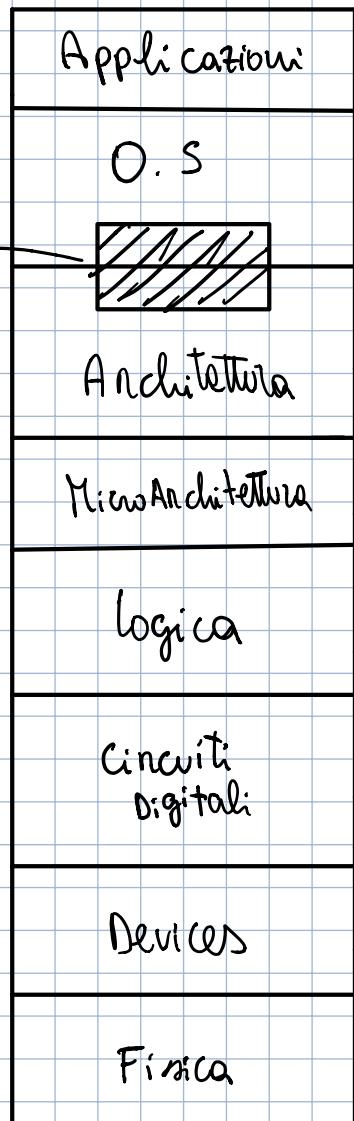


Livelli di Astrazione

Astrazione

- Poter utilizzare componenti di un livello superiore senza preoccuparsi come sono fatti quelli inferiori
- Ogni livello ha la sua implementazione che può essere utilizzata a livelli superiori



Operazioni che vengono messe a disposizione dalla microarchitettura

→ (Arm) x86

→ Processore

→ Dispositivo con un insieme di porte logiche

→ ALU (2 input, 1 controllo, 1 output)

→ Porte logiche

→ Transistor

Interfaccia

↓
Applicazione con le
quali si può andare
a livelli superiori

↓

Operazioni che vengono
messe a disposizione
dal livello sottostante
a quello superiore
senza preoccuparsi
di come sia fatto

↓

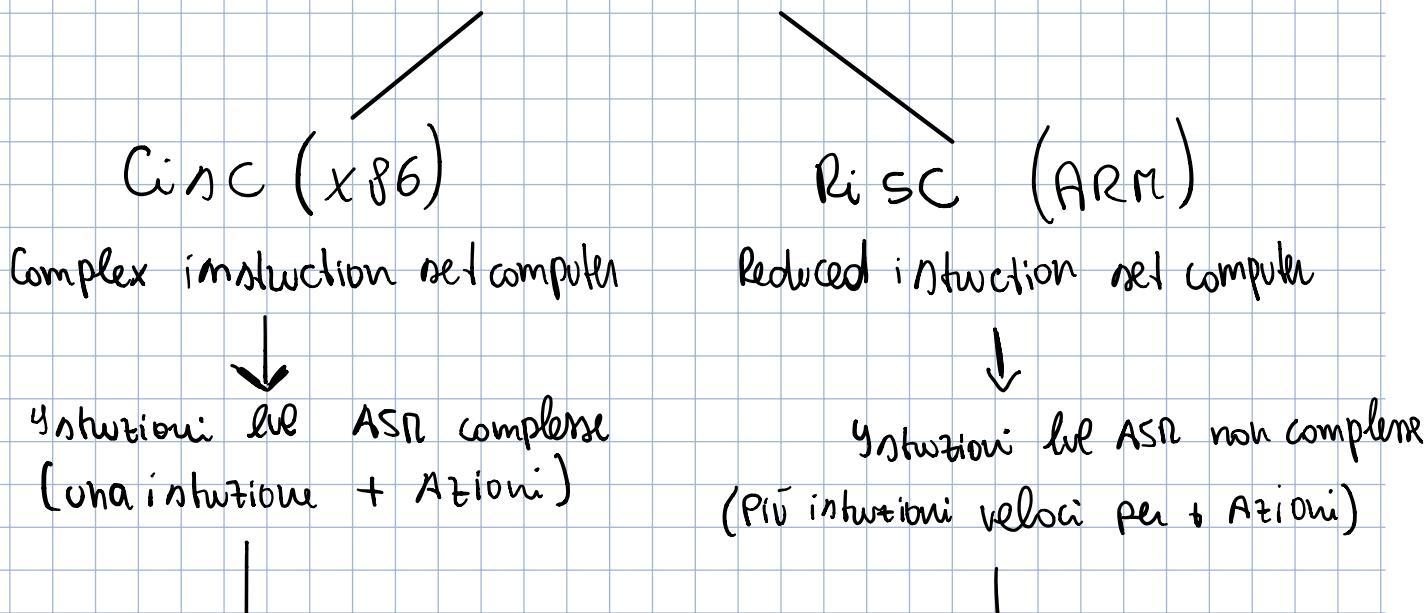
Un programma
che non utilizza
le interfacce viene

ridotta la portabilità

Principi fondamentali

- Genericità: Vedere componenti difficili scomponibili in più semplici
- Modularità: Mettere insieme più componenti / moduli (con diverse funzionalità per avere funzionalità più complete).
 - g moduli devono essere:
 - a) Autonomi : Proprio controllo
 - b) Indipendenti : Possibilità di svolgere azioni in autonomia
 - c) Sequentiali : Svolgere un'azione alla volta
- Regolarità : Usare moduli comuni più volte, riducendo il numero di volte che devono essere redisegnati.
- Disciplina: Azioni che ridurranno volontariamente le scelte di design in modo da lavorare più produttivamente ad alto livello di astrazione.

Calcolatori



Ystitutioni diverse e diverse tra loro, gli state nel caso sono diversi

↓
Di seguito ineguale

↓
Difficile riprodurla
sul c.p.

Case più piccole ed elementari

↓
Rete porte logiche

↓
Di seguito più regolare

con k bit rappresento 2^k numeri ma il numero più grande rappresentabile è $2^k - 1$

Arithmetica binaria

$$\begin{cases} 0 = F = 0V \\ 1 = T = 5V \end{cases}$$

- trasformazione Da binario a decimale

$$2^7 \ 16 \ 8 \ 4 \ 2 \ 1$$

$$101101_2 = 32 + 8 + 4 + 1 = 45_{10}$$

Prendo le posizioni base 2 e le moltiplico per $\times 01$. Alla fine sommo

- Da Decimale a binario :

$$\begin{array}{r} 132 \\ \hline 2 | 2 \\ 0 | 61 \\ 1 | 30 \\ 0 | 15 \\ 1 | 7 \\ 1 | 3 \\ 1 | 1 \\ 1 | 0 \end{array}$$

64 32 16 8 4 2 1
 $1111011_2 = 132_{10}$

Scivo al contrario

$$\begin{array}{r} 132 \\ \hline 64 \\ 59 \\ \hline 32 \\ \hline 27 \\ \hline 16 \\ \hline 11 \\ \hline 8 \\ \hline 3 \end{array}$$

64 32 16 8 2
Nou vero
il 6
bit a 0

$$\frac{2}{1}$$

Moltiplicare e dividere x2:

- Moltiplicare: $101 = 5 \Rightarrow 1010 = 10$
 $\times 2$

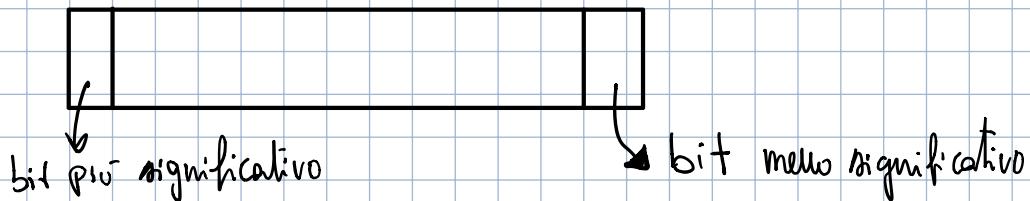
- Dividere: $1010 = 10 \Rightarrow 101 = 5$
 $: 2$

- Somma e prodotto tra binari

$$\begin{array}{r} 111 \\ 1011 + \\ 1101 = \\ \hline 11000 \end{array}$$

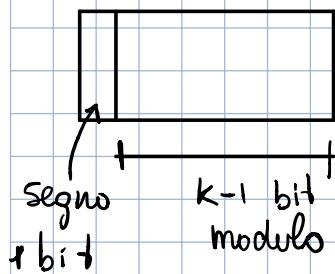
$$\begin{array}{r} 1101 \cdot \\ 11 = \\ \hline 1101 + \\ 11010 = \\ \hline 100111 \end{array}$$

bit più significativo e meno significativo



- Numeri Relativi

1) Modulo e segno



$$\text{Da } (-2^{k-1} + 1) \text{ a } (2^k - 1)$$

Problema della
doppia rappresentazione

Io \varnothing si può rappresentare
in 2 modi

Somma e moltiplicazione
servono circuiti diversi

2) Complementi a 2

a) $n_{10} \geq 0$ binario
normale

b) $n < 0$

b.1) Binario Normale

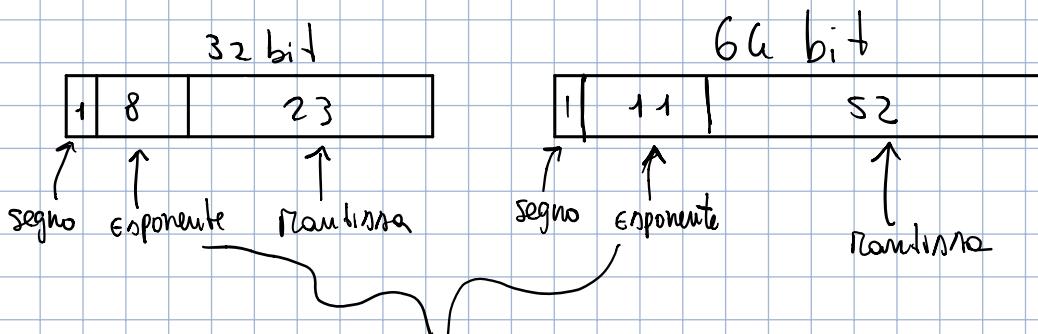
b.2) Complemento di tutti i bit

b.3) Sommo 1 ai bit

$$\text{Da } -2^{n-1} \text{ a } 2^{n-1} - 1$$

-1 perché con una ci napp. lo Ø

- Numeri con le virgole



Sottraggo la metà per sapere in quale direzione mettere le virgole

Potenze di 2

2^n	0	1	2	3	4	5	6	7	8	9	10	11	12	13
res	1	2	4	8	16	32	64	128	256	512	1024	2048	4096	8192

Bit:

Nibble = 4 bit = 16 possibilità

$$K = 2^{10} = 1024 \text{ byte}$$

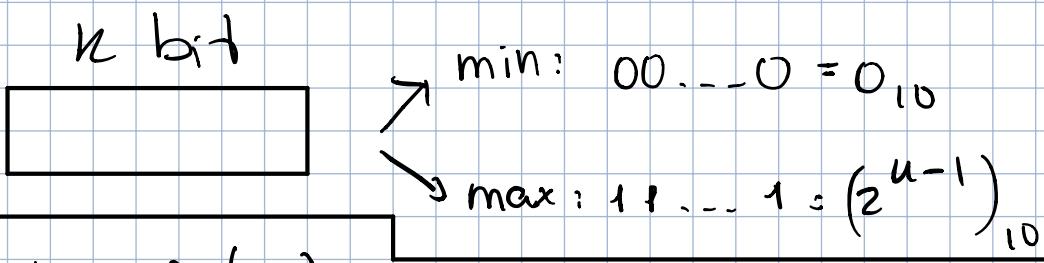
byte = 8 bit = 256 possibilità

$$M = 2^{20} = 1024 \cdot 1024 \text{ byte}$$

word = Quantità min
su cui neppure
intervi

$$G = 2^{30} \approx 4 \text{ milioni}$$

$$T = 2^{40} \text{ byte}$$



Base esadecimale (16)

$$10_{10} = A$$

bx

costanti
esadecimali

$$15_{10} = F$$

	ESQ	DECI	BINARIO
0	0	0	0000
1	1	1	0001
2	2	2	0010
3	3	3	0011
4	4	4	0100
5	5	5	0101
6	6	6	0110
7	7	7	0111
8	8	8	1000
9	9	9	1001

Dove min value

J

0xFF max Value

A	10	1	0
B	11	0	1
C	12	1	0
D	13	1	0
E	14	1	1
F	15	1	1

Porte logiche

AND

x_1	x_2	z
0	0	0
0	1	0
1	0	0
1	1	1



OR

x_1	x_2	z
0	0	0
0	1	1
1	0	1
1	1	1



XOR

x_1	x_2	z
0	0	0
0	1	1
1	0	1
1	1	0



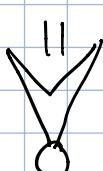
NAND

x_1	x_2	z
0	0	1
0	1	0
1	0	0
1	1	0



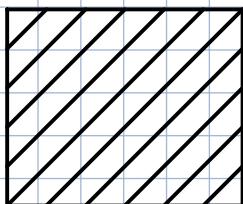
NOR

x_1	x_2	z
0	0	1
0	1	0
1	0	0
1	1	0

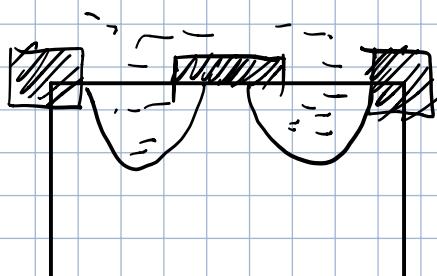


Semi conduttori

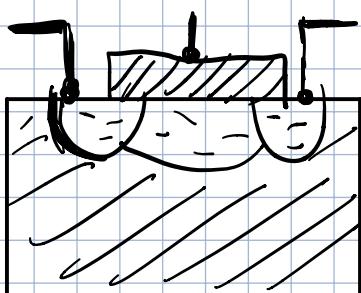
(si può comportare da conduttore o isolante
a seconda delle situazioni)



Silicio



1) Dopo il silicio con iioni negativi/
positivi e crea delle parti positive/
negative sul silicio

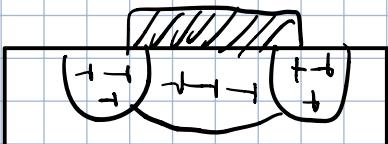


transistor (NMOS)

2) Leno gli isolanti

3) Se do una tensione al centro
riesco a creare un canale che
trasferisce corrente.

In questo modo ho creato un
intero circuito.

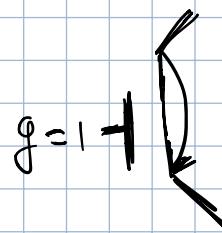
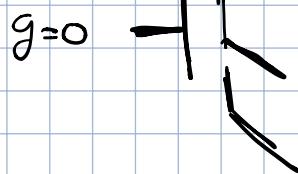
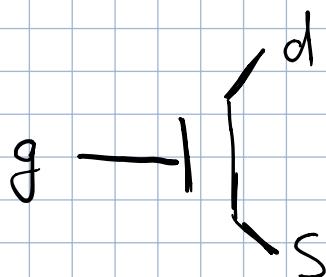


Inversione che a seconda se metto S volt al centro o no, crea un paesaggio di elettricità.

↓
transistor (PNP)

Rappresentazione NPN e PNP

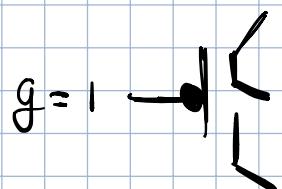
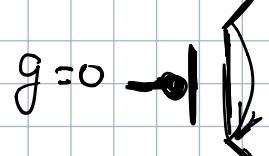
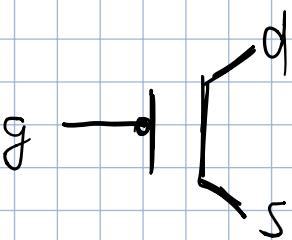
NPN:



circuito
aperto

circuito
chiuso

PNP:



circuito
chiuso

circuito
aperto

Cos'è la **Potenza** = Quantità di energia utilizzata per unità di tempo

↳ Dinamica: Potenza utilizzata per caricare la capacità dei condensatori quando i segnali cambiano tra 0 e 1

$$P_{\text{dinamica}} = \frac{1}{2} C \underbrace{V_{DD}^2}_{\text{Frequenza con cui cambia la tensione sul condensatore}} f$$

↳ energia prelevata dall'alimentatore per caricare la capacità C fino alla tensione V_{DD}

Frequenza con cui cambia la tensione sul condensatore

↳ Statica: Potenza consumata quando il sistema è inattivo

$$P_{\text{statica}} = I_{DD} V_{DD} \xrightarrow{\text{tensione}} \text{costante di dispersione}$$

Fermiologia

$$1) \text{ Not}(A) = \bar{A} \quad 2) \text{ AND}(A, B) = A \cdot B \quad 3) \text{ OR}(A, B) = A + B$$

A	B	C	Z
0	0	0	1
0	0	1	1
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	0
1	1	1	1

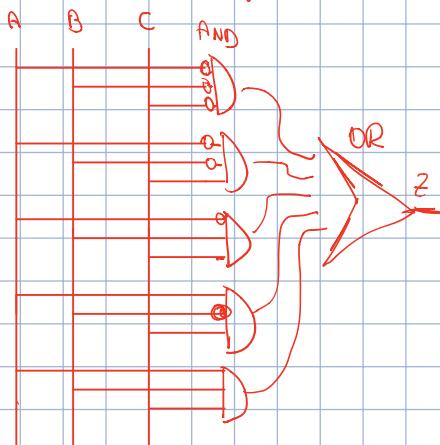
Minterm: Prodotto di tutte le variabili eventualmente negative

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

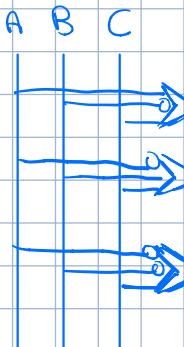
Maxterm: Somma di tutte le variabili eventualmente negative

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

Somma di prodotti



Prodotto di somme



Azioni

$$1) \text{ Not} \quad A=0 \rightarrow \bar{A}=1$$

$$A=1 \rightarrow \bar{A}=0$$

$$2) \text{ AND} \quad \begin{array}{ll} 00 \rightarrow 0 & 10 \rightarrow 0 \\ 01 \rightarrow 0 & 11 \rightarrow 0 \end{array}$$

$$3) \text{ OR} \quad \begin{array}{ll} 00 \rightarrow 0 & 10 \rightarrow 1 \\ 01 \rightarrow 1 & 11 \rightarrow 1 \end{array}$$

Teoremi

$$1) A \cdot 1 = A, \quad A + 0 = A \quad (\text{identità})$$

$$2) A \cdot 0 = 0 , A + 1 = 1 \quad (\text{Nullo})$$

$$3) A \cdot A = A , A + A = A \quad (\text{idempotenza})$$

$$4) A \cdot \bar{A} = 0 ; A + \bar{A} = 1 \quad (\text{complementi})$$

$$5) A+B = B+A ; A \cdot B = B \cdot A \quad (\text{commutativa})$$

$$6) (A \cdot B) \cdot C = A \cdot (B \cdot C) ; A+(B+C) = (A+B)+C \quad (\text{associativa})$$

$$7) (A+B) \cdot C = (A \cdot C) + (B \cdot C) ; (A \cdot B)+C = (A+C) \cdot (B+C) \quad (\text{distributiva})$$

$$8) A+B = \overline{\overline{A} \cdot \overline{B}} \quad ; \quad \overline{A+B} = \overline{A} \cdot \overline{B} \quad (\text{de Morgan})$$

Mappe di Karnaugh \rightarrow Reti di grafico risoluzione
di espressioni booleane

- Da tabella di verità a Karnaugh

A	B	C		2
0	0	0	1	\overline{AB}
0	0	1	1	\overline{C}
0	1	0	0	
0	1	1	0	
1	0	0	0	
1	0	1	0	
1	1	0	0	

Formule Booleane

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

1 1 1 | 0

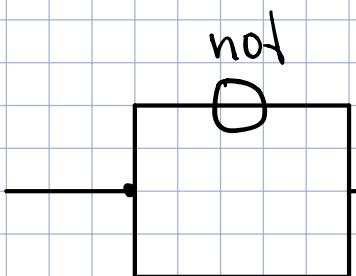
Prendo gli 1 vicini (esponente di 2) e li sommo tra loro. Posso prendere anche 2 righe una in cima e una in fondo, o 2 colonne una a destra una a sinistra. Devo fare i cerchi in modo che siano i più grandi possibili

Esempio:

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

Se ci sono dei non specificati "-" li prendo come "1"

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

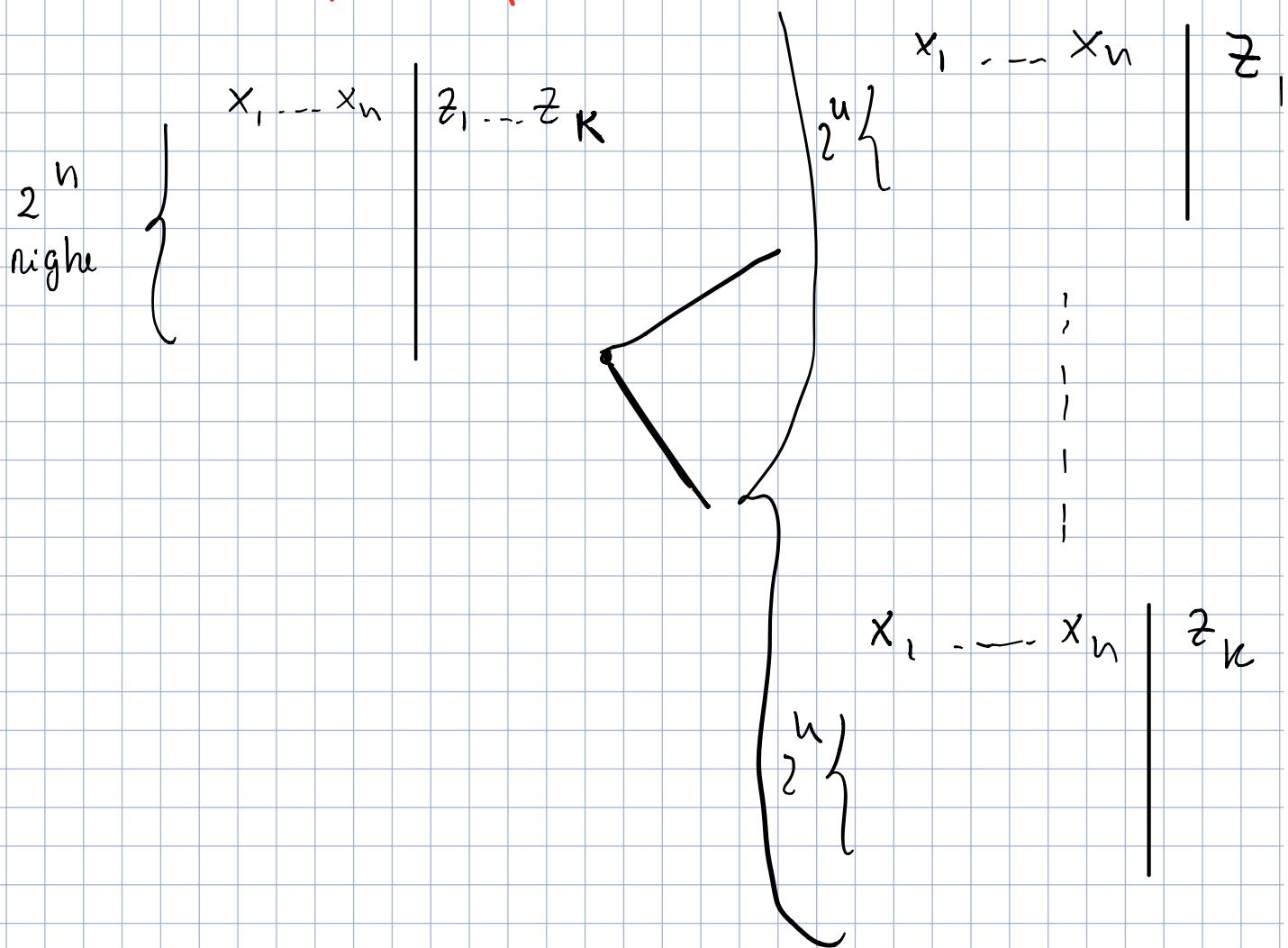


$x \rightarrow$ non so quanto vale perché ho solo
discreti

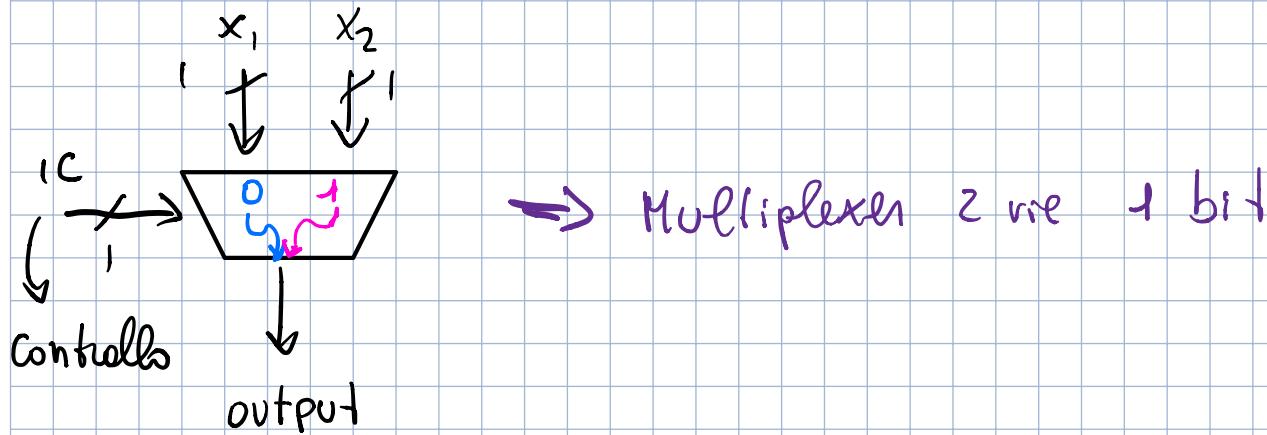
Valore
sconosciuto/illegal

Un'altra uscita che mi può avere è la "z" = Valore col alto
impendente
↓
Uscita che non ha nessun contributo

Circuiti con più output



Multiplexer (commutatore) \Rightarrow chiamato "mux"



Se il controllo è uguale a 0 \Rightarrow Collega lo "0"

Se il controllo è uguale a 1 \Rightarrow Collega l'"1"

Il numero di ingressi di controllo è pari a $\boxed{\log_2(\text{ingressi})}$

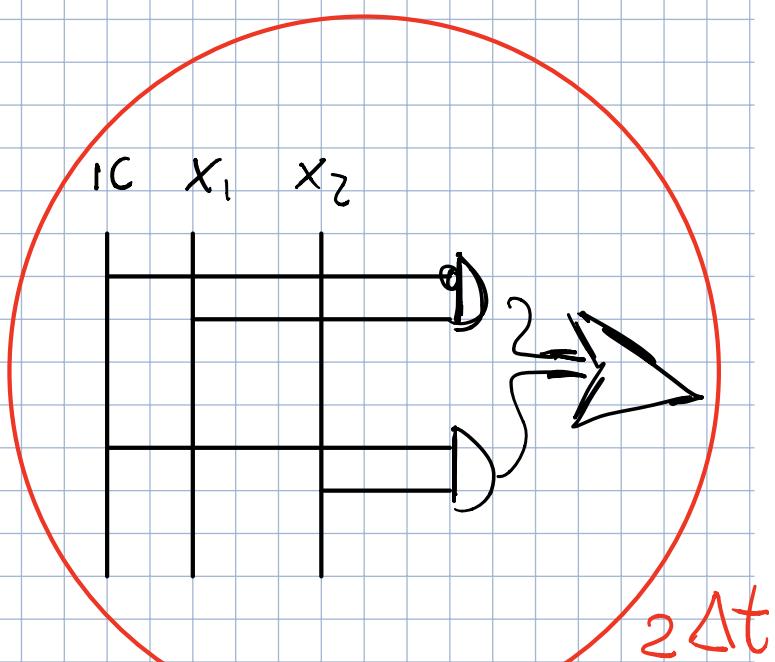
Tabelle semplificate multiplexer

IC	x_1	x_2	Z
0	1	-	1
-	-	1	1

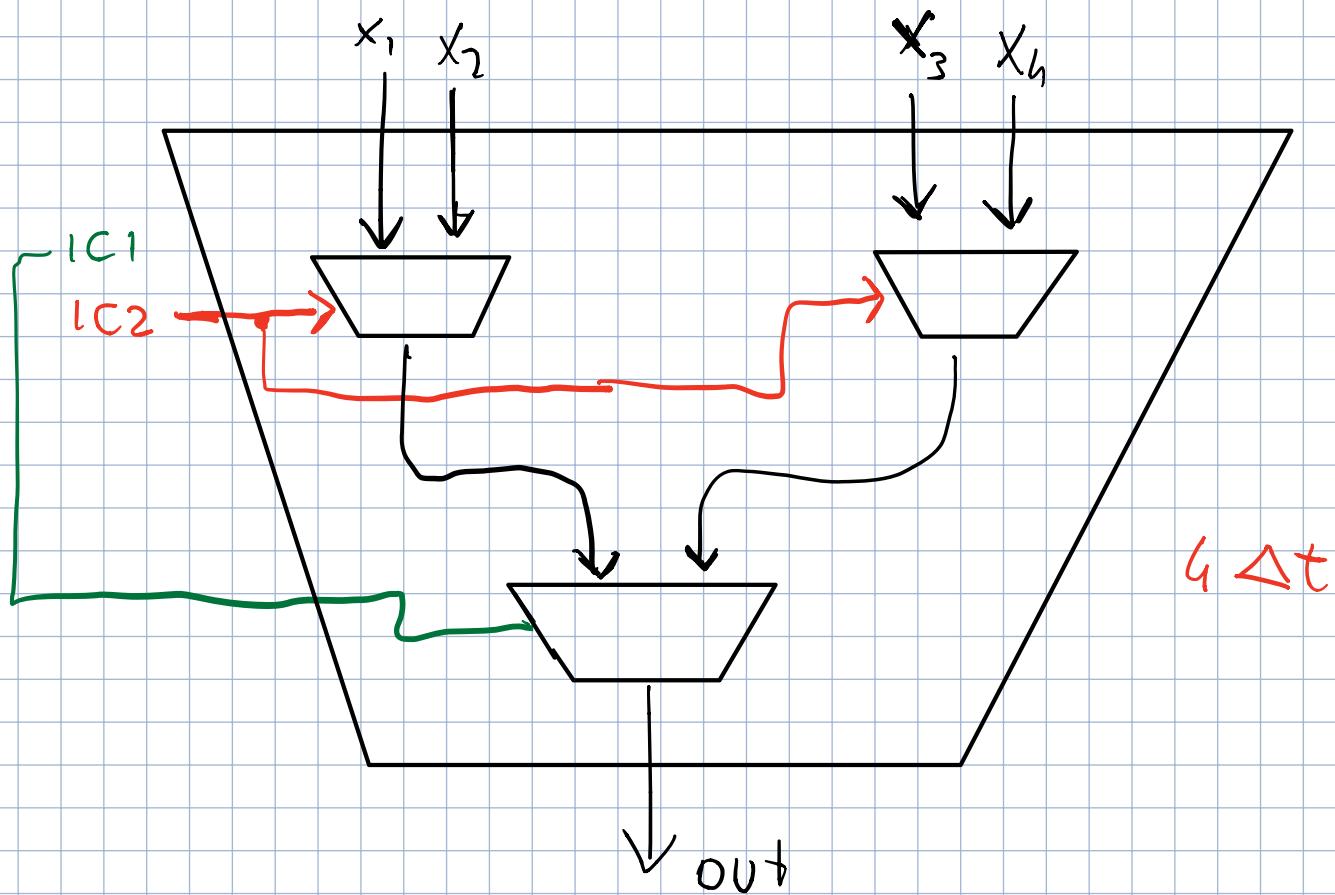
\rightarrow In generale si ha una diagonale sugli ingressi

$$\downarrow$$

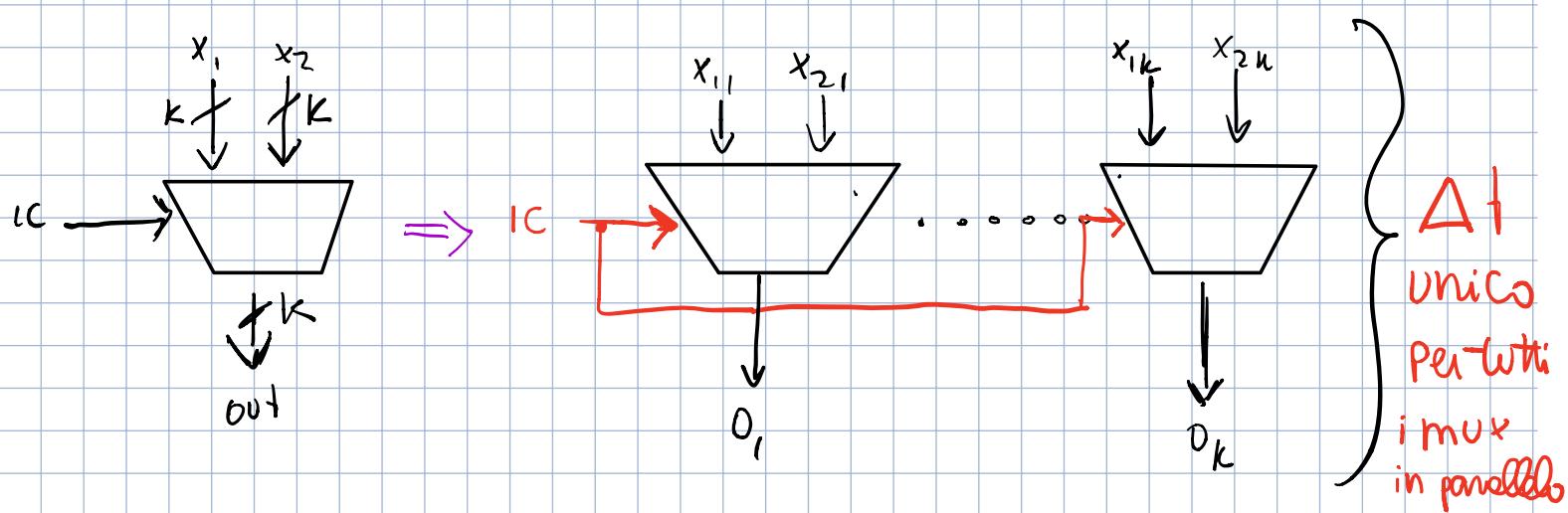
$$Z = (\overline{IC} \cdot x_1) + (IC \cdot x_2)$$



Multiplexer 4 ingressi 1 bit

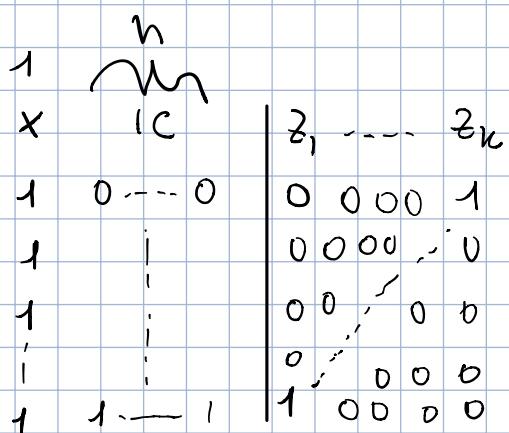
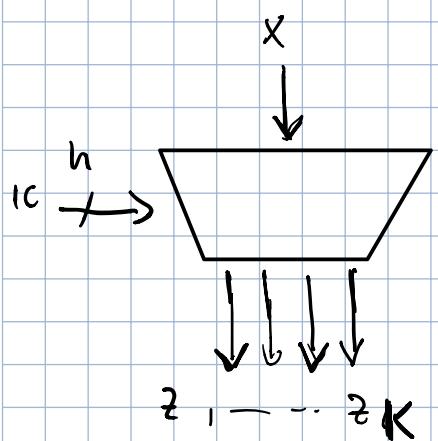


Multplexer 2 vie K bit



Demultiplexer (decoder) → 1 ingressi K uscite

n bit di controllo



Sceglie l'uscita da mettere a 1 in base all'ingresso di controllo

tempo multiplexer e demultiplexer

Multiplexer:

Demultiplex

1) Livelli AND: $\log_K (\#(\text{ingressi} + i))$

1) Livelli AND: $\log_{10}(\#ic + 1)$

$$2) \text{ livelli OR: } \log_K \left(2^{*(\text{ingressi} + \text{ic})} \right)$$

2) livelli OR: Nenuno

Se uso il multiplexer per leggere (bit da cercare come ingresso di controllo)

e il Demultiplexer per scrivere

Scrivere costa meno che leggere

Costo di un circuito

livelli AND : $\log_k(n)$ $n = \text{** ingressi}$

livelli OR : $\log_k(2^n)$ $k = \text{Num max di ingressi per porta}$

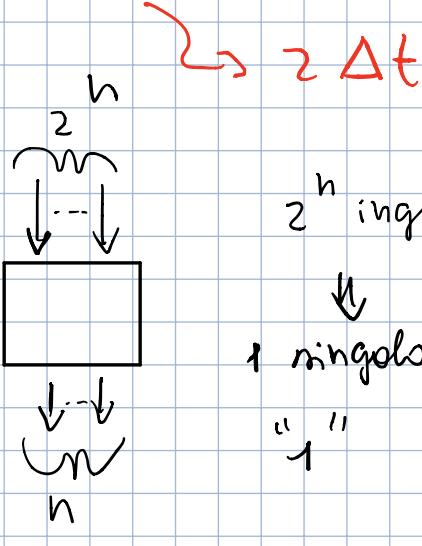
Cambio di base: $\log_8 n = \frac{\log_2 2^n}{\log_2 2^3}$ dove $2^n = n$

Ingressi : - Caso AND: Numero ingressi totali tab. di verita'

- Caso OR: Numero di "1" tab di verita'

$$\Delta t \text{ finale} = \Delta t (\#lvl(\text{AND}) + \#lvl(\text{OR}))$$

Coder \Rightarrow Da le posizioni dell'ingresso che sta domando l'"1"

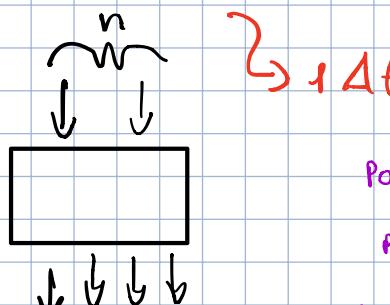


2^n ingressi $\Rightarrow n$ uscite
1 ingresso
"1"
 n bit che
rappresentano
la posizione

a	b	c	d	z	t	posizione
0	0	0	1	0	0	0
0	0	1	0	0	1	1
0	1	0	0	1	0	2
1	0	0	0	1	1	3

on "1" per riga

Decoder \Rightarrow Dalle posizioni da ie bit a "1"



Posizione 0 \rightarrow 0 0
Posizione 1 \rightarrow 0 1

a b z v u v t

0 0 0 1

0 0 1 0

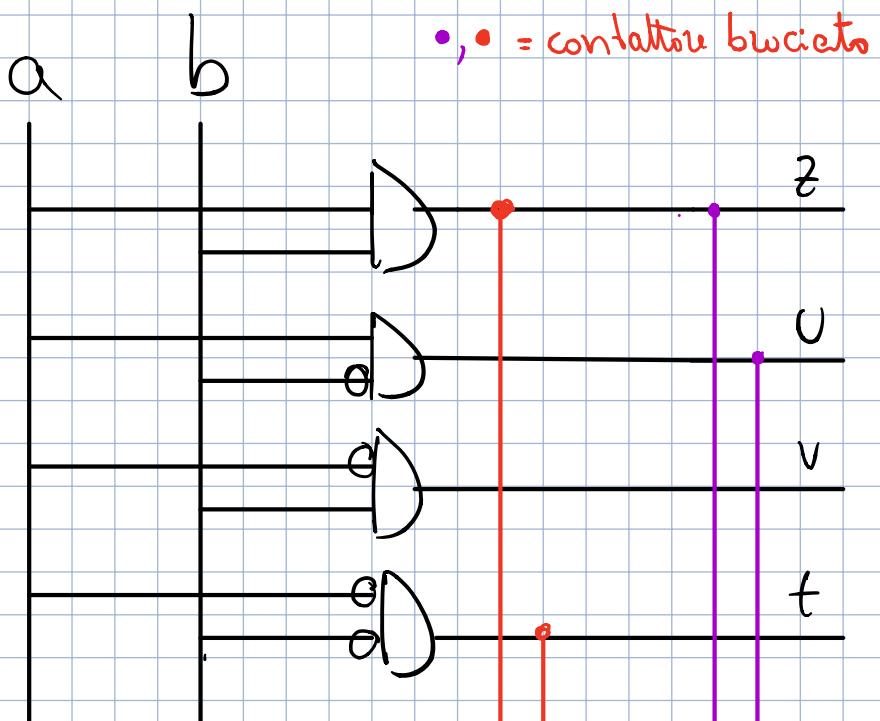
0 1 0 0

1 0 0 0



 Positione 2 \rightarrow 1 0
 Positione 3 \rightarrow 1 1

Nenun
OR



Con l'output del decoder posso fare
più tabelle di verità
collegando i fili
ad un OR
(Bivalvi i contattori)

\rightarrow utile perché posso
creare tanti blocchetti
uguali e farli:
collegamenti in port
produzione. Quindi:
le cose del chip
è uguale per tutti

a	b	z	a	b	z
0	0	1	0	0	0
0	1	0	0	1	0
1	0	0	1	0	1
1	1	1	1	1	1

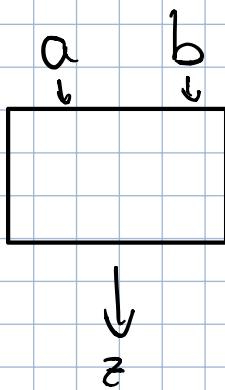
- Se metto dentro un decodificatore una memoria posso creare una memoria rescrivibile (RAM)
- Se bivalvi i contattori non posso riceverli (ROM)

Ritardi di propagazione e contaminazione

- 1) Ritardo di propagazione (t_{pd}) : tempo massimo che trascorre dal momento in cui avviene un cambiamento nell' ingresso al momento in cui l' uscita raggiunge il suo valore finale.
- 2) Ritardo di contaminazione (t_{cd}) : tempo minimo che trascorre dal momento in cui cambia l' ingresso al momento in cui una qualsiasi uscita comincia il processo di adattamento del suo valore.

Confrontatore ($z = 1$ se $a = b$)

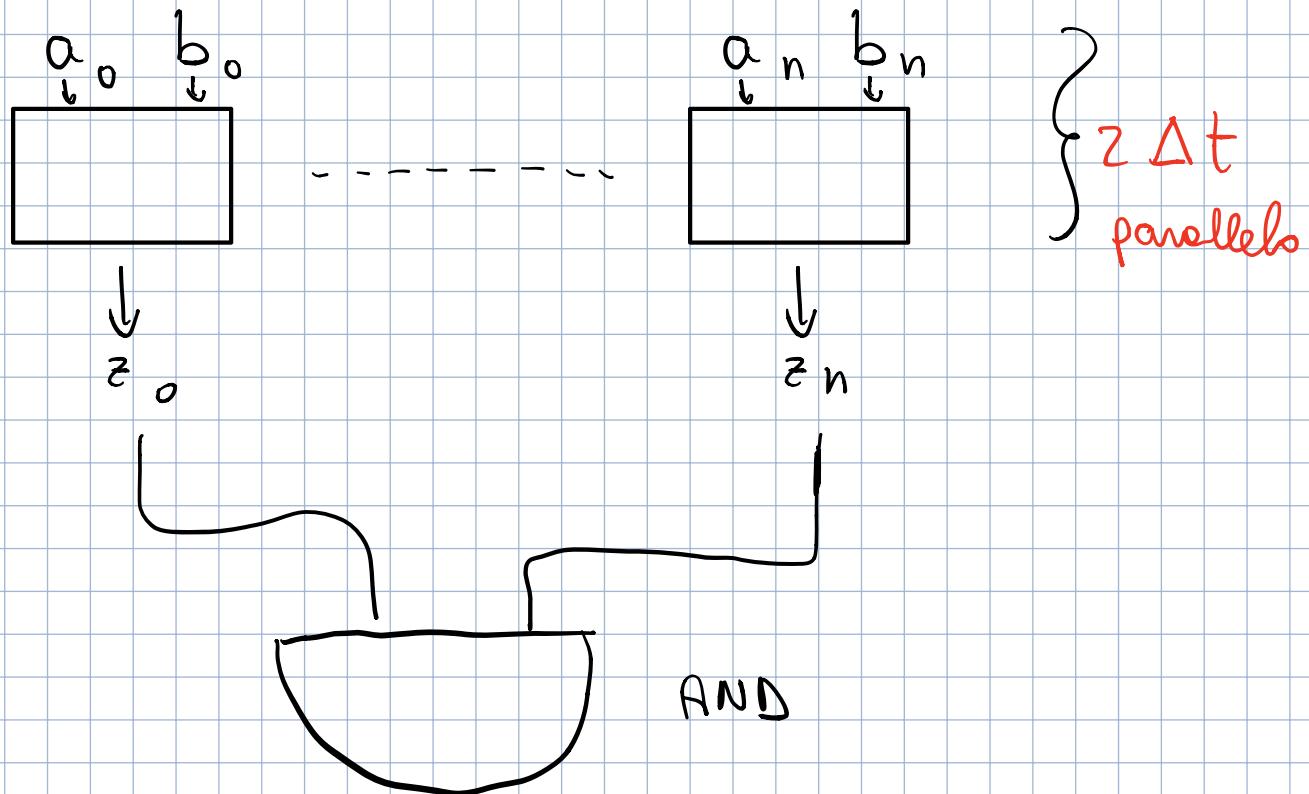
$2 \Delta t$



a	b	z
0	0	1
0	1	0
1	0	0
1	1	1

$$z = \bar{a}\bar{b} + ab \text{ (XOR)}$$

Confrontatore da n bit

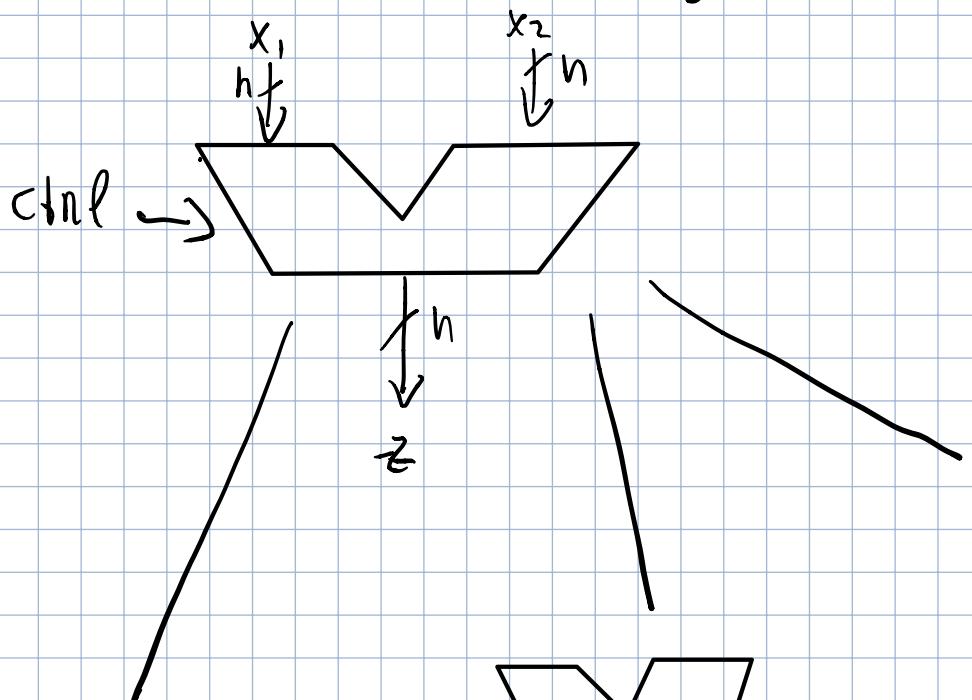


Costo:

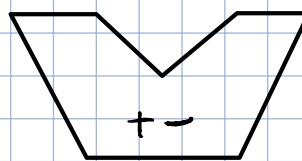
lvl confrontatori: $2 \Delta t$

lvl AND: $\log_k (\# \text{confrontatori}) > 2 \Delta t + (\log_k \# \text{confrontatori}) \Delta t$

ALU (Arithmetic Logic Unit)

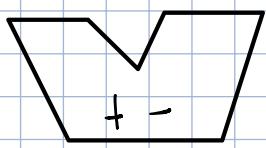


Sono di comp.
ripetuti o tab.
di venire -



FP

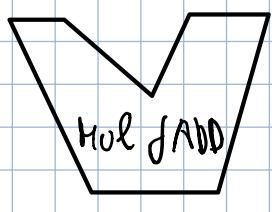
$+, -, \text{notazione, shift}, \cdot, /$



interv.

