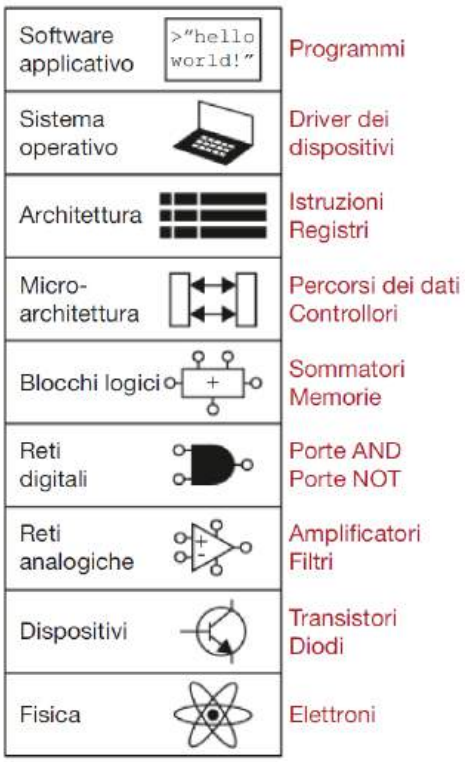
Introduzione

INTRODUZIONE

Il **ciclo di clock** è l'unità base di tempo che regola il ritmo con cui un processore (CPU) esegue le istruzioni. Un ciclo di clock corrisponde a un singolo impulso (un'oscillazione) e determina quando le operazioni possono iniziare o terminare all'interno del processore.

Legge di Moore: ogni 18 mesi circa raddoppia il numero di transistor che puoi mettere nello stesso spazio.

Possiamo vedere un calcolatore come una macchina a strati o livelli:



ds

Andare verso l'alto significa **astrarre**, andare verso il basso **concretizzare**. Il principio di astrazione si basa sul fatto che i livelli sopra possano comunicare con quelli sotto in maniera **modulare** e **indipendente**. Si può cambiare un certo livello o strato ma, se è stato implementato bene, tutti i livelli sopra e sotto continueranno a funzionare.

Principi di progettazione

1. *Principio di gerarchia*: progettare una componente complessa in termini di componenti più piccole e più semplici da progettare;
2. *Principio di modularità:* il sistema deve essere composto da componenti o moduli che implementino funzionalità ben chiare e che implementano interfacce che favoriscano la composizione di più componenti.
3. *Principio di regolarità:* indentificare funzioni e moduli comuni che possano essere riutilizzati.

Sistema binario

Potenze del due: 2^10 = KB, 2^20 = MB, 2^30 = GB, 2^40 = TB

Numeri relativi in binario

1) Modulo e segno

Il bit più significativo rappresenta il segno (se = 1 è negativo)

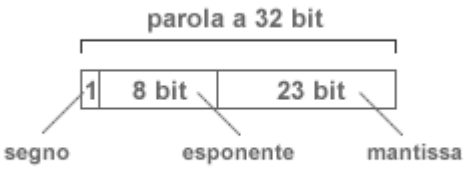
- **a)** posso rappresentare con n bit 2^n - 1 valori
- **b)** [-2^n + 1, 2^n - 1]
- c) problemi: **non posso fare la somma**, doppia rappresentazione dello zero (es: 1000 e 0000).

2) Complemento a 2

a) calcolo il complemento a 2 invertendo tutti i bit di un numero (1 => 0, 0 => 1) e aggiungendo 1 in binario.
b) 2^n possibili valori distinti
c) [-2^n, 2^n - 1]
d) **Condizione di Overflow**: il bit della cifra più significativa (segno) del risultato è diverso dal carry (resto o riporto).

Rappresentazione in virgola mobile

Per rappresentare i floating point number, utilizziamo lo standard IEEE.
Un numero in virgola mobile è rappresentato da un campo esponente e da un campo mantissa.
1.23 E2 Mantissa = 1.23



Shift logico a destra

Fissato k intero positivo, sposto k bit a destra e a sinistra aggiungo k “0”. Questa operazione equivale a dividere per 2 ^ k.
-esempio con k = 2: 1110 -> 0011

Shift aritmetico a destra (per rappresentazione in complemento a 2)

Fissato k intero positivo, faccio la stessa operazione dello shift logico solo che mantengo il segno quindi se all'inizio il bit più significativo era 1 a sinistra aggiungerò k "1" altrimenti k "0".
-esempio con k = 2: 1110 -> 1111

Shift a sinistra

Fissato un numero intero positivo k, sposto a sinistra k bit e **aggiungo** k "0" a destra. Questa operazione equivale a moltiplicare per 2 ^ k.
-esempio con k = 2: 1110 -> 111000

Porte logiche

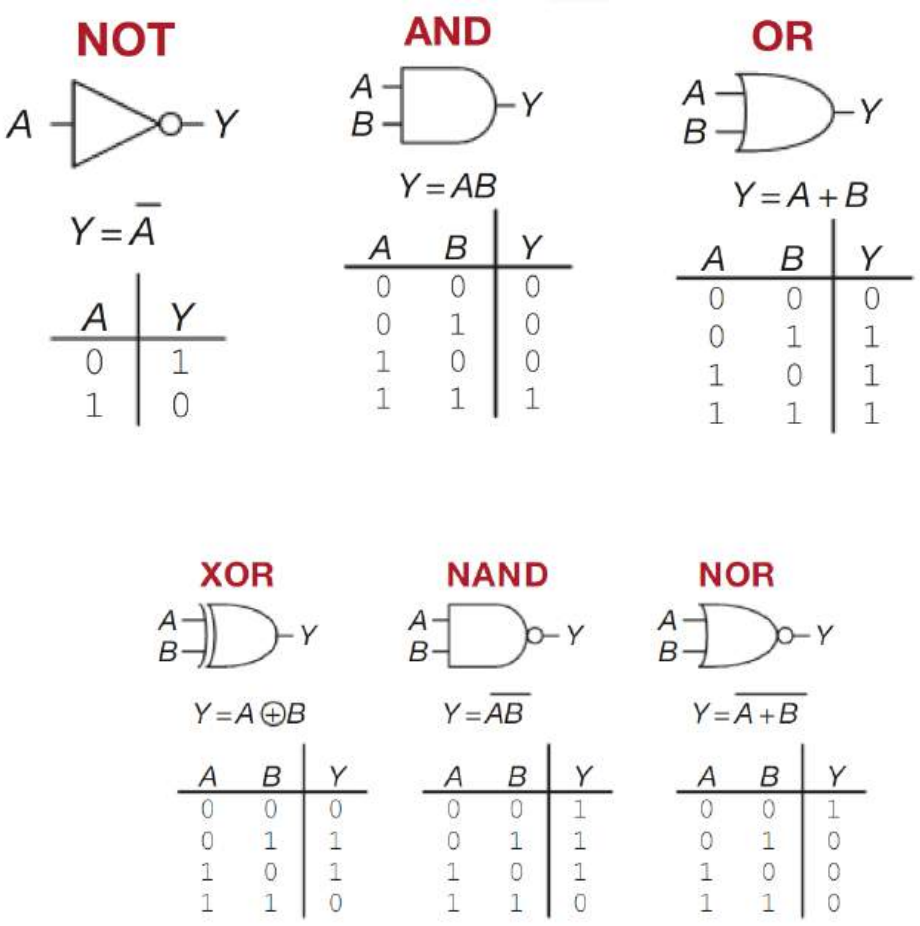


Figura 1.16 Altre porte logiche a due ingressi.

Ritardi delle porte logiche = un **TP** di ritardo per ogni livello logico.

Le reti logiche si dividono in:

1) **Reti combinatorie:** descrivono una funzione che per lo stesso input produce sempre lo stesso output.
2) **Reti sequenziali:** producono diversi output per lo stesso input a seconda dello stato (automi).

min termine = **prodotto** logico che coinvolge tutte la rete, nel quale compaiono tutti gli ingressi della rete.
max termine = **somma** logica che coinvolge tutta la rete, nel quale compaiono tutti gli ingressi.

Reti Combinatorie

Reti combinatorie

Sintesi di una rete combinatoria

La sintesi di una rete combinatoria viene fatta data una **tabella di verità**. Per ogni bit di output: scrivo la relativa funzione in algebra booleana, sommando i termini che portano l'output a 1.

Una volta che ho le funzioni scritte in algebra booleana posso disegnare lo schema delle **porte logiche**. Per ottimizzare lo schema delle porte logiche e andare a risparmiare sui componenti posso semplificare l'espressione con le **mappe di Karnaugh**.

Mappe di Karnaugh

Le mappe di Karnaugh servono per semplificare la rete logica descritta dalla tabella di verità. Funzionano solo con un massimo di 4 ingressi, quindi al massimo avremo una tabella di 4 x 4.

Ogni **casella** della tabella rappresenta il **bit** di **uscita** per quella tabella di verità.

La tabella rappresenta ogni combinazione di ingresso e per funzionare le combinazioni adiacenti devono **differire** di **un bit** !!!

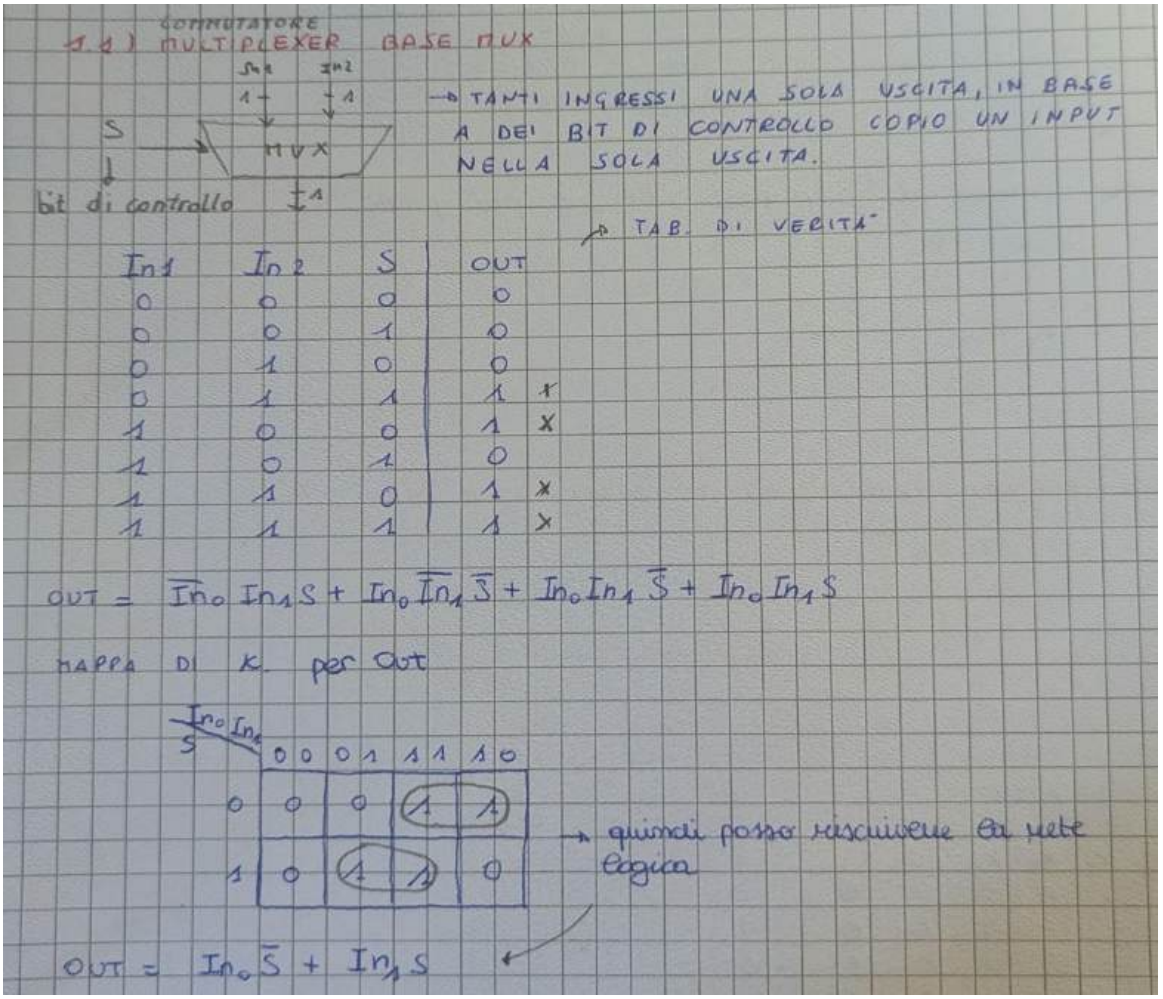
Una volta costruita la tabella procediamo col creare i cerchi tenendo conto che:

- ogni cerchio deve contenere solo 1.
- ogni cerchio deve avere un numero di 1 che rappresenti una **potenza del due** (1,2,4,8,16…)
- si cerca di fare il **minor numero di cerchi**.
- I **gruppi** possono anche **sovrapporsi** parzialmente (tenendo conto **l'effetto pacman**).
- si cerca di creare i **cerchi più grandi possibili**.
- Le **indifferenze** possono essere usate sia come 1 che come 0.

La potenza del due rappresentata dai cerchi rappresenta il numero di ingressi che andiamo a eliminare.

Il numero di cerchi rappresenta il numero di somme logiche e andremo a fare il prodotto tra tutti gli ingressi che per il cerchio di riferimento **non cambiano** bit.

Esempio di utilizzo delle mappe di K.



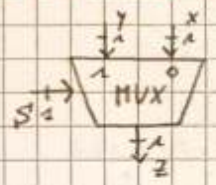
Definizione di alee

Le **alee** sono **variazioni temporanee indesiderate** del valore di uscita di un circuito logico **combinatorio**, dovute a **ritardi di propagazione** diversi nei segnali che lo attraversano. Le alee possono essere evitate aspettando il ritardo di propagazione. Esiste comunque la possibilità di evitare l'alea aggiungendo un'altra porta alla realizzazione della funzione. In generale, si presenta un'alea quando una variazione in una singola variabile di ingresso attraversa i confini di due implicanti primi in una mappa di Karnaugh.

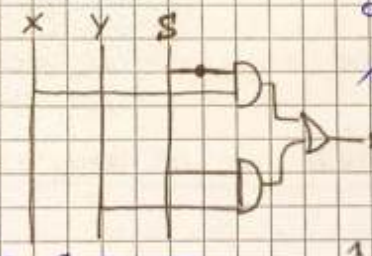
Come si verificano:

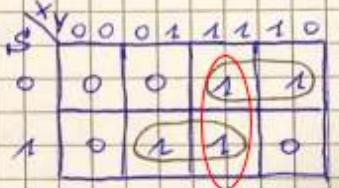
Quando più segnali cambiano simultaneamente, ma raggiungono l'uscita con tempi diversi, si possono generare **brevi impulsi errati** (glitch) anche se logicamente l'uscita dovrebbe rimanere stabile.

IL FENOMENO DELLE "ALEE"



$Z = \bar{S} \cdot X + S \cdot Y$

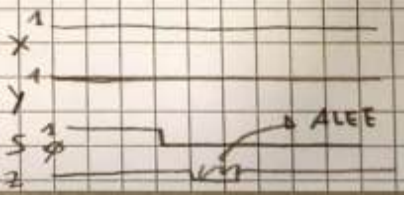


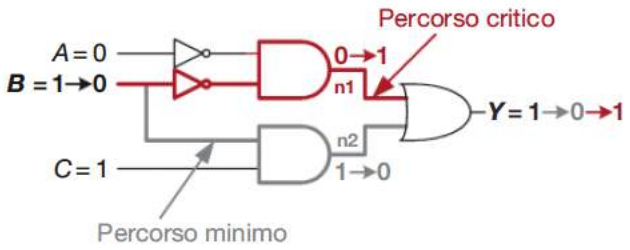


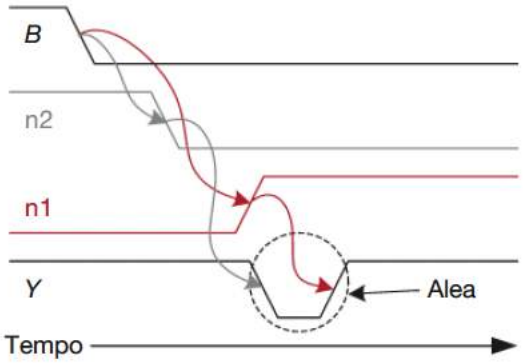
X Y S → X Y S

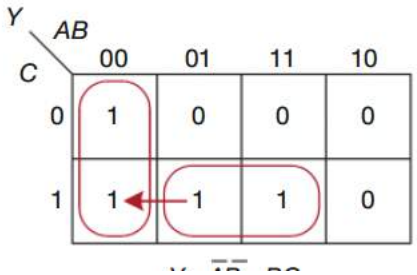
1 1 1 → 1 1 0

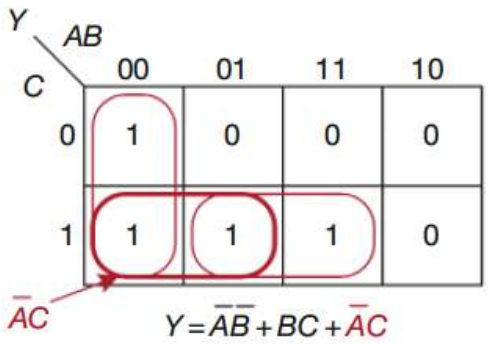
Si verifica poiché S su una strada è negato e nell'altra no.

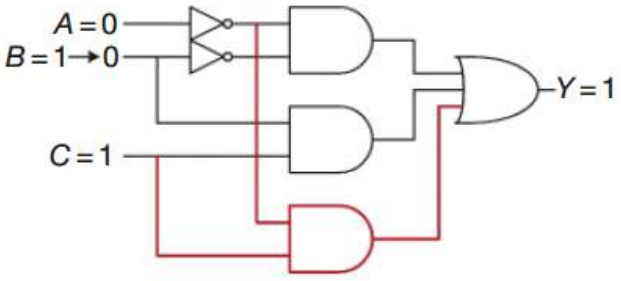



Percorso critico
A = 0
B = 1→0
0→1
n1
Y = 1→0→1
n2
1→0
C = 1
Percorso minimo


B
n2
n1
Y
Alea
Tempo


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


Y
AB
C
00
01
11
10
0
1
0
0
0
1
1
1
1
0
$\overline{A}C$
$Y = \overline{A}\overline{B} + BC + \overline{A}C$


A = 0
B = 1→0
C = 1
Y = 1

Per evitare le alee basta aggiungere un cerchio nelle mappe di K. che connetta le componenti che causano le alee.

Disciplina dei Ritardi

La logica combinatoria è caratterizzata da un ritardo di propagazione e da un ritardo di contaminazione.

- **Il ritardo di propagazione** (Tpd) è il **tempo *massimo*** che trascorre dal momento in cui *avviene* un *cambiamento* nell'ingresso al momento in cui l'uscita (o le uscite) raggiunge il suo valore finale.
- **Il ritardo di contaminazione** (Tcd) è il **tempo *minimo*** che trascorre dal momento in cui cambia un ingresso al momento in cui una qualsiasi uscita *comincia* il processo di adattamento del suo valore.

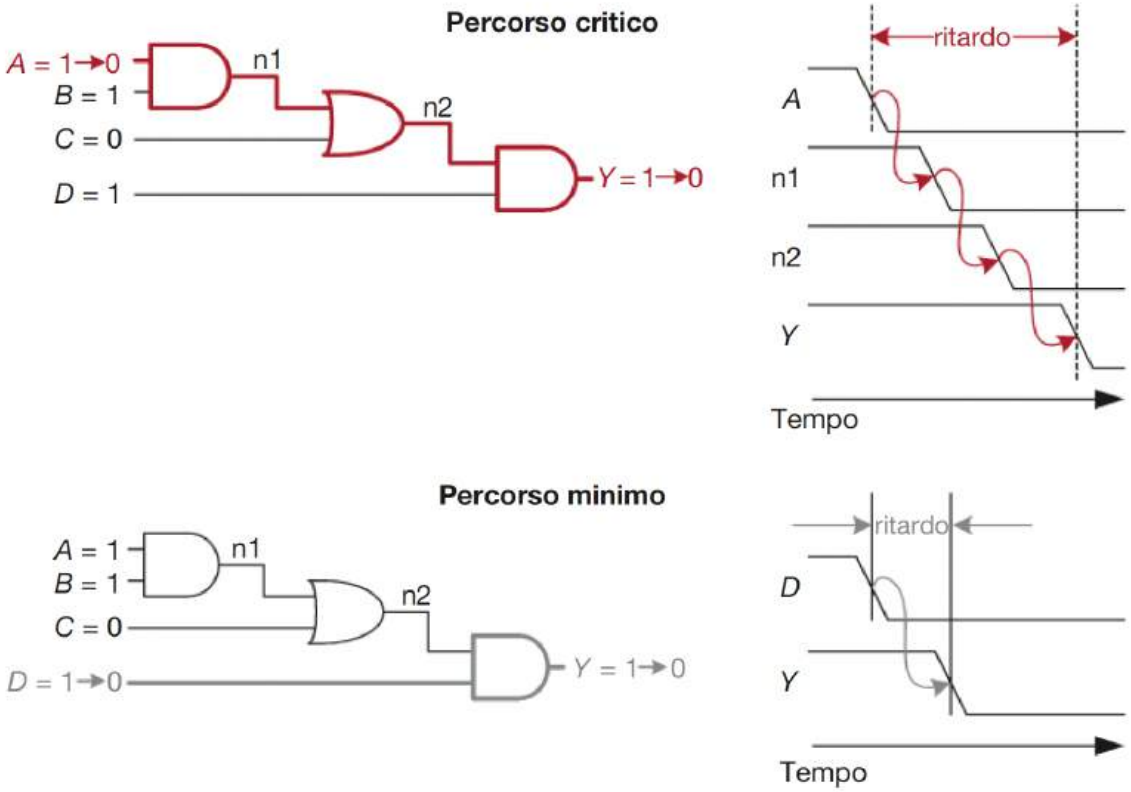
Questi ritardi sono determinati dal **percorso critico**, quello che limita la velocità della rete e dal **percorso minimo** cioè quello più breve possibile.

Il ritardo di propagazione di una rete combinatoria è uguale alla **somma dei singoli ritardi** di propagazione di ogni elemento del **percorso critico**, mentre **il ritardo di contaminazione** è la **somma dei ritardi** di contaminazione attraverso ogni elemento del **percorso minimo**.

Ad ogni modo, i produttori solitamente forniscono schede tecniche (data sheet) che specificano i ritardi di ogni porta.

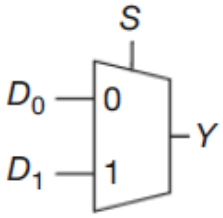
$$t_{pd} = 2t_{pd\_\mathrm{AND}} + t_{pd\_\mathrm{OR}} \quad \textbf{(2.8)}$$

$$t_{cd} = t_{cd\_\mathrm{AND}} \quad \textbf{(2.9)}$$



Componenti combinatorie

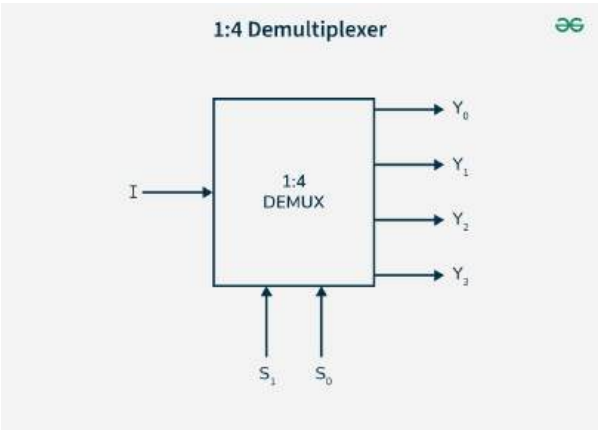
1) Multiplexer (MUX)



S	D_1	D_0	Y
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	0
1	1	0	1
1	1	1	1

Componente in grado di scegliere un'uscita a partire da un certo numero di ingressi possibili, basandosi sul valore di un segnale di selezione o segnale di controllo **S**.
Un multiplexer genera un ritardo di **2Δ***t*.
In realtà i multiplexer possono avere 2^k ingressi, necessitiamo quindi di **k bit di controllo**.
rappresentazione con porte logiche, 2 livelli di logica inducono 2 tp di ritardo.

2) Demultiplexer (DEMUX)



Ho un ingresso e tante **uscite** a **1 bit**. Per **2^k** uscite ho **k** bit di controllo.

- Il demultiplexer copia il valore di entrata in una delle uscite indicata dai bit di controllo.

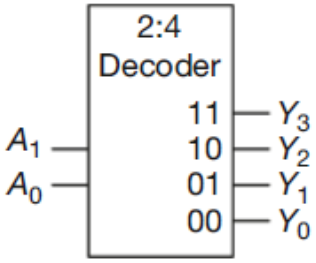
Per meno di 256 uscite il tempo di ritardo di un demultiplexer è **1Δ***t*.

Tabella di verità

I	S1	S0	Y0	Y1	Y2	Y3
0	-	-	0	0	0	0
1	0	0	1	0	0	0
1	0	1	0	1	0	0
1	1	0	0	0	1	0
1	1	1	0	0	0	1

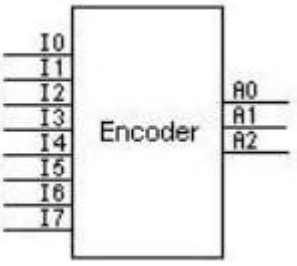
3) Decoder

Un decoder ha **N** ingressi e **2^N** uscite e attiva una delle sue uscite a seconda della combinazione di valori in ingresso (**A** = A_1A_0). Quando **A**=00, $\mathbf{Y_0}$ è 1. Quando **A**=01, $\mathbf{Y_1}$ è 1 e così via. Quindi imposta a "1" l'uscita indicata dall'input.



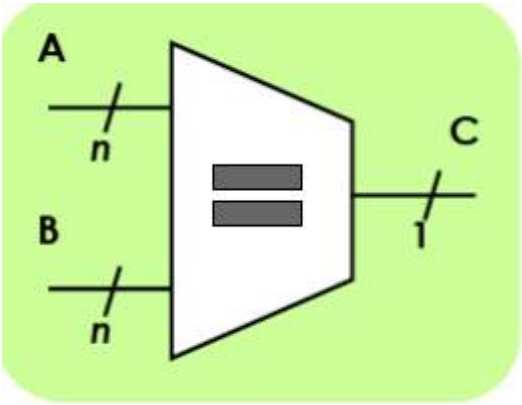
A_1	A_0	Y_3	Y_2	Y_1	Y_0
0	0	0	0	0	1
0	1	0	0	1	0
1	0	0	1	0	0
1	1	1	0	0	0

4) Encoder



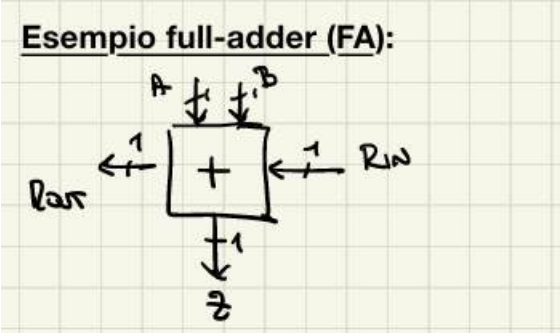
Prende in input **2^n** ingressi che rappresentano una sequenza di bit dove solo un bit è messo ad uno, un encoder ha poi **n uscite** che indicano la **posizione dell'unico bit ad 1**.

5) Confrontatore (=)

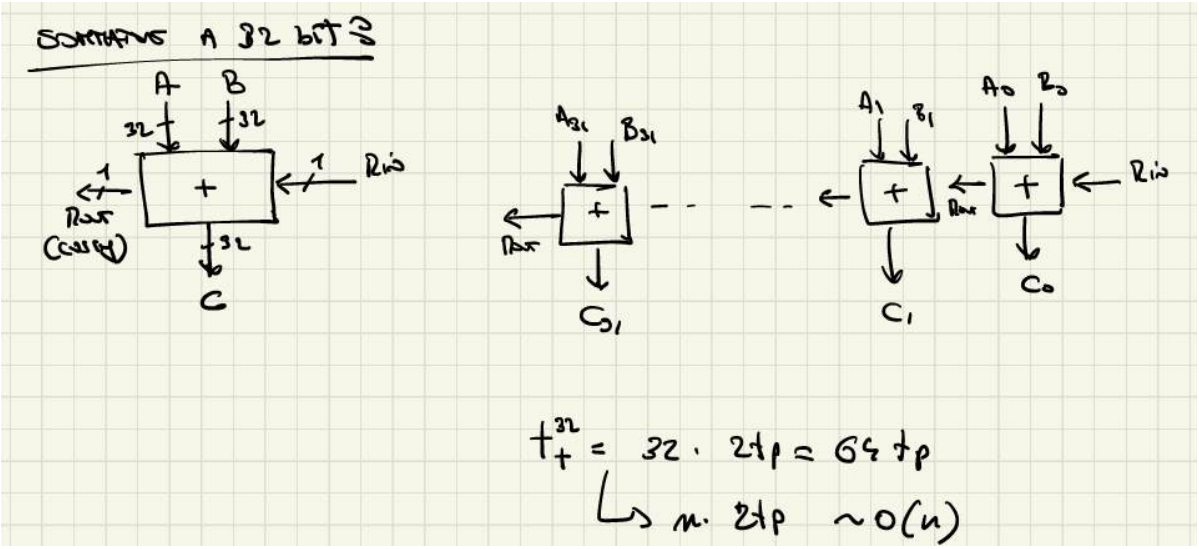


Serve per confrontare **parole** di **n** bit (A, B), produce un output **C** = 1 se le parole sono uguali, 0 altrimenti.

Full Adder

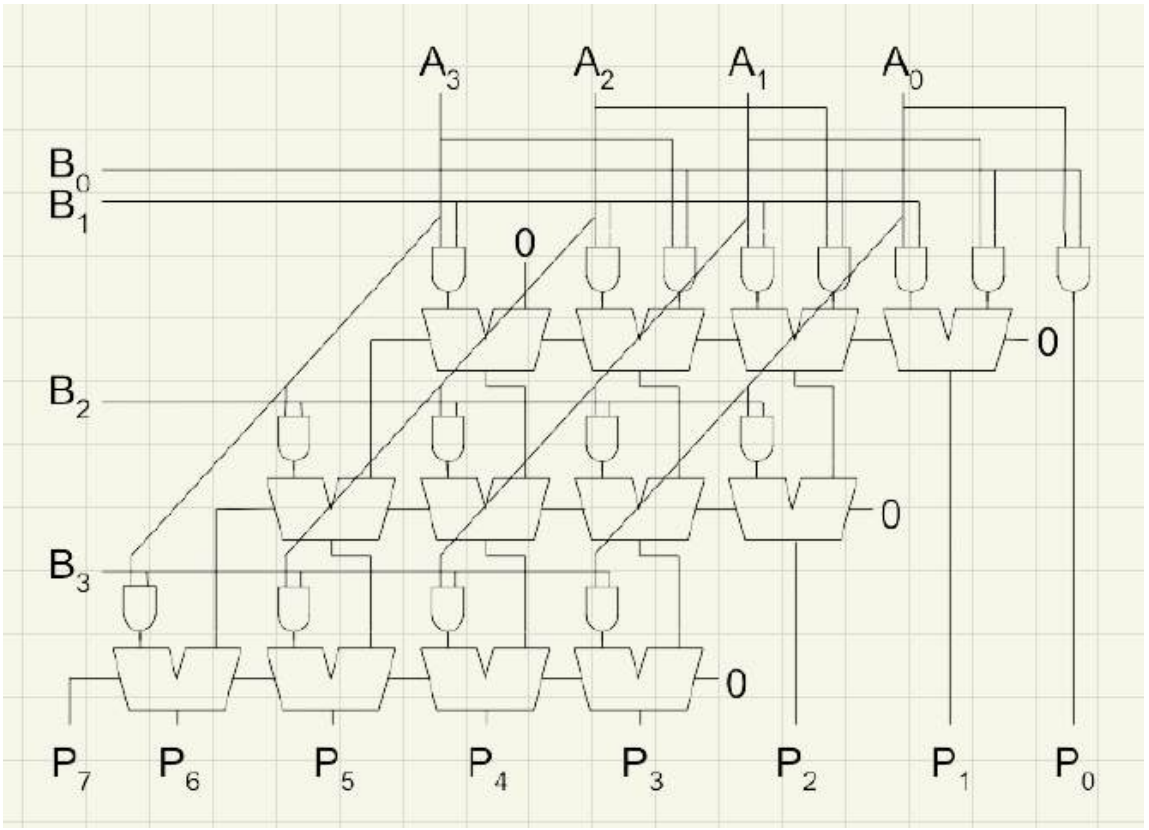


- In ingresso due bit A e B e un 1 bit di riporto in entrata **RIN**.
- In uscita 1 bit per la somma Z e 1 bit per il resto **ROUT**.



Mettendo insieme 32 Full Adder si ottiene un sommatore di due numeri a 32 bit con 1 bit di riporto in uscita. Il ritardo totale è pari a **64 tp** (2 per ogni Full-Adder che deve aspettare il risultato del precedente) quindi proporzionale al numero di cifre degli addendi n.

