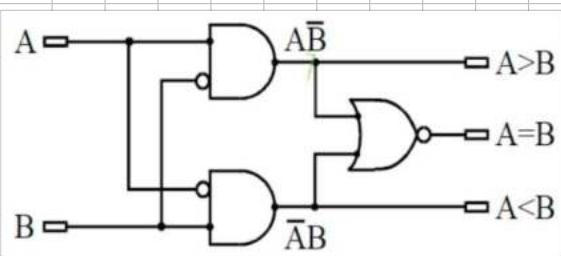


le porte XNOR danno 1 quando sono uguali

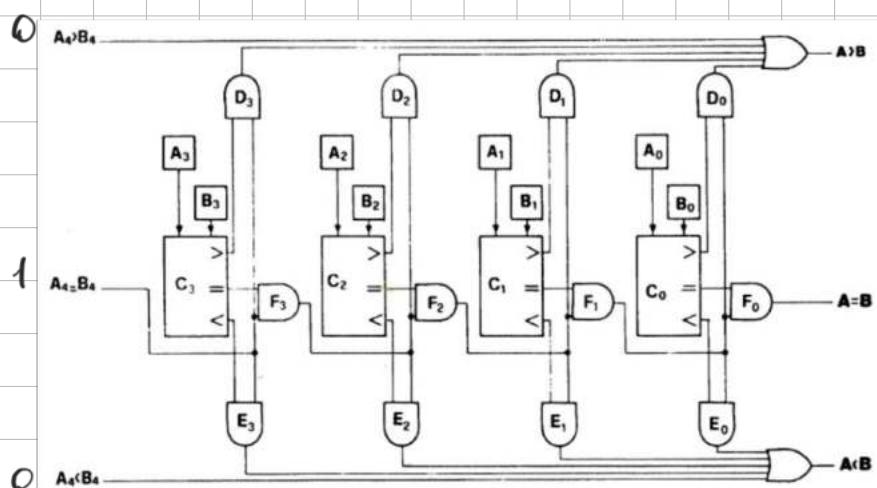
## COMPARATORE COMPLETO A 1-BIT

Il COMPARATORE COMPLETO ha 3 uscite:  $A=B$ ,  $A>B$ ,  $A<B$

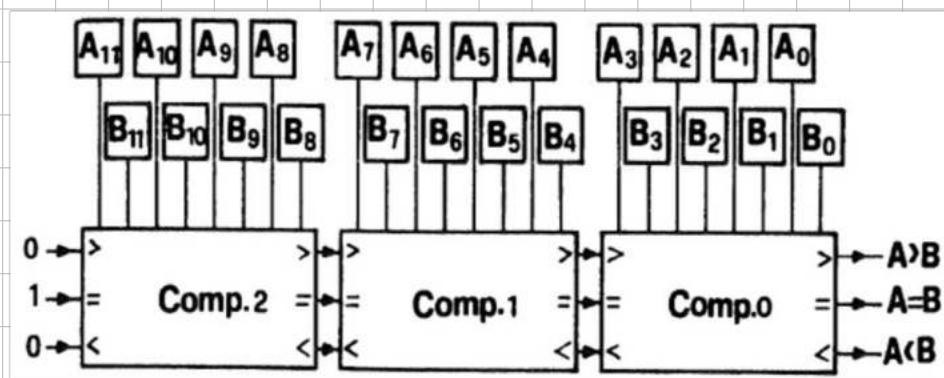
A	B	$A > B$	$A = B$	$A < B$
0	0	0	1	0
0	1	0	0	1
1	0	1	0	0
1	1	0	1	0



## COMPARATORE COMPLETO A 4-BIT



## COMPARATORE COMPLETO A 12-BIT



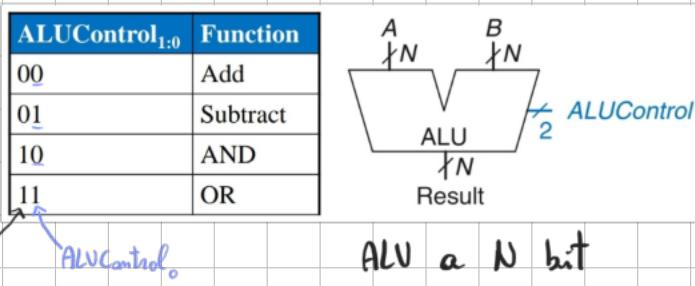


Non capisco  
Non capisco il perche' di questa cosa sincera  
In o' pretendere di vivere

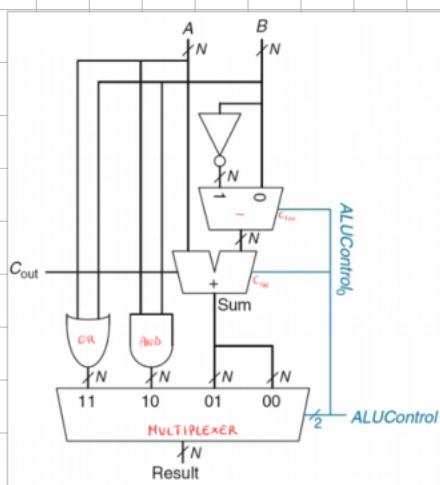
## ALU

L' ALU (aritmetico-logic unit) si occupa delle operazioni di

- ADDIZIONE
- SOTTRAZIONE
- AND (bit a bit)
- OR (bit a bit)



L' ALU e' cosi' composta



Esegue sempre tutte e 3 (somma e differenza contano come 1)

le operazioni, l' ALUControl<sub>1:0</sub> si occupa di selezionare l' output

Il caso 00 e' sfrutta ALUControl<sub>0</sub> (che e' 0) come C<sub>IN</sub>

dell'addizione (che ricordiamo parte da 0), senza utilizzare altre parti.

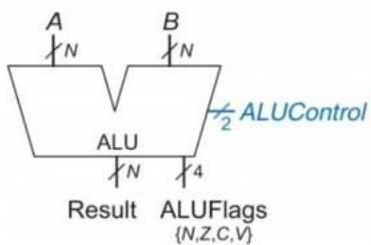
Analogamente accade con 01

$$C_{IN} = ALUControl_0$$

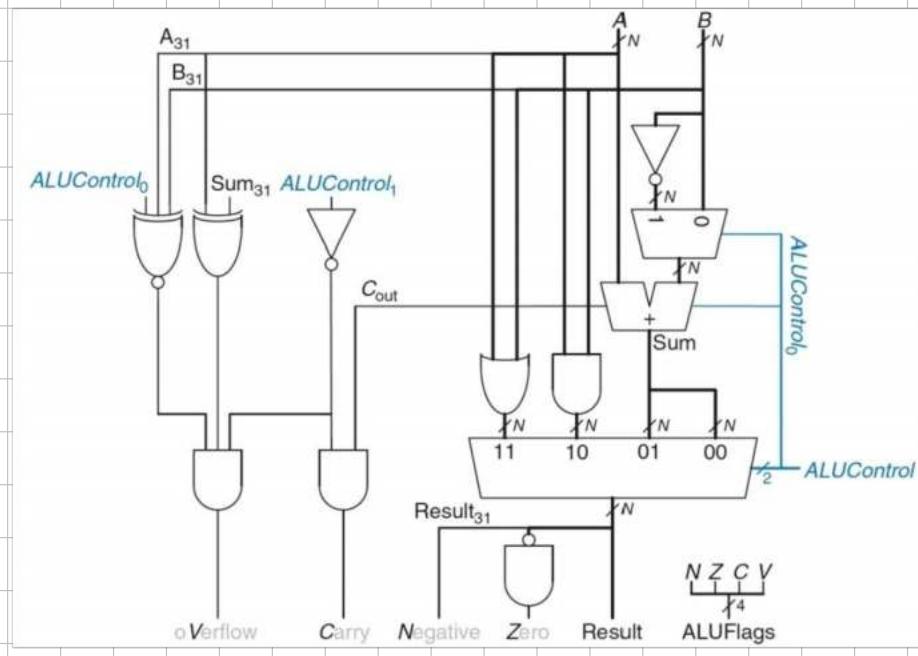
## ALU FLAGS

L'ALU completa produce dei valori **FLAGS** che ci danno delle informazioni aggiuntive sul calcolo svolto

Flag	Description
N	Result is Negative
Z	Result is Zero
C	Adder produces Carry out
V	Adder overflowed



Quindi l'ALU si presenta così:



● N: in rappresentazione complemento a 2, se il bit più significativo è 1, il numero sarà negativo. Quindi Result<sub>31</sub> (ultimo bit di un risultato a 32 bit) va direttamente a definire N (1=vero, 0=falso)

● Z: Basalmente sarà 1 se tutti i bit sono uguali a 0  
(AND di ogni bit negato)

● C: Sarà 1 se Cout = 1 e ALUControl<sub>1</sub> = 0 (quindi stiamo eseguendo un'addizione / sottrazione)

● V:

- Anzitutto ALUControl<sub>1</sub> deve essere 0 (addizione o sottrazione).
- Lo XOR verifica che il risultato e gli operandi abbiano segno opposto (condizione di overflow) confrontando il bit più significativo (Result<sub>31</sub> e A<sub>31</sub>)
- Lo XNOR Restituisce 1 se gli 1 sono in numero pari quindi 0 abbiamo ALUControl<sub>0</sub> = 0 e A<sub>31</sub> e B<sub>31</sub> = 1 (somma concorde) oppure ALUControl<sub>0</sub> = 1 e A<sub>31</sub> e B<sub>31</sub> discordi

## SHIFTERS

Gli **SHIFTERS** sono circuiti che si occupano di traslare i bit di un registro. Si dividono in:

### LOGICAL SHIFTER

trasla i bit a destra o sinistra riempendo gli spazi vuoti con 0

$$11001 \gg 2 = 00110$$

$$11001 \ll 2 = 00100$$

### ARITHMETIC SHIFTER

Funziona come il LOGICAL, riempendo gli spazi vuoti con il bit più significativo

$$11001 \ggg 2 = 11110$$

$$11001 \lll 2 = 00100$$

### ROTATOR

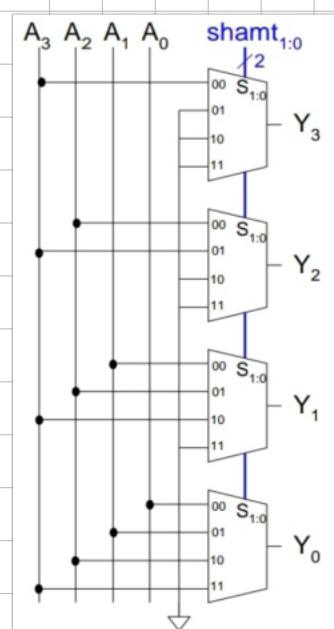
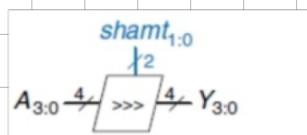
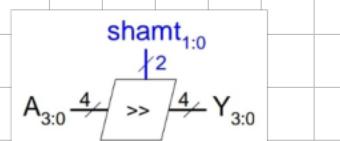
Effettua la "rotazione" del blocco

$$11001 \xrightarrow{\text{ROR}} 2 = 01110$$

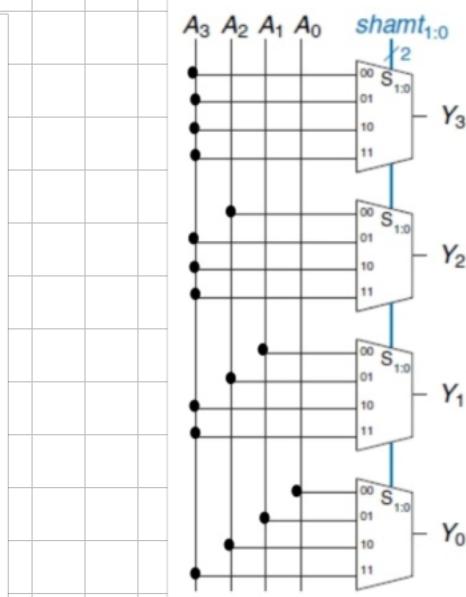
$$11001 \xleftarrow{\text{ROL}} 2 = 00111$$

## SHIFTERS DESIGN

Prendiamo un esempio con parole a 4 bit shiftate di 2 bit verso destra. **SHANT** (shift amount) indica il numero di bit da shiftare



LOGICAL



ARITHMETIC

## OSS

L' **ARITHMETIC** è così detto perché lo shift a SINISTRA equivale alla moltiplicazione per  $2^m$  e lo shift a destra equivale alla divisione per  $2^m$

$$00001 \lll 2 = 00100$$

$$(1 \cdot 2^2 = 4)$$

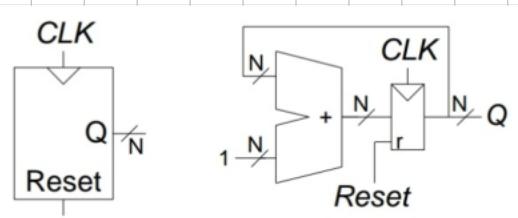
$$01000 \ggg 2 = 00010$$

$$(8 / 2^2 = 2)$$

## COUNTERS

E' un dispositivo che si incrementa al battere del clock

E' usato per eseguire cicli sui numeri e si usa come orologio digitale o come **PROGRAM COUNTER** per tenere traccia dell' istruzione da eseguire

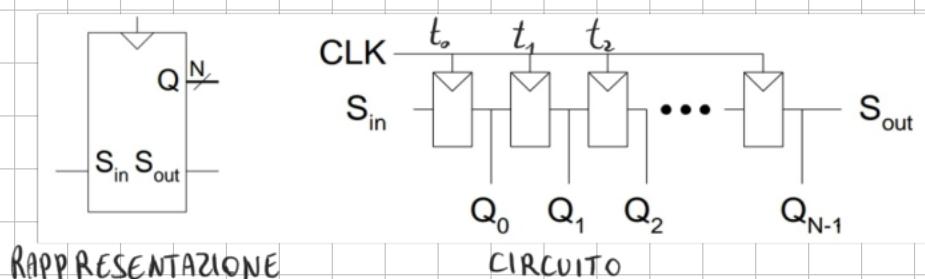


## SHIFT REGISTER

Lo **SHIFT REGISTER** trasla di un bit ad ogni battere del clock  
Ritorna  $S_{out}$  ad ogni clock.

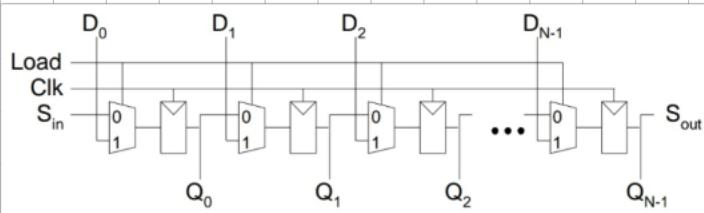
E' un **SERIAL-TO-PARALLEL CONVERTER**, ovvero

converte l' input seriale  $S_{in}$  in un output parallelo  $Q_{0:N-1}$



## SHIFT REGISTER CON LOAD PARALLELO

Esistono shift con un terzo valore di input detto **LOAD**



► Se LOAD=1 funziona come un registro a N bit

► Se LOAD=0 agisce da shift register

Può quindi convertire SERIALE  $\leftrightarrow$  PARALLELO

$S_{in}$  riceve il segnale che viene visualizzato in parallelo in  $Q_i$ :

Se LOAD=1 i valori di  $D_i = Q_i$ , passando da parallelo ( $Q_i$ ) a seriale ( $D_i$  visualizzati tutti insieme)

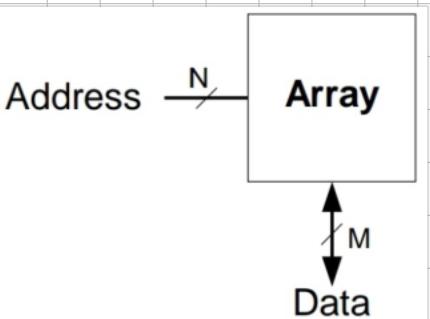
## MEMORY ARRAYS

Servono a memorizzare efficacemente grandi quantità di dati.

Si dividono in:

- DRAM (Dynamic random-access memory)
- SRAM (Static random-access memory)
- ROM (Read only memory)

In output avremo un dato a M-bit letto/scritto su un unico indirizzo a N bit



Tipicamente  $M=8$  e  $N=32$  o  $64$ .

Ogni byte (parola di 8 bit) ha il suo indirizzo.

Queste memorie vengono considerate come **ARRAY BIDIMENSIONALI** di **CELLE DI MEMORIA**.

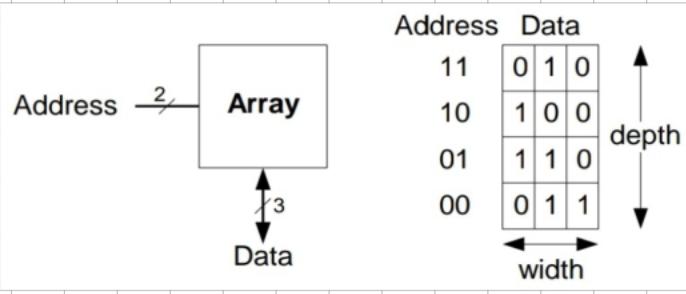
In particolare:

- Ogni cella memorizza un bit
- Con  $N$  bit di indirizzo e  $M$  bit data avremo:
  - $2^N$  righe e  $M$  colonne

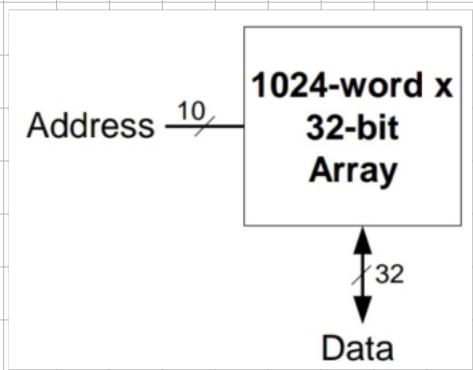
**DEPTH**: numero di righe (quindi parole)

**WIDTH**: numero di colonne (quindi lunghezza delle parole)

**DIMENSIONE DELL'ARRAY**:  $\text{DEPTH} \times \text{WIDTH} = 2^N \times M$



Ex.

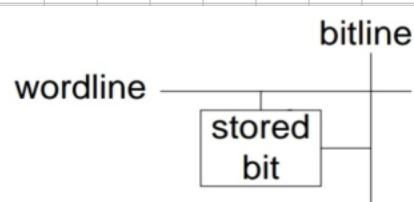


## CELLE DI MEMORIA

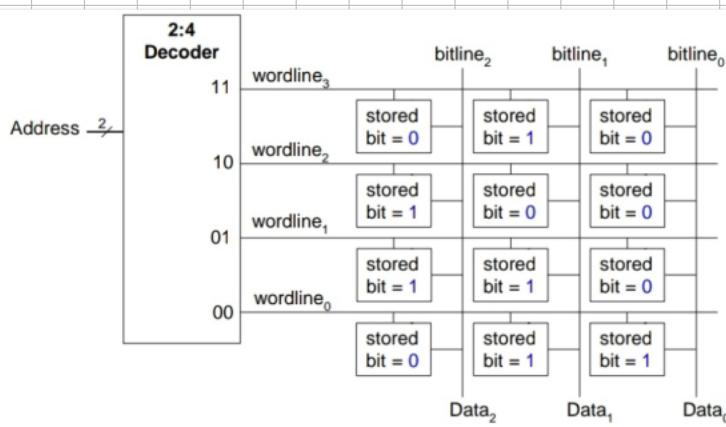
Sono composte da una WORDLINE e una BITLINE

Se la WORDLINE=1, questa "attiva" la parola e quindi tutte le bitline che la compongono

Ovviamente tutte le altre wordline delle parole non selezionate saranno = 0.



La wordline quindi agisce da enabler selezionando una riga nella memoria.



## MEMORIE ROM

Sono memorie **NON** volatili, più lente rispetto alle RAM.

Sono dette "read-only" perché originariamente verranno scritte

bruciando fusibili, rendendole non manipolabili.

Esempi di memorie ROM sono Drives, flash memories e Bios.

## MEMORIA RAM

E' una memoria **VOLATILE** (perde tutti i dati allo spegnimento della macchina). Le operazioni di scrittura/lettura sono più brevi rispetto alla memoria ROM.

E' detta "random-access" che significa che il tempo di accesso ad ogni cella è lo stesso.

Le memorie RAM si suddividono in: **DRAM** ed **SRAM**

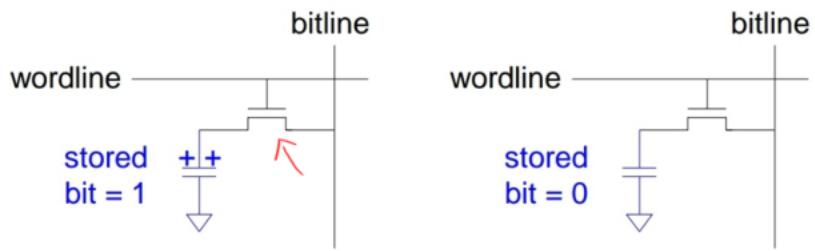
## DRAM

E' la memoria principale di un computer.

I **BIT DATA** sono immagazzinati in dei condensatori.

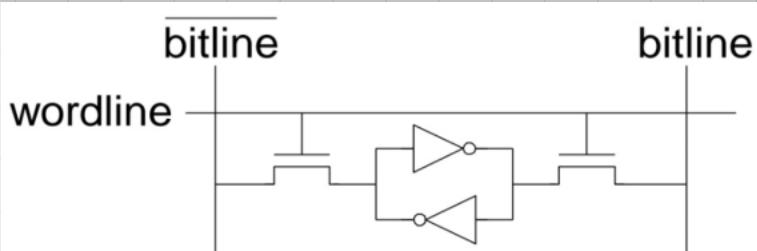
Per via della perdita di carica dei condensatori (quindi  $V_{DD}$  tende a tornare a 0), i valori conservati vanno periodicamente

**REFRESHATI**. Per questo è detta **DINAMICA**



Il condensatore è collegato alla bitline tramite un  
TRANSISTOR CHOS

## SRAH



BISTABILE

La SRAH è composta da un circuito BISTABILE, due bitline coniugate e due transistor. Il bistabile "blocca" i due valori delle bitline, che verrà letto o conservato se la wordline è 1.0

## DRAM vs SRAH

Le DRAM risultano essere più lente per via dei condensatori e dell'obbligato refresh dei dati.

Tuttavia sono più economiche (perché richiedono meno TRANSISTOR, componente più costosa)

Memory Type	Transistors per Bit Cell	Latency
flip-flop	~20	fast
SRAM	6	medium
DRAM	1	slow

I **flip-flop** compongono i registri, utilizzati dalla **CPU**.

Le **SRAM** compongono le memorie **CACHE**.

Le **DRAM** compongono la **MEMORIA CENTRALE**, comunemente **RAM**.