$+, -, \text{notazione, shift}$

$K \Delta t$ {costo
stabilizzazione
alu



Multiply
f ADD

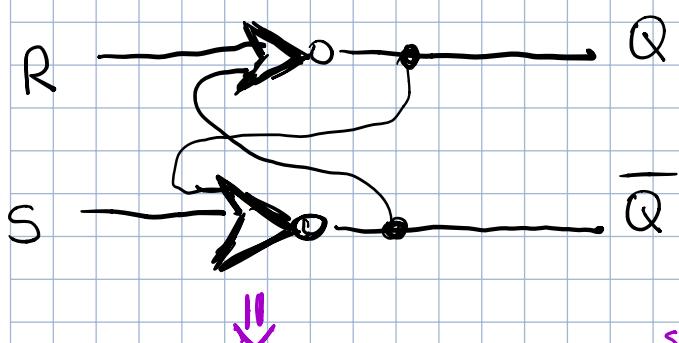
(Permette di fare la Σ)

SR LATCH

R = Reset

S = Set

↓
Set reset



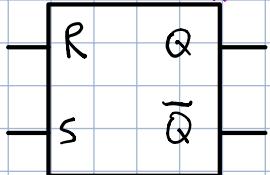
$$1) R=0, S=1 \Rightarrow \bar{Q}=0 \text{ e } Q=1$$

$$2) R=1, S=0 \Rightarrow \bar{Q}=1 \text{ e } Q=0$$

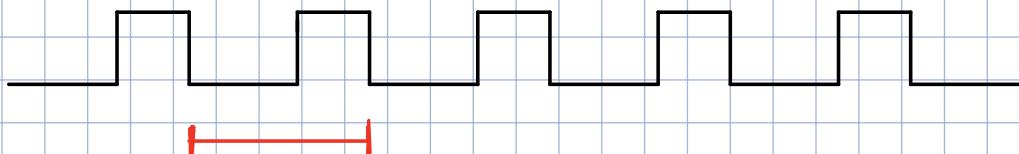
3) $R=0, S=0 \Rightarrow$ Preserva lo stato

SR CATCH

Porte NOR

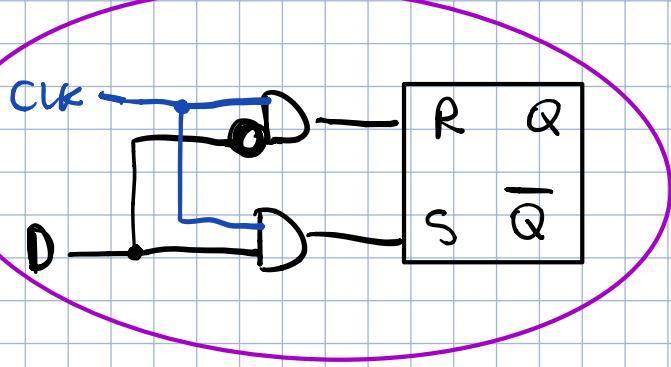


Aggiungo un clock in modo che lo stato delle nostre memoria cambia solo quando il clock è alto



$$\text{Lunghezza del clock} = T_0 \text{ (tow)}$$

Circuiti con l'aggiunta del clock:



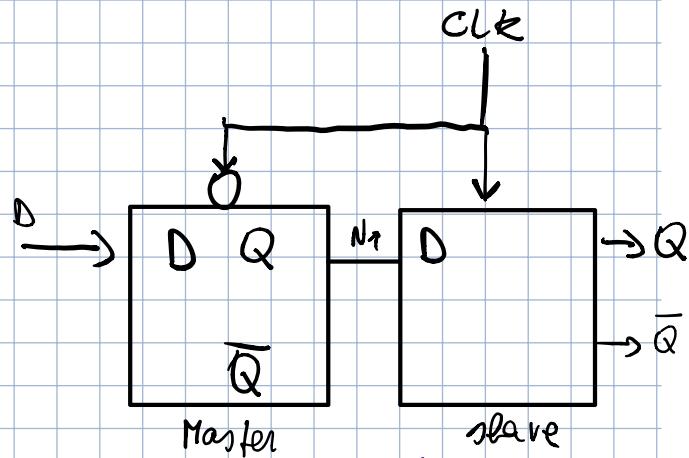
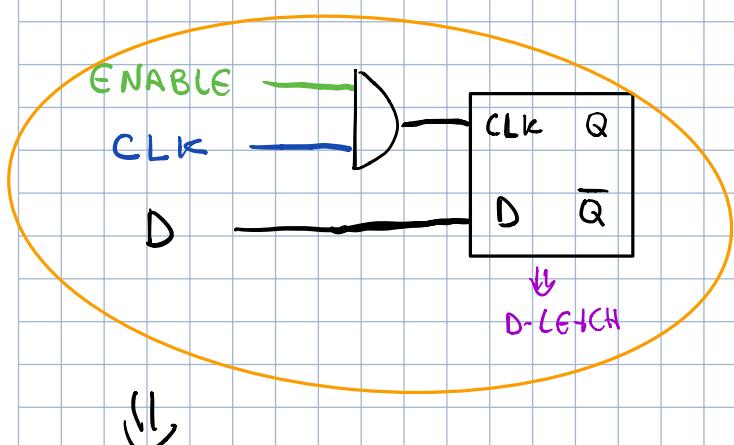
D-latch

$$\begin{cases} \text{CLK} = 1 \\ \text{D} = 0 \end{cases} \Rightarrow \begin{cases} \text{R} = 1 \\ \text{S} = 0 \end{cases}$$

$$\begin{cases} \text{CLK} = 1 \\ \text{D} = 1 \end{cases} \Rightarrow \begin{cases} \text{R} = 0 \\ \text{S} = 1 \end{cases}$$

Quando $\text{CLK}=0$ viene mantenuto lo stato precedente

Per decidere quando D deve scrivere, si usa il bit "ENABLE"



D-LATCH + ENABLE = D-FLIP-FLOP = REGISTRO 1 bit

per stabilizzarlo

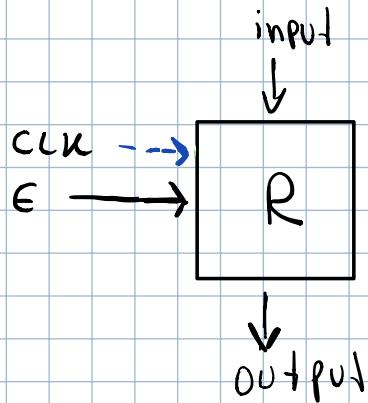
Si usano 2 D-LATCH

Quando $clock=0$, il D-latch mantiene le informazioni da D a Q, quando aggiorna N_1 . (il valore dell'uscita dello slave non viene mod.)

Quando il $clock=1$ viene bloccato il passaggio da D a Q nel master e viene effettuato il passaggio da D a Q nello slave quindi avremo in uscita Q il valore che aveva acquisito N_1 .

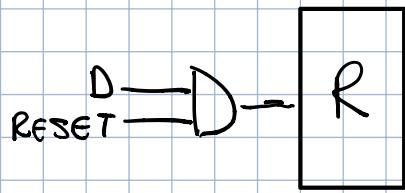
Solo se il bit enable = 1 perche' quando Enable = 0 viene ignorato il clock

Registro



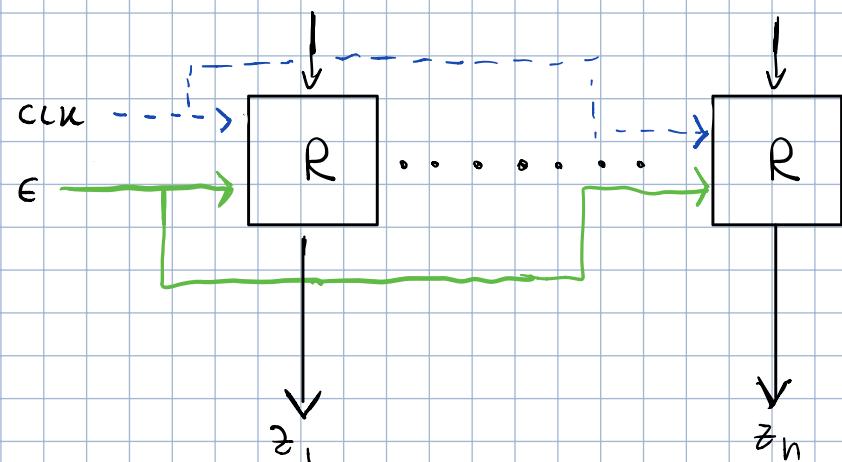
Si assume che il valore viene scritto quando il clk si alza e quando il clk si abbassa, l'uscita è instabile

Registro resettabile



Quando Reset = 0 viene resettato il bit restituendo il valore di 0

Registro n bit (più FLIP-FLOP in parallelo)



Reti sequenziali \Rightarrow Possibilità di memorizzazione

\downarrow
Un solo stato interno

X = 0

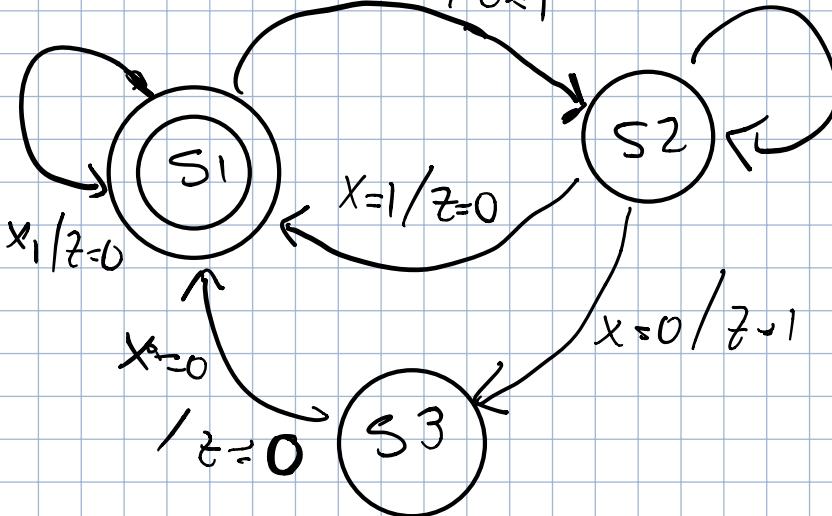
Y = 1

Implementazione Mealy

$x=0/z=1$

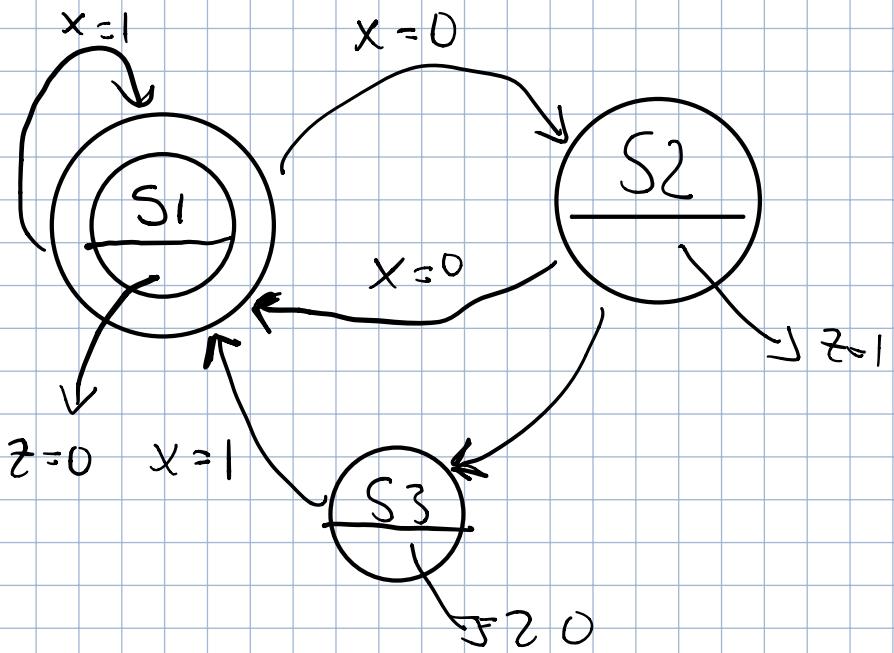
$x=0/z=0$

1) Automa di mealy:



Ad ogni arco associamo un valore di ingresso e uno di uscita

2) Automa di moore:



Ogni stato ha il valore di uscita predefinito a prescindere da quale sia l'ingresso

OSS: Automa di moore + lento e + stat. di quelli di mealy

Da automa a Rete sequenziale

Una rete sequenziale è composta da :

- 1 input → calcola lo stato (σ)
- 2 reti combinatorie → calcola l'uscita (w)
- 1 registro
- 1 output

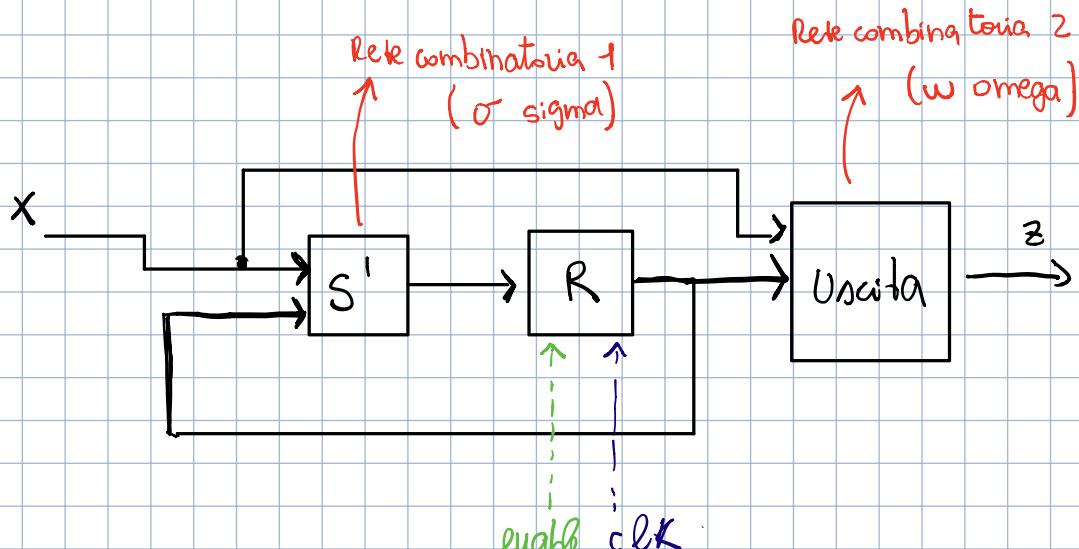
Attenzione: le reti con percorsi ciclici sono sequenziali. Per migliorare questo tipo di reti viene messo un registro fra un ciclo e l'altro.

Rete sequenziale sincrona: la temporizzazione del sistema è limitata dal segnale del clock dei registri.

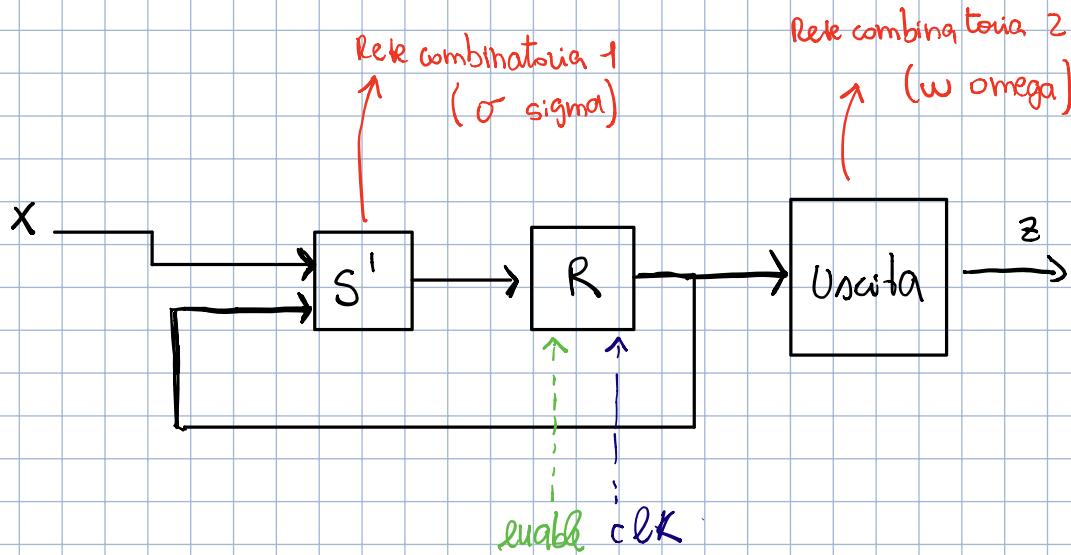
Macchine a stati finiti (FSM)

Sono composte da due blocchi di logica combinatoria, la logica di stato prossimo e la logica di stato di uscita ed un registro. Ne esistono 2 tipi:

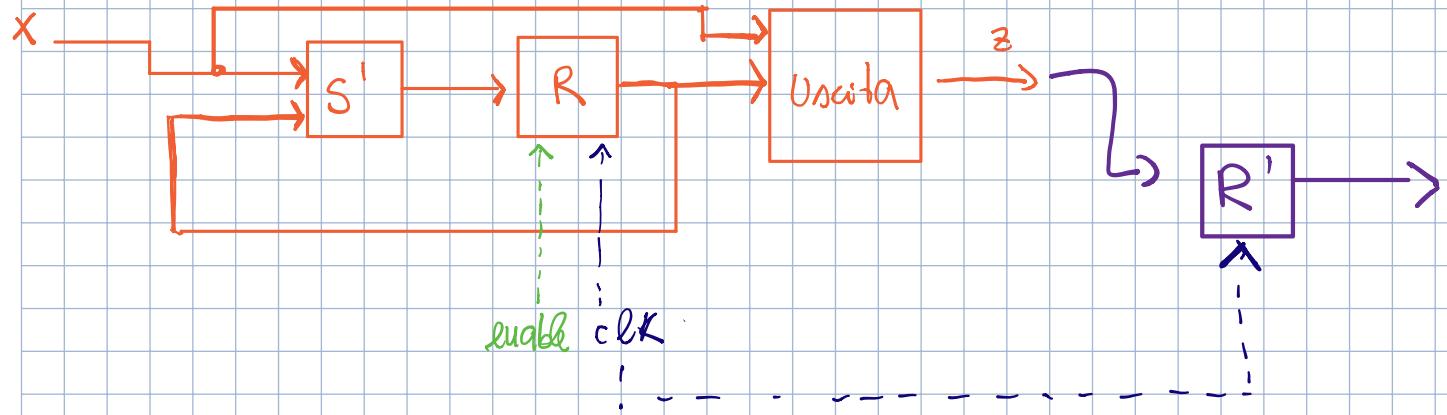
1) Automi di Mealy (l'uscita dipende sia dallo stato che dal val. di ingresso)



2) Automa di moore (l'uscita dipende unicamente dallo stato)

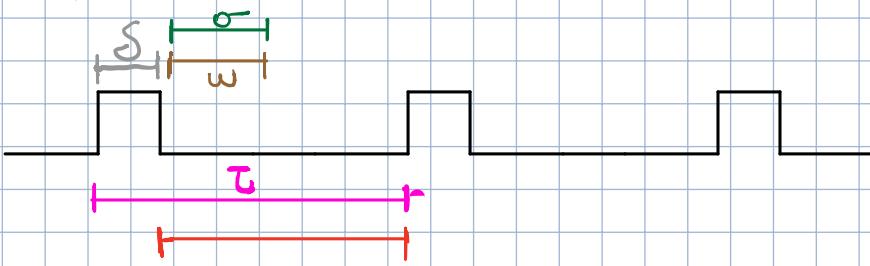


Passare da un automa di Moore a uno di Mealy



Mealy + R' = Moore (Mealy ritardato)

Osservazione:



Il tempo di stabilizzazione
non deve essere più piccolo

del $\max \{ \Delta t_\sigma, \Delta t_\omega \}$



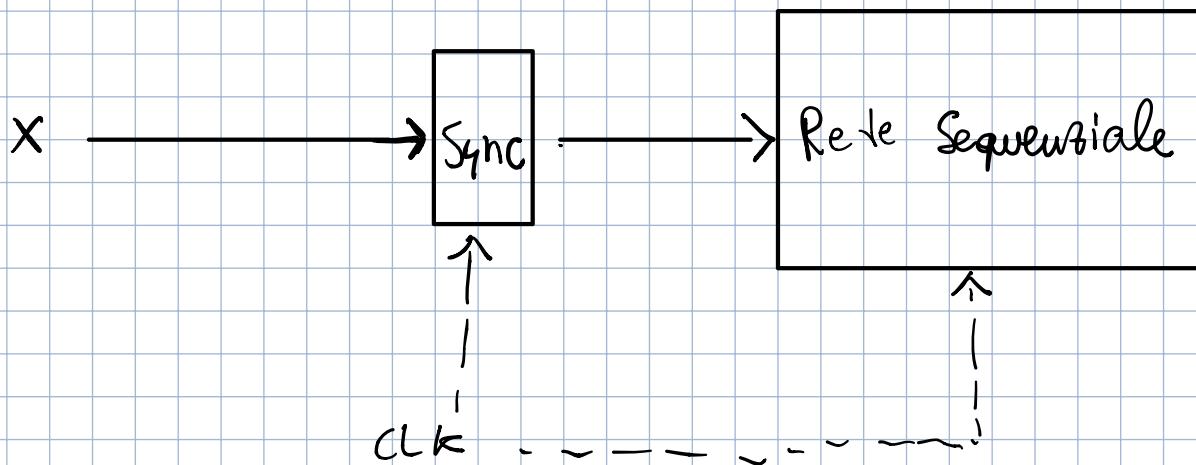
$$T = \delta + \max \{ \Delta t_o, \Delta t_w \}$$

Osservazione:

Se il valore delle x viene cambiato durante la stabilizzazione di o e w , o e w ci mettono di più a stabilizzarsi, quindi il ciclo di clock che ho dato potrebbe non bastare.

Per risolvere questo problema utilizzo un Sincronizzatore

Sincronizzatore - Registro con stesso clk delle reti sequenziali e bit enable sempre a "1"



Questo metodo risolve il problema perché invia l'input solo quando il ciclo di clock va alto, quindi la rete seq. si è stabilita.

2° osservazione: C'è un problema anche quando l'input x cambia durante il ciclo alto del clock, ma questa situazione non mi può risolvere.

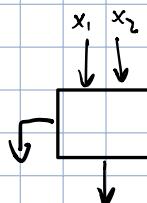
Progettazione FSM:

- 1) Capire i ingressi e uscite della FSM
- 2) Disegnare l'automa
- 3) Tabella degli stati \rightarrow si ricava il min termine
- 4) Tabella delle uscite \rightarrow Si ricava il min termine

Inverso del progettazione

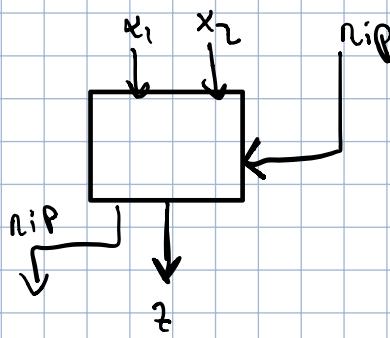
- 1) Dal circuito si ricava il min termine dello rett comb per lo stato prossimo e dell'uscita mettendo negate le entrate negate e non negate le entrate non negate
 - 2) Si ricava la tab. degli stati e delle uscite e si riduce il più possibile
 - 3) si disegna l'automa
-

Half Adder = Fa una singola somma senza riporto e produce un risultato e un riporto



x_1	x_2	z	r_p
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

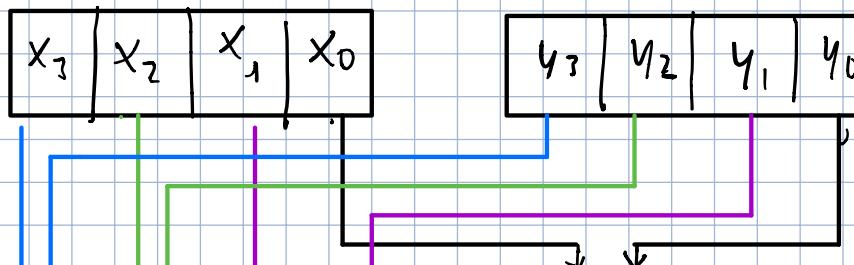
Full Adder = Fa le somme con il riporto

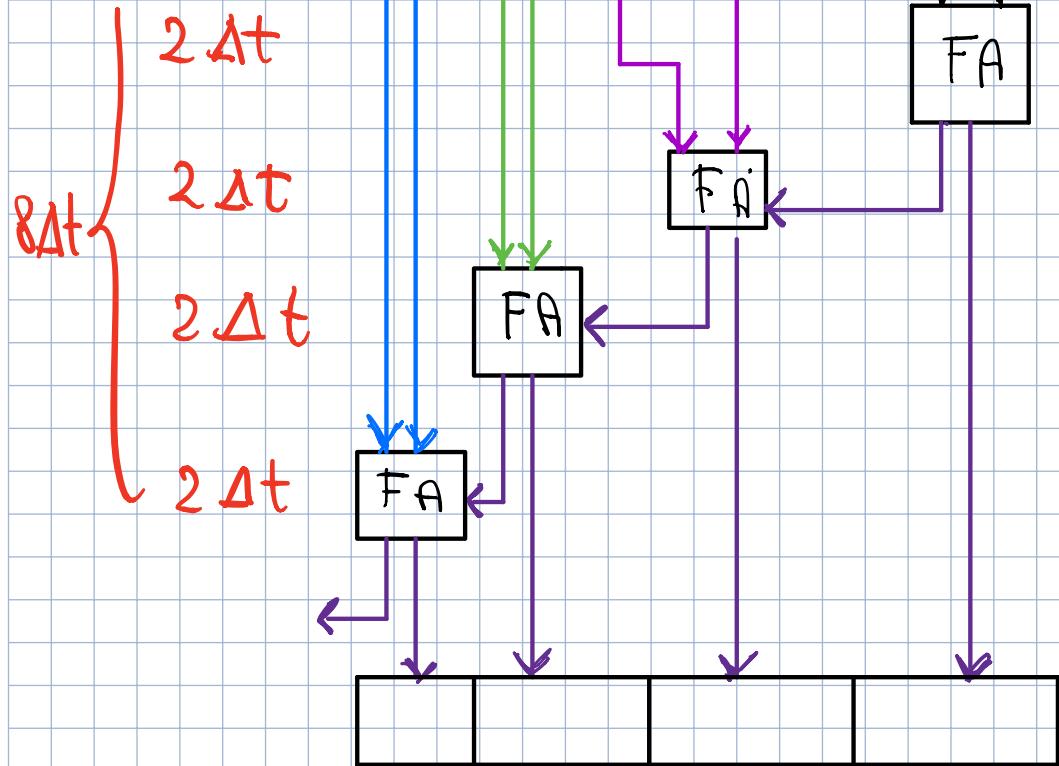


x_1	x_2	r_i	z
0	0	1	1
0	1	0	1
1	0	0	1
1	1	1	1

x_1	x_2	r_i	r_o
0	1	1	1
1	0	1	1
1	1	0	1
1	1	1	1

- 1) **Additionati ripetuti** (ottimo dal punto di vista della negligenza e modularità visto che si ripete il modulo del full adder più volte. Meno ottimo per le temporistiche)





2) Tabella di verità (ottimo per il tempo ma bisogna costruire un circuito ad-hoc)

$x_3 \dots x_0$	$y_3 \dots y_0$	$z_3 z_2 z_1 z_0 \pi$
2^{56}	2^8	

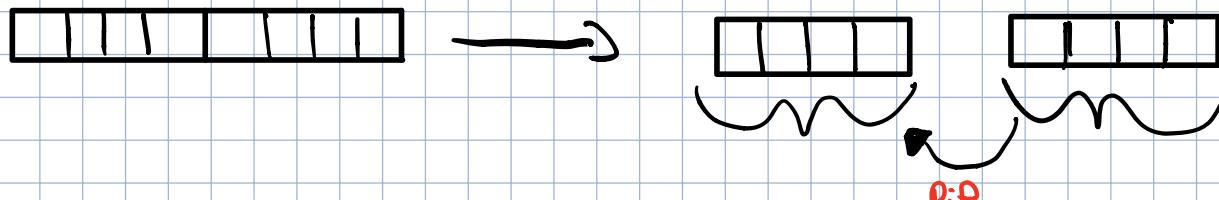
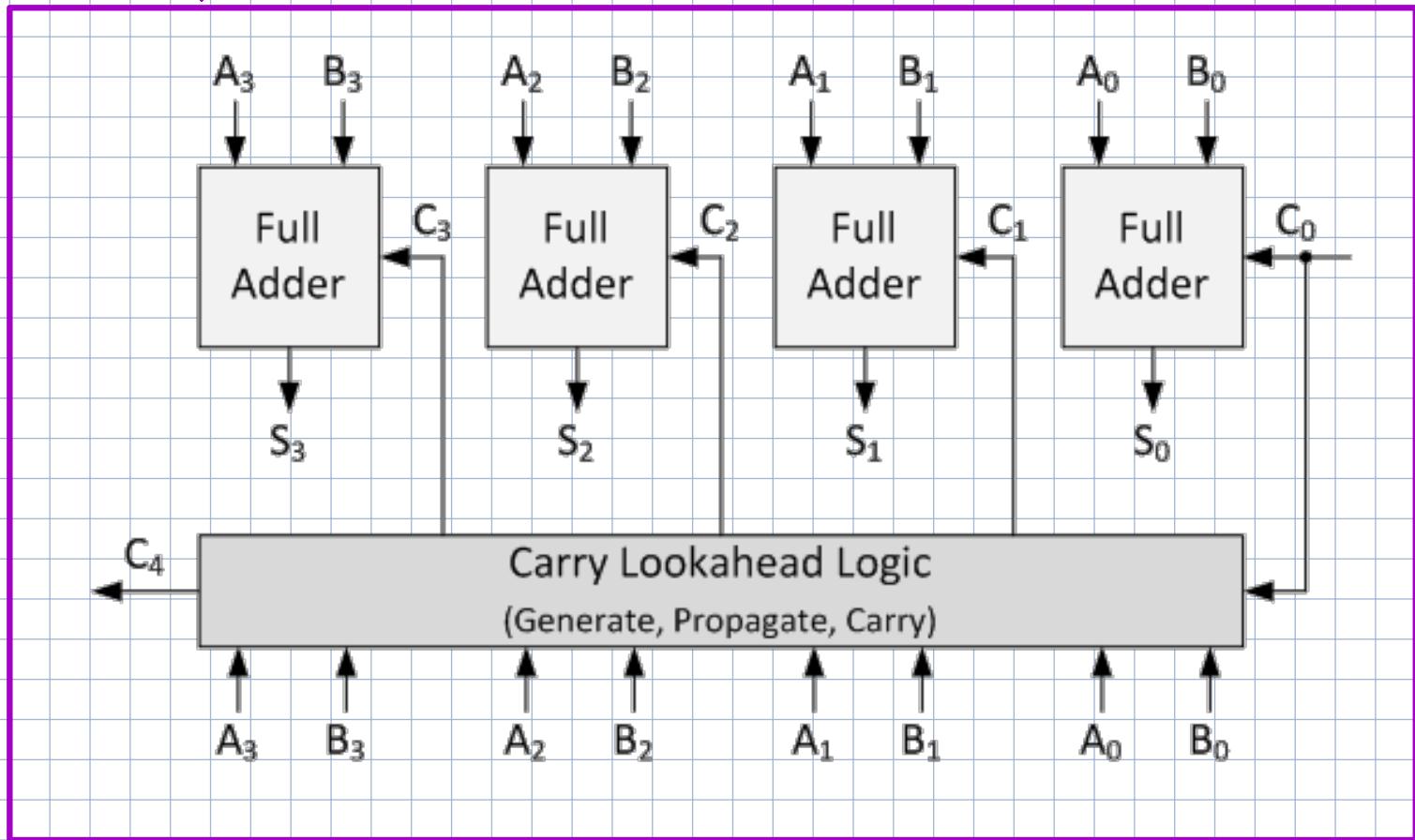
1 livello AND perché ho 8 ingressi

3 livelli OR perché ho al max 255 "1"

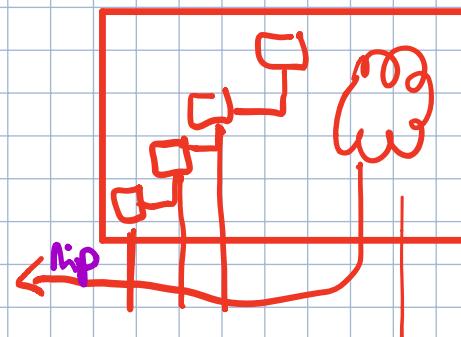
\downarrow
4Δt

Generazione / Propagazione del riporto (Anticipazione di riporto)

(carry lookahead logic)



Faccio la somma dei 4 bit meno significativi e passo il riporto ai bit più significativi



Il riporto viene dato a parte

n_{i-1} } Propagazione di un riporto } P_i
 $x_i \text{ OR } y_i$

$x_i \rightarrow$ genera un riporto \rightarrow

Propago un riporto
 → da una somma precedente
 se e solo se uno dei
 due bit è 1

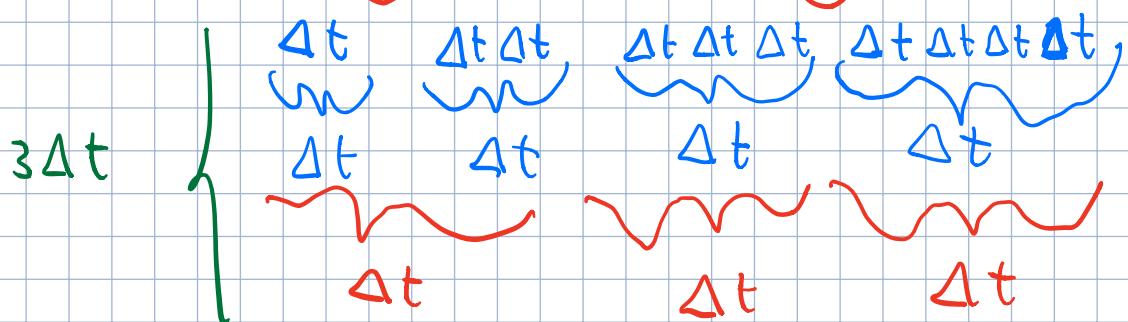
$$y_i \quad \left\{ \begin{array}{l} \\ \end{array} \right. \quad x_i \text{ AND } y_i \quad \left\{ \begin{array}{l} G_i \rightarrow \text{Genera sicuramente un} \\ \text{riporto se e solo se tutti} \\ \text{e due bit sono a 1} \end{array} \right.$$

$$G = G_3 + P_3(G_2 + P_2(G_1 + P_1 G_0))$$

Genera un riporto su 4 bit

$$G = G_3 + P_3(G_2 + P_2 G_1 + P_2 P_1 G_0)$$

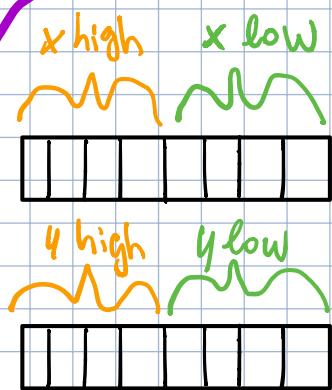
$$= G_3 + P_3 G_2 + P_3 P_2 G_1 + P_3 P_2 P_1 G_0$$



$$P = P_3 P_2 P_1 P_0$$

viene già calcolato in G
quindi non ha costo
aggiuntivo

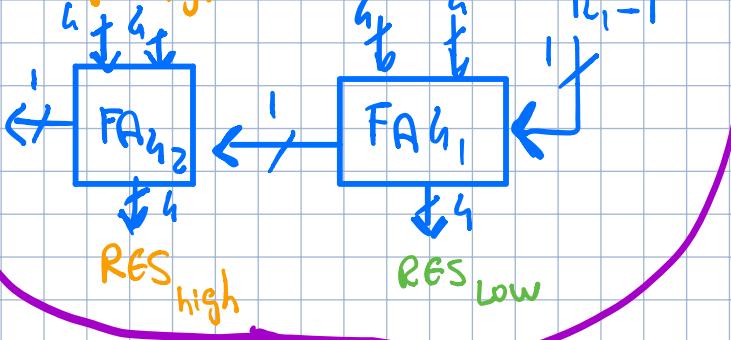
$$RIP = G + (P \cdot \text{riporti}) \quad ? \quad 4 \Delta t$$



$$\max \left\{ \Delta t_G, \Delta t_{(P \cdot \text{riporti})} \right\} + 1$$

Se voglio fare la somma
di 2 numeri da 8 bit

xhigh yhigh xlow ylow

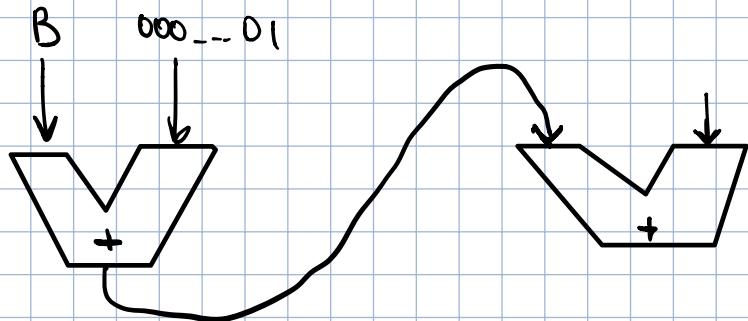


posso usare 2 adder da 4 bit
con Gen/prop riporta
dividendo i bit in 2
parti: e passa solo il carry
da un adder all'altro

Sottrazione

$$A - B = A + (-1) \cdot (B)$$

$$A + (\bar{B} + 1) \rightarrow \text{Sottrazione}$$



Moltiplicatore

4 bit \times 4 bit

Esempio su 2 bit

$$\begin{array}{r}
 x \rightarrow 10 \\
 y \rightarrow 11 = \\
 \hline
 10 + \\
 \hline
 10' = \\
 \hline
 110
 \end{array}
 \rightarrow
 \begin{array}{l}
 \text{AND}(y[0], x[1]) \quad \text{AND}(y[0], x[0]) \\
 \text{AND}(y[1], x[1]) \quad \text{AND}(y[1], x[0])
 \end{array}$$

AND(y[0], x[1]) AND(y[0], x[0]) /
 AND(y[1], x[1]) AND(y[1], x[0])

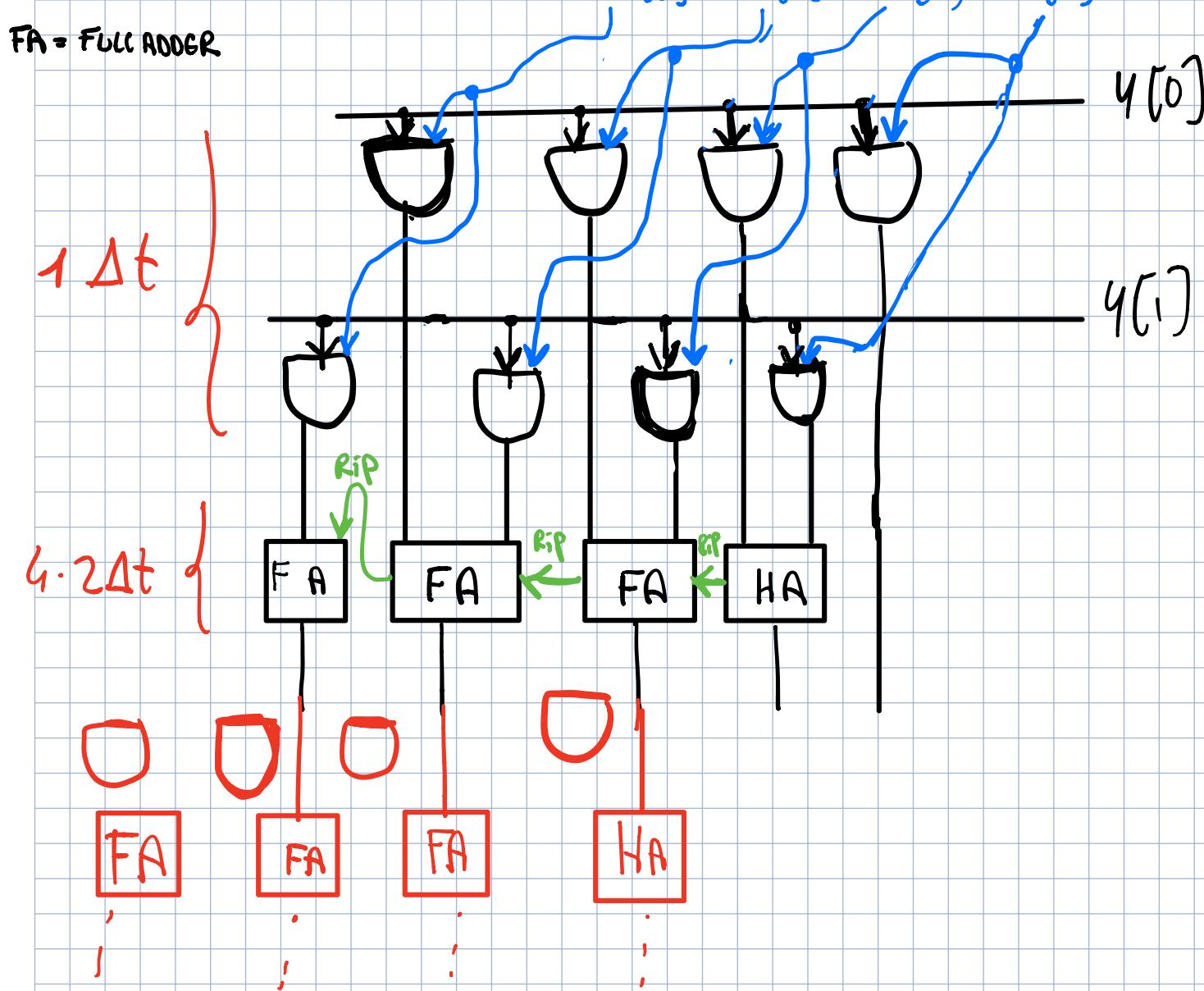
AND(y[0], x[0])
 AND(y[1], x[1])
 AND(y[0], x[1]) + AND(y[1], x[0]) + AND(y[0], x[0]) + AND(y[1], x[1])
 AND(y[1], x[1]) + AND(y[0], x[1]) + AND(y[1], x[0]) + AND(y[0], x[0])

Moltiplicatore con componenti

HA = HALF ADDER

$x[3] \quad x[2] \quad x[1] \quad x[0]$

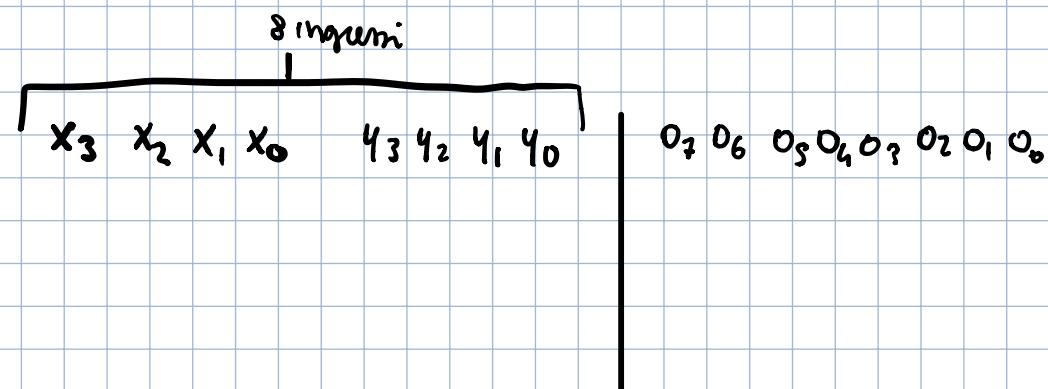
FA = FULL ADDER



In generale avrò gli and sempre spostati di uno verso sinistra per ogni lvl.

In fine gli adder a destra e a sinistra sono degli half adder e quelli in mezzo sono dei full adder.