Moltiplicatore



Dati in ingresso due numeri a **N** bit si produce in uscita il prodotto formato da **2N** bit (in figura N = 4).
Il funzionamento si basa sull' "algoritmo delle elementari":

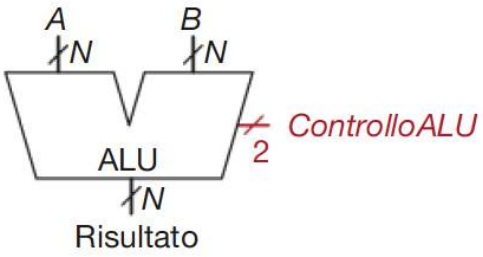
$$
\begin{array}{cccccccc}
 & & A_3 & A_2 & A_1 & A_0 & & \\
\times & & B_3 & B_2 & B_1 & B_0 & & \\
\hline
 & & A_3B_0 & A_2B_0 & A_1B_0 & A_0B_0 & & \\
 & A_3B_1 & A_2B_1 & A_1B_1 & A_0B_1 & & & \\
 & A_3B_2 & A_2B_2 & A_1B_2 & A_0B_2 & & & \\
+ & A_3B_3 & A_2B_3 & A_1B_3 & A_0B_3 & & & \\
\hline
P_7 & P_6 & P_5 & P_4 & P_3 & P_2 & P_1 & P_0
\end{array}
$$

ALU

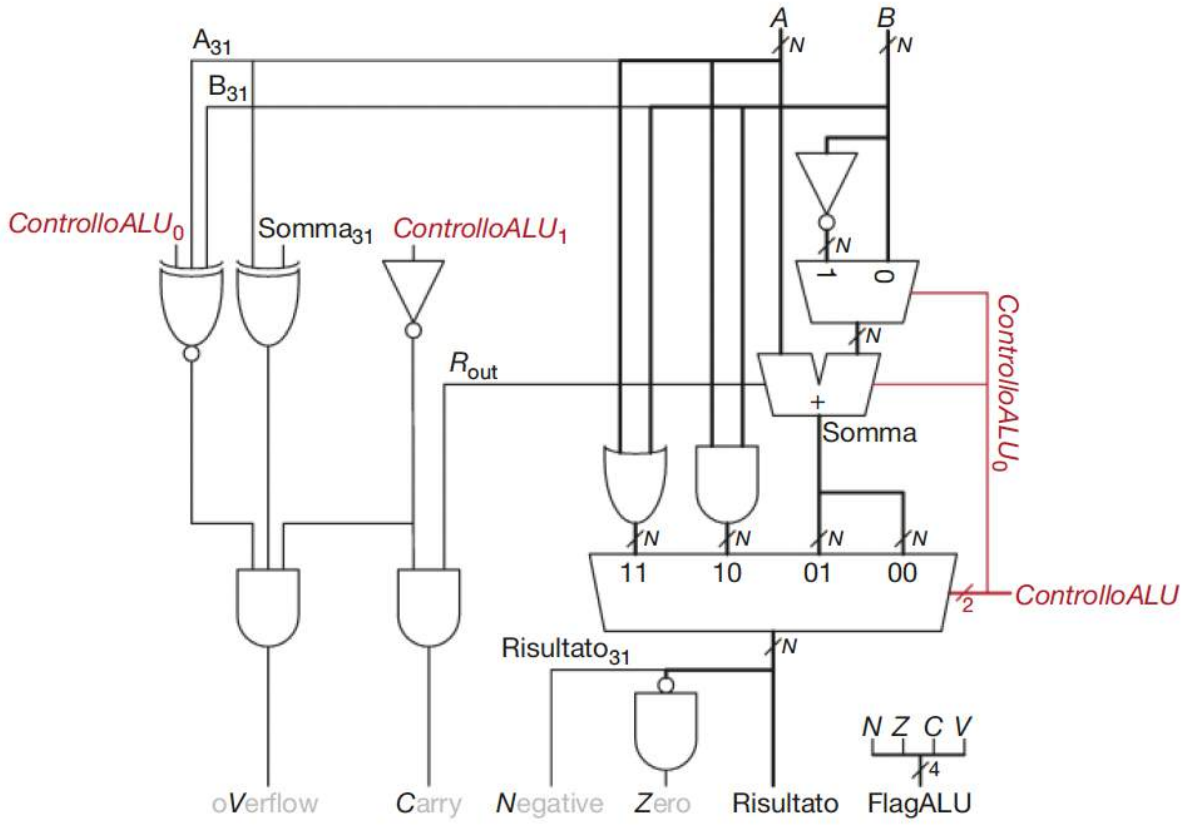
Operazioni svolte dalla ALU (versione semplificata).

$ControlloALU_{1:0}$	Funzione
00	Addizione
01	Sottrazione
10	AND
11	OR

Schema logico della ALU



Schema della ALU in termini di componenti standard di reti combinatorie.



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

C = **1** se il C_{out} del sommatore è a 1 e stiamo facendo una somma o una sottrazione (ALUControl = 00 / 01 quindi $ALUControl_1 = 0$).

Condizioni di overflow (**V** = 1):

Dobbiamo garantire che, dato **N** numero negativo a 32 bit e **P** numero positivo a 32 bit siano soddisfatte le seguenti proprietà:

1. P + P ≠ N
2. N + N ≠ P
3. P - N ≠ N
4. N - P ≠ P

Quindi facciamo un AND a 3 ingressi:

- NXOR a 3 ingressi (vero se tutti gli ingressi sono a “0” oppure se ci sono esattamente 2 bit a “1” e l’altro a “0”). I 3 ingressi sono: $ALUControl_0$ che indica se facciamo somma o sottrazione, A_{31} e B_{31}.
- XOR a 2 ingressi, vero se il bit più significativo del primo operando (A_{31}) è diverso dal bit più significativo della Somma ($Somma_{31}$).
- $ALUControl_1 = 0$ => siamo in una somma o sottrazione.

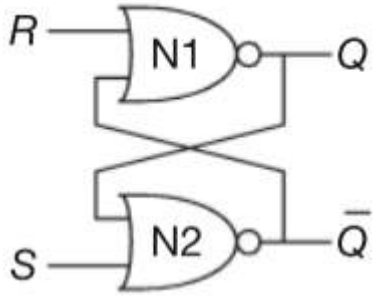
Questi flag vengono salvati in dei registri speciali **CPSR** (Current Program Status Register).

Reti Sequenziali

Reti sequenziali

Una rete sequenziale produce un output sulla base dei parametri in ingresso e della **memoria**.

Latch SR (S = set, R = reset)



Caso	S	R	Q	$\overline{Q}$
IV	0	0	Q_{prec}	$\overline{Q}_{prec}$
I	0	1	0	1
II	1	0	1	0
III	1	1	0	0

In questa tabella di verità notiamo che per input S=0 e R=0 abbiamo il valore di uscita precedente. come facciamo a ricordarlo? Fino ad ora abbiamo visto reti logiche acicliche, per “ricordare” dobbiamo creare un **ciclo**. Per questo per creare il latch sr andiamo a mettere **due NOR a croce**. Questa struttura permette quindi di ricordare 1 bit, per ricordarne **n** basta usare **n** Latch SR.

Problema: quando in **ingresso** ho **R = 1** e **S = 1 perde di senso l’uscita** dato che dovrebbero essere 1 0 o 0 1 invece sono entrambi 0. Per risolvere questo problema introduco il latch D.

Latch-D

Questo latch ha due ingressi: **D** che controlla il prossimo stato e **CLK** (clock) che controlla quando effettuare il passaggio di stato.

Rete logica

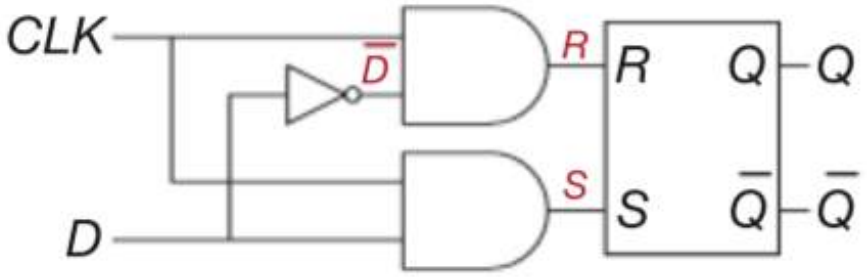


Tabella di verità

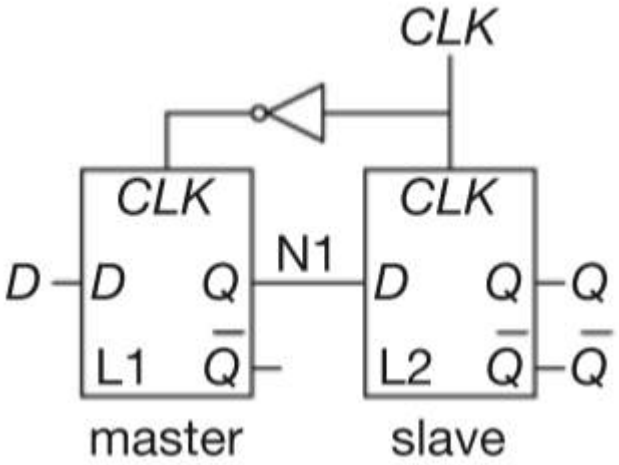
CLK	D	$\overline{D}$	S	R	Q	$\overline{Q}$
0	X	$\overline{X}$	0	0	Q_{prec}	$\overline{Q}_{prec}$
1	0	1	0	1	0	1
1	1	0	1	0	1	0

Il latch D è in grado di evitare il caso anomalo in cui gli ingressi S e R vengano attivati simultaneamente, questo è realizzato grazie al clock, se è basso non succede niente, ma se è altro grazie al **NOT** non potrò avere due ingressi che sono "1".
Quando CLK = 0, il latch è **opaco**: viene bloccato il passaggio dei dati verso Q, che mantiene il valore precedente
Quando CLK = 1 il latch è detto **trasparente**. I dati scorrono da D verso Q, come se il latch fosse un buffer.

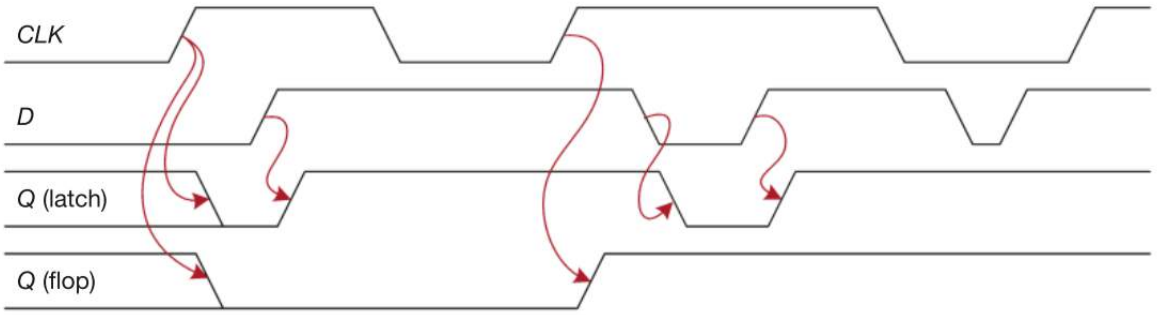
Problema: Il problema di questa componente è che quando il CLK è ad 1 e il latch è trasparente **viene portato in output un qualsiasi cambiamento di D**. Questo non va bene dato che noi vogliamo che quando il CLK = 1 si mantenga costante il valore di uscita, senza che cambi durante il ciclo di clock. Per questo introduciamo il D flip flop.

Flip flop D

Un flip-flop D può essere costruito a partire da **due latch D** in cascata, controllati da due segnali di clock **complementari**. Il primo latch, L1, viene detto **master**, mentre il secondo latch, L2, viene detto **slave**. Quando CLK = 0, il latch master è trasparente, mentre il latch slave è opaco. Di conseguenza, qualsiasi valore di D viene portato a N1. Quando invece CLK = 1, il latch master diventa opaco e quello slave trasparente. In questo caso, il valore di N1 viene trasmesso a Q, ma N1 resta isolato da D. Quindi, qualunque sia il valore di D subito prima del fronte di salita (passaggio da 0 a 1) del clock, questo è il valore che viene trasferito a Q al momento di tale fronte. In tutti gli altri casi, Q mantiene il suo valore precedente, dal momento che c'è sempre un latch opaco che blocca il passaggio di dati tra D e Q.
Questa componente è aperta alla scrittura solo quando ci sono fronti di salita del clock.



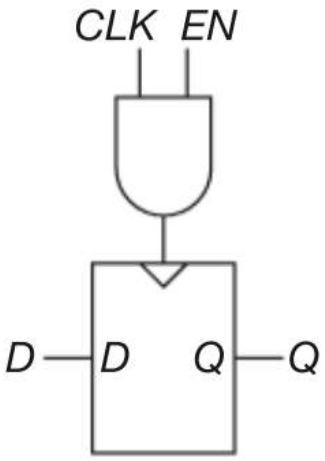
Differenza tra LATCH D e D FLIP FLOP



Nel Latch D quando il clock è alto si riscrivono le alterazioni di D. Nel Flip Flop D invece si scrivono i valori di D solo quando il clock **passa da 0 a 1** (fronte di salita), senza cambiare il valore tutte le volte che di D cambia e il clock è alto.

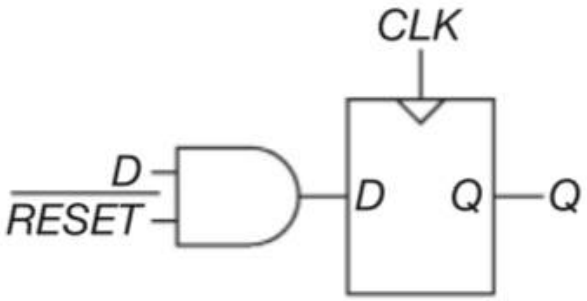
Varianti di flip flop D

Variante con abilitazione



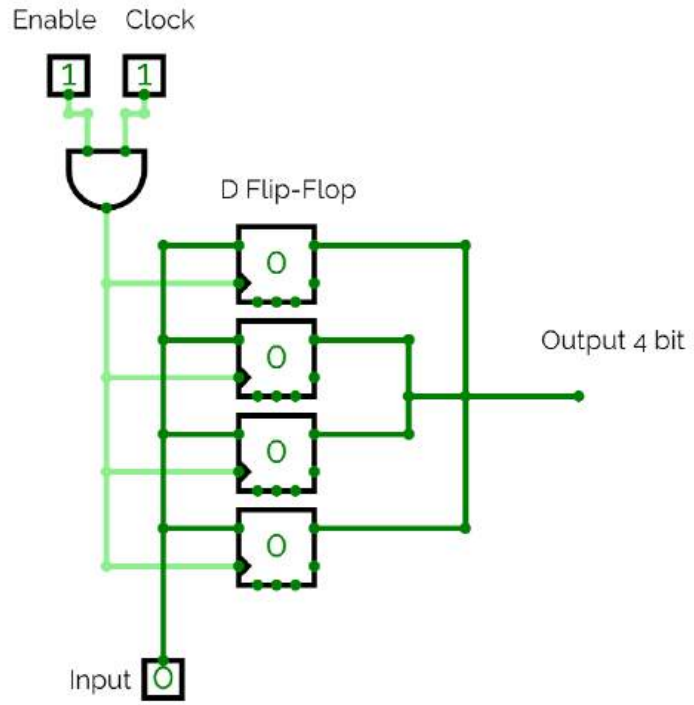
Possiamo modificare il flip-flop precedentemente mostrato aggiungendo un segnale che ci permette di **abilitare** o meno la **scrittura**. In questo modo siamo in grado di ignorare il ciclo di clock a 1 se la EN = 0. Quindi non andremo a cambiare stato per ogni ciclo di clock.
Questo segnale viene implementato aggiungendo un altro ingresso, chiamato EN o **ENABLE**, per determinare se memorizzare o no il dato sul fronte del clock. Quando EN è VERO, il flip-flop con abilitazione reagisce come un normale flip-flop D; quando invece EN è FALSO, il flip-flop con abilitazione ignora il clock e mantiene il proprio stato.

Variante resettabile



Un flip-flop resettabile aggiunge un altro ingresso (**negato**), chiamato **RESET**. Quando l'ingresso RESET è FALSO, il flip-flop resettabile si comporta come un normale flip-flop D. Quando invece RESET è VERO, il flip-flop resettabile **ignora D** e, appunto, resetta l'uscita a **0**. Questa tipologia di flip-flop è utile nel caso in cui si desideri forzare uno stato noto (cioè 0) in tutti i flip-flop della rete quando viene accesa.
Questi flip-flop possono essere resettabili in modo sincrono o asincrono. I flip-flop resettabili in modo sincrono si resettano solo al fronte di salita di CLK, i flip-flop resettabili in modo asincrono si resettano nel momento in cui RESET diventa VERO, indipendentemente dal valore assunto da CLK.

Registri



Per implementare dei registri basta costruire un sistema di **N flip-flop D** che condividono CLK, EN e Input comuni per la scrittura sui registri.

Reti sequenziali sincrone

Una rete sequenziale **sincrona** ha un ingresso di clock i cui fronti di salita indicano una sequenza di **istanti** di tempo nei quali hanno luogo le **transizioni di stato**. Le reti sequenziali con percorsi ciclici possono avere *race condition* o *comportamenti instabili*. Se il clock è abbastanza lento da far sì che tutti gli ingressi dei registri abbiano il tempo di adeguare il proprio valore prima del fronte di clock successivo, tutte le ***race condition*** vengono **eliminate**.

Una rete sequenziale sincrona è formata da elementi circuitali interconnessi tali che:

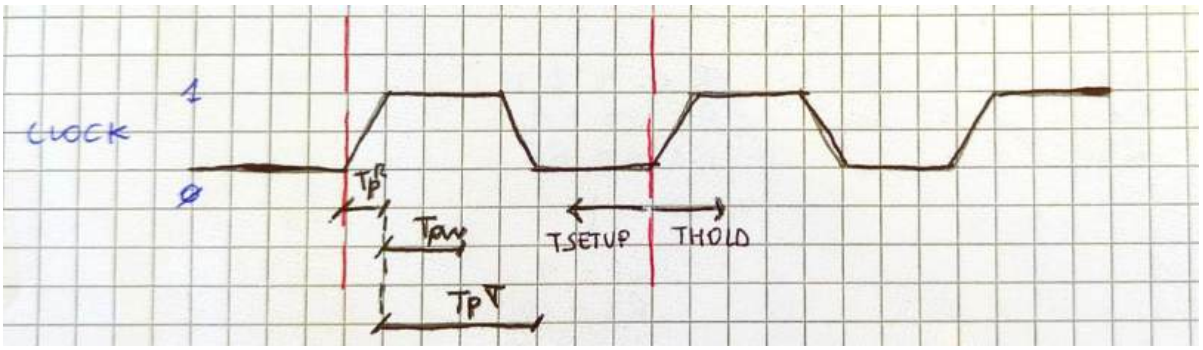
- ogni elemento della rete è o un **registro** o una **rete combinatoria**;
- deve essere presente necessariamente **almeno un registro**;
- tutti i registri ricevono lo **stesso** segnale di **clock**;
- ogni percorso ciclico contiene almeno un registro.

Le reti sequenziali che non sono sincrone sono dette **asincrone**.

Sincronizzatori

Per garantire livelli logici corretti tutti gli ingressi asincroni dovrebbero essere fatti passare attraverso **sincronizzatori**. Il sincronizzatore è un dispositivo che riceve un ingresso asincrono D e un clock, restituendo un valore. Se D è stabile durante il tempo di apertura, Q assume lo stesso valore di D. Se invece D cambia durante il tempo di apertura, Q può assumere un valore ALTO o BASSO, ma non può assumere un valore metastabile. È possibile costruire in modo semplice un sincronizzatore a partire da due flip flop.

Ritardi delle reti combinatorie e dei registri

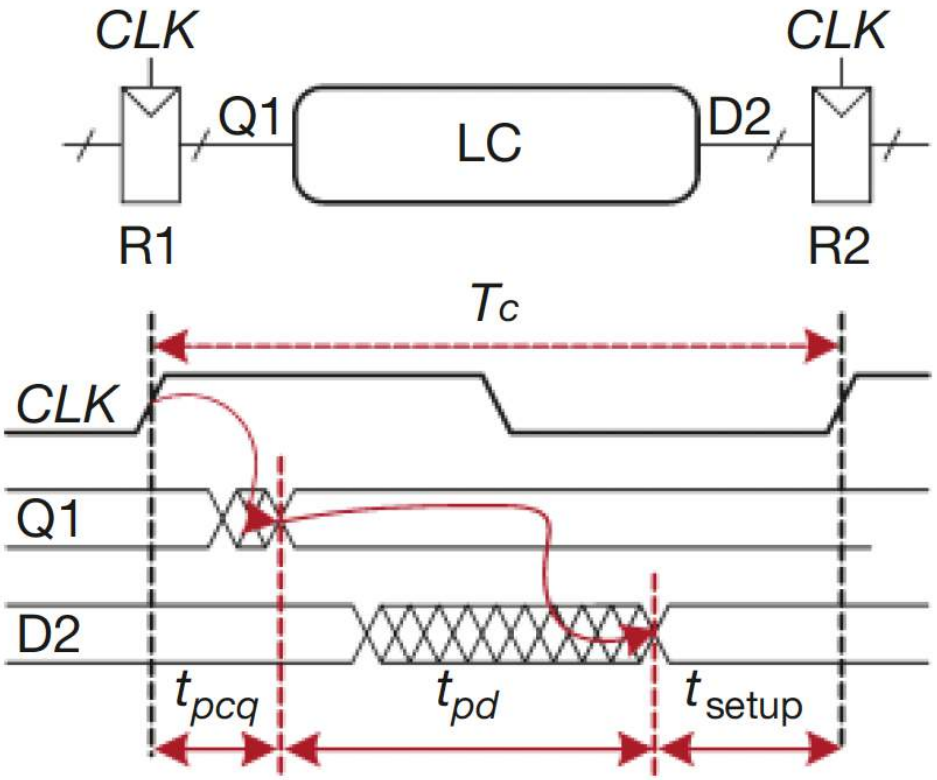


Se l'uscita è stabile a 0 o a 1 quando il clock ha il fronte di salita, il comportamento del flip-flop è definito in modo chiaro, ma cosa accade se l'output sta cambiando nello stesso momento in cui il clock passa da 0 a 1? Quando andiamo a scrivere su un fronte di salita del clock dobbiamo mantenere **stabile** il **valore** in **ingresso** e il **valore** in **uscita**, per questo stabiliamo due tempi:

1) **T-hold** = tempo **di mantenimento**, tempo massimo in cui gli ingressi devono mantenersi stabili. Il T-hold deve essere più piccolo rispetto al tempo in cui la rete sequenziale inizia a produrre un nuovo output, quindi **T-hold <= T-contaminazione.**

2) **T-setup** = tempo di attivazione, quanto tempo prima del prossimo fronte di salita del clock l'**output** della rete sequenziale deve mantenersi stabile.

Questi tempi sono definiti dal produttore della scheda o dei registri e vanno rispettati.



$$T_c \geq t_{pcq} + t_{pd} + t_{\text{setup}}$$

- $\mathbf{T_{pcq}}$ = ritardo interno del **flipflop R1,** tempo nel quale Q assume il valore corretto
- $\mathbf{T_{pd}}$ = tempo di propagazione (percorso critico) della **rete LC**, tempo nel quale **D2** assume il *valore corretto*.
- $\mathbf{T_{setup}}$ = tempo che serve al **registro R2** per prendere il corretto risultato in ingresso nel prossimo fronte di salita del clock.

In generale più semplice è la rete più alta la frequenza di cicli di clock, più alta è la frequenza di clock più operazioni sono in grado di fare.

Modello di una rete sequenziale

Una rete sequenziale rispetto a una rete combinatoria possiede dei registri che gli permettono di **ricordare**. Per questo a partire da una rete sequenziale possiamo costruire degli automi. Possiamo quindi dividere le reti di sequenziali in reti di **Mealy** e di **Moore**. Definiamo le reti sequenziali come automi a stati finiti, **FSM** (Finite State Machine).

Possiamo descrivere una qualsiasi rete sequenziale attraverso due funzioni: una che descrive l'**output della rete** e una che descrivere il **prossimo stato**.

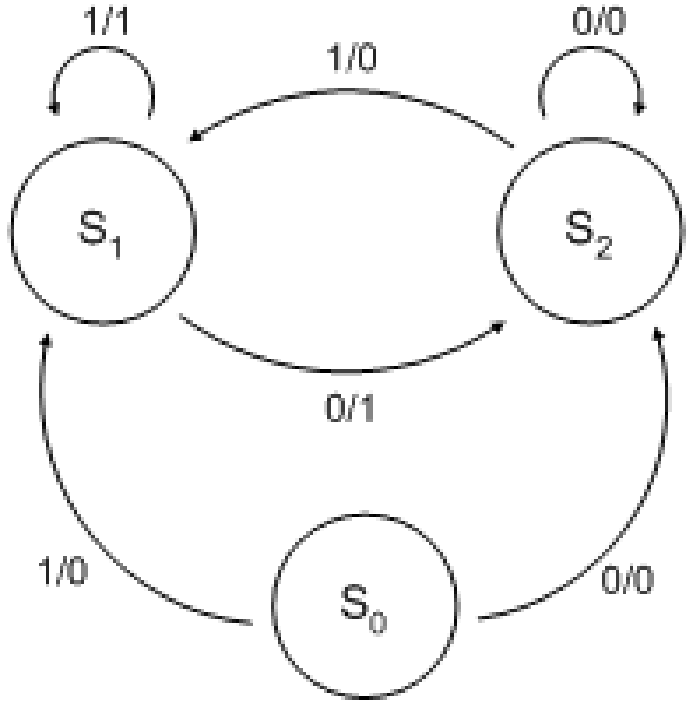
ω: funzione che descrive l'output.

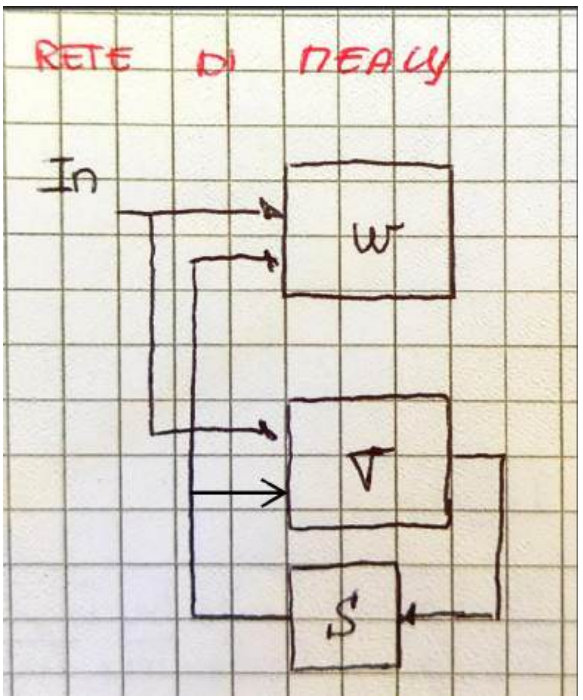
σ: funzione di transizione degli stati.

Rete di Mealy

E' un automa a stati finiti i cui valori di uscita sono determinati dallo **stato attuale** e dall'**ingresso corrente**. (l'output si trova sugli archi).

- **ω** = In * stato_corrente
- **σ** = In * stato_corrente





Esempio calcolo del ciclo di clock con rete di Mealy

Tempo di un ciclo di CLK >= **T_cs + max{ T_c ω, T_cσ + T-setup}**

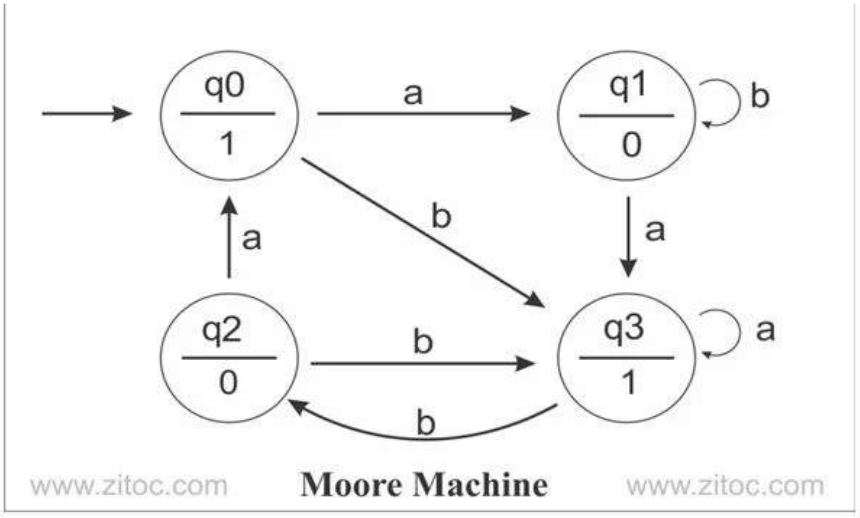
- T_cs è il tempo di propagazione del registro **S**, dopo il quale gli ingressi delle due reti ω e σ sono stabili.
- T_c ω è il tempo di propagazione di Omega, cioè quanto tempo ci metto a calcolare il valore di uscita.
- T_cσ è il tempo di propagazione di sigma.
- T-setup, tempo dovuto al fatto che l'output di sigma si deve stabilizzare nel registro.

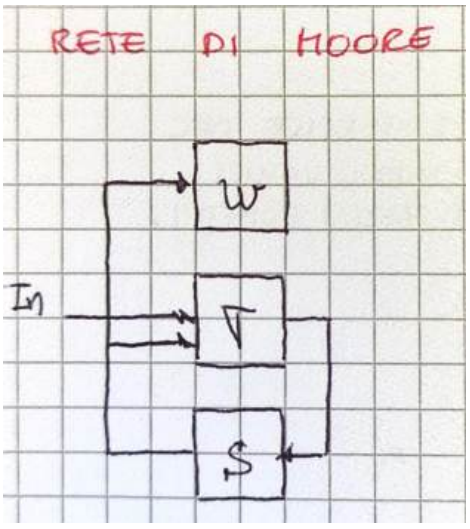
Le reti ω e σ si stabilizzano in parallelo quindi prendiamo il tempo massimo tra i 2.

Rete di Moore

Automa a stati finiti in cui le uscite sono determinate in funzione dei **soli stati correnti**. (l'output si trova negli stati).

- **ω** = stato_corrente
- **σ** = In * stato_corrente





- **NB:** #stati automa di moore >= #stati automa di mealy.

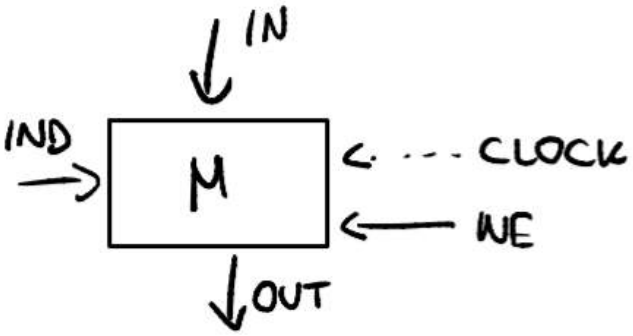
Per entrambi gli automi ricordiamo che:

- **T-hold <= T_cs + T_cσ** (tempo di contaminazione del registro, + ritardo di contaminazione di σ)

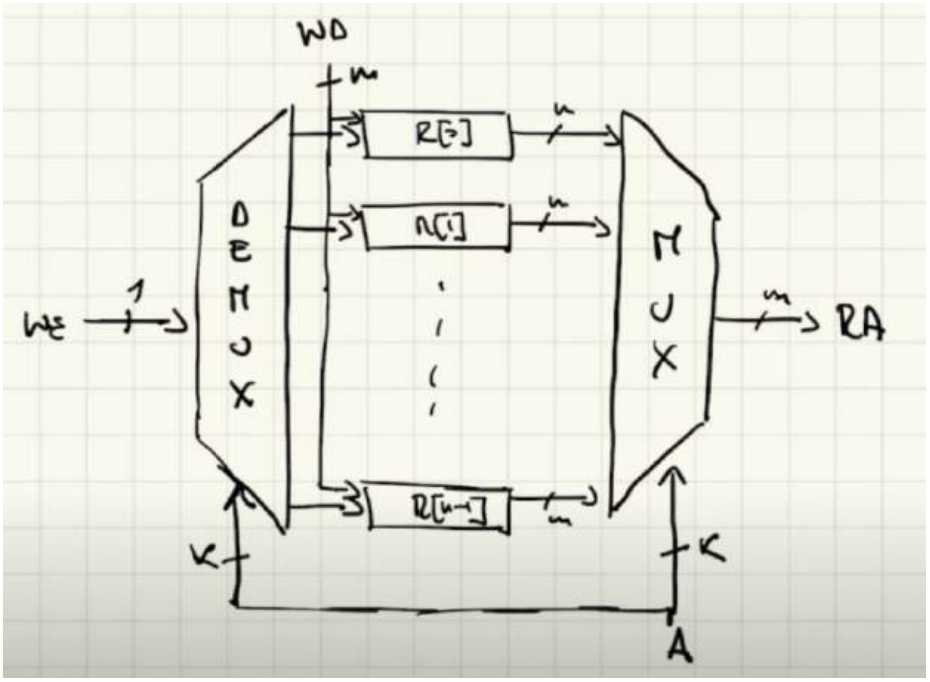
La Memoria

Visione logica

Possiamo vedere la memoria come un vettore, dove ogni entry ha un **indirizzo** e una certa grandezza. Le operazioni sulla memoria che andremo a fare sono quelle di *lettura* e *scrittura.*



Potremmo costruire una memoria con le componenti che abbiamo già studiato:

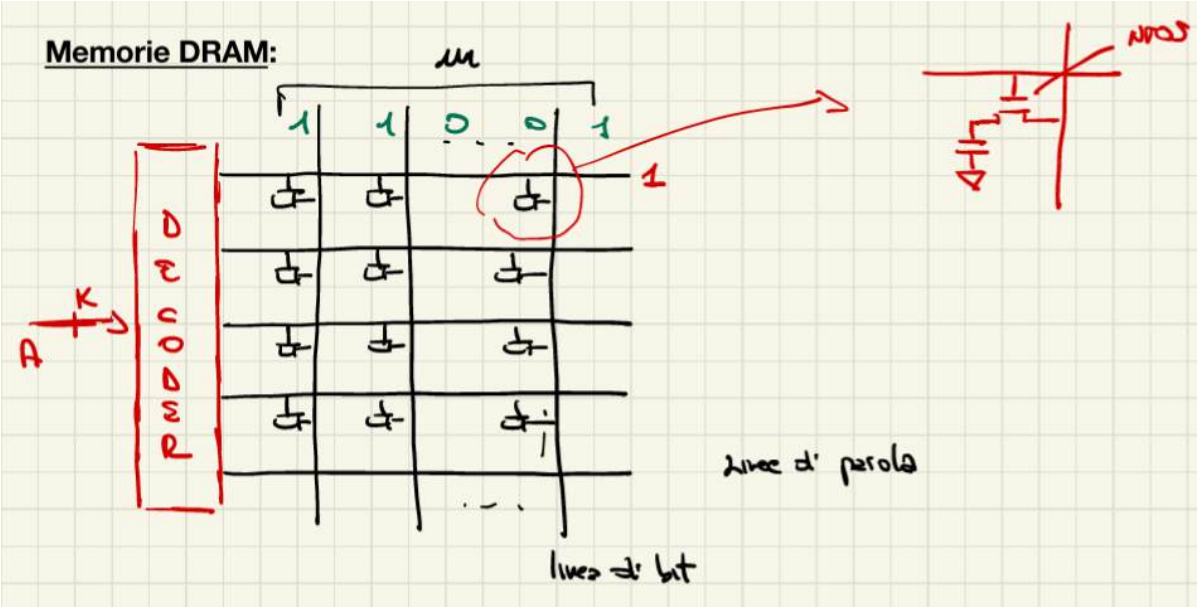


Questa versione non è realizzabile visto il numero di transistor utilizzato per flip flop D, diventerebbe **costosa** e un chip avrebbe **poca** memoria.