### DDR SDRAM

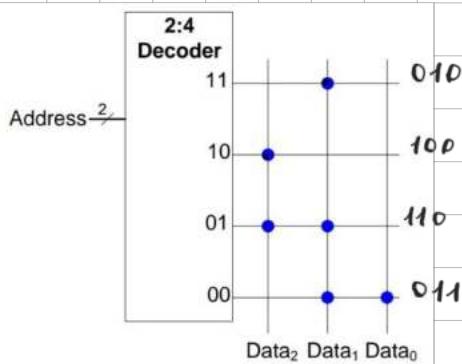
In questo tipo di memoria le operazioni di lettura/scrittura sono sincronizzate da un clock, sia quando questo passa da  $0 \rightarrow 1$  che viceversa. In tal modo la quantità di dati traspinta è doppia rispetto alle altre memorie RAM (che scrivono/leggono solo quando il clock passa da  $0 \rightarrow 1$ )

### ROM STORAGE

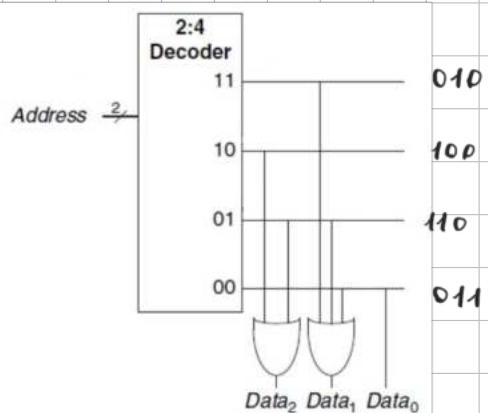
Queste memorie hanno i collegamenti tra bitline e wordline fisi, infatti non è possibile scrivere su queste memorie.

### OSS

Wordline orizzontali, bitline verticali



attraverso porte logiche



$$DATA_2 = A_1 \oplus A_0$$

$$DATA_1 = \overline{A}_1 + A_0$$

$$DATA_0 = \overline{A}_1 \overline{A}_0$$

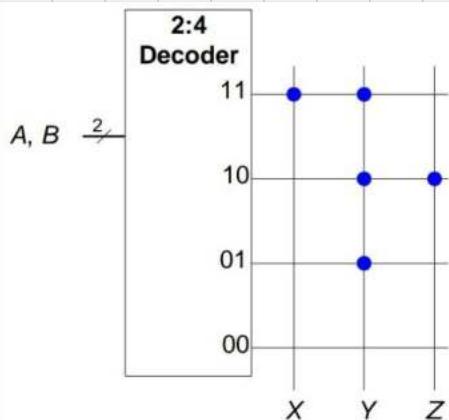
E.

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

$$X = AB$$

$$Y = A + B$$

$$Z = A\overline{B}$$



## ARCHITETTURA E MICROARCHITETTURA

Un' **ARCHITETTURA** rappresenta la struttura di un calcolatore dal punto di vista di un programmatore.

È definita da un set di **ISTRUZIONI** e **OPERANDI** che costituisce il **LINGUAGGIO MACCHINA**.

Al linguaggio macchina corrisponde un linguaggio **ASSEMBLY**, questo è più leggibile a noi umani rispetto al linguaggio macchina (composto da soli 0 e 1)

Una **MICROARCHITETTURA** definisce la disposizione specifica di registri, ALU, macchine a stati finiti, memorie etc. necessari per implementare un' architettura.

## OSS

**MICROARCHITETTURE** diverse possono realizzare stesse **ARCHITETTURE** (Intel e AMD con x86)

## PRINCIPI DI PROGETTAZIONE

L'architettura ARM segue i **PRINCIPI** di progettazione:

① REGULARITY SUPPORT SIMPLICITY

② MAKE THE COMMON CASE FAST

③ SMALLER IS FASTER

④ GOOD DESIGN DEMANDS GOOD COMPROMISE

### REGULARITY SUPPORT SIMPLICITY

Ogni istruzione è rappresentata da parole a 32 bit (anche se se ne richiedono di meno, questo semplifica la complessità)

I frammati sono per questo detti **CONSISTENTI**, questo facilita l'encoding in hardware.

### ESPRESSIONE

$$a = b + c$$

$$a = b - c$$

### ARM ASSEMBLY

ADD a,b,c

SUB a,b,c

### MAKE THE COMMON CASE FAST

L'architettura ARM comprende solo istruzioni **SEMPICI e VELOCI**

le istruzioni complesse sono spezzate in più operazioni più veloci  
così l'hardware non viene appesantito.

## ESPRESSIONE

$$a = b + c - d$$

## ARM ASSEMBLY

ADD t,b,c      ( $t = b + c$ )

SUB a,t,d      ( $a = t - d$ )

## OSS

Per questo principio le architetture si dividono in 2 tipi:

- RISC : poche e semplici istruzioni      (Es. ARM)
- CISC : molte istruzioni complesse      (Es. x86)

## SHALLER IS FASTER

I dati sono memorizzati:

- All'interno di un'istruzione

► COSTANTI

- In registri

► ARM ha solo 16 registri

► I registri sono più veloci della memoria

- In memoria

► Più lenta ma più capiente

► Si suddivide in: CACHE, CENTRALE, DI MASSA.

► le operazioni agiscono solo sui registri quindi tutti i dati devono essere trasferiti sui registri per elaborarli.

### GOOD DESIGN DEMANDS GOOD COMPROMISE

Per semplificare l'hardware è preferibile avere un numero ridotto di formati di ISTRUZIONI. Le microarchitetture ARM ne hanno 3:

- DATA PROCESSING (elaborano i dati)
- MEMORY, suddivise in
  - LOAD (carica dati da una memoria o un registro)
  - STORE (viceversa)
- BRANCH (un "salto" da un punto a un altro di un programma)

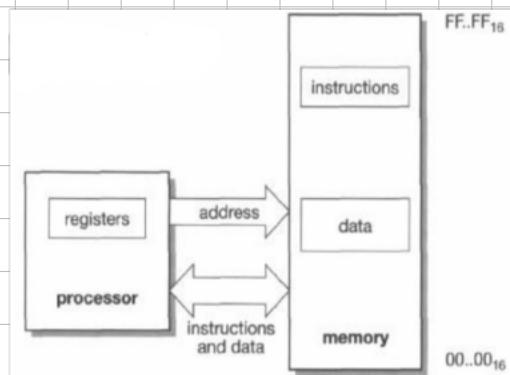
### PROCESSORE

Il **PROCESSORE** è un'automma a stati finiti che esegue operazioni indicate in memoria.

Lo **STATO** del sistema è definito dai valori contenuti nelle locazioni di memoria e da alcuni valori contenuti in alcuni registri (all'interno del processore stesso).

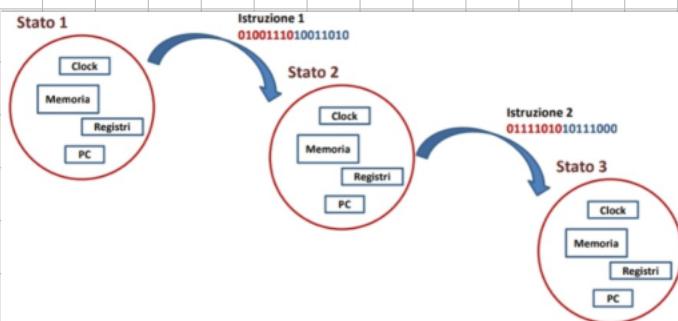
Ogni **ISTRUZIONE** definisce in che modo lo stato deve cambiare e

quale istruzione dev' essere eseguita successivamente



OSS

Una microarchitettura puo' quindi essere vista come un automa.



### VARIAZIONE DI STATO

Le variazioni sono regolate da un clock, in particolare i componenti (registri, memoria dati e memoria istruzioni, che operano tramite logica combinatoria) scrivono sul fronte alto del clock ( $0 \rightarrow 1$ ), quindi lo stato cambia solo su un fronte del clock.

La variazione viene regolata da degli **ABILITATORI** (che sono indirizzi, dati e un segnale di write enable) che devono essere

sestati prima dell'esecuzione di ogni operazione.

## MICROARCHITETTURE A CICLO SINGOLO

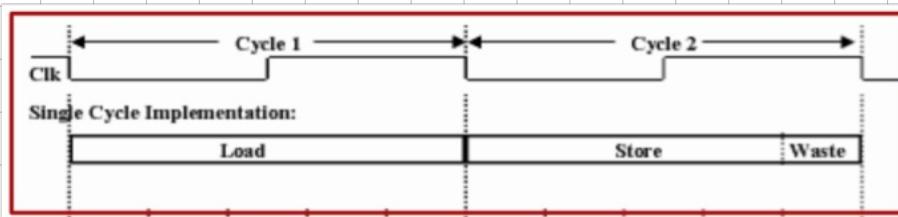
In questo tipo di microarchitettura l'intera istruzione è eseguita in un singolo ciclo del clock.

### ● PRO

- Semplificata da comprendere
- Unità di controllo molto semplice
- Non richiede stati non architettonici

### ● CONTRO

- Tempo pari a quello dell'istruzione più lenta
- Memoria **DATI** e memoria **ISTRUZIONI** separate



## MICROARCHITETTURE A CICLO MULTIPLO

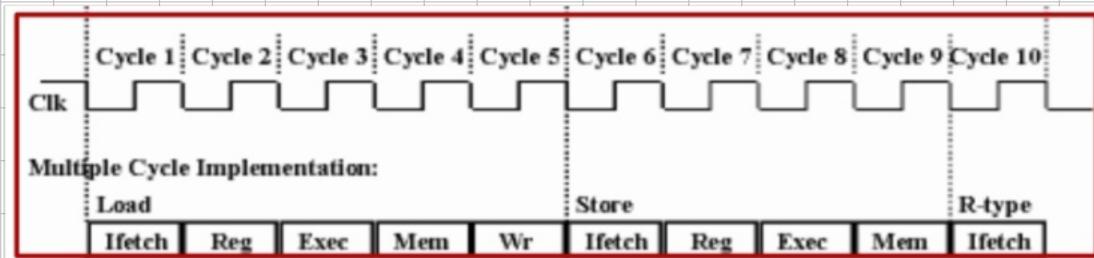
In questo tipo di microarchitettura l'istruzione è eseguita in più cicli brevi.

## ● PRO

- Ricuso dei componenti
- Diminuta variabile delle istruzioni
- Non richiede la separazione delle memorie (dati e istruzioni)

## ● CONTRO

- Richiede stati non architettonici
- Esegue un'istruzione per volta



## MICROARCHITETTURE CON PIPELINES

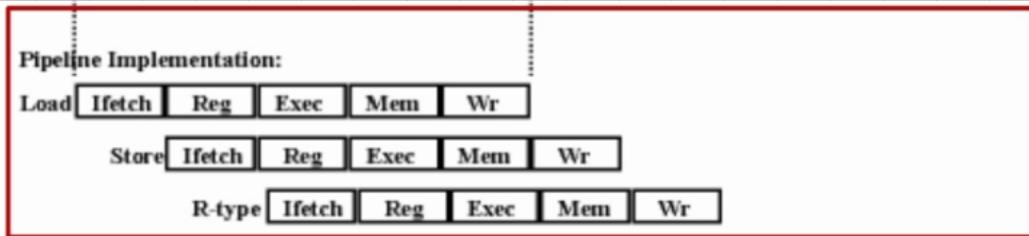
Sono ai ciclo multiplo in cui ogni istruzione viene divisa in più fasi e ad ogni ciclo mentre una fase viene eseguita, viene eseguita anche la prima fase dell'istruzione successiva

## ● PRO

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

## ● CONTRO

- Logica di controllo più complessa
- Richiede registri di pipeline



## MISURA DELLE PRESTAZIONI

Per misurare le prestazioni di un processore vengono eseguite una serie fissa di programmi detta **BENCHMARK** a seconda del tempo impiegato si può fare una stima sulle prestazioni.

Il **TEMPO DI ESECUZIONE** è dato da:

$$\text{TEMPO} = (\text{n° ISTRUZIONI}) \left( \frac{\text{CICLI}}{\text{ISTRUZIONE}} \right) \left( \frac{\text{SECONDI}}{\text{CICLO}} \right)$$

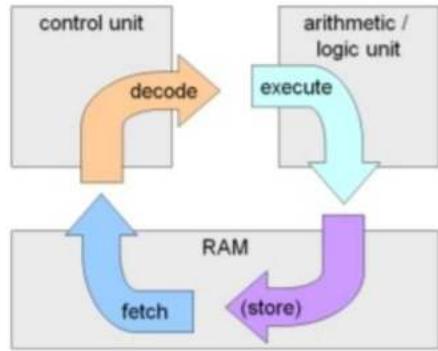
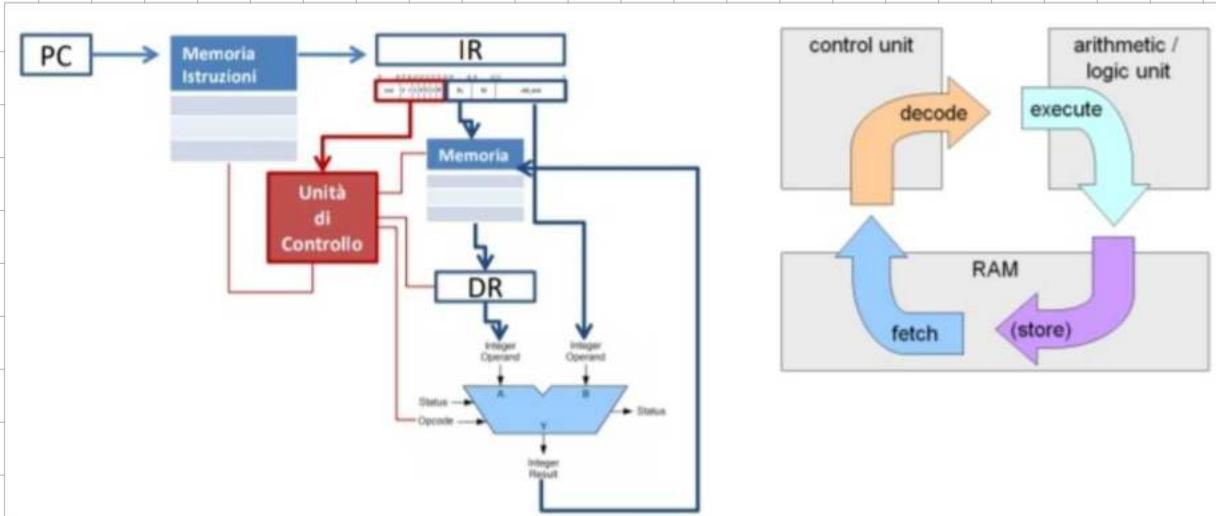
**CPI**   **CLOCK PERIOD**

OSS

La frequenza di clock corrisponde all'inverso del **CLOCK PERIOD**



# SCHEMA GENERALE DELLE MICROARCHITETTURE



- **FETCH:** Partendo dal registro contenente l'indirizzo dell'istruzione da eseguire, cerca in memoria tale istruzione e la carica nell'apposito registro.
- **DECODE:** Istruisce il DATAPATH su come operare
- **EXECUTE:** cercando nei registri gli argomenti dell'operazione, l'istruzione viene eseguita
- **STORE:** l'informazione elaborata viene conservata in memoria

## PROGETTAZIONE

I due componenti principali di una microarchitettura sono:

## ● **DATAPATH** (a 32 BIT)

Determina il flusso dei dati durante l'esecuzione di un'istruzione, opera su parole dati e contiene memoria, registri, l'ALU e i multiplexer.

## ● **UNITÀ DI CONTROLLO**

Riceve l'istruzione dal datapath e indica a questo come agire. Produce i valori di selezione dei multiplexer, i segnali d'abilitazione alla scrittura dei registri e della memoria.

## ELEMENTI DI STATO

La memoria si suddivide in 5 **ELEMENTI DI STATO**:

### ● PROGRAM COUNTER

### ● REGISTER FILE

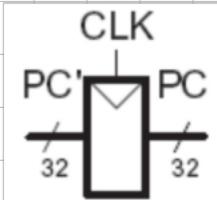
### ● STATUS REGISTER

### ● ISTRUCTION MEMORY } che coincidono con architetture a } ciclo multiplo

### ● DATA MEMORY

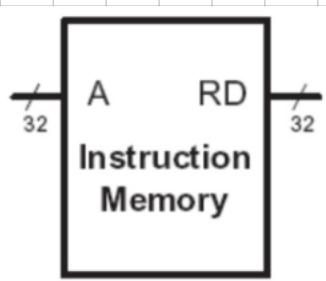
## PROGRAM COUNTER

E' un registro autonomo a 32 bit che memorizza l' istruzione corrente salvandone l' indirizzo al battere di ogni clock



## INSTRUCTION MEMORY

E' una memoria di sola lettura che contiene l' indirizzo dell' istruzione da eseguire



## FILE REGISTER

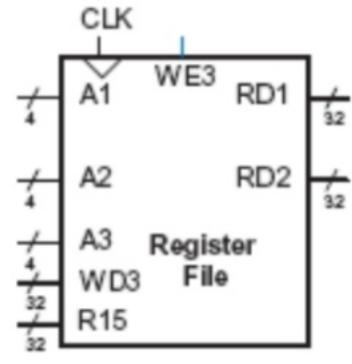
Consiste di 16 registri R<sub>0:15</sub> a 32 bit.

Svolgono funzioni diverse tra loro:

R13) STACK POINTER

R14) LINK REGISTER

R15) Proviene dal PC



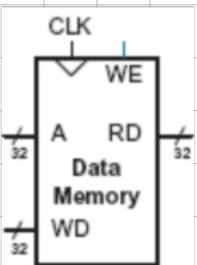
Esso e' costituito da 2 ingressi in **LETTURA** A<sub>1</sub> e A<sub>2</sub>  
e 1 in **SCRITTURA** A<sub>3</sub>

- In A<sub>1</sub> e A<sub>2</sub> arrivano gli indirizzi il cui contenuto verrà letto in RD1 e RD2.

Gli ingressi in lettura sono 2 perché le operazioni hanno sempre 2 argomenti (e un risultato), inoltre sono a 4 bit perché abbiamo solo 16 registri ( $2^4 = 16$ ).

- A<sub>3</sub> e WD3 servono a memorizzare il **risultato** nell'indirizzo A<sub>3</sub> contenente la parola WD3
- R15 serve a tenere aggiornato il PC.

### DATA MEMORY

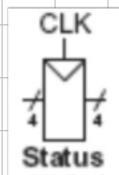


● È composta da una parola WD e un indirizzo A, sia per la lettura che per la scrittura.

● Il segnale WE (write enable) indica se WD dev'essere scritto (WE=1) in A oppure se deve solo essere letto (WE=0) il dato all'indirizzo A (letto nel registro RD)

## STATUS

Memorizza i flag N, Z, C, V forniti dall'ALU.



Serve inoltre a realizzare istruzioni condizionate nei linguaggi di programmazione ad alto livello.

## INDIRIZZI DI MEMORIA

Ogni BYTE ha il suo **INDIRIZZO DI MEMORIA**.

Usiamo parole a 32 BIT = 4 BYTE quindi gli indirizzi delle parole varranno di 4 (Byte) in 4.

L'indirizzo della m-esima parola è 4m

Per questo la memoria è detta **BYTE-ADRESABLE**

Byte address				Word address
13	12	11	10	:
F	E	D	C	00000010
B	A	9	8	0000000C
7	6	5	4	00000008
3	2	1	0	00000004
				00000000
MSB		LSB		

## ISTRUZIONI DI BASE

Le istruzioni di base di una microarchitettura ARM sono:

- DATA - PROCESSING

  - ADD-SUB, AND, OR

- ISTRUZIONI DI MEMORIA

  - LDR, STR

- ISTRUZIONI DI SALTO

  - B

## ISTRUZIONE LOAD

È un'istruzione di **MEMORIA**, si indica con **LDR** e carica dati dalla memoria.

LDR R0, [R1, #12]

ASSEMBLI

## ● INDIRIZZO (R0) :

► BASE ADDRESS + OFFSETT

$$\text{INDIRIZZO} = (R1 + 12)$$

## ● RISULTATO

► Il dato all' indirizzo ( $R1+12$ ) sarà caricato in R0

E.s.

Carica nel registro R3 la parola all' indirizzo 8

Word address	Data	Word number
...	...	...
00000010	C D 1 9   A 6 5 B	Word 4
0000000C	4 0 F 3   0 7 8 8	Word 3
00000008	0 1 E E   2 8 4 2	Word 2
00000004	F 2 F 1   A C 0 7	Word 1
00000000	A B C D   E F 7 8	Word 0
(b) Width = 4 bytes		

MOV R2, #0 (sposta in R2 il contenuto #0)

LDR R3, [R2, #8]

$$\text{INDIRIZZO} = (R2 + 8) = 8$$

$$R3 = 0x01EE2842$$

OSS

In alternativa si puo' indicare l'istruzione LDR usando come  
OFFSET un registro

● LDR R0, [R1, R2]

$R0 \leftarrow \text{mem32}[R1+R2]$

(carica in R0 il contenuto in memoria all'indirizzo  $R1+R2$ )

● LDR R0, [R1, R2, LSL #2]

$R0 \leftarrow \text{mem32}[R1+(R2 \cdot 2^2)]$

LSL = logical shift left = prodotto per  $2^m$

(carica in R0 il contenuto in memoria all'indirizzo  $R1+R2 \cdot 2^2$ )

## ISTRUZIONE STORE

E' un'istruzione di MEMORIA. Si indica con STR

ed e' l'istruzione duale (speculare) del LOAD.

La semantica ASSEMBLY e' speculare a LDR

STR R0 [R1, #12]

mem32 [R1+12] ← R0

Carica in (R1+12) il contenuto di R0

OSS

Analogamente avremo

● STR R1, [R0, R2]

mem32 [R0+R2] ← R1

● STR R0, [R1, R2, LSL #1]

mem32 [R1+R2·2] ← R0

E.

Memorizza il valore di R7 nella 21° parola di memoria

INDIRIZZO DI MEMORIA =  $21 \cdot 4 = 84 = 0x54$

MOV R5, #0

STR R7, [R5, #0x54]

↑

OFFSET IN ESADECIMALE

## ISTRUZIONI STRB E LDRB

Sono **VARIANTI** in cui "B" sta per **BYTE** ed eseguono  
un'istruzione per un singolo byte

OSS

E' importante conoscere come sono inoltrizzati i byte in una parola. Ci sono due modi:

- **LITTLE-ENDIAN**: la numerazione inizia dal byte **MENO** significativo
- **BIG-ENDIAN**: la numerazione inizia dal byte **PIU'** significativo

La più utilizzata è la **LITTLE ENDIAN**

Es.

Sia R2=0, si esegua LDRB R7 [R2,#8]

Verifichiamo R7 in little-endian e big-endian

Word address	Data	Word number	Big-Endian	Little-Endian
...	...	...	...	...
00000010	C D 1 9 A 6 5 B	Word 4	Byte Address	Byte Address
0000000C	4 0 F 3 0 7 8 B	Word 3	...	C
00000008	0 1 E E 2 8 4 2	Word 2	B	F E D C
00000004	F 2 F 1 A C 0 7	Word 1	A	B A 9 B
00000000	A B C D E F 7 8	Word 0	9	7 6 5 4
(b) Width = 4 bytes			MSB	LSB

B-E : 0x000000EE

L-E : 0x00000028

## CODIFICHE ISTRUZIONI DI MEMORIA

le istruzioni sono codificate in parole a 32 bit.