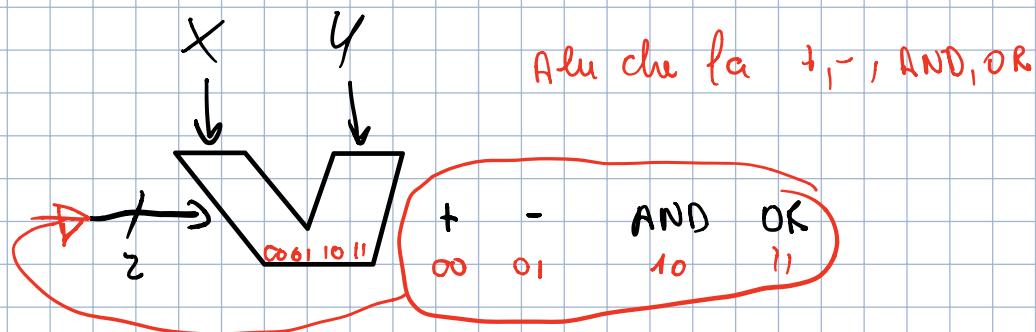
Moltiplicatore con tabella di verità



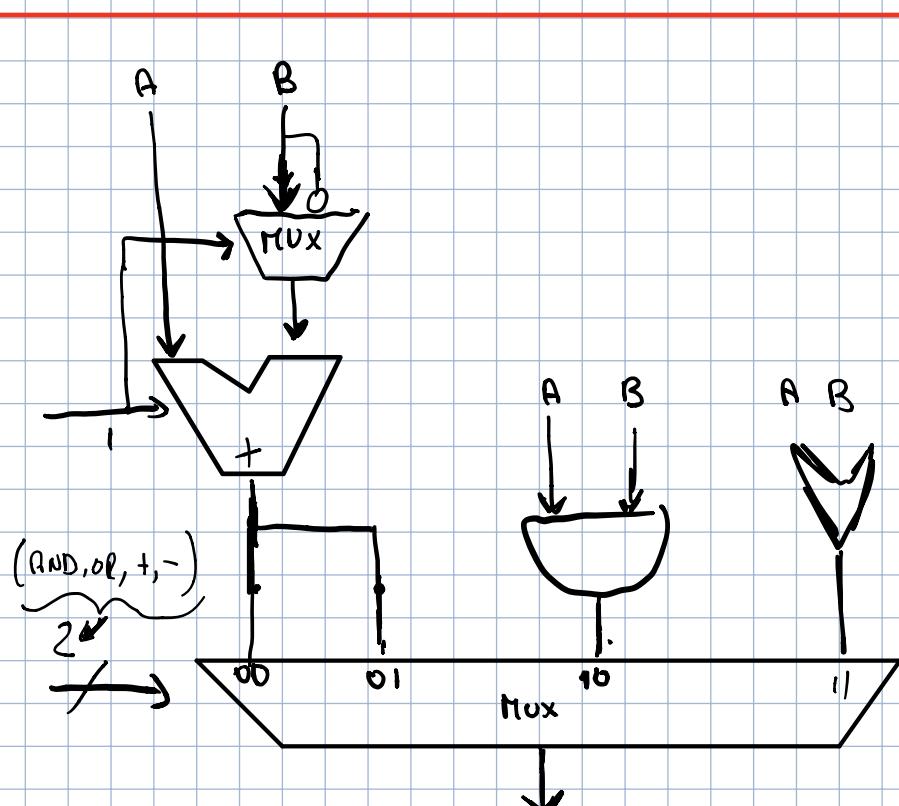
23

$$3 \text{ Pork OR } + 1 \text{ AND } = 4 \Delta t$$

Alu

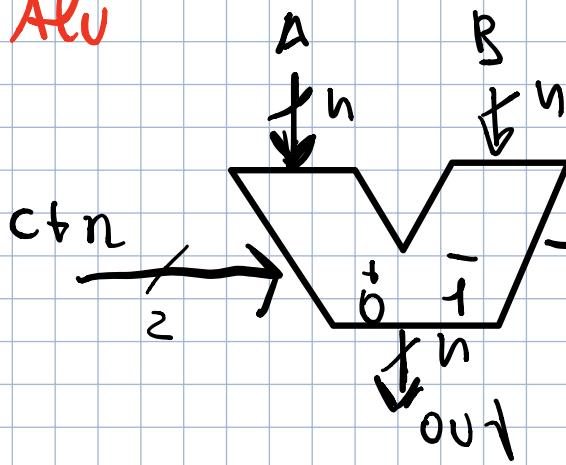


Realizzazione di un ALU



Per scegliere
quale operazione
fare devo stampante
tutto sul nascitudo
e poi scegliere
quelle da mi
intervenire

Alu



1 bit

$Z =$ Se il risultato è 0

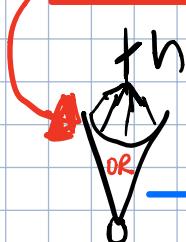
$N =$ Se il risultato è positivo o negativo

$C =$ se è stato generato un riporto

$V =$ se ho avuto problema
di overflow

tutte le uscite a 0

$$Z : \boxed{V_{OUT}(i) = = 0}$$



1 solo
di tutti 0

ultimo non
rappresentabile
su i bit
che ho
in uscita

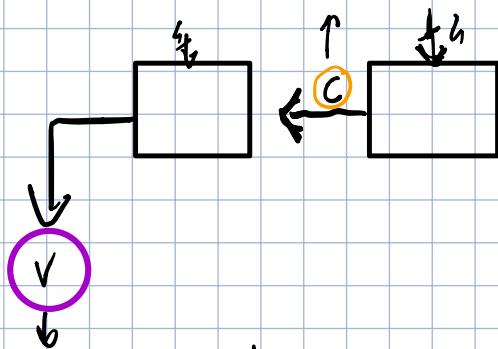
l'ultima
somma di
bit ha
generato un
riporto

$$N : \boxed{OUT[n-1]}$$

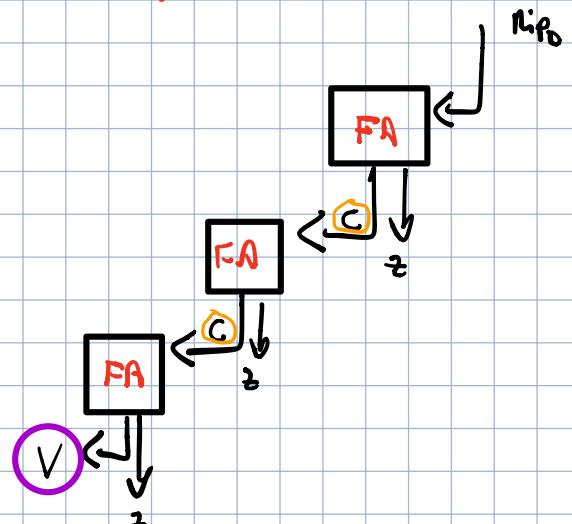
→ Bit più significativo

$V :$ Moduli carry - lookahead

se riporto = 1



Addizioniatori in cascata

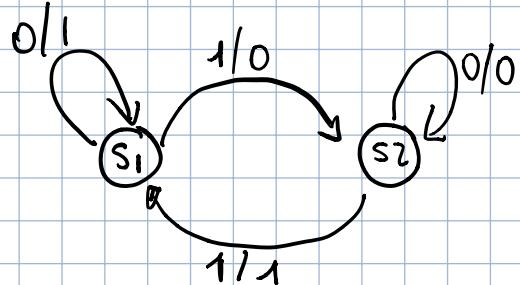


modo comi-clock-ahead vuol dire vuole al bit successivo

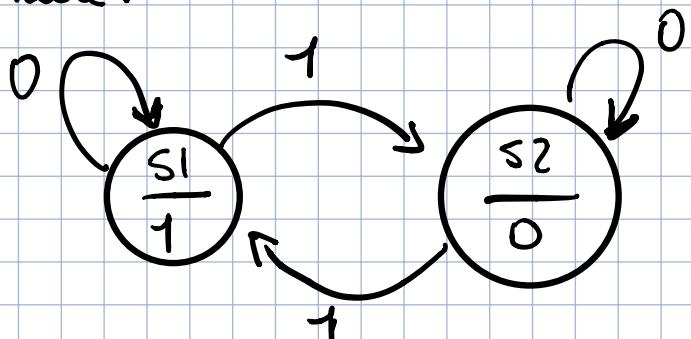
Esempio di automa Mealy / Moore con stesse quantità di stati

Punti

Mealy:



Moore:



Sigma

Mealy

Omega

$s \times$	s'
00	0
01	1
10	1
11	0

$\rightarrow R \rightarrow$

$s' \times$	z
00	1
01	0
10	0
11	1

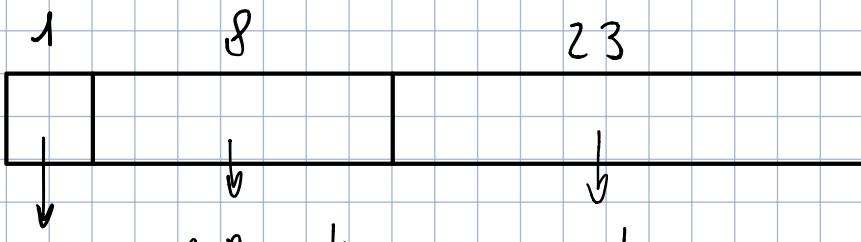
Moore

$s \times$	s'
00	0
01	1
10	1
11	0

s'	z
0	1
1	0

Numeri in virgola mobile

Singola precisione

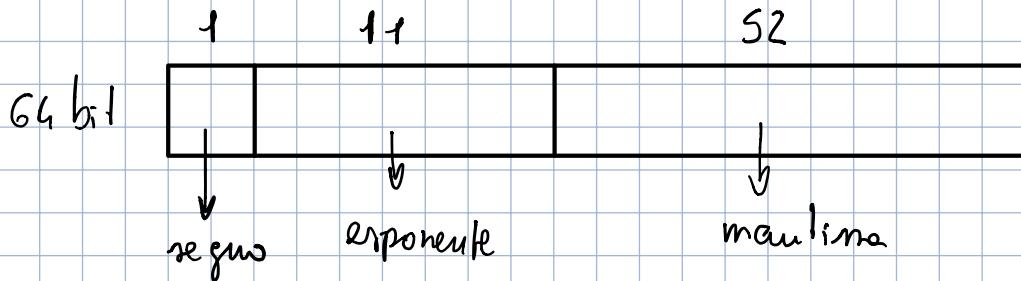


segno
 ↓
 esponente
 ↓
 (di quanto
 spostare la
 virgola)

mantissa
 (valore)

Per dire se l'esponente è positivo o negativo ci mettiamo 127
 quindi 0 diventa -127, 127 diventa 0 e 255 diventa 128

Doppia precisione

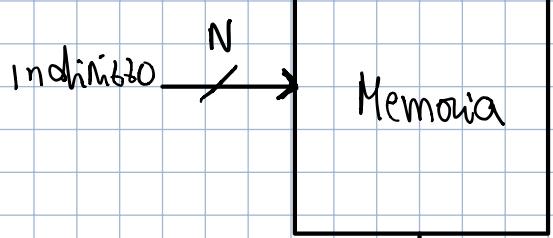


Quando faccio la somma di numeri con la virgola devo:

- 1) Allineare le mantisse \rightarrow Quando ho gli esponenti devo spostare in modo che siano incollabili nel modo giusto (traslone i bit in modo che gli esponenti coincidano)
- 2) Somma
- 3) Se la somma non è aggiustata come ho un esponente/mantissa devo ristenderla (Normalizzazione)

Memoria \rightarrow Gruppo di più registri

2^n righe (indirizzi (**Numero parole**))
 m colonne (bit di ogni parola)

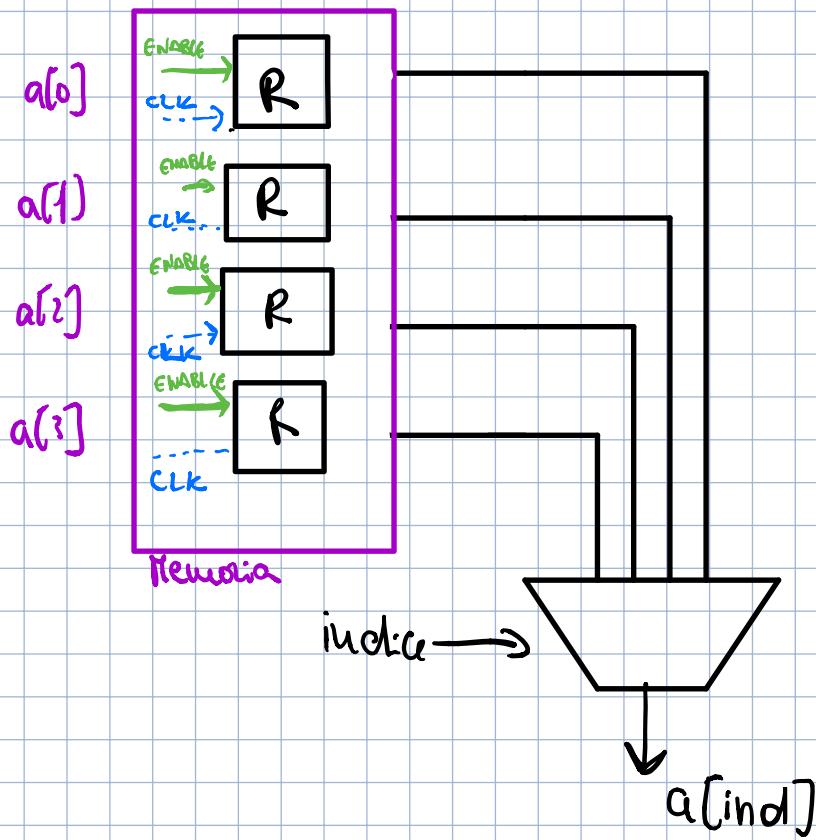


Una memoria è grande:

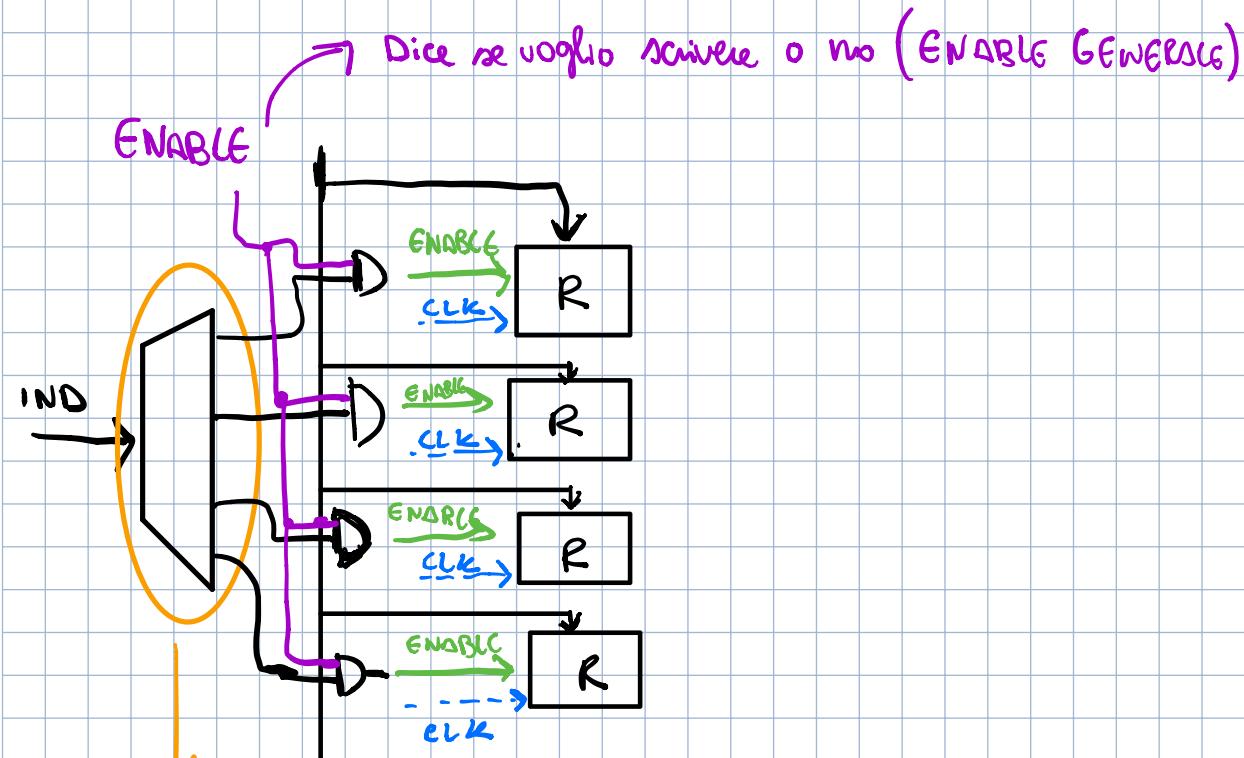
$n \times m \rightarrow n$ parole x m bit per ogni parola

$\downarrow M$
Data

Per leggere la memoria:



Scrivere nella memoria



DECODER

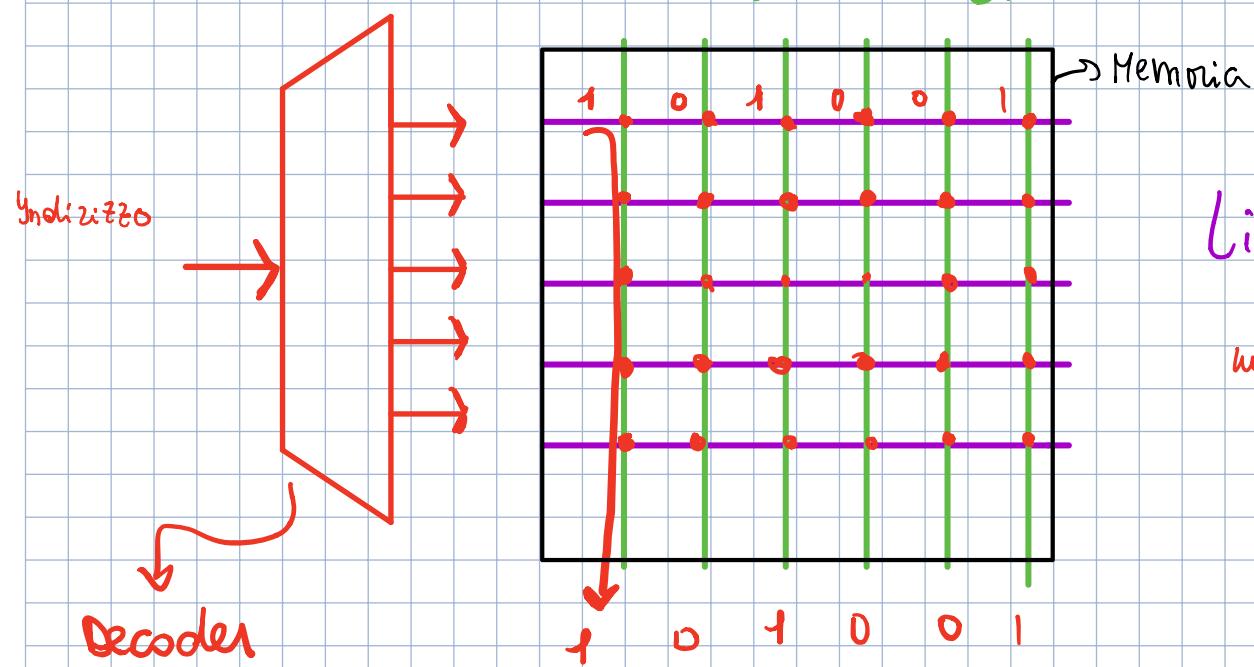
(DA UN 1 ALL'ENABLE DEL REGISTRO DOVE DEVO SCRIVERE)

↓
SPECIFICO
DEL REGISTRO

IMPLEMENTAZIONE VERA DELLA MEMORIA

→ Sempre organizzate
a matrice

Linee di bit - bit per ogni parola



Linee di Parole

||

numero di parole
2^{*} # (bit indirizzo)

Nelle DRAM:

All'incrocio delle linee ho un condensatore. (Pallini rossi)
Se quando mando il segnale il condensatore era carico, mi scava segnando la linea di bit.

Quindi in fondo alle linee di bit vedo il valore che contiene il bit. (lettura)

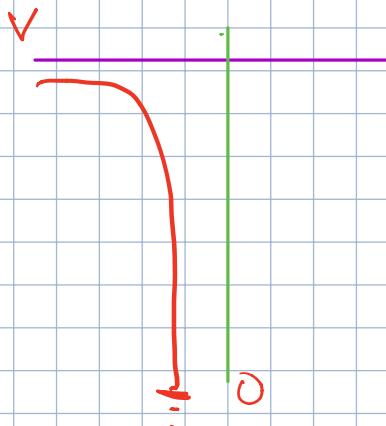
Ogni volta che effettuo una lettura la memoria deve essere ricarica perché perdo

Cioé che c'è scritto sopra.

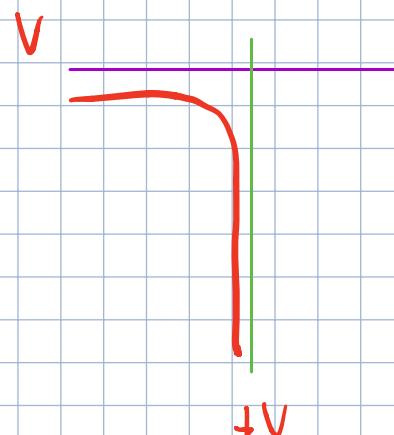
E' modo che il condensatore si scarica gradualmente, ovvero comunque effettuare un refresh per ricaricare i contenuti.

Se invece voglio scrivere, manda un po' più di corrente sulle linee di bit e all'incrocio il condensatore si carica.

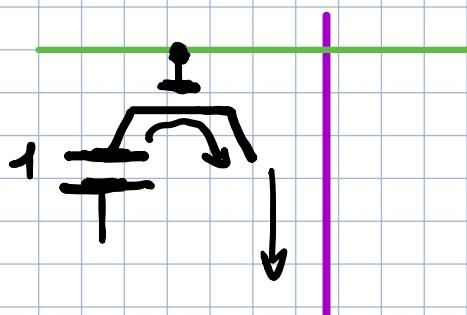
ps: ho un transistor che fa da interruttore tra la linea di bit e il condensatore.



Se il bit è a 0, la tensione
va a manzo



Se il bit è a "1"
scorre lungo la linea
di bit



transistor che fa da
interruttore

tempo di acceso alla memoria:

Dipende da quanto riesco a
fare velocemente la trasform.

Quindi se ho più linee di
parole il decodificatore sa



dall' indirizzo alle sue decodifiche

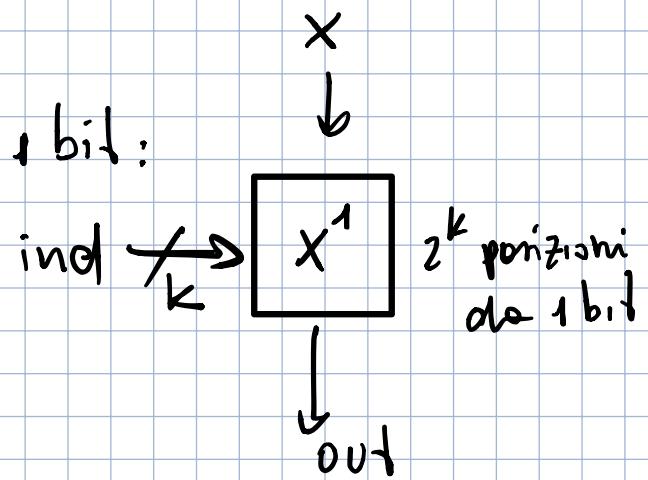
+ grano, quindi curto più
livelli logici, quindi il tempo
di accesso maggiore

Esempio:

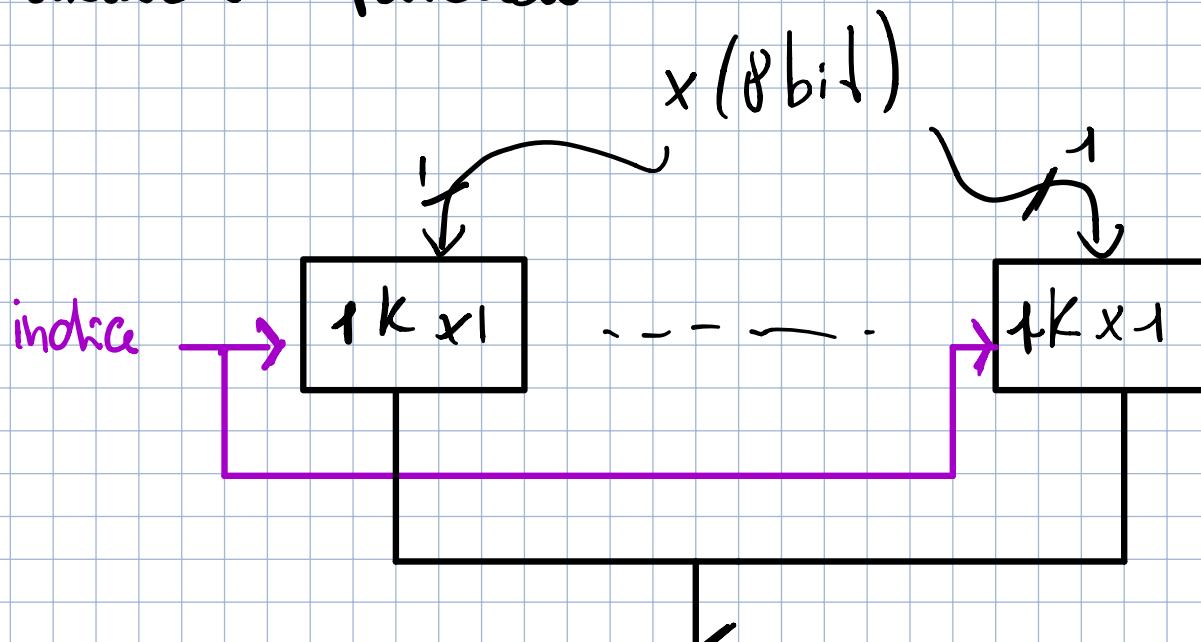
memoria $1K \times 1$ bit:
1 K righe orizzontali
1 sola riga verticale

memoria $1K \times 8$ bit:
1 K righe orizzontali
8 righe verticali

Se considero la memoria $1K \times 1$ bit:



Potrei prenderne più da $1K \times 1$ bit e formare la $1K \times 8$ bit
mettendole in parallelo



1
8 bit

RAM (RANDOM ACCESS MEMORY)

SCRIVIBICE
LEGGIBILE

- DRAM (Dynamic Random Access memory) → Usate per le RAM

- Molto economiche

- 1 transistor × 1 bit

- lenta

Condensatore tende a scaricarsi, quindi ogni tot devo andare a riscrivere

Usano un condensatore per memorizzare i valori

a riservare

Evolution:

SDRAM : Ciclo di clock che permette di rendere l'accesso ai dati pipeline

DDR SDRAM : Accesso ai dati ma ai punti di salita che di discesa di un clock

- SRAM (Static Random Access memory)

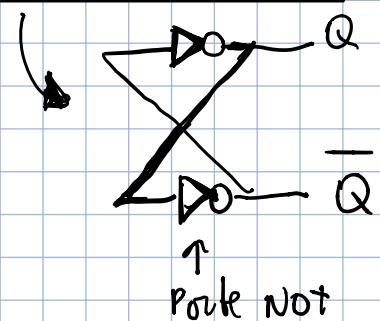
- Non c'è bisogno di rinfrescare

- 6 transistor × 1 bit

Usate per le memorie vicine al processore

- Costose
- Veloci

Coppia di Negatori collegati a croce



• FLIP - FLoP \rightarrow Usate per le CPU

- ~ 20 transistor \times 1 bit
- Veloci assimi
- Costano tantissimo

T_A

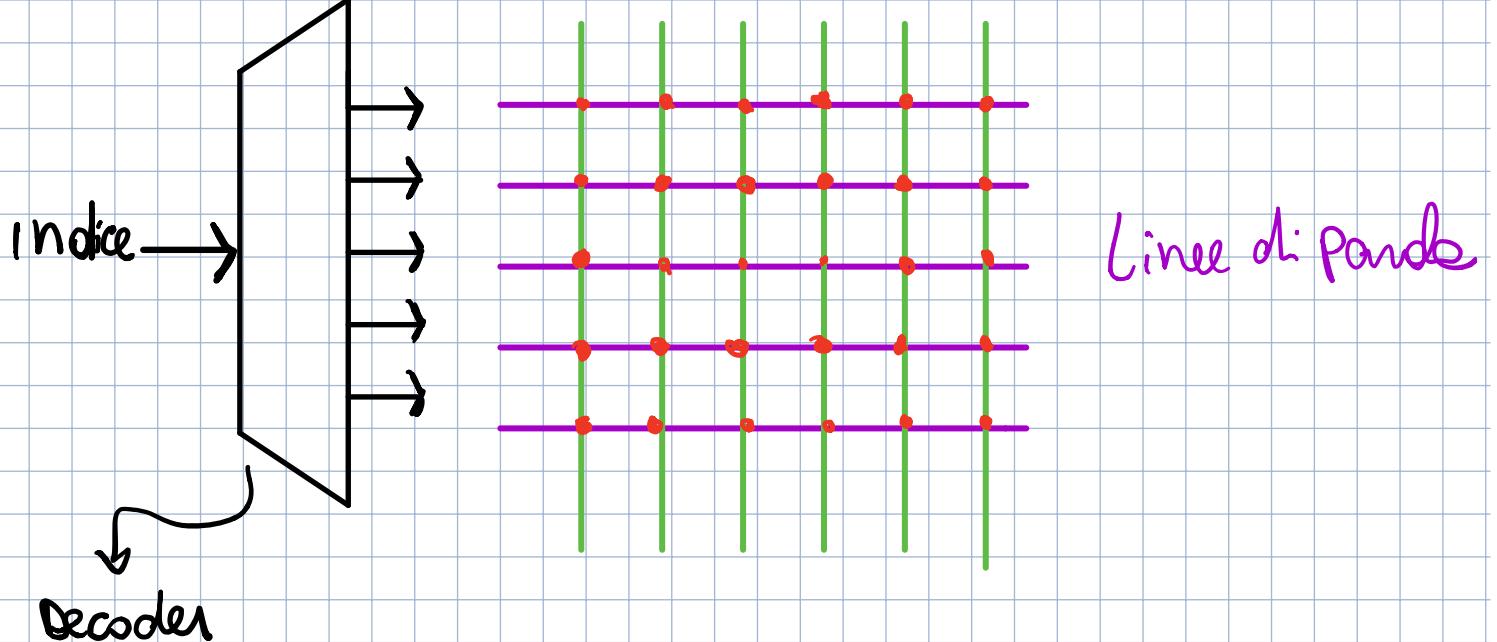
$\left. \begin{array}{l} < \text{ nano secondi} \Rightarrow \text{ registri (FLIP FLOP)} \\ \approx \text{ nano secondi} \Rightarrow \text{ memoriale (CACHE)} \\ \approx \mu \text{ secondo} \Rightarrow \text{ memoria dinamica (DRAM)} \end{array} \right\}$

tempo di
accesso

ROM (Read only memory)

1) ROM (Non Reiscrivibile)

:
linee di bit

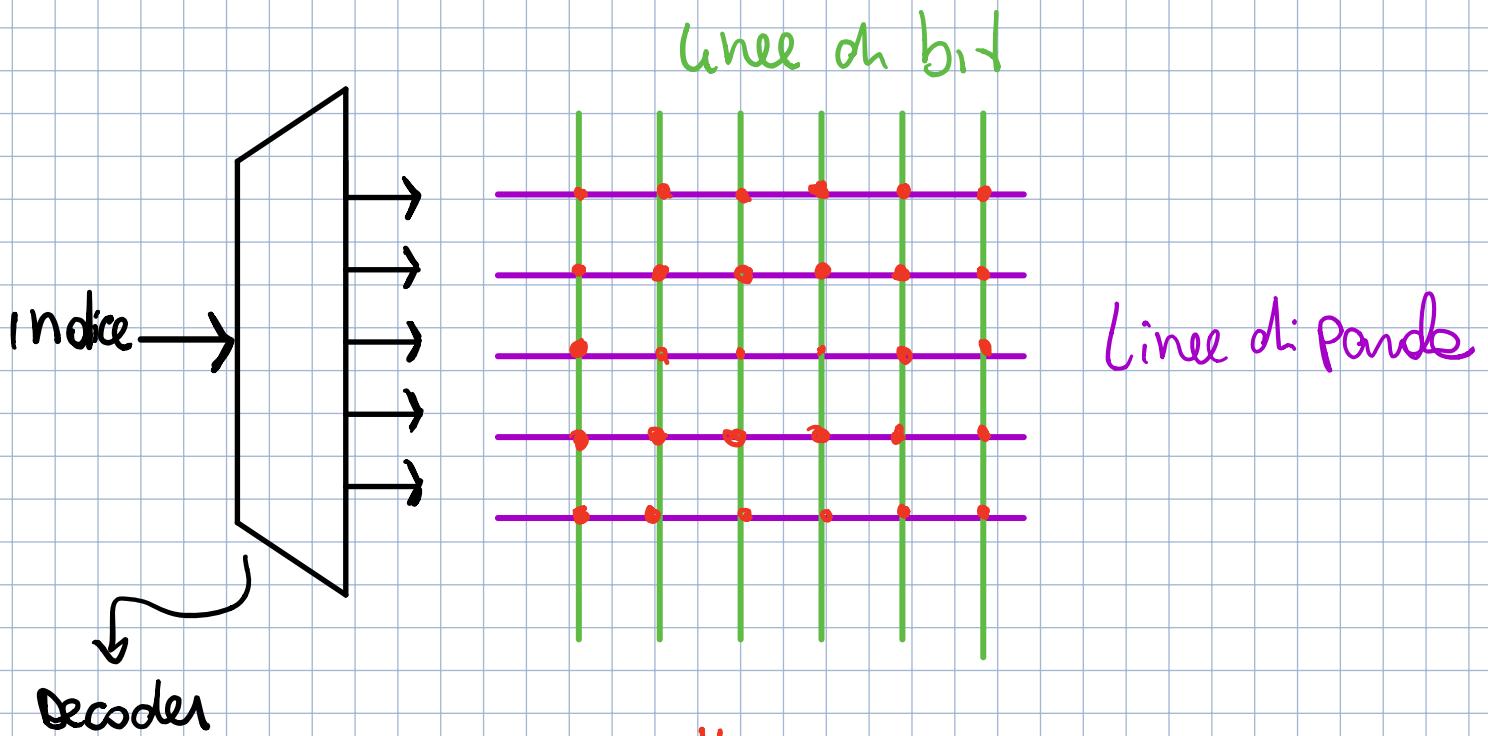


y contatti si toccano

Una ROM memorizza un bit come presenza o assenza di un transistore.

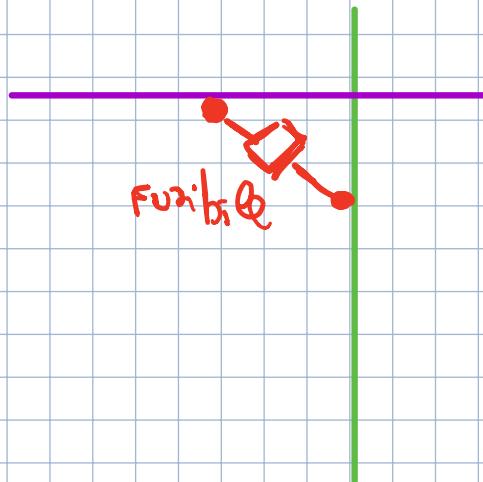
Le celle di una ROM sono reti combinatorie

2) PRON (Programmable ROM)



Nei contatti ho un fusibile

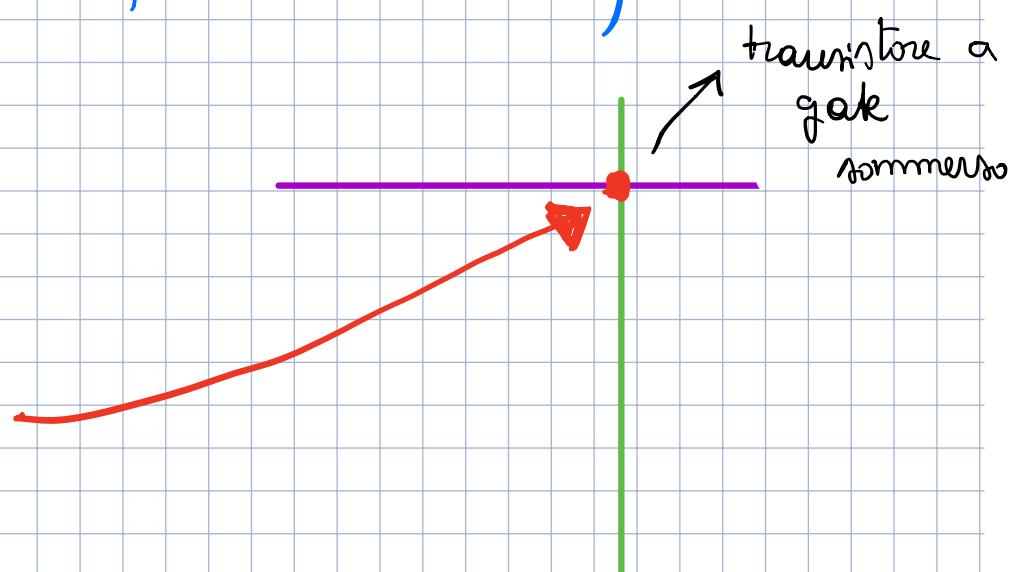
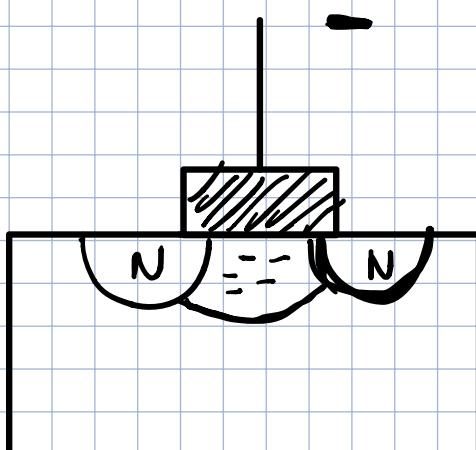
Tutti "1" quando esce dalla fabbrica.



Bruco i fusibili dove voglio "0"

↳ Programmabile una sola volta

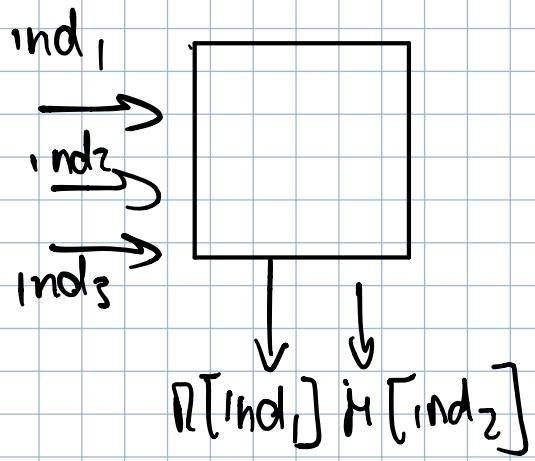
3) EEPROM (Electrically Erasable ROM)



Applico correnti forti per modificare l'intensità
(Posso modificare + volte)

Le ROM del PC sono EEPROM, infatti quando facciamo l'aggiornamento del bios cambiamo gli intensità dei vari transistors

Memoria Multiplo



Potremo accedere a più indirizzi allo stesso momento.

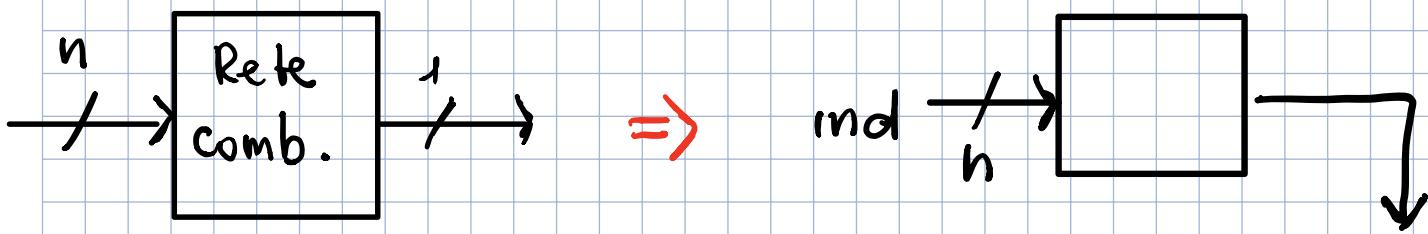
uso ind_1 e ind_2 per leggere
uso ind_3 per la scrittura
Non uscirà? : indirizzi non potranno farci i comandi che riguardano ad esempio 2 letture e 1 scrittura

Più decoder che prendono gli indirizzi:

LUT (Look Up Table) → Usata nelle FPGA

Potrei implementare una rete combinatoria con una memoria.

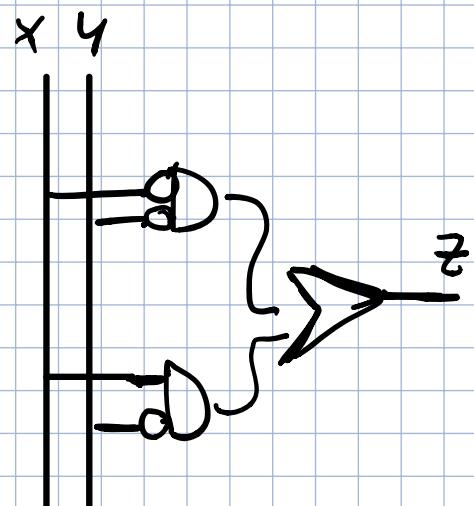
Una memoria 2^h parole per m bit può svolgere una qualsiasi rete combinatoria di N ingressi e m uscite



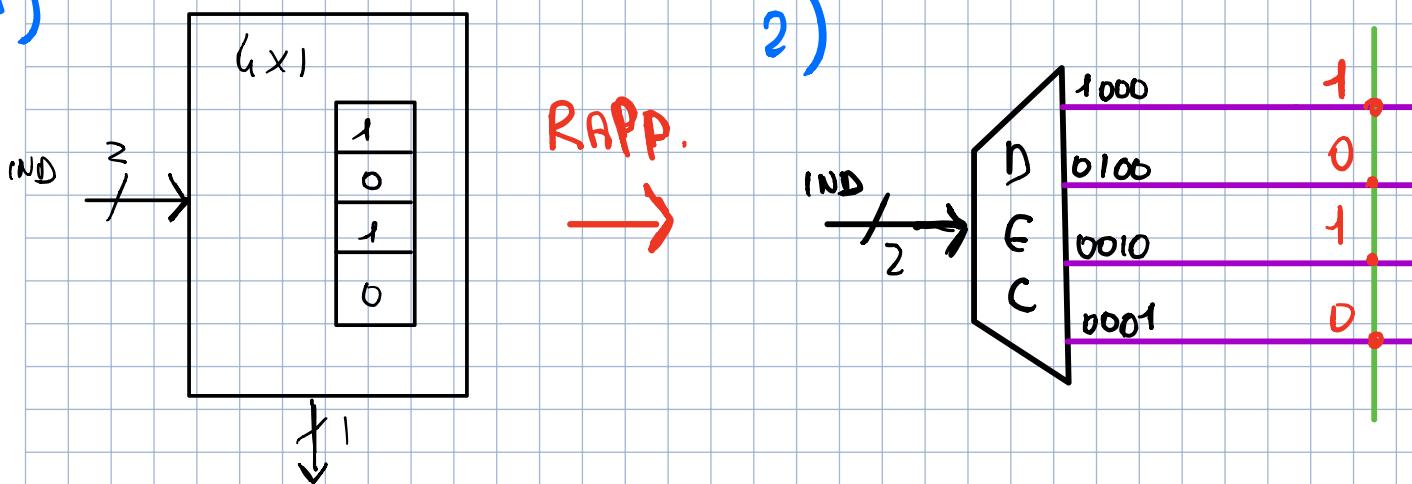
IMPLEMENTAZIONE STANDARD

x	y	z
0	0	1
0	1	0
1	0	1
1	1	0

$$\Rightarrow z = \bar{x}\bar{y} + x\bar{y}$$



IMPLEMENTAZIONE CON MEMORIA (EEPROM)

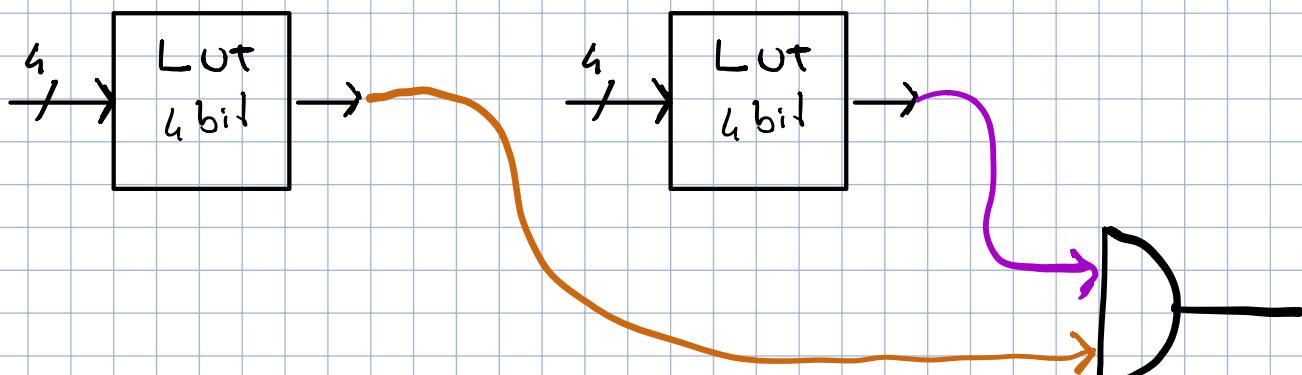


Utilità:

Se cambio la tabella di verità, nel primo caso devo rifare tutto, nel secondo caso con la memoria posso facilmente cambiare l'implementazione cambiando solo i bit in memoria

Ossevazione

Se devo implementare una funzione che ha in ingresso 8 bit e 1 bit in uscita con LUT da 4 Bit

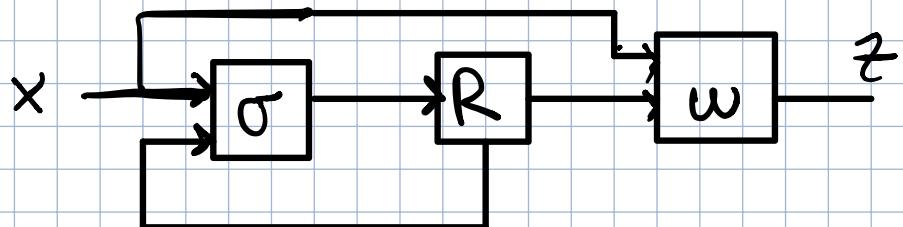


Se progetto con le tabelle di verità conviene ma ogni volta devo fare cd-bc per le mie soluzioni

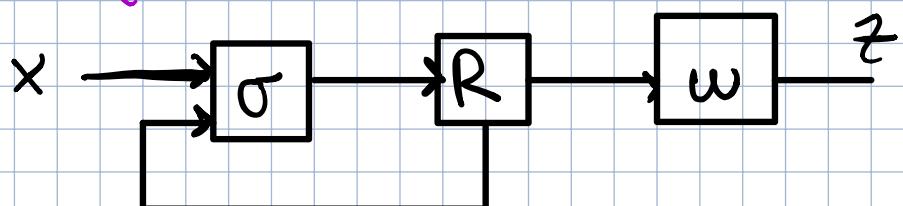
Reti sequenziali:

Mealy

- Automi

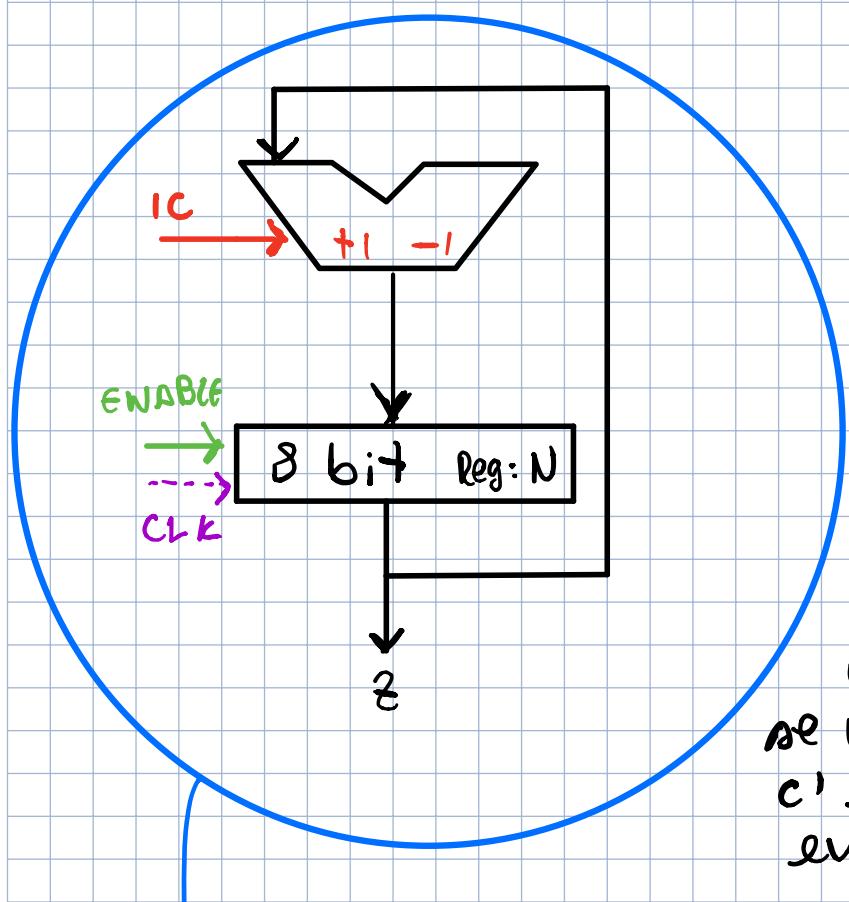


Moore



- Componenti standard: Mux, dec, reg, --

Esempio: Contare persone che entano/escono da una stanza



$$\left. \begin{array}{l} IC = 0 \Rightarrow +1 \\ IC = 1 \Rightarrow -1 \end{array} \right\}$$

ENABLE
/
0
se non
c'è un
evento
/
1
se c'è
un
evento

↓

INGRESSI: IC, enable

USCITE: 2

o (SIGNA): ALU

Stato interno (registro N): N

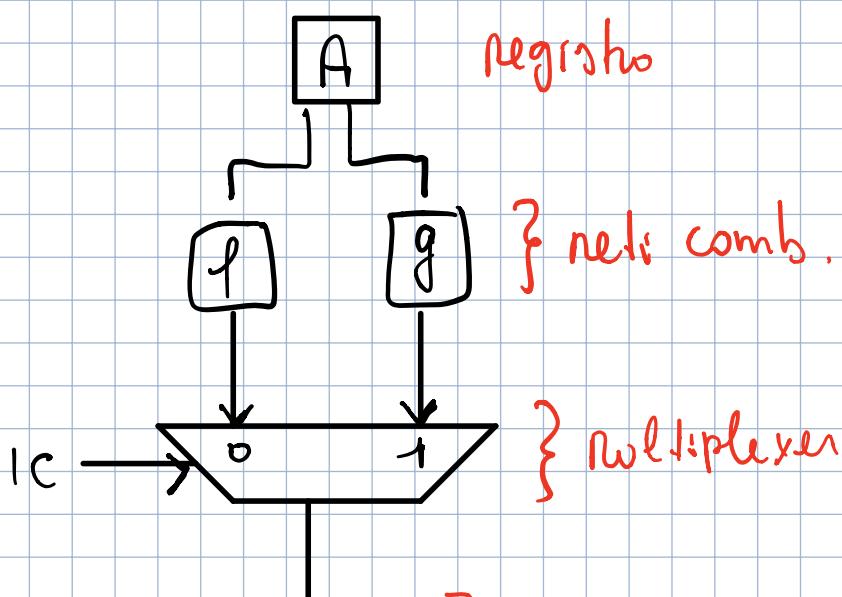
Omega: Prende lo stato e lo fa uscire

Rete di Moore

Sintesi di reti sequenziali con componenti standard

if (c) then $f(A) \rightarrow B$

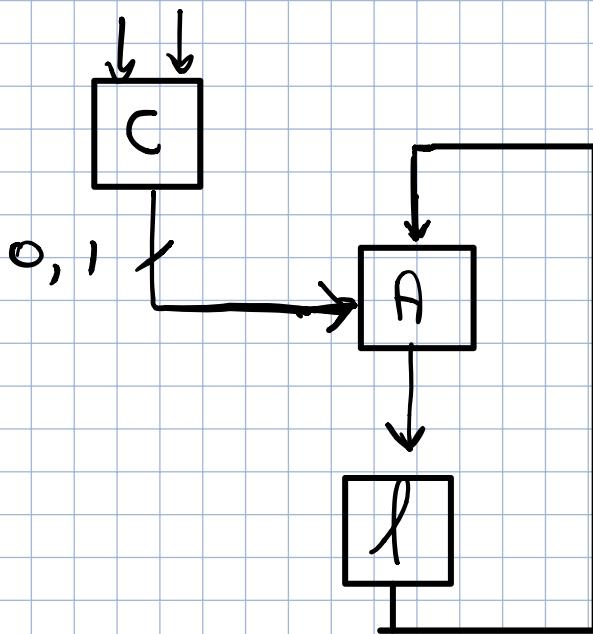
else $g(A) \rightarrow B$



B

{ Registro

2) $\text{while}(c) \{ A = f(A); \}$



3) $\text{switch}(x) :$

case 1 : C₁

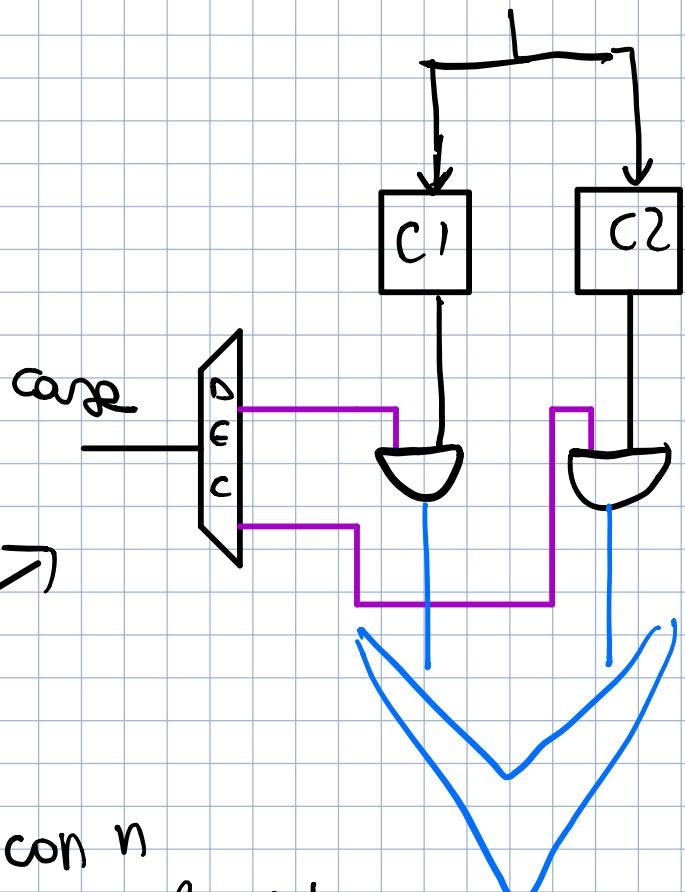
case 2 : C₂

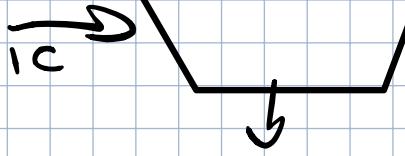
:
:

case n : C_n

C₁ C₂

Case con n





incremental $\log n \text{ bits}$

Parallelismo: Componenti che lavorano nello stesso istante di tempo, svolgendo compiti diversi
è l'obiettivo finale in meno tempo per arrivare a risultato

Latenza: tempo che intercorre fra l'inizio e la fine di un compito (task) (L)
Avolti da un componente

Esempio: leggere un libro su 4 libri da leggere

Tempo di completamento : tempo che intercorre tra l'inizio del primo task e la fine dell'ultimo esempio: leggere tutti e 41 libri
 (t_c)

$$t_c = m \cdot L$$

↓
humano
task

↓
tempo nighth
task

tempo di servizio : intervallo di tempo medio tra le consegne di 2 risoltivi successivi.