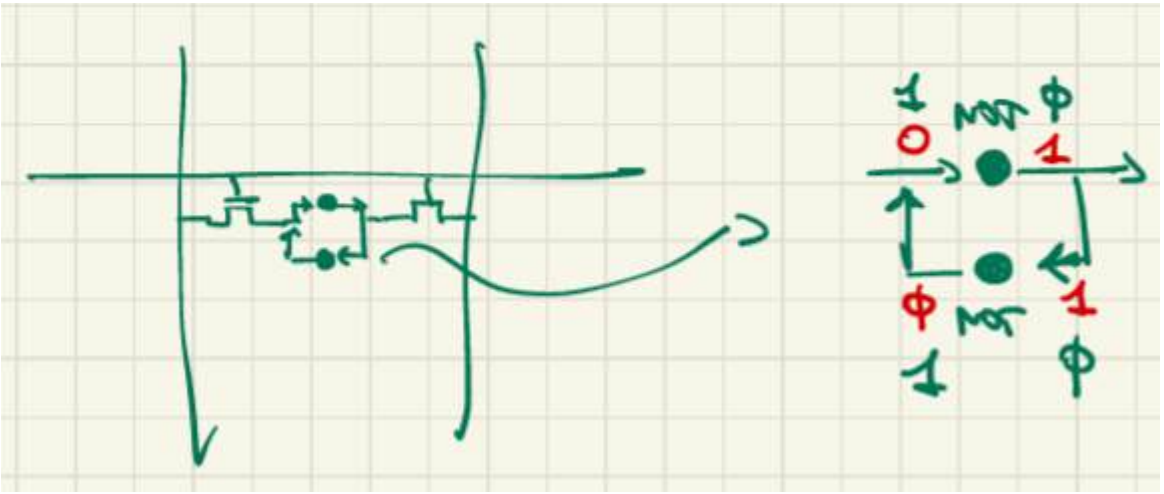
Il tempo di lettura è superiore o uguale al tempo di scrittura.

Le memorie si dividono in RAM e ROM. Le prime sono volatili e ad accesso casuale. Possono essere sia scritte che lette e il costo è il medesimo per ogni locazione di memoria non come i tipi di memorie ad accesso sequenziale, come i nastri. Le **ROM** sono memorie **a sola lettura**, quindi non possono essere riscritte, inoltre queste **non sono volatili**.
Esistono due tipi di **RAM**:
• Dinamiche: Per mantenere il bit nella griglia va "**refreshato**" periodicamente, sennò viene perso. I dati vengono memorizzati come una carica su un **condensatore**.
• Statiche: I bit che andiamo a mettere nella griglia nelle RAM statiche vengono **mantenuti finché alimentati**.

Memorie DRAM (*Dynamic Random Access Memory*)
Le DRAM hanno dei piccoli condensatori che trattengono le informazioni. Questi condensatori hanno bisogno di essere "refreshati" continuamente per mantenere la carica a 0 o 1 (dopo un po' perdono la carica). Le linee orizzontali sono anche dette **linee di parola**, mentre quelle verticali sono le **linee di bit**. Quando inserisco un indirizzo nel decoder vado a prendere una linea di parola che fa scorrere le informazioni della parola attraverso la linea di bit: questo produce in output il valore letto in quella locazione di memoria. Per la scrittura serve un dispositivo che metta sulle linee di bit i valori che si vogliono scrivere, in seguito tramite il **decoder** viene messa a 1 soltanto la linea di parola relativa all'indirizzo in cui vogliamo scrivere (Perchè se c'è tensione il condensatore riscrive il valore, mentre se non c'è tensione non succede nulla).



Memorie SRAM (*Static Random Access Memory*)



Usando **due porte NOT a croce** si può conservare un bit come con il condensatore, senza bisogno di refresh, quindi con un **meccanismo più veloce**. Si hanno prestazioni maggiorate rispetto alla DRAM ma serve un **numero di transistor maggiori** (6, di cui 4 per il NOT, invece che 1 transistor e 1 condensatore come sopra).

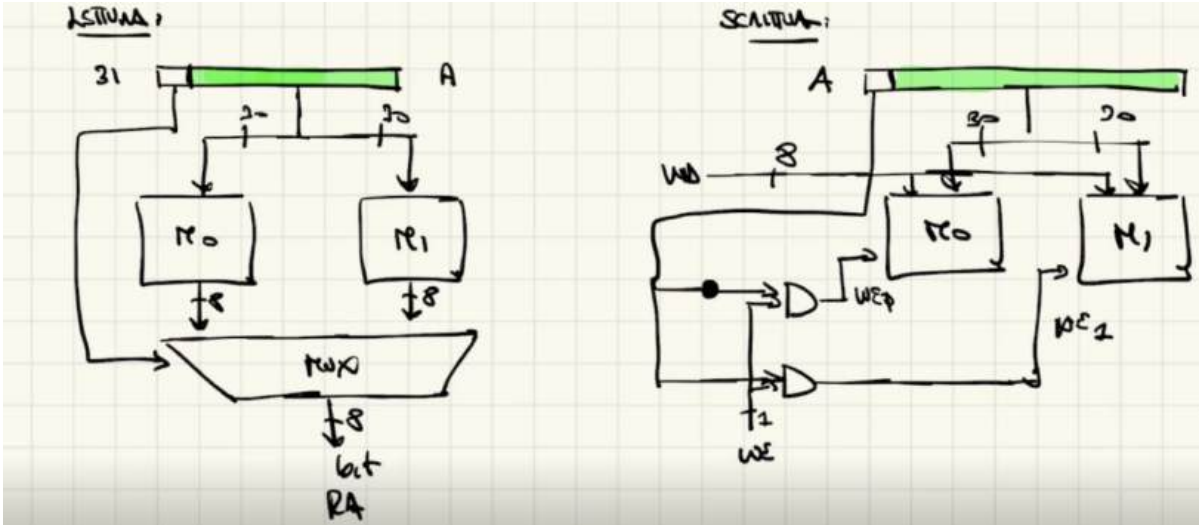
In sintesi:

- Flip-Flop D: memorizza un bit in modo stabile finchè c'è alimentazione, robusto, veloce, non ha bisogno di refresh, ma occupa molto spazio e consuma di più. Ideale per logica e registri interni della CPU, utilizzato nei registri temporanei, memorie SRAM.
- Condensatore: memorizza la carica (0 o 1) per un tempo limitato (qualche millisecondo). Più semplice e denso, permette di avere moltissimi bit in poco spazio però ha bisogno di refresh periodico per non perdere il dato, quindi più lento. Ideale per memorie grandi ma non per registri logici. Usato in memorie DRAM, quindi nella RAM principale.

Memoria Modulare

Si tratta di una memoria **suddivisa** in moduli: questo mi permette di comporre una memoria grande a partire da moduli di memoria più piccoli. Esistono due tipi di approcci per indirizzare la memoria modulare:

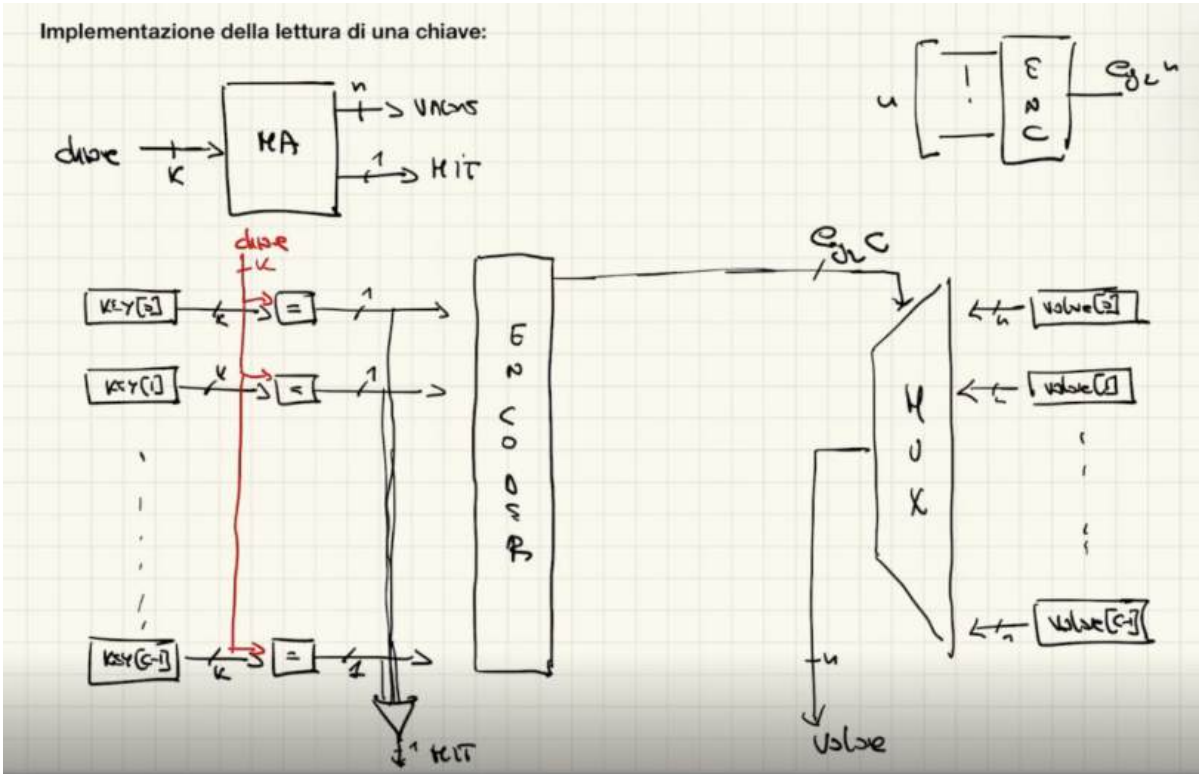
- **Sequenziale** : se ho N moduli per capire in che modulo guardare mi baso sui *sup(log2(n))* **bit più significativi**. In questo modo posso effettuare operazioni di lettura e scrittura sapendo a che modulo fare riferimento. modulo = indirizzo % #moduli.



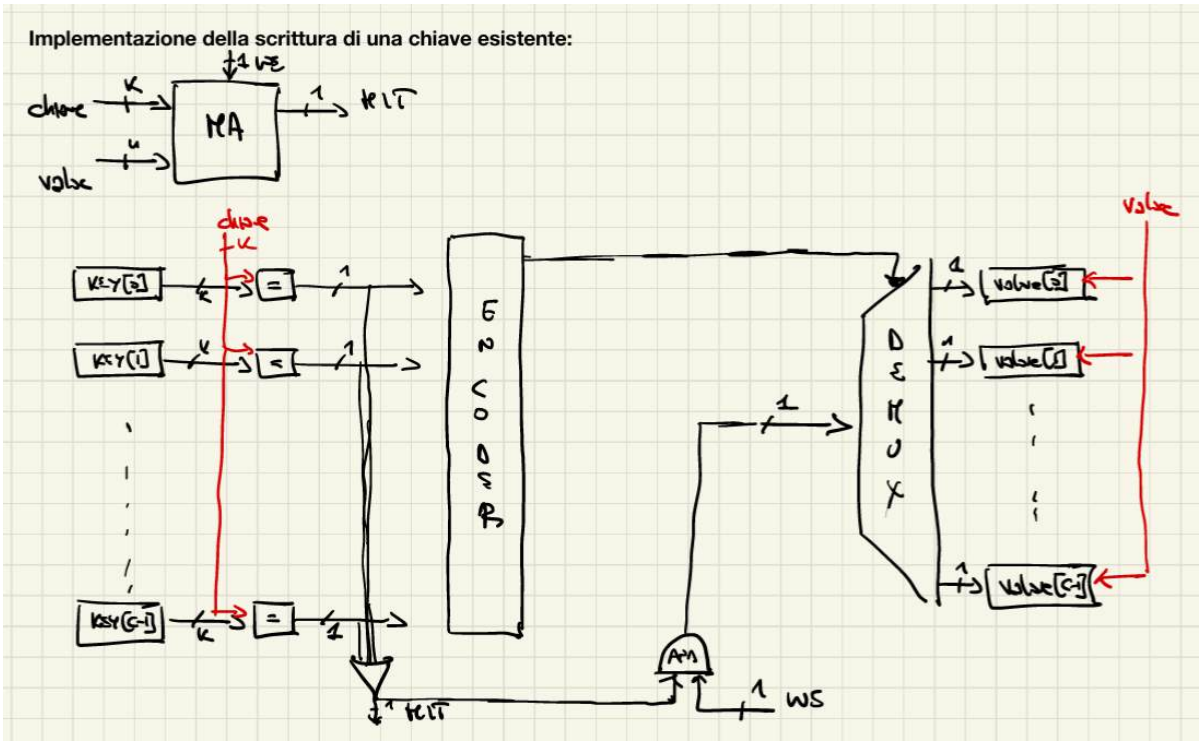
- **Interlacciata**: Se ho un mazzo di 10 carte e 2 giocatori posso dare al primo giocatore 5 carte e al secondo altre 5 oppure dare le carte una al primo una al secondo, una al primo una al secondo… allo stesso modo vengono distribuite le locazioni di memoria nell'organizzazione interlacciata. Con questo tipo di memoria per capire a quale modulo fare riferimento guardo il *sup(log2(n))* **bit meno significativi** dell'indirizzo in memoria. Il vantaggio di questo approccio è che posso fare delle letture/scritture in **parallelo** per parole contigue.

Memoria Associativa

Una memoria associativa è una memoria che immagazzina un certo numero di **coppie (chiave, valore)**. Questo ci permette di leggere e scrivere una certa locazione di memoria indirizzata con una chiave. Ovviamente quando vado a leggere/scrivere la chiave potrebbe anche non esistere, per questo oltre all'uscita del valore letto in quella locazione di memoria avrò anche un bit di **HIT** che mi dice se il valore letto è attendibile o meno.



Per la **lettura** inserisco una chiave e **n** confrontatori quante sono le chiave in memoria: il risultato dei confrontatori viene mandato a un encoder che traduce l'unico bit a "1" in una posizione che data in input al **multiplexer** ci dirà quale registro leggere. Potrebbe esserci il caso in cui io non ho una risposta attendibile, dato che la chiave inserita potrebbe non andare bene: per questo devo anche mettere un uscita per il bit di HIT.



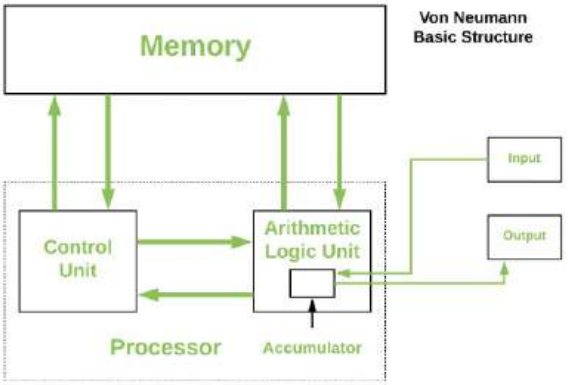
La **scrittura** funziona allo stesso modo con l'aggiunta di un AND che prende in input il bit di **HIT** e il bit di **WE** (Write Enable) e trasmette il suo output ad **demultiplexer** che mi andrà a scrivere nella locazione di memoria fornita dall'encoder.

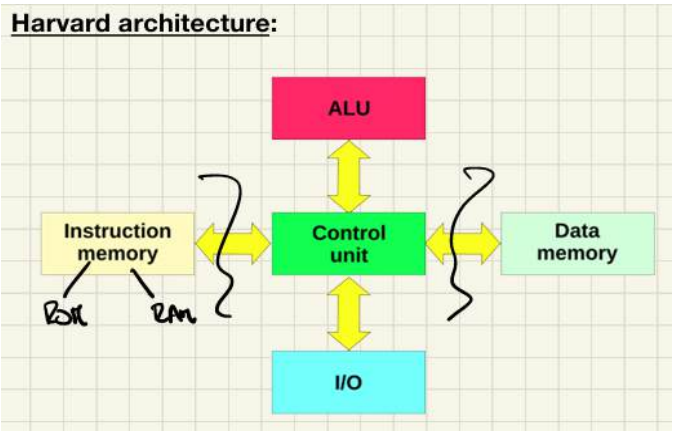
Architetture

Architetture

Intro

Von Neumann architecture vs Harvard architecture
Rappresentano modelli concettuali di architetture e si differenziano per la gestione della memoria: Von Neumann usa una memoria sola per istruzioni e dati mentre il modello di Harvard le separa.





Cosa rappresenta l'architettura di un calcolatore
Riguarda l'organizzazione e la progettazione a livello logico del sistema. Include:

- Il set di istruzioni (ISA, Instruction Set Architecture): le istruzioni che la CPU può eseguire (es. add, mov, jmp, ecc.).
- La tipologia e dimensione dei registri.
- Il formato dei dati (interi, float, ecc.).
- Le modalità di indirizzamento della memoria.
- Il comportamento del sistema durante l'esecuzione dei programmi.

📌 **È l'interfaccia tra hardware e software.**

Tipi di architetture:

- **RISC** = Reduced Instruction Set Computer => insieme di istruzioni semplici, per fare operazioni complesse ho bisogno di tante istruzioni (es: in ArmV7 per caricare in

memoria la somma di due valori presenti nello Stack servono 4 istruzioni - 2 load, 1 add e 1 store)

- **CISC** = Complex Instruction Set Computer => istruzioni complesse di base (nell'esempio di prima potrei avere una "add-memory" che fa tutto).

La **codifica** è un modo simbolico per rappresentare il linguaggio macchina. Quasi sempre un istruzione corrisponde a **32 bit** (ArmV7).

Esistono due tipi di istruzioni in ArmV7:

- **ARM (32 bit)**: istruzioni standard, più potenti.
- **Thumb (16 bit)**: più compatte, per ridurre spazio in memoria
  - Istruzioni tutte a **16 bit** di lunghezza fissa
  - Coprono un sottoinsieme delle operazioni ARM (principalmente data-processing, load/store, branch…)
  - Registri limitati: solo R0–R7 direttamente indirizzabili, condizioni di flag ridotte.

Alla fine di ogni ciclo si aumenta il Program Counter (PC) di 4 (4 byte = 32 bit = 1 istruzione). Tutte le istruzioni Fetch, Decode, Execute…sono reti combinatorie. Indirizzamento in memoria al byte.

Esistono diversi standard per descrivere in quale ordine i dati vengono presi

- **Big Endian:** Il **byte più significativo (MSB)** viene memorizzato **per primo** (cioè all'indirizzo di memoria più basso).
- **Little-Endian**: Il **byte meno significativo (LSB)** viene memorizzato **per primo** (indirizzo più basso). È l'ordine usato dai processori x86 (Intel/AMD) e da molte architetture **ARM** in modalità default.

I registri

I registri utilizzati da ArmV7 sono divisi in **registri generali** e il **registro di stato**.

I registri generali sono 16 e sono suddivisi in:

Nome	Utilizzo
R0	Parametro/valore da restituire/variabile temporanea
R1-R3	Parametri/variabili temporanee
R4-R11	Variabili salvate
R12	Variabile temporanea
R13 (SP)	*Stack Pointer*
R14 (LR)	*Link Register*
R15 (PC)	*Program Counter*

- **Stack Pointer**: puntatore alla prima cella vuota dello stack.

- **Link Register**: contiene l'indirizzo di ritorno per le chiamate di funzione e le eccezioni. Quando viene chiamata una subroutine, l'indirizzo di ritorno viene memorizzato qui.
- **Program Counter**: mantiene l'informazione riguardo l'istruzione corrente.

Registro di stato:

- **CPSR (Current Program Status Register)**: Memorizza i flag di condizione (**N**, **Z**, **C**, **V**).

Alla fine di ogni programma si scrive *mov pc, lr*, per mettere il valore del link register all'interno del program counter, così da ritornare al chiamante della funzione.
Quando si fanno più Branch & Link si salva il link register pushandolo nello stack (*push {lr}*).

Classi di operazioni in ArmV7:
- Operative
- Memoria (Load Literal, LDR, STR)
- Salti (Branch…)
- Speciali

Stack

Lo stack è una struttura dati che funziona come una pila **Last in First out** che può crescere verso il basso - **Descending** - o verso l'alto - **Ascending**. ArmV7 permette di usare lo stack in entrambe le versioni se specificato. Operiamo nello stack con le operazioni di POP e PUSH che sono uno pseudonimo di LDRM e STRM ovvero load e store multiple.
Per lavorare sullo stack serve uno **stack pointer**, questo può puntare o alla prima posizione vuota o alla prima posizione piena.
Quindi se ad esempio lavoriamo con lo stack descending (dovrebbe essere quello di default di ArmV7):

- Quando si esegue **push**, **sp** **viene decrementato** prima di scrivere (stack "discendente").
- Quando si esegue **pop**, **sp** **viene incrementato** dopo aver letto.

Istruzioni

Tutte le istruzioni possono avere una **label** che può servire per richiamare delle etichette all'interno del programma.

Operative

Sono quelle che ci permettono di eseguire calcoli e operazioni sui dati il primo registro rappresenta il registro nel quale andremo a scrivere il risultato e gli altri due registri sono i membri dell'operazione

Operazionali

ADD

ADD R0, R1, R2 ;-> R0 = R1 + R2
Somma i dati contenuti nel registro r1 e r2 e li va scrivere in r0

SUB

SUB R0, R1, R2   ; -> R0 = R1 – R2
Sottrae r1 e r2 e scrive il risultato in r0

MUL

MUL R0, R1, R2 ; ->R0 = R1 * R2
Moltiplica r1 e r2 e scrive in r3

DIV

SDIV R0, R1, R2 ; -> R0 = R1 / R2 (divisione intera)
Divide R1 per R2 e scrive in R0

Logiche

AND

AND R0, R1, R2 ; R0 = R1 AND R2
Effettua un'operazione AND bit a bit tra due operandi. I bit risultanti sono 1 solo se entrambi i bit degli operandi sono 1.

ORR

ORR R0, R1, R2 ; R0 = R1 OR R2
Effettua un'operazione OR bit a bit tra due operandi. I bit risultanti sono 1 se almeno uno dei bit degli operandi è 1.

LSL (moltiplico per 2^k)

LSL Rd, Rn, #shift
Logical Shift Left - Effettua un'operazione di shift logico a sinistra di #shift bit sul valore Rn e scrive il risultato in Rd.

LSR

Divisione per 2
LSR Rd, Rn, #shift
Logical Shift Right - Effettua un'operazione di shift logico a destra di #shift bit sul valore Rn e scrive il risultato in Rd.

ROR

ROR Rd, Rn, #shift
ruota i bit di un registro a destra. A differenza dello spostamento logico o aritmetico, la rotazione "riporta" i bit persi dal lato destro al lato sinistro del registro.

ASR

ASR Rd, Rn, #shift
È particolarmente utile per dividere numeri interi con segno (in rappresentazione binaria complemento a due) per una potenza di due.

ESEMPIO

- **RN** è 11111111.
- **RM** è 00001111.
- **RD** diventerà 11110000, perché i bit più bassi (dove **RM** ha 1) sono azzerati in **RN**.

Memoria

- **Rd**: Registro di destinazione o lettura.
- **Rn**: Registro che contiene l'indirizzo base da cui leggere.
- **offset**: Valore opzionale (costante o contenuto in un registro) che specifica un offset da aggiungere all'indirizzo base.
  Usare sempre offset che siano multipli di 4 quando si accede a dati a 4 byte consecutivi.

LDR
LDR Rd, [Rn, offset]
distinguiamo 3 tipi di load e analogamente esistono anche per l'istruzione STR. Queste sono le principali caratteristiche.

Istruzione	Cosa carica	Dimensione	Zero/sign extension	Quando usarla
LDR/STR	**Word (4 byte)**	32 bit	Nessuna	Leggi/scrivi un intero o puntatore
LDRB/STRB	**Byte (1 byte)**	8 bit	Esteso con zeri	Leggi/scrivi un carattere ASCII
LDRH/STRH	**Halfword (2 byte)**	16 bit	Esteso con zeri	Leggi/scrivi metà di un intero (es. char 16-bit o dati binari compatti)
LDM/STM	**{registri}**	32 * #registri	Nessuna	Legge/scrive nei registri

- **Indirizzamento diretto**: LDR R0, [R1] =>R1 contiene l'indirizzo di memoria da cui leggere.
- **Indirizzamento con offset immediato**: LDR R0, [R1, #offset] => L'indirizzo è calcolato sommando un offset a R1.
- **Indirizzamento post-incremento**: LDR R0, [R1], #offset => Prima carica il valore in R0 corrispondente a R1 e poi incrementa R1 con l'offset, aggiornandolo.
- **Indirizzamento pre-incremento**: LDR R0, [R1, #offset]! => Incrementa R1 , modificandone il valore sommando offset, e poi carica in R0.

Anziché scrivere come offset multipli di 4 essendo i dati salvati in 32 bit si può utilizzare:
[R0, R1, LSL #2] che **significa R1 * 4**.

ESEMPI
- LDR R0, [R1] => Carica in R0 il valore all'indirizzo contenuto in R1
- LDR R0, [R1, #4] => Carica in R0 il valore all'indirizzo (R1 + 4)
- LDR R0, [R1, R2] => Carica in R0 il valore all'indirizzo (R1 + R2)

STR
STR Rd, [Rn, offset]

ESEMPI
- STR R0, [R1] => Salva il contenuto di R0 all'indirizzo contenuto in R1
- STR R0, [R1, #4] => Salva il contenuto di R0 all'indirizzo (R1 + 4) PRE INDEX
- STR R0, [R1], #4 POST INDEX
- STR R0, [R1, R2] => Salva il contenuto di R0 all'indirizzo (R1 + R2)

MOV
MOV Rd, operando
- **Rd**: Registro di destinazione che riceve il valore.
- **operando**: Può essere un valore immediato (#valore) o il contenuto di un altro registro.

ESEMPI
- MOV R0, #5 => Carica il valore 5 in R0
- MOV R1, R0 => Copia il valore di R0 in R1

LDM/STM (LoaD/STore Multiple)

Es: STMFA SP!, {r0, r1, r2}

Presentano quattro versioni per stack pieni o vuoti, ascendenti o discendenti (**FD**, Full Descending; **ED**, Empty Descending; **FA**, Full Ascending; **EA**, Empty Ascending).

- **Ascending**: cresce verso indirizzi di memoria più **alti**.
- **Descending**: cresce verso indirizzi di memoria più **bassi**.

- **Full**: lo SP punta a un **elemento già usato** (l'ultimo dato presente).

- **Empty**: lo SP punta al **prossimo slot libero**.

La notazione **SP!** nell'istruzione indica di salvare i dati relativamente a **SP** e di **modificare opportunamente SP** dopo il salvataggio o il ripristino, per riflettere la nuova posizione in memoria.

PUSH {r0, r1, r2} = STMFD SP!, {r0, r1, r2}
Salva r0, r1 e r2 nello stack, **Full Descending** quindi andando a scendere all'interno nello stack pieno. SP viene decrementato.

POP {r0, r1, r2} = LDMFD SP!, {r0, r1, r2}
Carica i valori dallo stack nei registri r0, r1 e r2. SP viene incrementato

Salti

Le istruzioni di salto condizionale vengono implementate quasi sempre grazie all'utilizzo di CMP.

B
B etichetta
Questa istruzione salta a un'etichetta specifica senza alcuna condizione. (l'etichetta si rappresenta nel codice con etichetta:)

BX
BX Rn
utilizzata per saltare a un indirizzo specifico (contenuto in un registro)

BL
BL etichetta
Salta all'etichetta e salva nel link register il comando successivo, quindi permette di chiamare una funzione e di ritornare al comando successivo della BL.

ESEMPIO

```
MOV R0, #5 ; Imposta R0 a 5
MOV R1, #10 ; Imposta R1 a 10
BL somma ; Chiama la subroutine "somma"
Il programma continua qui dopo la subroutine

somma: ADD R2, R0, R1 ; Esegue R2 = R0 + R1
       BX LR
```

⚠️Al posto di fare Branch & Link si può ottenere lo stesso risultato manipolando PC e LR:

MOV LR, PC @salviamo il valore del **PC + 8** in LR (viene sommato in automatico), così che quando si ritorna in questo codice si passi alla prossima istruzione, quella dopo questa (+4) e dopo il salto (+4).

MOV PC, R2 @effettuiamo il salto vero e proprio, mettendo nel PC il puntatore a un'altra funzione in R0.

Condizionali

CMP

CMP R0, R1 => Confronta R0 con R1
Confronta due operandi sottraendoli, in base al risultato aggiorna i **flag** come una SUB che non salva il risultato.

NOTA BENE: possiamo rendere condizionali tutte le istruzioni aggiungendo una S in fondo all'istruzione.

In generale

Possiamo rendere una istruzione opzionale o comunque eseguirla solo se rispetta una certa condizione aggiungendo uno di questi suffissi che prima di eseguire l'istruzione, verifica i registri di stato CPSR.