L'ARM ha 3 formati di istruzioni, uno per ogni tipo d'istruzione di base.

Vediamo bit per bit come vengono codificate le istruzioni:



### ● COND

Serve a codificare eventuali condizioni per l'esecuzione dell'istruzione

### ● OP

Indica a quale formato (e quindi quale tipo d'istruzione) stiamo facendo riferimento

Le istruzioni di **MEMORIA** sono codificate con 01

- **FUNCT**

Sono bit di controllo

- **Rm**

E' il registro del **BASE ADDRESS**

- **Rd**

E' il registro che indica il primo **ARGOMENTO** dell' istruzione (sorgente in STR e destinazione in LDR)

- **Src2**

Indica l' **OFFSET** dell' istruzione (che puo' essere **IMMEDIATE** o **REGISTER**)

- **T**

Determina la codifica di Src2.

Se  $T=0$  indicherà un **IMMEDIATE** (una **COSTANTE**, imm12)  
altrimenti indicherà un **REGISTER**

In quest' ultimo caso avremo :

► R<sub>m</sub>

L' **INDIRIZZO** del registro

► 0

Il 4° bit e' sempre 0

► sh

Indica un eventuale **SHIFT** o **ROTAZIONE**

► shift 5

Indica di quanti bit bisogna shiftare il register

## INDICIZZAZIONE

L' **INDICIZZAZIONE** indica come l' indirizzo viene explicitato  
in un' operazione.

Esistono 3 tipi:

● OFFSET

L' indirizzo e' calcolato sommando/sottraendo un offset  
al contenuto del registro di base.

In questo caso non vi sono variazioni del valore del  
**REGISTRO DI BASE**.

## ● PRE-INDEX

L'indirizzo è calcolato come nell'OFFSET ma viene scritto nel registro di base.

In questo caso il registro di base viene sostituito dal nuovo indirizzo calcolato.

## ● POST-INDEX

L'indirizzo corrisponde al contenuto del registro di base.

Il registro di base viene aggiornato come nel PREINDEX

Mode	ARM Assembly	Address	Base Register
Offset	LDR R0, [R1, R2]	R1 + R2	Unchanged
Pre-index	LDR R0, [R1, R2]!	R1 + R2	R1 = R1 + R2
Post-index	LDR R0, [R1], R2	R1	R1 = R1 + R2

E.s.

## ● OFFSET

LDR R1, [R2, #4]

R1 = mem32 [R2 + 4]

## ● PRE-INDEX

LDR R1, [R2, #16]!

R1 = mem32 [R2 + 16]

R2 = R2 + 16

## ● POST-INDEX

LDR R1, [R2], #8

R1 = mem32[R2]

R2 = R2 + 8

## CAMPPI DI FUNCT

Vediamo i 6 bit di FUNCT come operano



## ● L

L=1 istruzione di LOAD, altrimenti STORE

## ● B

B=1 carica SOLO un bit (LDRB, STRB), altrimenti tutta la parola

## ● P W

Indicano se si sta usando INDICIZZAZIONE

## ● U

$U=1$  l' OFFSET va ADDIZIONATO, altrimenti va

SOTTRATTO

Value	T	U
0	Immediate offset in Src2	Subtract offset from base
1	Register offset in Src2	Add offset to base

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

P	W	Indexing Mode
0	1	Not supported
0	0	Postindex
1	0	Offset
1	1	Preindex

### CAMPО SH

Il campo SH codifica il tipo di SHIFT.

Shift Type	sh
LSL	00 <sub>2</sub>
LSR	01 <sub>2</sub>
ASR	10 <sub>2</sub>
ROR	11 <sub>2</sub>

### OSS

- ASL manca perchè abbiamo notato che coincide con LSL
- ROL manche perchè è ottenibile tramite ROR.

In particolare  $ROL\ n = ROR\ 32 - n$

Per questo SHAMT = 5 bit, perchè  $2^5 = 32$  bit  
quindi possiamo codificare tutte le combinazioni.

E.s.

Codifichiamo la seguente istruzione assembly in linguaggio macchina.

STR R11,[R5], # -26

- OPERATION :  $\text{mem32[R5]} \leftarrow \text{R11} ; \text{R5} = \text{R5} - 26$  (postindex)
- COND :  $1110_2$  (esecuzione incondizionata)
- OP = 01 (istruzione di memoria)
- FUNCT =  $000000_2$
- $\bar{I}=0, P=0$  (immediate offset postindex)
- $V=0, B=0, W=0$  (sottrai l'offset, considera la parola, postindex)
- $L=0$  (store)
- $R_d = 11, R_m = 5, \text{imm12} = 26$  ( $\text{R11}, \text{R5}, 26$  l'immediate)

31:28	27:26	25:20	19:16	15:12	11:0
$1110_2$	$01_2$	$0000000_2$	5	11	26
cond	op	$\bar{I}$ PUBWL	Rn	Rd	imm12
<u>1110</u>	<u>01</u>	<u>000000</u>	<u>0101</u>	<u>1011</u>	<u>0000</u> <u>0001</u> <u>1010</u>
E	4	0	5	B	0 1 A

ISTRUZIONE = E405B01A<sub>16</sub>

LDR R3, [R4, R5]

● OPERATION  $R3 \leftarrow \text{mem}[R4+35]$

● COND:  $1110_2$  (esecuzione incondizionata)

● OP =  $01_2$  (istruzione di memoria)

● FUNCT =  $111001_2$  ( $57_{16}$ )

●  $\bar{T} = 1$ , P = 1 (register offset, offset indexing)

● V = 1, B = 0, W = 0 (aggiunge l'offset, considera la parola, postindex)

● L = 1 (load)

●  $R3 = 3$ ,  $R_m = 4$  (R3, R4)

$R_m = 5$ , shamt5 = 0, sh = 00 (registro R5 non shiftato)

31:28	27:26	25:20	19:16	15:12	11:0
$1110_2$	$01_2$	$111001_2$	4	3	00000 00 0 0101 <sub>2</sub>
cond	op	$\bar{I}PUBWL$	Rn	Rd	SHAMT5 SH 0 R <sub>m</sub>

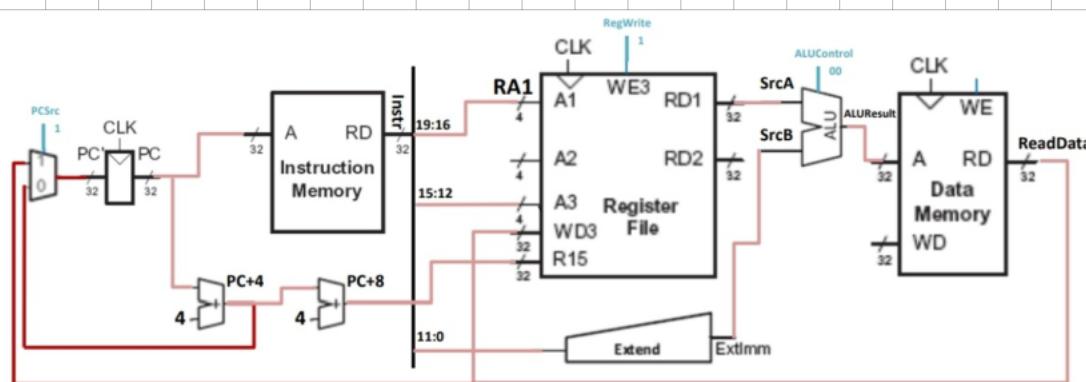
1110	01	11	1001	0100	0011	0000	0000	0101
E	7	3	4	3	0	0	0	5

ISTRUZIONE =  $0x E7943005$

## DATA PATH LDR

Vediamo il flusso dei dati dell'istruzione **LDR** con un offset **IMMEDIATE**

**LDR Rd [R<sub>m</sub>, imm12]**



**DATA PATH**



**INSTRUZIONE**

### ① FETCHING

Il **PC** contenente l'istruzione da eseguire è collegato

all'**INSTRUCTION MEMORY** che leggerà l'istruzione

rappresentata dall'etichetta **INSTR**

### ② Dobbiamo ora calcolare l'indirizzo di memoria

(nella memoria dati) in cui andare a prendere il dato

(operazione di **LOAD**). Come ci dice l'espressione **ASSEMBLY**

Quest'indirizzo di memoria e' dato da  $R_m + imm.12$

Carichiamo in A1 i bit 13:16 dell' istruzione che indicano la nostra nostra  $R_n$ .

③ Abbiamo caricato  $R_m$ , carichiamo ora l'**OFFSET** imm.12.

Questo e' contenuto nei bit 11:00 dell' istruzione

ma noi abbiamo bisogno di parole a 32 bit, non 12.

Per questo, il nostro  $Src2$  (l'immmediate) passa per un **EXTENDER** che aggiunge 20 0 così da avere una parola a 32 bit.

Ora l'immmediate e il base address hanno lo stesso formato

④ Possiamo finalmente sommare l'**OFFSET** e il **BASE ADDRESS**

Quindi carichiamo nell'**ALU** (con  $ALUControl = 00$  per indicare la somma) **R1** dal Register file (per il base address) e

**EXTMUL** (l'offset esteso) sottoforma di operandi nominati **SrcA** (base register) e **SrcB** (offset)

( $ALUControl$  verrà settato dalla **CU** che vedremo più avanti)

Otterremo quindi un **ALUResult**

⑤ L' ALUResult viene caricato come input A nel DATA

MEMORY che caricherà in RD il valore contenuto nell' indirizzo A

⑥ A questo punto il valore di output READDATA ottenuto dal data memory viene caricato come input in WD3.

Il REGWRITE vale 1 per abilitare la scrittura in A3 che contiene il valore di Rd ovvero i bit 15:12 di instr.

⑦ Ora che abbiamo completato l'operazione dobbiamo calcolare la successiva istruzione.

Siccome le istruzioni sono salvate ogni 4 byte (32 bit) dobbiamo aggiungere all' istruzione calcolata PC 4 byte.

Questo viene fatto grazie a un SOMMATORE che ci dà:

PC+4 caricato in input nel PC.

Siccome in RD15 (register file) carichiamo l'operazione successiva a quella del PC, sommiamo altri 4 byte e carichiamo il PC+8 in RD15

⑧ Si crea una complicazione se Rd è proprio R15.

Allora ReadData viene collegato a un multiplexer gestito da un segnale **PCSrc** che vale:

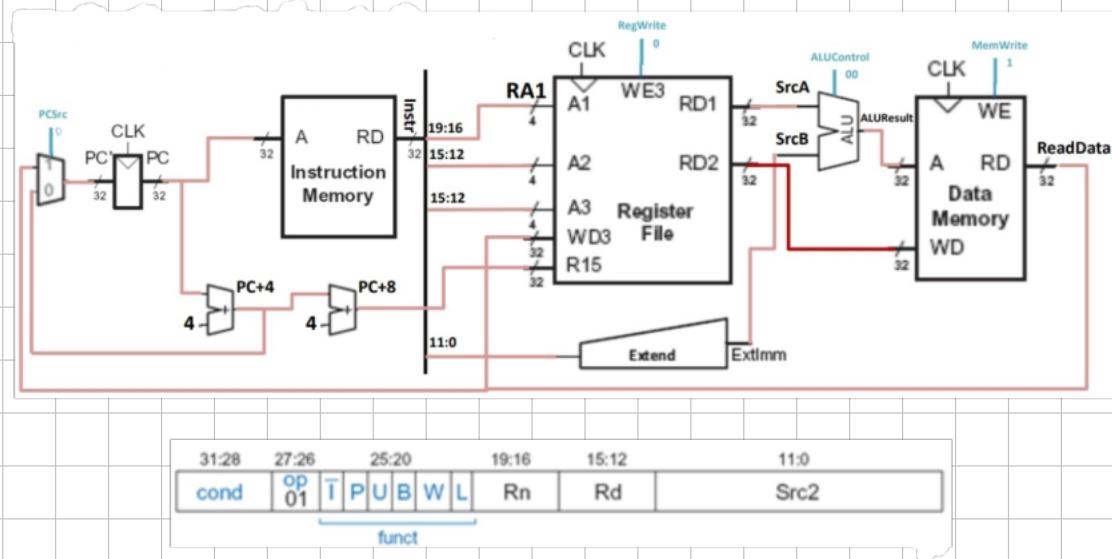
► 0 se Rd non è R15 (PCplus)

► 1 se Rd è R15 (ReadData)

## DATAPATH STR

Estendiamo il datapath precedente in modo che possa eseguire anche **STR**

**STR Rd [R<sub>n</sub>, imm12]**



Saltiamo le parti uguali

① Dobbiamo calcolare stavolta l'indirizzo in cui **SCRIVERE**

la procedura per il calcolo è analoga (punti 1/4)

② Siccome Rd sarà di scrittura e non di lettura  
lo inseriremo come input anche in A2 del register file, letto  
in RD2 che porterà il contenuto a WD

③ Nel data memory il segnale **MEMWRITE** sarà 1  
per indicare che i dati devono essere scritti in memoria  
(WD viene scritto in A)

④ Si osservi che montante Rd sia caricato anche in A3  
e ReadData è trasmesso in WD3, questo non produce  
effetti sul file register perché **RegXrite=0**  
disabilitando la scrittura su registro (che **non** viene modificato,  
a differenza della memoria)

⑤ Poiché il registro **NON** viene modificato, A15 non può  
essere aggiornato, quindi **PCSrc** sarà sempre 0

## ISTRUZIONI DI DATA-PROCESSING

Ecco delle istruzioni che processano dati salvati in registri e salvano il risultato in altri registri.

<b>MOV</b>	Move a 32-bit value	<b>MOV Rd, n</b>	Rd = n
<b>MVN</b>	Move negated (logical NOT) 32-bit value	<b>MVN Rd, n</b>	Rd = ~n
<b>ADD</b>	Add two 32-bit values	<b>ADD Rd, Rn, n</b>	Rd = Rn+n
<b>ADC</b>	Add two 32-bit values and carry	<b>ADC Rd, Rn, n</b>	Rd = Rn+n+C
<b>SUB</b>	Subtract two 32-bit values	<b>SUB Rd, Rn, n</b>	Rd = Rn-n
<b>SBC</b>	Subtract with carry of two 32-bit values	<b>SBC Rd, Rn, n</b>	Rd = Rn-n+C-1
<b>RSB</b>	Reverse subtract of two 32-bit values	<b>RSB Rd, Rn, n</b>	Rd = n-Rn
<b>RSC</b>	Reverse subtract with carry of two 32-bit values	<b>RSC Rd, Rn, n</b>	Rd = n-Rn+C-1
<b>AND</b>	Bitwise AND of two 32-bit values	<b>AND Rd, Rn, n</b>	Rd = Rn AND n
<b>ORR</b>	Bitwise OR of two 32-bit values	<b>ORR Rd, Rn, n</b>	Rd = Rn OR n
<b>EOR</b>	Exclusive OR of two 32-bit values	<b>EOR Rd, Rn, n</b>	Rd = Rn XOR n
<b>BIC</b>	Bit clear. Every '1' in second operand clears corresponding bit of first operand	<b>BIC Rd, Rn, n</b>	Rd = Rn AND (NOT n)
<b>CMP</b>	Compare	<b>CMP Rd, n</b>	Rd-n & change flags only
<b>CMN</b>	Compare Negative	<b>CMN Rd, n</b>	Rd+n & change flags only
<b>TST</b>	Test for a bit in a 32-bit value	<b>TST Rd, n</b>	Rd AND n, change flags
<b>TEQ</b>	Test for equality	<b>TEQ Rd, n</b>	Rd XOR n, change flags
<b>MUL</b>	Multiply two 32-bit values	<b>MUL Rd, Rm, Rs</b>	Rd = Rm*Rs
<b>MLA</b>	Multiple and accumulate	<b>MLA Rd, Rm, Rs, Rn</b>	Rd = (Rm*Rs)+Rn

## ISTRUZIONI LOGICHE

Prendiamo il seguente registro di riferimento per gli operandi:

<b>R1</b>	0100 0110	1010 0001	1111 0001	1011 0111
<b>R2</b>	1111 1111	1111 1111	0000 0000	0000 0000

I risultati per le istruzioni logiche saranno:

AND R3, R1, R2	<b>R3</b>	0100 0110	1010 0001	0000 0000	0000 0000
ORR R4, R1, R2	<b>R4</b>	1111 1111	1111 1111	1111 0001	1011 0111
EOR R5, R1, R2	<b>R5</b>	1011 1001	0101 1110	1111 0001	1011 0111
BIC R6, R1, R2	<b>R6</b>	0000 0000	0000 0000	1111 0001	1011 0111
MVN R7, R2	<b>R7</b>	0000 0000	0000 0000	1111 1111	1111 1111

### ● AND

Esegue l' AND bit a bit

### ● ORR

Esegue l' OR bit a bit

### ● EOR

Esegue lo XOR bit a bit

### ● BIC (Bit Clear)

Se R2 e' 1 riporta 0, altrimenti riporta R1

### ● MVN (Move and NOT)

Ha solo un argomento (Rm) e un Rd.

Sposta R2 (o R1) in R7 e lo nega bit a bit

### OSS

- Le istruzioni AND e BIC sono utili per "**MASCHERARE**" bit
- L' istruzione OR e' utile per **COMBINARE** byte fields

## ISTRUZIONI DI SHIFTING

Queste sono:

### ● LSL

LSL R0, R7, #5 ;  $R0 = R7 \ll 5$

### ● LSR

LSR R3, R2, #31;  $R3 = R2 \gg 31$

### ● ASR

ASR R9, R11, #4 ;  $R9 = R11 \ggg 4$

ROR

### ● ROR R8, R1, #3 ; $R8 = R1 \text{ ROR } 3$

Ricordiamo che:

- Lo shift aritmetico a sinistra opera come lo shift logico
- Per effettuare una rotazione di  $n$  bit a sinistra possiamo effettuare una rotazione di  $32-n$  bit a destra.

- Lo shift di valore a sinistra di  $n$  bit equivale al prodotto per  $2^n$ . Analogamente a destra avremo la divisione.

	R5	1111 1111	0001 1100	0001 0000	1110 0111
Assembly Code		Result			
LSL R0, R5, #7	R0	1000 1110	0000 1000	0111 0011	1000 0000
LSR R1, R5, #17	R1	0000 0000	0000 0000	0111 1111	1000 1110
ASR R2, R5, #3	R2	1111 1111	1110 0011	1000 0010	0001 1100
ROR R3, R5, #21	R3	1110 0000	1000 0111	0011 1111	1111 1000

Se come **OFFSET** abbiamo un registro usciremo il valore dell' offset per stabilire di quante posizioni eseguire l'istruzione

	R8	0000 1000	0001 1100	0001 0110	1110 0111
	R6	0000 0000	0000 0000	0000 0000	0001 0100
					= 20 <sub>10</sub>
Assembly code		Result			
LSL R4, R8, R6	R4	0110 1110	0111 0000	0000 0000	0000 0000
ROR R5, R8, R6	R5	1100 0001	0110 1110	0111 0000	1000 0001

## ISTRUZIONI DI MOLTIPLICAZIONE

Queste sono:

### ● MUL

Esegue la moltiplicazione 32x32 bit, con risultato a 32 bit

MUL R1, R2, R3 ; R1 = (R2 × R3)<sub>31:0</sub>

### ● UMUL

Esegue la moltiplicazione **UNSIGNED** 32x32 bit con risultato a 64 bit (salvato in 2 registri)

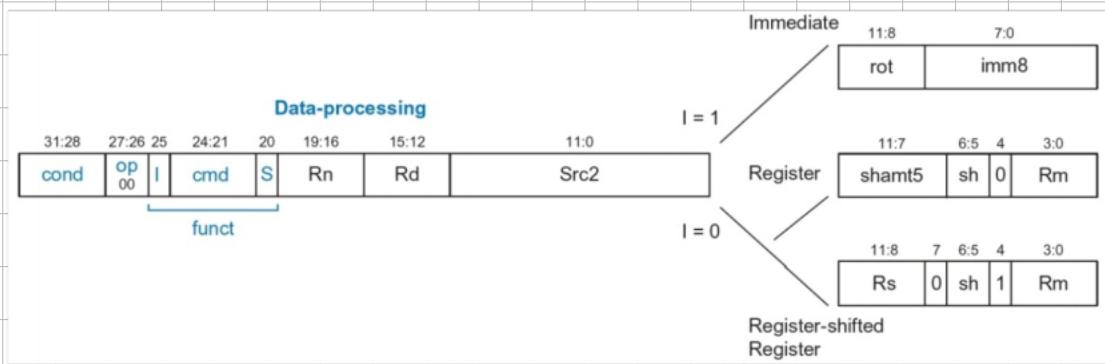
UMULL R1,R2,R3,R4 ;  $\{R1, R4\} = R2 \times R4$  (unsigned)

### SMULL

Esegue la moltiplicazione SIGNED 32x32 bit con risultato a 64 bit (solvato in 2 registri)

SMULL R1,R2,R3,R4 ;  $\{R1, R4\} = R2 \times R4$  (signed)

## FORMATO ISTRUZIONI DI DATA-PROCESSING



E' il formato piu' comune

Il 1° operando SORGENTE e' un REGISTRO

Il 2° operando sorgente puo' essere un REGISTRO o un IMMEDIATE.

La DESTINAZIONE e' sempre un REGISTRO

## ● OPERANDI

- R<sub>m</sub> : Registro sorgente
- Src2: Secondo registro sorgente o immediate
- Rd : Registro di destinazione

## ● CAMPI DI CONTROLLO

- cond: specifica l'esecuzione condizionale in base ai flag
- op: 00 è l'opcode per istruzioni di DATA PROCESSING
- funct: specifica il tipo di funzione da eseguire

FUNCT 25:20

È a sua volta suddiviso in:

- cmd: indica l'istruzione specifica da eseguire

0100<sub>2</sub> sta per ADD

0010<sub>2</sub> sta per SUB

● I

I=0  $\Rightarrow$  Src2 è un registro

I=1  $\Rightarrow$  Src2 è un immediate

● S

Indica se le **CONDITION FLAGS** dell' ALU devono essere aggiornate

S=0 non le considera

S=1 considera le flags

OSS

Di base istruzioni come ADD e SUB non prevedono l'aggiornamento delle flags. Tuttavia si puo' farlo con quest'aggiornamento inserendo (in ASSEMBLY) una S all'istruzione

E.s.