Esempio: Se sono de soli, per tradurre un libro impiego L

Se rianimo in 2, per tradurre 2 libri impiego 2 quinot:
in media traduco un libro ogni $\frac{1}{2}$

throughput: Numero di calcoli eseguiti nell'unità di tempo prestabilita' (task)

Stream Parallel

→ Serve ad cumulative il
throughput

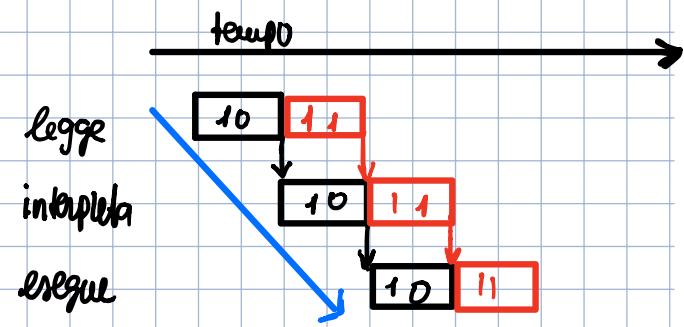
1) Esempio: Un processore esegue un'istruzione alle volte

while (true) {

legge una istruzione (ASN)

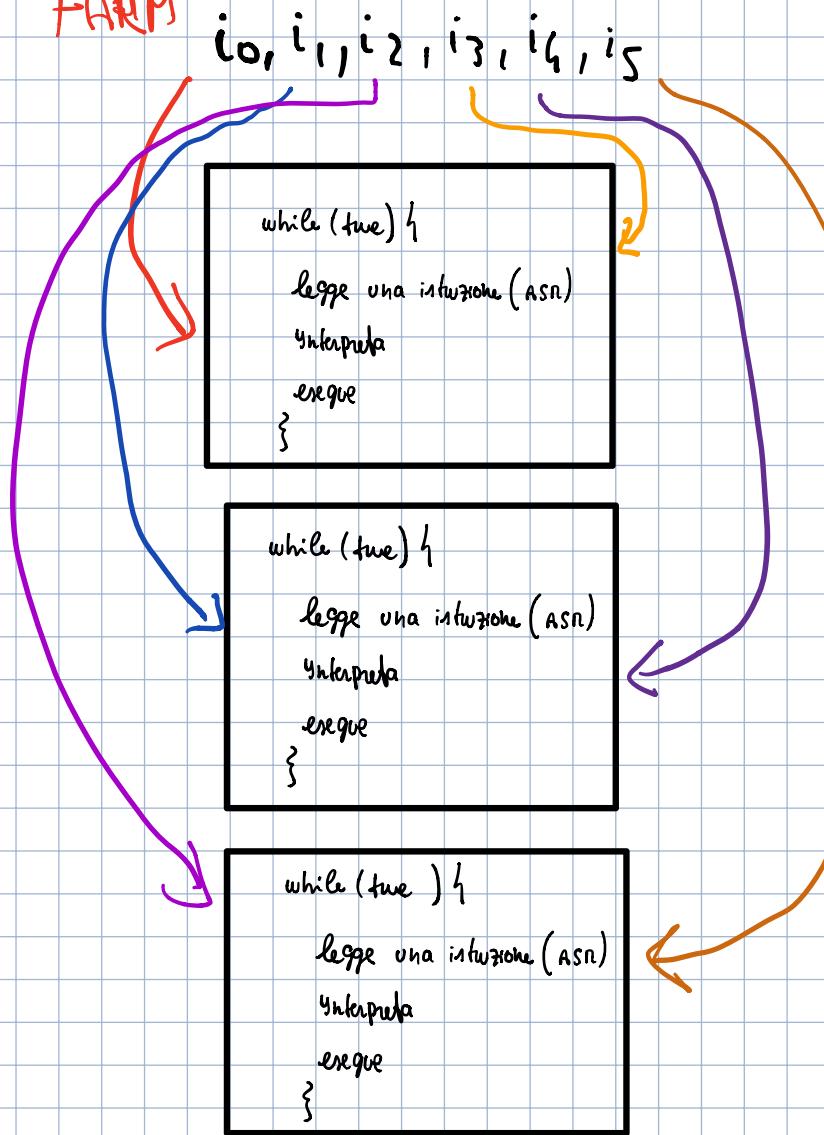
Interpreta

ex que

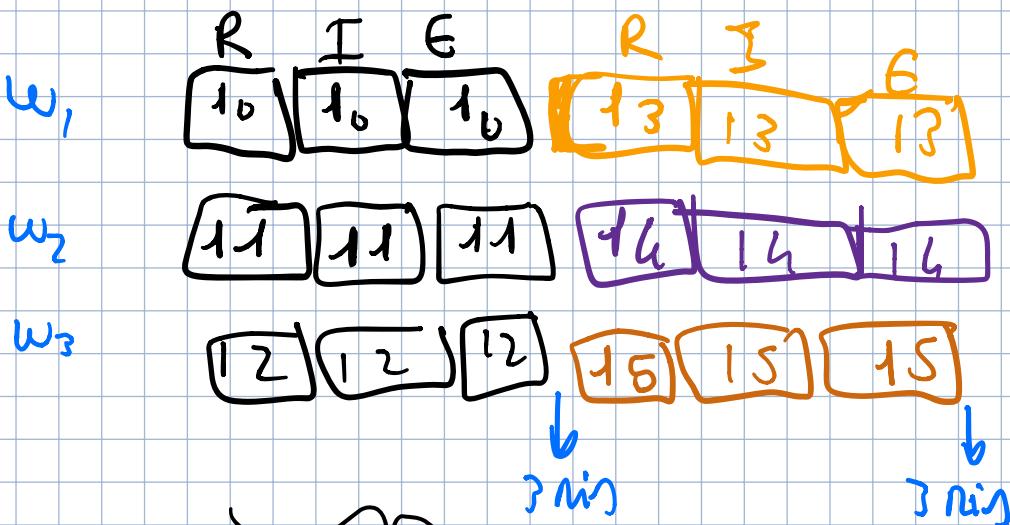


$$t_c = \text{tempo iniziale di riempimento} + \text{tempo} (\propto \ln \cdot \text{tempo della singola riga. di lavoro})$$

2) FARM



Prendo 3 processori, assegno col ognuno un processo, quando uno finisce, prende l'istruzione precedente

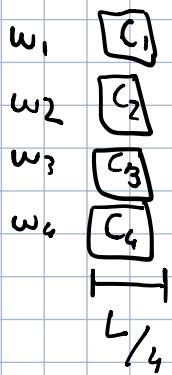


tempo d. servizio: $\frac{1}{3}L$
 (do 3 mis ogni $\frac{1}{3}L$)

Data Parallel

\Rightarrow Serve a diminuire la latenza

$$T_0 \quad \boxed{c_1 \mid c_2 \mid c_3 \mid c_4} \quad t_1 = L$$



Divido le istruzioni di un task e do ognuna ad un worker diverso

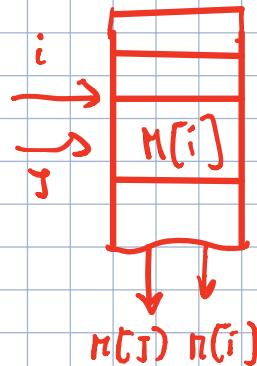
while (true) {

legge una istruzione (ASN)

interpreta

executa

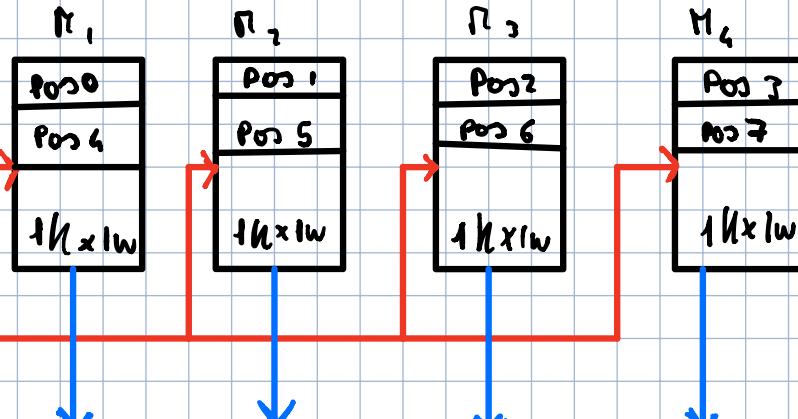
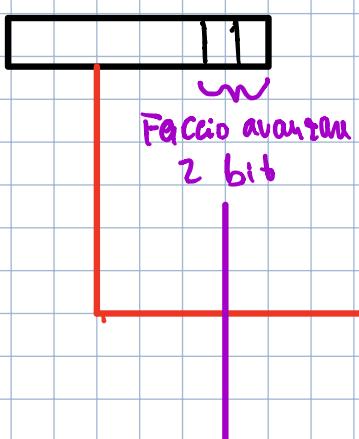
}

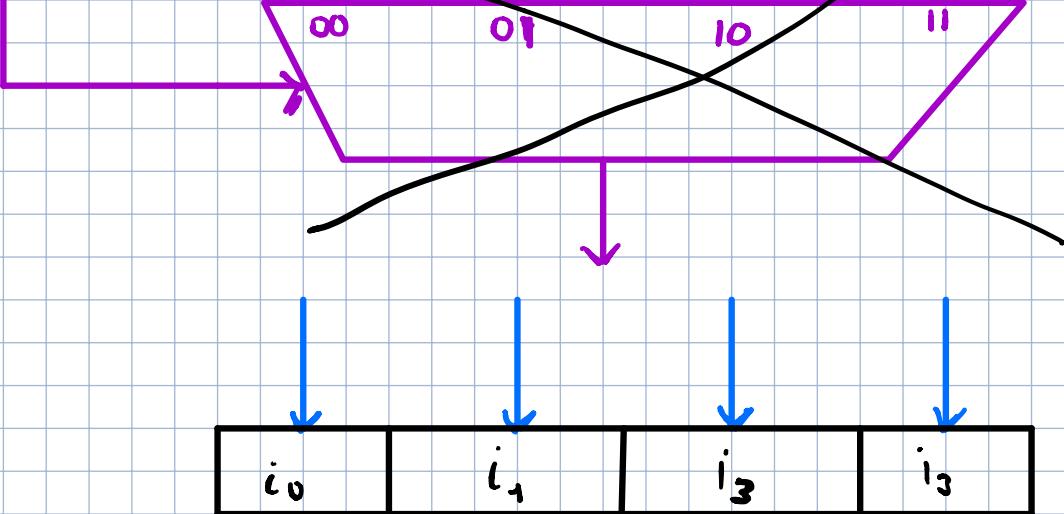


(leggere più istruzioni alla volta)

leggere più istruzioni alla volta per applicare il parallelismo spaziale

INDICE Kbit





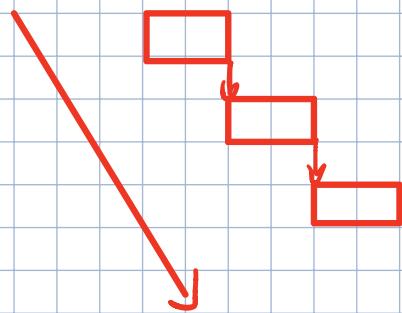
Butta via il multiplexer e prendo ciò che leggo come istruzione che posso fare insieme con il parallelismo spaziale usando i_w che sanno fare decodifica ed esecuzione in parallelo

Processori pipeline: Varie fasi fatte una appena all'altra (pipeline)

Processori superscalari: Usano il Parallelismo spaziale (data parallel)

→ Stream di dati in input

Pipeline :
(STREAM
PARALLEL)
↓
aumenta throughput



Catena di montaggio

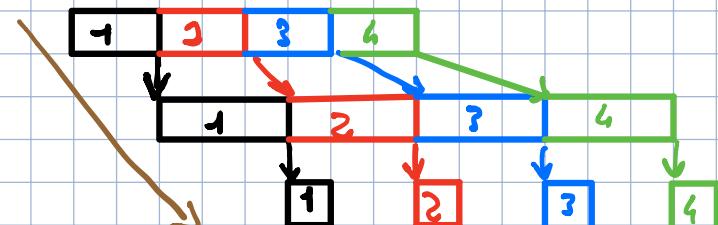
(ognuno fa il suo lavoro e l'output lo invia al successivo)

Aumenta il throughput

Aumenta la quantità di lavori fatti e non diminuisce la velocità per fare cosa

Pipeline con lavori di tempo differente

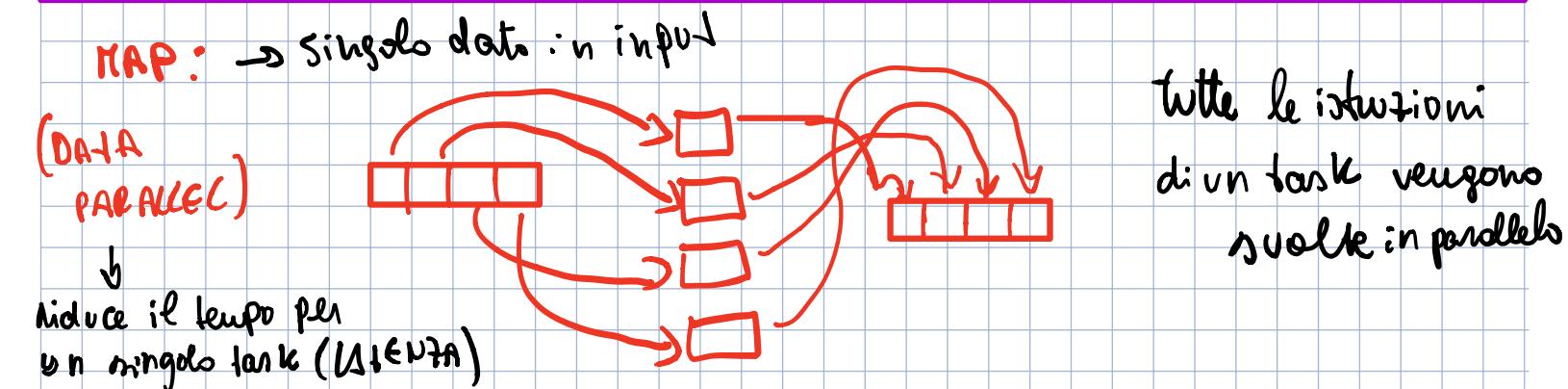
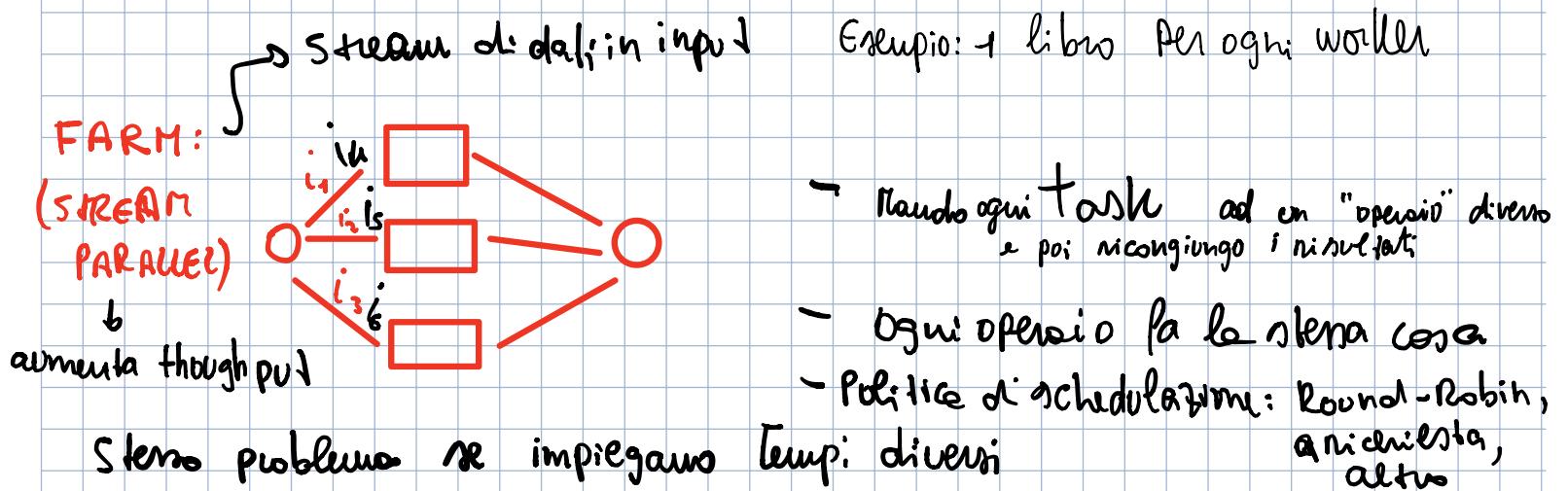
2t w_1 ,
3t w_2 ,
1t w_3



$$\text{Tempo di completamento} = \sum t_i + m \cdot \max \{t_1, t_2, t_3\}$$

↓
tempo con cui
riesce a eseguire
il primo
task

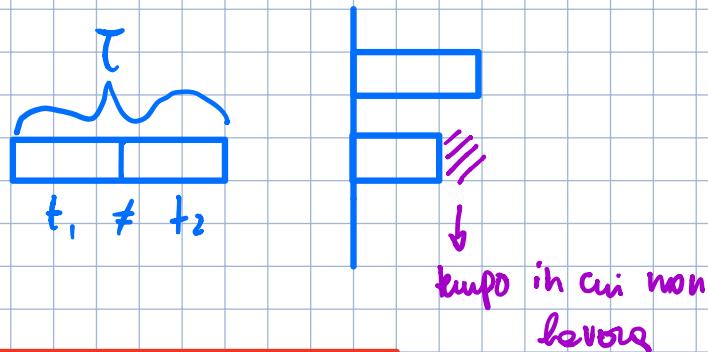
Questa cosa è un problema perché w_3 lavora $\frac{1}{3} w_2$ e w_1 lavora $\frac{2}{3} w_2$
quindi non lavorano tutti con lo stesso tempo (non equo)



- task da 6 istruzioni, lo dividono per ogni Operario e poi ricongiungono i risultati
- Politica scheduling: Round-Robin, altro

Map con tempi diversi

Esempio: 20 pagine di un libro per ogni worker



tempo ideale : $t_{idl}(n) = \frac{T_{req}}{n}$

se non divido e ho lo assegno a nessun worker
Divido in più parti: n e voglio che il mio tempo ideale sia il tempo che ci mettere senza parallelismo
grado d'parallelismo

* operai (grado di parallelismo) grado d'parallelismo

Speed up : $speedup(n) = \frac{T_{req}}{t(n)}$

miglior tempo sequenziale
tempo parallelo con grado d. parallelismo n

$speedup(n) \leq n$ (Quanto ci metto con n operai)



Scalabilità: $scalab(n) = \frac{t(1)}{t(n)} \leq n$

tempo diviso il task in un solo blocco e lo assegno a un worker
 $t(1) > T_{req}$

tempo del task in blocchi e li assegno a n worker

Efficienza: $eff(n) = \frac{t_{idl}(n)}{t(n)} \leq 1$

tempo che vorrei metterci / grado di parallel. n
tempo che ci metto / parall. n

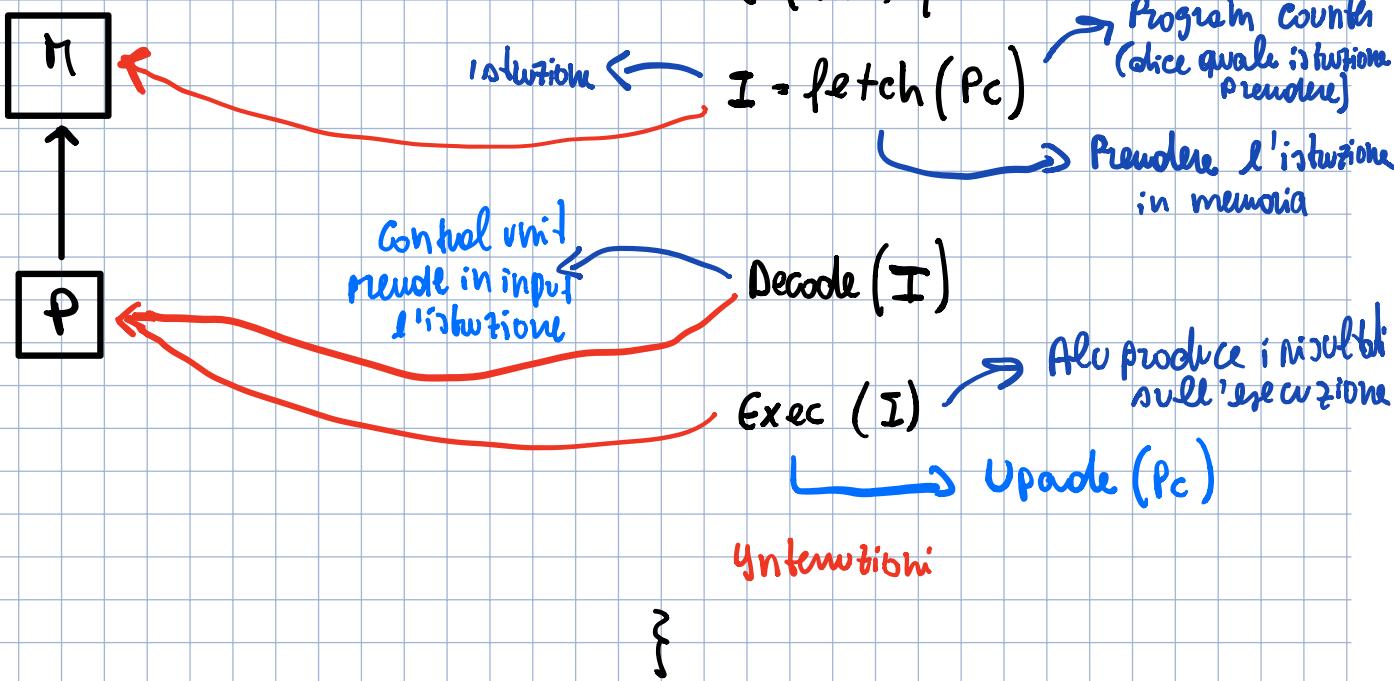
$$= \frac{speedup(n)}{n}$$

Quante volte sono più veloci rispetto a quanto roba usc

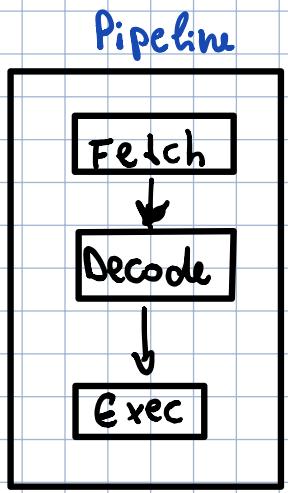
T : A seconda di quale tempo sceglio ho le caratteristiche per quel tempo. Esempio: Se sceglio $T = \text{lentezza}$: $T_{req} = \text{lentezza}$ speedup: speedup sulla lentezza

Procedere:

while (true) {

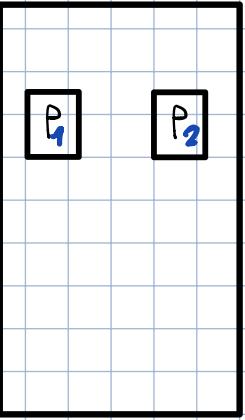


Scatole Processore



parallelismo temporale

Superscalare (comporta che molte ALU o multi-core)



parallelismo spaziale

↓
Aviamo in esecuzione più di un istruzione
ogni ciclo di clock

Problema pipeline: Se ho un'istruzione chi gallo devo riaprire la pipeline del precedente flusso e caricare il nuovo



Se due istruzioni richiedono che una sia finita prima di eseguire l'altra quindi ci sono dei tempi di attesa lunghi

Esempio: $(x \cdot y) + (z \cdot t)$

⇒ Per eseguire la somma devo aspettare che finiscano entrambi.

Pipeline (parallel)

s_1

s_2

s_3



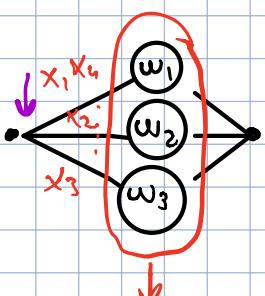
$$1) T_S = \max \{ t_i \}$$

$$2) T_C = \sum_i T_i + (m-1) \cdot T_S$$

3) $\max(\text{speedup}(n)) = n$ stadii della pipeline

4) $t_i = t_S$ stadio i

FARM (Stream parallel)



$x_1, x_2, x_3, x_4, \dots$ = taski interni

\rightarrow grado di parallelismo = n_w = numero worker

Parallelismo

- tempo che un w impiega per un task

= t_w

$$1) T_S = \max \left\{ t_{\text{scuadri}}, \frac{T_w}{n_w}, t_{\text{conze}} \right\}$$

tempo di scelta
worker

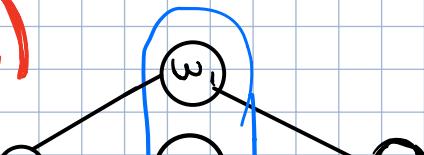
tempo di prendere
res e mandarla null' osata

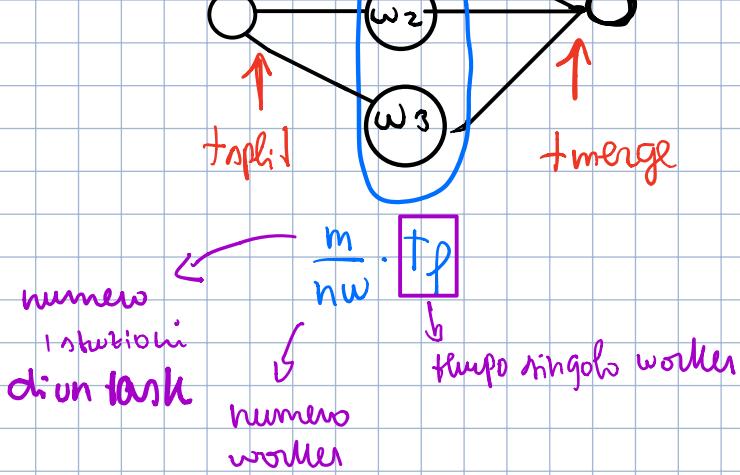
tempo task su nw

$$2) T_C = (t_{\text{scuadri}} + h_w \cdot T_w + t_{\text{conze}}) + (m - n_w) \cdot t_S \approx m \cdot t_S = m \cdot \frac{T_w}{n}$$

$$3) \max(\text{speedup}(n)) = n$$

MAP (Data parallel)





$$1) T_c = t_{\text{split}} + \frac{m}{n_w} \cdot t_p + t_{\text{merge}} = t_{\text{split}} + \frac{T_{\text{req}}}{n} + t_{\text{merge}}$$

$$2) T_s = \max \left\{ t_{\text{split}}, \frac{m}{n_w} \cdot t_p, t_{\text{merge}} \right\}$$
(LATENZA)

$$3) \max (\text{speed}(p(n))) = n$$

Composizione forme di Parallelismo

→ Componenti composizione Stream parallel possono essere :

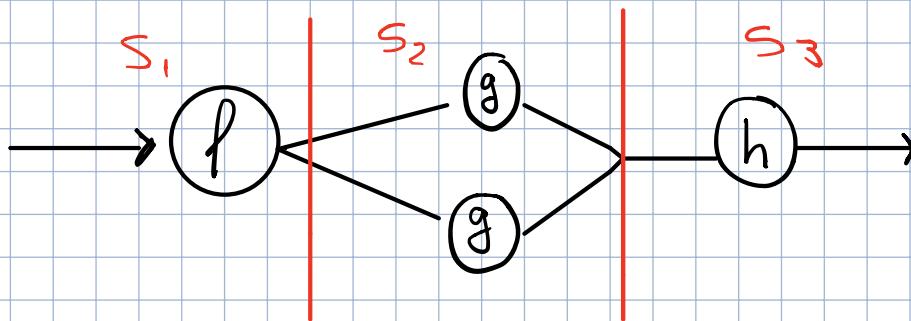
- Data Parallel
- Stream Parallel
- Sequential:

→ Componenti composizione Data parallel:

- Data parallel
- Sequential

Esempio: Pipe + parz

Pipe (s_1 , farm (s_2), s_3)



Essendo una pipeline:

$$1) T_S = \max \{ s_1, s_2, s_3 \}$$

$$= \max \left\{ t_p, \max \left\{ t_{\text{schedule}}, \frac{t_g}{n_w}, t_{\text{call}} \right\}, t_h \right\}$$

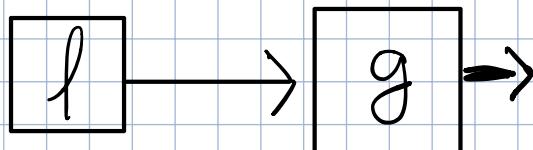
$$= \max \left\{ t_p, \underbrace{t_{\text{schedule}}}_{\text{m. trascurabili}}, \frac{t_g}{n_w}, t_{\text{call}}, t_h \right\}$$

$$\approx \max \left\{ t_p, \frac{t_g}{n_w}, t_h \right\}$$

$$2) T_C \approx m \cdot T_S$$

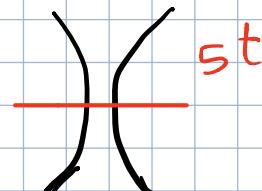
Collo di bottiglia

Se ho la pipeline:



Dove $T_p = 5t$ e $T_g = 2t$

Si verifica un collo di bottiglia :

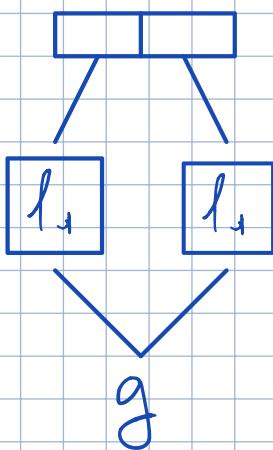


2t

Per risolvere:

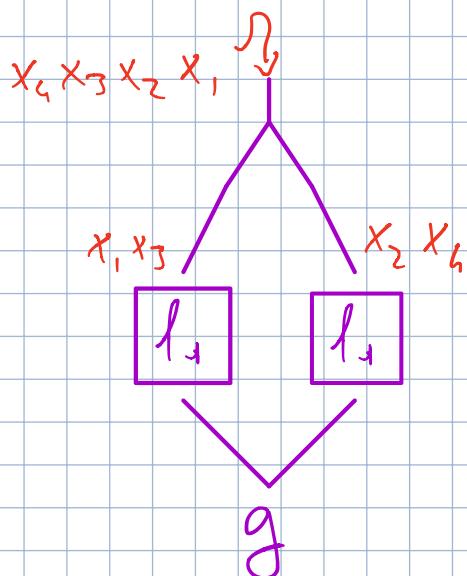
1) Vedere se l'è data parallel \rightarrow è possibile usare la MAP

Se non
si può
usare la
MAP



Singolo vettore diviso in
più parti

2) Renderlo un FARM



Hai meno che ammesso
gli amici vengono assegnati
ai worker

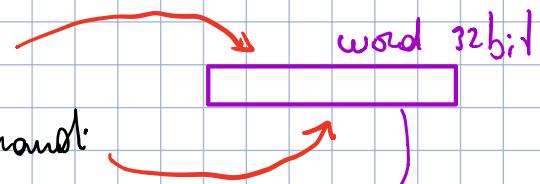
Assembler

Lettura Istruzioni : Sorgenti a destra e destinazioni a sinistra

Esempio
Assembler $\leftarrow \boxed{\text{ADD } R_0, R_1, R_2} = R_0 = R_1 + R_2$

le istruzioni vengono contenute in una parola che contiene:

- bit per identificare l'istruzione
- bit che rappresentano gli operandi:



Procedere:

while (true)

{ Prelevare istruzione

decodificare

→ Decodificare le parole e capire di che operazione si tratta / quali parametri sono coinvolti

eseguire

→ Eseguire l'istruzione decodificata

interruzioni

}

Assembler ARM (Advanced Risc machines)

ACORN

Risc = Reduced Instruction set computer

Principi :

1) Regolarità : Usare gli stessi componenti più volte \Rightarrow Semplicità nella realizzazione delle macchine

2) Supportare il caso più frequente : Usare più operazioni veloci

Risc supporta solo operazioni tra registri

per fare una istruzione

\hookrightarrow ciclo di clock più corto

3) Piccolo e bello : Mantenere le cose piccole quanto basta per essere

efficaci/veloci e fai quello di cui ho bisogno.

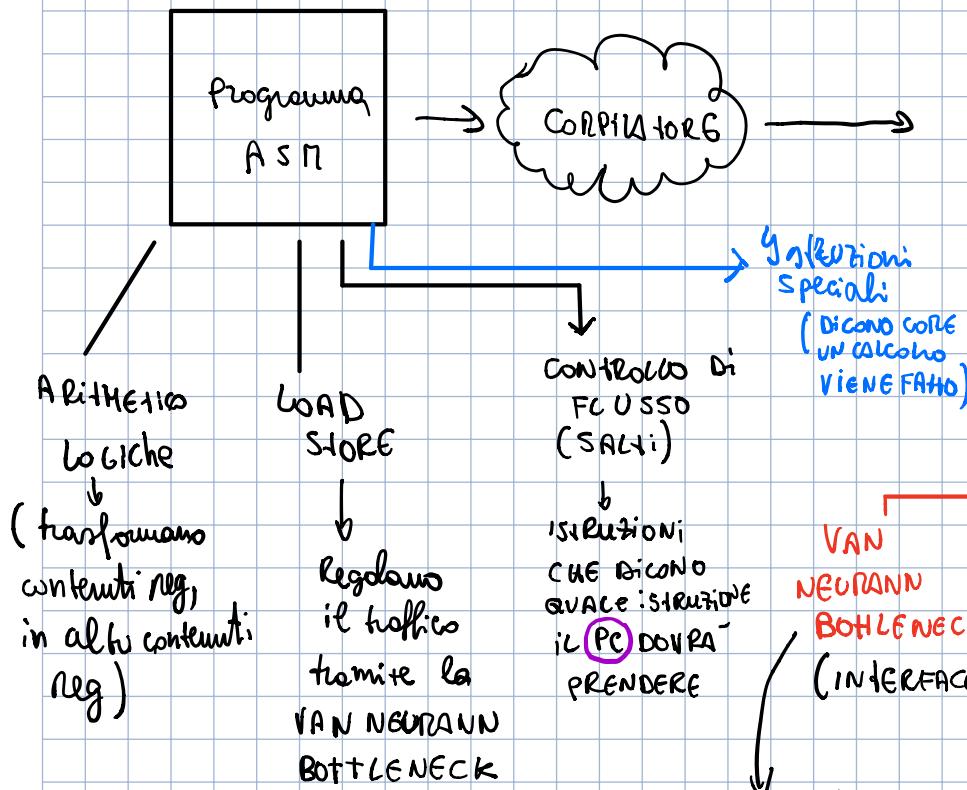
ARM v7 (32 Bit) : 16 registri = 4 bit di indirizzamento

EN: Se prendo 1024 registri permette di fare molte operazioni ma mi costa anche quanto ho da fare poche operazioni

4) Buon risultato è frutto di un buon compromesso (Studio sui risultati e cerco di fermarmi dove i pro e contro si bilanciano)

ASSEMBLER

ADD R0, R1, R2 (Istruzioni)



MACHINA

32 bit

ADD 0,1,2 ecc...

Ponete che indica l'istruzione, i Registi ecc..

Linguaggio
Macchina

Carica

Memoria
funzione

P
PC

* Program Counter che ci dice quale sarà la nostra prossima istruzione (Dove prenderla nella memoria)
(VAN Neumann)

Si AGGIORNA CON LE PROSEGUO ISTRUZIONI
ENTRO LO WHILE

OPERANDI ISTRUZIONI ARM (ARM)

✓ vengono salvate le var meno frequenti

1) REGISTRO "R_n"

16 reg
32 bit

PRASSI (come dovrebbero essere utilizzati)

HARDWARE (come devono essere utilizzati)

R₀ → Parametri, temporanei o valori di ritorno

R₁ - R₃ → Parametri, variabili temporanee

R₄ - R₁₁ → Var. Salvate

R₁₂ → Var. temporanee

R₁₃ → Stack pointer

R₁₄ → Link register

R₁₅ → PC (Program Counter)

2) IMMEDIATI: #1 = Costante 1

↳ Vengono presi da un pezzetto dell'istruzione

in linguaggio macchina

#1

costanti piccoli

permette di non salvare in registri o altro

✓ ogni var inizializzata occorre 32 bit = 4 byte = 4 indirizzi di memoria

3) Memoria (LOAD e STORE)

vengono salvate le var meno frequenti

32 bit per gli indirizzi e 32 bit per le parole (word)

[R_n, #-]

Base

offset (Byte)

LDR

leggere una parola di dato della mem in un registro

L'indirizzo che andiamo a leggere o a scrivere nella memoria è dato da :

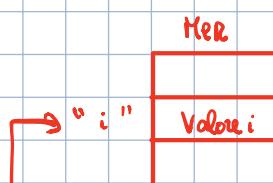
$$\text{Ind} = \text{base} + \text{offset}$$

Esempio:

1) Programma C

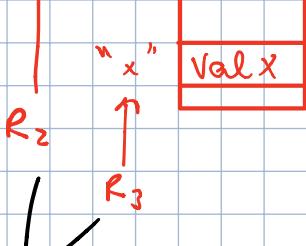
int i;

/*



Per prendere il valore di *i* dovrà come indirizzo, l'indirizzo di *i* e

float x;



come offset #0

LDR R0, [R2, #0]

LDR R3, [R2, #0]

non contigui (Nou no accedere
da R2 all'ind di R3
e viceversa)

indirizzo di "x"

indirizzo di "x"

commento in
ASR

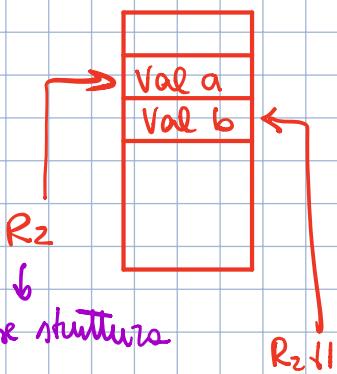
2) Programma C

Struct {

int a;
float b;

};

Ind. Base struttura

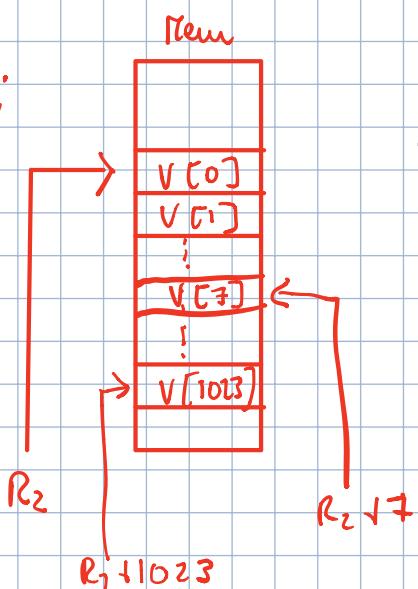


LDR R0, [R2, #0]; valore di a

LDR R1, [R2, #1] j valore di b

3) Programma C

int v[1024];
x = v[7];



LDR R0, [R2, #7]

Importante: A seconda del tipo degli elementi salvati devono saltare più posizioni da un elemento all'altro

E esempio:

int n[]

$\text{sizeof(int)} = 4$

$n[0] \rightarrow \text{ind}$

$n[1] \rightarrow \text{ind} + 4$

$n[2] \rightarrow \text{ind} + 8$

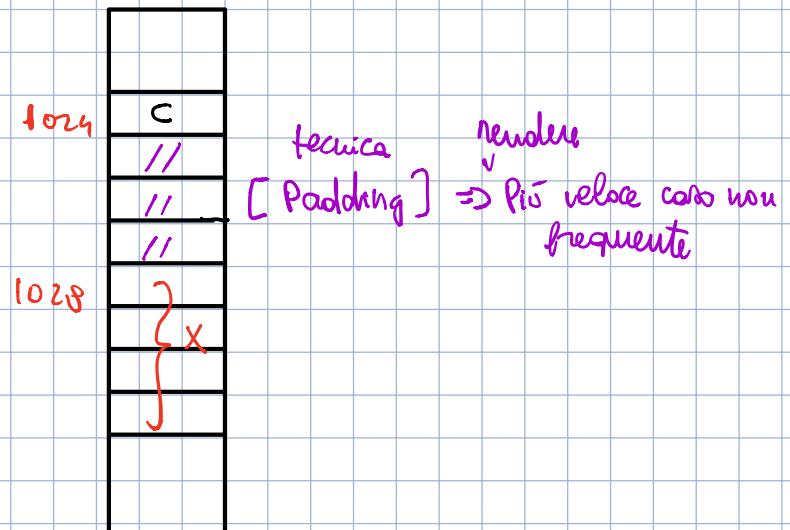
} Perché la memoria è al
byte quindi $n[0]$ occupa 32 bit = 4 byte

Allineo a 4 byte perché l'architettura è a 32 bit = 4 byte

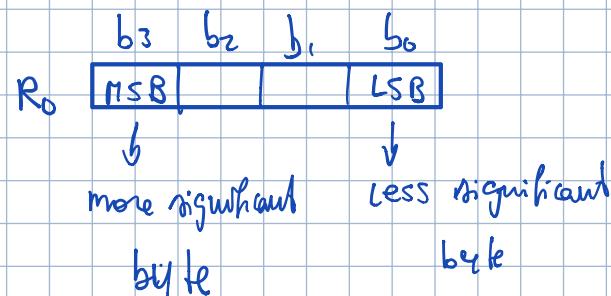
Quindi non posso allineare x4 ma devo farlo 2 accari in memoria

Esempio:

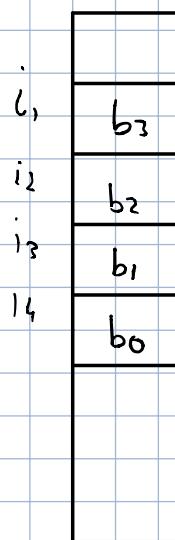
struct {
 char c;
 float x; };



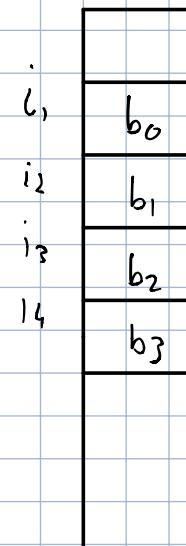
Rappresentare by k in memoria:



1) Big endian



2) little endian



ARM
Li SUPPORTA
entrambi

Gestione speciale

SE1END

(specifica se lavorare
in big o little endian)

CLASSI DI ISTRUZIONI

- AND (tre negativi)

R₂ 0101
R₃ 0110

1) Arithmetico LOGICHE

LOGICHE

- ORR (OR normale)

- EOR (esclusive OR)

- BIC R_1, R_2, R_3 (maschera)

6 SORGENTI

maschera

4 bit a "1" della
maschera ci dicono
i bit da attivare
nello source

ARITMETICHE

- ADD

ADC

CON RIPORTO

DAL BH PRECEDENTE

- SUB

SBC

 $R_1 = R_3 + R_2 + \text{Carry}$ (operazioni
precedenti)

$$\text{SUB } R_1, R_2, R_3 = R_1 = R_2 - R_3$$

- CMP R_1, R_2 Fa $R_1 - R_2$ e
setta il flag

RiTorna un flag:

1) Z \Rightarrow Dice se il risultato è uguale
a 02) N \Rightarrow // è negativa
 \hookrightarrow bit $31^{\circ} = 1$ 3) C \Rightarrow Riporto sull'ultima cifra4) V \Rightarrow Se operazione da overflow

Per impostare il flag di riporto nell'operazioni tipo ADD bisogna mettere la S infondo:

ADD S

4 flag vengono salvati in una parola di stato

(REGISTRO NON GENERAL PURPOSE)



Contiene anche
bit endianism
per sapere come
svolgono le OP.

Non si può scrivere
o leggere ma si può
accedere implicitamente
con alcune istruzioni

Si puo' fare la comparsa direttamente in un'istruzione:

ADDQ $R_1, R_2, R_3 \Rightarrow$ Fai l'operazione di add se e solo se

www
ZCAR ZCAR

l'istruzione precedente ha impostato
il flag EQ = 1

$$- RSB \quad R_0, R_1, R_2 \quad = \quad R_0 = R_2 - R_1$$

↳ Utile perché una costante non la posso mettere al posto di R_1

R_2 lo posso chiamare anche SRC2 che indica che puo' essere:

- Registratore
- Costante

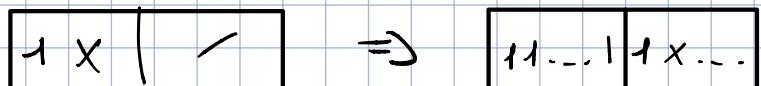
- Operazioni di:
shift

shift logico (se non voglio mantenere il segno)



Se faccio lo shift
a sinistra non
ho differenze tra
shift logico e aritmetico

Shift Aritmetico (se voglio tenere il segno)
negativo



ASR (shift aritmetico a destra)
LSL (shift logico a sinistra)
LSR (shift logico a destra)

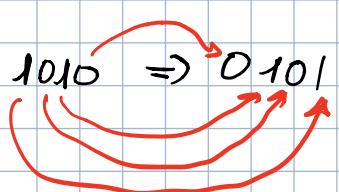
} tre operandi: LSL $R_0, R_1, \#...$

$$R_0 = R_1 \ll \#...$$

Sposta di $\#...$ posizioni
a sinistra

- ROR (rotate right) \Rightarrow Sposta tutti i bit tranne l'ultimo a destra
e mette l'ultimo in cima

Esempio:



- MUL $R_0, R_1, R_2 \Rightarrow$ Se R_1 e R_2 sono di 32 bit e ho un overflow, basta via la parte minima, fermo la parte a destra, e dati flag overflow = 1

Primi 32 bit del risultato

$$\xrightarrow{\text{UNULL}} R_0, R_1, R_2, R_3 = \boxed{R_0 | R_1} = R_2 * R_3$$

Divide in 2 registri il risultato

Signed \rightarrow con segno

- MOV $R_0, SRC2$ sposta in R_0 il valore di $SRC2$

- LDR R_0, \dots Carica nel reg R_0

ISTRUZIONI DI SALTO

In generale quando passo da un'istruzione all'altra del PC faccio:

PC + 4

- BRANCH (B)

B etichetta

INDIRIZZO DI MEMORIA

B offset

Numero da aggiungere / togliere al pc corrente

Al posto di fare $PC + 4$ alla fine del while del processore faccio $PC + offset$
quando vado l'istruzione B

Potrei utilizzare le condizioni nell'BANCH e quando accade se è solo se le condizioni sono vere.

B XX

↓ esempio

BEQ \Rightarrow fa il B solo se solo se $EQ = 1$

Flag per l'

Istruzioni ASR per fare un for ($i=0; i < n; i++$)

Prendo $R_1 = i$ e $R_2 = n$

Mov $R_1, \#0$; Azzero R_1

loop: CMP R_1, R_2

BGE fine

≡
≡
≡

ADD $R_1, R_1, \#1$

B loop

fine: ≡
≡
≡

Direttive: Istruzioni che vanno sul compilatore ASM e non sul processore

Sono indicate dal .

Potrei richiamare le direttive tramite etichetta

Riservazione parti di memoria per i dati

1) • Data : Indica che sto passando dati e non istruzioni in quella parte del programma

• word : Permette di definire una o più parole di memoria che hanno valori che seguono separati da virgole. **val 1, val, val?**

Dice: Prendi un'area di memoria e fa in modo che dentro ci niente questi valori.

Esempio: .word 123 → Riserva una parola che dentro ha il valore 123

• word 1,2,3,4 → Riserva 4 parole consecutive le quali la prima ha val 1, la seconda ha val 2 ecc...

• byte : Permette di riservare un byte o più byte **val {, val, val?}**

• 2 byte : Permette di riservare 16 bit

• 4 byte : Permette di riservare 32 bit = alle word

• ascii : Permette di riservare memoria per contenere una stringa di caratteri

• Space : Riservare n byte di memoria

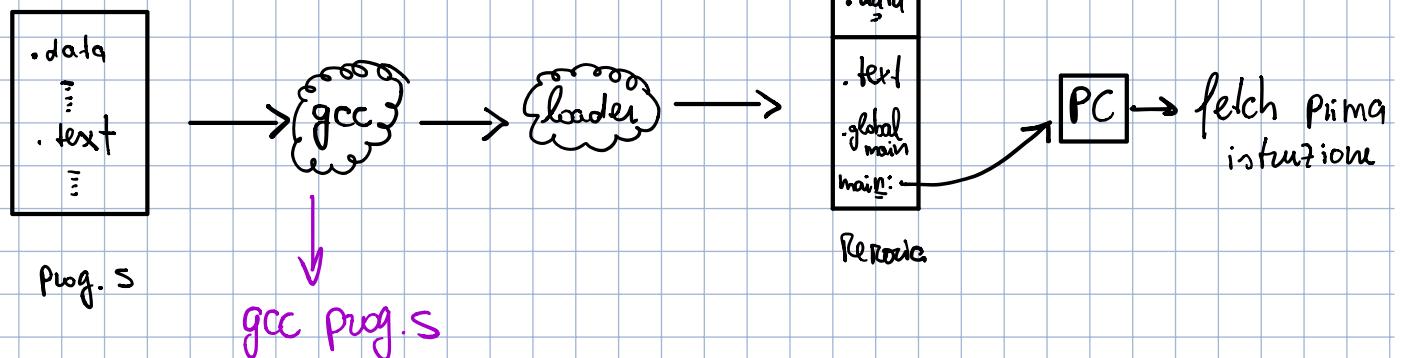
2) • nodata : Indice dati su cui posso solo scrivere

3) • bss : Dati non initializzati

4) • text : Sezione che contiene il testo del programma

• global **main** : Definisce l'entry point del programma
etichetta

Le direttive vanno a finire in memoria e PC prende l'etichetta dell.global come prime istruzioni



Per vedere l'uscita: `./a.out | od -d`

Pseudo istruzioni: Istruzioni che vengono tradotti e vanno a finire sul processore

Istruzioni del compilatore che compila la sorgente scritta in ascii
Non sono istruzioni in ASN ARM
all'assemblatore ARM

LDR Registru, = constant
etichetta

registro ← costante

registro ← globalizzata etichetta

Esempio:

`G + CHE + A`

`x: word 123`

`;`

`LDR R0, =x` Pseudo istruzione

`LDR R1, [R0, #0]` Commento il numero 123

`COSTANTE`

`LDR R0, =123`

CHIAMATE AL S.O.