Code	Suffix	Flags	Meaning
0000	EQ	Z set	equal
0001	NE	Z clear	not equal
0010	CS	C set	unsigned higher or same
0011	CC	C clear	unsigned lower
0100	MI	N set	negative
0101	PL	N clear	positive or zero
0110	VS	V set	overflow
0111	VC	V clear	no overflow
1000	HI	C set and Z clear	unsigned higher
1001	LS	C clear or Z set	unsigned lower or same
1010	GE	N equals V	greater or equal
1011	LT	N not equal to V	less than
1100	GT	Z clear AND (N equals V)	greater than
1101	LE	Z set OR (N not equal to V)	less than or equal
1110	AL	(ignored)	always

Best practice

- La chiamata di funzione avviene tramite un salto all'etichetta con il nome della funzione utilizzando il comando BL che registra nel link register la riga di ritorno dalla chiamata.
- Nelle funzioni si possono usare solamente i registri dal 0 a 3 se servono altri parametri si può usare lo stack con le funzioni push e pop
- Uso registri non temporanei => utilizzo dello stack
- Chiamo altre funzioni => salvataggio del link register

Esempio Codice Assembler

```
1         .text
2         .global fibo @ r0 contiene n
3                      @ r0 conterrà F(n)
4         .type fibo, %function
5
6   fibo: cmp r0, #0
7         moveq pc, lr @ caso base 1: r0 = fib(0) = 0
8         cmp r0, #1
9         moveq pc, lr @ case base 2: r0 = fib(1) = 1
10        sub r0, r0, #1 @ calcolo n-1
11        push {r0, lr} @ salvo (n-1) e lr sullo stack
12        bl fibo @ al termine ho in r0 = Fib(n-1)
13        pop {r1} @ metto in r1 il valore n-1
14        push {r0} @ salvo fib(n-1) sullo stack
15        sub r0, r1, #1 @ r0 contiene n-2
16        bl fibo @ al termine ho in r0 = Fib(n-2)
17        pop {r1} @ metto in r1 il valore Fib(n-1)
18        add r0, r0, r1 @ metto in r0 = Fib(n-1) + Fib(n-2)
19        pop {lr}
20        mov pc, lr
21
```

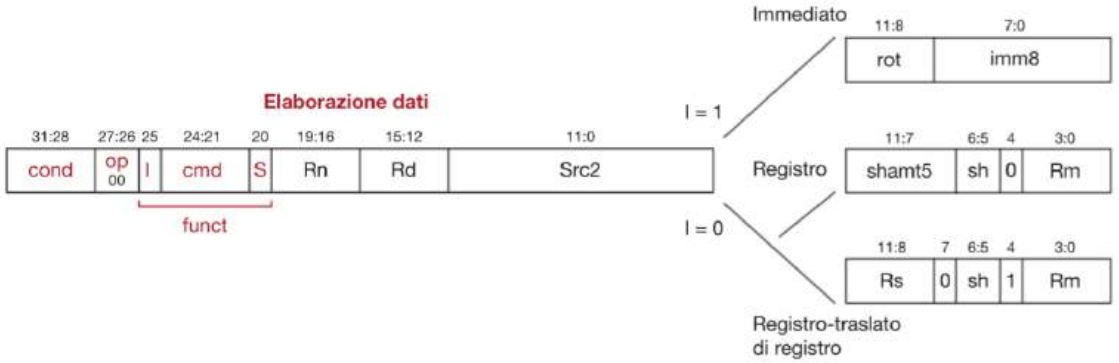
```
1         .text
2         .global merge @ r0 <- puntatore prima lista
3                       @ r1 <- puntatore seconda lista
4                       @ r0 <- valore di ritorno (puntatore lista unita)
5         .type merge, %function
6
7  merge: cmp r0, #0 @ prima == NULL ?
8         moveq r0, r1
9         moveq pc, lr
10        cmp r1, #0 @ seconda == NULL ?
11        moveq pc, lr
12        mov r2, r0 @ r2 puntatore prima (che sicuro non è vuota)
13  loop: cmp r2, #0 @ prima == NULL ?
14        beq fine
15        mov r3, r2 @ r3 contiene precedente elemento
16        ldr r2, [r2, #4] @ r2 contiene successivo elemento
17        b loop
18  fine: str r1, [r3, #4] @ concateno prima con seconda
19        mov pc, lr
20
```

Codifica in linguaggio macchina

Le istruzioni vengono codificate secondo lo schema ArmV7 dal codice binario. Una qualsiasi istruzione ha le seguente suddivisione:

COND = indica se si tratta di una istruzione condizionale e se lo è quale (EQ, LE, GE..), se non lo è 1110.
OP = a quale classe appartiene l'istruzione. (operative 00, memoria 01, salto 10, 11 riservato per il futuro).
FUNCT = campo che riservato che cambia il suo schema per i diversi tipi di istruzione.
Rn = registro **sorgente** dove prendo il dato. 4 bit per codificare i 16 registri.
Rd = registro **destinazione**, dove viene scritta l'operazione.
Src2 = viene usato per denotare il **3° registro** oppure per una combinazione di registri o un immediato.

Istruzioni operative



FUNCT

- **I** = mi dice se il terzo registro è un immediato quindi descrive la forma di Src2.
    - Se **I** = 0 allora è un registro che può essere letto in due modi differenti a seconda del bit tra Sh e Rm
    - Se **I** = 1 allora è un immediato
- **cmd** = il tipo di istruzione tra quelle operative.
- **S** = Mi dice se l'istruzione deve settare i **flag** del registro di stato CPSR.
- **Src2** = può essere di due tipi: un immediato, un registro o una combinazione.

Immediato:

rot = quanti bit ruoto per rappresentare un immediato > 255

imm8 = rappresentazione di un numero in 8 bit.

Registro - puo' essere ruotato o shiftato con un immediato o con un terzo registro:

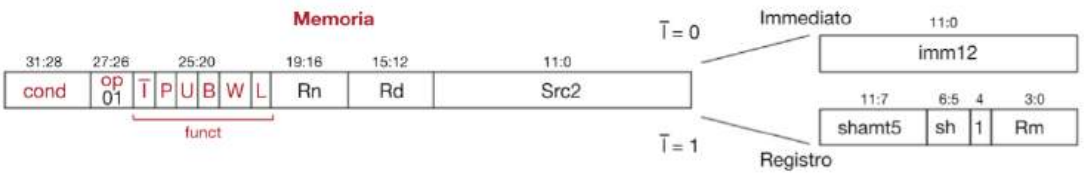
Se translo con un **IMMEDIATO:**

- **shamt5** = i primi 4 bit di questo campo rappresentano il numero di bit che devo ruotare per ottenere effettivamente quel numero.
- **sh** = rappresenta il tipo di traslazione dei bit che devo operare
- **0** = questo bit a 0 mi indica che ruoto o shifto con un immediato
- **Rm** = 3° registro

Se traslo con un **REGISTRO:**

- **Rs** = il registro che mi indica di quanto devo ruotare
- 0 = bit a 0
- **sh** = tipo di traslazione da fare
- Rm = 3° registro.

Istruzioni di memoria



Il campo **FUNCT** viene suddiviso in dei flag speciali: **I**, **P**, **U**, **B**, **W**, **L**: ad esempio B dice se vogliamo fare un byte o una word. Ognuno di questi flag forma uno dei parametri che ci dice come è fatta un'istruzione di accesso in memoria.

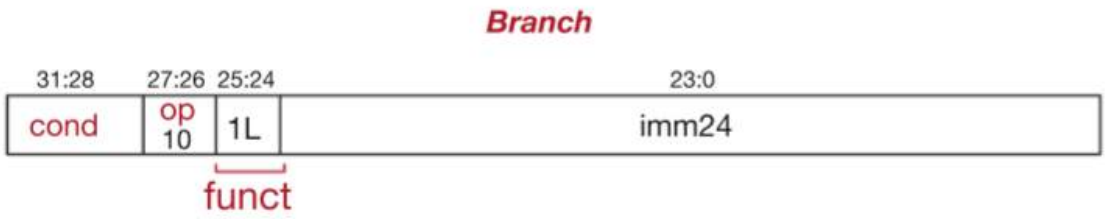
Bit	Significato	
	I	U
0	Spiazzamento immediato in Src2	Sottrae lo spiazzamento dalla base
1	Spiazzamento a registro in Src2	Somma lo spiazzamento alla base

P	W	Modo di gestione indice
0	0	Post-indice
0	1	Non supportato
1	0	Spiazzamento
1	1	Pre-indice

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

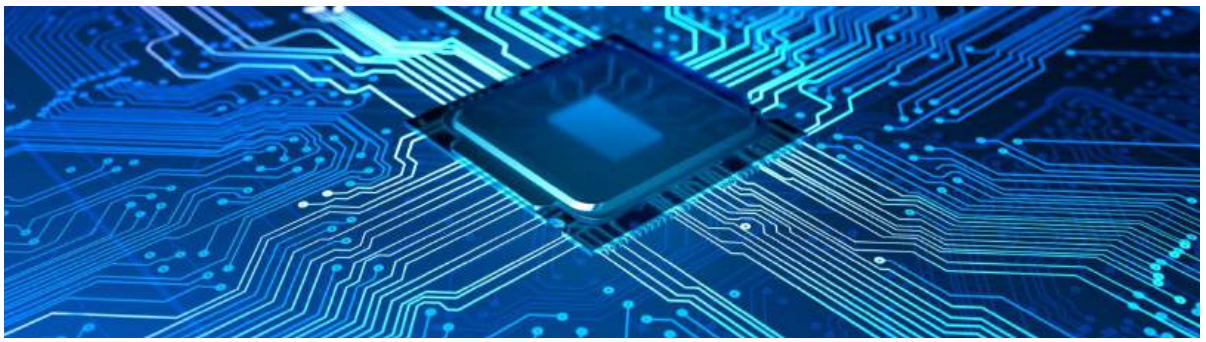
La **Source2** nel caso sia solo un immediato. Non abbiamo alcuna rotazione, ma solo **12 bit** che vengono usati per rappresentare il numero.

Istruzioni di salto



FUNCT ha soli 2 bit, uno dei bit è sempre 1, mentre l'altro, distingue tra B e BL. I 24 bit che rimangono sono un immediato **con segno**.

MicroArchitetture



Intro

Le microarchitetture definiscono i **circuiti logici** che **implementano** le **istruzioni** dell'architettura.
Sono organizzate sulla base di una **rete sequenziale**.
Lo **stato architetturale** definisce lo stato di un programma assembler. In questo stato includiamo tutte le informazioni fondamentali affinché un programma funzioni: i **registri architetturali** (i 16 registri definiti dall'architettura, usati per le istruzioni, PC, LR, SP…), il **CPSR**, una **memoria** dati in lettura/scrittura, una memoria per le istruzioni di sola lettura.
Lo stato si definisce architetturale perché fa riferimento all'**Instruction Set** (insieme delle istruzioni che deve implementare).
Lo stato architetturale non dipende dalla versione dell'architettura che stiamo utilizzando.
Lo stato *non architetturale* invece a seconda della versione che si utilizza potrebbe includere dei registri in più.
Le microarchitetture si suddividono in 3 tipi di microprocessori: **single cycle**, **multi cycle**, **pipeline**, con differenti prestazioni a seconda della tipologia.
La prestazione "migliore" minimizza il **tempo** in cui riesco a svolgere un lavoro:

tempo di esecuzione = #istruzioni * (#cicli / istruzione) * (#tempo/ ciclo)

Per fare una microarchitettura veloce dobbiamo o diminuire i cicli per istruzione o diminuire il ciclo di clock.

La microarchitettura svolge le seguenti operazioni:

```
While (1){
        Fetch (PC)
        Decode
        Execute
        Memory
        Write Back
        PC +=4
        …
}
```

Concetti di Parallelismo

Parallelismo Spaziale

Il parallelismo spaziale si ha quando la computazione avviene su dati diversi, in parallelo, grazie alla **replicazione** delle **componenti** che fanno il calcolo. Per questo si parla di "set di dati" che coesistono in un tempo t.

1) Esempio versione "**data parallel**"

Supponiamo di avere un libro in italiano da tradurre in inglese e di impiegare un tempo **t** per tradurre una pagina (supponendo che tutte le pagine siano = e che per tutte si impieghi t). Abbiamo **m** pagine. Il tempo di completamento se sono da solo a tradurre sarà:

$$T_C^{(1)} = t * m$$

Supponiamo ora di avere **n** lavoratori (Worker). **n** è detto grado di parallelismo. Occorre introdurre due fasi, la prima di **split** per suddividere il lavoro tra i lavoratori e una finale di **merge** in cui si mettono insieme i singoli risultati. Quindi adesso il tempo di completamento con grado di parallelismo n sarà:

$$T_C^{(n)} = T_{Split}^{(n)} + t * m/n + T_{Merge}^{(n)}$$

- Introduciamo una metrica chiamata **scalabilità** con grado di parallelismo **n**, che dice quanto siamo andati più veloce usando **n** lavoratori rispetto al caso sequenziale.

  $$S^{(n)} = T_C^{(1)} / T_C^{(n)} = (m * t) / (T_{Split}^{(n)} + t * m/n + T_{Merge}^{(n)})$$

  se consideriamo $T_{Split}^{(n)}$ e $T_{Merge}^{(n)}$ circa = 0 otteniamo la scalabilità **ideale**:

  $S^{(n)} = (m * t) / (t * m/n) = n$, cioè con n lavoratori vado n volte più veloce.

- Definiamo adesso anche l'**efficienza relativa** con grado di parallelismo n:

  $$\Sigma^{(n)} = T_{C\text{-}Ideale}^{(n)} / T_C^{(n)} = (T_C^{(1)} / n) / T_C^{(n)} = S^{(n)} / n$$

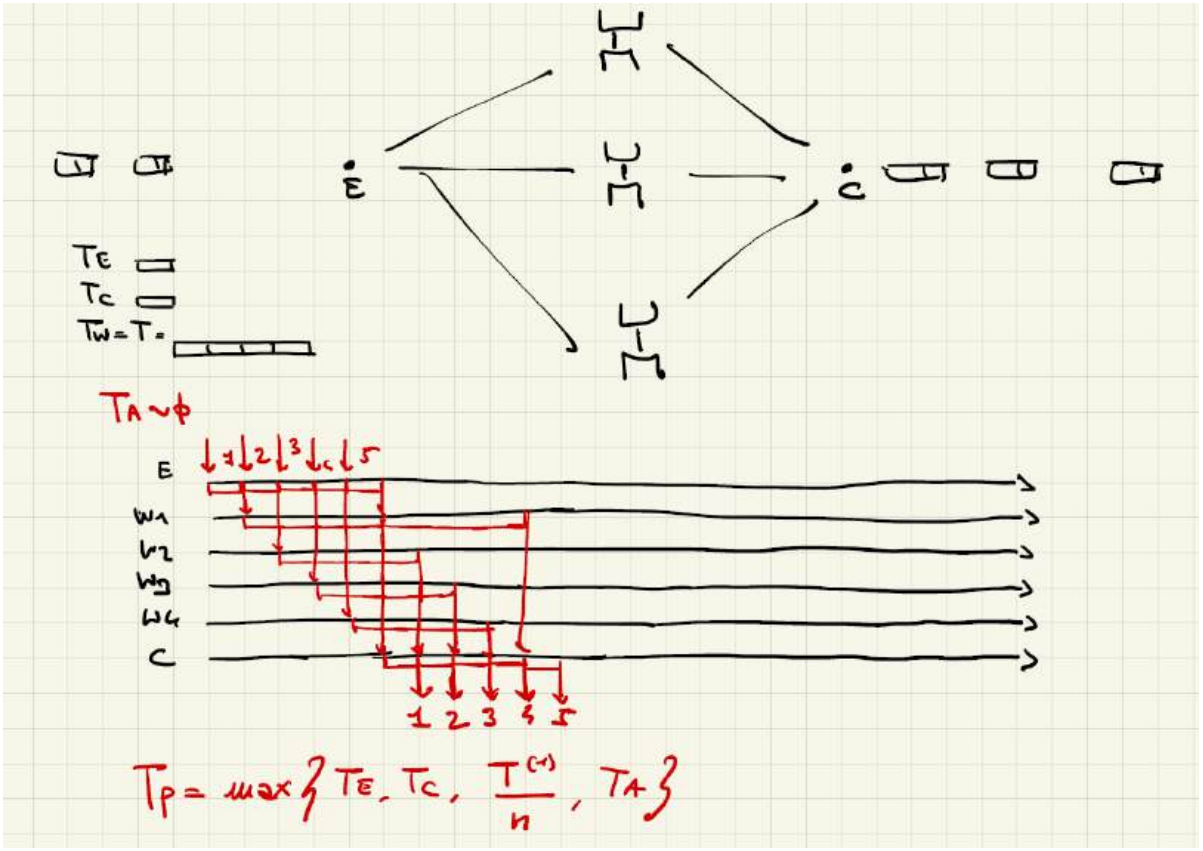
  questo osservando che il tempo di completamento ideale con parallelismo **n** è uguale al tempo di completamento **sequenziale** (1) diviso **n,** e poi andando a sostituire la **scalabilità**. Quindi se usando 5 lavoratori andiamo 5 volte più veloce otteniamo **efficienza relativa** = 1 cioè 100% (ideale).
  Un esempio in cui non ottengo le prestazioni ideali è quando per risparmiare sul tempo di split **non** divido il carico **equamente** tra i lavoratori. In quel caso dovrò aspettare il lavoratore che ha ricevuto più lavoro.

2) Esempio versione “**stream parallel**”

Supponiamo di avere un casello autostradale in cui abbiamo un tempo di **interarrivo T_a**, ovvero il tempo medio tra l’arrivo di una macchina e un’altra (es: 1 min). Ogni macchina impiega poi un tempo **t** al casello per pagare e andarsene, = $T_S^{(1)}$ tempo del servizio del casello con parallelismo 1. Se le macchine arrivassero infinitamente veloci il tempo $T_S^{(1)}$ indicherebbe il tempo tra l’inizio del servizio di una macchina e quello della successiva. Infine abbiamo il tempo di **interpartenza** T_p tempo che separa l’uscita di una macchina dall’uscita della successiva.

- se ho $T_a = 0$, quindi infinite macchine che arrivano avrò $T_C^{(1)} = t * m$, con $t = T_S^{(1)}$ e m il numero di macchine. T_p sarà anch’esso = **t**.
- se ho $T_a > 0$, posso avere 2 casi: $T_a < t$ e ho la stessa situazione di prima, oppure $T_a > t$ in qual caso il casello deve aspettare che arrivi la macchina e $T_p = T_a$.
  Quindi in generale $T_p = \max\{T_a , T_S^{(1)}\}$

Per velocizzare possiamo introdurre più caselli o lavoratori, un **emettitore E** che dice ad ogni macchina in che casello andare impiegando tempo T_E e un **collettore C** che instrada le macchine all’uscita in un tempo T_C.



Seguendo l'immagine avremo una latenza di 6 quadratini per ogni macchina (tempo in cui la macchina permane nel sistema). Nota che anche se la latenza è peggio di quando non avevamo il parallelismo, riusciamo a liberare una macchina ogni quadratino (dopo un certo tempo iniziale), quindi Tp = 1.

Dobbiamo anche tener conto del tempo di interarrivo Ta (nel disegno è assunto = 0), se hai troppi caselli e poche macchine in arrivo il parallelismo è sprecato. Abbiamo in generale la seguente formula:

T_p = `max{tempo emettitore, tempo collector, tempo worker/n, tempo di interarrivo}`

Parallelismo Temporale

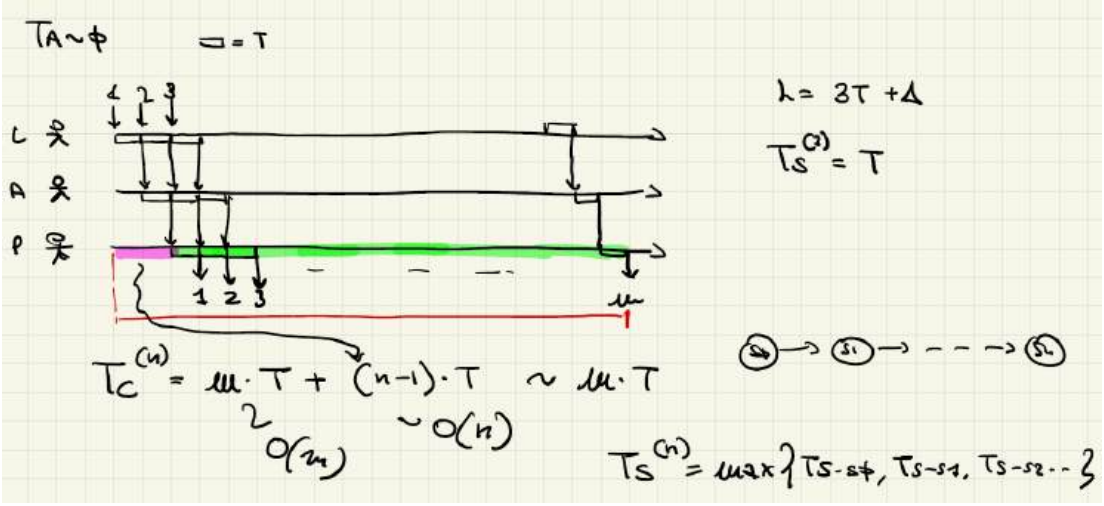
Con parallelismo temporale prendiamo in considerazione uno stream di dati (flusso) con l'idea che ad un tempo t1 avremo un dato, ad un tempo t2 un altro dato, ecc…

```
        a1     a2     a3
X1 |-------|--------|---------|
              a1      a2      a3
X2          |---------|--------|--------|
```

L'**HW** che compie un certo compito viene quindi **diviso** in **sottocomponenti** una parte del compito su dati diversi, in parallelo rispetto alle altre componenti.

- Esempio: supponiamo di avere una persona che deve lavare (in un tempo **t**), asciugare (**t**) e piegare (**t**) una serie di **m** vestiti che arrivano con un tempo T_a. Viene prodotto in uscita uno stream di abiti con un certo tempo di interpartenza T_p.
  In questo caso abbiamo $T_S^{(1)} = T_p = 3t$ e $T_C^{(1)} = 3t * m$ perché deve fare tutta la singola persona.

  Per parallelizzare potremmo aggiungere un'altra persona che sa fare tutte e 3 le operazioni (parallelismo spaziale), approccio costoso a livello di HW che va **replicato tutto**.
  Altrimenti si possono aggiungere 2 persone (quindi in tutto ne avremo 3) che svolgono ognuno una singola operazione (uno lava, uno asciuga, uno piega) => **pipeline**.

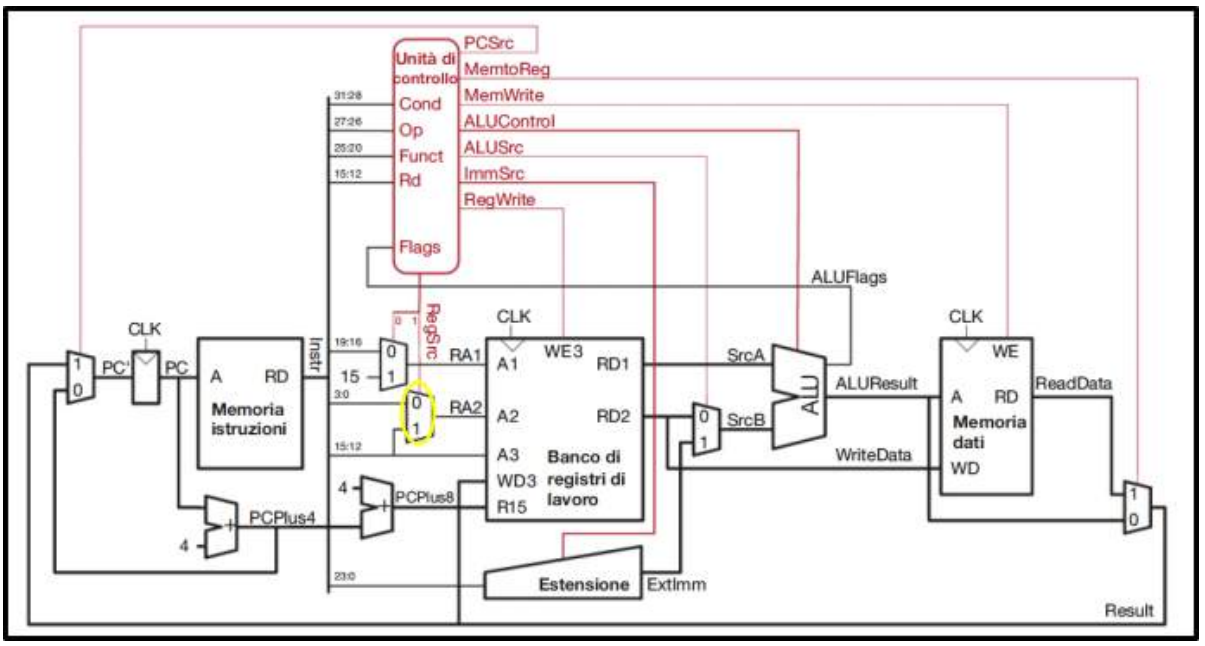


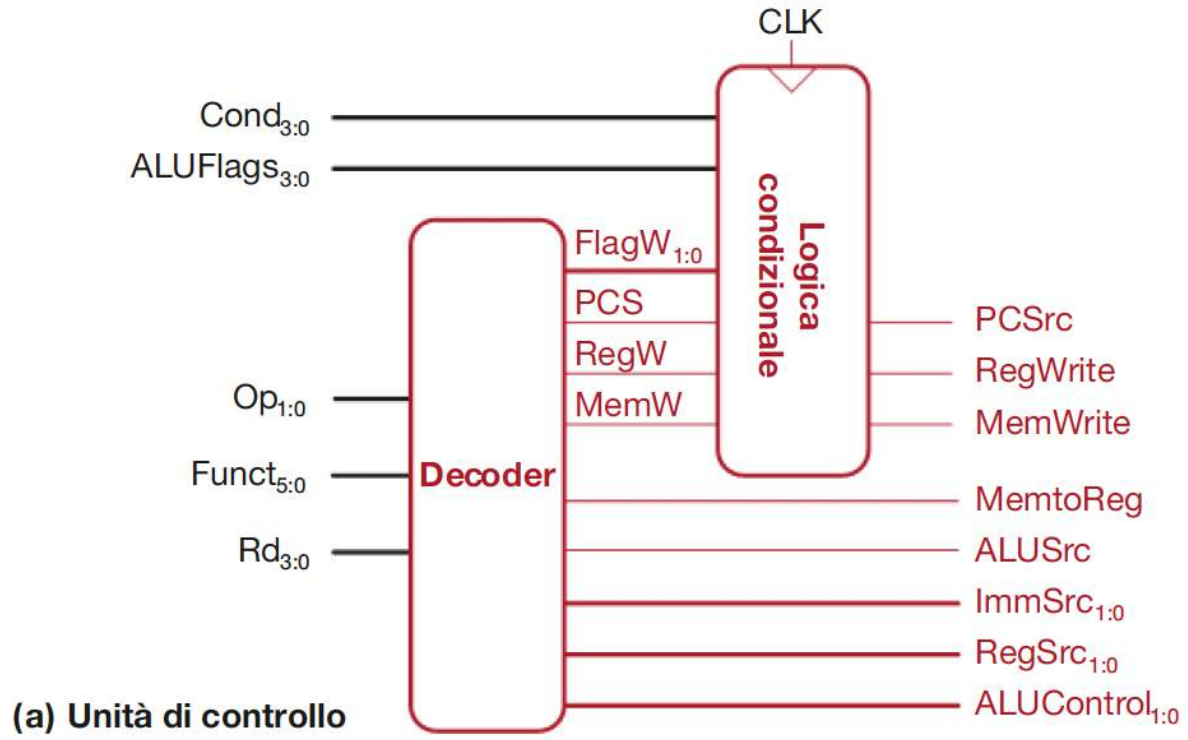
  La latenza rimane sempre uguale o peggiore (sul singolo abito si perde tempo), si migliora però il tempo di servizio o **throughput** to banda. $T_S^{(3)} = t$, circa (considerando $T_a = 0$).
  Non stiamo considerando il tempo iniziale prima che tutti gli worker lavorino contemporaneamente, che è **(n-1)*t** per **n** omini, quindi qualcosa di trascurabile rispetto all'input **m**.
  - Nel caso in cui le operazioni singole impieghino tempi diverse, quella più lunga determina il tempo dell'uscita $T_S^{(n)}$.

Si possono poi combinare le due forme di parallelismo (spaziale e temporale). Ad esempio potremmo avere 2 persone che asciugano. Concretamente si fa riferimento al processore **superscalare**.

Single cycle

Il processore single cycle esegue tutte le istruzioni in un ciclo di clock, il che permette di non avere registri non architetturali, però la **durata del ciclo** dovrà dipendere dall'**istruzione più lunga** in termini di **tempo**. Si tratta dell'implementazione più semplice da realizzare.





Questo è lo schema della microarchitettura single cycle, in nero c'è la parte del data path e in rosso la parte dell'**unità di controllo**.

Data Path

Nel datapath notiamo che abbiamo un dispendio di sommatori elevato, infatti oltre alla ALU abbiamo 2 sommatori per la lettura del registro 15 (Program Counter). La lettura del PC infatti prevede una doppia somma di +4: ogni volta che leggiamo il PC quindi leggiamo in realtà **PC+8**, così nel caso ad esempio di una **Branch&Link** quando dobbiamo tornare al programma chiamante salteremo le 2 istruzioni che hanno causato il salto. (la Branch&Link si implementa come un salto in cui si salva il valore del PC+8 nel LR).

Abbiamo **2 memorie** una per le istruzioni, che ricevuto in ingresso il PC restituisce l'istruzione corrispondente - **FETCH** - e una per i dati. *Questa astrazione non è attuabile realisticamente.*

Un **registro** contenente il PC, che viene letto ad ogni ciclo di clock. Viene scritto da PC + 4 o riceve una scrittura a seconda del settaggio di **PCSrc** se l'istruzione precedente scriveva nel PC.
(esempio MOV PC, LR).

Il PC viene poi mandato in un sommatore che calcola l'indirizzo della prossima istruzione.

La nostra istruzione viene decomposta e una parte inviata all'unità di controllo che setta tutto il percorso per il tipo di istruzione che dobbiamo eseguire. A questo punto leggiamo i registri nel **Register File** (banco dei registri), che presi in input 2 registri restituisce il valore in essi contenuto.

Da notare che non sempre prendiamo il registro sorgente classico, nei casi di **branch** prendiamo il PC grazie a un multiplexer che comandato da **RegSrc** seleziona quale ingresso mettere nel banco dei registri.

La **Store** è l'unica istruzione che va a **leggere** il primo registro **Rd**. Il multiplexer cerchiato in giallo in figura serve pertanto per distinguere se il secondo registro in lettura sarà il registro **Rm** contenuto in Src2 quando cioè non si usa un immediato, OPPURE nel caso di una Store quando dovremo leggere **Rn** (sempre con il flag **RegSrc**).

Un **estensore** per gli immediati, che permette di estendere un immediato da 8, 12, 24 bit in uno da 32, aggiungendo gli "0" o "1" mancanti a sinistra, a seconda del segno. Il **numero di bit in ingresso** (dell'immediato) a seconda del tipo di operazione viene specificato da **ImmSrc**.

Abbiamo completato l'operazione di **DECODE**.

All'uscita dal nostro banco dei registri abbiamo la **ALU** che permette di fare le operazioni con i due registri appena letti o con solo Rn e usare solo l'immediato dall'estensore, se specificato correttamente da **AluSrc.**
La ALU invia il suo risultato ai registri se deve esser scritto, o alla memoria se ha calcolato un indirizzo da leggere o scrivere - **EXECUTE.**

Infine abbiamo la memoria dati che riceve due ingressi, uno che indica l'indirizzo su cui scrivere o leggere e il secondo con l'eventuale dato da scrivere. Anche questo componente riceve in ingresso dall'unità di controllo un bit **MemWrite** per decidere se abilitare la scrittura o no.
Ad esempio per la STR MemWrite = 1.

Il dato letto in memoria viene poi trasferito o nel banco dei registri nel caso in cui venga scritto..
Questo definisce l'operazione di **WRITE BACK**.

Unità di controllo

L'unità di controllo descrive attraverso una rete combinatoria e un registro contenente i registri di stato (N, Z, C, V) il **comportamento del datapath** per l'esecuzione dell'istruzione.
E' divisa in 2 parti: **decoder** e **logica condizionale.**
Il decoder prende in ingresso tutti i dati (come i campi **OP, FUNCT**, **COND** e **RD**) e decide come impostare i flag. Alcuni di questi dati vengono filtrati dalla logica condizionale.
In particolare abbiamo che:

- **MemW** in ingresso alla logica condizionale dice se l'operazione che stiamo eseguendo vorrebbe scrivere in memoria, hanno settato MemW = 1 le operazioni di **STR**. All'interno dell'unità di logica condizionale viene messa in AND con i flag che deve rispettare (nel caso sia condizionata) e produce in output **MemWrite** che rappresenta la **WE** della memoria dati.

- Analogamente vale per **RegW** - **RegWrite**, con la differenza che non si tratta di una store ma di istruzioni operative (add, sub ecc....) => se un'istruzione è operativa di sicuro scrive nel Register File (ADDEQ, SUBLT..).

- **PCS** - **PCSrc** gestisce i salti, anche condizionati: **BEQ**, **BGT**, **BLO** ecc... => se l'istruzione è di salto allora PCS = 1, però PCSrc non gestisce un WE, ma il multiplexer: questo è vero perché nella nostra microarchitettura ad ogni ciclo di clock si scrive sempre sul PC, quindi è come se avesse un WE sempre impostato ad 1.

Componente Decoder

Questa componente è una rete combinatoria, senza alcun registro, divisa in 3 sottocomponenti:

1. **Main decoder**
2. **ALU decoder**
3. **PC logic**.

1) Main Decoder
Questa tabella descrive il comportamento del main decoder. Ricevendo in ingresso OP e FUNCT manda all'unità di logica condizionale RegW e MemW invia al datapath senza passare dall'unità di logica condizionale MemToReg, AluSrc, ImmSrc, RegSrc.

Op	$Funct_5$	$Funct_0$	Type	Branch	MemtoReg	MemW	ALUSrc	ImmSrc	RegW	RegSrc	ALUOp
00	0	X	DP Reg	0	0	0	0	XX	1	00	1
00	1	X	DP Imm	0	0	0	1	00	1	X0	1
01	X	0	STR	0	X	1	1	01	0	10	0
01	X	1	LDR	0	1	0	1	01	1	X0	0
11	X	X	B	1	0	0	1	10	0	X1	0

2) Alu Decoder

Prende 5 bit del campo Funct CMD + Funct 0 (esclude quindi solo il bit che dice se source 2 è Imm / Reg).