ADDS R8, R1, R4 or CMPS R3, #10 (con S=1)

SRC 2

Puo' essere:

● IMMEDIATE

► ROT 11:8

Indica un'eventuale rotazione (che va moltiplicata x2)

► IMM8 7:8

OFFSET UNSIGNED

## ● REGISTER

► SHAMT<sub>5</sub> 11:7

Indica l'ammunt di un eventuale shift

► SH 6:5

Indica il tipo di shift

► O 4

Il 4° bit indica il tipo di registro (è shiftato con un registro)

► RH 3:0

Secondo operando

## ● REGISTER SHIFTED CON UN REGISTER

► RS 11:8

Indica il registro che ci indica lo shift (come uno shamt)

► O 7

Il 7° bit vale 0

► 1 4

Il 4° bit in questo caso vale 1

E.

● ADD R0, R1, #42

► COND:  $1110_2$  (incondizionata)

► OP =  $00_2$  (istruzione di data-processing)

► CMD =  $0100_2$  (codifica ADD)

► I = 1 (src immediate)

► S = 0 (non vanno aggiornate le flags)

► RD = 0 (destinazione)

► RN = 1

► IMM8 = 42 (OFFSET)

► ROT = 0

31:28	27:26	25	24:21	20	19:16	15:12	11:8	7:0
$1110_2$	$00_2$	1	$0100_2$	0	1	0	0	42
cond	op	I	cmd	S	Rn	Rd	shamt5	sh Rm
1110	/00	1	0100	0/0001	/0000	/0000	/0010	1010

E 2 8 1 0 0 2 A

0xE281002A

● SUB R2, R3, #0xFF0

► COND= 1110<sub>2</sub> (incondizionata)

► OP= 00<sub>2</sub> (istruzione di data-processing)

► CMD= 0010<sub>2</sub> (codifica SUB)

► I= 1 (Src2 immediata)

► S= 0 (non verranno aggiornate le flags)

► RD= 2 (destinazione)

► RN= 3

► IMM8= 0xFF0 (OFFSET)

Per produrre IMM8=FF0 dobbiamo ruotare a sinistra di 4

0x FF0 = 1111 1111 0000

partendo da soli 8 bit di IMM8, saremo 1111 1111 che  
rotati di 4 bit a sinistra daranno 1111 1111 0000

► ROT=14 (ruotare a sx di 4= ruotare a destra di 32-4, ROT  
si moltiplica x2)

● ADD R5, R6, R7

- COND =  $1110_2$  (incondizionata)
- OP =  $00_2$  (istruzione di data-processing)
- CMD =  $0100_2$  (codifica ADD)
- I = 0 (Src2 registro)
- S = 0 (non vorremo aggiornare le flags)
- RD = 5 (destinazione)
- RN = 6
- RH = 7
- SHAMT = 0 (shift amount 0)
- SH = 0 (R6 non shiftato)

31:28	27:26	25	24:21	20	19:16	15:12	11:7	6:5	4	3:0
$1110_2$	$00_2$	0	$0100_2$	0	6	5	0	0	0	7
cond	op	I	cmd	S	Rn	Rd	shamt5	sh	Rm	
1110	/00	00	0100	0/0110	0101	/00000	00	00	0111	

0xE0865007

● ORR R9, R5, R3, LSR #2

► OPERATION :  $R9 = R5 \text{ OR } (R3 \ggg 9)$

► COND:  $1110_2$  (incondizionata)

► OP =  $00_2$  (istruzione di data-processing)

► CMD =  $1100_2$  (codifica ORR)

► I = 0 (Src2 registro)

► S = 0 (non vanno aggiornate le flags)

► RD = 9 (destinazione)

► RN = 5

► RH = 3

► SHAMT = 2 (shift amount)

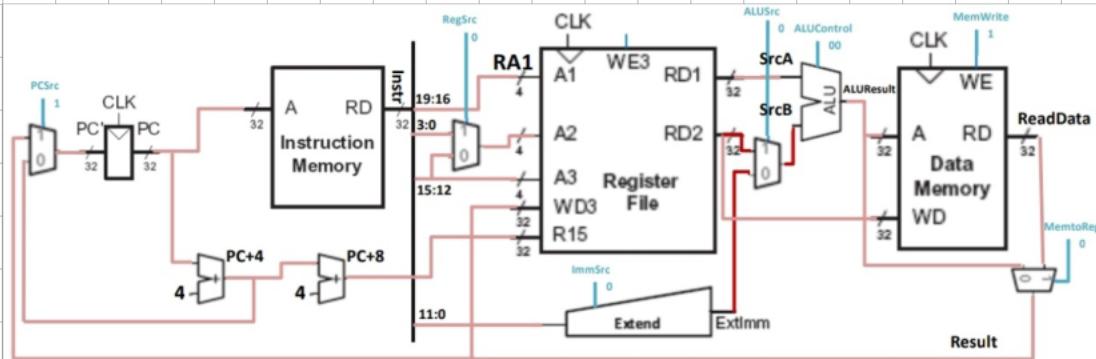
► SH =  $01_2$  (LSR)

31:28	27:26	25	24:21	20	19:16	15:12	11:0	11:7	6:5	4	3:0
cond	op 00	I	cmd	S	Rn	Rd	Src2	Register	shamt5	sh	Rm
funct											
1110	000	1100	0	0101	1001			I = 0	00010	010	0011

0xE1859123

## DATAPATH ADD, SUB, AND, ORR

Estendiamo il datapath alle seguenti istruzioni utilizzando la modalità immediata.



### ● IMM SRC

Le istruzioni di data-processing usano costanti a 8 bit quindi l'extend riceve un segnale aggiuntivo:

► IMM SRC = 0 estendi INSTR<sub>7:0</sub> da 8 a 32 bit

► IMM SRC = estendi INSTR<sub>11:0</sub> da 12 a 32 bit (LDR, STR)

### ● MEMTOREG

Siccome abbiamo operazioni aritmetiche, l' ALUResult non per forza indica indirizzi di memoria, bensì potrebbe

essere il risultato da dover scrivere in un registro (andando quindi in WD3). Per ovviare a questo problema inseriamo un multiplexer 2:1 HEMTOREG che avrà in input ALUResult e

ReadData e:

- HEMTOREG=0 input preso da ALUResult (istruzione di data-processing)
- HEMTOREG=1 input preso da ReadData (LDR,STR)

### ● REGSRC

Le istruzioni di data-processing con secondo argomento da registro ricevono quest'ultimo da RM, quindi A2

può essere da Rd per STR e LDR e da RM per data-processing

con secondo argomento un registro. Per questo inseriamo un

multiplexer che da in output il segnale ad A2 che varrà

- REGSRC=0 carica in A2  $R_m = \text{INSTR}_{3:0}$  (data processing)
- REGSRC=1 carica in A2  $R_d = \text{INSTR}_{15:12}$  (STR,LDR)

### ● ALUSRC

Di conseguenza dovremo indicare all'ALU se il secondo operando

varrà un registro (letto in RD2) o una costante proveniente dall'estend

Inseriamo quindi un multiplexer collegata all'ALU che porterà:

► ALUSRC=0 carica ExtImm (costante)

► ALUSRC=1 carica RD2 (registro)

## ESECUZIONE CONDIZIONATA

I primi 4 bit dell'istruzione sono bit di **CONDIZIONE**.

Queste istruzioni condizionate permettono l'esecuzione di if, then etc...

Le **CONDITION FLAG** sono:

① **MEMORIZZATE** nel **CURRENT PROGRAM STATUS REGISTER** (CPSR)

② **SETTATE** da un'istruzione

## ISTRUZIONE DI COMPARAZIONE : CMP

CMP R5,R6

Esegue R5-R6 senza salvarne il risultato.

Aggiorna i flag nel CPSR se R5-R6 soddisfa le condizioni di Z,N,C,V.

Flag	Name	Description
N	Negative	Instruction result is negative
Z	Zero	Instruction results in zero
C	Carry	Instruction causes an unsigned carry out
V	Overflow	Instruction causes an overflow

## SUFFISSO S

Il suffisso **S** (posto dopo un'istruzione in ASSEMBLY) indica che, in base al risultato ottenuto dall'istruzione, elevano essere aggiornati i flag.

## ISTRUZIONI CONDIZIONATE

le istruzioni condizionate in ASSEMBLY sono indicate da un suffisso detto **CONDITION MNEMONIC**

CMP R1, R2

SUB NE R3, R5, R8

NE ad esempio esegue l'operazione SUB solo se  $Z = 0$

Ricorda che il flag viene aggiornato dall'istruzione precedente  
in questo caso CMP.

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

Es.

- Immaginiamo di eseguire un CMP tra due valori e vediamo le condizioni flag con eventuali condizioni

$A = 1001_2$

UNSIGNED

g

SIGNED

-7

$B = 0010_2$

2

2

$$A - B = \begin{array}{r} 1001 \\ - 0010 \\ \hline 1011 \end{array}$$

NZCV = 0011<sub>2</sub>

La condizione HS e' verificata.

La condizione GE non e' verificata.

● CMP R5 R9

R5-R9 e' setta le flags

SUBEQ R1, R2, R3

EQ: esegui se R5=R9 ( $Z=1$ )

ORRM1 R4, R0, R9

M1: esegui se R5-R9 e' negativo ( $N=1$ )

Sia  $R5 = 17$  e  $R9 = 23$

CMP esegue  $17-23 = -6$  aggiornando come flags:

$N=1, Z=0, C=0, V=0$

SUBEQ non verrà eseguita ( $Z=0$ ), ORRM1 verrà eseguita ( $N=1$ )

## ISTRUZIONI DI BRANCHING

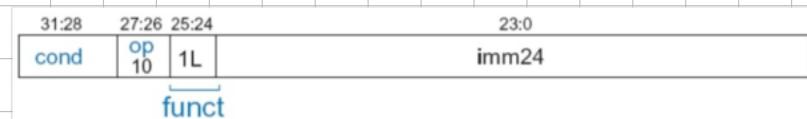
Le istruzioni di **BRANCHING** permettono di cambiare il valore del PC e quindi di saltare delle istruzioni.

Ne esistono di 2 tipi:

- SIMPLE BRANCH (B)
- BRANCH AND LINK (BL)

I branch possono essere **CONDIZIONATI**.

## FORMATO DELL'ISTRUZIONE



### ● COND

Sono i 4 bit che indicano la condizione sull'istruzione

### ● OP

le operazioni di branching si codificano con 10

### ● FUNCT

Il primo bit è sempre 1, il secondo bit L vale 1 per **BL** e 0 per **B**

## ● IMM24

Indica l' istruzione in PC+8.

Questo immediato è indicato in complemento a 2 a differenza degli altri immediati che sono **UNSIGNED**.

Questa costante viene moltiplicata  $\times 4$  per indicare direttamente a quale istruzione in memoria dobbiamo fare riferimento.

Inoltre imm24 viene estesa **CON SEGNO** (vengono aggiunti 8 bit uguali al bit più significativo). L'extend deve quindi eseguire 3 tipi di estensione. **IMMSrc** necessita ora di 2 bit.

## BRANCH AND LINK

L' istruzione **BL** è usata per la chiamata di una funzione (**SUBROUTINE**). In particolare salva R15 (che contiene l' indirizzo dell' istruzione successiva) in R14, effettua l' istruzione della funzione e per "tornare" a R15 copia R14 in R15 (**MOV R14, R15**)

R14 è detto **LINK REGISTER**, in esso viene salvato l' indirizzo di ritorno (il contenuto di R15) quando viene eseguita l' istruzione BL

Es.

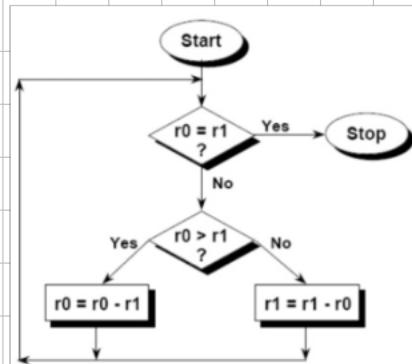
Vediamo a livello macchina come viene eseguito il massimo comune divisore tra due numeri.

MCD    ① CMP R0, R1

② SUB GT R0, R0, R1

③ SUB LT R1, R1, R0

④ B NE MCD



MCD indica l'etichetta di questa istruzione (anziché usare l'indirizzo dell'istruzione)

① Verifico che i due numeri siano diversi (altrimenti l'MCD è il numero stesso)

② Se  $R0 > R1$ ,  $R0 \leftarrow R0 - R1$

③ Se  $R1 > R0$ ,  $R1 \leftarrow R1 - R0$

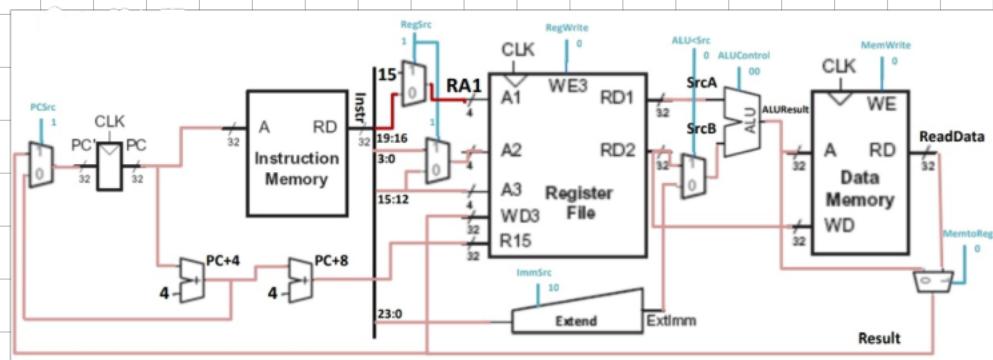
④ Torna a MCD se non sono uguali  $R0$  e  $R1$

Quando saranno uguali, ②, ③ e ④ non verranno eseguiti e si uscirà dalla funzione

OSS

Si puo' osservare come la scrittura sia quella di un CICLO WHILE

## DATAPATH BRANCHING



Siccome  $PC + 8$  deve poter essere letto in  $A_1$ , inseriamo un multiplexer che selezioni 15 per le istruzioni di Branch attraverso il segnale REGSRC

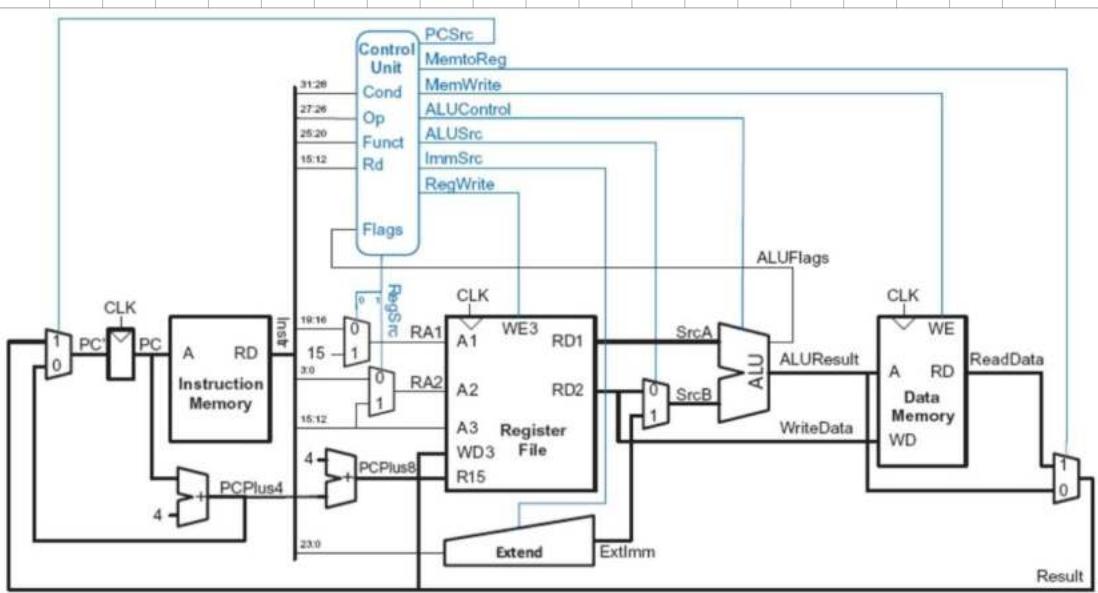
$MemtoReg = 0$ ,  $PCSrc = 1$  per selezionare PC da ALUResult

## UNITA' DI CONTROLLO

L' UNITA' DI CONTROLLO setta i segnali dei multiplexer in base a :

- COND, OP e FUNCT in Instr
- i FLAG
- Se il Rd è PC

Inoltre memorizza e aggiorna i flag nel CPSR



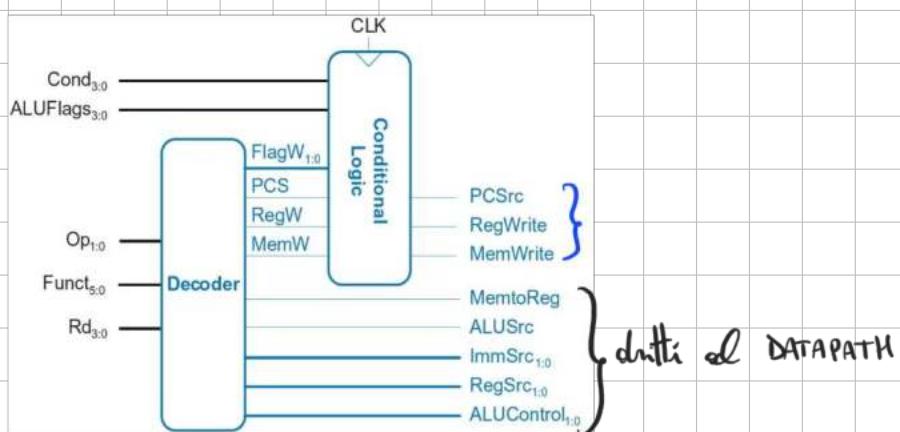
Può essere suddivisa in:

## ● DECODER

Genera i segnali di controllo sulla base di Instr

## ● LOGICA CONDIZIONALE

Gestite i flag di stato ed eventualmente li aggiorna.



► **PCSrc**, **RegWrite**, **MemWrite** saranno aggiornati dal decoder solo se

(Cond è soddisfatta, altrimenti restano settati a 0).

►  $\text{FLAGW}_{1:0}$  indica quando le ALUFlags vanno aggiornate ( $S=1$ )

le istruzioni ADD e SUB aggiornano tutti i flag, AND e ORR aggiornano solo C e V quindi  $\text{FlagW}_1=1$  attiva N2,  $\text{FlagW}_0=1$  attiva CV

## DECODER

Si puo' suddividere in:

### • MAIN DECODER

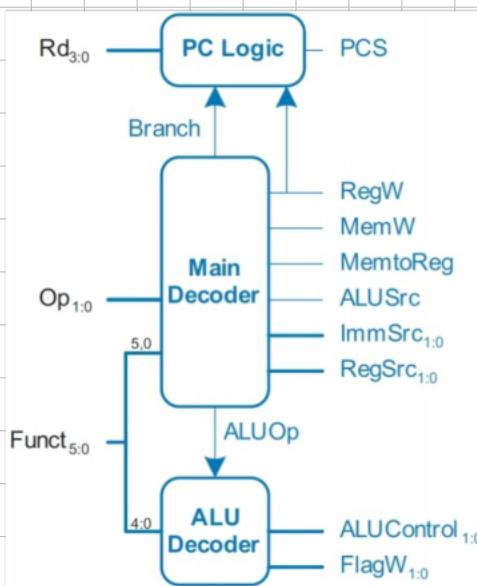
Produce la maggior parte dei segnali

### • ALU DECODER

Utilizza il campo FUNCT per determinare istruzioni di data-processing

### • PC LOGIC

Determina se il PC dev' essere aggiornato o no R15 o da un' istruzione di branch



- I segnali PCS, MEMW e REGW vengono trascritti in PCSource, MemWrite e RegWrite passando per la logica condizionale che li puo' TRASCRIVERE, AZZERARE o scambiarla del campo Cond.
- I segnali Branch e ALUOp vengono utilizzati per indicare l' istruzione B o le istruzioni di data-processing.

- Possiamo riassumere il funzionamento del main decoder in una tavola di verita'

Op	Funct <sub>5</sub>	Funct <sub>0</sub>	Type	MemW	ALUSrc	ImmSrc	RegW	RegSrc	ALUOp		
00	0	X	DP Reg	0	0	0	0	XX	1	00	1
00	1	X	DP Imm	0	0	0	1	00	1	X0	1
01	X	0	STR	0	X	1	1	01	0	10	0
01	X	1	LDR	0	1	0	1	01	1	X0	0
11	X	X	B	1	0	0	1	10	0	X1	0

- L' ALUDecoder sara':

ALUOp	Funct <sub>4:1</sub> (cmd)	Funct <sub>0</sub> (S)	Type	ALUControl <sub>1:0</sub>	FlagW <sub>1:0</sub>
0	X	X	Not DP	00	00
1	0100	0	ADD	00	00
		1			11
	0010	0	SUB	01	00
		1			11
	0000	0	AND	10	00
		1			10
	1100	0	ORR	11	00
		1			10

- FlagW<sub>1</sub> = 1: NZ (Flags<sub>3:2</sub>) devono essere salvate
- FlagW<sub>0</sub> = 1: CV (Flags<sub>1:0</sub>) devono essere salvate

- La PC-Logic controlla unicamente se l'istruzione è una scrittura in R15 o un BRANCH.