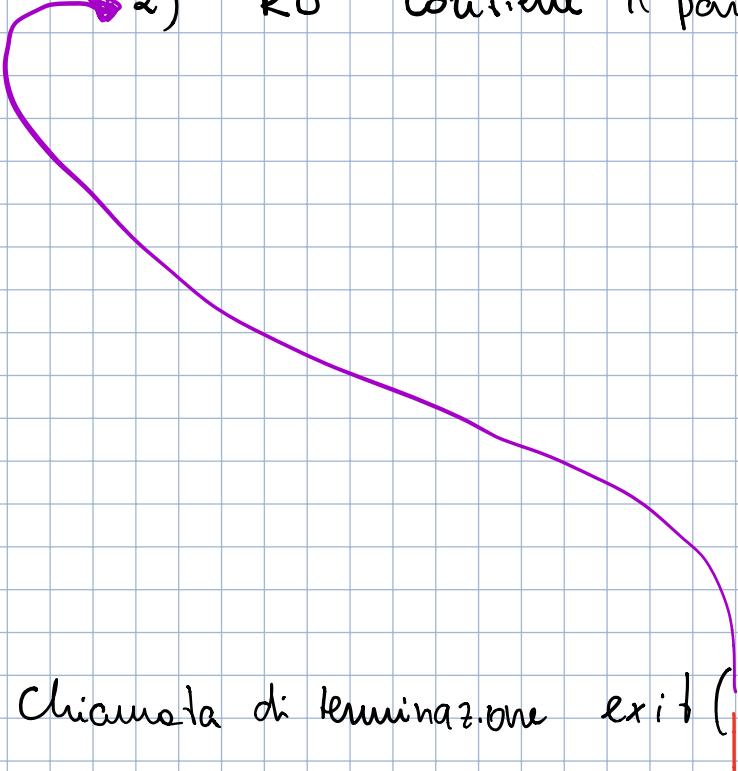
Funzioni di libreria messe a disposizione dal S.O.

Alcune Regole
procesi

Alcune regole
I/O

le chiamate al S.O. in ARM v7 utilizzano la seguente convenzione

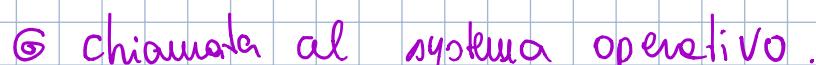
- 1) R7 contiene il motivo della chiamata $\Rightarrow \#$ Syscall
- 2) R0 contiene il parametro che passiamo alla Syscall

Chiamata di terminazione exit ()

codice di ritorno

Syscall delle exit = 1

codice

Mov R7, #1

SVC 0 

Fa quello che viene detto di fare in R7

chiamata write (fd, buffer, #byte)

- **fd** = Descrittore di file (intero)



Indice/tabella di file aperti che contiene tutto ciò che serve per lavorare sul file

std in	0
stdout	1
stderr	2

vettore con posizione 0, 1, 2 riservate

Syscall della write = 4

R₀ = fd

R₁ = Indirizzo buffer

R₂ = # Canti da scrivere

- **buffer** = Puntatore ad un'area di memoria (char*)

- ***byk** = Quanti byte dobbiamo trascrivere sul descrittore puntato da fd

Codice

mov R0, #1 ⑥ Descrittore stdout (perché = a 1)

ldr R1, =x ⑥ Indirizzo buffer

mov R2, #4 ⑥ Numeri byte da scrivere (4 in questo caso)

mov R7, #4 ⑥ Syscall = 4

SVC 0 ⑥ Syscall

mov R7, #1 } ⑥ Termination Programma
SVC 0

Costrutti

1) For

⑥ initializzazione i=0

elichetta: ⑥ testare condizioni

CNP ...,-

BEQ elicfine



? corpo ciclo FOR

ADD R_i, R_i, #1

⑥ increments i

B elichetta for

→ effic fine:

—
—
—
—
—

⑦ continua programma

2) IF then , IF then else

IF then

1)

⑥ testare condizioni

CMP R₁, R₂

BEQ then

cont :

—
—
—
—
—

then :

—
—
—
—
—

B cont

IF then else

⑥ testare condizioni

CMP R₁, R₂

BNE else

then :

—
—
—
—
—

B cont

else :

—
—
—
—
—

⑦ Se la condizione i vera esegui
nomo then e dopo salta l'else
e va al resto del programma.
Se la condizione è falsa salta il
nomo then e poi il nome else

cont:

—
—
—
—

3) switch case

② festeggia la prima le condizioni

Case 1 {

CMP R0, #1	③ Se non è questo caso vai al succ.	}	REVEQ R0, #100
BNE next1			
Mov R0, #100			
B fine ④ Break			

next1: CMP R0, #2

BNG next2	③ Se non è questo caso vai al succ.	}	MOVEA R0, #200
Mov R0, #200			
B fine ④ Break			

next2: Mov R0, #300

fine:

—
—
—

4) while

loopw: CMP R1, #0 ② festeggia condizioni

BLS fine ③ Se non è >0 Vai a fine (interrup;
loop)

B loopw ④ Ciclo

fine:

—
—
—

⑤ Resto programma

Tabella 6.3 Mnemonici di condizione.

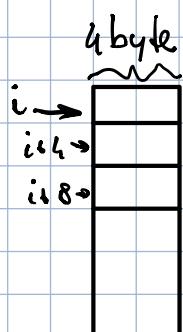
cond	Mnemonico	Nome	CondEse
0000	EQ	Uguale (Equal)	Z
0001	NE	Diverso (Not Equal)	\bar{Z}
0010	CS/HS	Attiva riporto/maggiore o uguale senza segno (Carry Set/unsigned Higher or Same)	C
0011	CC/LO	Disattiva riporto/minore senza segno (Carry Clear/unsigned Lower)	\bar{C}
0100	MI	Meno/negativo (Minus/negative)	N
0101	PL	Più/positivo o nullo (Plus/positive or zero)	\bar{N}
0110	VS	Traboccameto/attiva traboccameto (overflow/oVerflow Set)	V
0111	VC	No traboccameto/disattiva traboccameto (overflow/oVerflow Clear)	\bar{V}
1000	HI	Maggiore senza segno (unsigned Higher)	$\bar{Z}C$
1001	LS	Minore o uguale senza segno (unsigned Lower or Same)	Z OR \bar{C}
1010	GE	Maggiore o uguale con segno (signed Greater than or Equal)	$\bar{N} \oplus V$
1011	LT	Minore con segno (signed Less Than)	$N \oplus V$
1100	GT	Maggiore con segno (signed Greater Than)	$\bar{Z} (N \oplus V)$
1101	LE	Minore o uguale con segno (signed Less than or Equal)	Z OR ($N \oplus V$)
1110	AL (o niente)	Sempre/incondizionato (Always/unconditional)	Ignorato

Gli mnemonici di condizione differiscono confronti tra numeri con e senza segno. Per esempio, ARM offre due tipi di confronti maggiore o uguale: HS/CS è usato per numeri senza segno e GE per numeri con segno. Per numeri senza segno, $A - B$ genera un riporto di uscita (C) se $A \geq B$. Per numeri con segno, $A - B$ porta N e V entrambi a 1 se $A \geq B$. La Figura 6.7 sottolinea le differenze tra i confronti HS e GE con esempi di numeri a 4 bit per facilitare la comprensione.

Senza Segno

Con Segno

Mem



Memoria indirizzata al byte = Ogni 4 byte ho un registro

tipi di indirizzamento oltre a LDR R1, [...]

1) Preincremento (Preindice)

$$\text{LDR } R_1, [R_2, R_3]! \leftarrow \begin{cases} R_1 \leftarrow R_2 + R_3 \\ R_2 \leftarrow R_2 + R_3 \end{cases}$$

2) Postincremento (Post indice)

$$\text{LDR } R_1, [R_2], R_3 \leftarrow \begin{cases} R_1 \leftarrow R_2 \\ R_2 \leftarrow R_2 + R_3 \end{cases}$$

↑
snc 2

(o neg o costante)

GO:

1) loop: LDR R₂, [R₁], #4

;

.

Se e solo se R₁ < base + 4N allora forma un loop

2) for (i=0 ; i < n ; i++)

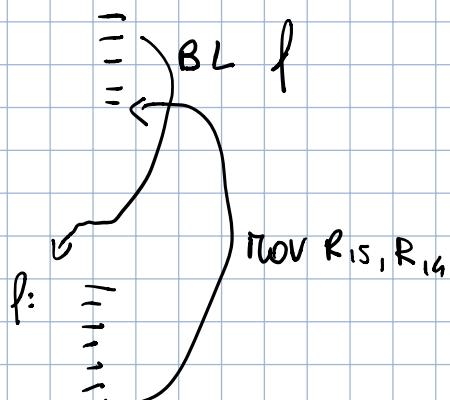
x[i] ++;

LDR R₂, [R₁]

++

STR R₂, [R₁], #4

Funzioni / Metodi / Procedure



1) nel R14 (link reg.) salva l'indirizzo
della prossima istruzione PC+4
2) PC \leftarrow "etichetta"
(Branch & link)

Parametri:

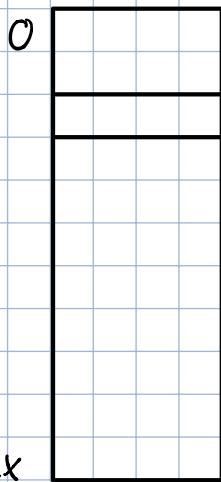
$f(x) \xrightarrow{x} \text{return } y$

Parametri main \rightarrow fun 6 param
max
 R_0, R_1, R_2, R_3

Parametri main \leftarrow fun
 R_0

Se abbiamo più di questi parametri
li passiamo attraverso lo stack

Stack Pointer



Se faccio una push scrivo in basso

→ Può crescere anche verso l'alto

Usare lo stack \Rightarrow lo stack cresce verso il basso (conventionalmente)

chiamante (Rain) 3 param da 1 word

alloc spazio \rightarrow SUB SP, SP, #12

Salvo R_{van1} [SP, #0] 1° pos

Salvo R_{van2} [SP, #4] 2° pos

Salvo R_{van3} [SP, #8] 3° pos

BL f .



← SP

chiamato (f)

LDR R_i, [SP, #0] \rightarrow per prendere i parametri

ADD SP --

Per far ritornare lo stack pointer alle pos.

dopo i parametri

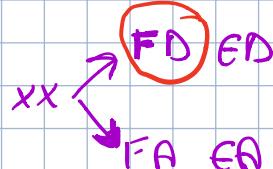
(riallocazione mem dello stack ris ai param.)

Istruzioni per salvare blocchi di registri e caricarli

STNXX

Push {R_i, R_s} Carica R_i e R_s

(Intervi multipli)



LDM XX

(Load multiple)

Folle Empty

✓ da dove si parla

Descent A scendere



Decremente crescente

$\{ R_1, -R_5 \}$ Lancica i reg da R_1 a R_5

Pop

$\{ R_1, R_5 \}$ Rimuovi R_1 e R_5

$\{ R_1, -R_5 \}$ Rimuovi da R_1 a R_5

Penso o devo usare lo stack?

1) Penso quando non ho parametri di ritorno o ne ho da passare pochi e riferimenti pochi (e non ho da salvare i valori di R_0, R_1, R_2, R_3 per usarli successivamente)

2) Devo quando ho una funzione Ricorsiva



Devo quando utilizzo

Quando chiamiamo una funzione quando sono già in una funzione

Registri da R_4 in poi

devo salvare il link register per tornare al main

Perché prima di utilizzarli

vogli salvare i valori del chiamante



Perché devo salvare anche i valori precedenti quindi li riguarderò sui registri li perderei



Ad ogni punto di ricorsione devo salvare i reg. che conoscevo il punto dopo che mi servono da utilizzo e il link register

LDR

LDR R_{dest} ,

} = etichetta
= valore

Se carico un'etichetta, carico l'indirizzo quindi per caricare il valore dovrò fare una seconda LDR

X: word 123

← Riservo una parola di mem. con valore 123
Se metto più valori separati da una virgola

LDR R₀, =X ← carico l'indirizzo

LDR R₁, [R₀, #0] eq. LDR R₁, [R₀] ← carico il valore 123

LDR R₁, =123 ← metto il valore 123 in R₁

riservo delle
di mem.
consecutive

Specifica no SRCZ

en: ADD [S][xx]

R₀, R₁, SRCZ

#immediato

reg

Reg shiftato →

LSE
LSL
ASR
ROR

Quando eseguo un'operazione posso mettere un'inistruzione che agisce sull'immediato

Prima di eseguire l'operazione

ADD R₀, R₁, R₂ LSL #2,

R₂ shift a sinistra di 2 posizioni

Reg base
array

prima di eseguire la ADD

STR R₀, [R₁, R₀ LSL #2] esempio load

Riassunto convolutione: passaggio param.

$$F(x_1, \dots, x_6) \rightarrow y$$

PARAT. IN ENTRATA

Mov R₀, "x₁"

Mov R₁, "x₂"

Mov R₂, "x₃"

Mov R₃, "x₄"

PUSH {x₅, x₆}

PARAT. IN USCITA

Mov R₀, "y" = R₀

BC
POP hxs, x6 }

Funzione main

```
int main ( int argc, char * argv[] ) {
```

;

return ()

}

ASN

global main

main:

=
-
-
-
-

R0 argc

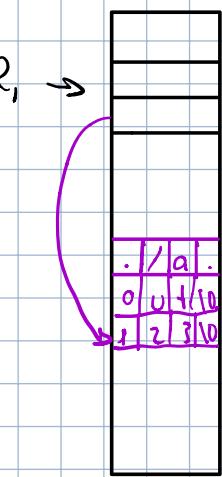
R1 indirizzo dell'array
di puntatori delle
stringhe degli argv[i]

}

Argomenti passati

per parametro
quando eseguo

.a.out



Prendere parametro 123

./a.out 123

LDR R2, [R1, #4]

./a.out 123 10

LDR R3, [R1, #8]

Il parametro viene preso in ASCII

ASCII to integer:
1) Prende 1 per volta il carattere ASCII

2) Utilizza un accumulatore che parte da 0 per neg valori intermedi

3) Faccio: (Accumulatore * 10) + ("carattere" - "0")
↓
valore ASCII 0 (48)

valore ASCII
del carattere

4) Metto il risultato nell'accumulatore

PRINTF dato

fmt: " Risultato=%d "

Format String PRINTF

Prima di richiamarne

Quando richiamiamo la funzione:

R₀ → contiene il format della string

R₁ → 1° parametro v.d

R₂ → 2° parametro v.d

R₃ → -

R₄

:}

→ Parametri successivi vanno salvati sullo stack → push di R₄, R₅, ...

Per fare il G, S, ... parametro devi perforza fare un push di R₆, R₇, ... ecc... Per poter fare anche di R₁, R₂ nel caso mi additivo

Salvo LR → push {PC}

Dopo riprendo LR nel PC

Pop {PC}

Funzioni Ricorsive

fact: cmp R₀, #1 → CifP se sono al caso base

se non sono al caso base
bne Ric → Se non voglio restituire il valore di R₀ ma il caso base
richiede un altro valore
mov PC, LR → Return valore caso base

Ric: push {R₀, LR} → Eventuali valori che mi dero ricordare
sub R₀, R₀, #1
* INDIRIZZO DI RITORNO

bl fact → Fin quando non arrivo al caso base faccio il loop

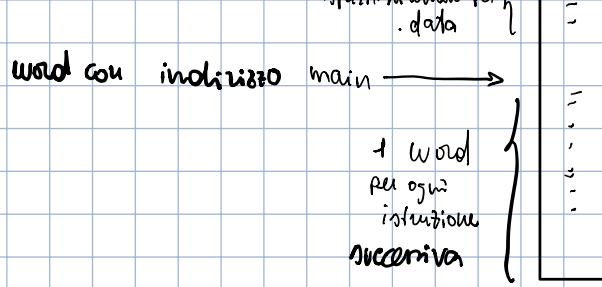
pop {R₁, LR} → Riprendo i valori che mi servono al ritorno

mul R₀, R₁, R₀ } CALCOLO risultato in R₀ e lo restituisco
mov PC, LR

In Assembler posso fare come in C e richiamarmi funzioni
da file diversi compilandoli insieme
↓
faccendo bl "Funzione"

Quando compilo un programma il compilatore: Esegue 2 "passe"

- 1) legge il codice → Capisco quanto spazio in memoria mi serve per il programma



2) Genero codice del programma (codice macchina)

Memorie generali di un calcolatore



↓ ↑
→ Stack, Heap

→ Dati che non cambiano di lunghezza, area di mem. che non cambia di dim.
Esempio: .ascii, .space ...

convenzione Dati
dinamici

Max

stack dal max verso il basso

min

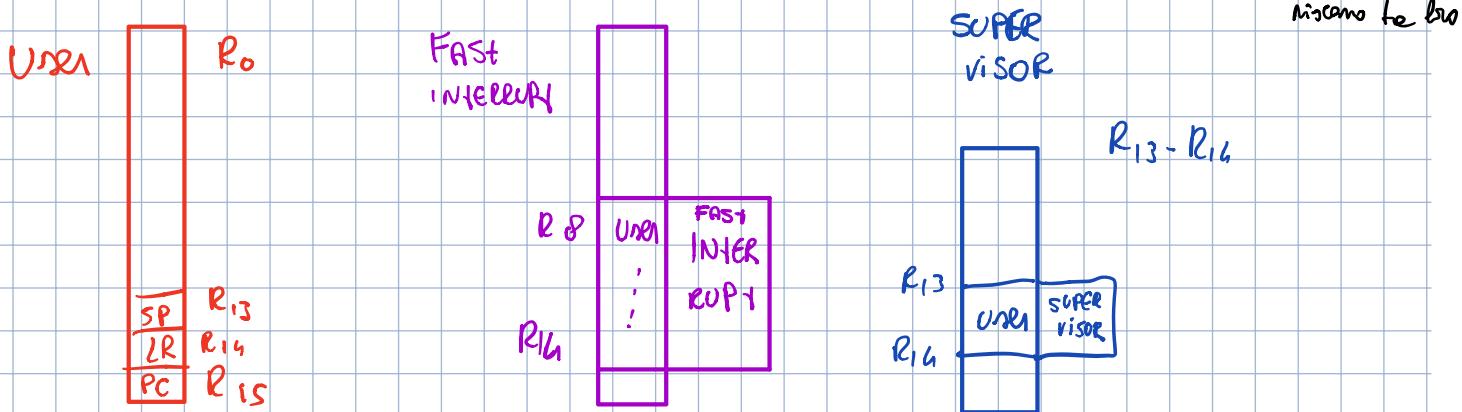
Area libera in modo che le due "Aree" non si sovrappongano a vicenda

heap dal minimo verso l'alto

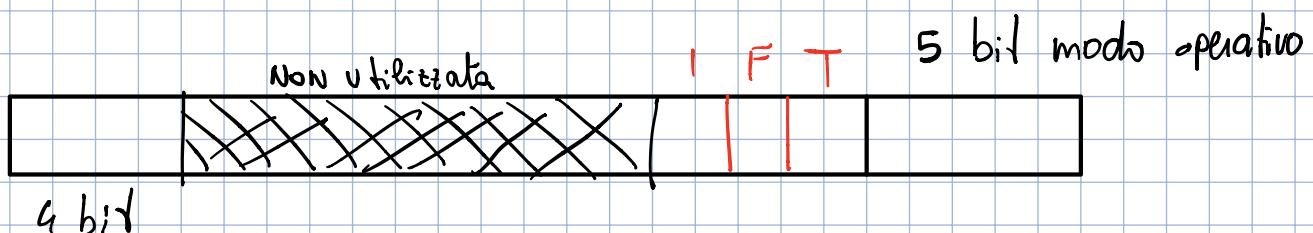
Modi Operativi | user → privilegi minimi
S.O.

ARM	USR	10000	X
	Fast interrupt	10001	
	interrupt	:	
	supervisor (svc)	:	
	abort	:	
	system	,	
	undefined	1111	

A seconda di quale modo op. utilizzo i reg. si duplicano e non intere =



PAROLE DI STATO DEL PROCESSORE (REGISTRO CPSR) → Registro appartenente



CPSR (current program status register)
flag
condizione

interruzione → I → nascherare le interruzioni

fast interrupt → F → se sono a 1 non vuole reutire cosa succ. nell' I/O

thumb state \rightarrow T \Rightarrow le istruzioni che vado a prendere sono a 16 bit invece che a 32 bit

Boot di un processore ARM

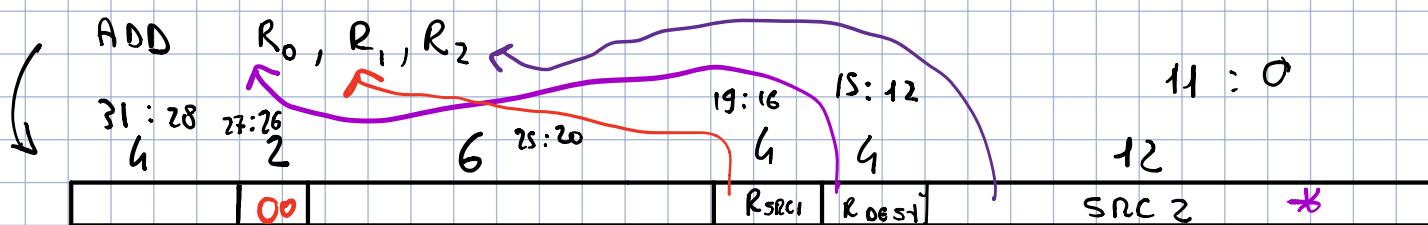
- 1) PC $\leftarrow \emptyset$
- 2) CPSR \leftarrow Supervised



codice
che mi serve
per caricare il
S.O.

Da istruzioni a linguaggio macchina

Operative



bit flag
 \downarrow
CONDITIONE
tipi di istruzione
00 operativa
01 memoria
10 salto

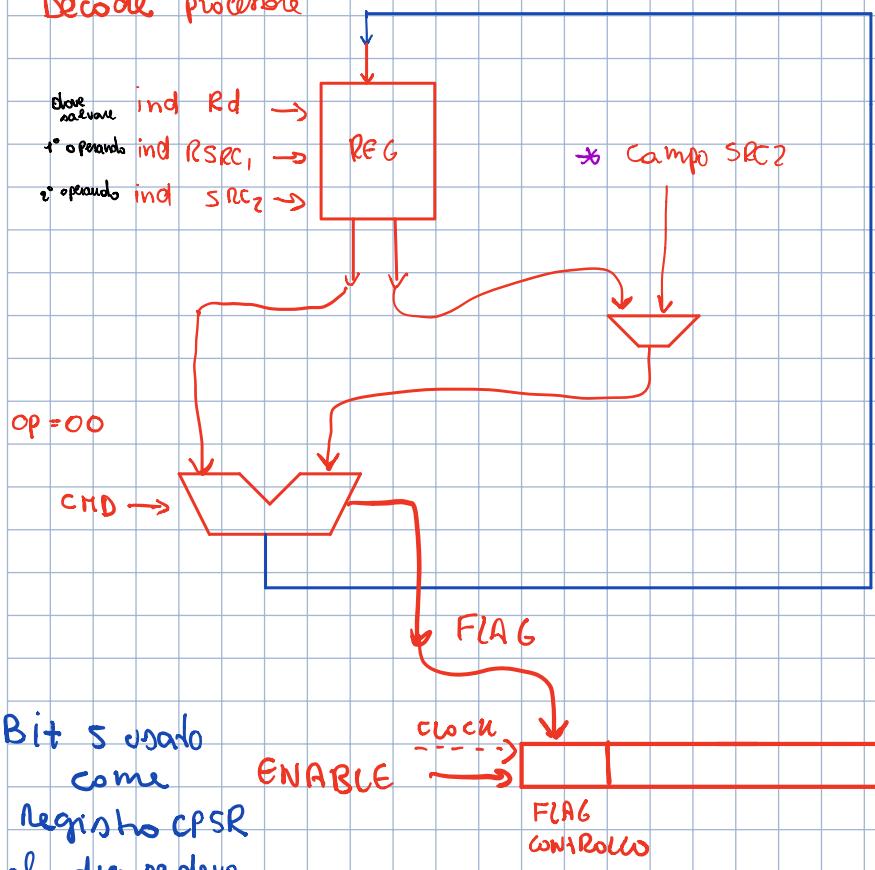
OP
 \downarrow
I | CMD | S
 \downarrow
Bit S se setta o no il flag
+
Bit I formato SRC2
+
CMD quale operazione eseguire
(ADD, SUB, ...)

valori leg possibili
 \downarrow
numero neg

A seconda del bit I
vengono usati in modo diverso

- 00 operativa
- 01 memoria
- 10 salto

Decode processore



Bit 5 usato
come ENABLE
Registri CPSR
che dicono se devo
scrivere o no

1)
quando $I = 1$
per una operazione con una costante (immediato)
di rotazione

shift dell'immediato

immediato	sh	0	Reg da shiftare
-----------	----	---	-----------------

tipo shift

2)
quando $I = 0$

shift del Reg

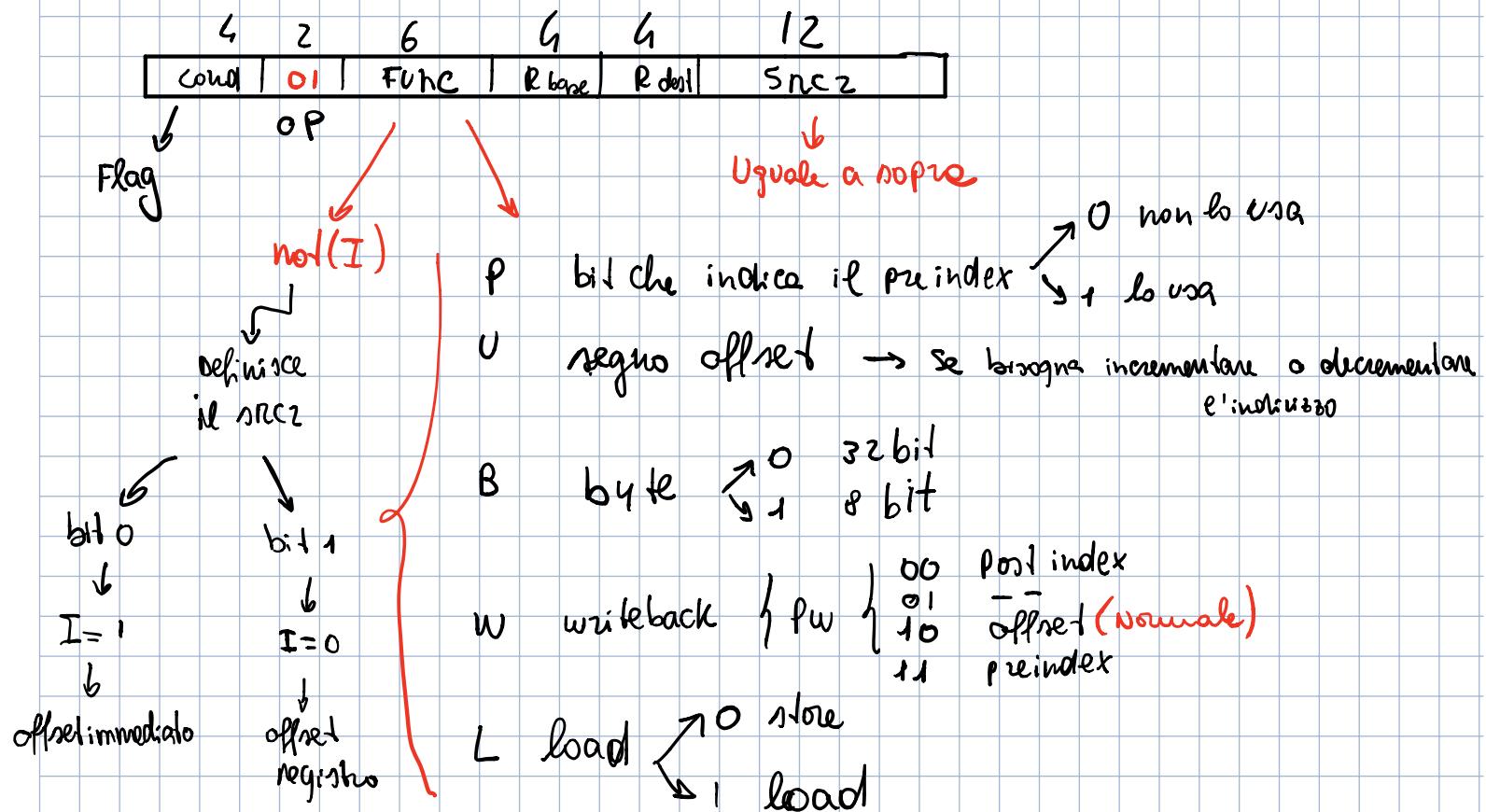
Rshift	0	sh	1	Reg da shiftare
--------	---	----	---	-----------------

R che indica di quanto shiftare

bit non utilizzato

num Reg

Memoria



Salto

4	2	2	24
---	---	---	----

COND OP FUNC OFFSET da aggiungere o togliere a PC+8

per andare all'etichetta

10 B
11 BL

Posso andare al max 2^{23} posizioni indietro e $2^{23}-1$ pos. in avanti

per stabilire nelle posizionanti posso usare il MOV PC, indirizzo perché il PC è un reg generale

Thumb

6
16 bit

bit risparmiati

3

1) registri \Rightarrow 8 registri \Rightarrow 3 bit per ric. quale neg è

3

2) uso dei registri

Utilizzano un reg sia come base che destinazione nel caso in cui reg destinazione e sorgente siano uguali

c'è il bit T nel CPSR

che specifica se stiamo

lavorando in THUMB o NOT

Per saltare da THUMB A NON

Si usano le istruzioni di salto

BX e BLX

3) Immmediati costi \Rightarrow più piccoli

4

4) No esec. condizionale per le operazioni

\uparrow

le lancio solo per i salti

1

5) I flag li setti sempre

funtions a

Per dire che uso v 16 bit in un programma assembler : .thumb_func

Fact con accumulator

```
fact(a,n) = if(n=1) return(a);  
           fact(axn, n-1);
```

main: bl aqvi

mov r3,r0

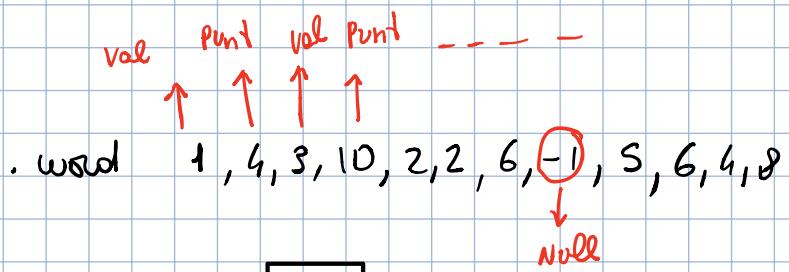
mov r1,#1

bl fact

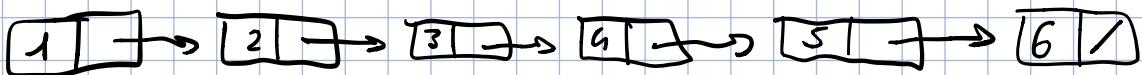
```
fact: cmp r0,#1  
     ble fine  
     mul r1,r1,r0  
     sub r0,r0,#1  
     b fact
```

```
fine: mov r0,r1  
      mov pc,lr
```

liste in ARM



0	1
1	4
2	3
3	10
4	2
5	2
6	6
7	-1
8	5
9	6
10	4
11	8



Ricerca in una lista

main: mov R0, #0 \rightarrow indirizzo al primo elemento
 ldr R1, =l \rightarrow indirizzo dell'array che rapp. la lista
 mov R2, #5 \rightarrow valore da cercare

ricerca: cmp R0, #1
 moveq R0, #0 { -1 } \rightarrow INDIRIZZO
 moveq PC, LR } PUNTATORI
 ldr R3 [R1, R0, LSL #2]

cmp R2, R3 } CONFRONTO CON IL VALORE CERCATO
 moveq R0, #1
 moveq PC, LR

add R0, R0, #1 \rightarrow Incremento R0 per avere l'indirizzo del puntatore

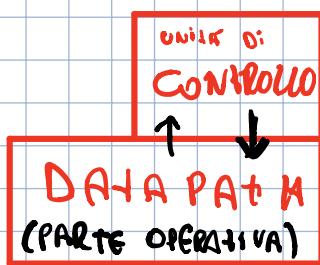
ldr R0, [R1, R0, LSL #2] \rightarrow Carico il valore dell'indirizzo puntato
 b ricerca \rightarrow Salto a ricerca

Microarchitettura

Processore → while (true) {

Fetch
decode
exec

?



DATA PATH: Implementa tutte le mosse che permettono di realizzare i calcoli che sono diretti all'esecuzione delle istruzioni (multiplexer, ALU, Reg, memoria ecc...)

Controllo: Attiva gli ingreni delle mosse del Data Path a seconda di cosa bisogna fare

1) Decide l'operazione da fare a seconda dell'istruzione / condizioni

Ricorre l'istruzione corrente
del datapath e dice come
va eseguita

(comunicazione da
DATAPATH A Controllo)

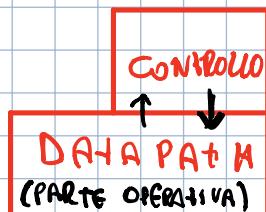
Segnali di controllo dispositivi che si possono comportare
in maniera diversa (es. ALU)

(come comportarsi)

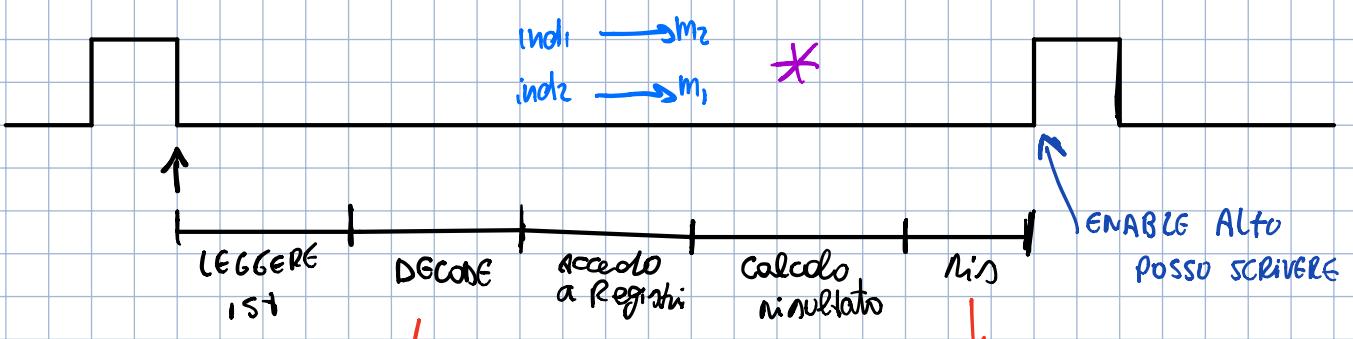
(comunicazione da
controllo a DATAPATH)

Scrivere o no ; moltiplicare delle OP nei Registri

Versioni:



1) Single Cycle : l'istruzioni vengono fatte tutte in un singolo ciclo di clock



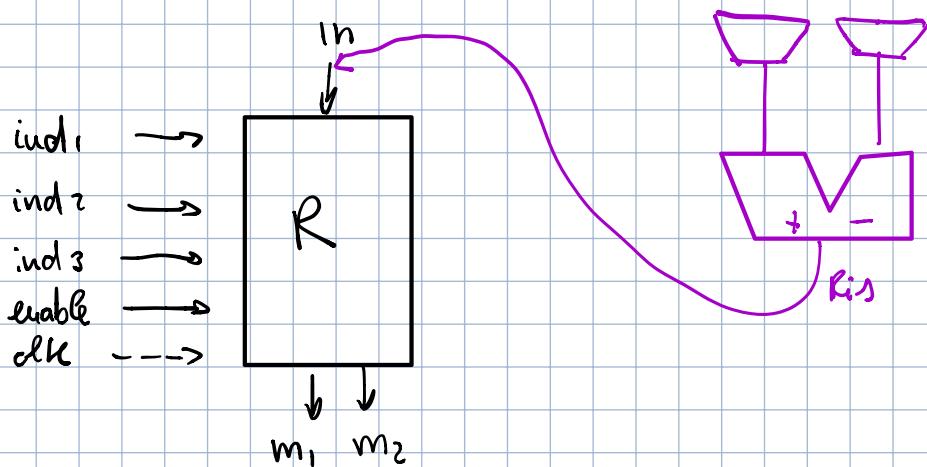
Problema:

2 memorie
separate

(ind e dati)

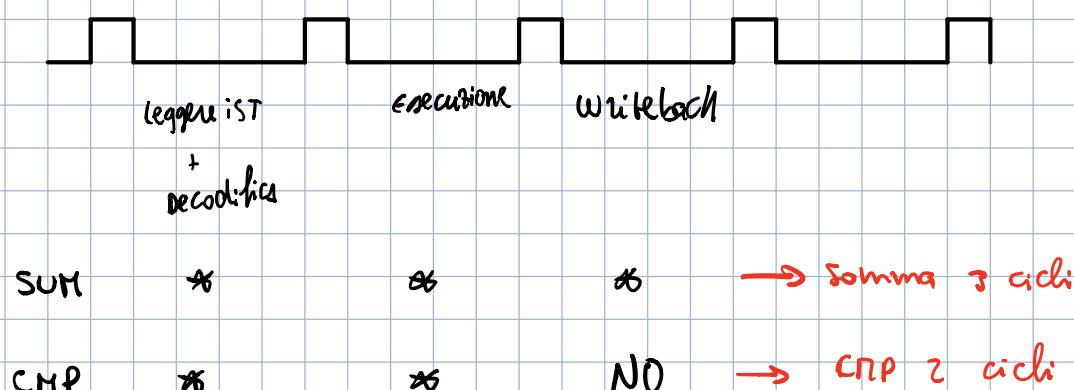
Ritardo l'ij
alla parte di
controllo per sapere
come operare

Devo essere pronti prima
che il clock ritorni alto



Single cycle : Devo capire quanto deve essere lungo il clock per poter fare tutte le operazioni (clock basato sull'istruzione più lenta)

2) Multicycle : 9 istruzioni in più cicli di clock \rightarrow 1 sola memoria



Ottimizzo lunghezza ciclo di clock a scapito di dover fare una intuizione in più cicli di clock

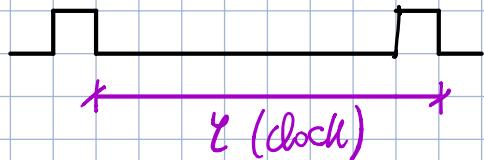
3) Pipeline : Più unità che svolgono una catena di montaggio

→ Dopo avere riempito la pipeline dovrei tirare fuori un risultato ogni ciclo di clock (ipoteticamente)

Prestazioni

Clock (τ)

↓
tempo che intercorre
tra 2 cicli di clock
pres nello stesso punto



CPI (clock per instruction) = cicli / istruzione

↓
Quanti cicli di clock servono
per eseguire un'istruzione

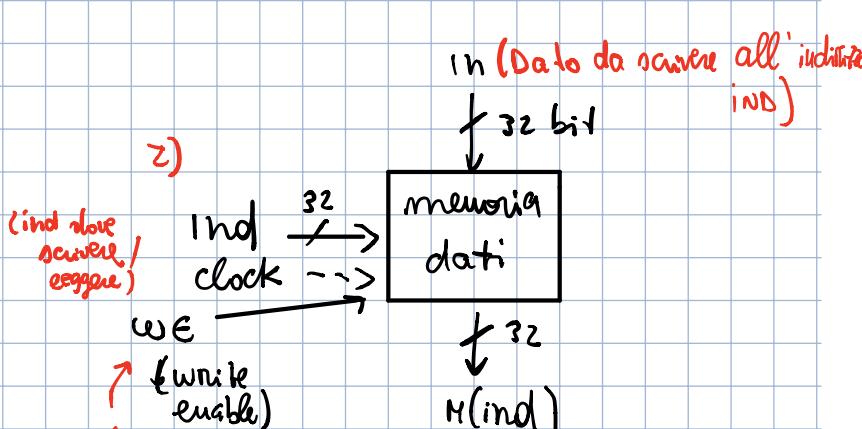
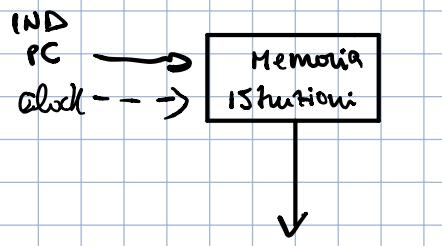
Esempio: Nel single cycle : CPI = 1

Tempo esecuzione prog ASI = N · CPI · τ

↓
numero
istruzioni ↓
numero
clock
x istruzione ↓
· tempo clock

COMPONENTI DI SISTEMA

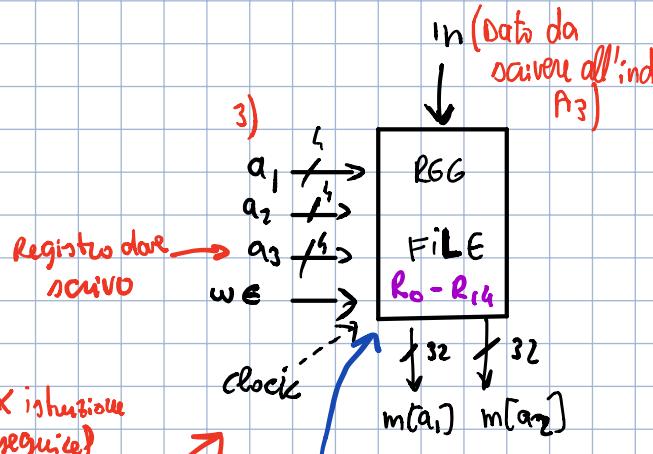
1)



$M(ind)$ (istruzione dato)

(se scrivere o no
il dato)

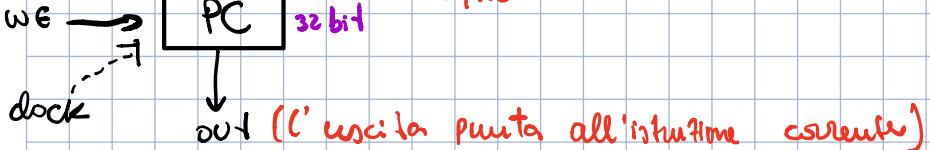
↓
Dato all'indirizzo passato



4)



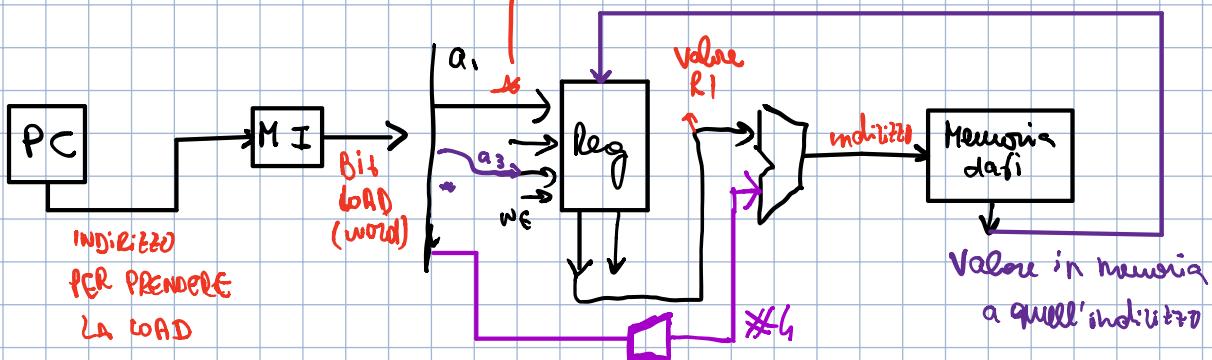
Li mettiamo nella parte
controllo



LDR con offset = immediato

LDR $R_0, [R_1, \#4]$

→ Indirizzo a_1 indica il reg R_1 . 4 bit per l'indirizzo vengono presi dalla istruzione. (4 bit)



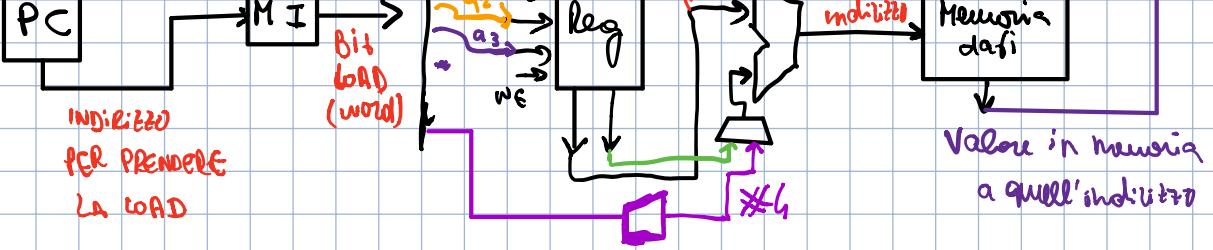
* Indirizzo R_0

(Registro dove
bisogna
scrivere)

LDR $R_0, [R_1, R_2]$

→ Indirizzo a_1 indica il reg R_1 . 4 bit per l'indirizzo vengono presi dalla istruzione. (4 bit)



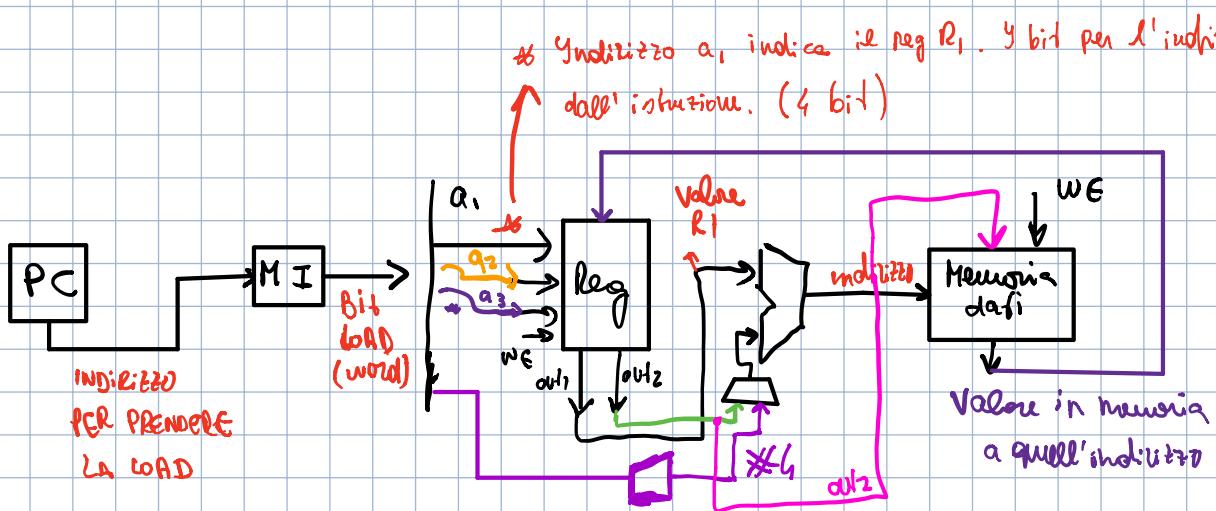


* Indirizzo R₀
(Registro dove
bisogna
scrivere)

Rende i bit dell'immediato
dalla parola e li trasforma
a 32 bit

* Indirizzo R₂
(register offset)

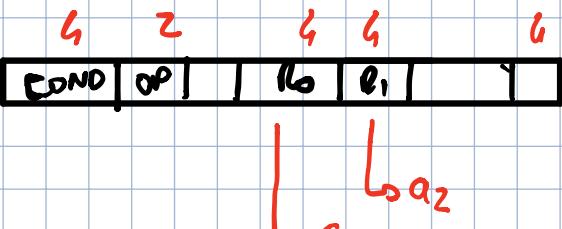
STR R₀, [R₁, #6]



* Indirizzo R₀
(Registro dove
bisogna
scrivere)

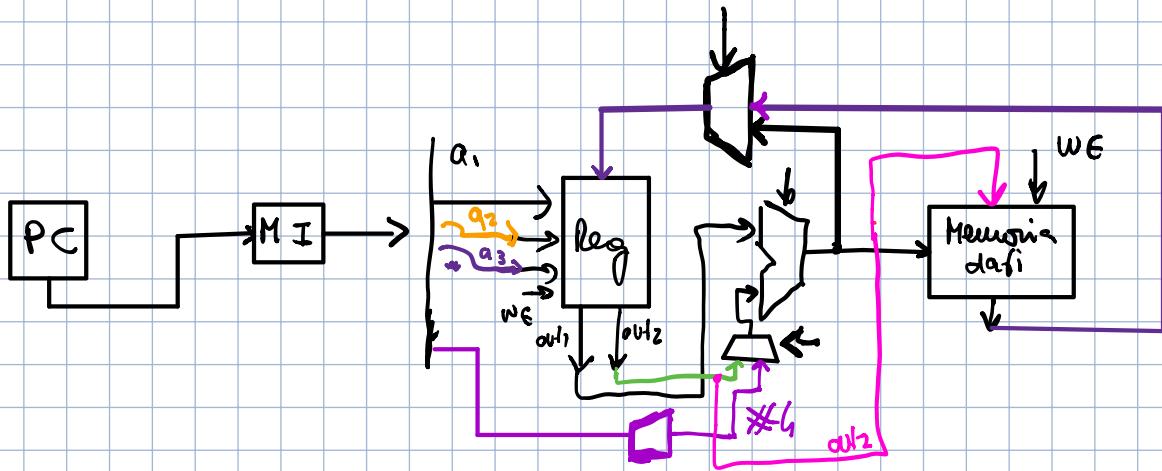
Rende i bit dell'immediato
dalla parola e li trasforma
a 32 bit

* Indirizzo da scrivere nella memoria



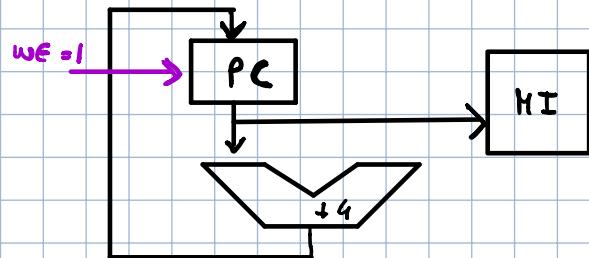
Rega sottostante rappresenta la parola
estesa da R₁