Produce i seguenti flag per il datapath:

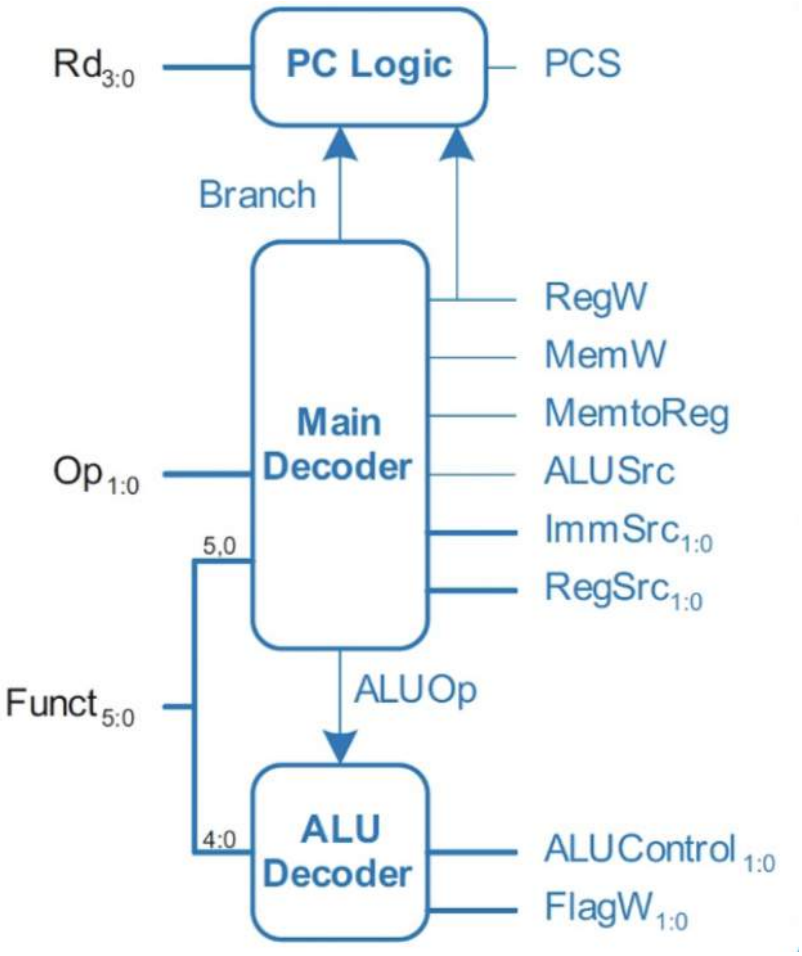
1. **ALU Control** specifica che operazione deve eseguire la ALU (somma, sottrazione, AND, OR).
2. **FlagW** (fatto da 2 bit) => **Abilito la scrittura del registro di stato (CPSR)**.
   perchè ho bisogno di 2 bit per scrivere sui flag?
   Dividiamo i flag in 2 registri: uno per n, z (nz) e uno per c, v (cv) => 2 segnali WE diversi => dei 2 bit di FlagW, uno abilita la scrittura in nz e uno in cv (scrivo su tutti i flag è 00)

ALUOp	$Funct_{4:1}$ (*cmd*)	$Funct_0$ (*S*)	Type	$ALUControl_{1:0}$	$FlagW_{1:0}$
0	X	X	Not DP	00	00
1	0100	0	ADD	00	00
		1			11
	0010	0	SUB	01	00
		1			11
	0000	0	AND	10	00
		1			10
	1100	0	ORR	11	00
		1			10

3) PC logic

Input: **Rd** (destinazione dell'istruzione), un bit **Branch** (è una branch?) e **RegW**

output: **PCS** che va filtrato dal logical conditional: dice se va fatto PC+4 o no
quando Rd è 15 e RegW vale 1 (quindi ad esempio scrivo sul PC con un'istruzione operativa) allora PCS vale 0 perché solo in questi casi non faccio PC+4.



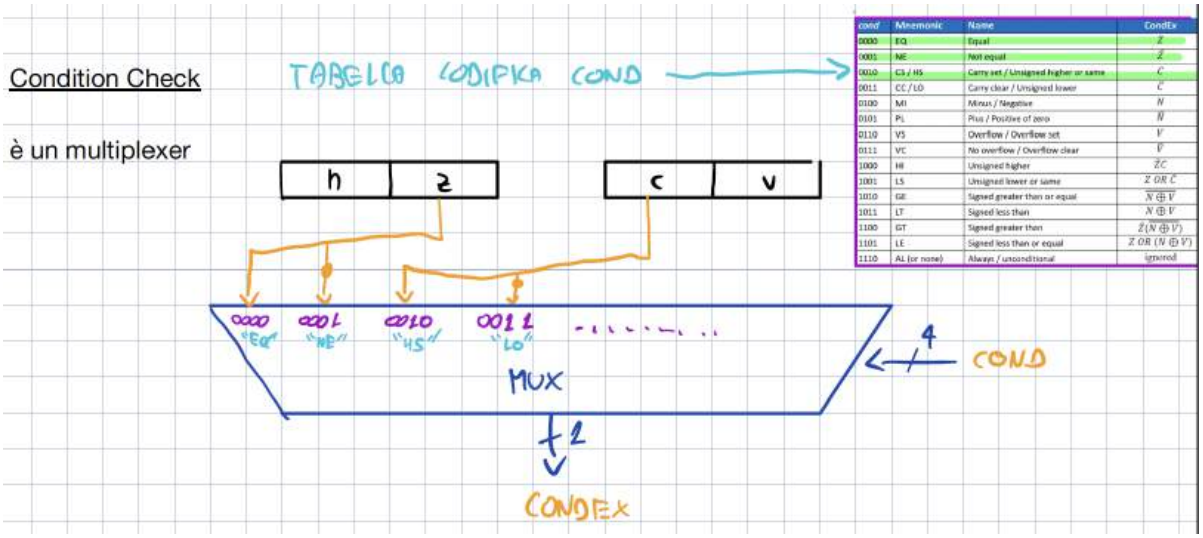
Componente di unità di logica condizionale

L'obiettivo di questa componente è filtrare alcuni flag del decoder e aggiornare i registri di stato.
Input :

- **FlagW** => 2 bit indica quali registri di stato posso scrivere.
- **ALUFlags** => flag mandati dalla ALU che potrebbero essere salvati nel registro di stato.
- **PCS** = 1 quando riscrivere il PC e non fare il semplice PC + 4. Questo flag viene inviato dal decoder.
- **RegW** = 1 se scrivo nei registri, altrimenti 0 (per le istr. operative). Questo flag viene inviato dal decoder.
- **MemW** = 1 se scrivo in memoria, altrimenti 0 (per la STR). Questo flag viene inviato dal decoder.
- **Cond** = mi dice quale condizione deve essere rispettata.

il Condition Check è una rete combinatoria che produce in **output** il bit **CondEx** che dirà se la condizione è rispettata o meno. Se cond = 1110 (op. non condizionale) allora CondEx = 1.

Implementata come un **multiplexer** dove gli ingressi sono una combinazione di registri di stato, la COND è l'ingresso di controllo del multiplexer che dice quali registri di stato (o flag) utilizzare per produrre la condizione desiderata.
L'output è **CondEx** che verrà a sua volta messo in **AND** con l'operazione condizionata da svolgere (nel caso di una ADDEQ con RegW) per determinare il WE.



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

Per estendere le flag per la CMP viene aggiunto anche una flag noWrite alla AluDecoder.

Esempi di operazioni svolte

Esempio di istruzione operativa

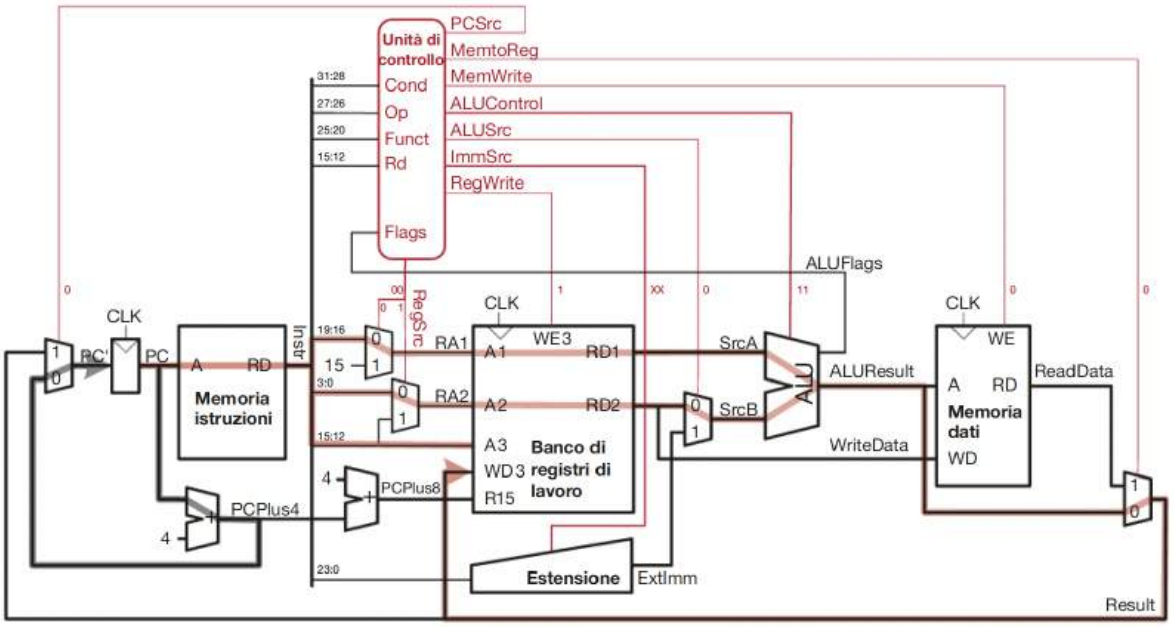
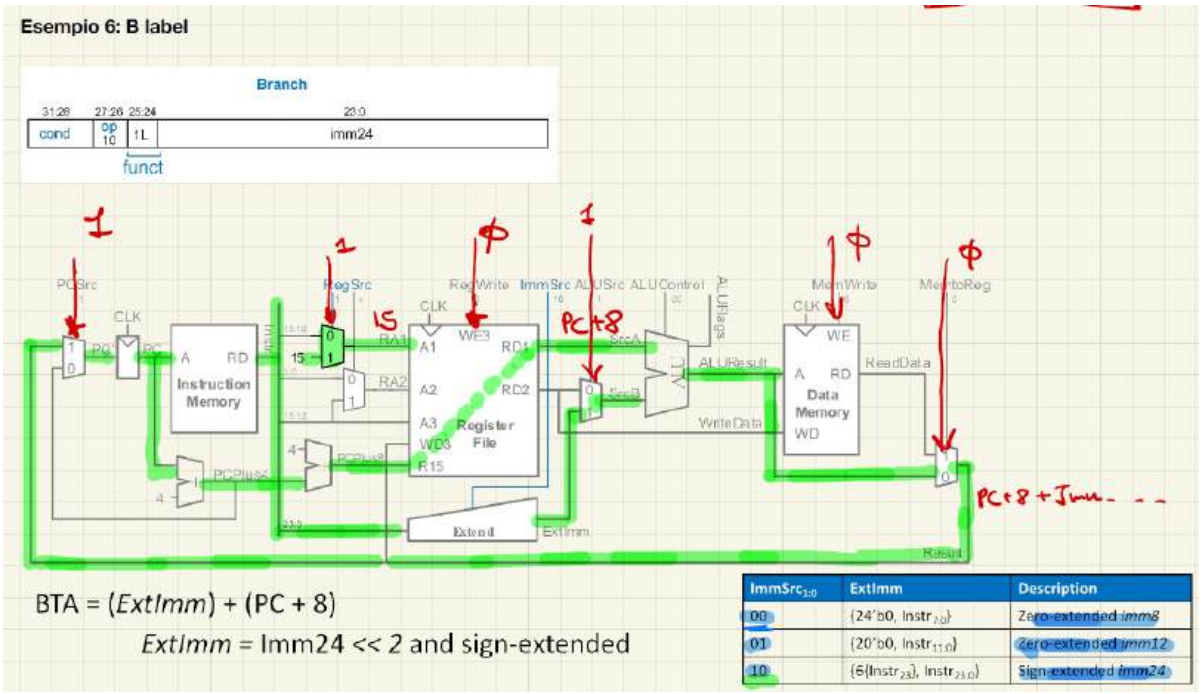


Figura 7.15 **Segnali di controllo e flusso dei dati durante l'esecuzione dell'istruzione** ORR.

Esempio di istruzione di salto



Esempio di istruzione di Load

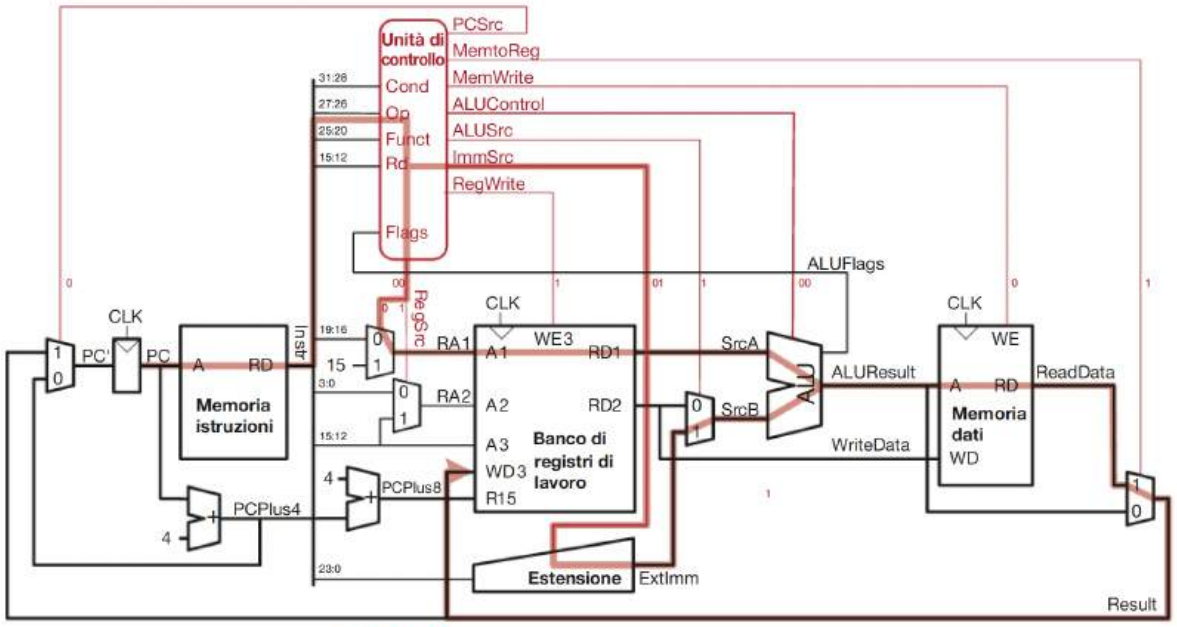


Figura 7.18 Percorso critico per l'istruzione LDR.

Analisi delle prestazioni

Ogni istruzione occupa un ciclo di clock, quindi Cicli Per Istruzione (CPI) = 1. La lunghezza del ciclo di clock è quindi la lunghezza dell'istruzione più lunga, ovvero la LDR (l'unica che deve eseguire Fetch, Decode, Execute, Memory e WriteBack). Quindi calcoliamo il tempo impiegato da questa istruzione per essere eseguita considerando tutti i ritardi delle componenti della nostra microarchitettura.

$$T_{c1} = t_{pcq\_PC} + t_{mem} + t_{dec} + \max[t_{mux} + t_{RFread}, t_{ext} + t_{mux}] + t_{ALU} + t_{mem} + t_{mux} + t_{RFsetup}$$

t_{pcq_PC} = tempo di lettura del registro PC

t_{mem} = tempo di lettura in memoria istruzioni

t_{dec} = tempo impiegato dall'unità di controllo

$t_{mux} + t_{RFread}$ = tempo dei multiplexer e della lettura dei registri

$t_{ext} + t_{mux}$ = tempo estensore

t_{ALU} = tempo calcolo ALU

t_{mem} = tempo di lettura in memoria dati

t_{mux} = tempo impiegato dal multiplexer finale

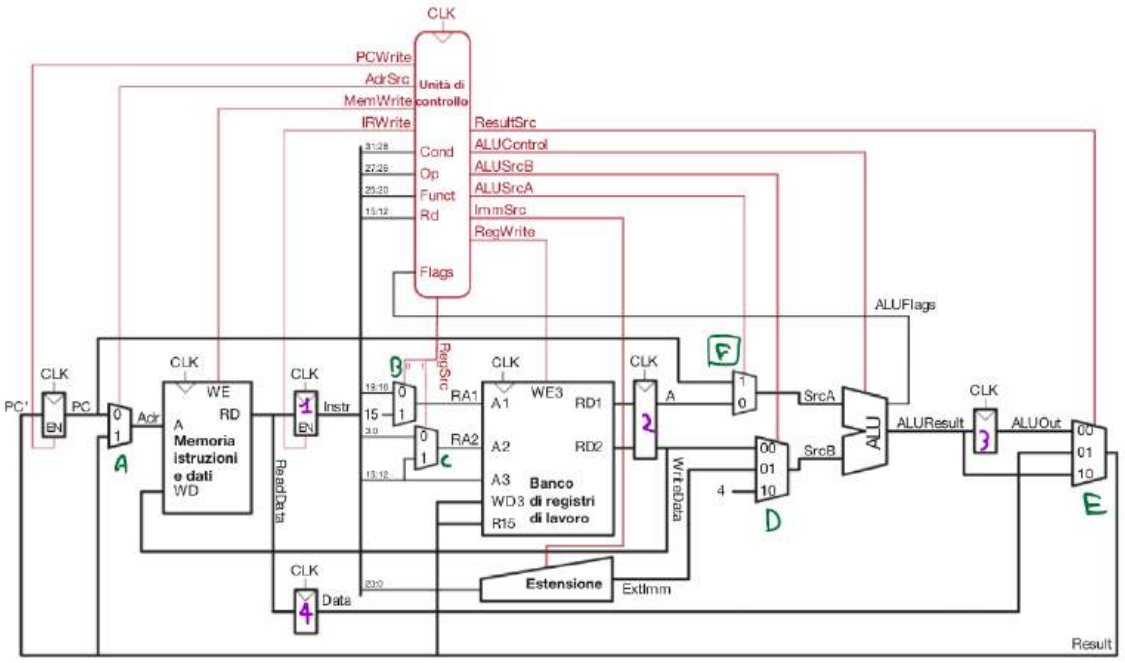
$t_{RFsetup}$ = tempo per il calcolo della nuova istruzione PC

Problematiche

Il processore a ciclo singolo ha tre elementi di debolezza:

1. Richiede **memorie separate** per istruzioni e dati, mentre la maggior parte dei processori ha una sola memoria esterna che contiene sia istruzioni sia dati.
2. Richiede un **ciclo di clock** abbastanza **lungo** da consentire l'esecuzione dell'istruzione più lenta (LDR) anche se molte istruzioni potrebbero essere più veloci.
3. Richiede **tre circuiti sommatori** (uno nell'ALU, e due per la Logica del PC): circuiti relativamente costosi soprattutto se devono essere veloci.

Multi cycle



In questa microarchitettura l'idea è quella di risolvere i problemi della microarchitettura single cycle ovvero:
1) dispendio HW -> non vengono usati i 2 sommatori e abbiamo un'unica memoria per le istruzioni e per i dati.
2) ciclo di clock lungo (= LDR) -> **cicli di clock più corti**, dove eseguo solo una parte del ciclo dell'istruzione.

💡L'idea è quella di suddividere i vari passaggi per l'esecuzione di una istruzione in cicli di clock quindi

1) FETCH -> 1 ciclo di clock
2) DECODE -> 1 ciclo di clock
3) EXECUTE -> 1 ciclo di clock
4) MEMORY -> 1 ciclo di clock
5) WRITE BACK -> 1 ciclo di clock

In questo modo per le istruzioni che non richiedono tutti e 5 i passaggi (es. le istruzioni di salto) non spreco tempo. Inoltre la suddivisione dei cicli di clock in questo modo mi permette anche un notevole risparmio nel datapath infatti così sono in grado di **riutilizzare le componenti** per scopi diversi. Questo rende però più complicata **l'unità di controllo** che in questa microarchitettura non sarà una semplice rete combinatoria ma una **FSM** poiché ogni fase dell'istruzione richiede flag diversi a seconda di quello che si sta facendo.

Data path

DIfferenze con Single-cycle

- **4 registri non architetturali** in più, presenti dentro il datapath per conservare le informazioni durante il passaggio da un ciclo di clock all'altro: questi registri non si trovano nel Register File e l'utente neanche saprà che esistono, non si possono manipolare. Separano le fasi di Fetch, Decode, Execute, Memory e Write Back.

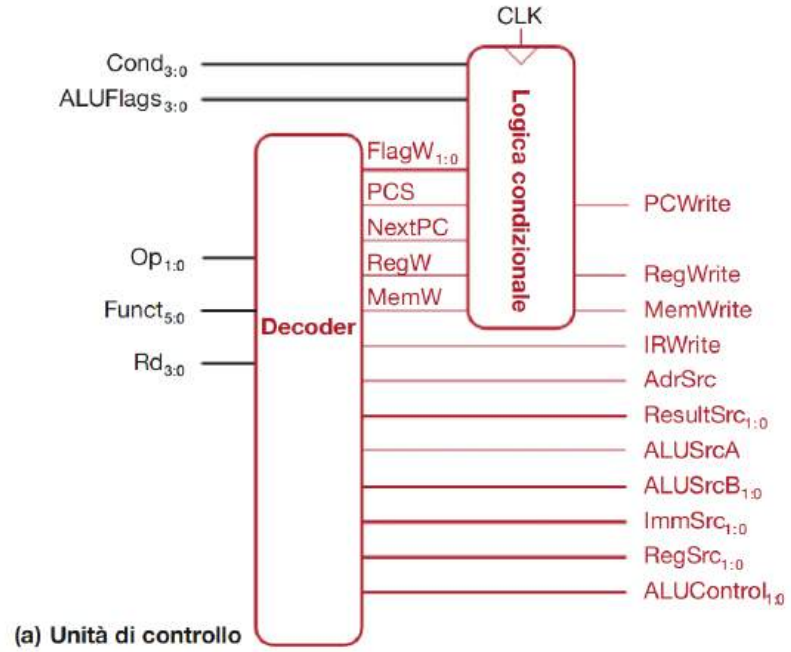
- Ho **una** sola **memoria**: memoria istruzioni e memoria dati coincidono. Il multiplexer (E), quello più a destra, ha un'entrata in più che *serve a propagare un eventuale dato letto dalla memoria in fase di memory*: tale valore arriva dal registro non architetturale (4), quello più in basso che va all'uscita "01" del multiplexer, e verrà propagato in ingresso alla porta di scrittura del RF.

- **PC+4** viene **svolto dalla ALU**, senza usare i due sommatori, così da risparmiare risorse. La ALU viene usata solo in fase di Decode quindi PC + 4 viene fatto dalla ALU in fase di **FETCH**. per questo viene spostato il multiplexer (A). Viene aggiunto un multiplexer (F) per lasciar passare
  SrcA = PC in input alla ALU invece che un registro letto da RF. Inoltre il multiplexer per scegliere il secondo termine da sommare con la ALU ha un'entrata in più con il valore 4.

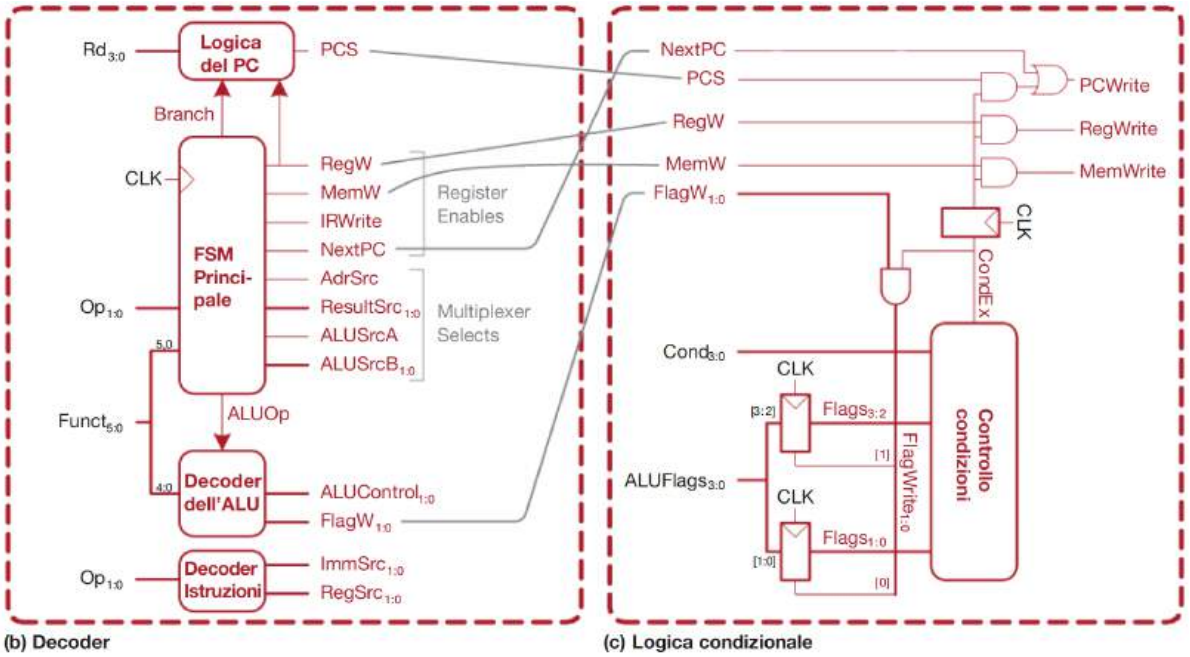
Registri non architetturali
Permettono di memorizzare dati nelle varie fasi dell'istruzione permettendo di eseguire la stessa istruzione con più cicli.

- **REGISTRO 1** Memorizza i 32 bit dell'istruzione, ci scrivo solo in fase di fetch.
- **REGISTRO 2** salva il valore letto di **Rn** e **Rm** (se source2 è un registro) dopo la fase di decode => per accogliere entrambi questo è l'unico registro da **64 bit** (diviso in 2 parti da 32 bit)
- **REGISTRO 3** salva l'output della ALU dopo la fase di execute
- **REGISTRO 4** nella fase di memory, se c'è un valore letto dalla memoria questo verrà stampato e salvato in questo quarto registro non architetturale ("torno indietro, leggo mem e salvo valore letto") → quando viene stampato l'output della memoria, il primo registro non architetturale a WE uguale a zero in quanto non ho letto un'istruzione ma appunto un dato.

Unità di controllo

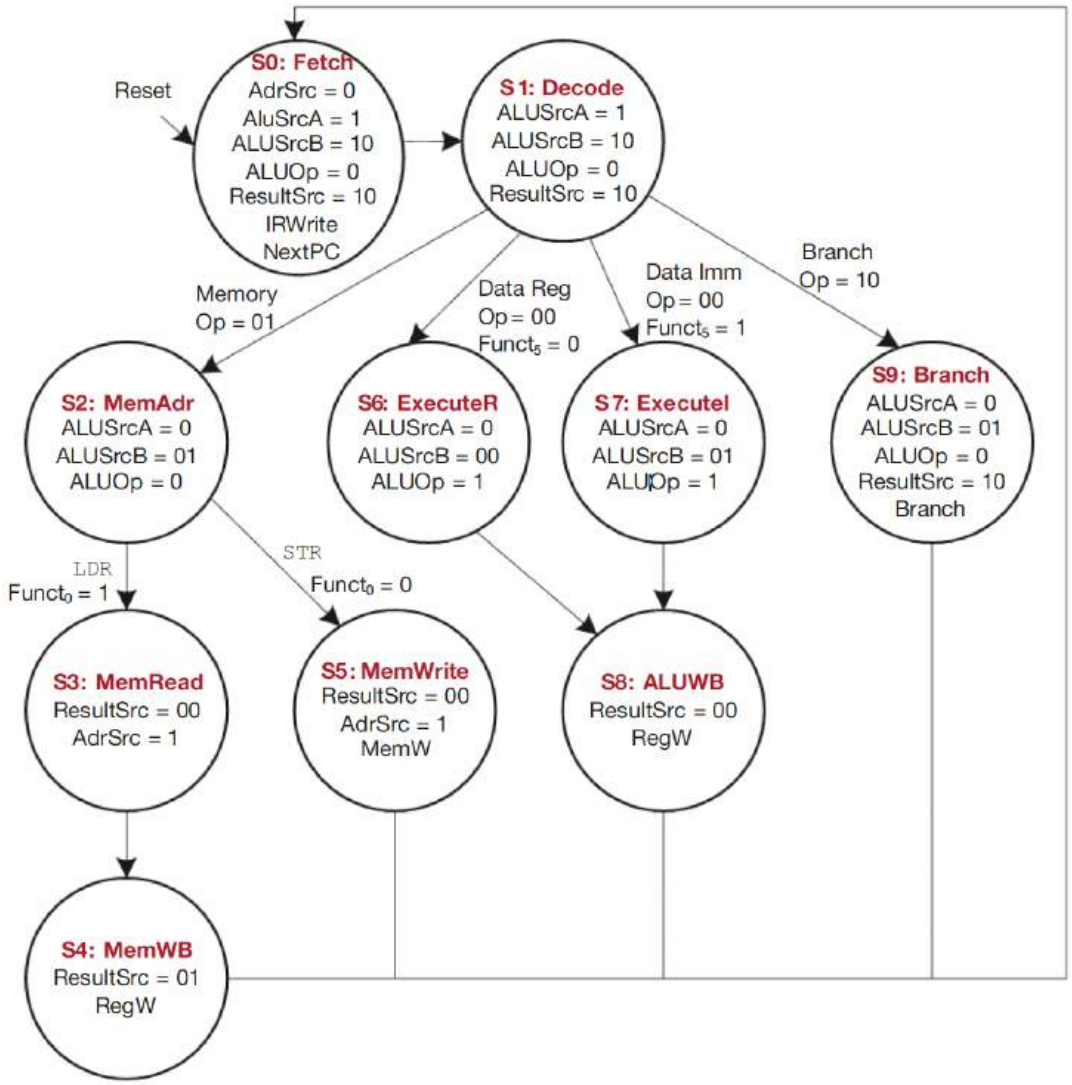


(a) Unità di controllo



(b) Decoder (c) Logica condizionale

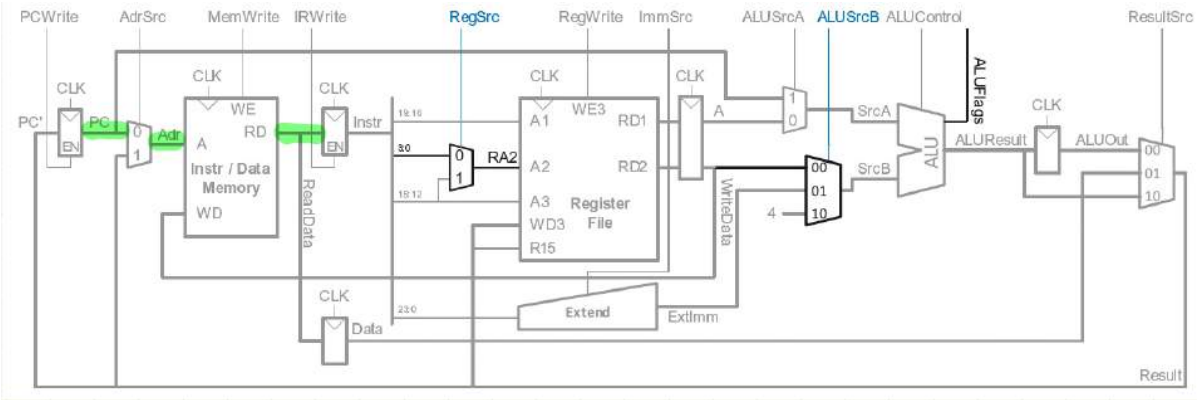
Da un punto di vista logico la parte di controllo della multi cycle e quella della single cycle sono uguali l'unica differenza è che il **main decoder** che nella single cycle è una rete combinatoria mentre nella multi cycle è sostituito dall' **FSM principale:** un automa di **moore**. Dato che deve ricordarsi lo stato in cui si trova ovvero lo stato di esecuzione dell'istruzione, inoltre ci sono più bit che vengono mandati al datapath rispetto a prima.