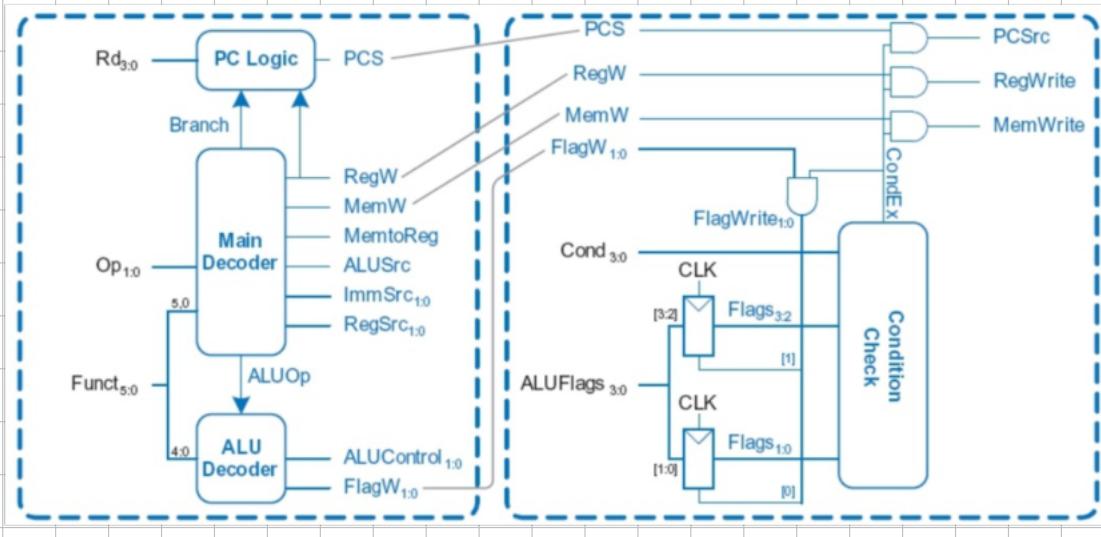
La condizione che verifica tale controllo è:

$$PCS = ((Rd == 15) \text{ AND } RegW) \text{ OR Branch}$$

Se l'istruzione è eseguita  $PC_{Src} = PCS$ , altrimenti sarà 0  
(e quindi  $PC = PC + h$ )

## LOGICA CONDIZIONALE

La logica condizionale calcola (attraverso l'ALU) SEMPRE i segnali MemW, RegW, PCS, forzandoli a 0 (con delle porte AND) tramite CondEX se la condizione non è soddisfatta



DECODER

LOGICA CONDIZIONALE

Nel CONDITION CHECK arrivano in input Cond<sub>3:0</sub> e i 5 flag di stato. In output Condex varia 0 quando il flag non verifica la condizioni, azzerando i segnali in uscita

## CONDIZIONI SU FLAG DI STATO

Suddividiamo i flag in base al tipo di operazione (che aggiorni i flag)

## OPERAZIONI ARITMETICHE

Mnemonic	Name	CondEx
MI	Minus / Negative	$N$
PL	Plus / Positive or zero	$\bar{N}$
VS	Overflow / Overflow set	$V$
VC	No overflow / Overflow clear	$\bar{V}$
AL	Always / unconditional	ignored

} FLAG

## CONFRONTO TRA DUE INTERI

Ricordiamo che in particolare in questo caso stiamo studiando il risultato dell'operazione **CMP**

Mnemonic	Name	CondEx
EQ	Equal	$Z$
NE	Not equal	$\bar{Z}$
CS / HS	Carry set / Unsigned higher or same	$C$
CC / LO	Carry clear / Unsigned lower	$\bar{C}$
HI	Unsigned higher	$\bar{Z}C$
LS	Unsigned lower or same	$Z \text{ OR } \bar{C}$
GE	Signed greater than or equal	$\bar{N} \oplus \bar{V}$
LT	Signed less than	$N \oplus V$
GT	Signed greater than	$\bar{Z}(N \oplus \bar{V})$
LE	Signed less than or equal	$Z \text{ OR } (N \oplus V)$

Mnemonic	Name	CondEx
GE	Signed greater than or equal	$\bar{N} \oplus V$
LT	Signed less than	$N \oplus V$
GT	Signed greater than	$\bar{Z}(N \oplus \bar{V})$
LE	Signed less than or equal	$Z \text{ OR } (N \oplus V)$

• LE

$R1 < R2$  significa che  $R1 - R2 < 0$

Non basta solo N come flag perché R1-R2 potrebbe produrre un overflow.

Quindi se V=0 (**NON** c'è overflow), N=1  
(il risultato è negativo).

Se V=1 (avremo un overflow), N=0 perché A-B  
dà un overflow se A e (-B) sono **CONCORDI** e il  
risultato della somma ha segno opposto.

Se abbiamo avuto un overflow, il risultato ci sarà dato  
positivo e quindi N=0.

N e V devono essere **DISCORDI** per dare 1 (vero) quindi  
 $N \oplus V$

- GE Sarà la condizione complementare (quindi negato)

Mnemonic	Name	CondEx
CS / HS	Carry set / Unsigned higher or same	C
CC / LO	Carry clear / Unsigned lower	$\bar{C}$
HI	Unsigned higher	$\bar{Z}C$
LS	Unsigned lower or same	Z OR $\bar{C}$

In complemento a 2 (quindi con segno) A-B si calcola  
 $A + (\sim B + 1)$  dove  $\sim B$  è la **NEGAZIONE** bit a bit di B.

In rappresentazione senza segno il complemento si calcola:

Per calcolare  $\sim B$  regalo la differenza bit a bit della parola composta da tutti 1 e B. Questo significa calcolare  $2^m - 1 - B$  ( $2^m - 1$  è il massimo numero rappresentabile)

Quindi  $A + (\sim B + 1)$  unsigned =  $2^m + A - B$

$2^m + A - B$  NON produce carry out se

$$\cancel{2^m} + A - B \leq \cancel{2^m} - 1 \rightarrow A + 1 \leq B \rightarrow A < B$$

In definitiva abbiamo dimostrato come  $A < B$  sia verificato da  $\bar{C}$  (non viene prodotto carry out).

Ese.

Vediamo come si controlla la CU

● EOR  $\text{REQ}$  R5, R6, R7  $(R6 \text{ XOR } R7)$

$\text{Cond}_{3:0} = 0000$  (EQ)

if  $\text{flag}_{3:0} = X1XX \Rightarrow \text{Cond}_{\text{ex}} = 1$  (condizione soddisfatta)  
NZCV

Se  $R6 \text{ XOR } R7 = 0$  i due numeri sono uguali (condizione soddisfatta)

## OSS

Flag Write<sub>1</sub> = 1 aggiorna NZ

Flag Write<sub>0</sub> = 1 aggiorna CV

## TEMPI DI RITARDO

Nei processori a ciclo singolo, ogni istruzione impiega un ciclo di clock.

Quindi il CPI = 1.

I ritardi prodotti da un'istruzione LDR sono:

- OPERAZIONI ESEGUITE IN PARALLELO →
- ▶ ( $t_{pcq\_PC}$ ) – caricamento di un nuovo indirizzo (PC) sul fronte di salita del clock;
  - ▶ ( $t_{mem}$ ) – lettura dell'istruzione in memoria;
  - ▶ ( $t_{dec}$ ) – il Decoder principale calcola RegSrc0, che induce il multiplexer a scegliere Instr<sub>19:16</sub> come RA1, e il register file legge questo registro come srcA;
  - ▶ ( $\max[t_{mux} + t_{RFread}, t_{ext} + t_{mux}]$ ) – mentre il register file viene letto, il campo costante viene esteso e viene selezionata dal multiplexer ALUSrc per determinare srcB.
  - ▶ ( $t_{ALU}$ ) – l'ALU somma srcA e srcB per trovare l'indirizzo effettivo.
  - ▶ ( $t_{mem}$ ) – La memoria di dati legge da questo indirizzo.
  - ▶ ( $t_{mux}$ ) – il multiplexer MemtoReg seleziona ReadData.
  - ▶ ( $t_{RFsetup}$ ) – viene impostato il segnale Result ed il risultato viene scritto nel register file.

$t_{RFread}$   
e' sempre max

Il tempo totale di ritardo è dato dalla somma di tutti questi.

Sostituendo con dei valori reali, il tempo per eseguire tutta l'istruzione è di 840ps, quindi ogni clock deve durare almeno 840ps.

## LIMITI DEL CICLO SINGOLO

Le problematiche principali sono 3:

### ● SEPARAZIONE DELLE MEMORIE

La memoria **ISTRUZIONI** e quella **DATI** devono essere separate poiché devono essere gestiti nello stesso ciclo

### ● INEFFICIENZA TEMPORALE

Il clock deve avere la durata dell'istruzione più lenta, spreco di tempo inutile per istruzioni veloci.

### ● DUPLICAZIONE DELLE COMPONENTI

Ogni componente svolge uno e un solo compito.

## VANTAGGI DEL CICLO MULTIPLO

Le architetture a ciclo **MULTIPLIO** ovviano a questi problemi suddividendo ogni istruzione in più passi (**FETCH, DECODE, EXECUTE, STORE**) ognuna delle quali viene eseguita in un ciclo di clock. In questo modo operazioni più brevi sono svolte più rapidamente.

A livello hardware le principali differenze sono due:

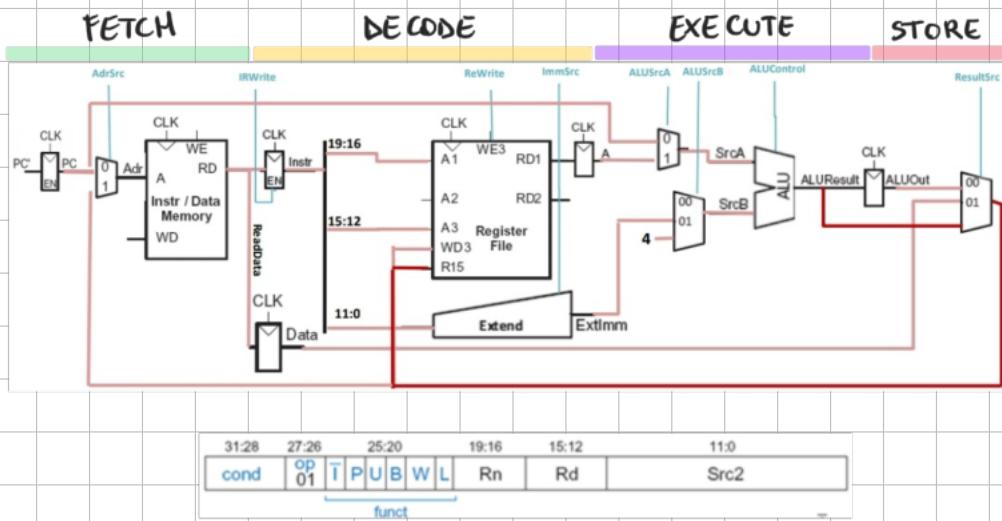
- la memoria istruzioni / dati è **UNICA**, questo è possibile perché l'istruzione viene letta in un ciclo diverso rispetto alla gestione dei dati.
- L' **ALU** aggiorna il **PC** ed esegue le istruzioni.

### DATA PATH A CICLO MULTIPLO

I DATA PATH sono sviluppati in modo **INCREMENTALE**, salvando per ogni fase, l'informazione in un registro.

Anziché vedere step-by-step le istruzioni.

#### LDR



- ① Il **PC** porta l'istruzione al data memory che la legge in **RD**. L'informazione arriva a un registro **IR** che

riceve un segnale **IRWRITE** che indica quando eseguire l'istruzione.

② A questo punto dobbiamo leggere il registro sorgente.

Lo ricaviamo dal campo **Rm** di  $\text{Instr}_{15:16}$ .

Questo viene caricato in **A1** del file register, letto in **RD1** e **MEMORIZZATO** in un registro **A** (che al battuta del clock darà il via per proseguire con l'istruzione).

③ Come nel ciclo singolo l'**OFFSET** viene esteso a 32 bit.

Questo **non** viene salvato in alcun registro (perché dipende solo da  $\text{Instr}$  e questa non cambia fino al termine)

④ L'**ALU** calcola il registro dal quale prendere l'informazione sommando / sottraendo al registro di base l'**offset**.

I due operandi **SrcA** e **SrcB** provengono quindi rispettivamente da **register file** ed **extender**.

Il risultato **ALUResult** sarà salvato nel registro **ALUout**

⑤ Ora il contenuto dell'indirizzo calcolato dev'essere prelevato, quindi l'indirizzo arriva alla porta **A** della memoria.

A questa parte era letta anche l' istruzione, allora inseriamo un multiplexer controllato dal segnale  $AdrSrc$ .

Questi dati vengono letti sul bus **Read Data** e memorizzati in un registro chiamato **DATA**

⑥ Siccome i dati letti vanno scritti nel registro di destinazione **Rd** l' informazione deve arrivare alla porta **WD3**.

Siccome eventualmente dev' essere scritto direttamente l' **ALUResult**,

Data non arriverà direttamente a **WD3**, bensì va in un multiplexer che riceve in input anche **ALUOut** e seleziona quale tra i due dev' essere scritto tramite il segnale **ResultSrc**

Si noti che in questa fase la scrittura è abilitata da **RegWrite**

⑦ A questo punto va eseguita l' istruzione successiva, e abbiamo visto che nel caso del ciclo multiplo l' ALU si occupa dell' incremento di **PC**.

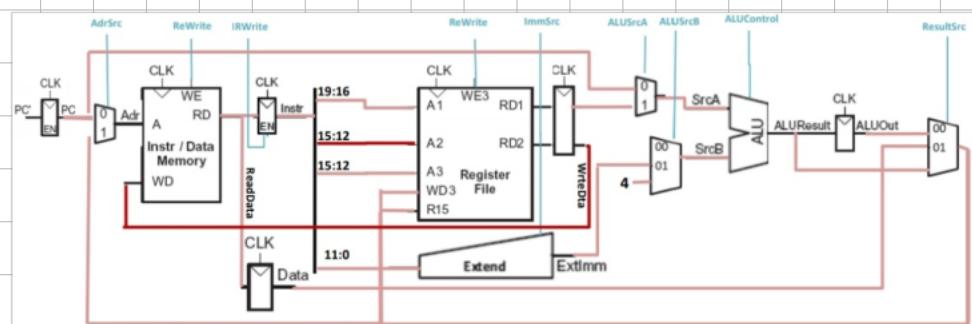
Quindi l' ALU riceverà come **SrcA** il contenuto del registro **A** oppure **PC**. Di conseguenza **SrcB** sarà **ExtImm** (offset) oppure "4" (per ottenere l' istruzione successiva  $PC+4$ )

⑧ Infine ricordiamo che in R15 del register file va caricato

$PC + 8$ . Durante il **FETCH**  $PC + 4$  (portato nel multiplexer che gestisce **ALUResult**) viene portato a **PC**; durante il **DECODE ALUResult** viene caricato in **R15** che fondamentalmente sara'  $PC + 8$  (perche' gli andiamo a sommare 4).

## STR

Implementiamo anche l' istruzione di **STORE**



Il processo per individuare le informazioni e' analogo, dobbiamo caricare il contenuto di **R1** in **R2** attraverso **A2**.

Il contenuto viene salvato in un registro **Write Data** che porterà l'informazione a **WD** (per essere scritta)

## DATA PROCESSING

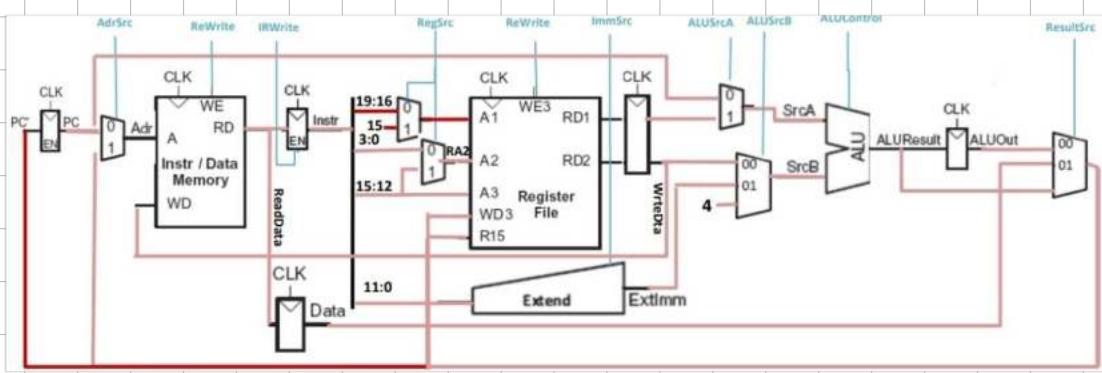
I comandi per eseguire un' istruzione di data-processing sono già implementati nel datapath.

In questo caso legge il primo operando da  $R_m$ , estende la costante a 32 bit ed esegue l'operazione tramite l'ALU.  
L'ALU control indicherà l'operazione di data-processing (ADD, OR, ...).  
Per selezionare il secondo operando  $R_m$  in A2 dobbiamo inserire un multiplexer che selezioni cosa caricare in A2 tra  $R_m$  (3:0) o  $R_d$  (15:12) attraverso un segnale  $IR\_{Write}$ .

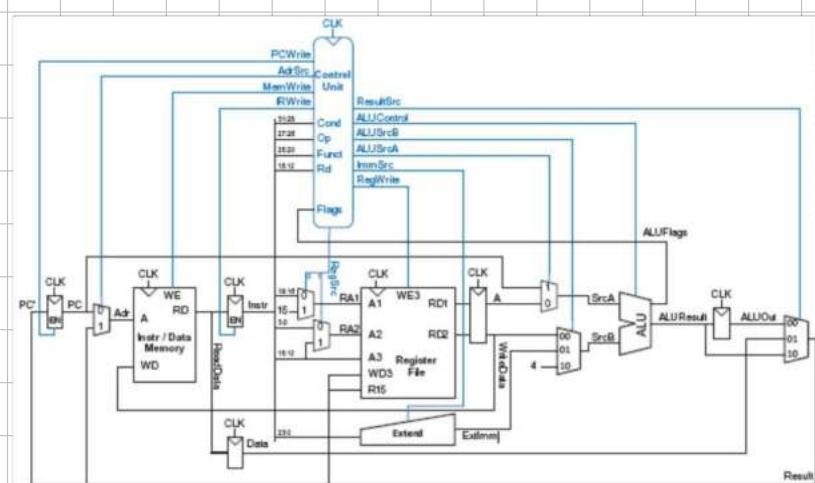
Questa ambiguità va riproposta anche nell'inserimento del secondo operando dell'ALU ( $SrcB$ ) che già possedeva un multiplexer che sceglieva tra  $4$  (per  $PC+4$ ) e l' $R_d$  (per LDR e STR), e ora anche per  $R_m$  (che ricordiamo essere i 4 bit meno significativi).

## BRANCH

Per indicare il salto da effettuare dobbiamo inserire un multiplexer che selezioni per A1  $R_m$  (19:16) oppure  $R_{15}$ , questo perché il salto viene calcolato dall'ALU e portato diritto a  $PC$ , ma necessitiamo di  $PC+8$  ( $R_{15}$ ) per tornare indietro una volta terminata l'operazione.



## CONTROL UNIT A CICLO MULTIPLO



La differenza sostanziale sta nel **MAIN DECODER** che, a ciclo singolo calcola tutti i segnali univocamente, mentre a ciclo multiplo ad ogni fase il valore dei selettori deve cambiare.

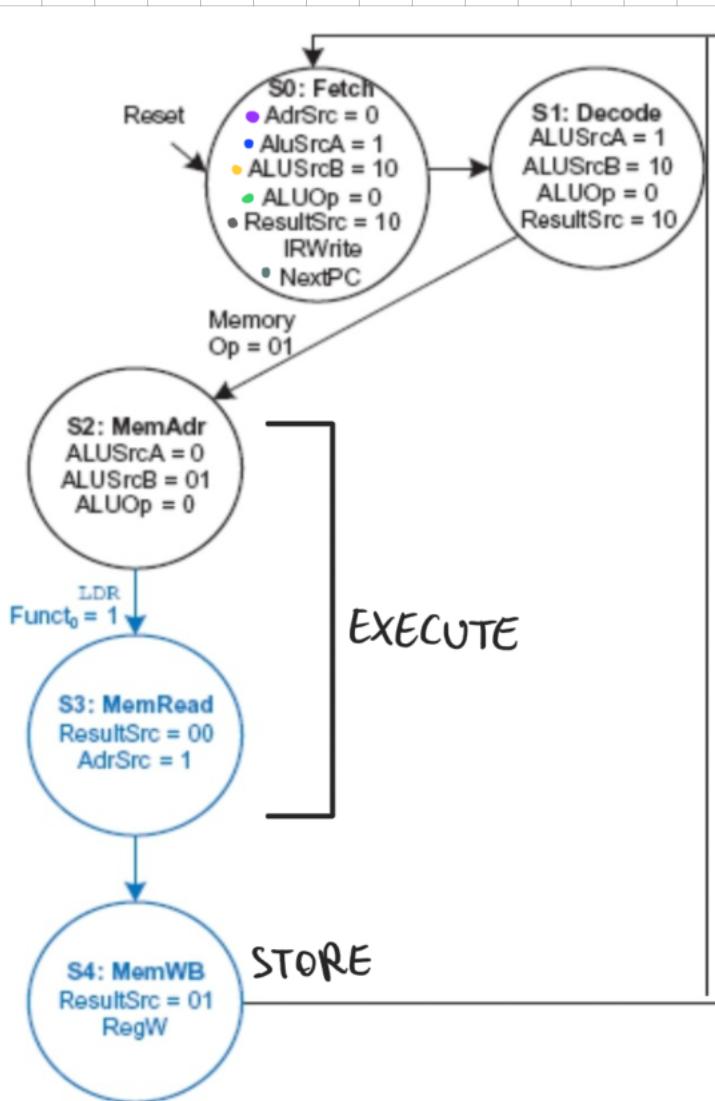
Il main decoder non puo' essere una logica combinatoria, sarà una **MACCHINA A STATI FINITI** (macchina di Moore)

Si noti che c'e' un segnale in piu' per aggiornare **PC**.

## DATA FLOW

Abbiamo visto che il decoder è una macchina a stati finiti, questi stati cambiano con le fasi di un'istruzione e ovviamente ciò che cambia nei vari stati sono i segnali da aggiornare

## LDR



● S0: Fetch

► A deve caricare PC

► Bisogna aggiornare PC

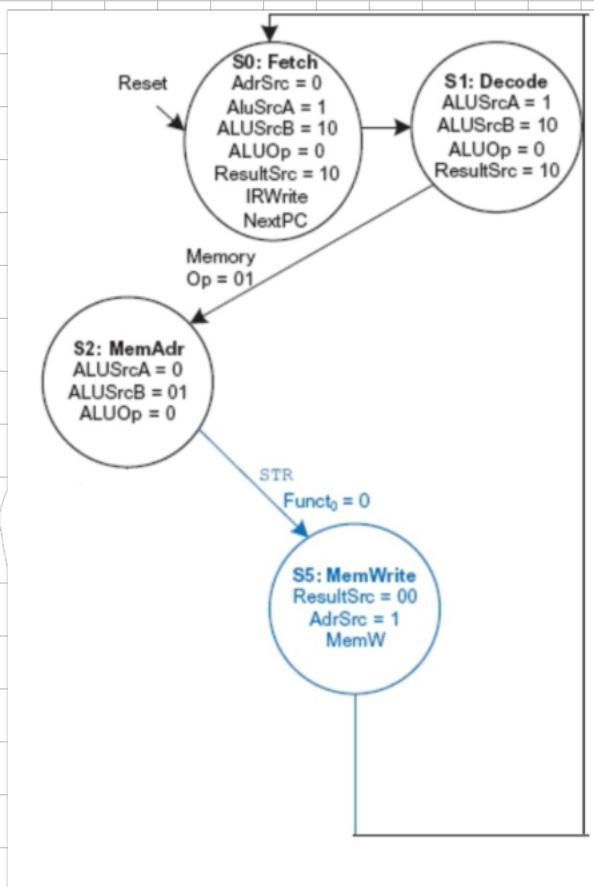
► Quindi sommare 4

► ALUOp è il segnale nel main decoder che attiva la logica (non ci serve) Quindi ResultSrc seleziona stesso l'ALUResult da portare diritto a PC (ricorda che non viene salvato in un registro)

► NextPC interno al main decoder sarà l per aggiornare PC

Seguono di conseguenza nelle altre fasi

STR



Lo stato di fetch è comune a tutti  
(con gli appropriati segnali).

Decode coincide con LDR ma in  
S2 troviamo direttamente dove andare  
a conservare il dato in S5.

Non abbiamo lo stato di scrittura  
del dato, quindi sarà più breve  
(4 clock/stati anzide 5)

## ARCHITETTURA

E' la descrizione operazionale di un computer, un insieme di istruzioni e registri.