$\rightarrow a_1$
 $\text{ADD } R_0, R_1, \#5$



Aggiornare PC

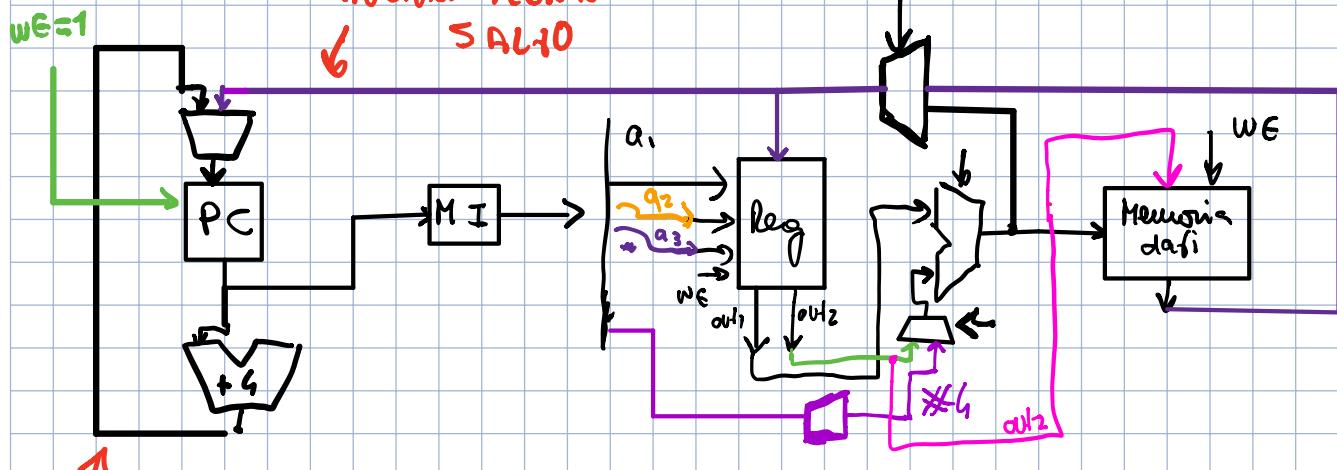
Metodo 1 (Nessun salto)



Metodo 2 (Salto a etichetta)



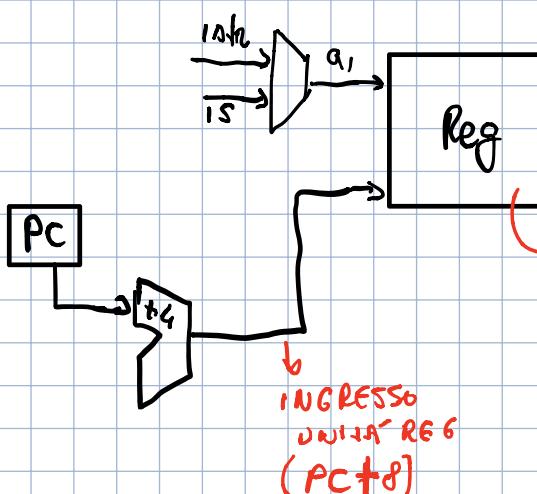
AGGIORNAMENTO PC
5 ALTO



AGGIORNAMENTO
PC CLASSICO

Copia aggiornata del

Ho un PC anche "dentro" il register file e quindi si aggiorna così. Il PC dentro il reg file è $PC+8$



Ho i reg da 0 a 16, il IS non ce l'ho perche prendo il pc esterno e gli metto +8 quando glielo passo al reg file con un ALU

ADD R0, R1, R2

PROCESSORE A CICLO SINGOLO COMPLETO

Flag salvati nello control unit che vengono controllati con il flag generato dalla ALU

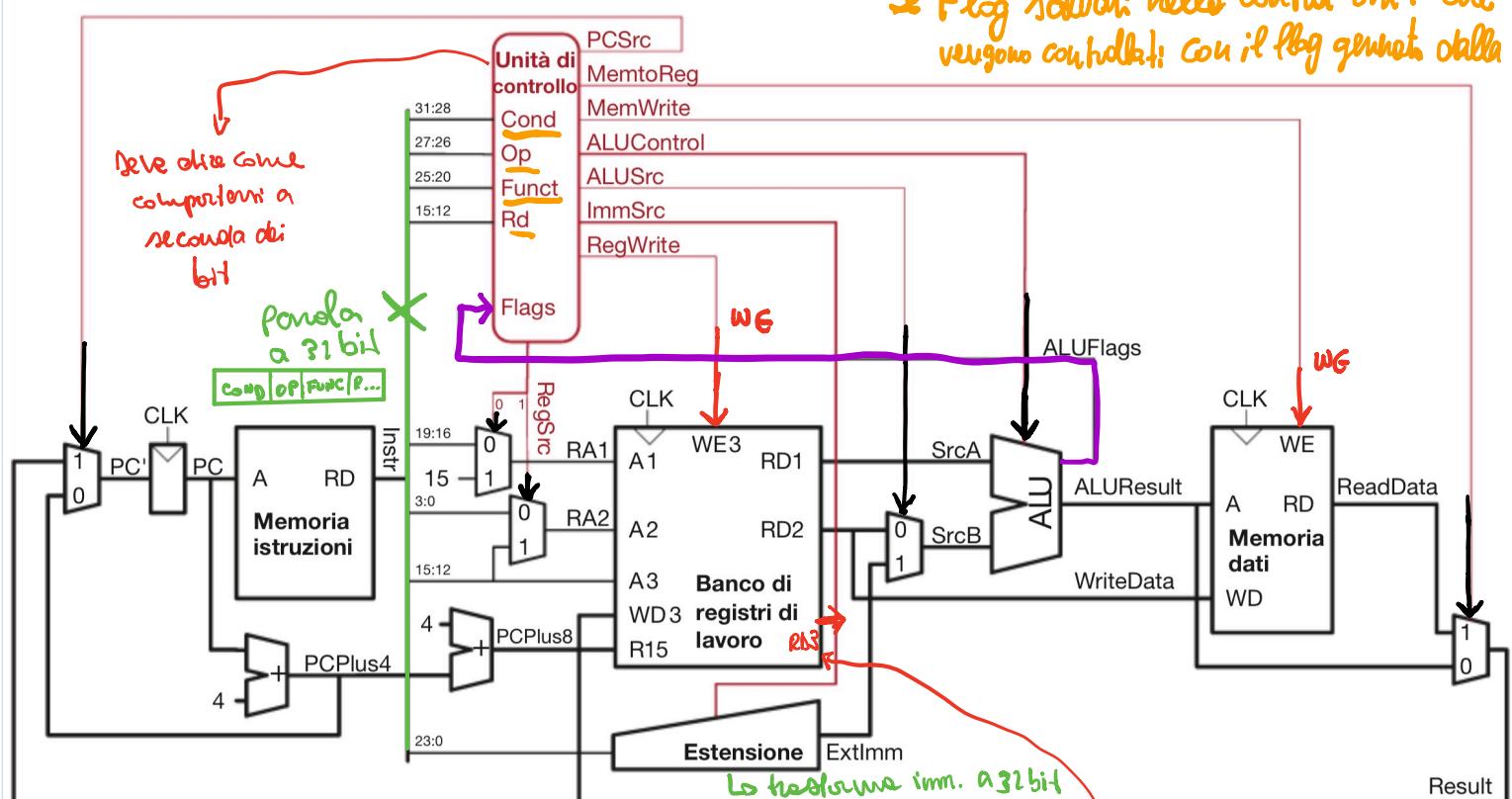


Figura 7.13 Processore a ciclo singolo completo.

Resiste RD3 che serve per fare la store con 3 registri

Fatti dal control unit

- 6 controlli che cambiano il tipo di calcolo fatto
- 2 segnali di controllo che determinano effetto dell'esecuzione sullo stato

1 Flag di controllo mandati dalla alu

Problema

procedere a ciclo singolo: Devo considerare ogni tipo di ritardo anche per le istruzioni che non usano tutte le parti. Devono aspettare comunque il ciclo di clock predefinito



Ciclo di clock lungo



Più tempo

ci sono 2 memorie una per le istruzioni e una per i dati.

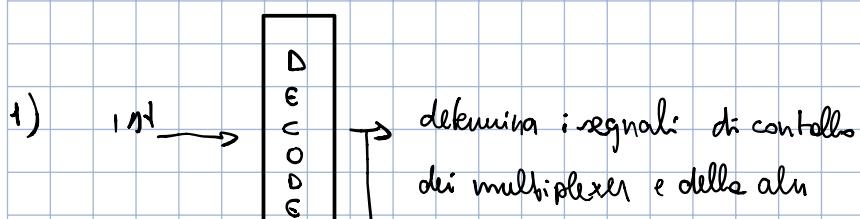
\Rightarrow Dovrebbe essere una sola

CLK procedere single cycle ≥ 800 psec

Ponte controlli

Nel proc. single cycle è una rete combinatoria!!!

È divisa in 2 pezzi

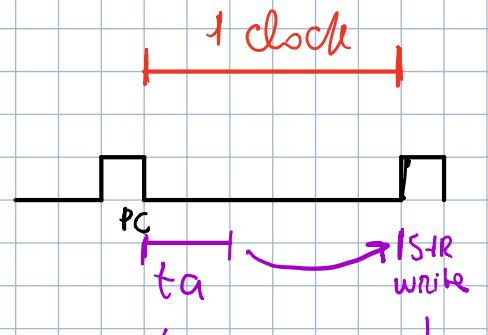
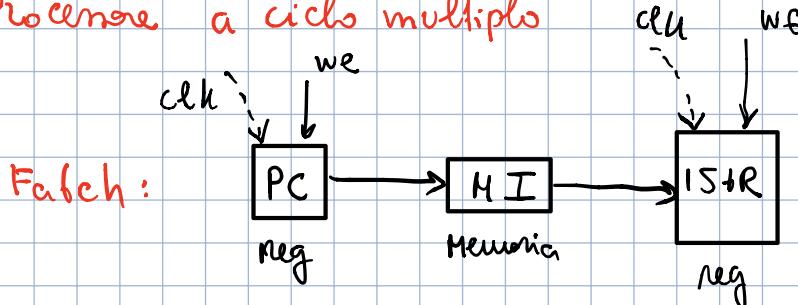


(calcolati dalla ALU
o mantenuti
in un registro)

CPSR

contiene CPSR

Procedere a ciclo multiplo

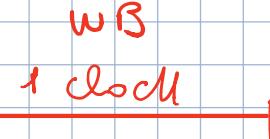
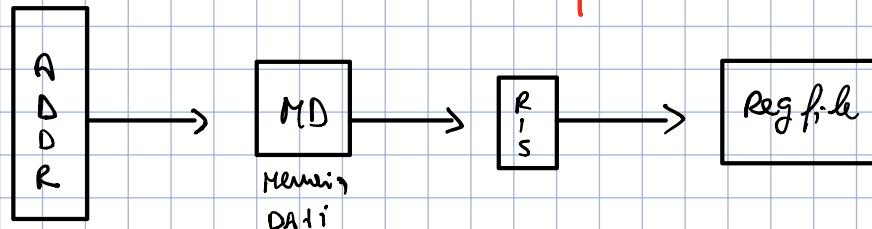
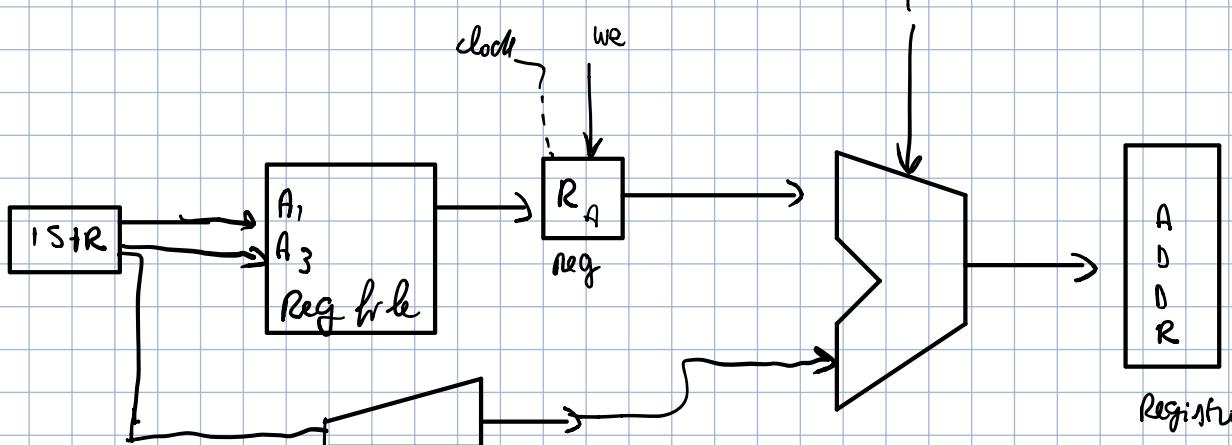


Decode: - LDR R0,[R1, #off]



Alex

1 clock



Univore:

l₁ ①

l₂ ②

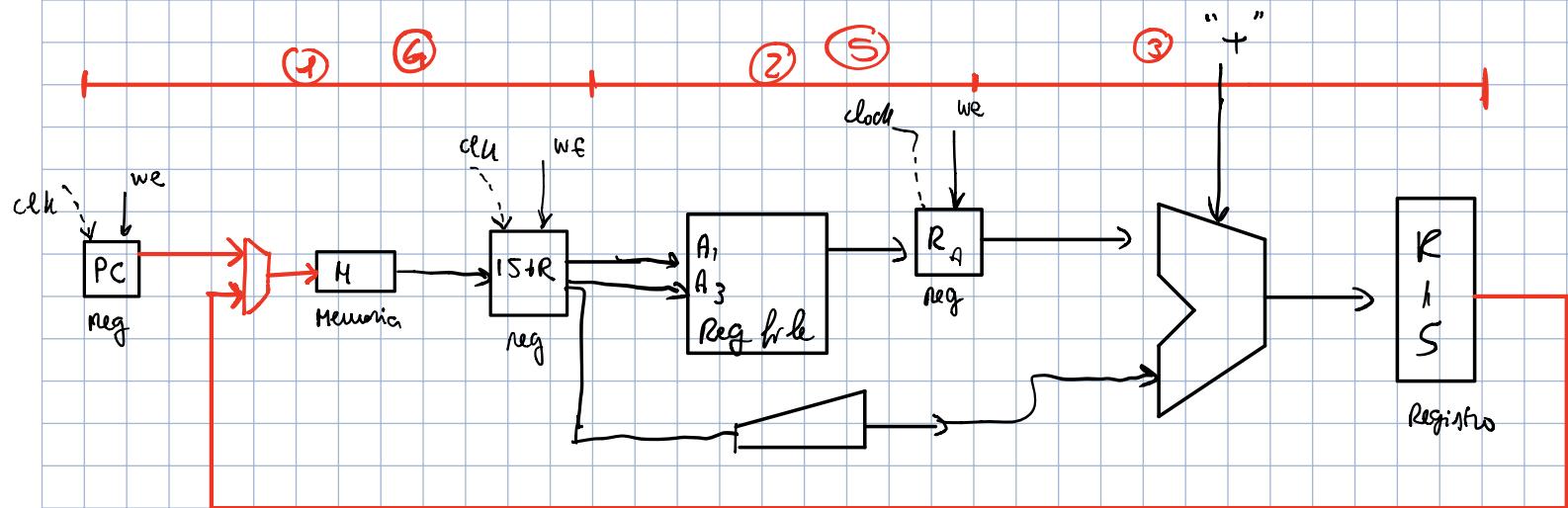
l₃ ③

l₄ ④

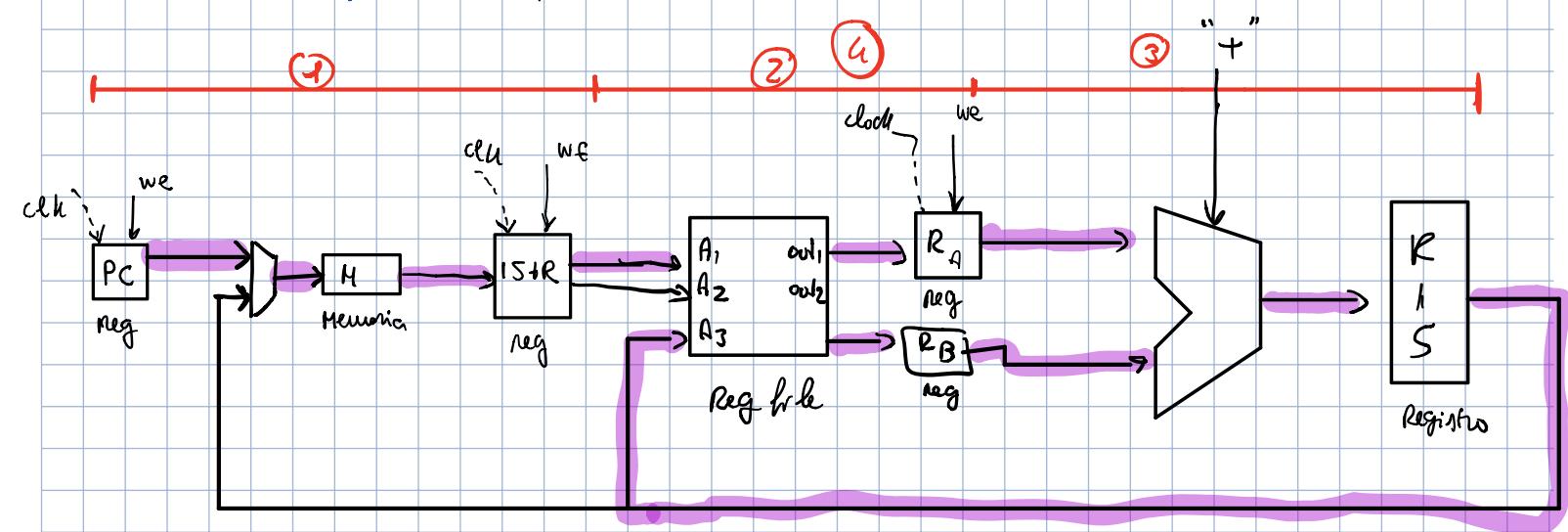
l₅ ⑤



ATTENZIONE: Posso rispettare il tempo facendo la memoria unica
Però mi è complice l'unità di controllo perché diversa da automatica.



- Esecuzione Add R_0, R_1, R_2



Uno un ciclo di clock in meno



Osservazione: Ci vuole un γ in meno per eseguire le operazioni rispetto alle Load e Store

$$\text{OP. SALVO} = 3 \gamma \quad \text{OP. OPERATIVI} = 4 \gamma \quad \text{OP. LOAD/STORE} = 5 \gamma$$

Processore multi ciclo completo

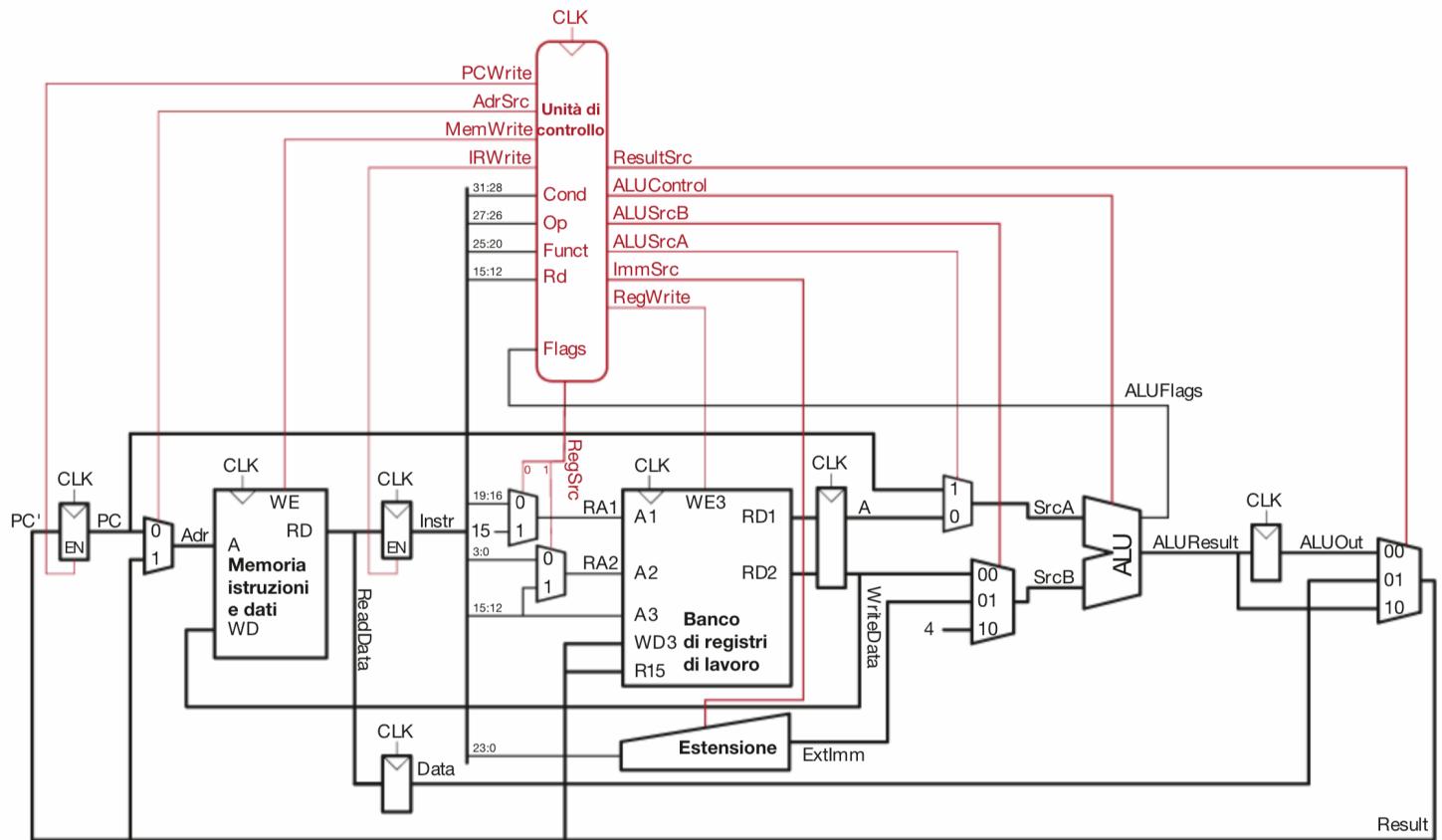
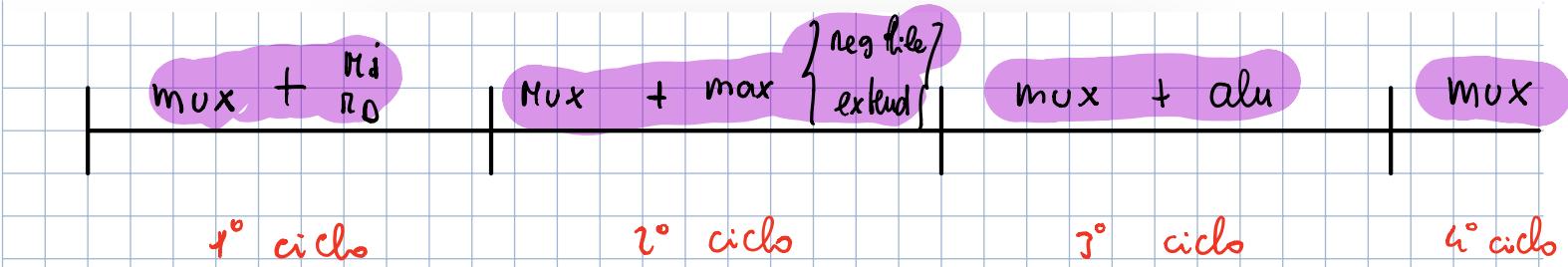


Figura 7.30 Processore multi ciclo completo.



Un viole sono i tempi richiesti per eseguire il singolo insieme d'ist. in quella park

gl massimo tra {ciclo 1, 2, 3, 4} = Ciclo di clock

$$\downarrow \\ \text{min CLK} \geq 340 \text{ psec}$$

Esempio: LDR R0, [R1, #offset]

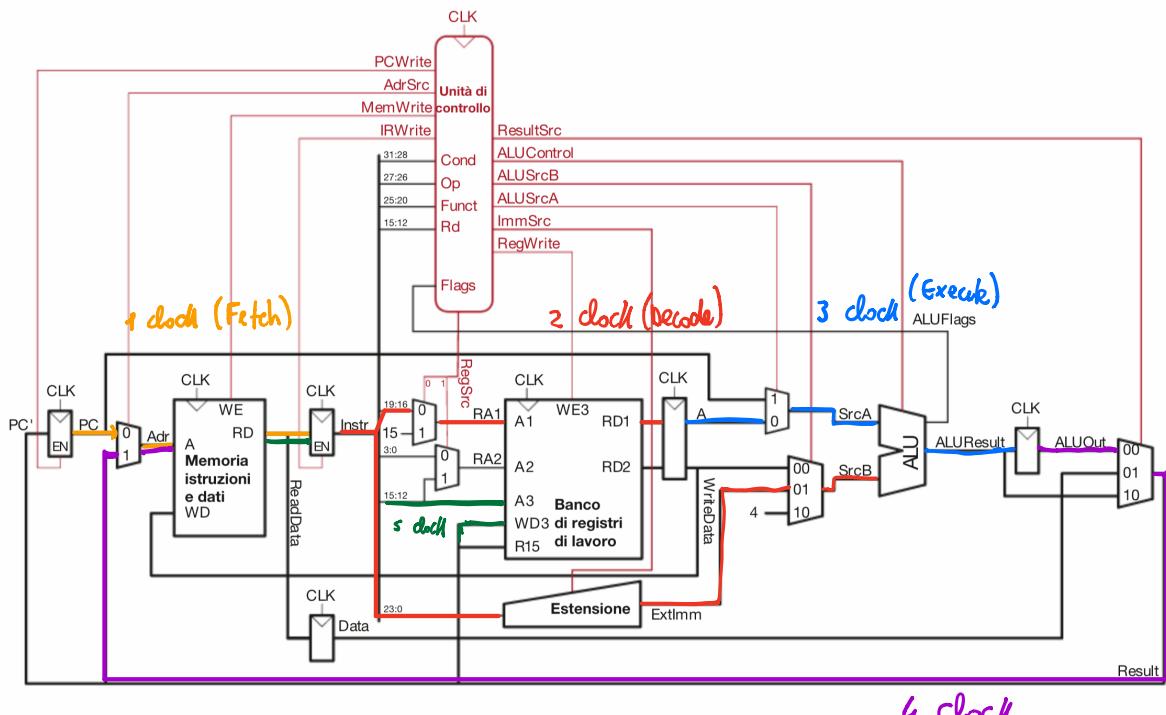


Figura 7.30 Processore multi ciclo completo.

esempio: ADD R0, R1, #val

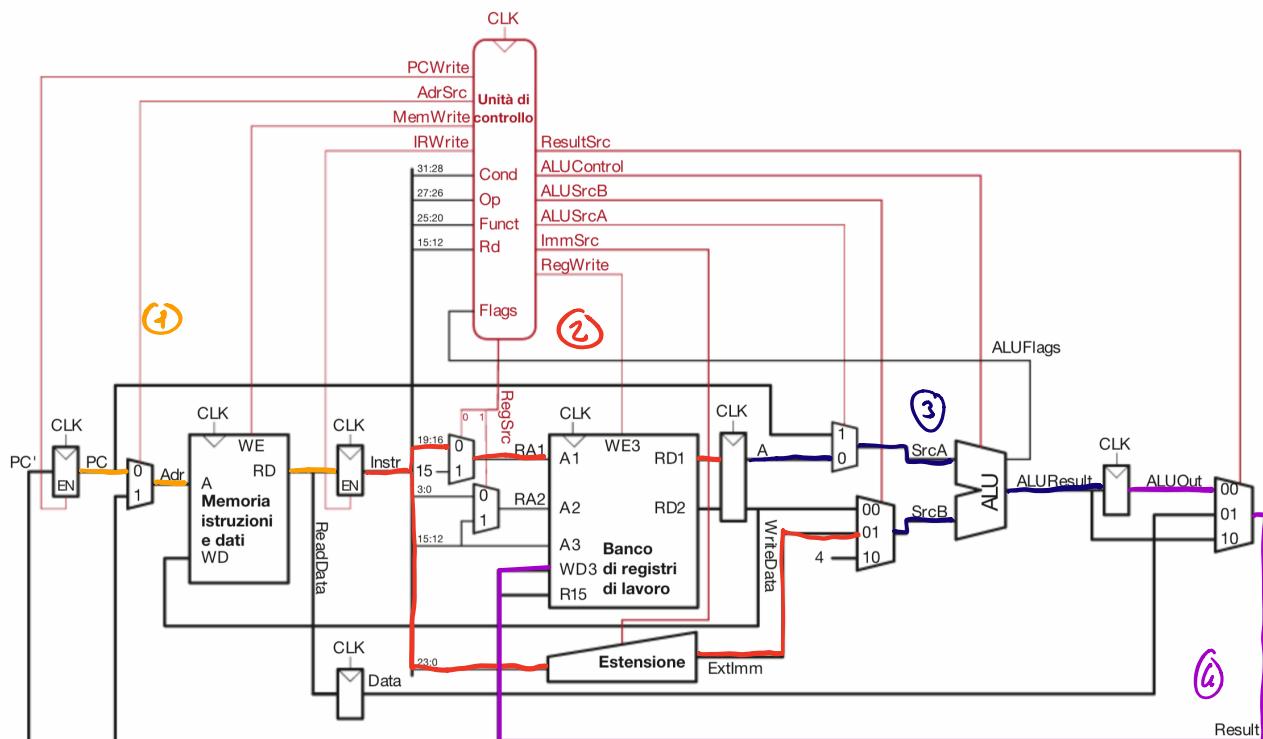


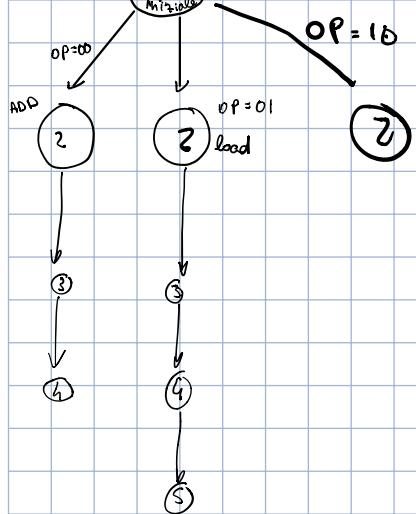
Figura 7.30 Processore multi ciclo completo.

Ponte controllo

Stato

Usate segnali di controllo x il fetch

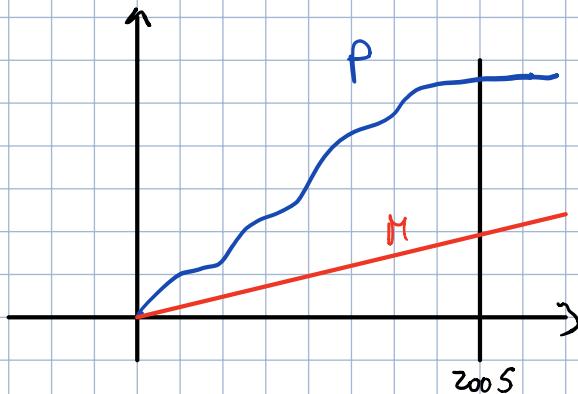
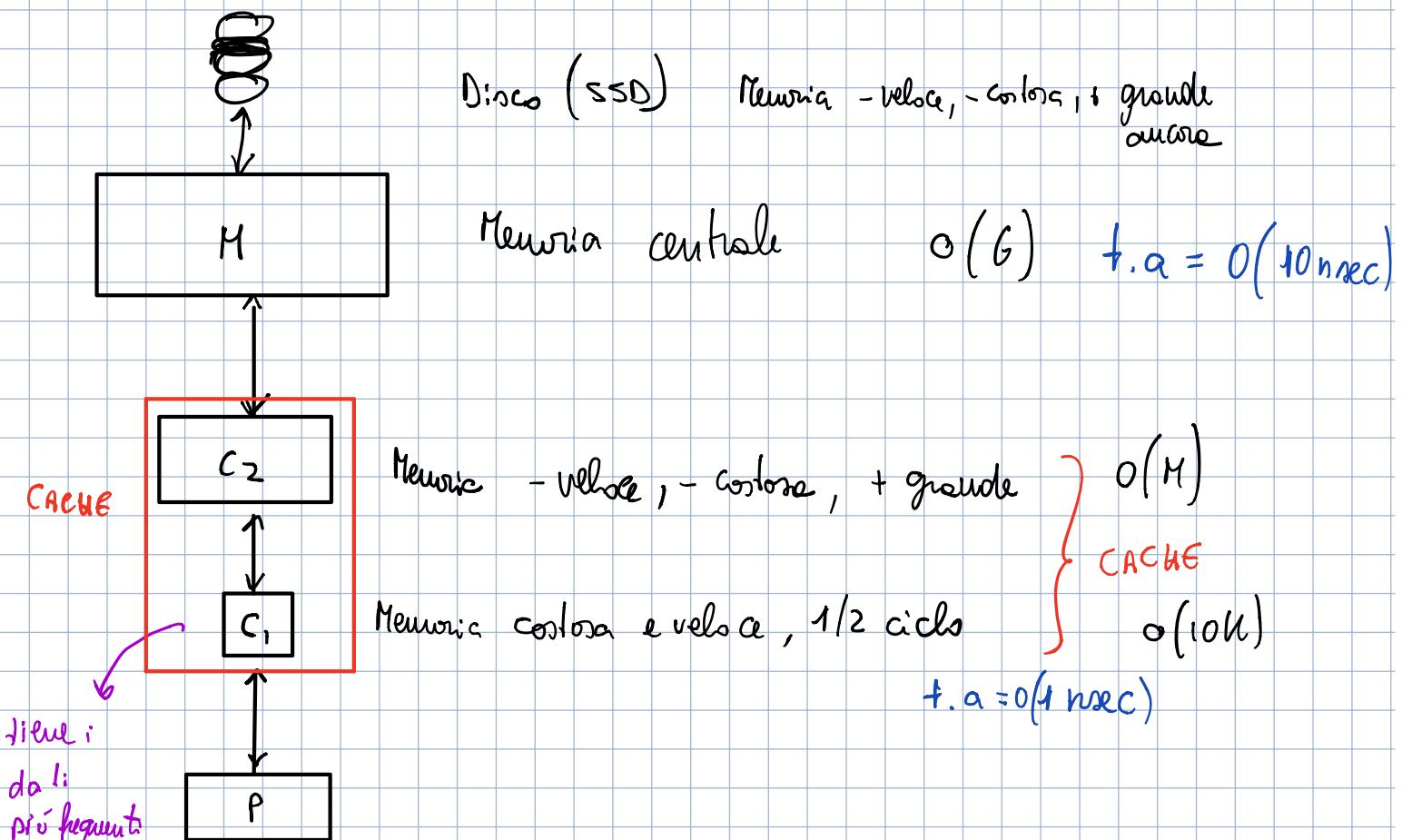
istruzioni diverse vengono eseguite passando da
seguente diverse di stati della pente di controllo



→ Diventa un automa perché si deve ricordare in che fase sono.

Più esempio se sono in fase dove leggo l'ist. dalla mem. o il dato

Sottosistema memoria



Il processore ha uno smesso di migliorare di velocità mentre le mem. aumentano
più

Hit \rightarrow cercare una locazione di memoria in cache e lavorare

Miss \rightarrow se non trova e non lavorare

$$P_H = 1 - P_M \Rightarrow \text{probabilità Hit}$$

$$P_M = 1 - P_H \Rightarrow \text{probabilità Miss} > \% \text{ degli accessi in memoria hit o miss}$$

$$\text{t.a dal processore} = t.a = P_H \cdot \text{t.a}_{\text{CACHE}} + P_M \cdot \text{t.a}_{\text{Memoria}}$$

Località Spaziale

Se sto lavorando su una

locazione di memoria x è molto
probabile che in poco tempo vada
a toccare locazioni che stanno
intorno a x



Se a tempo t accedo a x è prob

che a tempo $t+...$ acceda a $x+1$

$x+2$

$x-1$

:

Località temporale

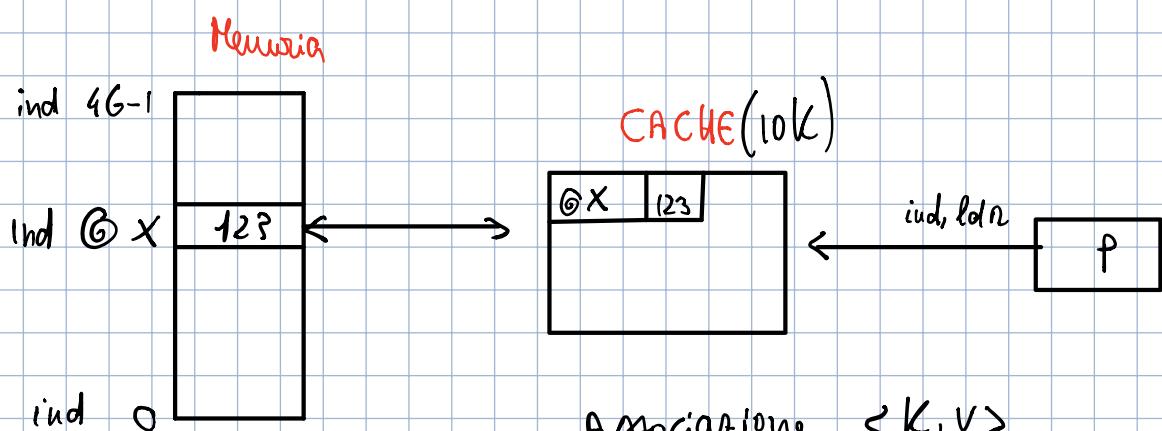
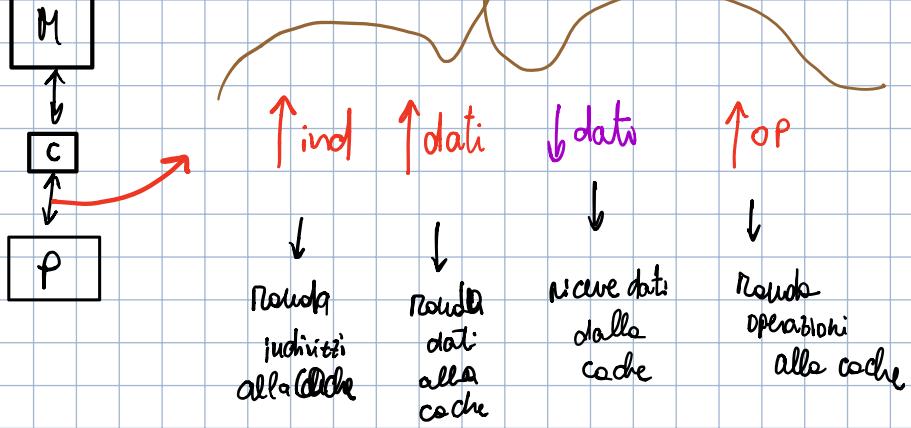
Se a tempo t ho acceduto a x

è molto probabile che a tempo
 $t+...$ mi accedo a x

Working Set:insieme di dati e codice necessari per il funzionamento di un
programma in un certo lasso di tempo

N.B.: Il processore ignora che sia una cache con cui comunica

INTERFACCIA PROCESSORE

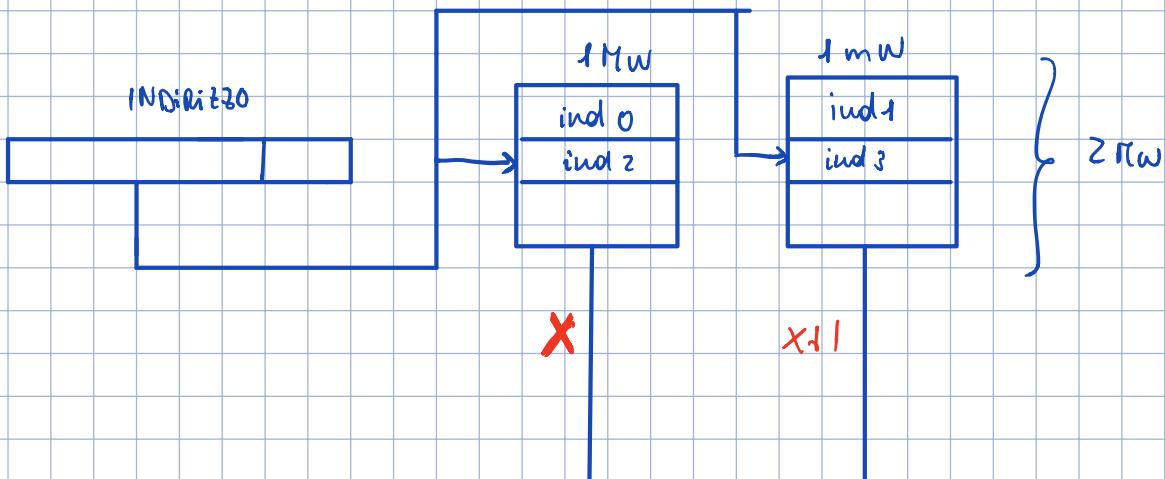


Associazione $\langle K, V \rangle$

indirizzo \downarrow
contenuto location
di indirizzo K

LOCALITÀ SPAZIALE

Per sfruttare il principio di località posso ogni volta che accedo a X , caricare
nella cache un intorno di X



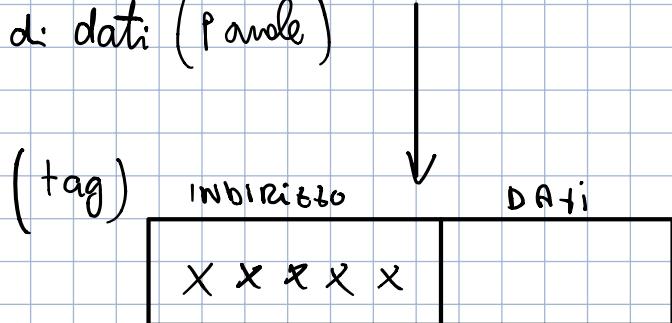


Locu^{ri} temporale: Nella cache cercherò di tenere le cose che ho utilizzato ultimamente (i più recenti)

Politica LRU (last recently used) = Rimuovo il dato usato meno recentemente

CACHE

Una mem. CACHE è composta da un insieme di SET ciascuno dei quali contiene uno o più blocchi di dati (Parole)



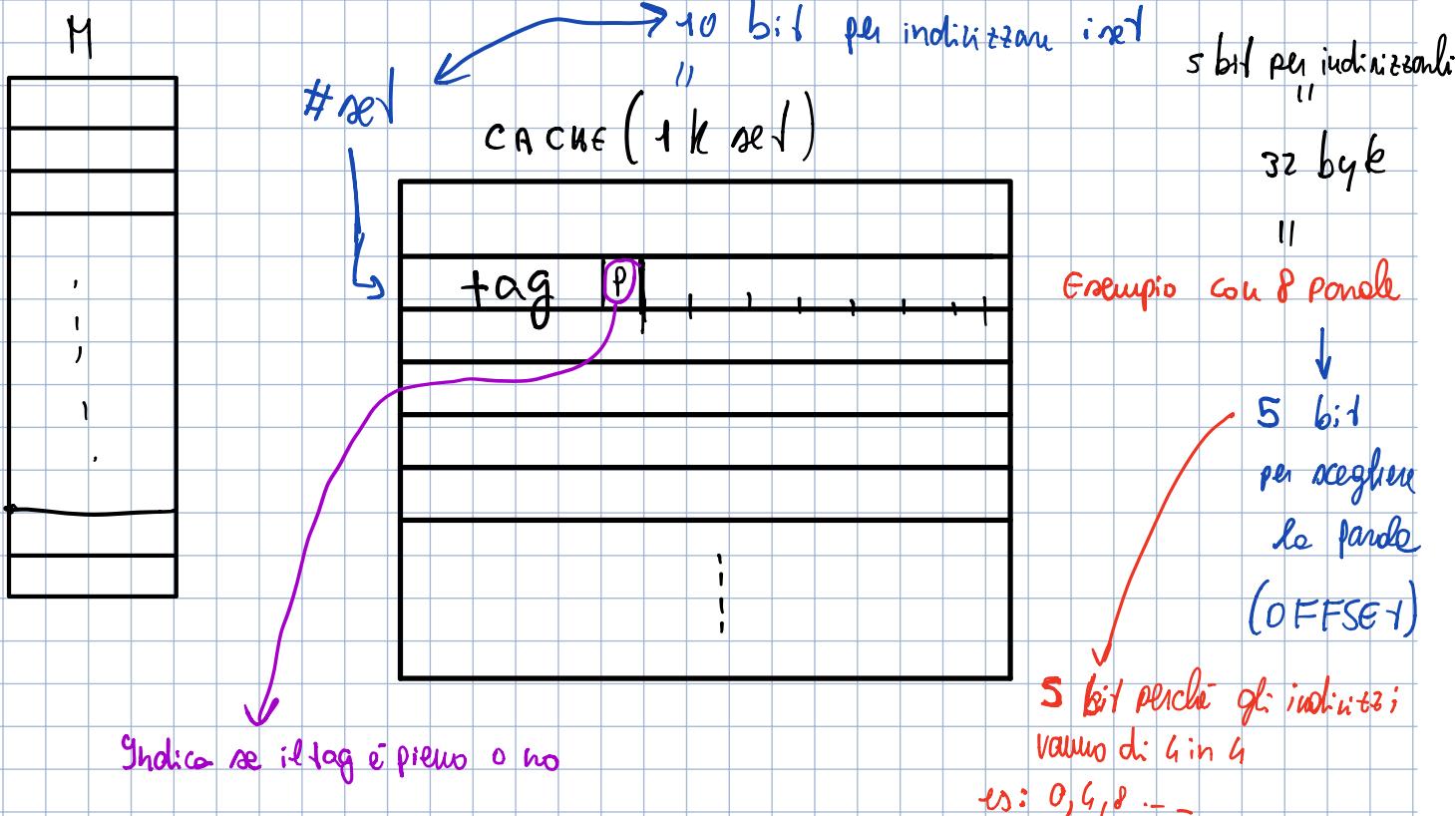
Mappatura: Relazione tra l'indirizzo di un dato in memoria principale e locazione di tale dato in cache.

OSS: Ogni indirizzo di memoria viene mappato in un set della cache, alcuni dei bit dell'indirizzo sono utilizzati per determinare in quale set della cache si trova il dato

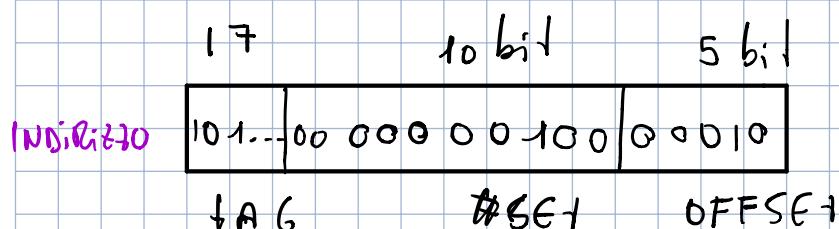
Tipi di CACHE:

1) Mappamento diretto





La mappatura è diretta \Rightarrow Ad ogni blocco della memoria corrisponde un solo set delle cache

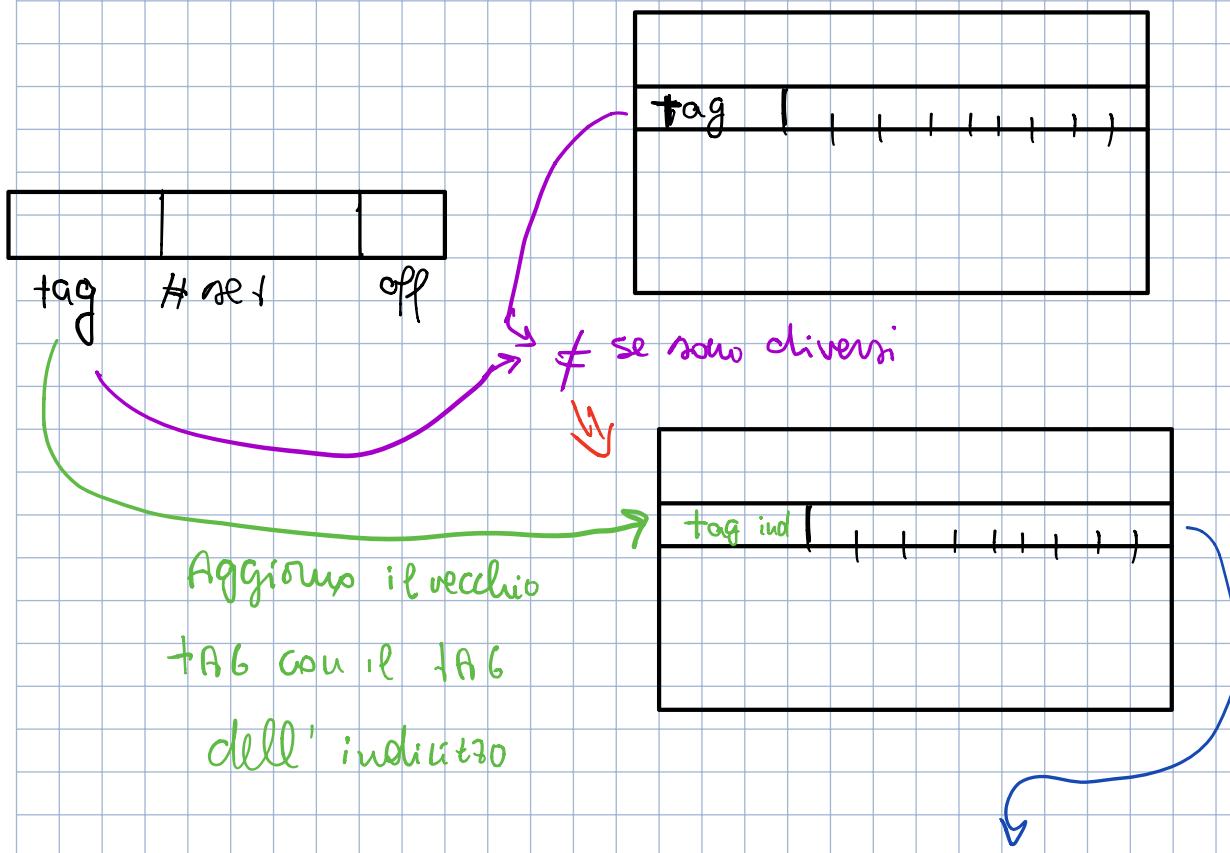


Vado all'indirizzo #SET delle cache, controllo se tag dell'indirizzo e delle cache corrispondono, se corrispondono vuol dire che trovo il dato cercato all'offset dell'indirizzo.