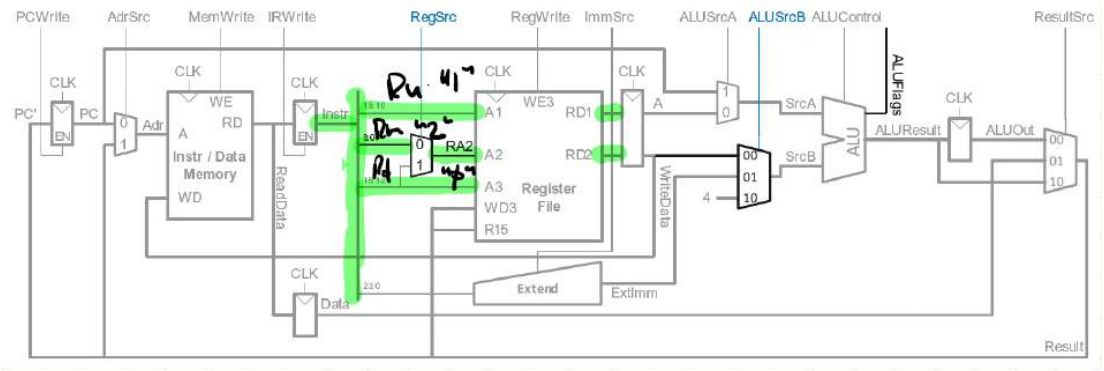
Esempi di operazioni

Esempio : ADD r0, r1, r2

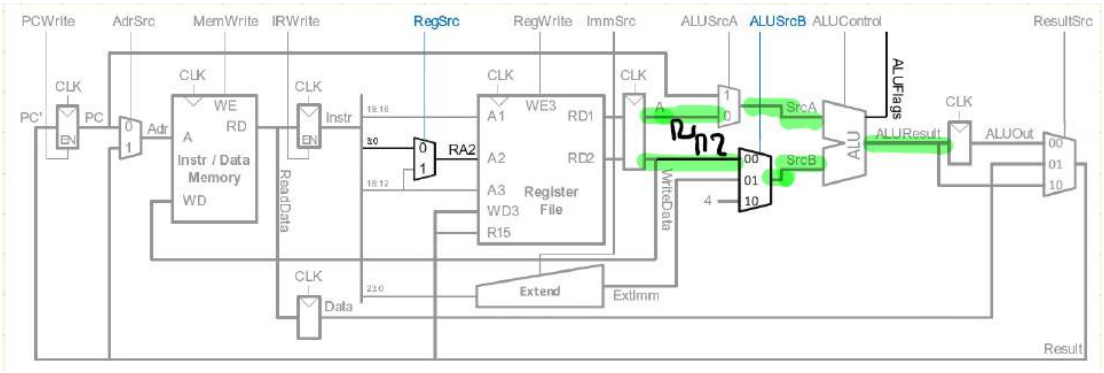
1) Fetch



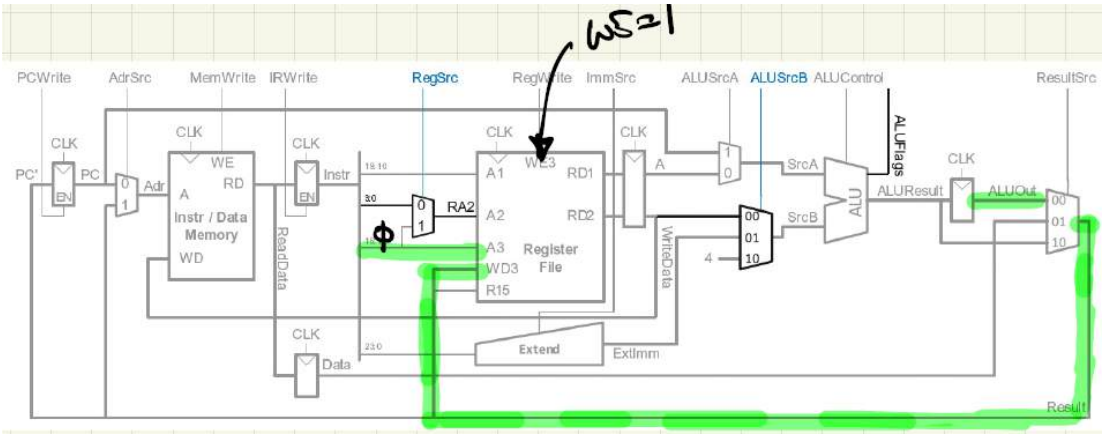
2) Decode



3) Execute

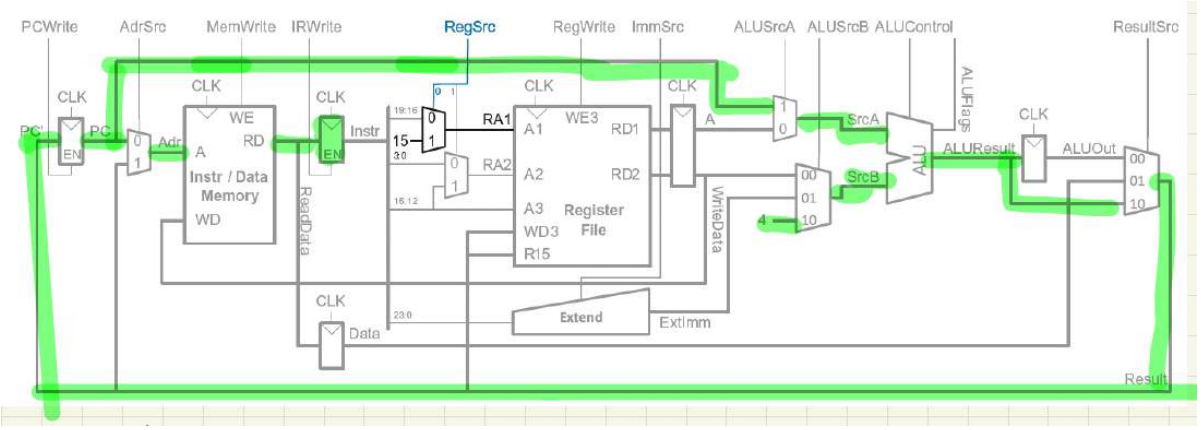


4) WriteBack



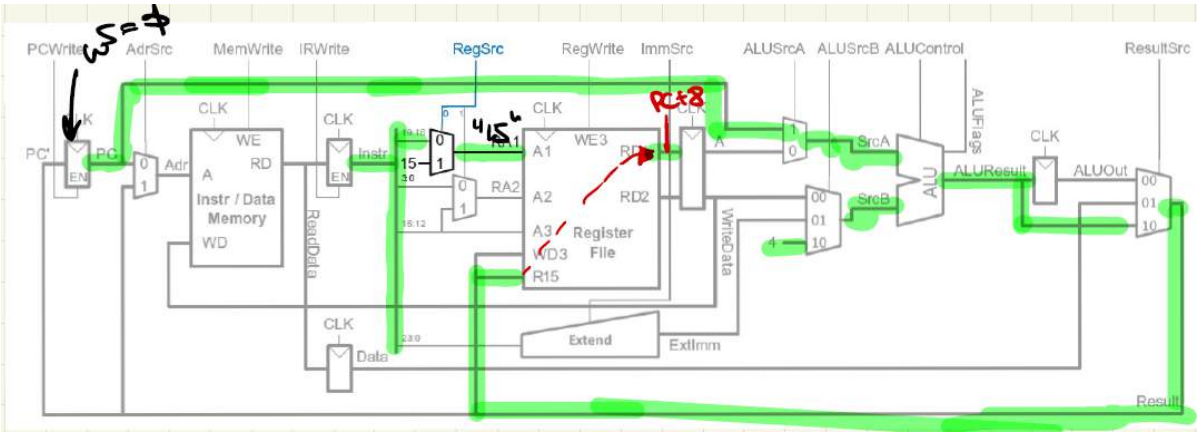
Esempio: B label

1) Fetch

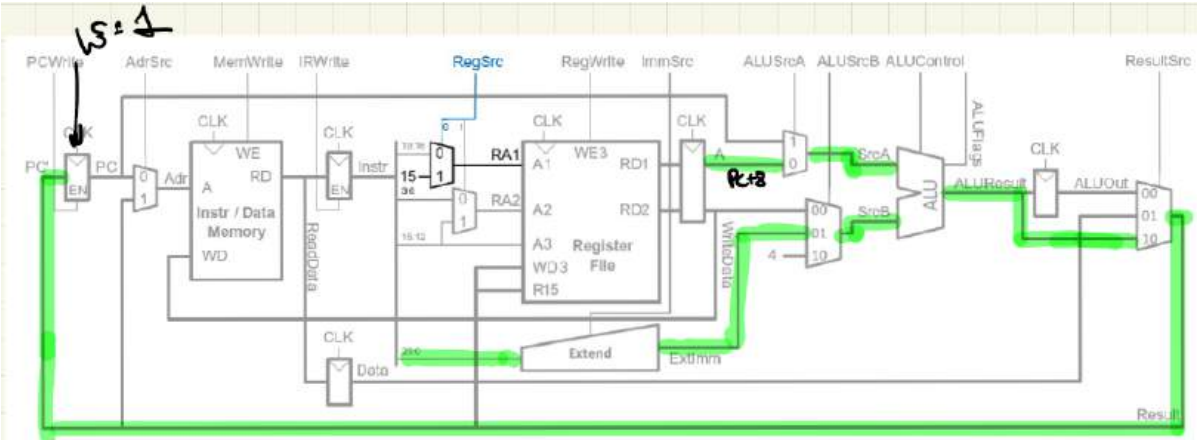


durante la **Fecht** nel multi cycle viene fatta la PC + 4.

2)Decode

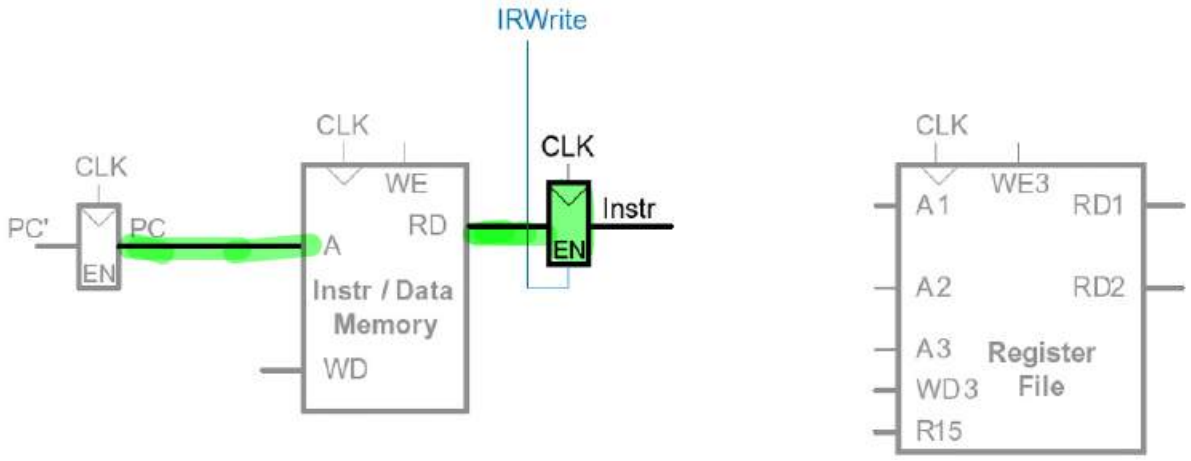


3)Execute

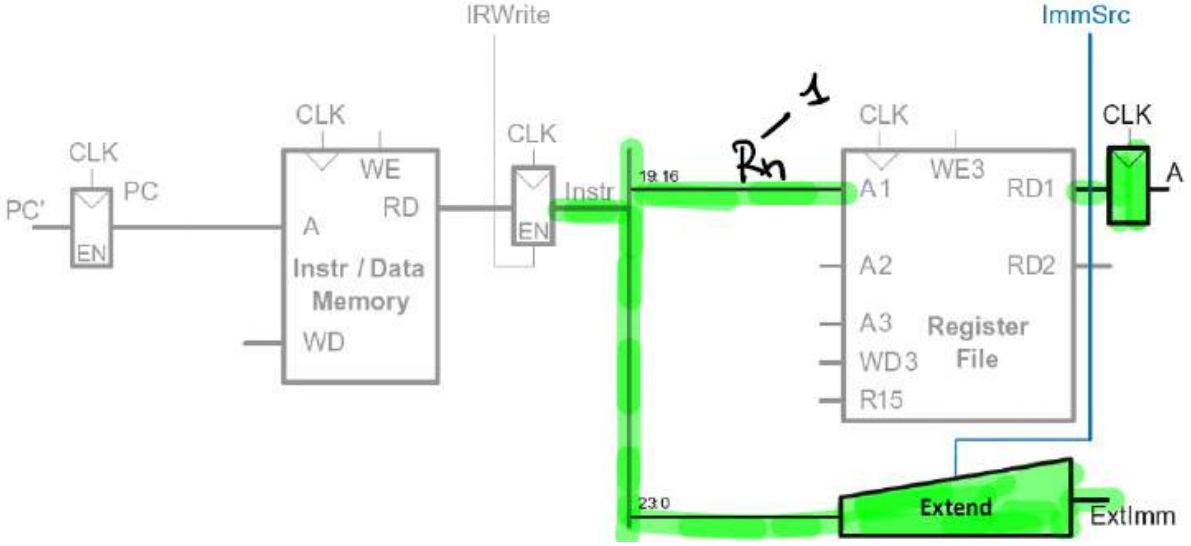


Esempio: LDR r0, [r1, #imm12]

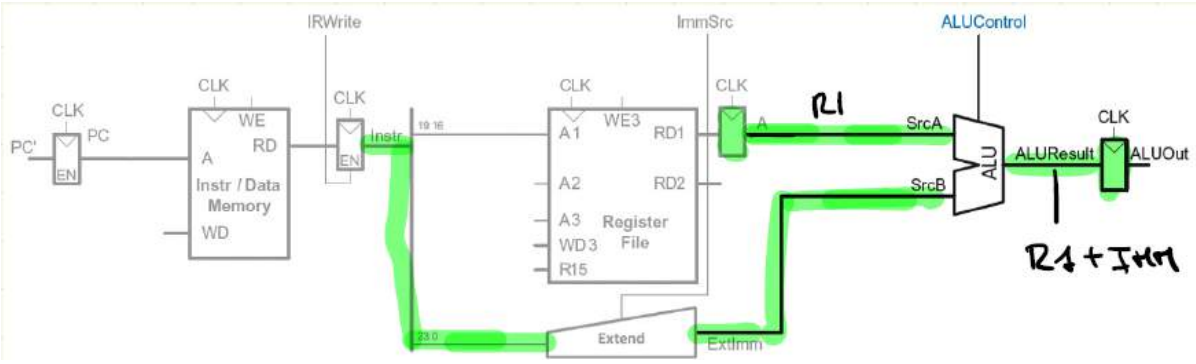
1) Fetch istruzione



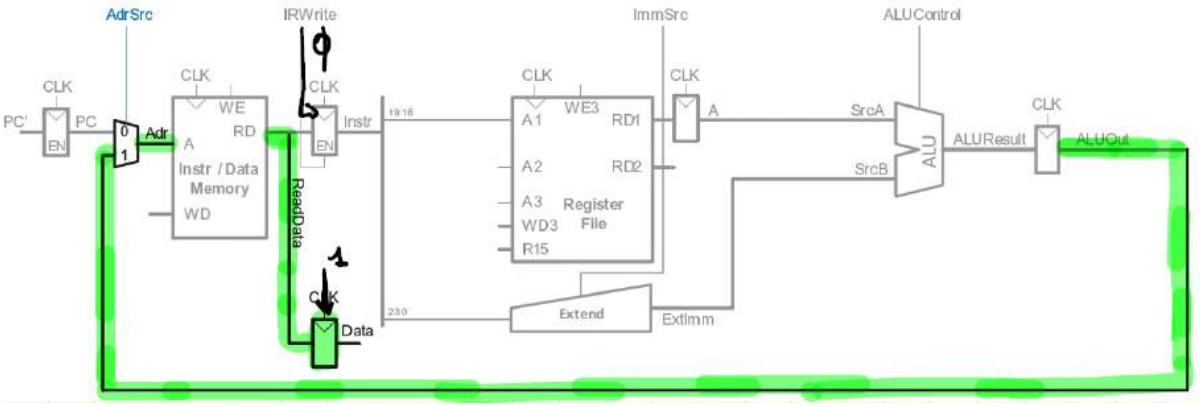
2) Lettura operandi (registro ed immediato)



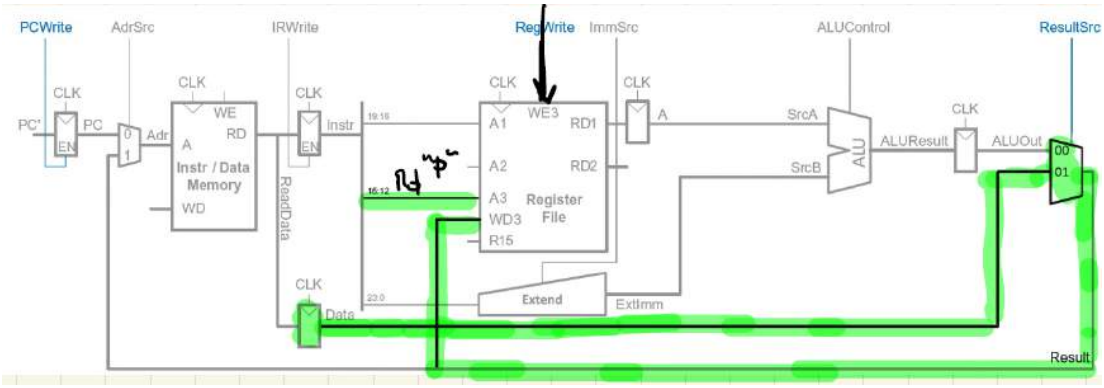
3) Calcolo indirizzo



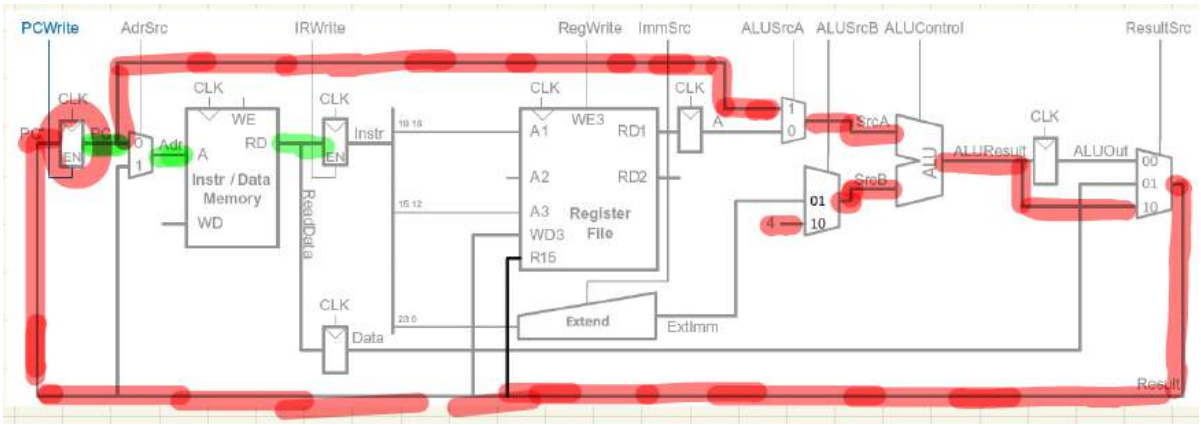
4) Accesso alla memoria



5) Scrittura risultato nel RF (WriteBack)

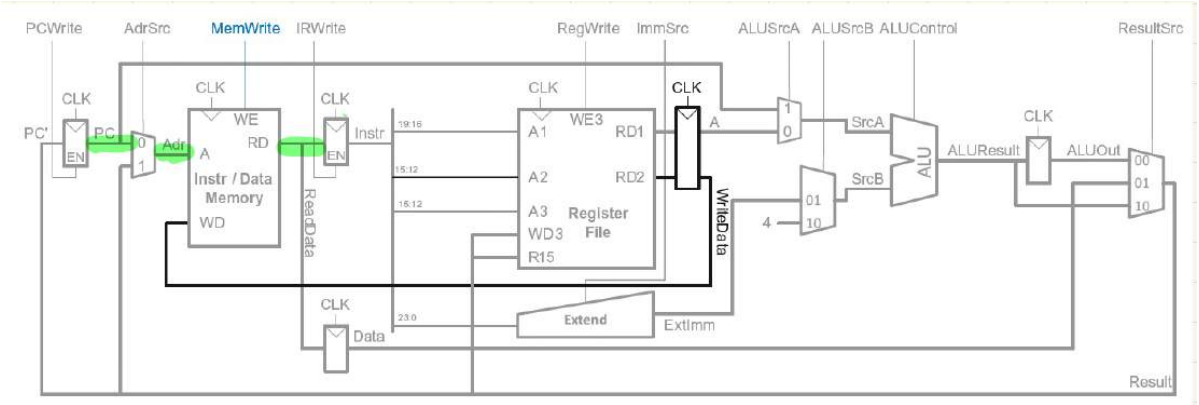


6) Incremento del PC

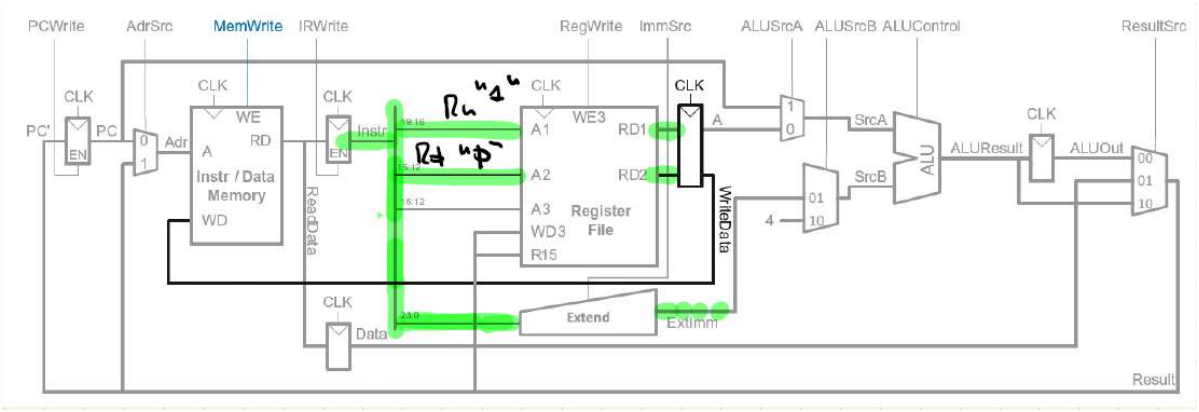


Esempio: STR r0, [r1, #imm12]

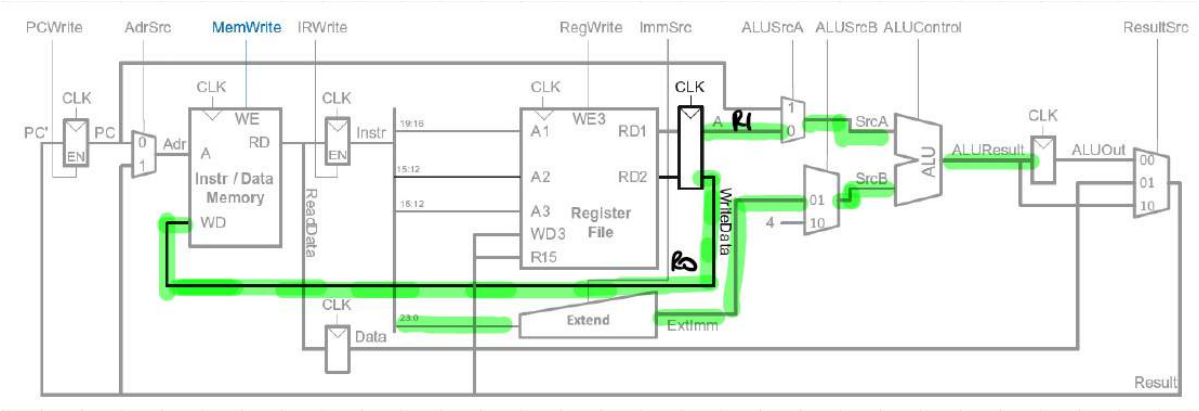
1) Fetch



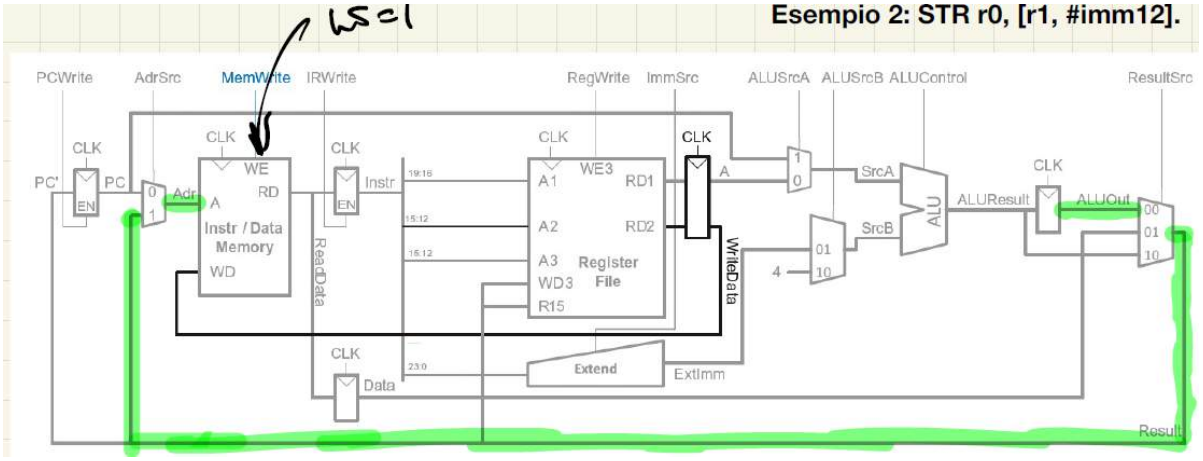
2) Decode



3) Execute



4) Memory



Analisi delle prestazioni

Mentre nella microarchitettura single cycle ogni istruzione occupa un singolo e lungo ciclo di clock, qui ogni istruzione ha un numero variabile di cicli di clock che effettuano operazioni più piccole. Quindi tanti cicli per istruzione di durata breve.
Si contano per le istruzioni di **salto 3 cicli**, per quelle di **elaborazione dati o di scrittura** in memoria **4 cicli** di clock mentre per le **letture in memoria 5 cicli di clock**. Per determinare il tempo minimo di ciclo di clock esamino i percorsi più critici della microarchitettura: **Execute** e **Memory**.

1. dal PC attraverso il multiplexer SrcA, l'ALU e il multiplexer del risultato
fino alla porta R15 del banco di registri;

2. da ALUOut attraverso i multiplexer Result e Adr fino alla lettura da memoria
nel registro Data:

$$T_{c2} = t_{pcq} + 2t_{mux} + \max[t_{ALU} + t_{\text{mux}}, t_{mem}] + t_{setup}$$

ESEMPIO calcolo del CPI medio
CPI del processore multi ciclo. Il benchmark SPECINT2000 è costituito all'incirca da 25% di istruzioni di lettura da memoria, 10% di scritture in memoria, 13% di salti e 52% di istruzioni di elaborazione dati.2 Determinare il CPI medio per questo benchmark.

Il CPI medio è la somma per tutte le istruzioni del CPI di ogni istruzione moltiplicato per la frazione di tempo per la quale l'istruzione è usata. Per questo benchmark si ottiene quindi: **CPI medio** = (0.13) × 3 + (0.52 + 0.10) × 4 + (0.25) × 5 = 4.12. Tale valore è migliore del caso pessimo di CPI = 5, che si avrebbe se tutte le istruzioni richiedessero il tempo più lungo per essere eseguite.

Quindi ci conviene usare la multi cycle quando le istruzioni che andiamo ad eseguire mantengono un CPI basso e quindi l'aumento del ciclo di clock + i relativi ritardi di sincronizzazione (T-setup, T-hold) rimangono marginali. **Vinco in efficienza quando l'aumento del CPI è più piccolo del decremento del ciclo di clock.**

load --> 5 cicli
store/operative --> 4 cicli
salti --> 3 cicli

Problematiche

Una delle principali motivazioni per passare ad una microarchitettura di tipo multiciclo era di non spendere più tempo di quello che serve per eseguire un'istruzione. In realtà il processore single cycle è migliore indipendentemente dalle tipologie delle istruzioni che vengono eseguite.
Nonostante l'istruzione più lenta (cioè LDR) sia divisa in cinque passi, il tempo di ciclo del processore multi ciclo non è migliorato di cinque volte, anzi.

Effettuare tanti cicli di clock per una singola istruzione porta ogni volta a dover aspettare i **ritardi in uscita del clock** e al **ritardo di setup**. Inoltre la durata del **ciclo di clock** viene scelto in base alla fase che richiede più tempo (tra Fetch, Decode, Execute, …), quindi per le altre fasi avrò tempo ridondante.

Il multiciclo fa comunque risparmiare 2 circuiti sommatori e unifica la memoria dati con quella delle istruzioni, anche se introduce 4 registri non architetturali e alcuni multiplexer aggiuntivi. Dunque non siamo riusciti a migliorare le prestazioni della microarchitettura single cycle.

Riassunto

- **FSM di controllo** più complessa → progettazione e verifica più onerose
- **Registri interstadio** introducono ritardi extra
- **Equilibrio**: risparmio HW e clock più corti vs. CPI aumentato

🔍 **Riassumendo**, la microarchitettura a multicycle ottimizza l'uso delle risorse e riduce l'hardware duplicato, frazionando ogni istruzione in passi atomici, ma richiede un'unità di controllo a stati più sofisticata ed introduce variabilità di CPI.

Pipeline

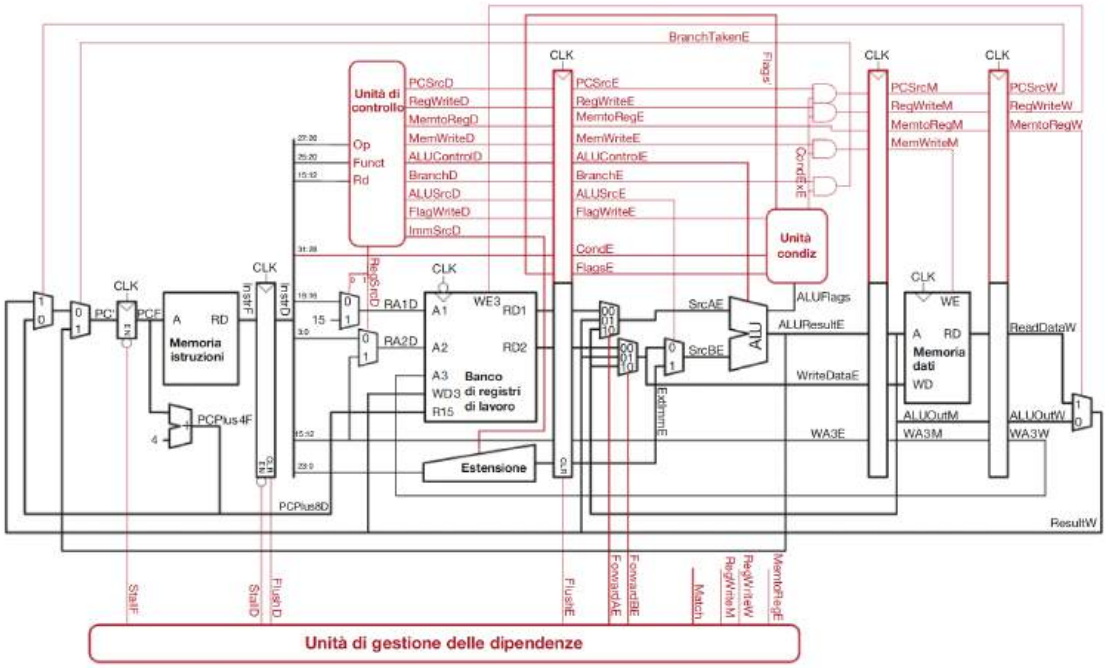


Figura 7.58 Processore pipeline con gestione completa delle dipendenze.

La microarchitettura pipeline suddivide il datapath della microarchitettura single cycle in **5 settori** che suddividono i vari passaggi del ciclo di esecuzione di una istruzione. Vengono quindi aggiunte **4 serie** di **registri** non architetturali per preservare i dati calcolati dopo ogni operazione. Ogni settore viene eseguito in un ciclo di clock ed essendo che ogni settore ha circa un quinto della tecnologia del single cycle, il ciclo di clock dovrebbe essere circa un quinto di quello originario senza nessun guadagno in termini di latenza di una singola istruzione. Viene però aumentato il carico di lavoro o **throughput**, che per calcolatori che eseguono miliardi di istruzioni conta molto di più della latenza di una singola istruzione.
Quindi ogni istruzione viene eseguita in rapida successione suddividendo il circuito in 5 aree, in cui ognuna lavora in maniera indipendente. In questo modo quando sono a pieno regime ogni ciclo di clock riesco a terminare un'istruzione.

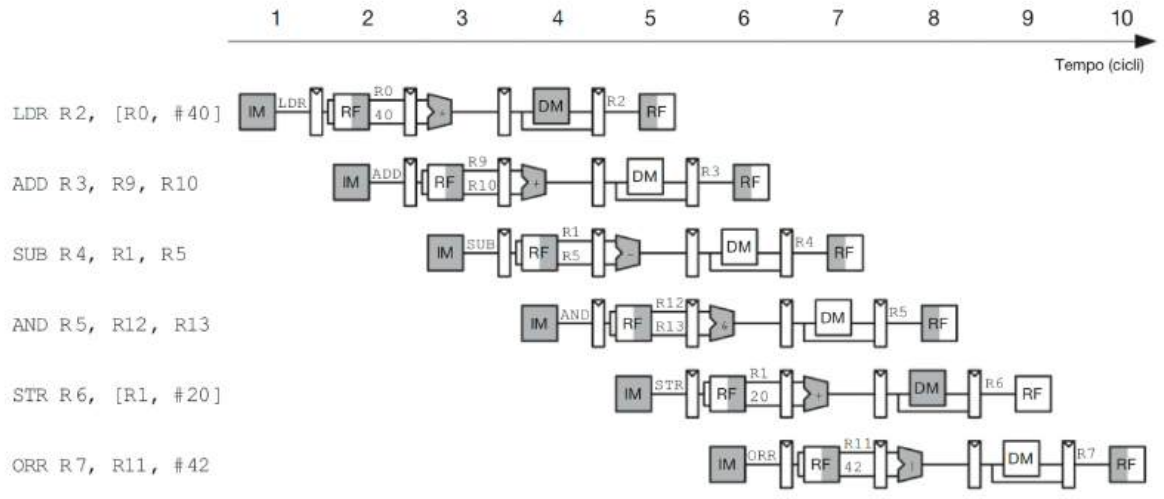


Figura 7.43 Rappresentazione schematica del funzionamento della pipeline.