Le istruzioni ARK sono "codificate" in ASSEMBLY (per noi umani) e in linguaggio MACCHINA.

## OPERANDI

Gli operandi di un'istruzione sono REGISTRI e COSTANTI.

L'ARK ha 16 registri identici a 32 bit, che hanno scopi diversi, che però hanno quasi tutti a che vedere con la chiamata a funzione

Name	Use
R0	Argument / return value / temporary variable
R1-R3	Argument / temporary variables
R4-R11	Saved variables
R12	Temporary variable
R13 (SP)	Stack Pointer
R14 (LR)	Link Register
R15 (PC)	Program Counter

le SAVED variabili NON vengono modificate prima e dopo la chiamata a funzione.

## GENERAZIONE DI COSTANTI

E' possibile inizializzare dei registri di costanti con l'istruzione **MOV**

La precisione degli immediate e' sempre 8 bit (che poi possono essere ruotati per aumentare/diminuire)

### C CODE

```
INT A=23
```

```
INT B=0x45
```

### ASSEMBLY

```
RD=a RT=b
```

```
MOV RD, #23  
MOV RT, #0x45
```

Per avere una costante con precisione maggiore di 8 bit (quindi più grande) possiamo usare l'istruzione **ORR**

Sia  $RD = a$

```
INT A= 0x7EDC8765
```

```
MOV RD, #0x7E000000
```

```
ORR RD, RD, #0xDC0000
```

```
ORR RD, RD, #0x8700
```

```
ORR RD, RD, #0x65
```

**ORR** sommerà bit a bit dandoci l'output desiderato

## IF THEN

Vediamo in **ASSEMBLY** il costrutto if then

C CODE

IF ( $i=3$ )  
 $F = G + H$   
 $F = F - i$

$R0 = F, R1 = G, R2 = H, R3 = I, R4 = J$

CMP R3, R4

setta i flag di R3-R4

BNE L1

se non sono uguali, salta a L1

ADD R0, R1, R2       $F = G + H$

L1

ETICHETTA  
("nome" di un'istruzione)

SUB R0, R0, R3       $F = F - 1$

In alternativa potremo

CMP R3, R4

setta i flag di R3-R4

ADDEQ R0, R1, R2

se sono uguali addiziona

SUB R0, R0, R3       $F = F - 1$

## OSS

L1 è un' **ETICHETTA**, il "nome" di un' istruzione, utile per individuare le istruzioni per il branch

## ELSE

Implementiamo l'indicazione ELSE

IF ( $I == J$ )

$F = G + H$

ELSE

$F = F - 1$

CMP R3, R4  
BNE L1  
ADD R0, R1, R2  
B L2

se non sono uguali esegui L1

L1

SUB R0, R0, R3

L1 viene eseguita se  $I \neq J$

L2

---

In alternativa

CMP R3, R4

ADDEQ R0, R1, R2

i flag **NON** vengono aggiornati

SUBNEQ R0, R0, R3

## CICLO WHILE

Calcoliamo il valore di  $x$  tale che  $2^x = 128$

INT Pow=1

INT x=0

WHILE ( Pow != 128 )

Pow \*= 2

X ++

$RO = \text{POW}$ ,  $R1 = X$

MOV RO, #1  
MOV R1, #0

WHILE CMP RO, #128  
BEQ DONE se  $\text{POW} = 128$ , esci

LSL RO, RO, #1  $\text{POW} *= 2$   
ADD R1, R1, #1  $X++$   
B WHILE ritorna all'inizio

DONE

### OSS

Andiamo sempre a studiare la condizione d'**USCITA**, al contrario del **C**

### CICLO FOR

Il ciclo for e' così costituito:

for (initialization; condition; loop operation)  
statement

- INITIALIZATION: eseguita prima che il ciclo inizi
- CONDITION: condizione di continuazione, controllata all'inizio
- LOOP OPERATION: eseguita alla fine del ciclo

● STATEMENT: eseguito ad ogni iterazione

Vediamo la somma dei numeri da 1 a 9

INT SUM=0, I

FOR (I=1; I!=10; I++)

SUM+=i

R0=1, R1=SUM

MOV R0, #1

MOV R1, #0

FOR CHP R0, #10

BEQ DONE

controlla  $I \neq 10$

se  $I = 10$  esci

ADD R1, R1, R0

SUM+=1

ADD R0, R0, #1

I++

B FOR

ritorna al for

DONE

OSS

In ARM i loop **DECRESCENTI** sono più efficienti

INT SUM=0, I

MOV R0, #9

MOV R1, #0

FOR (I=9; I!=0; I--)

FOR ADD R1, R1, R0

SUBS R0, R0, #-1

aggiorna i flag

SUM+=1

BNE DONE

NE verifica solo il flag  
Z, se "I--" ha dato come  
risultato 0, esci

DONE

In questo caso NE non lo usiamo con il CHP, bensì con SUB

quindi ci indicherà solo se la sottrazione ha dato 0 ( $Z=1$ )

## ARRAYS

Un **ARRAY** è costituito da un **SIZE** (numero di elementi) e un **INDEX** (quale elemento dell' **ARRAY**).

Se il nostro array è formato da parole a 32 bit, ogni indirizzo (quindi ogni  $A[i]$ ) occupa 4 byte consecutivi.

Si parte dal **BASE ADDRESS** (indirizzo di  $A[0]$ ), gli altri elementi verranno raggiunto con  $B.A. + 4$

● INT A[5]

$A[0] += 8$

$A[1] += 8$

$R0 = \text{base address}$

MOV R0, #0x60000000 indirizzo di  $A[0]$

LDR R1, [R0]  
LSL R1, R1, #3  
STR R1, [R0]

$R1 = A[0]$   
 $R1 = R1 \ll 3$   
 $A[0] = R1$

LDR R1, [R0, #4]  
LSL R1, R2, #3  
STR R1, [R0, #4]

$R1 = A[1]$   
 $A[1] = A[0] + 4 \text{ byte}$

● INT A[200], I

FOR ( $I=199 ; I>=0 ; I--$ )

ARRAY[I] += 8

$R0 = \text{base address}$ ,  $R1 = 1$

MOV R0, #0x60000000

MOV R1, #193      b.a. indice  $\times 4$

FOR LDR R2, [R0, R1, LSL #2]  
LSL R2, R2, #3  
STR R2, [R0, R1, LSL #2]  
SUBS R1, R1, #1

per avere l'indirizzo di  $A[i]$   
devo sommare al base address ( $A[0]$ )  
l'indice moltiplicato  $\times 4$   
(ogni parola e' di 4 byte)

BNE FOR

## CHIAMATE DI FUNZIONI

Suddividiamo le funzioni in:

- CALLER (chiamante, main) e

- Passa gli argomenti al CALLEE

- Esegue un jump al CALLEE

- CALLEE (chiamato, function)

- Esegue la funzione chiamata

- Ritorna il risultato al CALLER

- Torna al punto di chiamata del caller

- Non sovrascrive registri e memoria del caller

Per eseguire la chiamata a funzione, il **CALLER** esegue un'operazione di **BRANCH and LINK**.

Il return al caller ripristina in **PC** il valore del **LINK REGISTER** (**MOV PC,LR**).

VOID HAIN()  
SIMPLE()  
A = B+C

0x 00000200 (HAIN) BL SIMPLE  
// 0x 00000204 ADD R4,R5,R6

VOID SIMPLE  
RETURN

0x004,01020 (SIMPLE) MOV PC,LR

Il **HAIN** salva nel **LR** l'istruzione successiva (quindi l'indirizzo corrente + 4), esegue il **BL** alla **funzione** e questa al termine (return) carica in **PC** **LR** (ovvero l'istruzione successiva), proseguendo da dove avevamo interrotto il **HAIN**.

Esempio di chiamata a funzione:

```
int main(){
    int y;
    ...
    y = diffofsums(2, 3, 4, 5); // 4 arguments
    ...
}

int diffofsums(int f, int g, int h, int i){
    int result;
    result = (f + g) - (h + i);
    return result;           // return value
}
```

R4=y

R4= result

MAIN    MOV R0, #2  
          R1, #3  
          R2, #4  
          R3, #5

BL DIFF  
MOV R4, R0

DIFF    ADD R8, R0, R1  
          ADD R9, R2, R3  
          SUB R4, R8, R9  
          MOV R0, R4  
          MOV PC, LR

La funzione DIFF sovrascrive i registri 4,8,9 dedicati alle saved variables nel MAIN, che NON possono essere modificate.

La soluzione è salvare i contenuti di R4,R8,R9 in memoria, utilizzare i registri precedentemente occupati e infine ricaricare in R4,R8 e R9 i valori salvati in memoria.

Questi valori sono salvati in uno STACK

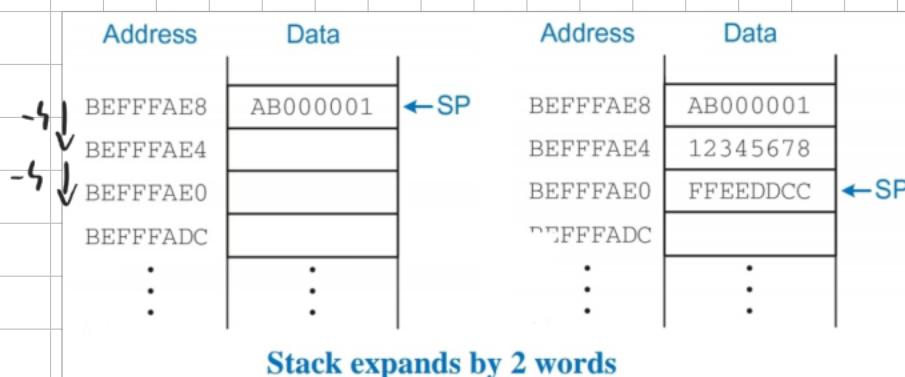
## STACK DELLE FUNZIONI

Lo **STACK** costituisce la memoria usata momentaneamente per salvare le **VARIABILI**. E' organizzato secondo una logica **LIFO**.

Le operazioni eseguibili su uno stack sono :

- **POP** : espandi lo stack
- **PUSH** : riduci lo stack

L'espansione avviene in modo **DECRESCENTE**, i registri che aggiungiamo decrescano. **SP** (stack pointer) punta al top, quando aggiungiamo valori allo stack, **SP** non punterà più al top, bensì al registro che prima era al top



Il programma di prima conetto risulta essere :

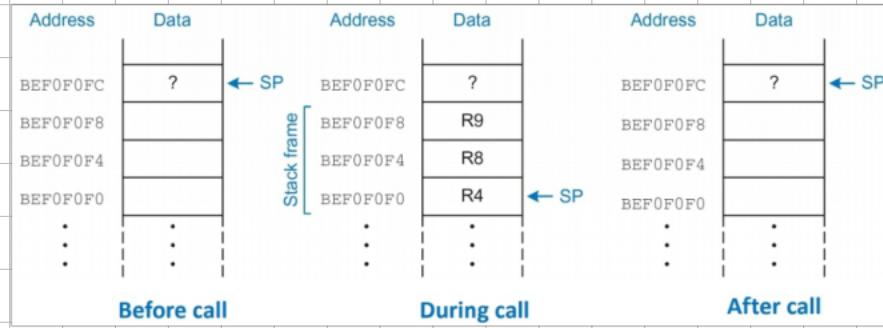
```

; R2 = result
DIFFOFSUMS
{
    SUB SP, SP, #12      ; make space on stack for 3 registers
    STR R4, [SP, #-8]   ; save R4 on stack
    STR R8, [SP, #-4]   ; save R8 on stack
    STR R9, [SP]         ; save R9 on stack
    ADD R8, R0, R1       ; R8 = f + g
    ADD R9, R2, R3       ; R9 = h + i
    SUB R4, R8, R9       ; result = (f + g) - (h + i)
    MOV R0, R4            ; put return value in R0
    LDR R9, [SP]          ; restore R9 from stack
    LDR R8, [SP, #-4]    ; restore R8 from stack
    LDR R4, [SP, #-8]    ; restore R4 from stack
    ADD SP, SP, #12      ; deallocate stack space
    MOV PC, LR            ; return to caller
}

```

Con **STR** salvo i valori contenuti in **R4, R8, R9**. Al termine ricaricherò i valori che avevo salvato nello stack con **LDR**.

Al termine **DEALLOCIAHO** lo spazio usato per lo stack riportandolo **SP** al suo valore originale.



La funzione chiamata si deve occupare di non sovrascrivere le saved variables.

La funzione chiamante si deve occupare di non occupare gli spazi di memoria dedicati a **SP** e i suoi "superiori" (**SP+4m**).

## FUNZIONI POP E PUSH

### ● POP

Consente di salvare in memoria più registri aggiornando lo SP (decrementando)

### ● PUSH

Ripristina i valori di memoria incrementando SP

$$R0 = a, R1 = b, R4 = i, R5 = x$$

E.  
C.  
int f1(int a, int b) {  
 int i, x;  
 x = (a + b) \* (a - b);  
 for (i=0; i<a; i++)  
 x = x + f2(b+i);  
 return x;  
}  
int f2(int p) {  
 int r;  
 r = p + 5;  
 return r + p;  
}

F1  
PUSH {R4, R5 LR}  
ADD R5, R0, R1  
SUB R12, R0, R1  
MUL R5, R5, R12  
MOV R4, #0

solviamo LR  
per il BL  
 $i = 0$

FOR  
CMP R4, R0  
BGE RETURN  
PUSH {R0, R1}  
ADD R0, R1, R4  
BL F2

$i = a$ ?  
solvio R0, R1 perché  
F2 li potrebbe  
modificare

LR → ADD R5, R5, R0  
POP {R0, R1}  
ADD R4, R4, #1  
B FOR

RETURN  
MOV R0, R5  
POP {R4, R5, LR}  
MOV PC, LR

R0 = P R4 = L

F2 PUSH {R4}

ADD R4, R0, #5

ADD R0, R4, R0

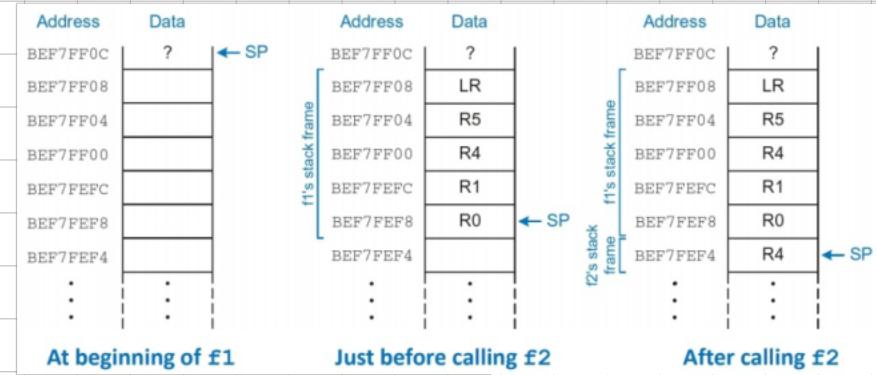
POP {R4}

Mov PC, LR

R4 mi serve, ma c'è una saved variable

Ripristina R0

Nello stack avremo



## FUNZIONI RICORSIVE

```

int factorial(int n) {
    if (n <= 1)
        return 1;
    else
        return (n * factorial(n-1));
}

```

R0 = n

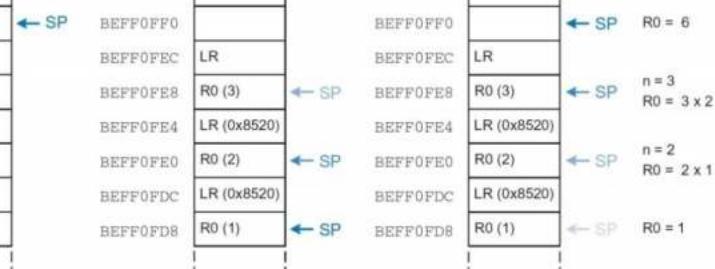
POP {	0x94	FACTORIAL	STR R0, [SP, #-4]!	; store R0 on stack
	0x98		STR LR, [SP, #-4]!	; store LR on stack
	0x9C		CMP R0, #2	; set flags with R0-2
	0xA0		BHS ELSE	; if (r0>=2) branch to else
	0xA4		MOV R0, #1	; otherwise return 1
	0xA8		ADD SP, SP, #8	; restore SP 1
	0xAC		MOV PC, LR	; return
PUSH {	0xB0	ELSE	SUB R0, R0, #1	; n = n - 1
	0xB4		BL FACTORIAL	; recursive call
	0xB8		LDR LR, [SP], #4	; restore LR
	0xBC		LDR R1, [SP], #4	; restore R0 (n) into R1
	0xC0		MUL R0, R1, R0	; R0 = n*factorial(n-1)
	0xC4		MOV PC, LR	; return

Address	Data	Address	Data	Address	Data
BEFF0FF0		BEFF0FF0		BEFF0FF0	
BEFF0FEC		BEFF0FEC	LR	BEFF0FEC	LR
BEFF0FE8		BEFF0FE8	R0 (3)	BEFF0FE8	R0 (3)
BEFF0FE4		BEFF0FE4	LR (0x8520)	BEFF0FE4	LR (0x8520)
BEFF0FE0		BEFF0FE0	R0 (2)	BEFF0FE0	R0 (2)
BEFF0FDC		BEFF0FDC	LR (0x8520)	BEFF0FDC	LR (0x8520)
BEFF0FD8		BEFF0FD8	R0 (1)	BEFF0FD8	R0 (1)

Before call

During call

After call



```
1 int main() {  
2     return 4+3;  
3 }
```

```
1 main:  
2     sub    sp, sp, #4  
3     mov    r0, #0  
4     str    r0, [sp]  
5     mov    r0, #7  
6     add    sp, sp, #4  
7     bx    lr
```

↖ indica se tornare a 32 bit o 16 bit

- ② alloca una parola a prescindere
- ③ assegna già un valore di ritorno
- ④ Stava il valore di ritorno 0 nello SP
- ⑤ Il compilatore calcola automaticamente il valore di ritorno  
a compile time, senza dover salvare 6 e 2
- ⑥ Libera la memoria dello SP
- ⑦ Invece di fare MOV PC LR fa un branch a LR.  
La X indica se l'istruzione successiva è a 16.

```
1 int main() {  
2     int i=2;  
3     return 4+i;  
4 }
```

```
1 main:  
2     sub    sp, sp, #8  
3     mov    r0, #0  
4     str    r0, [sp, #4]  
5     mov    r0, #2  
6     str    r0, [sp]  
7     ldr    r0, [sp]  
8     add    r0, r0, #4  
9     add    sp, sp, #8  
10    bx    lr
```

- 2) alloca 2 parole (una di default e una per i)
- 3) salva il valore di ritorno di default
- 4) nella prima parola (nel registro SP+4) carica il ritorno di default
- 5) Associo a R0 il valore di i (2)
- 6) e ne faccio lo store in SP
- 7) meccanicamente, qualora una variabile viene usata ne viene fatto il load dallo stack, in questo caso viene caricato proprio in R0 il valore di i
- 8) calcola il valore di ritorno
- 9) libera lo spazio dello stack

```

1 int main() {
2     int i=2;
3     int j=1;
4
5     if(i==j)
6         i=i+j;
7
8     i=i-j;
9     return 0;
10 }
```

```

1 main:
2     sub    sp, sp, #12
3     mov    r0, #0
4     str    r0, [sp, #8]
5     mov    r0, #2
6     str    r0, [sp, #4]
7     mov    r0, #1
8     str    r0, [sp]
9     ldr    r0, [sp, #4]
10    ldr   r1, [sp]
11    cmp   r0, r1
12    bne  .LBB0_2
13    b    .LBB0_1
14 .LBB0_1:
15    ldr    r0, [sp, #4]
16    ldr    r1, [sp]
17    add   r0, r0, r1
18    str   r0, [sp, #4]
19    b    .LBB0_2
20 .LBB0_2:
21    ldr    r0, [sp, #4]
22    ldr    r1, [sp]
23    sub   r0, r0, r1
24    str   r0, [sp, #4]
25    mov    r0, #0
26    add   sp, sp, #12
27    bx    lr|
```

② Allora 3 spazi di memoria

---

⑤ salva in R0 il valore di i (2) e lo salva nel  
secondo spazio disponibile dell' SP (SP+4)

⑦ Ugualmente per j

9-11 salva in R0 i e in R1 j e ne fa il CMP

⑫ Se non sono uguali (NE) non eseguire l' istruzione

⑬ Se sono uguali (quindi non è stato effettuato il branch  
precedente) fai un branch all' istruzione dell' if

(15) Meccanicamente ricarica in R0 e R1 le variabili i, j  
(carica le variabili ogni volta)

(17) esegue l'operazione e salva il risultato in R0

(18) Salva R0 (i) in SP+4 ed esegue il branch all'istruzione  
successiva

(21) Ancora una carica le variabili

(23) Calcola l'espressione e la salva in R0

(25) Restorna 0 e libera lo spazio

Cosa succede se nella condizione scriviamo ( $i=j$ ) ?

Fino alla riga 8 non ci sono differenze.