Se i tag non corrispondono vuol dire che il dato cercato non lo trovo da nessun'altra parte perché l'inizializzamento è di tipo

Affenzione

se ottengo un miss tra i due tag dell'indirizzo e cache, butto via tutto ciò che ho nella cache a indirizzo \neq nel dell'indirizzo e ricarico i valori cercandoli in memoria aggiornando il tag con i tag dell'indirizzo



IND

(esempio con 2 parole in ogni set)

MEMORIA

Dove prendere i dati per il set

1) +A 6 | # SET | 000
2) +A 6 | # SET | 001
3) +A 6 | # SET | 010
⋮
⋮
⋮
⋮
8) +A 6 | # SET | 111

Se accedo a un dato che c'è dentro la cache: $P_H \cdot t.a_{cache}$

Se invece ho avuto un miss e devo accedervi alla mem cache: $P_H \cdot t.a_{cache} + P_R \cdot t.a_{mem}$.

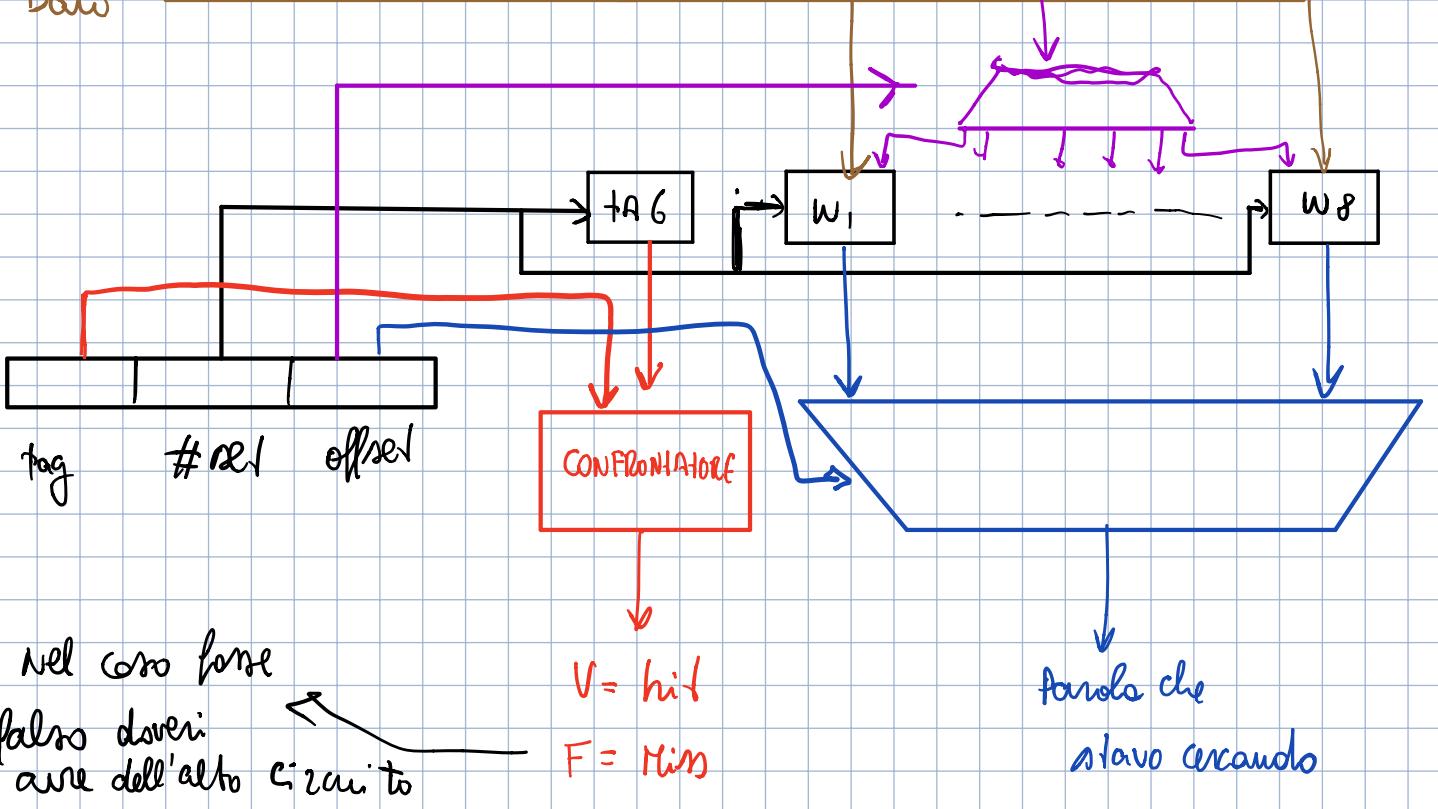
Come è fatto un accesso alla cache con i componenti logici

Scrittura

WG

8 parole (moduli di memoria)

Data



N.B: - Rosso e viola \Rightarrow Scrivere dato

Il demultiplexer con WE dice dove scrivere

- Rosso e blu \Rightarrow lettura dato

Confronto i tag e tramite l'offset scelgo le parole da leggere

Problema iholizzamento diretto:

$$\text{Se } \# \text{Set}_{\text{load}} = \# \text{Set}_{\text{store}}$$

Ogni volta riconico i valori

e butto via quelli vecchi!

Fenomeno chiamato **THRASHING**

\hookrightarrow se alloco in ordine reg. ho meno poss. di avere il thrashing.

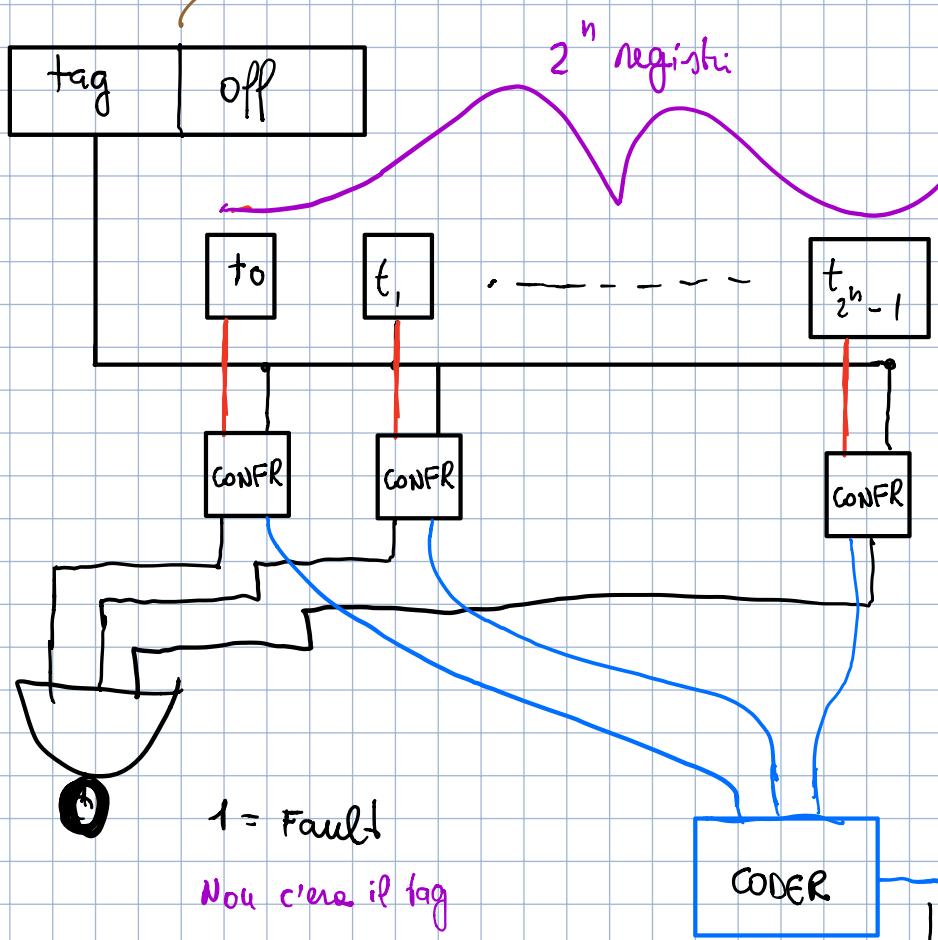
Due iholizzamenti che si mappano sul medesimo set causano nemmeno in conflitto

INDIRIZZAMENTO CONCGIURANTE ASSOCIATIVO (Utilizza politica LRU)

+DG	P	PAROLE
+AC	P	PAROLE
+AG	P	PAROLE
		>
.		/
:		/
,		,

$n-1$
2
set

Non ho il set, confronto con ogni tag nella cache



NOR

1 = Fault

Non c'era il tag

(Vale solo se i confronti danno 1
se gli elementi sono diversi)

Quali linee prendere

TROPPO COSTOSO!

n = numero set

2) Indirizzamento

SET - ASSOCIAZIO

a

n-vie

per ogni
insieme

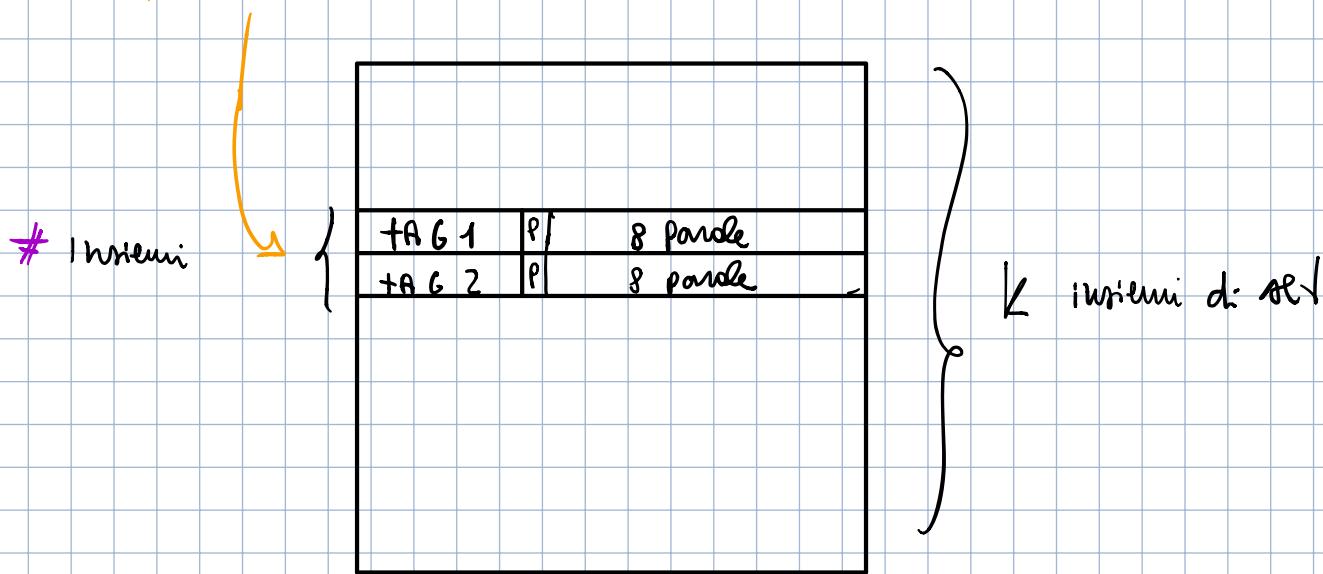


Uso il modo
diretto per
individuare #SET



Devo nel set ho più opzioni:
che cerco in modo associativo

In questo caso $n=2$ (insiemi composti da 2 set)



INDIRIZZO

TAG	# insiem	offset
-----	----------	--------



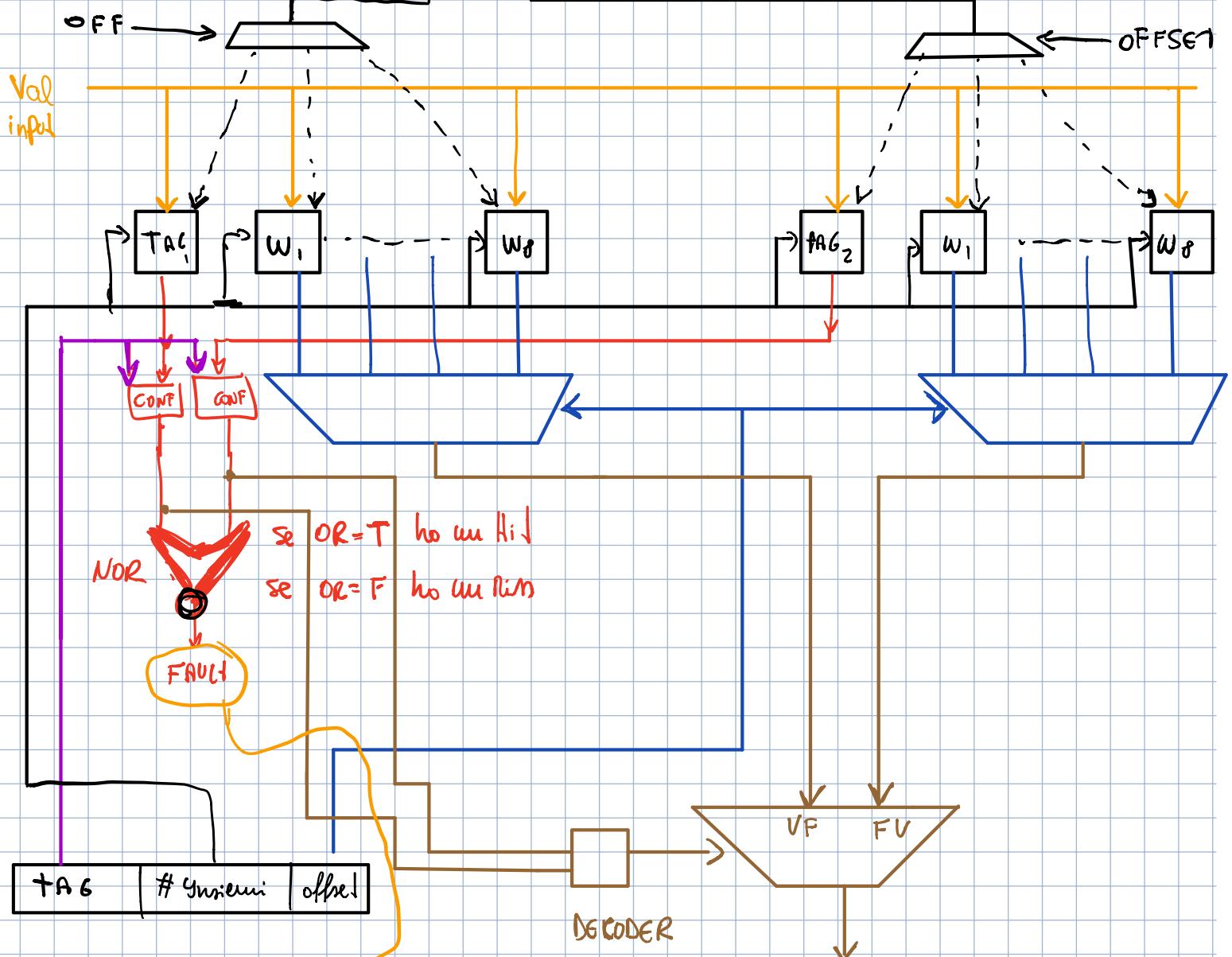
CONFRONTO IL
TAG CON TUTTI
QUELLI DELL'INSIEME
A INDIRIZZO #INSIEMI

Vedo a prendere
il #insiemi
al posto del #tag

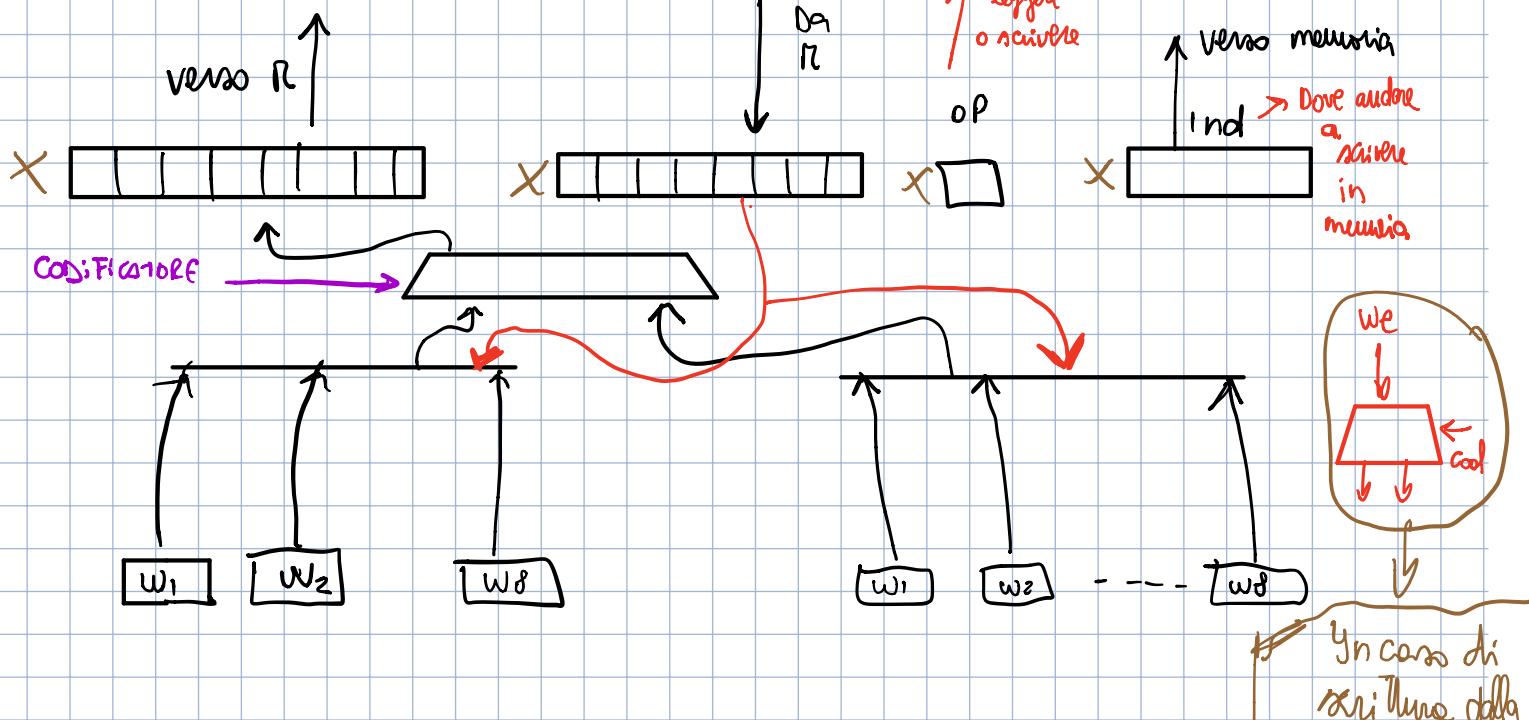
Accesso / Scrittura alle cache con componenti logici

WE

codifica offset

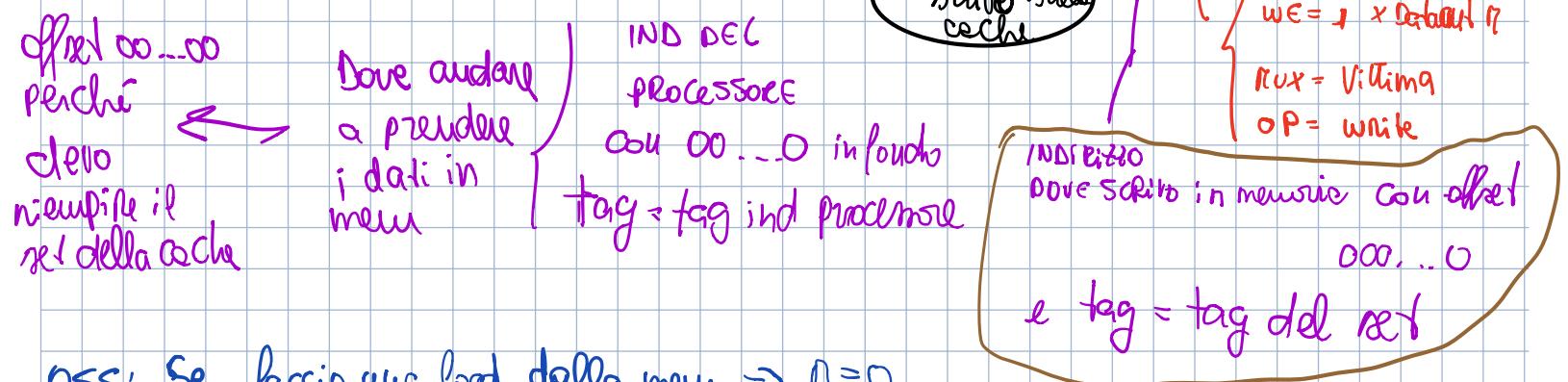
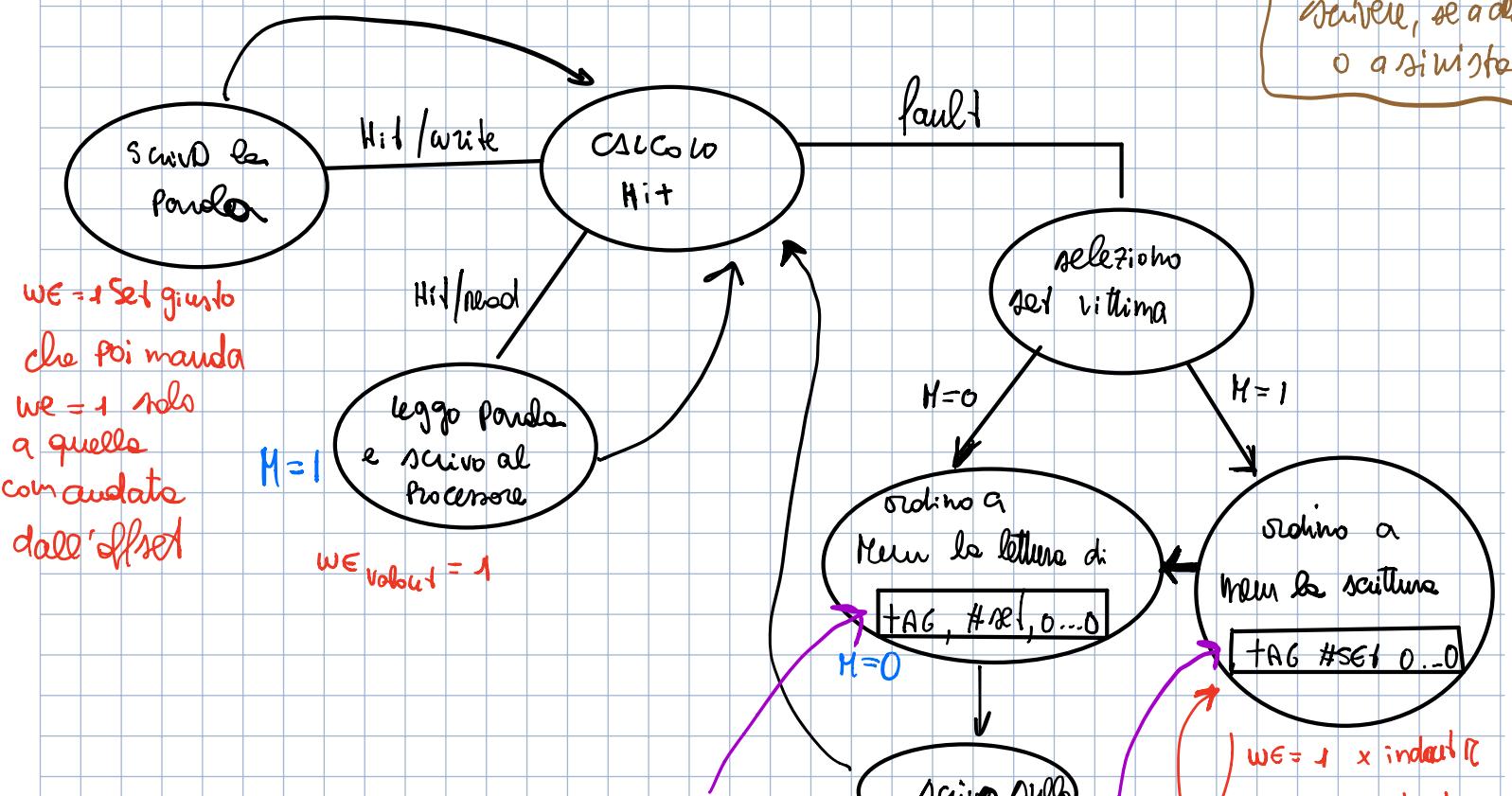


Caso Fault! $X = \text{Registori}$



Ponte di controllo cache associativa a 2 vie (Automa)

menu alle cache
decido dove
scrivere, se a destra
o a sinistra



OSS: Se faccio una load dalla mem $\rightarrow H=0$

Se faccio una store $\rightarrow H=1$
(modifico quello che ho nello cache)

Come capire quale set buttare via in caso di miss:

Politica LRU (Last recently used) = set che non ho usato per tanto tempo

Per implementazione si fa un'approssimazione (selezione della pag. vittima)

tag	P	U	poodle
-----	---	---	--------

$U=1$ linea utilizzata

$U=0$ linea non è stata utilizzata

tutti gli

refresho U ogni t_0

tempo in modo che tutti

gli U siano mass 1 dopo

un po' di tempo

2° problema

Devo modificare i valori in memoria quando modifico i dati nella cache

Soluzione 1 (WRITE BACK) \Rightarrow Accede in mem. solo quando i dati nella cache modificati vanno per essere sovrascritti

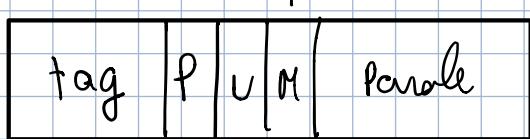
2 tempi di accesso ogni

miss/fault

alla mem. superiore

$M=0$ se non c'è stata modifica, quindi è uguale a quelle in memoria

$M=1$ se c'è stata modifica rispetto a quella che c'è in memoria



Se $R=1$ scrivo la linea vecchia in mem. e poi la rimpiazza altrimenti la rimpiazza direttamente (nel caso di un miss)

Soluzione 2 (WRITE THROUGH) \Rightarrow Richiede più accessi in memoria !!!

Non c'è bit M , ogni store nella cache compie una scrittura asincrona nel livello superiore di memoria

Spiegazione: Appena ho modificato la cache, in modo asincrono modifico anche la memoria

Potrei utilizzare il write through solo se le mem. in cui scrivo il dato è poco più lenta delle cache, perché nell'incontro tra 2 stesse intuizioni, la prima deve aver finito quando inizia la seconda

CACHE MULTILEVELLO

Vengono usati più livelli di cache perché l'accesso in mem. è molto lento.

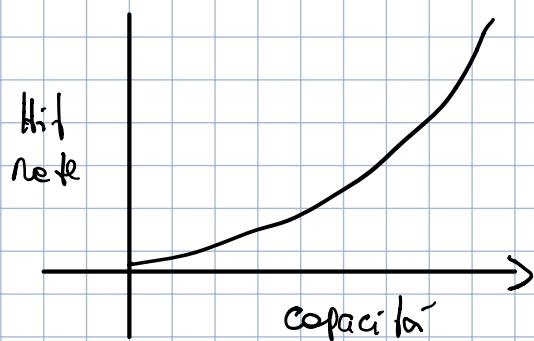
La cache di livello 1 è quella più vicina al processore ed è la più piccola. Quindi sono più veloci ma con un min rate alto.

La cache di livello 2 è un po' più grande delle cache di livello 1 quindi sono più lente ma con un min rate più basso rispetto alle cache di lvl 1.

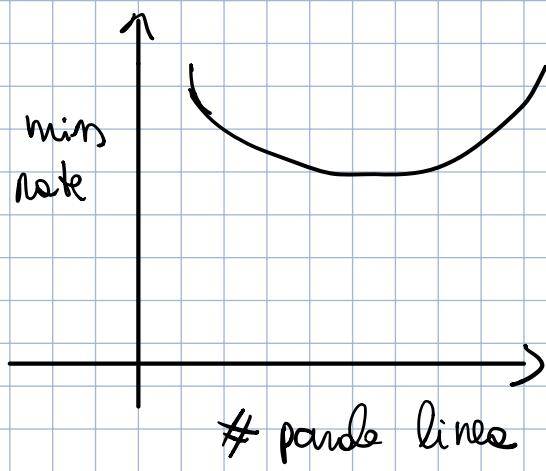
Non manca che si aumenta con i lvl di cache saranno sempre più lente, ma avremo un min rate sempre più basso.

Capacità in parole in una cache = n° set x n° parole per ogni set

Hit rate:



L'hit rate aumenta all'aumentare delle capacità di parole



Il miss rate aumenta quando ho poche parole per linea perché avendone poche ho più possibilità di non avere quella che cerco, quando invece aumentano il numero di parole quando ho più possibilità di trovare parole che non mi servono, quindi di avere un miss

Classificazione del tipo di miss nelle cache (Perché' posso avere un miss)

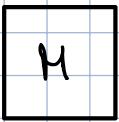
1) Capacity = Non ho capacità sufficiente per tenere tutti i dati che mi servono in quel momento

2) Compulsory (obbligatori) = Miss che devo avere per forza

Esempio: la prima che carico dati nella cache ho per forza un miss

3) Conflict = Problema sugli indirizzi (codice compilato male)

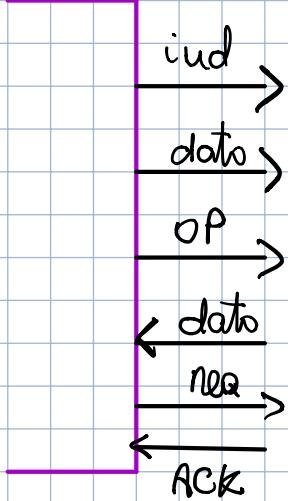
Lo elementi si fanno fuori a vicenda dalla cache perché



INTERFACCIA



IMPLEMENTA UN
AUTOMA



mappa: sullo
stesso
ret

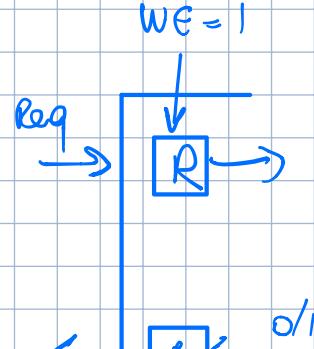
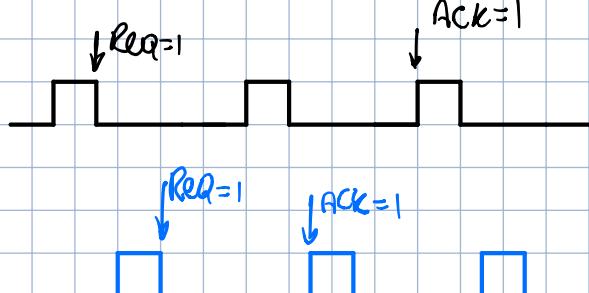
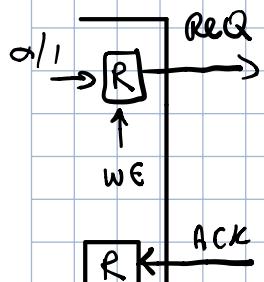
Come sincronizzazione Cache e memoria (ho bisogno di un protocollo di sincronizzazione)

Usa i segnali req e ack

Protocollo:

- 1) Cache manda $\text{REQ}=1$ dicendo che l'altra interfaccia deve fare qualcosa
con: neg dell'interfaccia che manda $\text{REQ}=1$
 - 2) l'altra interfaccia lavora e quando ha finito manda un $\text{ACK}=1$
 - 3) Per rispondere che la prima interfaccia ha visto che ha finito di fare il lavoro manda un $\text{REQ}=0$
 - 4) Infine l'altra interfaccia risponde con $\text{ACK}=0$
- e si ricomincia da capo con un altro blocco,

clock interfaccia,





$WE=1$ sempre

clock interfaccia



Ack

WE

Sincronizzazione tra unità

Nel caso dovremmo inviare un singolo dato, non avremmo problemi perché il dato venrebbe inviato con il tempo del ciclo di clock dell'unità che lo invia, e l'unità che lo riceve lo leggerebbe quando il suo ciclo di clock va alto.

Nel caso però voleremo inviare più dati, dovremmo sincronizzare le unità perché ci presentano due casi problematici:

- 1) Se U_a è molto più veloce di U_b , U_a potrebbe inviare il secondo dato prima che U_b legga il primo, quindi leggerebbe solo il secondo perdendosi il primo dato.
- 2) se U_a è molto più lento di U_b , avremmo che U_b non è in grado di capire ne la sequenza di comunicazione è composta da tre messaggi o da due messaggi

Per risolvere i problemi occorre introdurre un protocollo di comunicazione tra le due unità.

Può essere implementato in 2 modi: (comunicazioni unidirezionali)

- 1) Protocollo a livelli (per neg d'interfaccia da 1 bit)
- 2) Protocollo a transizione di livello (dispositivi più complessi)

Protocollo a livelli

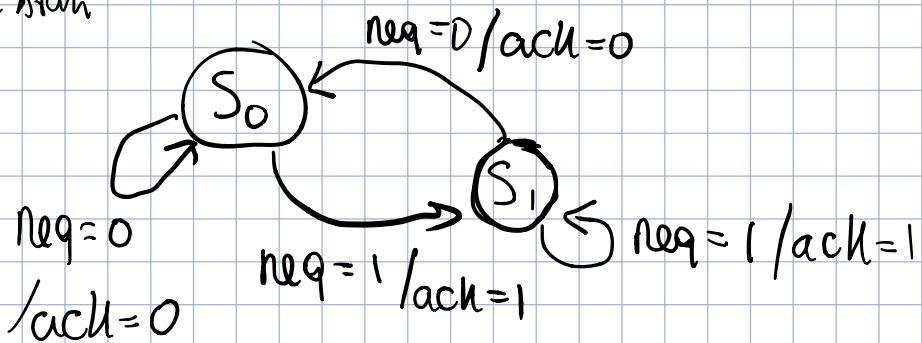
Le due unità hanno un reg d'entrata ed uno in uscita da 1 bit.

Il reg d'uscita dell'unità Ua è collegato al reg d'entrata di Ub e viceversa.

Quando Ua deve comunicare un dato a Ub il protocollo per la comunicazione è il seguente:

- 1) Ua verifica che il proprio reg di dest sia 0
- 2) Scrive il dato da comunicare in SRC e mette il reg d'invio a 1 con $\text{reg dest} = 1$
- 3) Ub trova il risultato nel suo reg d'arrivo \checkmark e mette il proprio reg d'invio a 1
- 4) L'unità Ua trova il reg dest a 1 e impone il reg d'invio a 0
- 5) L'unità Ub vede il suo reg dest a 0 e setta il reg d'invio a 0

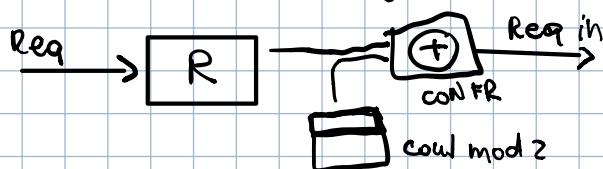
Il controllo dell'unità in questo caso è organizzato come un automa a stati.



Richiede 4 cicli di clock!!

Protocollo a transizione di livelli

- Utilizziamo contatori modulo due per le interfacce d'uscita al posto di registri da 1 bit.
Si tratta di un reg da 1 bit che viene complementato ogni volta che il suo WE = 1 e clock = 1
(Viene chiamato: Indicatore a transizione di livelli in uscita)
- Utilizziamo una rete sequenziale composta da un reg, una contattura mod 2 e un confrontatore (XOR) per avere il valore in ingresso (Req).



Utilizzando questi dispositivi puo' essere effettuato il reset alle condizioni iniziali, in autonomia da parte dell'unità che ha utilizzato l'indicatore.

Operazioni protocollo:

- 1) U_A mette in src il valore e setta l'indicatore d'invio (set del WE del contatore mod. due da parte della control unit)
- 2) U_B vede l'indicatore d'ingresso = 1, salva il messaggio che è in DST e esegue una op. di set dell'indicatore d'invio e di reset dell'indicatore di ingresso
- 3) U_A osserva che l'indicatore di arrivo è = 1

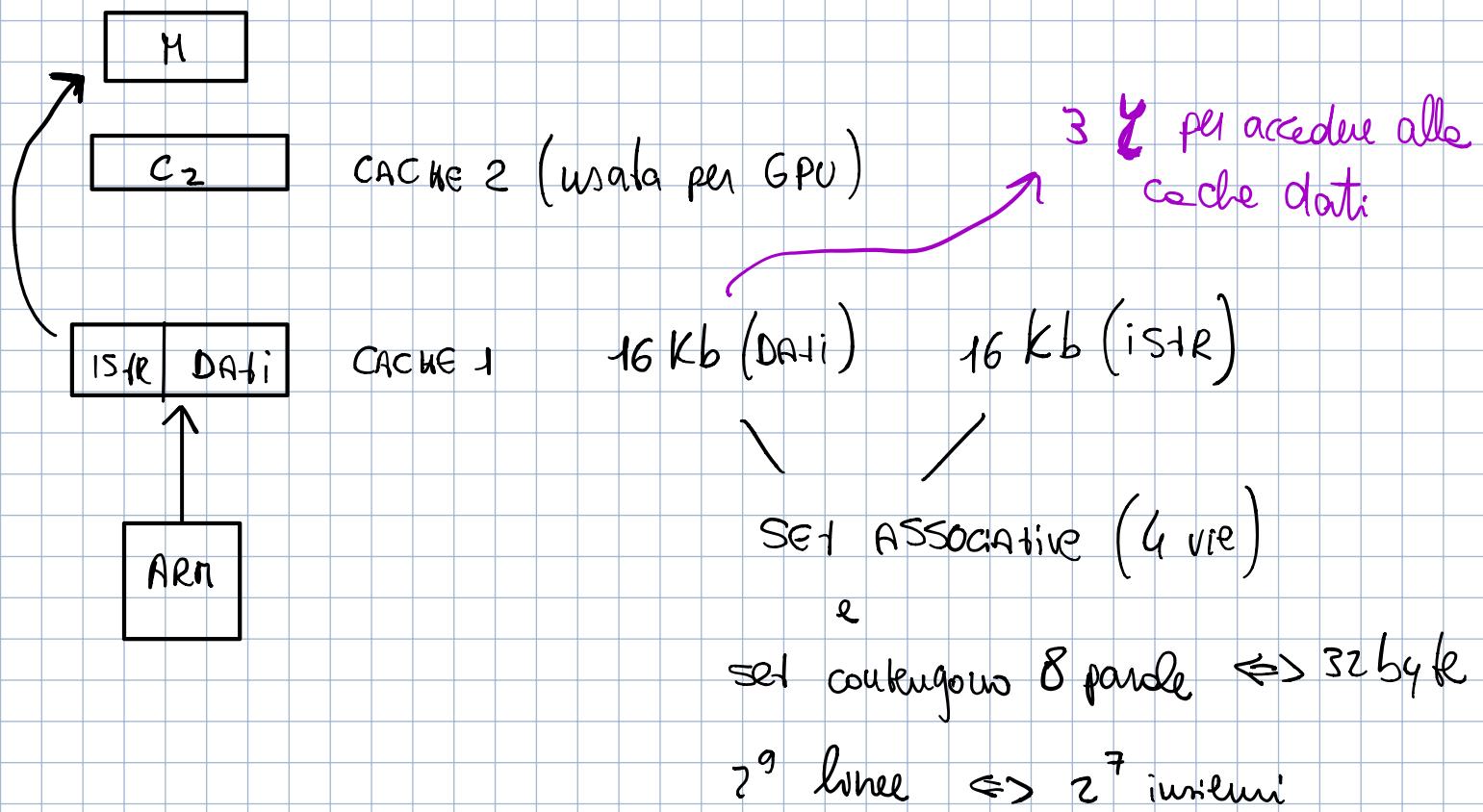
quindi puo' eseguire una nuova comunicazione verso
VB settando l'indicatore d'invio e restituendo quello
d'ingresso.

Richiede 2 cicli di clock !!

Comunicazioni non unidirezionali:

Per effettuare comunicazioni non unidirezionali:
entrambe le parti dovranno avere un registro
SRC e DST per inviare e ricevere i dati

ARM



indirizzo

tag	#set	off
20 bit	7 bit	5 bit

	parole
	32 byte

Problemi: programmare ARM con le memorie

- del
- 1) SPAZIO = Devo poter adattare le granularità e gli indirizzi del programma alle granularità della memoria
 - 2) Posizione = Devo capire cosa avviene agli indirizzi a seconda di dove viene letta o scritta (in determinati indirizzi)

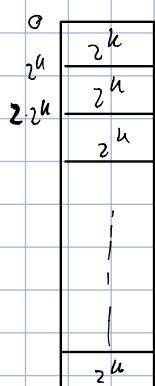
concerito il prog. in memoria (ind dell'ind \rightarrow ind Mem) e dallo spazio occupato

3) Fragmentazione: Non ho spazio di caricare il prog in memoria con indirizzi consecutivi (memoria contigua)

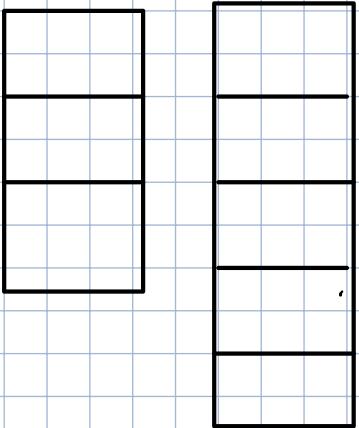
Prog \Rightarrow Spazio di Mem virtuale (contiene ind. generati dal P)

Memoria \Rightarrow Spazio di Mem Fisica (contiene ind memoria)

PAGINA: 2^k posizioni consecutive che iniziano dall'ind con ultime k bit a 0
Mem V o Mem F
Potenza di 2 torna bene



Disco Rigido
spazi di mem virtuale \leftarrow non devono per farla coincidere \rightarrow spazio di Mem Fisica



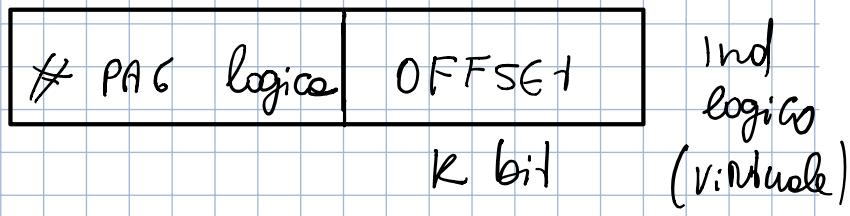
Divido i prog in pagine



Divido le M in pagine

1) Carico del prog solo le pagine che mi servono (working set)
in memória física

2) Divido l'ind in 2 parti

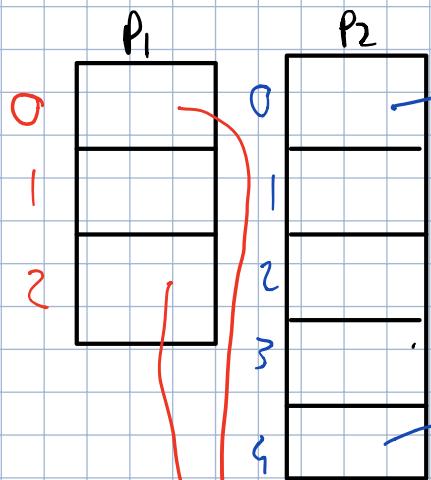


QUESTO MUORE LO FA L'S.O.

Per ogni programma ho una tabella di n° pos = num pagine programma

dove indica la posizione in cui è stato messo il prog in mem.

spazio di mem virtuale



spazio di Mem Fisica

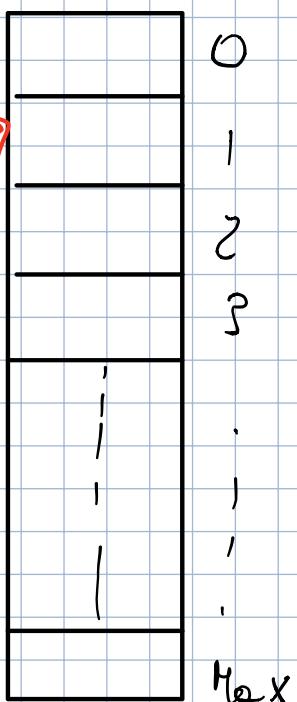
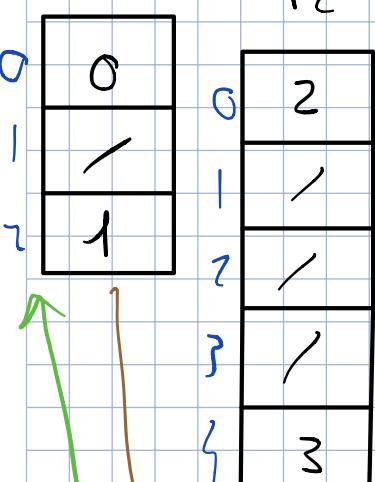


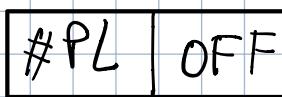
tabella
di mappatura



IND logico transf in indirizzo fisico

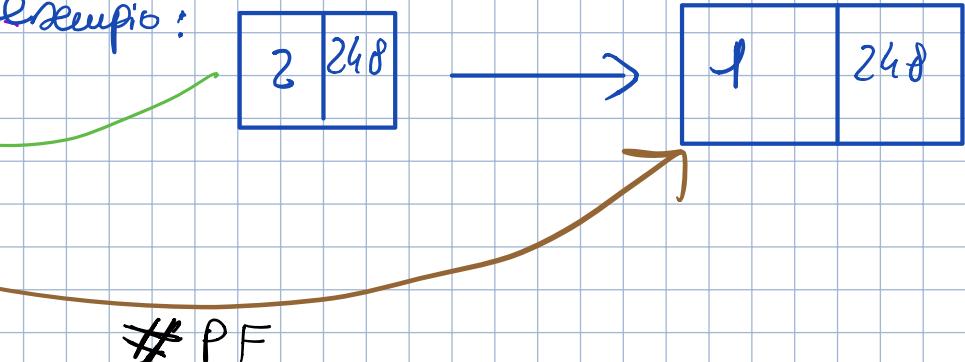
bit per rappresentazione → esempio: $nk = 12$ bit

\uparrow la grandezza
delle pag.