Unità di controllo

Il processore pipeline usa gli stessi segnali di controllo del processore a ciclo singolo, quindi ha la stessa unità di controllo che prende in considerazione i campi OP e FUNCT dell'istruzione tramite il Decoder per generare i segnali di controllo. I **segnali** vengono però **salvati** nei **registri pipeline** per essere tramandati da una fase a un'altra.
In aggiunta a questo abbiamo altri **3 registri** nella parte di controllo, uno che trattiene i flag nella fase di esecuzione, uno che trattiene i flag per la memoria e un altro che trattiene i flag per il Write Back.

Data path

Il percorso dati della pipeline è ottenuto dividendo il datapath del processore a ciclo singolo in 5 stadi separati dai registri di pipeline.

Gestione delle dipendenze

Si verifica dipendenza (**hazard**) quando un'istruzione dipende dai risultati di un'istruzione che la precede e che non è ancora conclusa. Se ad esempio eseguiamo:

ADD R0, R1, R2
SUB -, R0, -

andremo a leggere il registro R0 contenente un valore sbagliato. Questo tipo di dipendenza viene denominato **RAW** (Read After Write, lettura dopo la scrittura).

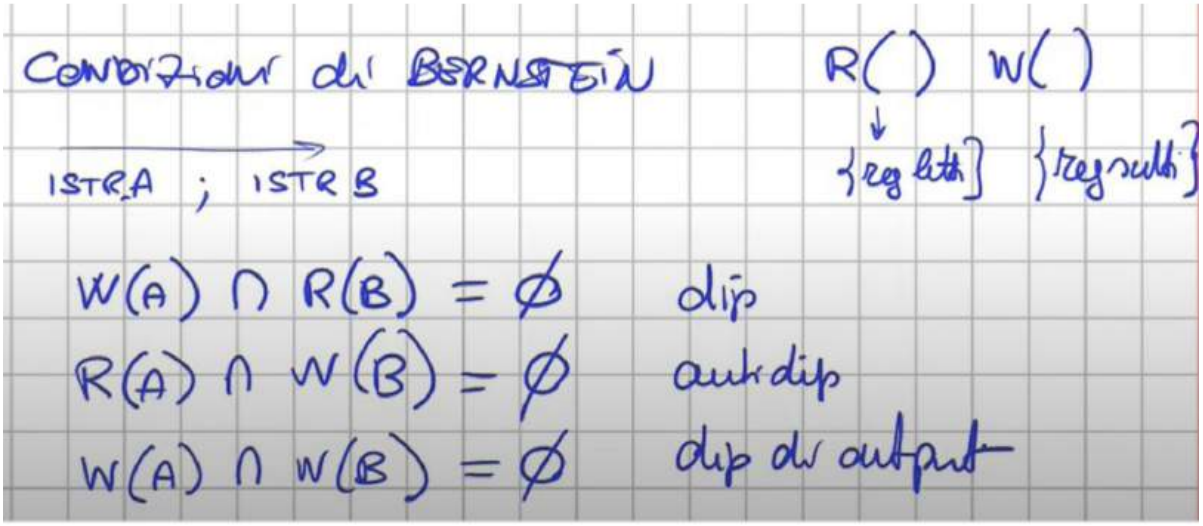
Approccio statico
se ho una dipendenza, il compilatore mette in mezzo alle 2 istruzioni dipendenti delle istruzioni indipendenti (eseguibili quindi in qualsiasi ordine) se non ci sono istruzioni indipendenti da mettere nel mezzo, vengono messe delle **nop**. Risolvo le dipendenze senza modificare il datapath della pipeline. Degrado delle prestazioni molto alto.

Approccio dinamico
Modifico il datapath andando a correggere le dipendenze nel momento nel quale si verificano.

Si suddividono in due tipi: una **dipendenza di dati** si verifica quando un'istruzione vuole leggere un registro che non è ancora stato aggiornato da un'istruzione precedente (RAW).
Una **dipendenza di controllo** (oppure di salto o scrittura del PC) si verifica quando la decisione di quale istruzione debba essere prelevata dalla memoria nella fase di fetch non è ancora stata presa al momento della fase di fetch.

Dipendenze e condizioni di Bernstein

Descrivono in modo formale le dipendenze tra le istruzioni, nella pipeline ci occupiamo solo delle dipendenze RAW.



Nel caso siano soddisfatte tutte le **condizioni di Bernstein** (anche Write after Write => **WAW** e Write after Read => **WAR**) si possono eseguire tutte le istruzioni in parallelo.

Tecnica del forwarding o inoltro

Alcuni tipi di dipendenze possono essere risolte tramite la tecnica del **forwarding.** Questa può risolvere le dipendenze di tipo RAW, infatti quando eseguiamo due istruzioni tipo

ADD R0, - , -
SUB -, R0, -

il risultato della ADD può essere **inoltrato** all'istruzione successiva senza farlo passare dallo stadio di memory e writeback.

In generale, la tecnica dell''inoltro è necessaria quando un'istruzione nello stadio Execute ha un registro sorgente coincidente con il registro destinazione delle istruzioni nello stadio Memory oppure Writeback.
L'unità di gestione delle dipendenze riceve quattro segnali di uguaglianza dal percorso dati che indicano se un registro sorgente nello stadio Execute è uguale al registro destinazione negli stadi Memory o Writeback e se effettivamente viene modificato. L'unità di gestione delle dipendenze genera i segnali per i multiplexer di inoltro per selezionare gli operandi dal banco dei registri oppure dai risultati negli stadi Memory o Writeback. Se entrambi gli stadi Memory e Writeback contengono registri destinazione uguali allo stesso registro sorgente di Execute, si deve dare la precedenza allo stadio Memory che contiene il valore più recente.

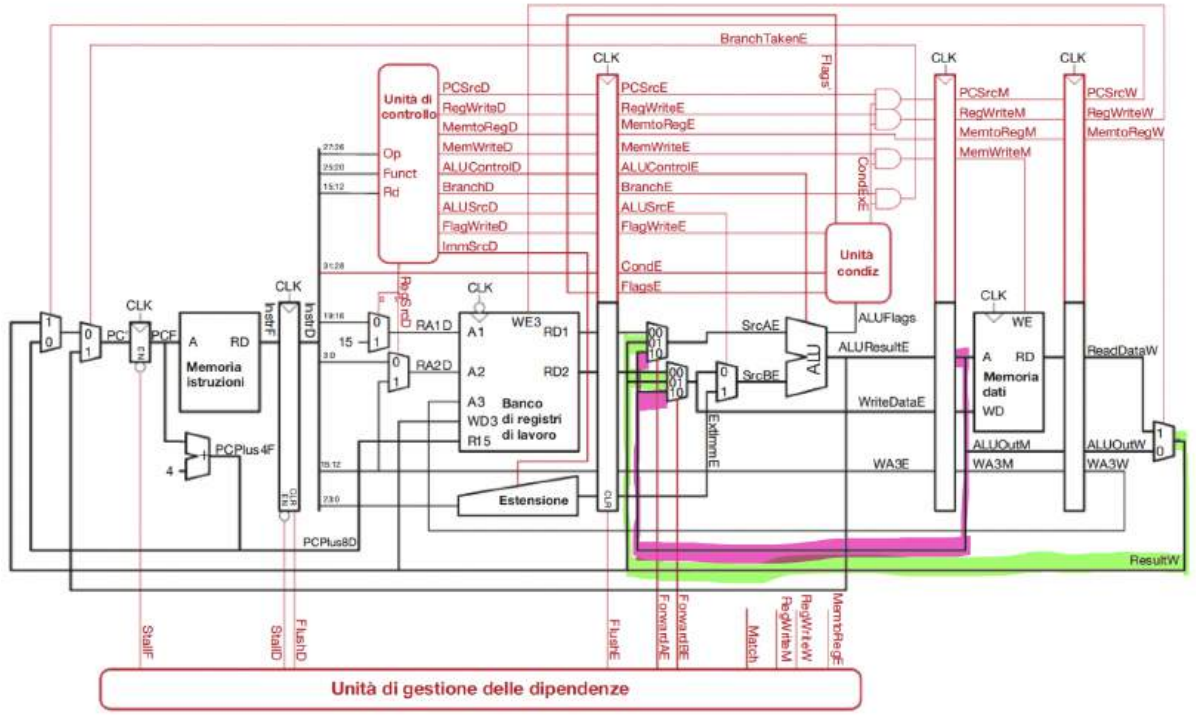
Questa operazione deve essere gestita da un'unità esterna che si occupa di memorizzare più istruzioni alla volta controllando la presenza o meno di dipendenze e gestendo opportunamente le flag per permettere l'inoltro, questa unità si chiama **Hazard unit** (o unità di gestione delle dipendenze). L'Hazard Unit comanda i multiplexer che gestiscono l'ingresso della ALU nello stadio di EXECUTE grazie ai flag **forwardeAE**, **forwardBE**. La gestione delle dipendenze da parte della Hazard unit conterrebbe molte più flag, complesse da implementare.

ATTENZIONE

LDR R0, [R1, R2, LSL #2]

SUB -, R0, -
ADD -, R0, - (c'è dipendenza anche con questa istruzione ma non crea stalli perchè ha distanza 2. Viene risolta tramite **forwarding**).

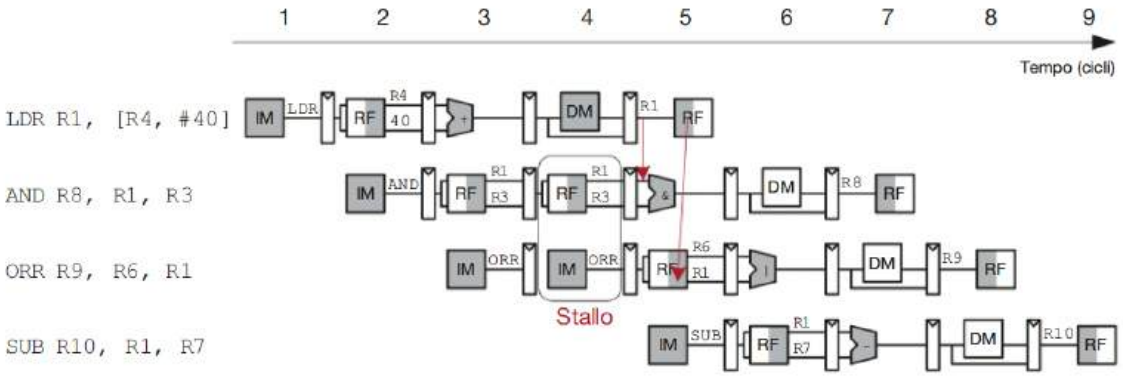
La tecnica di inoltro funziona solo se si tratta di istruzioni che calcolano il valore del registro in scrittura durante la fase di Execute. Infatti se si trattasse di una **LDR** dovremmo comunque aspettare l'accesso in memoria che richiederebbe un ciclo di clock aggiuntivo e questo renderebbe impossibile inoltrare il valore corretto del registro in tempo per l'istruzione successiva considerando la **distanza 1** tra le istruzioni. Per questo vediamo **un altro tipo di tecnica**.



In **rosa** è evidenziato il datapath del forwarding quando si tratta di una dipendenza con **distanza 1** tra le istruzioni ad **eccezione** della **LDR**. In **verde** invece quando la **distanza** è **2**.

Tecnica della bolla

Come abbiamo visto nell'esempio precedente nel caso di una LDR la dipendenza RAW non può essere risolta tramite inoltro si dice in questo caso che la LDR ha una latenza di 2 cicli. Per risolvere questo tipo di dipendenza devo necessariamente indurre lo stallo nella pipeline che mi permetta di eseguire la LDR ed avere il risultato corretto nell'istruzione successiva. Quando dobbiamo indurre lo stallo, **freeziamo** l'istruzione dopo la LDR, che è nello stato di decode, quindi la facciamo ritardare di un ciclo di clock. Nello stato di execute vengono **flushati** tutti i dati e si crea la cosiddetta bolla che verrà poi propagata negli stadi successivi. Intanto la LDR esegue nello stato di MEMORY e successivamente l'istruzione che è stata messa in stallo esegue lo stato di EXECUTE mentre la LDR esegue la WRITEBACK e scrive il valore corretto nei registri.



Quando serve generare uno stallo per LDR, StallD e StallF sono attivati per mettere in stallo gli stadi Decode e Fetch disabilitando la scrittura nei registri pipeline, viene attivato anche FlushE per resettare (Flip Flop D con reset) il contenuto del registro di pipeline dello stadio Decode, introducendo una bolla. Questo permette alla LDR di arrivare allo stato di Write back e di inoltrare il risultato. Questo viene ovviamente gestito dalla **Hazard Unit**. Questo stallo implica un forte degrado delle prestazioni quindi deve essere usato solo quando strettamente necessario.

Gestione delle dipendenze di controllo: scrittura di R15

L'istruzione B e le istruzioni di scrittura su R15 generano questo tipo di dipendenza.

Un modo per gestire questo tipo di dipendenza è quello di mandare in stallo la pipeline dato che la corretta decisione per il salto viene presa nella Write Back dovrò indurre uno stallo di 4 stadi con un notevole degrado delle prestazioni.

Un'altra opzione è quella di prevedere se il salto verrà fatto oppure no quindi iniziare l'esecuzione delle istruzioni sulla base di questa previsione e cancellarle nel caso si riveli sbagliata. L'architettura pipeline vista fino ad adesso assume che la decisione di salto non venga presa.

Per ridurre la penalizzazione dovuta al caricamento delle istruzioni prima che il salto venga calcolato posso applicare la tecnica del forwarding **anticipando** nello stato execute questo mi permette una volta calcolato il valore del PC con la ALU di inoltrarlo subito al registro tramite un commutatore in questo modo dopo dovrò inserire 2 bolle facendo la flush del solo settore Fetch e Decode.

Quando scrivo nel PC e devo aspettare la Write Back dovrò per forza indurre lo stallo bloccando la fetch dell'istruzione successiva e cancellando i dati presenti per evitare che riesegue due volte la stessa istruzione. Quindi avrò 3 cicli di penalizzazione (2 istruzioni sprecate + 1 del salto nella fetch).

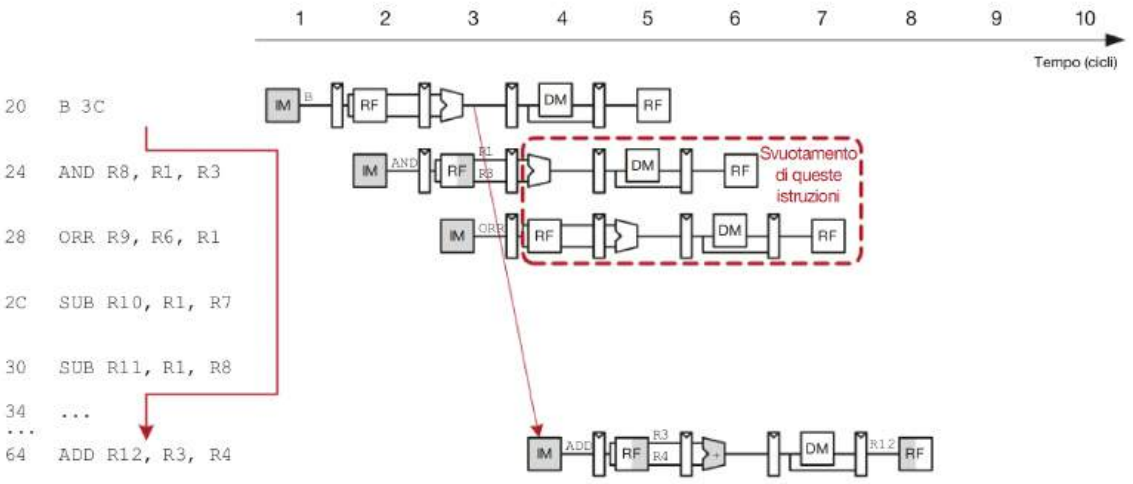
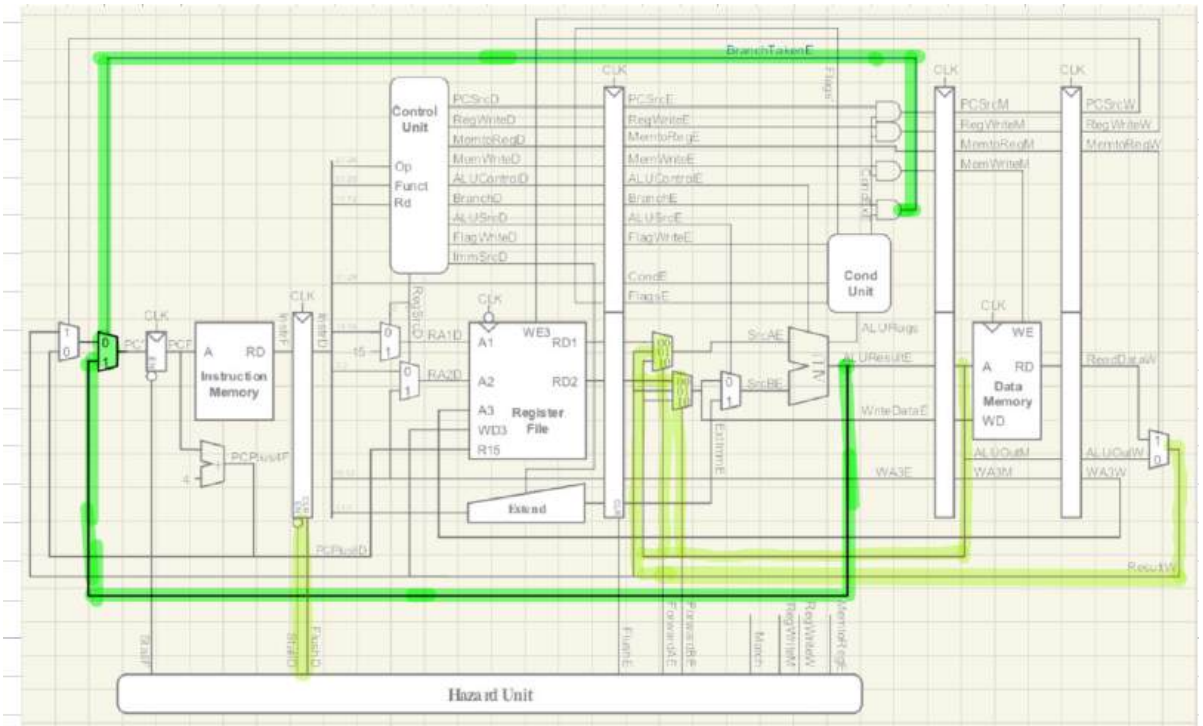


Figura 7.56 Rappresentazione schematica della pipeline con illustrazione dell'anticipazione del salto.

In caso di salto mal previsto andiamo a perdere 2 cicli di clock resettando la fase di Fetch e Decode.

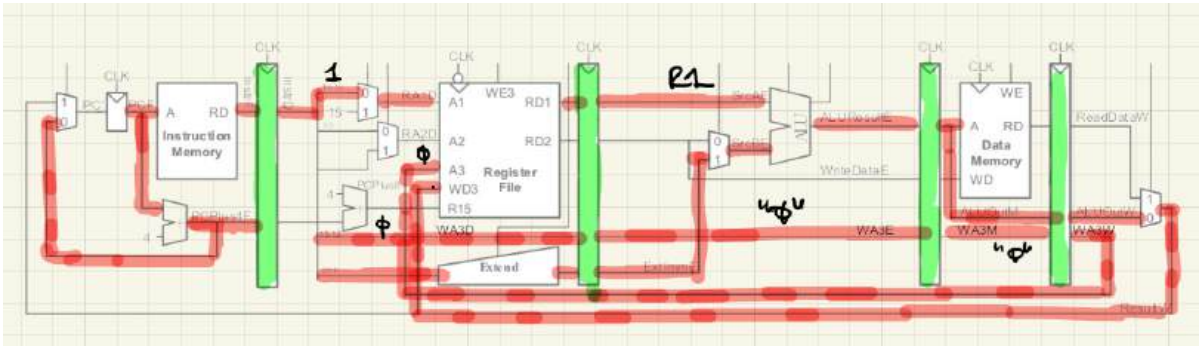
Possiamo andare a gestire le dipendenze RAW anche attraverso il riordinamento delle istruzioni se questo non varia il comportamento del codice ovviamente non può sempre esser fatto però in questo modo non andremo ad applicare stalli con un conseguente degrado delle prestazioni. Questo fa parte dell'ottimizzazione del codice.

Pipeline Completa

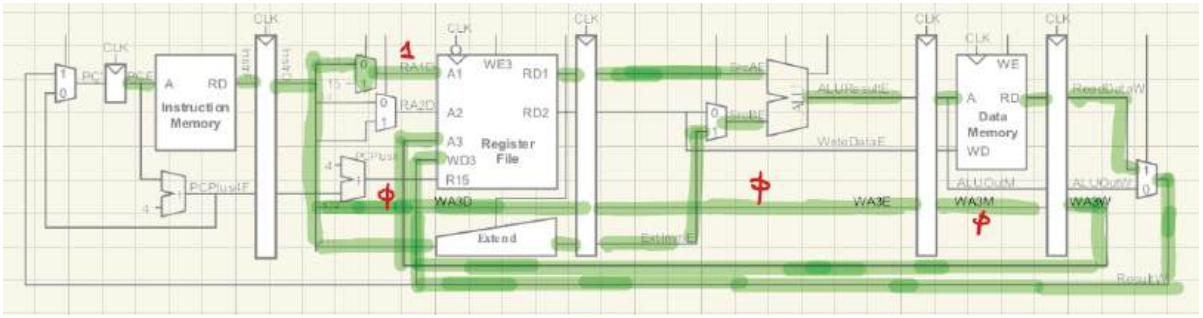


Esempi di operazioni

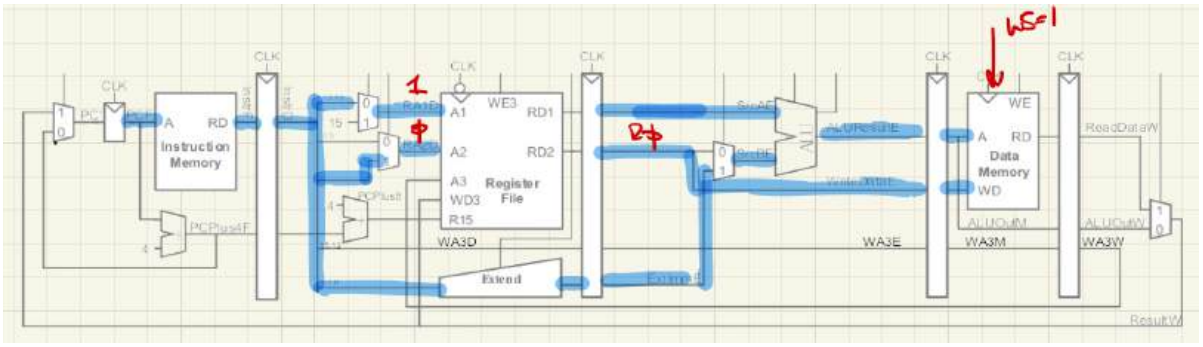
Esempio 1: ADD r0, r1, #imm8



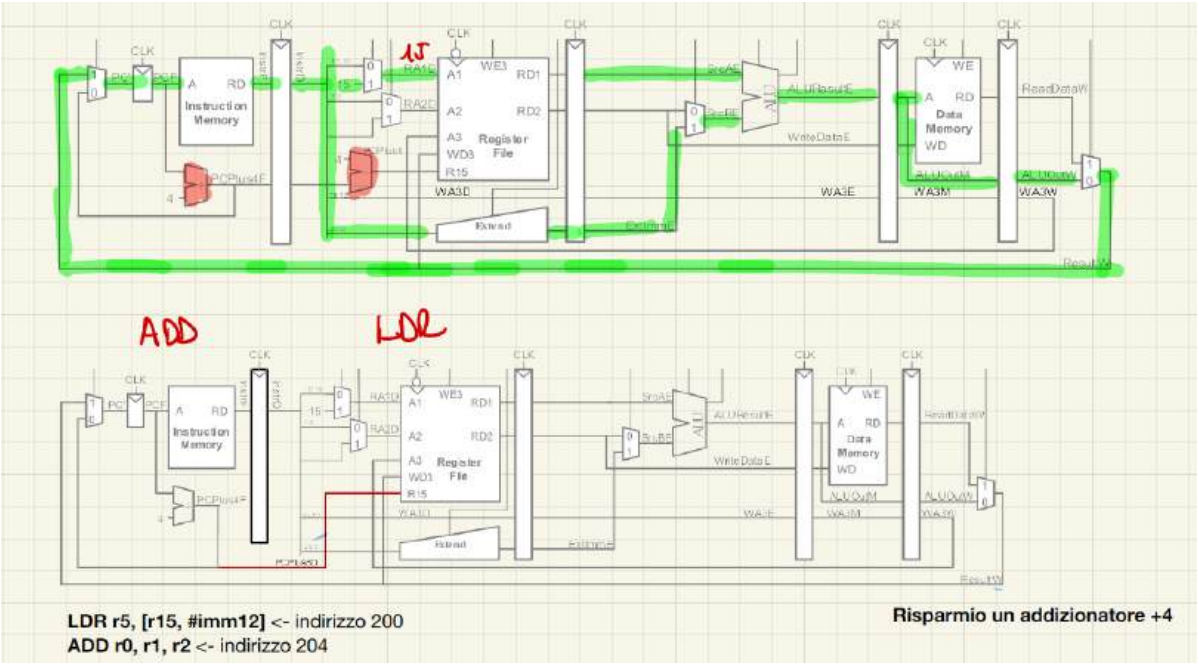
Esempio 2: LDR r0, [r1, #imm12]



Esempio 3: STR r0, [r1, #imm12]



Esempio 4: B label



Analisi di prestazioni

Idealmente il processore pipeline dovrebbe avere un CPI pari a 1, perché a ogni ciclo viene attivata una nuova istruzione. Tuttavia, gli stalli e gli svuotamenti sprecano cicli, quindi il CPI è un po' maggiore di 1 e dipende dallo specifico programma in esecuzione.

ESEMPIO CPI-medio

CPI del processore pipeline. Il benchmark SPECINT2000 considerato nell'Esempio 7.5 è costituito all'incirca da 25% di istruzioni di lettura da memoria, 10% di scritture in memoria, 13% di salti e 52% di istruzioni di elaborazione dati. Si faccia l'ipotesi che il 40% delle istruzioni di lettura da memoria sia immediatamente seguito da un'istruzione che usa il risultato della lettura, con la necessità quindi di introdurre uno stallo, e che il 50% dei salti sia da fare (quindi risultino mal previsti) richiedendo svuotamento. Calcolare il CPI medio del processore pipeline.

Il CPI medio è la somma per tutte le istruzioni del CPI di ogni istruzione moltiplicata per la frazione di tempo per la quale l'istruzione è usata. Le letture da memoria richiedono un ciclo di clock se non c'è dipendenza e due cicli di clock se il processore deve essere messo in stallo perché c'è dipendenza, quindi hanno un CPI pari a (0.6) × 1 + (0.4) × 2 = 1.4. I salti richiedono un ciclo di clock se sono previsti correttamente e tre cicli di clock se sono mal previsti, per cui hanno un CPI pari a (0.5) × 1 + (0.5) × 3 = 2.0. Tutte le altre istruzioni hanno un CPI pari a 1. Per questo benchmark, si ottiene quindi: CPImedio = (0.25) × 1.4 + (0.1) × 1 + (0.13) × 2.0 + (0.52) × 1 = 1.23.

La durata del ciclo di clock è data dal peggior percorso critico di ogni settore

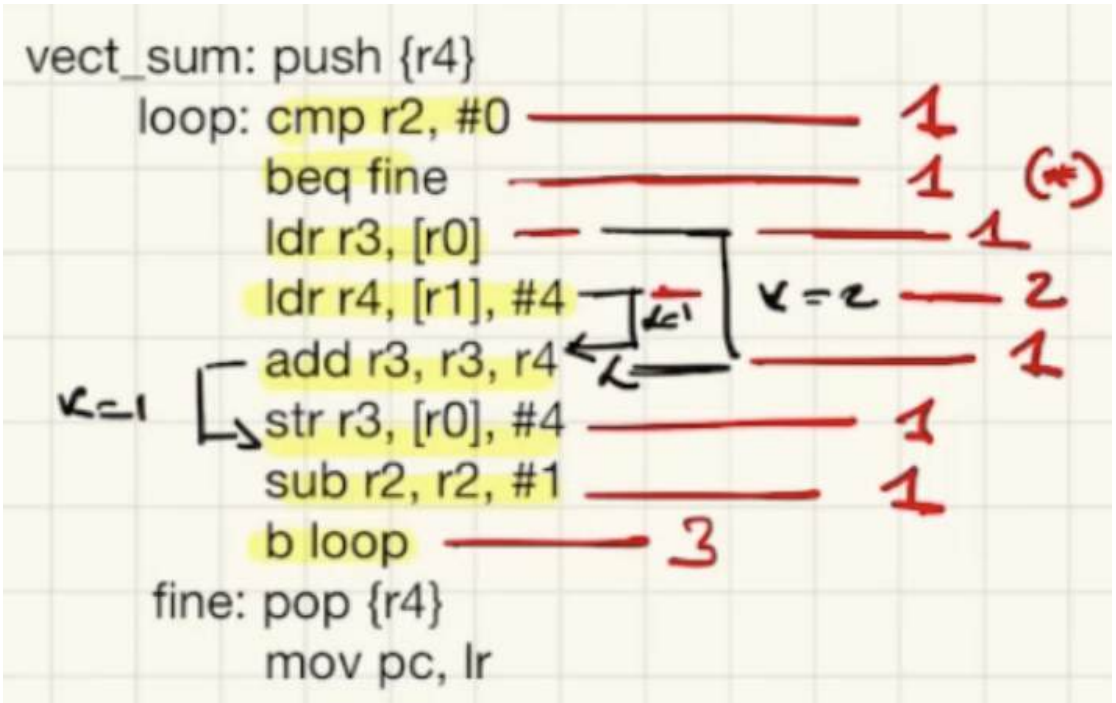
$$T_{c3} = \max \begin{bmatrix} t_{pcq} + t_{mem} + t_{setup} & \text{Fetch} \\ 2(t_{RFread} + t_{setup}) & \text{Decode} \\ t_{pcq} + 2t_{mux} + t_{ALU} + t_{setup} & \text{Execute} \\ t_{pcq} + t_{mem} + t_{setup} & \text{Memory} \\ 2(t_{pcq} + t_{mux} + t_{RFsetup}) & \text{Writeback} \end{bmatrix}$$

Confronto con le altre architetture
Il processore pipeline è significativamente **più veloce** degli altri due. Tuttavia il suo vantaggio sul processore a ciclo singolo è ben lontano dal fattore cinque che ci si potrebbe aspettare con una pipeline a cinque stadi a causa delle dipendenze, che introducono una penalizzazione in termini di CPI.
Più significativo è il ritardo dovuto ai **ritardi di stabilizzazione** del clock (setup, hold).

Problematiche

La problematica più grande è la gestione delle dipendenze che degrada le prestazioni di questa architettura. Idealmente si ha un CPI = 1 dato che le istruzioni vanno in pipe e vengono eseguite una dietro l'altra a causa della gestione delle dipendenze a seconda del codice da eseguire, avrò un CPI compreso tra 1-2.

esempio di codice con relativi CPI per istruzione



Riassunto

- **5 stadi**: IF→ID→EX→MEM→WB
- **Throughput**↑, latenza/istr. ≈ idem

Hazard unit

- **RAW**:
  - Forwarding → 0 cicli
  - Load-use bubble → 1 ciclo
- **Control (branch/PC write)**:
  - Stall fino a WB → 3 cicli
  - Predizione static-not-taken → 2 cicli mis-predicted
  - Early EX-resolution → 1–2 cicli

CPI reale ≈ 1.2–1.3 (>1 per stalli/flush)

Ciclo di vita di Cell.

Tecniche Avanzate

Per aumentare ulteriormente le prestazioni devo o ridurre il ciclo di clock o cicli per istruzione. Andiamo a vedere una serie di tecniche aggiuntive per aumentare le prestazioni della nostra microarchitettura.

Pipeline lunghe

Nei moderni processori abbiamo tra i 10-20 stadi, infatti sappiamo che aumentando il numero di stadi e suddividendo la logica della microarchitettura riduco il ciclo di clock, conseguentemente aumento le dipendenze e il costo della loro gestione mi porterà ad un degrado delle prestazioni.
Quindi il compromesso di questa tecnica sarà tra la riduzione del ciclo di clock e l'aumento del CPI per la gestione delle dipendenze.

Micro operazioni

Siamo abituati ad una architettura di tipo RISC con istruzioni semplici e facili da implementare a livello HW. Se prendiamo ad esempio l'architettura x86 che è di tipo CISC una singola istruzione è molto complessa e per essere eseguita in un solo ciclo di clock richiederebbe un alto costo HW e sarebbe inutilmente più lenta nell'eseguire le istruzioni più semplici e più frequenti nei programmi. L'utilizzo di istruzioni complesse a sua volta permette un risparmio di lettura in memoria che permette un risparmio di potenza. I progettisti delle microarchitetture devono decidere se fornire hardware per realizzare direttamente **una** operazione **complessa** oppure se suddividerla in una **sequenza** di **micro operazioni**. Naturalmente queste decisioni portano a punti differenti nello spazio progettuale prestazioni/potenza/costo.