9	ldr	r0, [sp]
10	str	r0, [sp, #4]
11	cmp	r0, #0
12	beq	<u>.LBB0_2</u>
13	b	<u>.LBB0_1</u>

(9) Carica in R0 il valore di j

(10) Salva R0 in SP+4, dove era salvato i (quindi  $i=j$ )

(11) Se  $j=0$  non esegue la condizione, altrimenti esegue

```

int main(){
    int i=1;
    int j=1;
    int f=4;
    int g=3;
    int h=2;

    if( i==j )
        f=g+h;
    else
        f=f-i;
    return 0;
}

```

```

1   main:
2       sub    sp, sp, #24
3       mov     r0, #0
4       str    r0, [sp, #20] RETURN
5       mov     r0, #1
6       str    r0, [sp, #16] i
7       str    r0, [sp, #12] j
8       mov     r0, #4
9       str    r0, [sp, #8] f
10      mov    r0, #3
11      str    r0, [sp, #4] g
12      mov    r0, #2
13      str    r0, [sp]
14      ldr    r0, [sp, #16]
15      ldr    r1, [sp, #12]
16      cmp    r0, r1
17      bne   .LBB0_2
18      b     .LBB0_1

19  .LBB0_1: THEN
20      ldr    r0, [sp, #4]
21      ldr    r1, [sp]
22      add    r0, r0, r1
23      str    r0, [sp, #8]
24      b     .LBB0_3

25  .LBB0_2: ELSE
26      ldr    r0, [sp, #8]
27      ldr    r1, [sp, #16]
28      sub    r0, r0, r1
29      str    r0, [sp, #8]
30      b     .LBB0_3

31  .LBB0_3:
32      mov    r0, #0
33      add    sp, sp, #24
34      bx     lr

```

⑦ Si noti che il compilatore riconosce che  $i$  e  $j$  hanno lo stesso valore, che quindi viene allocato due volte nello stack (dopo aver associato ad  $R0$  il valore della variabile)

```
int main(){
```

```
    int z=2;
```

```
    int x;
```

```
    int y=(x=z+4)+2;
```

```
    z=x+3;
```

```
}
```

```
1  main:
2      sub    sp, sp, #12
3      mov    r0, #2
4      str    r0, [sp, #8]
5      ldr    r0, [sp, #8]
6      add    r1, r0, #4
7      str    r1, [sp, #4]
8      add    r0, r0, #6
9      str    r0, [sp]
10     ldr   r0, [sp, #4]
11     add   r0, r0, #3
12     str   r0, [sp, #8]
13     mov   r0, #0
14     add   sp, sp, #12
15     bx    lr
```

```
int main(){
```

```
    int z=2;
```

```
    int x= z-1==0? z+1:0;
```

```
}
```

La dichiarazione equivale a

if (z-1==0)

x = z + 1

else

x = 0

(CONDIZIONALE FUNZIONALE)

```
1  main:
2      sub    sp, sp, #16
3      mov    r0, #0
4      str    r0, [sp, #12]
5      mov    r0, #2
6      str    r0, [sp, #8]
7      ldr    r0, [sp, #8]
8      cmp    r0, #1
9      bne   .LBB0_2
10     b     .LBB0_1
11 .LBB0_1:
12     ldr    r0, [sp, #8]
13     add    r0, r0, #1
14     str    r0, [sp]
15     b     .LBB0_3
16 .LBB0_2:
17     mov    r0, #0
18     str    r0, [sp]
19     b     .LBB0_3
20 .LBB0_3:
21     ldr    r0, [sp]
22     str    r0, [sp, #4]
23     ldr    r0, [sp, #12]
24     add    sp, sp, #16
25     bx    lr
```

```

int main(){
    int a[3];
    int i=4;

    for(int j=0;j<3;j++)
        a[j]=j+i;
}

```

```

1   main:
2       sub    sp, sp, #24
3       mov     r0, #0
4       str    r0, [sp, #20]
5       mov     r1, #4
6       str    r1, [sp, #4]
7       str    r0, [sp]
8       b      .LBB0_1
9   .LBB0_1:
10      ldr   r0, [sp]
11      cmp   r0, #2
12      bgt  .LBB0_4
13      b      .LBB0_2
14   .LBB0_2:
15      ldr   r0, [sp]
16      ldr   r1, [sp, #4]
17      add   r1, r0, r1
18      add   r2, sp, #8
19      str   r1, [r2, r0, lsl #2]
20      b      .LBB0_3
21   .LBB0_3:
22      ldr   r0, [sp]
23      add   r0, r0, #1
24      str   r0, [sp]
25      b      .LBB0_1
26   .LBB0_4:
27      ldr   r0, [sp, #20]
28      add   sp, sp, #24
29      bx    lr

```

Allora 6 variabili :

- SP+20 ritorno default

- +16 A[2]

- +12 A[1]

- +8 A[0]

- +4 i

- SP J

⑪ Dopo aver caricato in R0 J , ne seguo il compare

⑫ Se  $j > 2$  (quindi  $j \geq 3$ ) segue il branch fuori dal ciclo

⑬ carica in  $R0 = i$  e  $R1 = j$

⑭ Calcola  $j+i$  e salvalo in  $R0$

⑮ in  $R2$  salva il **BASE ADDRESS** dell' ARRAY.

Il base address di un array coincide con  $A[0]$ .

⑯ Il valore salvato in  $R0$  deve andare in  $A[j]$ , questo indirizzo viene calcolato sommando al base address il valore di  $j$  (salvato in  $R1$ ) moltiplicato per  $4$  (LSL 2) perché ogni parola è di 4 byte e gli array distano tra di loro tutti 4 byte.

## AND e OR

Il compilatore verifica, nel caso di condizioni che contengono **AND** e **OR**, che sia verificata una specifica condizione:

### • OR

Il compilatore verifica le condizioni finché una non è verificata (vale 1), le restanti non verranno analizzate

## ● AND

Analogamente, per l'<sup>1</sup> AND cerca una condizione che NON  
sia verificata, così da uscire subito

Questa tecnica è detta **LAZY EVALUATION**

## MEMORIE E CPU

Le memorie e il processore lavorano a velocità molto diverse

Per ovviare a questo problema si ricorre all'utilizzo di una **PICCOLA** quantità di memoria molto **VELOCE** (**cache**) e una **GRANDE** quantità di memoria **LENTA**

## LOCALITÀ DEI RIFERIMENTI

La CPU tende ad eseguire ripetutamente un numero ristretto di istruzioni, e più raramente accede al resto del programma.

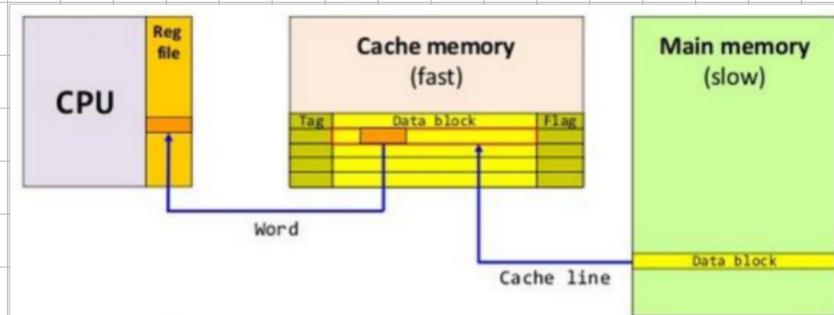
Questa proprietà è detta **LOCALITÀ DEI RIFERIMENTI**, e si suddivide in:

- **TEMPORALE**: Rappresenta la probabilità che un'istruzione eseguita da poco venga rieseguita a breve

- **SPAZIALE**: Rappresenta la probabilità che un'istruzione vicina (in termini di indirizzi) a una eseguita da poco, venga rieseguita a breve

## MEMORIA CACHE

La **MEMORIA CACHE** si interpone tra la CPU e la memoria per permettere tempi di accesso a quest'ultima maggiori, senza modificare alcuna struttura.



Dal Register File della CPU si verifica se l'istruzione sia già presente nella cache. Se è presente (**CACHE HIT**) allora sarà disponibile senza dover accedere alla memoria, altrimenti: (**CACHE MISS**) il dato sarà prelevato dalla memoria.

Per la proprietà **TEMPORALE**, l'indirizzo prelevato dalla memoria viene aggiunto alla cache. Per la proprietà **SPAZIALE** ci conviene aggiungere un intero **BLOCCO** da 16 o 32 byte (contigui).

## DIMENSIONE DELLA CACHE

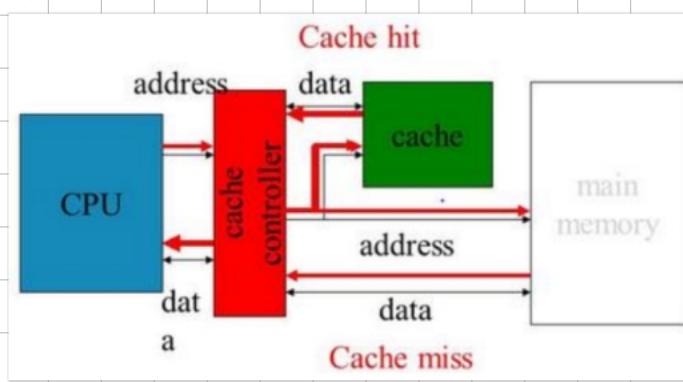
La cache è suddivisa in 2 livelli:

Il primo è il più veloce ma è poco capiente.

## GESTIONE DELLA CACHE

La relazione tra i blocchi della memoria principale e quelli della cache è definita dalla funzione di **HAPPING**

Quando la cache si esaurisce si ricorre a un algoritmo di **SOSTITUZIONE** per liberare i blocchi.



La gestione di tutti questi casi è affidata a un **CACHE CONTROLLER**, integrato nella memoria cache

## CACHE HIT

Vediamo cosa accade nel caso di una cache hit

## READ HIT

Semplicemente la memoria non viene coinvolta

## WRITE HIT

Si puo' procedere in due modi:

- WRITE THROUGH

Scrivo il dato nella cache e lo muovo nella memoria,  
implicando necessariamente scritture superflue.

- WRITE BACK

Invece di scrivere tutto il blocco nella memoria ogni volta,  
aggiorniamo solo il blocco nella cache contrassegnato con un  
bit detto **DIRTY**. Quando il blocco dovrà essere rimosso dalla  
cache, verrà ricopiato o meno in memoria

## CACHE MISS

Vediamo cosa accade nel caso di una cache miss

## READ MISS

La parola non viene trovata nella cache, accediamo alla memoria.  
Le possibilità ora sono due:

- Il dato della memoria viene copiato nella cache e  
successivamente letto dalla CPU.

- la parola viene inviata direttamente alla CPU  
(sostituendo nel frattempo il blocco nella cache).

Questa tecnica è detta **LOAD THROUGH**

## WRITE MISS

Analogamente abbiamo **WRITE BACK** e **WRITE THROUGH**  
(non diretto, diretto)

## PRESTAZIONI

- HIT RATE = n° di hit / n° di accessi
- MISS RATE = //
- 1 = MISS RATE + HIT RATE
- AMAT = Average memory access time

$$AMAT = t_{CACHE} + MR_{CACHE} [t_{MM} + MR_{MM} (t_{VH})]$$

↑                              ↑                              ↗  
 tempo di accesso            tempo di accesso            tempo di accesso  
 alla cache                    alla main memory            alla virtual memory

E.s.

Un programma ha 2000 load e stores, 1250 di questi sono presenti nella cache, il resto è fornito da altri livelli.

Trovare hit rate e miss rate

$$HR = 1250 / 2000 = 0,625$$

$$MR = 1 - 0,625 = 0,375$$

### TERMINOLOGIA CACHE

CAPACITA' (c) : n° di byte nella cache

BLOCK SIZE (b) : Dimensione di un blocco, ovvero numero di bytes che sono trasmessi nella cache ogni volta

N° BLOCKS (B) : N° di blocchi  $B = c/b$

DEGREE OF ASSOCIATIVITI (N) : N° di blocchi in un set  
(insieme di blocchi)

N° SETS (S) :  $S = B/N$

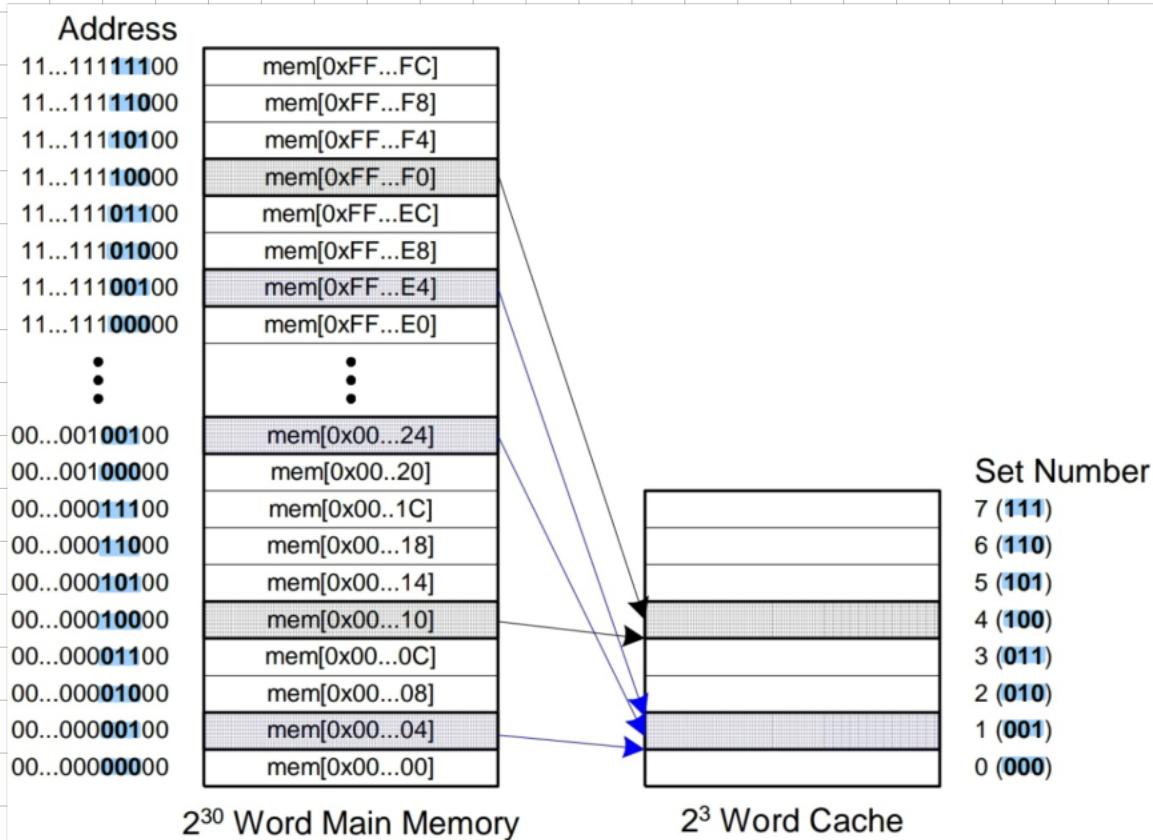
Ogni indirizzo di memoria è mappato in esattamente un SET

## TIPOLOGIE DI CACHE

A seconda del numero di blocchi per set, sono suddivise in:

- DIRECT MAPPED: 1 blocco per set
- N-WAY SET ASSOCIATIVE: N blocchi per set
- FULLY ASSOCIATIVE: 1 unico set

### DIRECT MAPPED



Esecuto direct mapped ogni blocco è contenuto in un set.

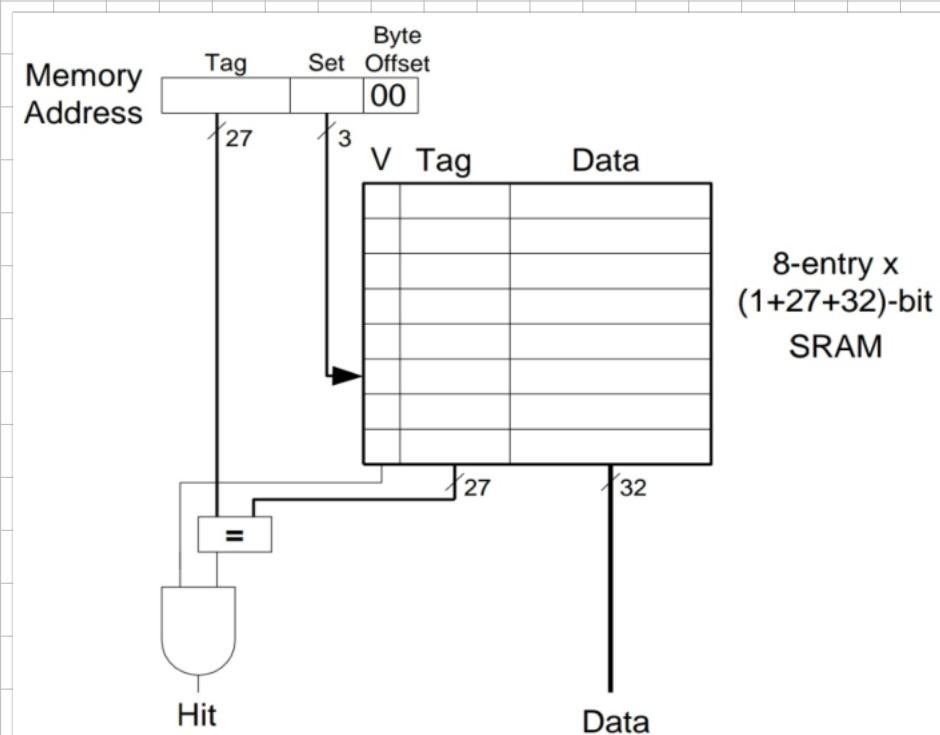
In quest' esempio ogni blocco è costituito da una parola, la cache contiene 8 blocchi ( $\rightarrow$  8 parole)

Come individuare il set mappato ad ogni blocco?

Si noti che, essendo una parola composta da 4 byte, i primi due bit dell'indirizzo sono sempre 00 (avendo +4 ogni volta)

3°, 4° e 5° bit ci danno proprio il set mappato (3 bit perché sono  $2^3 = 8$  blocchi nella cache)

Vediamo il **CACHE CONTROLLER**



L'indirizzo viene suddiviso in 3 parti: TAG (27), SET (3) e OFFSET BYTE (2).

Ogni riga della cache ha un bit di validità V, TAG (27) e DATA (32)

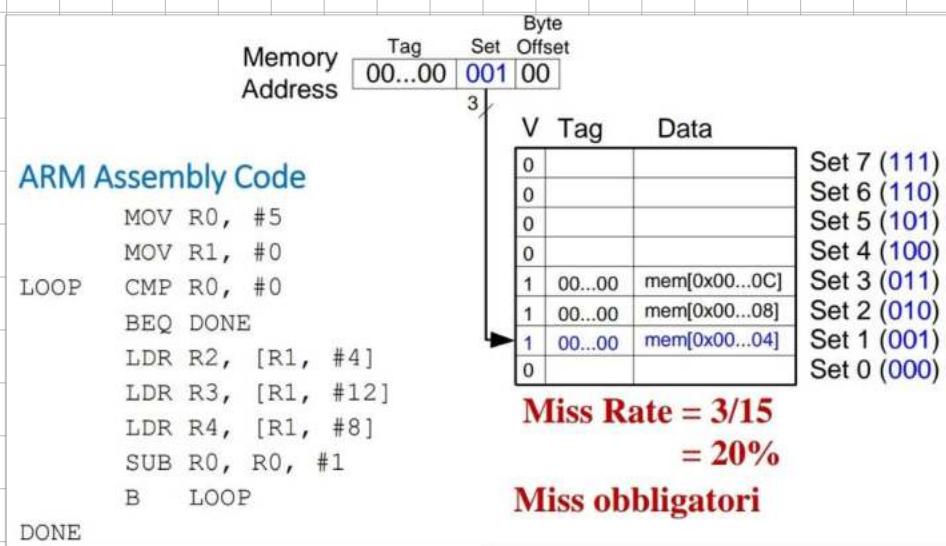
Il set come abbiamo visto indica la riga. Effettuo il confronto

del tag. Questo va in AND con V (poi verrà spiegato).

Se V=1 e il tag corrisponde avremo un HIT.

Ese.

Vediamo cosa succede con dei programmi

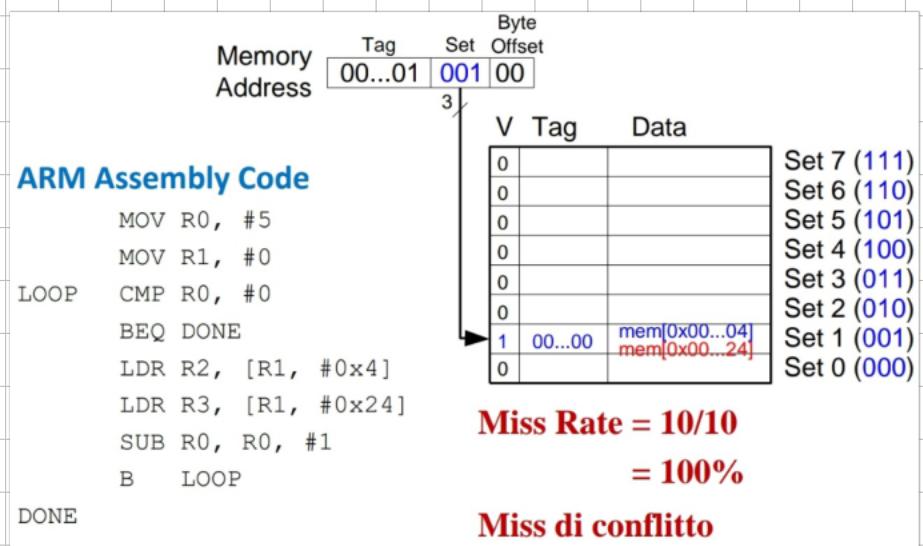


All'primo avvio i dati in cache sono tutti "sporchi", non  
vanno considerati, quindi tutti i bit V=0.

Quindi al primo loop prende tutti i dati in memoria e  
li carica anche in cache (LDR R2,...) e setta V=1.

Dopo il primo scorrimento, troverà sempre in cache i registri.

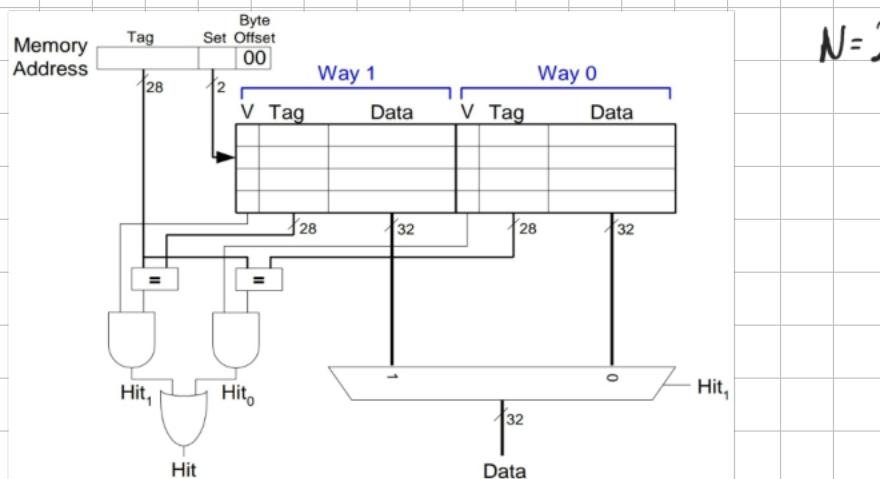
I primi 3 miss sono detti **OBBLIGATORI**.



Il caso e' molto simile, solo che gli indirizzi 0x24 e 0x4  
hanno lo stesso set, quindi si sovrascrivono sempre a vicenda  
in ogni loop.

## N-WAY ASSOCIATIVE

Così questo tipo di cache possono ridurre i miss di conflitto.



Consideriamo sempre una cache a 8 parole.

Ogni set ha 2 parole, ognuna col proprio tag (e dati).

Se abbiamo 2 vie per 8 parole e ogni set ha 2 parole,  
avremo solo 4 set distinti, quindi il set è indicato da solo 2 bit.