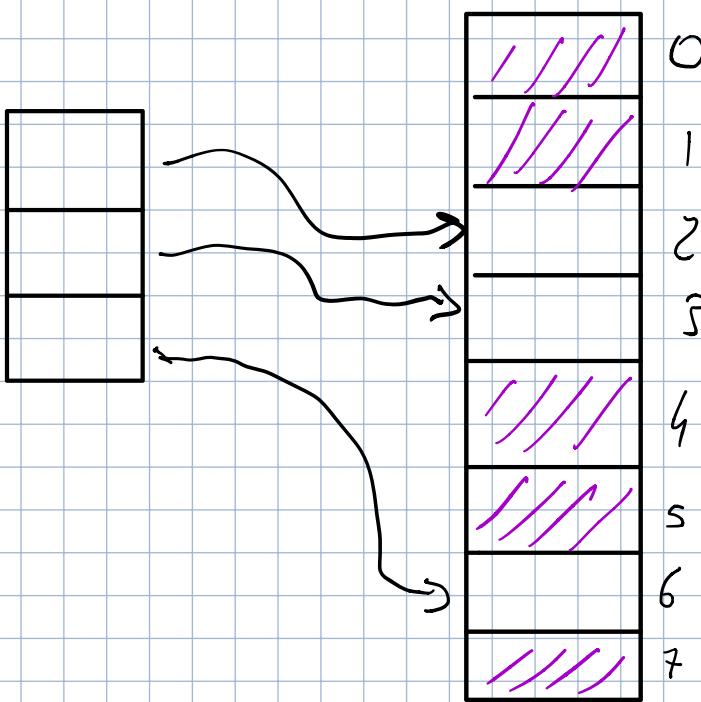


cerco nelle tab. d' indirizzamento
p_j l'ind della
mem al ~~PL~~

Esempio:

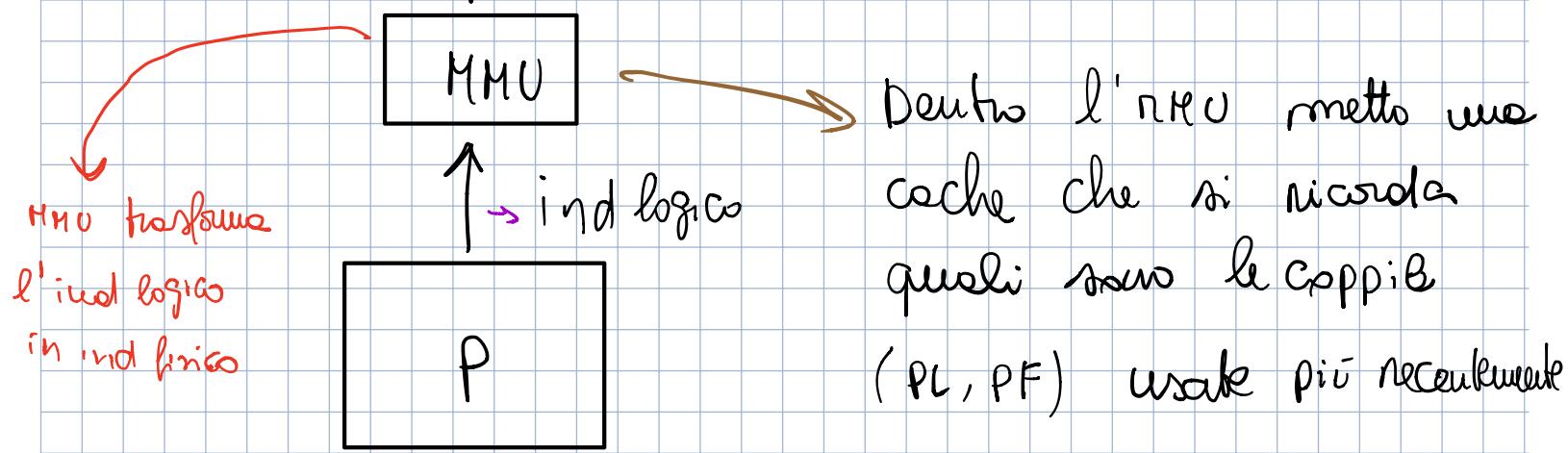


- 3) le pagine del prog. li carica nelle pag. libere delle mem.
anche se non sono consecutive
(risolto perché le pag. hanno regolare lunghezza)

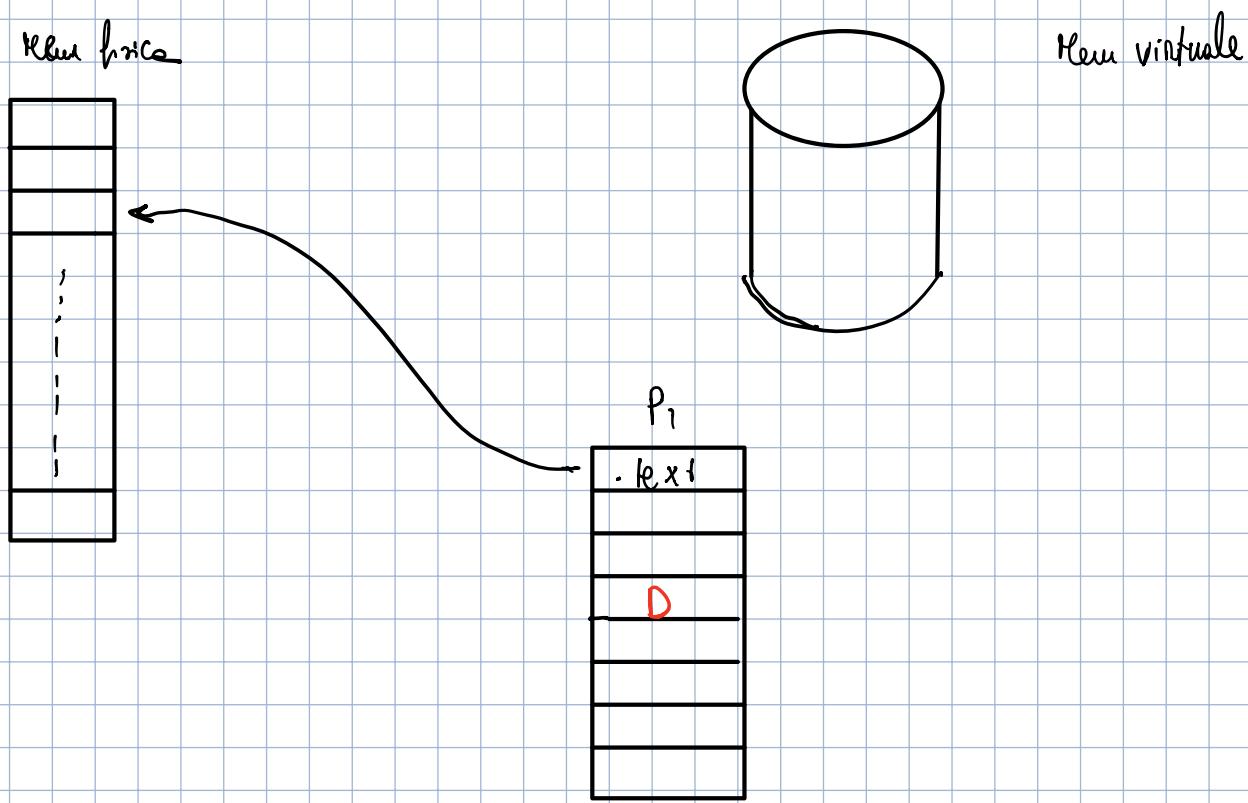


Per fare questo ho un'unità chiamata MMU (Memory Management unit)

↑ IND Fisico



Paging continuo:



Fault di pagina

Nel caso doverne fare la load di un dato che sta in una pag. del prog.

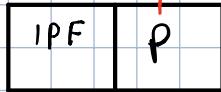
1) Genero ind. logico per D



2) L'MMU utilizza l'IPL per andare dentro le tabelle di mappatura del processo (la tab. sta dentro la memoria)

$$\Rightarrow P=0$$

3) All'indirizzo IPL dovei avere un altro indirizzo



(bit di presenza)

Essendo che la pag non è stata caricata ancora in memoria, dentro IPF ho nasc che non mi serve

4) P=0 da un'eccezione che invoca S.O. che cerca una pag libera in memoria e carica la pag del dato in memoria

5) Mette l'indirizzo della pag. in memoria al posto di IPF e mette P=1

6) Viene rifatta la load, rigenero lo stesso ind logico, trovo l'ind della pag. finita con P=1 e carico il dato

Questi passaggi avvengono on-demand \rightarrow fai mani che generi gli ind logici che non corrispondono a una pag libera della memoria, prende le pag dal disco e le carica in memoria.

Pagina da caricare = offset lunghezza pag * una pag logica

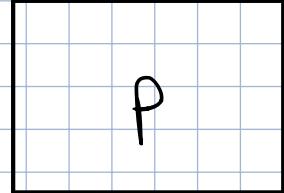
La memoria finita si comporta come una cache di ciò che è sul disco e le pagine vengono gestite come se fosse una cache.

Nelle MMU trovo una cache perché se non faccio 2 accessi in memoria ogni volta che devo tradurre un indirizzo



6 (ne ho una per ogni processo)

l'ind logico
in ind fisico



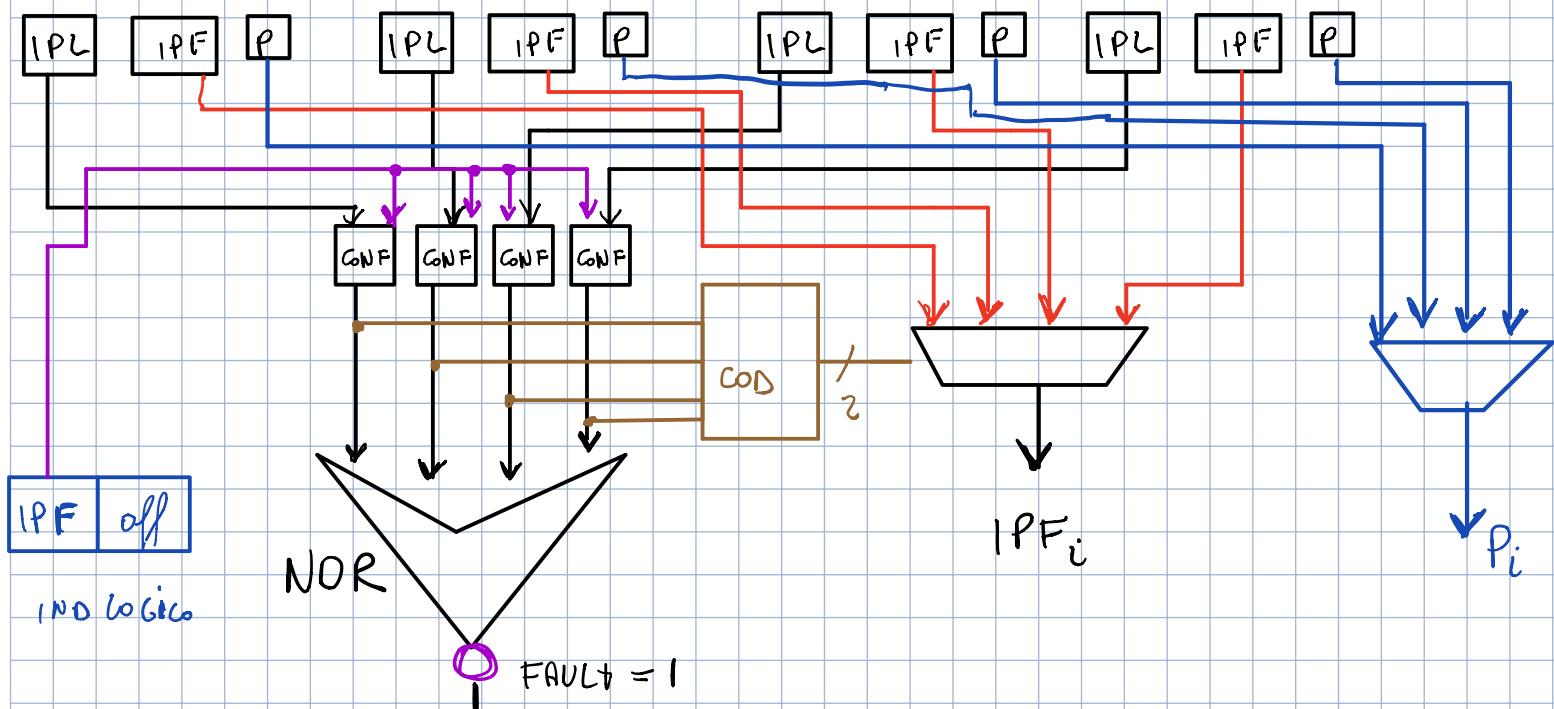
che si ricorda
questi sono le coppie
(PL, PF) usate più recentemente

Deve essere molto veloce

la cache dentro l'RRU è completamente associative e si chiama

TLB ←

traduzione indirizzo



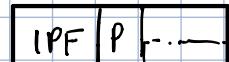
Indice i t.c $IPL_{\text{indlogico}} = IPL_i$

1) Se ho un fault devo accedere in memoria con ind:

base tabnfil + IPL

2) leggo il valore alla pos della tab di nolocation

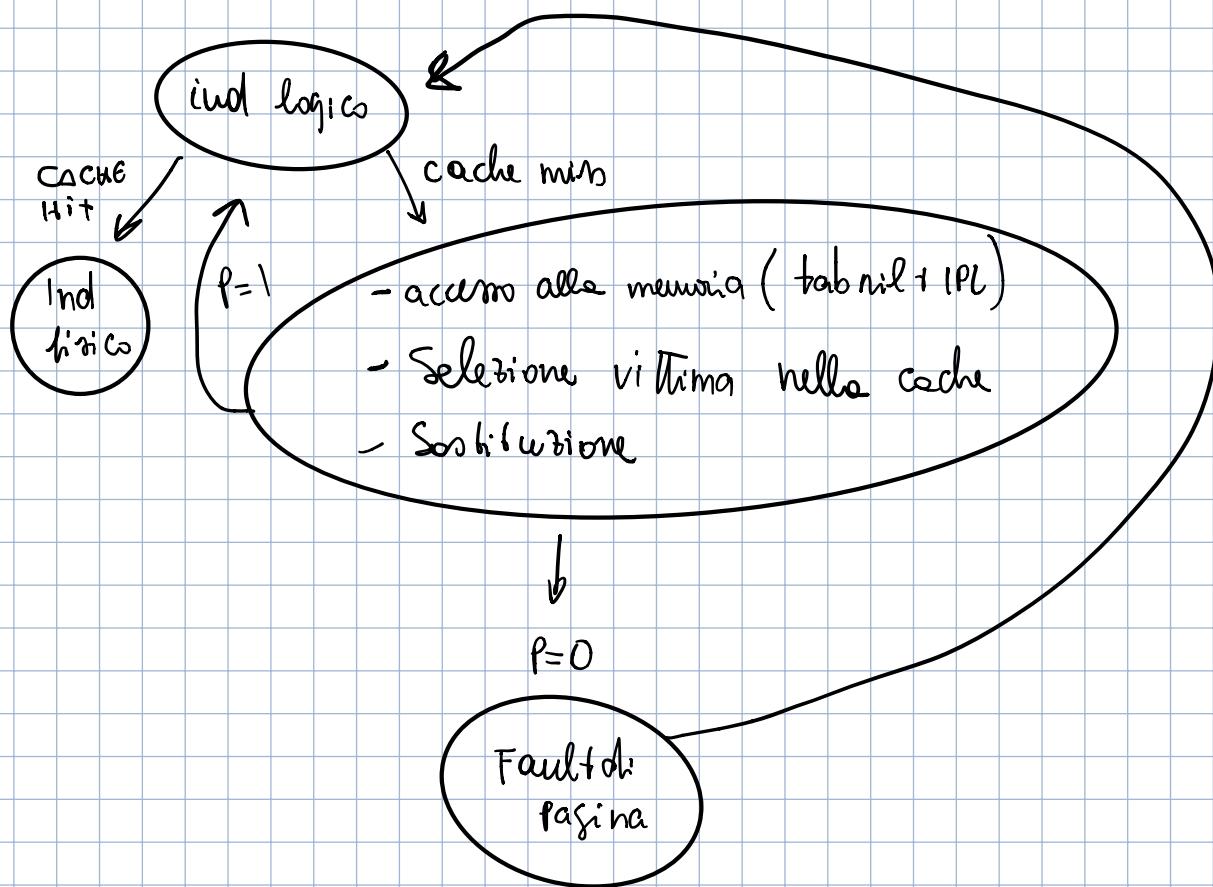
IPL



3) Prendo una delle 4 posizioni, la butto via, e ci metto la mia nuova associazione \hookrightarrow politica LRU

(IND PAG Logica, IND PAG Fisica + P)

Automa parte di controllo MMU



Attenzione: l'LRU è implementata come unità dentro il processore perché se la implementassi esternamente farei delle op in più inutile quando ho un fault di pagina

L'ARM che utilizziamo ha una gerarchia per la TLB

μ + TLB $\xrightarrow{\quad}$ I (una per le istruzioni)
 $\xrightarrow{\quad}$ D (una per i dati)

\Rightarrow hanno capacità 10 entry e lavorano in un ciclo di clock 1 μ

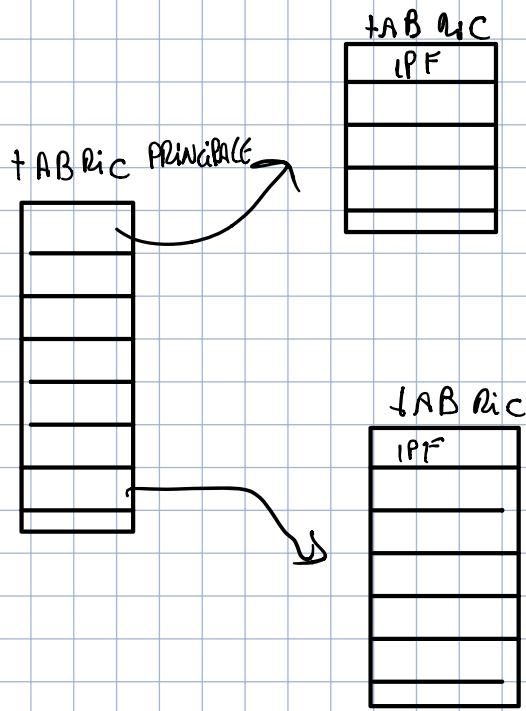
Utilizzano una Politica RoundRobin

TLB \rightarrow Se non trovo dentro μTLB \Rightarrow è unica per istruzione
 uso TLB
 sol. =
 Set-Associativa

se non funziona la TLB

Vedo in memoria

le tabelle di ricollocazione vengono fatte in questo modo



la tabella di ric. principale è composta da puntatori a altre tab di ricollocazione e ci dicono quale parte prendere delle tabelle "secondarie".

Faccendo ciò non ho bisogno di una tab. di ric. principale molto grande, e le frego in cache. Poi mando che mi servono le pag dall tab. secondarie le inserisco in cache.

Quindi l'inoltro



ha IPCL diviso in

2 parti dove una parte dice che indice prendere nella tab.
principale, mentre la seconda parte indica quale prendere delle tab.

Secondaria

Condivisione di una parte di mem a più programi

Faccio corrispondere zone diverse dello spazio logico alla stessa
zona fisica

Esempio: Mem video IPL 

P₁ IPL 

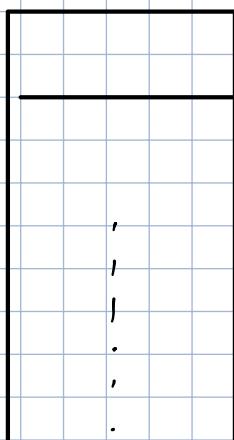
P₂ IPL' 

Per far sì che processore e memoria possano comunicare indipendentemente
senza un "master" che decida di iniziare, ho bisogno di 2
indicatori.

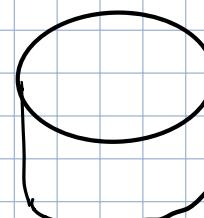
Se invece ho un master e uno slave ne basta 1

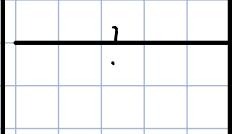
Riassunto Paging

Memoria Fisica (DRAM)



MEMORIA VIRTUALE (Disco Rigido)





Divido tra la mem. fisica che quello virtuale in pagine tutte della stessa grandezza.

La memoria fisica funziona come da cache per la mem. virtuale cioè, solo le pag usate sono caricate nella mem. fisica.

Il processore cerca d'individuare l'indirizzo fisico tramite quello virtuale.

Per fare ciò abbiamo una tabella di ricollocazione che associa l'indirizzo virtuale a quello fisico.

La tab. di ricollocazione è salvata sulla mem. fisica e contiene un elemento per ogni pag. virtuale associato ad un bit di presenza P che indica se la pagina è allocata sulla mem. fisica o unicamente sulla mem. virtuale.

Quindi, ogni volta che viene richiesta una lettura o scrittura in mem. bisogna fare 2 accesi in era.

La prima volta per controllare tramite l'ind. virtuale se la pag. è caricata in mem. fisica o no (bit P).

Se il bit P=0 viene mandata una call di sistema e viene inserita la pag. in mem. fisica con alcune operazioni e viene settato il P=1

Se il bit P=1 allora si fa un secondo accesso in mem. fisica tramite l'indirizzo ricevuto dalla tab. di ricollocazione e viene preferita la pagina cercata.

L'indirizzo fisico e logico condiviscono un offset che

indice quale pagina prendere.

L'indirizzo base delle tabelle di ricollocazione è salvato in un reg speciale chiamato "Registro di tabella delle pagine".

Quindi per trovare quale indirizzo fisico leggere somma all'indirizzo della tab. di ricollocazione, l'indirizzo virtuale.

Il processore contiene una mem. cache interna "TLB" full-associative che permette di salvare gli ind. fisici e logici usati più recentemente.

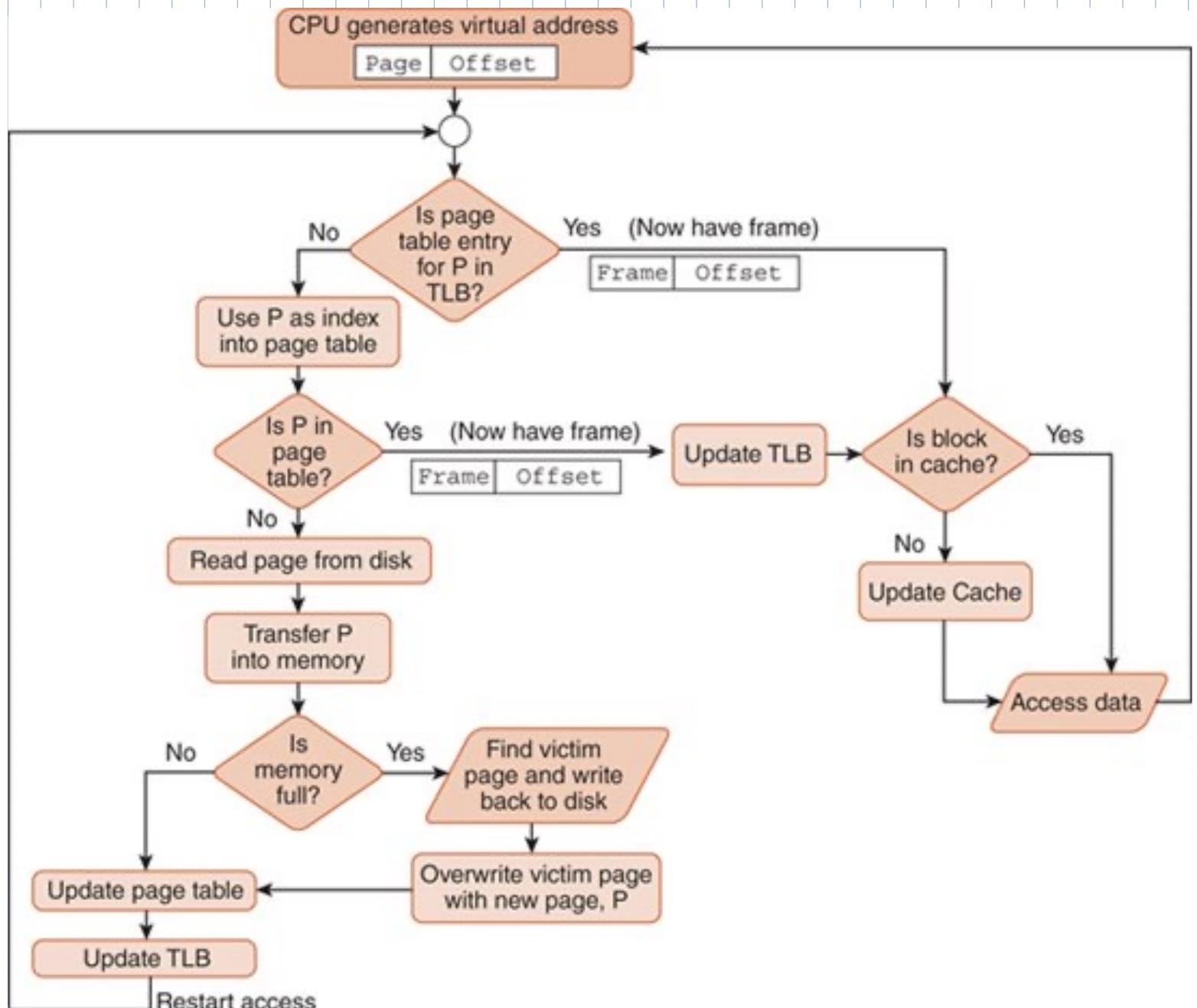
Protezione nella memoria

Ogni programma ha un suo proprio spazio di indirizzamento virtuale, volendo può usarlo tutto ma solo una parte delle pag. è caricata in mem. fisica.

Un programma quindi può accedere solo alla pagine fisiche mappate nella propria tab. delle pagine, quindi non può accedere alle pag. fisiche degli altri programmi perché tali pag. non sono mappate nella sua tabella.

Nel caso dei prog. dovesse accedere ad una risorsa condivisa, il sistema operativo si occupa di determinare quali programmi hanno diritto di scrivere su pagine condivise.

Da processore a cache / memoria



ciclo di clock > 330 perciò ma esistono pipeline
ma nel dire che tira fuori non ris. ogni & se non ho
problemi.

Processore Pipeline!

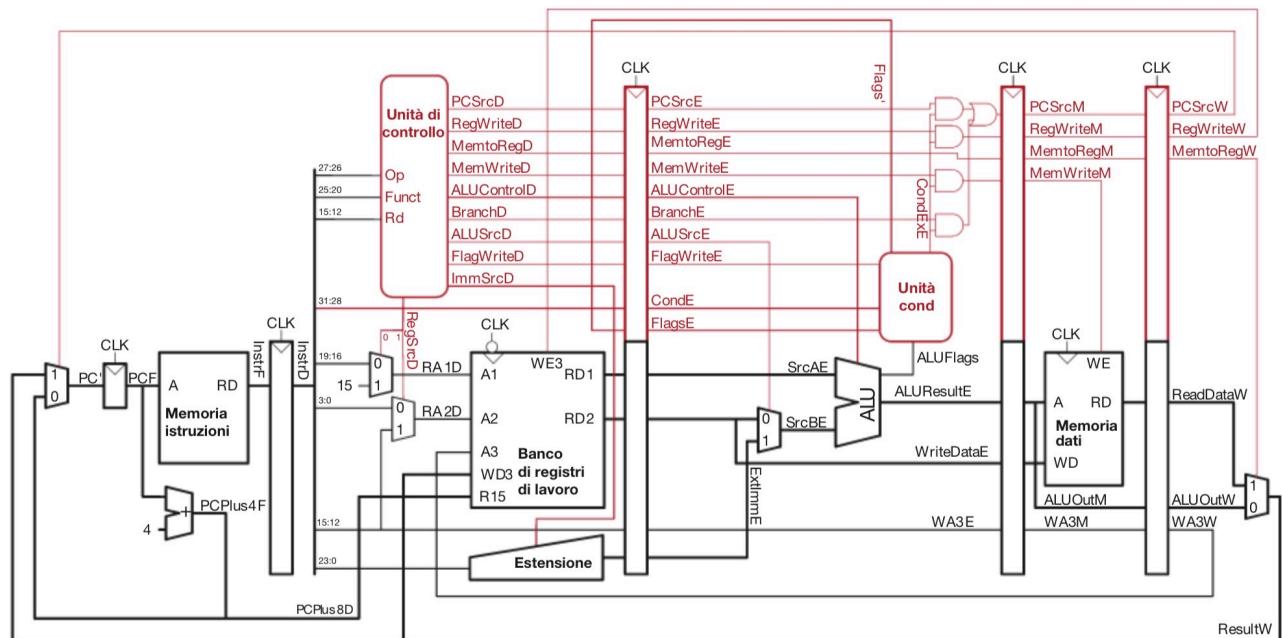


Figura 7.47 Processore pipeline con controllo.

la parte di controllo ha dei registri per dare nei momenti giusti i segnali.

Problemi pipeline:

1) Date hazard (dipendenza)

Questo problema si verifica quando ho un RAW (read after write)

Esempio:

ADD R0, R1, R2

SUB R3, R3, R0

Non ho ancora finito di eseguire la write dell' ADD nel reg R0 e voglio leggere quel reg.

zioni:

1) Uso un comando NOP. Fa saltare dei cicli senza fare nulla in modo da far finire il write prima delle read. Molto costoso! Tempo di compilazione

Tanti clock, poca efficienza

2) Quando ho l'uscita delle ALU, posso direttamente mandare il res senza aspettare, perché è già il valore che deve utilizzare nelle pross. istruzioni

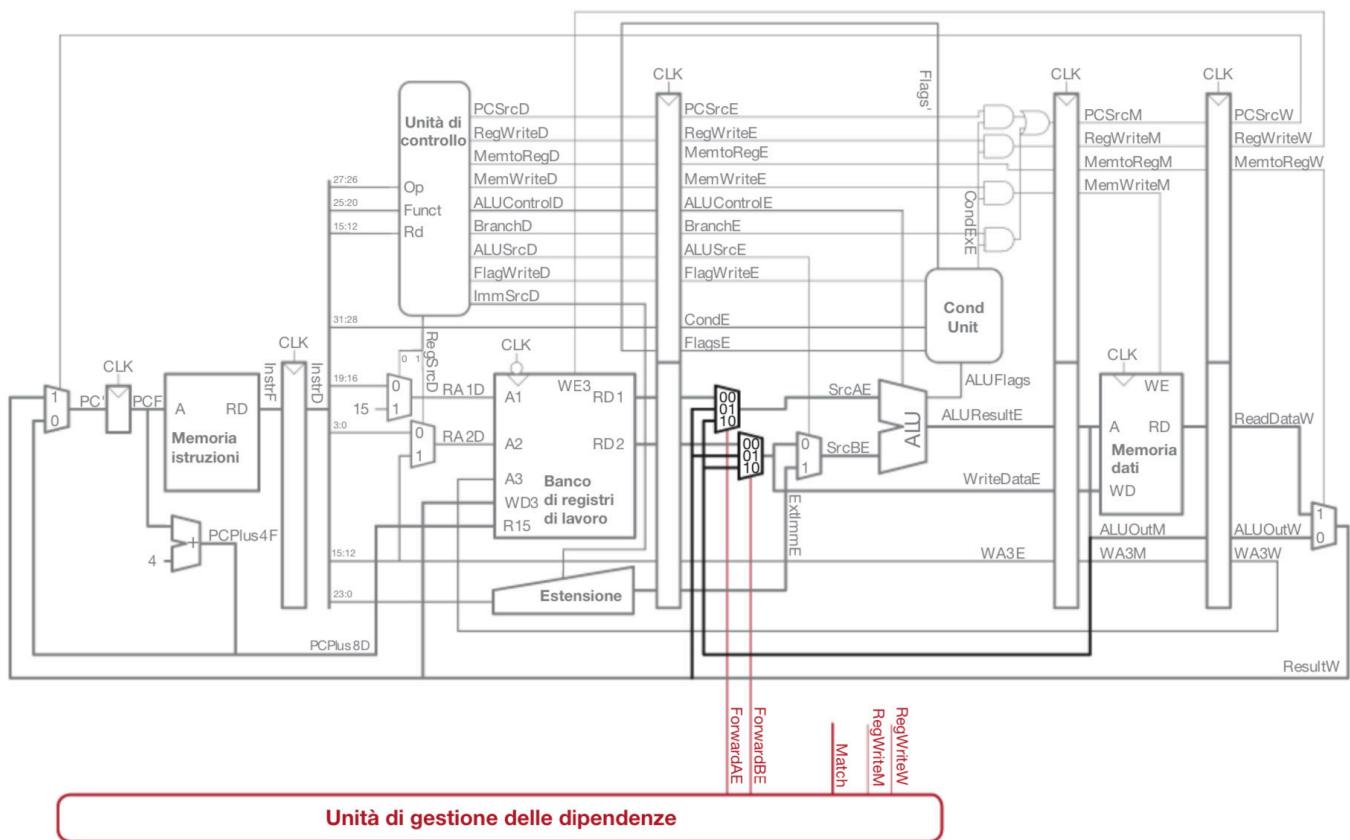


Figura 7.51 Processore pipeline con inoltro per risolvere le dipendenze.

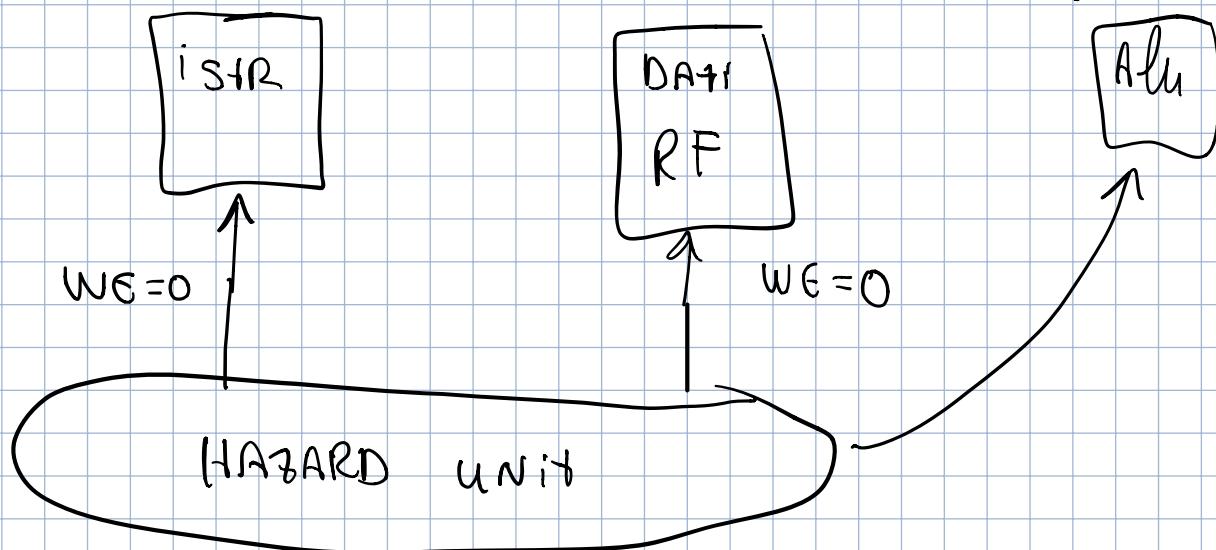
Aggiungo una unità chiamata hazard unit per gestire le "incorrette".

Se la prima instr è dipendenza della seconda attiva i controlli sui multiplexer

Nel caso di una load non ho modo di avere scorrimento perché deve passare per prima dal mem dati

Quando ci vuole un ando in più e faccio uno stall (non faccio nulla per 1 ciclo)

Per fare uno stall metto i WE = 0 nel reg istru e dati



DATA HAZARD
NOP
FORWARDING
STALL

Faccio una istru ogni ciclo tranne quando ho una load che ce ne metto 2 (con pipeline a regime)

2) Control hazard

Se ho un salto devo evitare di eseguire i comandi

che ho dopo il Branch

B loop

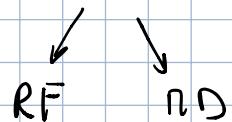
Devo
evitare
di eseguire
ADD
SUB
:
:

↓
loop :

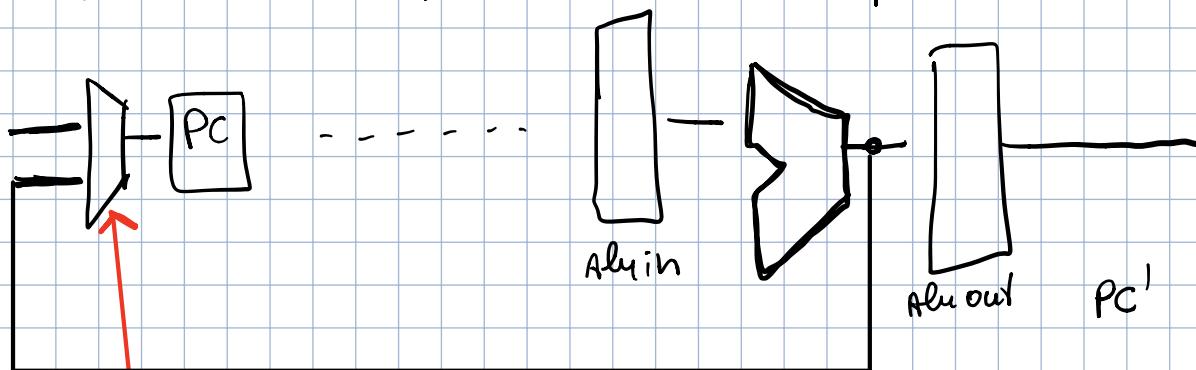
e farci
fare il WB

Soluzione:

1) flush: finire istruzioni inviolate con WE=0 sempre



2) Prendo il PC aggiornato nel filo dopo la alu e lo collego ad un multiplexer con cui decido quale PC prendere



HAZARD Unit

Per non perdere nessuna istruzione produco uno stall di 2 cicli di clock

Quindi per ogni ist. di salto ho 3?

Nelle realtà i salti che prendo più volte, gli eseguo a prescindere con $WE=0$

Quando ho verificato che il salto era giusto, allora eseguo le istruzioni.

Risolve però pipeline a tempo di compilazione

Se ho delle istruzioni RAW e cambiano l'ordine non cambia l'esec del prog. allora posso migliorare a tempo di compilazione

Esempio:

LDR --

ADD ---

LDR --

ADD ---

⇒
trasforno

ANALISI DATAFLOW

LDR---

LDR---

ADD - -

ADD - -

Rende distante di un ciclo e quindi

uso lo stalls delle prima LDR per calcolare la seconda LDR

CONDIZIONI DI BERNSTEIN → Fare in modo di non avere dipendenze logiche

$$1) R_1 \cap W_2 = \emptyset$$

$$2) R_2 \cap W_1 = \emptyset$$

$$3) W_1 \cap W_2 = \emptyset$$

→ Condizioni per non avere RAW

Diminuire numero bolle per i salti

Uno ha la tecnica "loop unrolling" che consiste nel fare più iterazioni del ciclo prima di fare un branch.

Se so il numero di iterazioni totali posso fare n iterazioni del ciclo con n divisore del numero di cicli totali.

Memory mapped I/O

tecniche per l' I/O

1) Memory mapped I/O :

c'è una tecnica con la quale si intende la possibilità di accedere ed interagire con i dispositivi I/O utilizzando istruzioni ARM per interagire con la memoria interna del dispositivo (LOAD e STORE)

Vengono associati degli indirizzi alla mem. interna di ogni dispositivo I/O.

Per far sì che non vengano interpretate come normali operazioni di LOAD e STORE:

il processore riconosce alcuni indirizzi come appartenenti all'I/O. tramite la MMU invia le richieste relative all'I/O sul bus che collega processore e dispositivi I/O.

tutti i dispositivi I/O vedono passare l'indirizzo, ma solo il dispositivo a cui è associato lo intercerterà.

Successivamente il dispositivo effettuerà l'operazione di LDR o STR inviata dal processore.

Gli indirizzi di I/O sono definiti dal costruttore del processore.

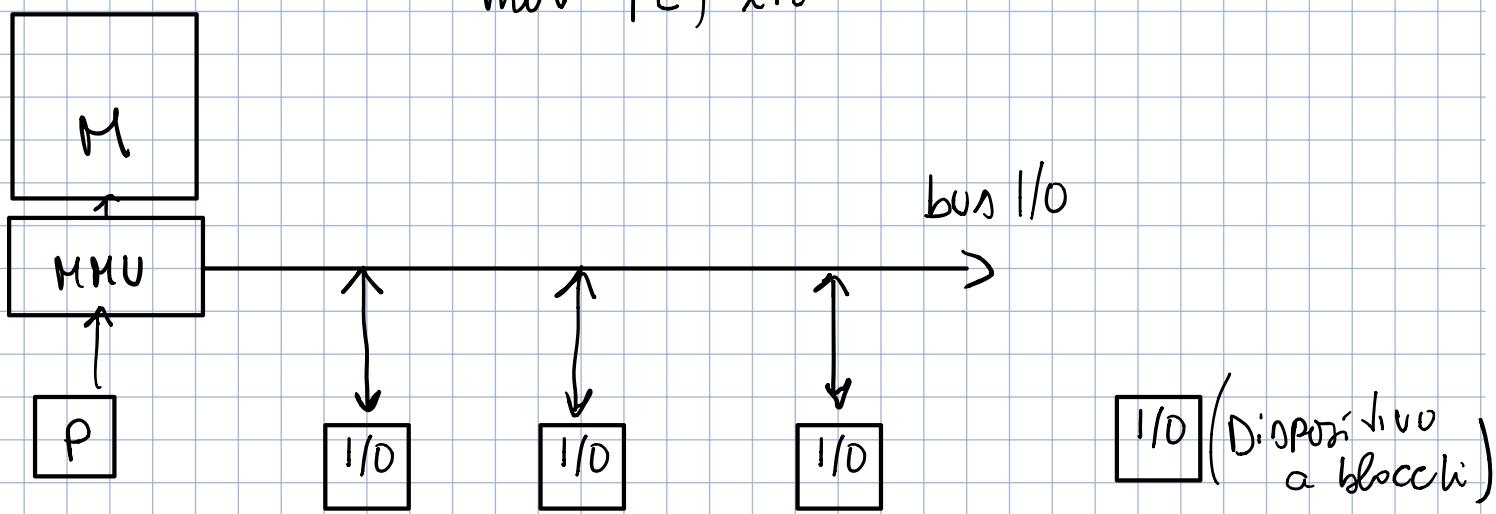
Nel caso di una lettura dalla mem. del dispositivo, il dato letto viene inviato sul bus I/O.

È molto inefficiente perché il ciclo "wait" (che aspetta che venga premuto un tasto) esegue istruzioni a vuoto.

Esempio di codice ASM:

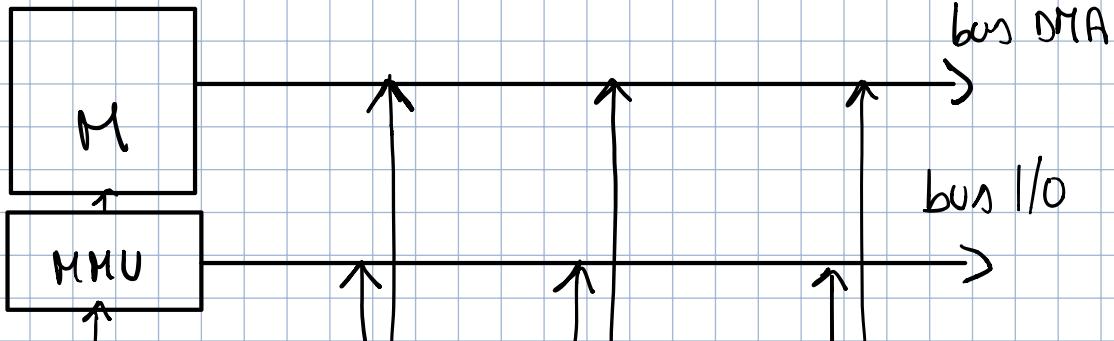
```

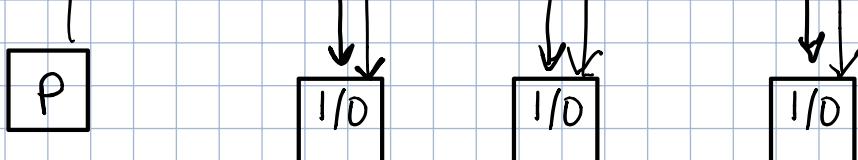
        mov r0, #1024
        mov r1, #1
        sJr r1, r0
        wait: ldn r1, [r0]
        cmp r1, #1
        beq wait
        ldn r0, [r0, #1]
        mov PC, lr
    
```



2) DMA (Direct memory access)

La tecnica DMA permette ad un dispositivo I/O di accedere direttamente alla memoria centrale.





Come funziona:

- Il dispositivo ^{I/O} è collegato alla mem. centrale mediante un BUS DMA e può effettuare lettura e scrittura in memoria quando ha ottenuto la possibilità di usare il BUS
- Il BUS DMA è condizionato da tutte le periferiche in grado di lavorare in DMA
- Il processore farà tramite il bus I/O, gli indirizzi necessari per eseguire le op. richieste direttamente in mem. centrale
- La mem. diventa un dispositivo che deve saper gestire le richieste sia da I/O che processore, quindi le porte di controllo deve capire quale op. fare

Operazioni con Memory mapped I/O e DMA :

- 1) tramite il Memory mapped I/O ordine l'esec. di lettura che contiene:
 - Specifica che si tratta di un op. di lettura
 - Indirizzo del blocco di disco interessato
 - Indr. in mem. del buffer da utilizzare per i dati letti dal disco
- 2) legge i dati e li salva in un buffer interno (Dovrebbe attendere troppo per l'accesso DMA e alla mem. centrale)
- 3) Acquisisce l'accesso al bus DMA e avvia il trasf. del blocco

Betto all'ind. di mem. prestabilito.

Dispositivi DMA: Dispositivi che lavorano a blocchi (disp di massa, interfacce di rete, video ecc..)

Dispositivi a canelli: tastiere, mouse ecc..

3) Interruzioni

↓
Eventi asincroni
rispetto all'exc del
programma

↓
Generati da dispositivi
esterni al processore
(I/O, page fault, ecc..)

Eccezioni

↓
Eventi esterni/non usuali
che hanno una relazione
con ciò che sto facendo

↓
generate dal processore in caso
di errori di vario genere.
(Esempio: Fault di pagina)

Le interruzioni vengono gestite alla fine del ciclo del processore.

```
while(true){
```

```
try{
```

```
patch;
```

```
decode;
```

```
execute; } catch(exception e) { exception management(); }
```

```
if(interrupt) gestisci interrupt; }
```

Risultato: quando si verifica un'eccezione vengono fatti una serie

a: parz:

- l'infra viene completata
- Si salva parte dello stato del processore (PC, LR, SP) e si esegue una parte di codice ASM che dipende dal tipo di eccezione generata e contiene la soluzione per l'eccezione.

Il codice viene eseguito con privilegi più alti rispetto al corrente user (ci si muove nello stato giusto)

- Al termine si ripristina lo stato del processore.

Oss: le eccezioni vengono gestite immediatamente prima di aggiornare il PC

Stati ARM utilizzati:

Fast Interrupt / Interrupt → Stati utilizzati per il trattamento delle interruzioni generate dall'I/O

Abort / Undefined → Accessi illegali in memoria o istruzione assembly illegale

Supervisor → Esecuzione chiamate di sistema SVC

Mode	Privileged	Purpose
User	No	Normal operating mode for most programs (tasks)
Fast Interrupt (FIQ)	Yes	Used to handle a high-priority (fast) interrupt
Interrupt (IRQ)	Yes	Used to handle a low-priority (normal) interrupt
Supervisor	Yes	Used when the processor is reset, and to handle the software interrupt instruction swi
Abort	Yes	Used to handle memory access violations
Undefined	Yes	Used to handle undefined or unimplemented instructions
System	Yes	Uses the same registers as User mode

Figura 2.4: Stati del processore ARM

Puplicazione registri vari stati :

User	System	Fast Interrupt	Interrupt	Supervisor	Abort	Undefined
R0	R0	R0	R0	R0	R0	R0
R1	R1	R1	R1	R1	R1	R1
R2	R2	R2	R2	R2	R2	R2
R3	R3	R3	R3	R3	R3	R3
R4	R4	R4	R4	R4	R4	R4
R5	R5	R5	R5	R5	R5	R5
R6	R6	R6	R6	R6	R6	R6
R7	R7	R7	R7	R7	R7	R7
R8	R8	R8_fiq	R8	R8	R8	R8
R9	R9	R9_fiq	R9	R9	R9	R9
R10	R10	R10_fiq	R10	R10	R10	R10
R11	R11	R11_fiq	R11	R11	R11	R11
R12	R12	R12_fiq	R12	R12	R12	R12
R13 (SP)	R13 (SP)	R13_fiq	R13_irq	R13_svc	R13_abt	R13_und
R14 (LR)	R14 (LR)	R14_fiq	R14_irq	R14_svc	R14_abt	R14_und
R15 (PC)	R15 (PC)	R15 (PC)	R15 (PC)	R15 (PC)	R15 (PC)	R15 (PC)

Program Status Registers

CPSR	CPSR	CPSR SPSR_fiq	CPSR SPSR_irq	CPSR SPSR_svc	CPSR SPSR_abt	CPSR SPSR_und
------	------	------------------	------------------	------------------	------------------	------------------

Il codice eseguito per frattare un'eccezione è eseguito ad interruzioni disponibilità (cioè non può generare a sua volta un'interruzione)

Questa eccezione viene specificata dai bit I e F del CPSR che indicano rispettivamente se le eccezioni IRQ e FIQ debbano essere ignorate.

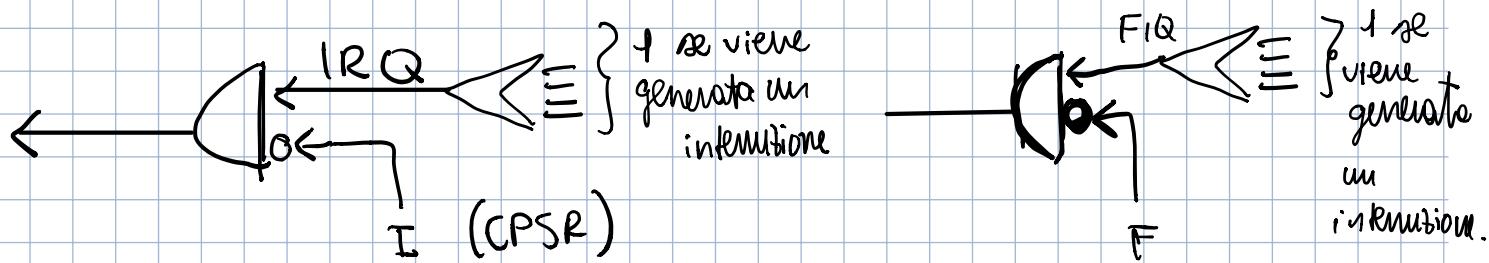
Se $I=1 \rightarrow$ ignora IRQ (interruzioni "normali")

Se $F=1 \rightarrow$ ignora FIQ (fast interrupt \rightarrow interruzioni con priorità)

Quindi quando valutiamo se bisogna gestire o no un'interruzione

avremo un "arbitro" che decide se c'è un'interruzione e

l'ordine di importanza, in AND con il bit del CPSR negato



I bit I e F vengono settati a 1 dopo aver salvato una copia del CPSR

Quando si verifica che c'è un'interruzione (il motivo dell'interruzione è posto a un indirizzo particolare della memoria, nota alla routine che gestisce le interruzioni.) una volta aggiornato il PC si:

- Salva il valore del PC nella copia del reg LR
- Si salva il CPSR nel SPSR
- Si setta il modo del programma (5 bit meno significativi del CPSR)

Per accedere le parti del reg vengono utilizzate 2 istruzioni particolari:

- MRS (Move nstatus to register) : Permette di copiare il contenuto del CPSR in un registro generale in modo successivamente di poter manipolare il contenuto
- MSR (Move register to nstatus) : Permette di scrivere un reg generale nel CPSR
- Setta il bit T del CPSR a 0

- A seconda dei casi setta I e F a 1 per evitare altre interruzioni (masker)
- Salta ad eseguire il codice che si trova all'ind 0x00000000 + tipo dell'interruzione definito da questo tabella (vettore delle interruzioni)

Indirizzi
nella memoria

0x 00000004

0x 00000008

0x 0000000C

0x 00000010

0x 00000014

0x 00000018

0x 0000001C

0	reset
4	undef instruction
8	sw interrupt (SVC)
12	abort (fatch da indirizzo illegale)
16	abort (ldr/str indirizzo illegale)
20	reserved
24	IRQ
28	FIQ

Figura 2.7: Tipo di interruzioni ARM

Al ritorno, dopo aver trattato l'interruzione:

- Sposta LR in PC
- Copia SPSR in CPSR
- Azzerà bit I e F del CPSR