Il tag della parola cercata, dev'essere confrontato con entrambi  
i tag (Way 1 e 0) del set indicato

### ARM Assembly Code

```

MOV R0, #5
MOV R1, #0
CMP R0, 0
BEQ DONE
LDR R2, [R1, #0x4]
LDR R3, [R1, #0x24]
SUB R0, R0, #1
B LOOP

```

DONE

$$\begin{aligned} \text{Miss Rate} &= 2/10 \\ &= 20\% \end{aligned}$$

L'associatività riduce i miss di conflitto

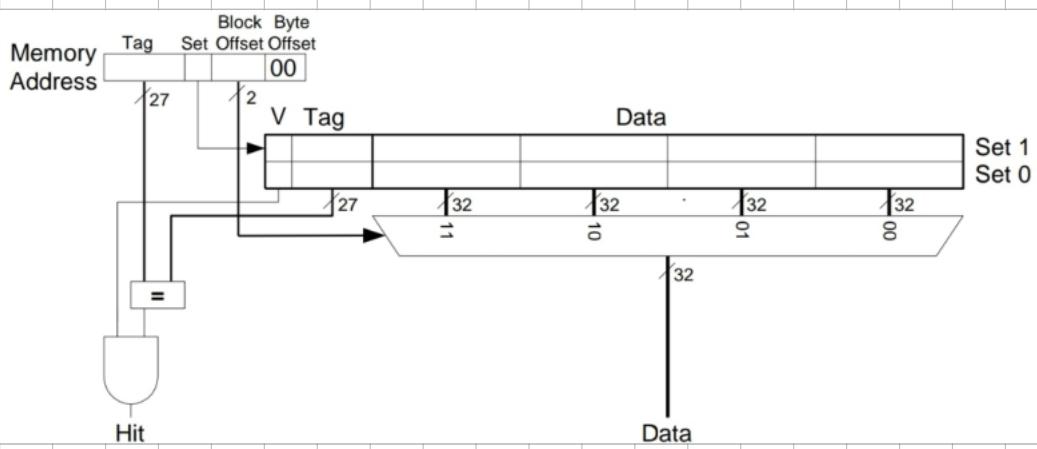
	Way 1		Way 0			
	V	Tag	Data	V	Tag	Data
Set 3	0			0		
Set 2	0			0		
Set 1	1	00...10	mem[0x00...24]	1	00...00	mem[0x00...04]
Set 0	0			0		

Carico prima in Way 0 e dopo Way 1

Il miss rate sarà  $2/10 = 20\%$

### SFRUTTARE LA LOCALITÀ SPAZIALE

Incrementiamo il block size a 4 parole



Vediamo un caso di cache sempre a 8 parole. Ogni block ha 4 parole e siamo in una direct mapped ( $4 \text{ blocks} \Rightarrow 4 \text{ set}$ )

Anzio quindi solo 1 set per il bit e 2 bit di BLOCK OFFSET

(che indica la parola)

Si confronta il tag relativo al set, e il block offset ci indica quale parola

### ARM assembly code

```

MOV R0, #5
MOV R1, #0
LOOP   CMP R0, 0
       BEQ DONE
       LDR R2, [R1, #4]
       LDR R3, [R1, #12]
       LDR R4, [R1, #8]
       SUB R0, R0, #1
       B  LOOP

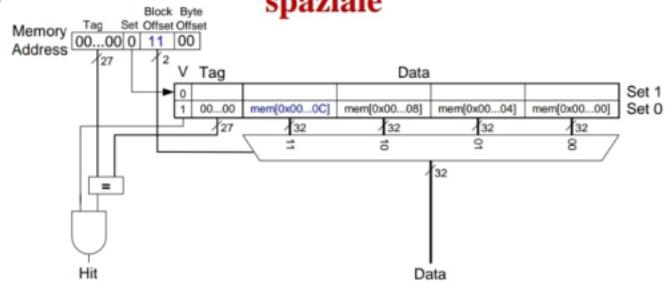
```

DONE

**Miss Rate = 1/15**

= 6.67%

**Blocchi più grandi riducono i miss obbligatori attraverso la località spaziale**



La prima istruzione di load sarà sicuramente un miss, ma caricherà in memoria un blocco di 4 parole (in questo caso 0, 4, 8 e C). Questo comporta che l'istruzione di LDR che cerca l'offset #12 (C in esadecimale) traversa nella cache la parola.

### FULLY ASSOCIATIVE



Percorre una flessibilità maggiore nella scelta di salvataggio delle parole ma risulta molto più dispendioso poiché nei confronti (vanno controllati tutti essendo in uno set)

## BIT DI VALIDITÀ

Indica se l'indirizzo fa riferimento al programma attuale, e quindi va considerato, oppure no.

All'accensione del sistema sono tutti settati a 0

## DMA

Il **DIRECT MEMORY ACCESS** è un meccanismo che permette il trasferimento di dati dalla memoria principale al disco.

Il DMA **NON** è gestito dalla CPU.

I dati sono trasferiti in **PAGINE** (analoghe dei blocchi, ma molto più grande).

Il trasferimento di solito non prevede l'utilizzo della cache.

## SVUOTAMENTO DELLA CACHE

Può capitare, utilizzando un protocollo write back, che trasferendo dati dal disco alla memoria, questi non siano stati ancora aggiornati (copiati dalla cache).

Per ovviare a questo problema si può fare una copia dei

dati su disco con bit di modifica attivo. Quest'operazione è detta **FLUSH** della cache.

Questa necessita di avere la stessa copia di dati in cache e memoria e' detta **COERENZA** della cache.

### ALGORITMI DI SOSTITUZIONE

Anzitutto, nelle cache a indirizzamento **DIRETTO** la posizione di ogni blocco è predefinita quindi non ci sono particolari algoritmi di sostituzione.

Vediamo nelle **N-ASSOCIATIVE** cosa accade:

Prima sostituisci tutti i set con bit di validità 0.

Una volta riempiti tutti i blocchi proseguiamo con la "sovrascrithura" degli altri. Scriviamo la locazione temporale per decidere quale sostituire, usando un algoritmo **LRU** (least recently used) che ci porta a sostituire il blocco con accesso meno recente.

Per realizzare ciò la cache deve mantenere traccia degli accessi e farne "una classifica".

## CONTATORE DI ACCESSI

Gli accessi di ogni set vengono segnati da un numero di bit associato ad ogni set. originariamente

Vediamo come sono gestiti questi bit:

- HIT

Il counter viene messo a 0, i contatori con valori (precedentemente) minori vengono aumentati di 1, gli altri non vengono toccati

- MISS

Abbiamo due possibilità

- ▷ La cache non e' piena, setto il counter a 0 e aumento tutti gli altri di 1
- ▷ La cache e' piena, si rimuove il blocco con counter 3 e si sostituisce con il nuovo blocco (settato a 0), aumentando gli altri di 1

## ALTERNATIVE ALL' LRU

L' LRU risulta poco efficiente se i dati non possono essere tutti contenuti nella cache. In definitiva il caso più efficiente risulta essere sostituire casualmente un blocco.

## TIPI DI MISS

Si dividono in 4 tipi:

- **COMPULSORY**: sono dovuti al primo accesso ad un dato
- **CAPACITY**: la cache è troppo piccola per contenere tutti i dati
- **CONFLICT**: due dati sono indirizzati nella stessa locazione

Definiamo **MISS PENALTY** il tempo necessario ad accedere a blocchi "inferiori"

Blocchi più grandi riducono i miss obbligatori grazie alla facilità spaziale

## FRAZIONAMENTO DELLA MEMORIA

Per ottimizzare le prestazioni di trasferimento dati, si vuole avere la stessa velocità di trasferimento per ogni dato.

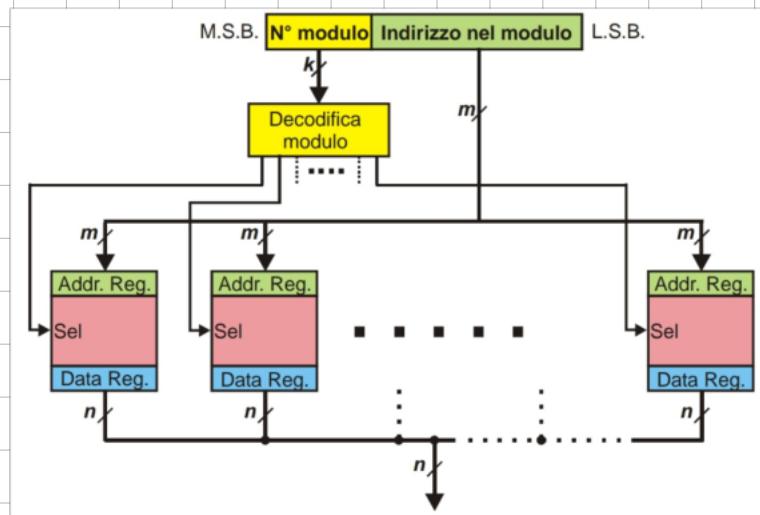
Già è possibile distribuire la memoria in **MODULI PARALLELI**

ognuno con il proprio registro degli indirizzi e dei dati, permettendo di accedere contemporaneamente a più moduli.

Questa distribuzione può avvenire in 2 modi:

- **NON INTERLACCIAZATO**

l'indirizzo di memoria generato dalla CPU viene decodificato nel seguente modo: I **k** bit più significativi indicano uno degli **n** moduli paralleli, gli **m** bit meno significativi indicano una particolare parola del modulo.



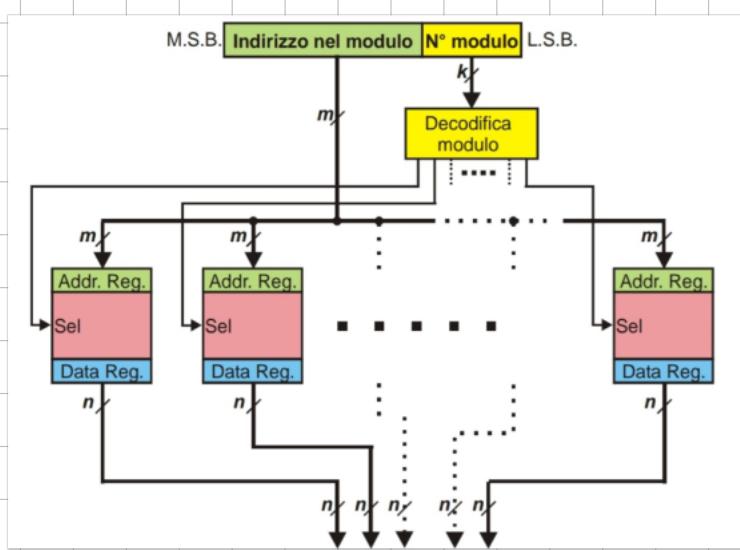
In questo caso viene interpellato solo un modulo alla volta, trasferendo il contenuto del data register alla CPU **selezionalmente**.

## ● INTERLACCIAUTO

I  $K$  bit meno significativi individuano il modulo e gli  $m$  bit più significativi indicano la locazione all'interno del modulo.

Cose paeno i indirizzi consecutivi sono posizionati in moduli successivi, ottimizzando velocità di accesso e utilizzo della memoria.

Si osservi che avremo bisogno di  $2K$  moduli per evitare stati vuoti.



## CACHE MULTILIVELLO

In genere le cache hanno 2 o 3 livelli. Questo implica che tutti i ragionamenti si svolgono su più livelli.

Esistono 2 tipi di cache multilivello:

## ● INCLUSIVA

L2 contiene tutti i dati di L1

- L2 contiene sia i dati correnti di L1, sia quelli che erano presenti in precedenza e sono stati sovrascritti mediante l'algoritmo di sostituzione (copy back)
- Per mantenere la coerenza L1 ha una politica di write-through verso L2, e così L2 verso L3
- Miss in L1 e hit in L2: il blocco relativo viene copiato da L2 a L1 (in caso di sostituzione un dato va da L2 a L1 e un'altro da L1 a L2)
- Miss in L1 e L2: il blocco dalla main memory viene copiato sia in L1 che L2

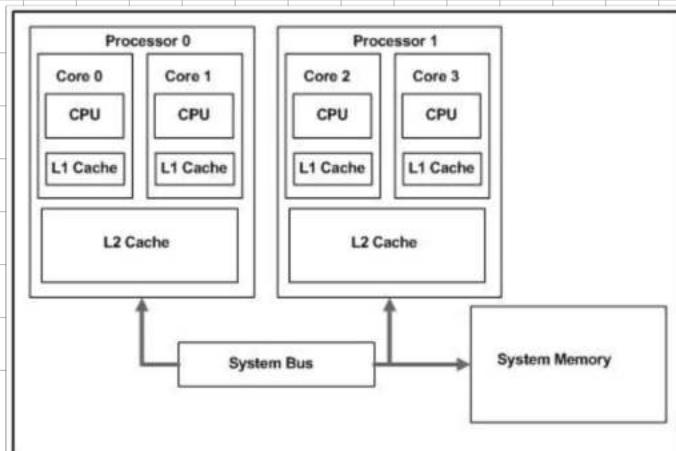
## ● ESCLUSIVA

L2 contiene solo i dati di copy back da L1

- La memoria complessiva della cache è data dalla somma delle capacità dei vari livelli, in realtà nelle moderne architetture la capacità di L2 è un ordine di grandezza rispetto a quella di L1, e così fra L3 e L2, il guadagno rispetto alle cache inclusive non è molto significativo).
- 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 memorizzata in L2 e la linea di L2 in L1.
- Miss in L1 e L2: Il dato letto dalla memoria è memorizzato direttamente in L1 e la linea di cache rimpiazzata di L1 (*victim data*) è trasferita in L2 rimpiazzando un'altra linea di cache secondo la politica di rimpiazzo usata.

## CACHE CONDIVISE

Nel caso di processori a più core (cpu) i livelli delle cache possono essere condivisi tra questi (in particolare i livelli "più alti"). Risulterà che: se uno dei due core che condividono un livello non è in funzione, l'altro avrà comunque più cache; ottimizza la programmazione multi-threading.



## MEMORIA VIRTUALE

Il Sistema operativo si occupa di assegnare parti di memoria centrale al programma che si sta eseguendo così che il programma abbia più spazio sul quale operare (non sempre questo spazio è contiguo).

Programmi più grandi vengono così dislocati nella memoria centrale attraverso un'operazione di "**SWAPPING**", ovvero la sostituzione dinamica dei segmenti del programma nello spazio assegnato.

Sono operazioni pesanti vista l'enorme differenza di velocità tra memoria centrale e disco.

Il funzionamento è quanto analogo al funzionamento di una cache, incluso l'algoritmo di sostituzione.

Le Pagine (in cui è suddiviso il programma) però sono molto più grandi dei blocchi.

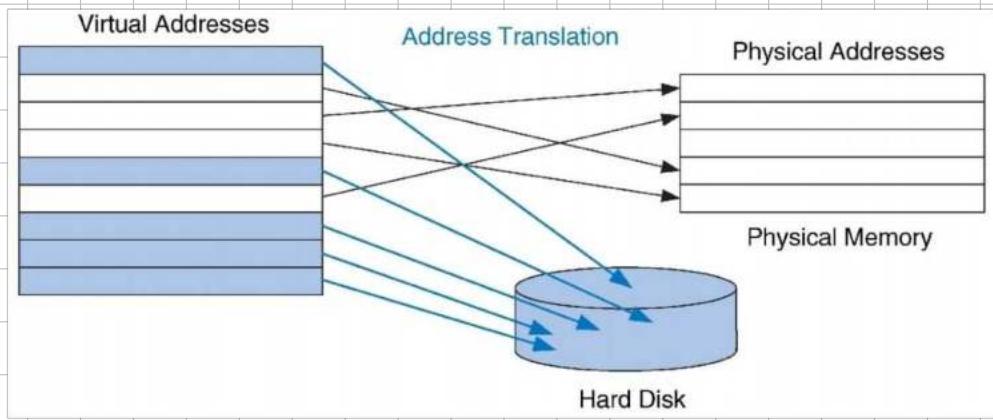
## GESTIONE DELLA MEMORIA VIRTUALE

I programmi quindi usano indirizzi virtuali (spazi di memoria nell'hard drive). Essendo la memoria centrale più piccola della memoria di massa, solo una parte dei programmi è contenuta in main memory.

Quindi, analogamente a una cache, gli indirizzi virtuali sono (in parte) copiati sulla main memory, se un dato non viene trovato in quest'ultima si accede alla memoria di massa.

In questo caso manca però un mapping specifico in quanto lo swapping è gestito dal sistema operativo (non esendo così frequenti) attraverso una tabella di traduzione indirizzi fisici → virtuali ottenendo una sorta di forma full associative (senza mapping) eludendo l'accesso a tutte le pagine grazie alla tabella di traduzione

Cache	Virtual Memory
Block	Page
Block Size	Page Size
Block Offset	Page Offset
Miss	Page Fault
Tag	Virtual Page Number



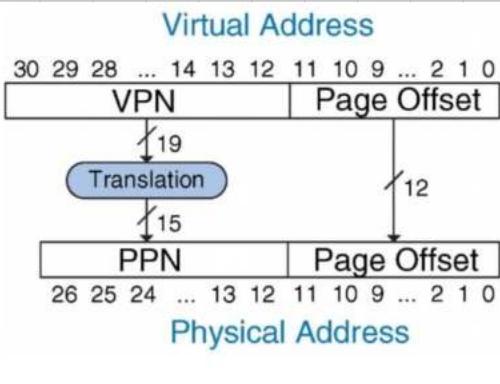
Es.

### System:

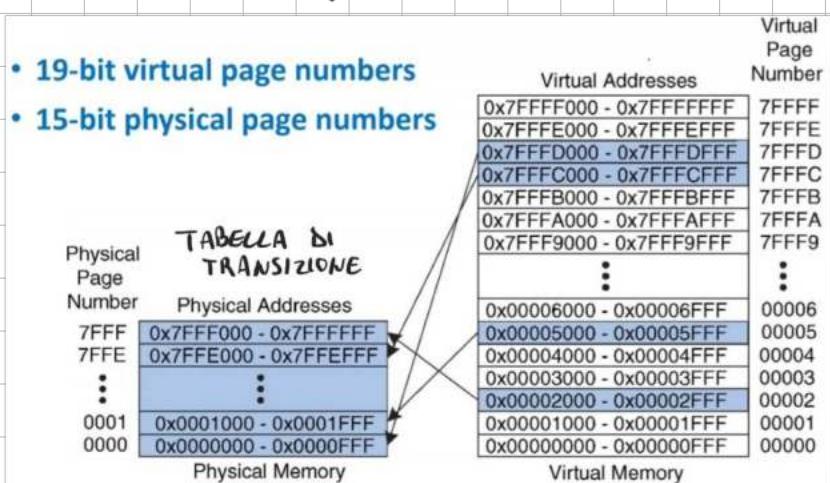
Virtual memory size: 2 GB =  $2^{31}$  bytes  
Physical memory size: 128 MB =  $2^{27}$  bytes  
Page size: 4 KB =  $2^{12}$  bytes

### Organization:

Virtual address: 31 bits  
Physical address: 27 bits  
Page offset: 12 bits  
# Virtual pages =  $2^{31}/2^{12} = 2^{19}$  (VPN = 19 bits)  
# Physical pages =  $2^{27}/2^{12} = 2^{15}$  (PPN = 15 bits)



I primi 12 bit del virtual address indicano a quale indirizzo mio sto riferendo all'interno di una pagina, i restanti indicano il virtual page number (l'indirizzo virtuale) che dev'essere tradotto in un physical page number (da 19 a 15 bit), mentre il page offset viene basilmente ricopiatò



Il virtual address arriva fino a 7 perché  $2GB = 31$  bit quindi  $2^8$  (abbiamo bisogno di 8 bit per gli esadecimali) con il bit più

significativo fino a 7.

Ogni pagina e' di  $4\text{Kb} = 12$  bit  $\rightarrow$  3 esadecimoli (ultime 3 cifre dell'indirizzo).

Qual e' l'indirizzo fisico di  $0x2\text{47C}$ ?

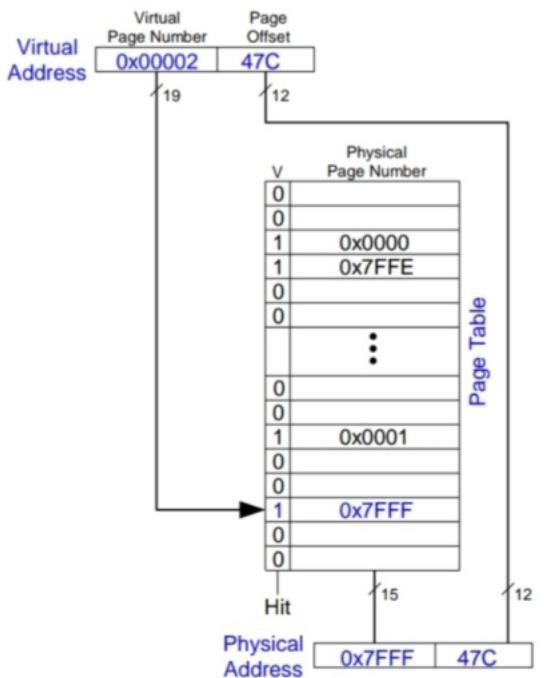
47C sara' il page offset. Il VPN sara'  $0x00002$ , viene mappato in FFFF (vedi tabella) come PPN.

Aggiungendo il page offset sappiamo che l'indirizzo e'  $0x7FFF\text{47C}$

La traduzione da VPN a PPN e' effettuata dal **MEMORY MANAGEMENT UNIT (MMU)**

## PAGE TABLE

E' una tabella che ha un entry per ogni pagina virtuale ed ognuna di queste ha come campi un bit di validita' (se la pagina e' presente in MM) e il PPN



Ese.

Qual è l'indirizzo fisico di 0x5F20?

VPN = 5, prendiamo la 5 entry  $\Rightarrow$  PPN = 1

Il bit di validità vale 1,

P. address = 0x1F20

Se V=0 allora il dato non è presente in

MM, dobbiamo accedere alla memoria di

mossa.

Physical Page Number	
0	
0	
1	0x0000
1	0x7FFE
0	
0	
	⋮
0	
0	
1	0x0001
0	
0	
1	0x7FFF
0	
0	

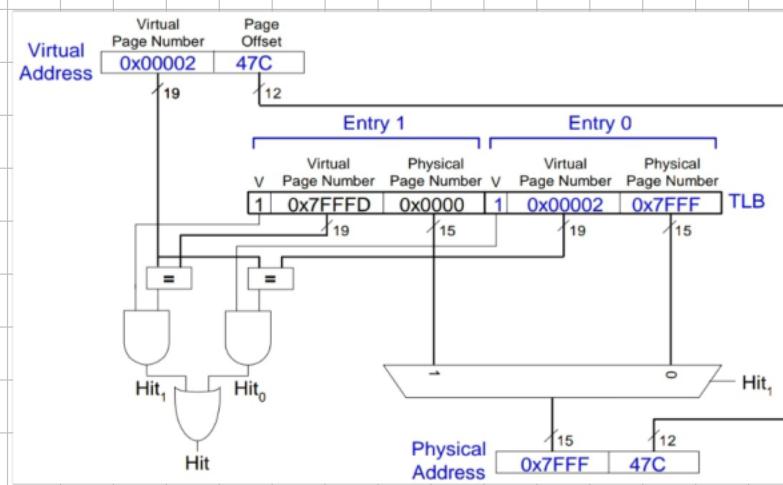
### TLB

Se il dato non è presente in MM avremo bisogno di 2 accessi

(uno alla page table e uno alla memoria di mossa).

Per ottimizzare usiamo una **TRANSLATION LOOKASIDE BUFFER**

ovvero una piccola cache che tiene conto delle traduzioni recenti,  
sfruttando la validità temporale



2-entry TLB