Previsione dei salti

La previsione dei salti in una pipeline con più stadi è essenziale per rendere più efficiente la nostra microarchitettura e non **svuotare** la pipeline delle istruzioni che vengono caricate dopo un'istruzione di salto. Per questo vengono implementati dei veri e propri predittori di salto o ***branch predictor*** che permettono di mantenere la storia del programma e predire con maggiore accuratezza se il salto venga preso o no e calcolare il salto.

I cicli sono di solito eseguiti molte volte, quindi i salti all'**indietro** molto spesso sono da fare.
Abbiamo la **previsione statica dei salti** che senza conoscere la storia del programma effettuano delle previsioni come ad esempio dei salti all'indietro che vengono presi e i salti in avanti che non vengono presi.

La **previsione dinamica** dei **salti** che si basa sulla storia del programma per cercare di indovinare se il salto vada o meno eseguito. I predittori dinamici memorizzano in una tabella, denominata **buffer** delle **destinazioni** di **salto**, che include la destinazione di ciascun salto e la storia del salto, ovvero se sia stato o meno eseguito in passato.

Un predittore dinamico a **un bit** ricorda se il salto è stato eseguito oppure no l'ultima volta che è stato incontrato, sbagliando la previsione nella prima e nell'ultima iterazione di un ciclo.

Un predittore dinamico dei salti a **due bit** memorizza quattro stati relativi a ogni salto, denominati strongly taken, weakly taken, weakly not taken, strongly not taken.

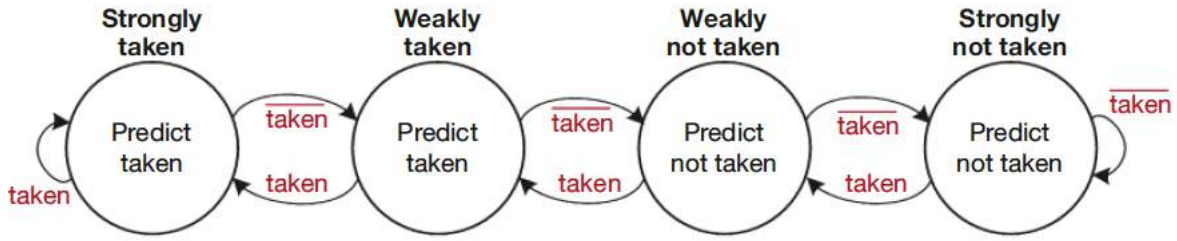


Figura 7.62 Diagramma degli stati per un predittore di salto a due bit.

Processore Superscalare

Un processore superscalare contiene più **copie** dell'**hardware** del percorso dati, per poter eseguire più istruzioni contemporaneamente. L'esecuzione simultanea di più istruzioni crea problemi per le dipendenze, non solo di dati ma anche di controllo come i **salti**.
I processori superscalari sfruttano entrambe le forme di parallelismo (spaziale e temporale) per "spremere" dalla microarchitettura tutte le prestazioni possibili.

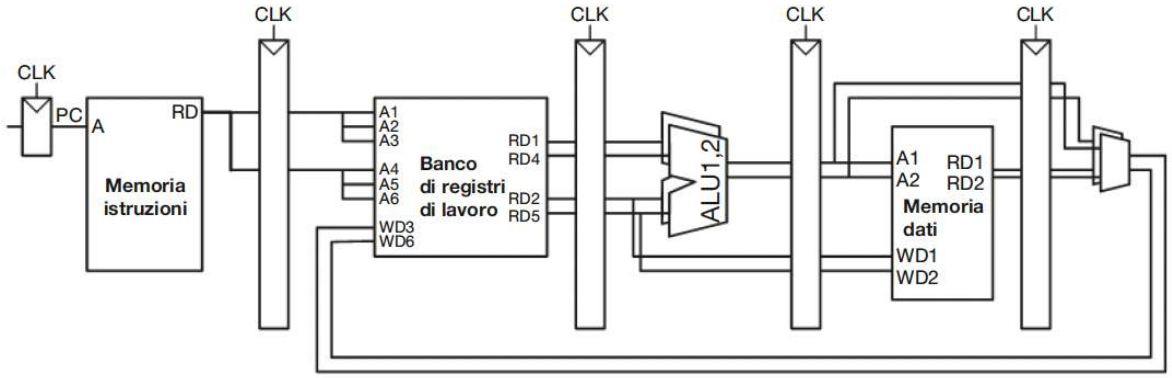
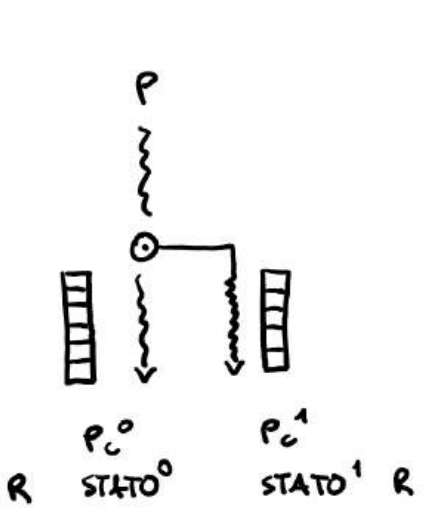


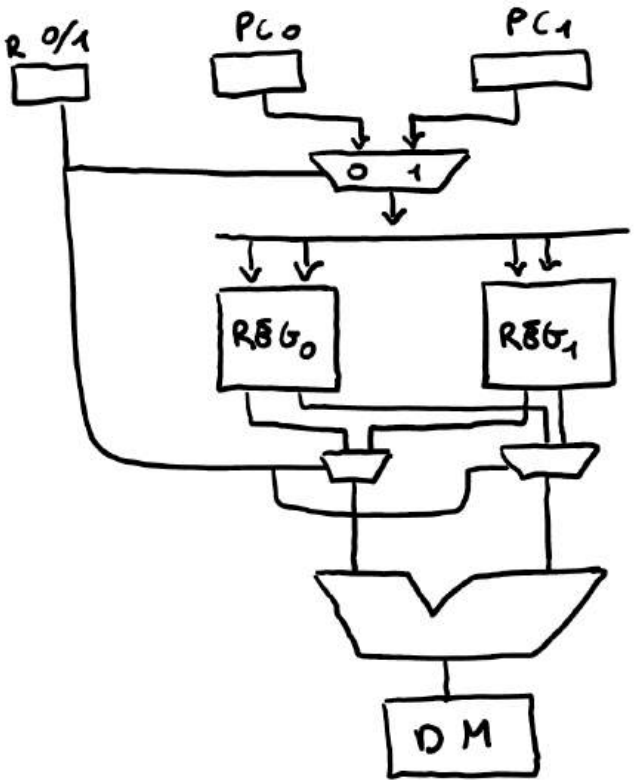
Figura 7.63 Percorso dati di tipo superscalare.

Processore Out-of-Order

I processori out-of-order usano una **tabella** per tenere traccia delle istruzioni che attendono di essere attivate; la tabella contiene informazioni riguardanti le **dipendenze** e la sua dimensione determina il numero di istruzioni che possono essere eseguite.
A ogni ciclo il processore esamina la tabella e attiva il maggior numero di istruzioni possibile, tenendo conto anche del numero di ALU e porte di memoria disponibili.
Il parallelismo a livello di istruzioni (ILP, Instruction Level Parallelism) è il numero di istruzioni che possono essere eseguite simultaneamente per un certo programma e una certa microarchitettura. **Deve tenere presente di tutti i tipi di dipendenze.**

Multithreading



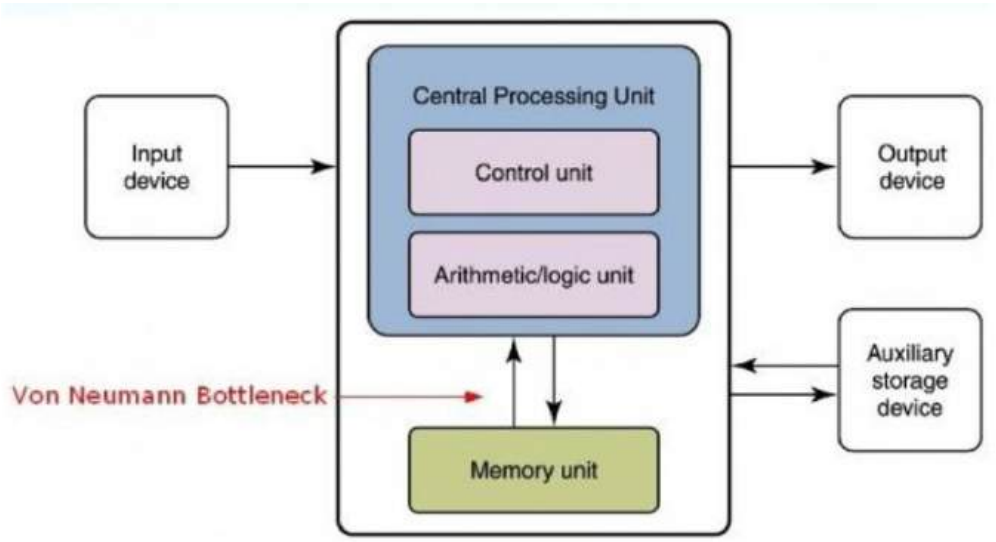


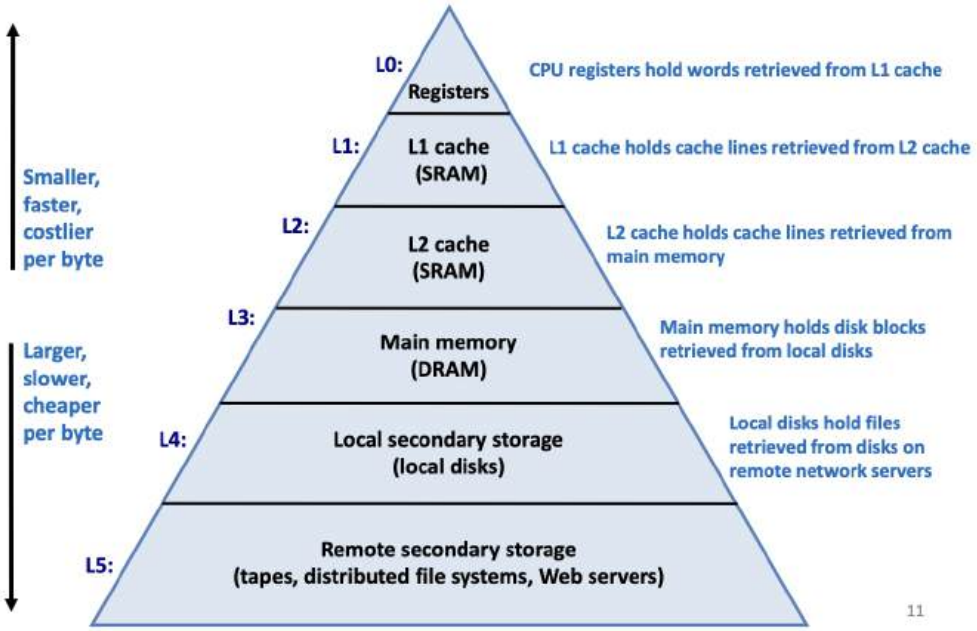
Posso immaginare un multiplexer che sceglie o il primo o il secondo PC, c'è poi la parte di decodifica che manda i segnali a tutti e due i register file, le cui uscite vanno in due multiplexer, comandati con la stessa cosa che comandava il multiplexer iniziale, quanto costa passare da un thread all'altro? Praticamente 0, in quanto basta cambiare il bit di controllo del multiplexer che esegue le scelte.

Memory Hierarchy

Memory Hierarchy

Problema del collo di bottiglia
L'architettura di Von Neumann ha un grosso problema ovvero Le prestazioni sono limitate dalla disparità di velocità tra la CPU e dalle velocità massime di trasferimento dati dalla memoria esterna al chip a quella interna (ad esempio, i registri).





Realizzando una gerarchia di memoria si crea l'illusione di avere una memoria sia grande che veloce.

Analisi delle prestazioni del sistema di memoria
Cache hit = quando il dato richiesto è trovato nella cache.
Cache miss = quando il dato richiesto non è trovato nella cache.
Hit rate = # cache hit / # accessi in memoria.
Miss rate = # cache miss / # accessi in memoria.

Miss penalty = # cicli di clock che perdo per ogni miss, è il tempo totale impiegato per recuperare il dato mancante e aggiornare la cache.
Miss time = miss penalty + hit time.
Tempo impiegato per un ciclo di clock = 1 / CPU SPEED (GHZ) (ns).
Average Memory Access Time (**AMAT**) = hit time + miss rate * miss penalty (posso moltiplicare tutto per il ClockCycleTime)

Esempio
Suppose $L1$ (the closest to processor) has a miss rate of 5% and hit time $tL1$, $L2$, which is larger, has a miss rate of 2% and hit time $tL2$. Finally, $L3$ the largest, has a hit rate of 100%. What's the AMAT?
AMAT= $tL1 + 0.05 * (tL2 + 0.02 * (tL3)) = tL1 + 0.05 * tL2 + 0.001 * tL3$

ClockCycleTime = tempo del ciclo di clock => 1/frequenza della cpu.

Per aumentare il tasso di hit in una memoria si usano **due principi fondamentali** nei programmi:
Località spaziale
significa che, se un dato viene usato, è probabile che anche i dati vicini vengano utilizzati presto. Ad esempio, quando leggi un elemento di un array, è probabile che leggerai anche quelli accanto.

Località temporale
significa che, se un dato è stato usato di recente, è probabile che venga usato di nuovo a breve. Ad esempio, una variabile dentro un ciclo viene spesso riutilizzata più volte in poco tempo.

Memoria Cache
La cache è organizzata in linee, ogni linea contiene un blocco di parole. La prima volta che viene richiesta una parola ci sarà un cache miss e il blocco in memoria contenente la parola viene caricato in cache (località spaziale).

CPU-time è il tempo in cui la cpu processa un certo programma.
CPU-time = #IC * #CPI * #ClockCycleTime

- IC = istruzioni da eseguire
- CPI = cicli di clock per ogni istruzione
- ClockCycleTime = tempo del ciclo di clock

Il CPI può essere suddiviso in **CPI-perfect** e **CPI-stall**, andando a considerare il CPI relativo alle istruzioni di calcolo cioè il CPI-cpu e il CPI relativo alle istruzioni di memoria posso calcolare il CPI-perfect/stall:

$$\text{CPI} = \underbrace{\frac{IC_{CPU}}{IC} * CPI_{CPU} + \frac{IC_{MEM}}{IC} * CPI_{MEM-HIT} +}_{CPI_{Perfect}}$$
$$\underbrace{\underbrace{\frac{IC_{MEM}}{IC} * Miss\ rate}_{\text{Miss rate per memory instruction}} * Miss\ penalty}_{CPI_{Stall}}$$

- **CPI-stall** = CPI-stall-instr + CPI-stall-data
- **CPI-stall-…** = memory instruction * miss rate * miss penalty
- **Degradazione del tempo di accesso in memoria** **=** CPU time with stall / CPU time perfect

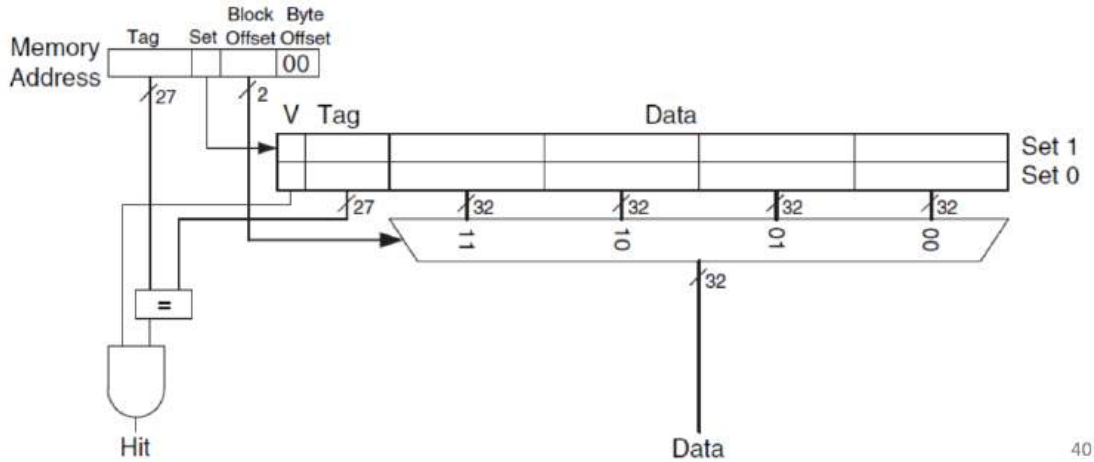
Design cache
Una cache può essere descritta da:
- C = capacità della cache (numero di parole b * B)
- S = numero di set, dove un set è un insieme di blocchi
- **B = numero di linee o blocchi**
- b = numero di parole per blocco o linea
- N = numero di blocchi per set

Schematicamente
- **Set** (insieme di blocchi)
  - **Blocchi** (#blocchi = N)
    - **b** = #parole per blocco

Mapping function => numero di blocchi in un set
1. **Direct mapped**: S = B;
2. **N-way set-associative: ogni set contiene N blocchi o linee**, S = B / N;
3. **Fully associative**: S = 1.

Memory address
- **Tag** = identificativo blocco o linea (bit restanti)
- **Set** = identificativo del set (log2(#S) bit) //nella fully associative non c'è
- **Block offset** = posizione parola (dato) nel blocco (log2(#b) bit)
- **Byte offset** = indice del byte della singola parola. (log2 (parola BYTE))

DIRECT MAPPED (tanti set quanti blocchi)



La cache usa un **bit di validità (V)** per ogni set che indica se il set contiene dati significativi. Se il bit di validità è 0, il contenuto del set non è significativo e avrò un cache miss.

Quando due indirizzi generati di recente dal processore si mappano nel medesimo blocco di cache, si verifica un conflitto, e il dato cui si accede per ultimo espelle il precedente dal blocco. Le cache a mappatura diretta hanno un solo blocco in ogni set, quindi **due indirizzi che si mappano nel medesimo set causano sempre un conflitto**.

PRO	Contro
Semplice da realizzare	Alto numero di conflitti
Molto veloce in caso di Hit	

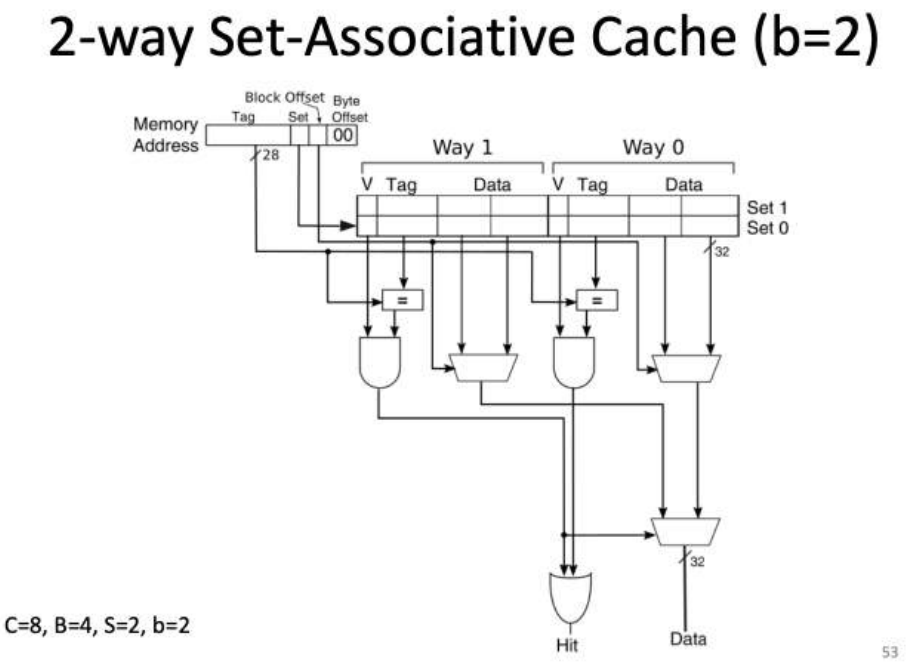
Per ridurre il problema dei conflitti troviamo altri tipi di soluzioni.

Associative Caches

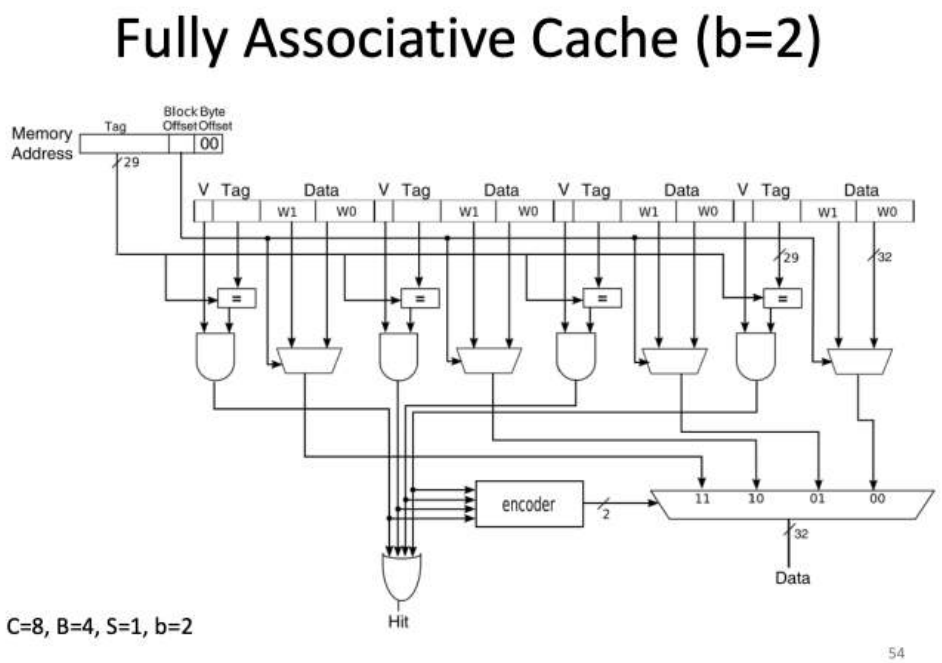
- **Fully associative cache** (1 set)
  - ❖ Ogni blocco in memoria può essere inserito in una qualsiasi linea di cache.
  - ❖ Per trovare un blocco devo scorrere tutti i blocchi.
  - ❖ Ogni blocco avrà un comparatore quindi incrementa il costo di implementazione.
- **N-way set associative cache**
  - ❖ Ogni blocco è mappato in esattamente un set ma ogni set contiene N blocchi
  - ❖ Un blocco può essere piazzato in ogni linea di cache di un set
  - ❖ Riduco così il numero di comparatori e costi delle fully associative cache

Il numero di comparatori è dato dal numero di blocchi per set.

Il numero di mux è dato dal numero di blocchi per set e la dimensione dal numero di parole



per blocco.



AND + CONFRONTATORE Decide se il blocco è quello giusto (controlla la corrispondenza col tag) e se il bit di validità V è uguale a 1 che indica se il dato trovato in cache è attendibile

MULTIPLEXER decide quale parola dal blocco prendere prende come input di controllo il block offset e le parole del blocco

ENCODER serve per decidere quale parola di quale blocco mandare in output.

MULTIPLEXER riceve in input di controllo 2 bit dall'encoder che gli dice quale parola selezionare la manda in output

Working set = tutti i dati utilizzati da un programma più di frequente.

Tipi di cache miss

- **Compulsory misses** = cache miss dovuto al primo accesso al blocco che non è mai stato in cache. (**INEVITABILE**)
- **Capacity misses** = sono causati quando **la cache non è in grado di ospitare** tutto il **working set**. Può essere risolto aumentando la capacità della cache o migliorando l'algoritmo.
- **Conflict misses** = **solo per direct mapped** e **N-way set-associative caches**, quando vado ad eliminare un blocco in cache perché ha lo stesso hash di un altro. Questo problema non si verifica nelle fully associative, mentre può essere aggirato attraverso le **victim caches**.

Victim cache = Una **victim cache** è una piccola cache fully-associativa posta fra la cache di primo livello (L1) e il livello di memoria successivo (L2 o RAM) il cui scopo è “salvare” le linee appena espulse dalla L1, per ridurre gli effetti negativi dei conflict miss.

OSS: Associativity reduces conflict misses; from 4 to 8 there is a small difference.

Metodi di riduzione del miss rate

- Incrementare la dimensione del blocco = trasportare più dati
    - Aumenta la **località spaziale** quindi riduco il miss rate, ma **aumenta la miss penalty** (perché devo caricare un po più dati).
- Aumentare l'associatività = aumentare i blocchi per set
    - Diminuisce i **conflict miss** ma aumenta l'hit time (scorro tutti i blocchi di un set).
- Incrementare la capacità = aumentare il numero totale di parole (b * B)
    - Meno **capacity miss** e conflitti ma un hit time maggiore.

Per aumentare le prestazioni del processore si possono inserire più livelli di cache per ridurre le miss penalty mantenendo un' alta velocità di risposta.

Multilevel cache

- CPI-stall = Miss rate l1 * miss penalty l1 + globalmissrate l2 * misspenalty l2 + globalmissrate l3 * misspenalty l3
- $GlobalMissRateLN = MissRateL1 * MissRateL2 * \ldots * MissRateLN$
- $MissPenaltyLN = HitTimeLN+1$

Gestione delle WRITE HITS scritture di dati in cache
Esistono 2 tecniche per la gestione di WRITE HIT:

- **Write-Through:** aggiorna sempre sia la cache che il prossimo livello di memoria.

Pro	Contro
Facile da implementare	La velocità è quella della memoria a basso livello
I dati in memoria mantengono la coerenza	Più traffico dati in memoria

- **Write-Back**: aggiorno solo la cache, aggiorno il dato in memoria solo quando quel dato viene eliminato dalla cache.

Pro	Contro
La velocità di scrittura è quella della cache	Più complesso da implementare con un **extra bit dirty bit**
Minore traffico di memoria rispetto la WT	Il cache replacement costa dipiù.

Per gestire il **Write MISS** ovvero quando il dato non viene trovato in cache ho due politiche:

1. **Write-allocate**: prima di riscriverlo viene caricato in cache, e dopo si esegue il WRITE HIT
    a. Usato prevalentemente con la politica di **WRITE-BACK**.
    b. Più lento perché carico il dato in CACHE, però sfrutto la località temporale assumendo che il dato venga riacceduto in seguito.
2. **No-write-allocate**: scrive la parola direttamente nel livello più basso di memoria, il blocco non viene caricato in cache
    a. Usato prevalentemente con la politica **WRITE-THROUGH**.
    b. Minore latenza dato che non devo caricare il blocco nella cache, ma non sfrutto la località temporale.

Ottimizzazione sulle scritture

Dato che le scritture in memoria sono particolarmente costose viene implementato un **write buffer** che mantiene i dati che devono essere scritti in memoria, quindi mentre il processore va avanti il dato viene scritto in background da un sottosistema che non interrompe l'esecuzione del programma.
Il processore va in **stallo** quando il write buffer è **pieno**.

Esempio di Write-back with Write Buffer

Il processore effettua la scrittura nel write buffer che riscrive in cache il dato in parallelo. Quando il blocco deve essere sostituito e il dirty bit è a 1 devo scrivermelo nel write buffer che penserà ad aggiornarlo in memoria successivamente, poi posso sostituirlo col nuovo blocco.

Cache replacement

- **Least Recently Used:** viene ucciso il blocco dall'accesso meno recente.
- **Random:** viene ucciso un blocco random per cache molto associative ha le stesse performance ad una complessità ridotta.

Design the memory system

Possiamo ridurre il miss penalty attraverso l'organizzazione della memoria:

- **Simple:** viene trasferita una parola alla volta
    - Semplice da implementare
    - Larghezza di banda limitata
- **Wide-memory:** vengono lette N parole alla volta
    - Viene allargato il bus dati
    - Non c'è un vero parallelismo le parole vengono trasferite a bocchi.
- **Interleaved:** K banche di memoria indipendenti operano in parallelo servendo K richieste contemporaneamente

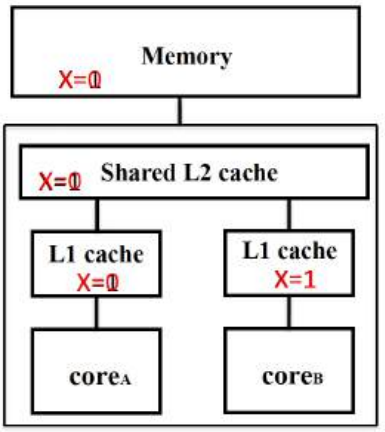
Problema di coerenza delle cache

L'uso della cache aumenta le prestazioni dei nostri calcolatori in termini di velocità di lettura e scrittura in memoria. **Nelle architetture multiprocessore** però introducono un nuovo problema che è quello di coerenza. Nei calcolatori multiprocessore ogni core ha le sue unità di cache private quando un core scrive un dato presente in cache non è aggiornato su tutti gli altri core che hanno quel dato nella loro cache.

Write-through caches

Time	Event	L1-core1	L1-core2	M
0				X=1
1	A reads X	X=1		X=1
2	B reads X	X=1	X=1	X=1
3	A writes X	X=0	X=1	X=0

The two L1 caches are not coherent!



Questo problema di coerenza può essere risolto tramite **Write Invalidate protocol** che durante la scrittura di un dato andrò ad invalidare i dati presenti nelle cache private degli altri core, o lo **snooping bus protocol** che assicura l'accesso esclusivo a quel dato.

False sharing

Def breve

due variabili non correlate sono nello stesso blocco di cache e sono accessibili in modalità lettura/scrittura da diversi thread dello stesso processo.

False sharing

- **Contesto**: due thread/proc. A e B su core diversi lavorano su due variabili distinte (A e B) non condivise semanticamente.
- **Perché succede**: entrambe le variabili finiscono per caso nella stessa linea di cache.
- **Effetto**: quando A scrive sulla sua variabile, invalida l'intera linea in tutti gli altri core → B subisce miss e reload sulla sua variabile, pur non condividendo i dati → thrashing di cache.
- **Soluzione tipica**: padding—allineare o inserire "riempimento" tra variabili in modo che ciascuna risieda su linee di cache differenti (almeno una linea di distanza).

I/O

Legge di Amdhal =

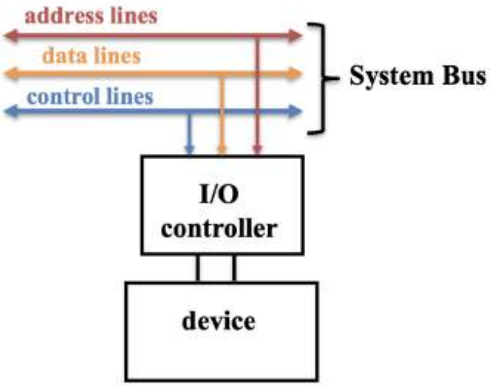
$$Speedup = \frac{Exec.time\ before\ enhancement}{Exec.time\ after\ enhancement} = \frac{T(1-f)+Tf}{T(1-f)+\frac{Tf}{N}} = \frac{1}{(1-f)+\frac{f}{N}}$$

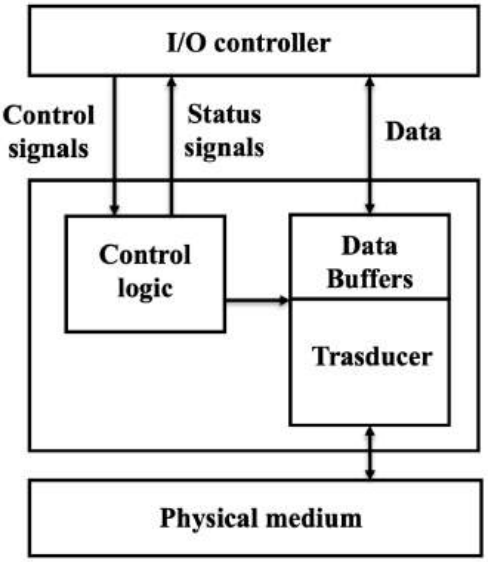
La velocità della CPU è superiormente limitata dalle operazioni di I/O. Possiamo migliorare le prestazioni della CPU quanto vogliamo ma saranno sempre penalizzate dalle istruzioni di I/O. La legge di Amdhal descrive il massimo potenziale di velocità di un sistema quando una sua porzione viene ottimizzata.

f rappresenta la parte ottimizzata, quindi (1-f) la parte non ottimizzata e N rappresenta il "grado" di ottimizzazione.

Un dispositivo di Input Output è composto da:
I/O = control + data

- **Control**
  - Indica l'operazione da fare, cosa deve fare il dispositivo (es. print a pdf)
  - Leggere lo stato dei dispositivi
- **Data**
  - Dove passano i dati





- Address lines used to select the specific device
- Data lines to send/receive data
- Control lines carry signals

Un controller I/O è un componente che gestisce la comunicazione tra CPU, memoria e dispositivi periferici, garantendo un trasferimento efficiente dei dati.

Step logici device-processor
• La CPU interroga lo stato del dispositivo del modulo I/O
• Il controller I/O restituisce lo stato del dispositivo
• Se pronto, la CPU invia un comando al controller per richiedere l'inizio del trasferimento dei dati
• Il controller I/O ottiene i dati dal dispositivo periferico

• Il controller I/O trasferisce i dati al processore

Possiamo vedere il controller I/O come tramite tra la CPU e il dispositivo fisico.

Bus Design

Il sistema di bus permette la connessione tra dispositivi fisici quali processore, memoria e I/O device.

Tipi di bus

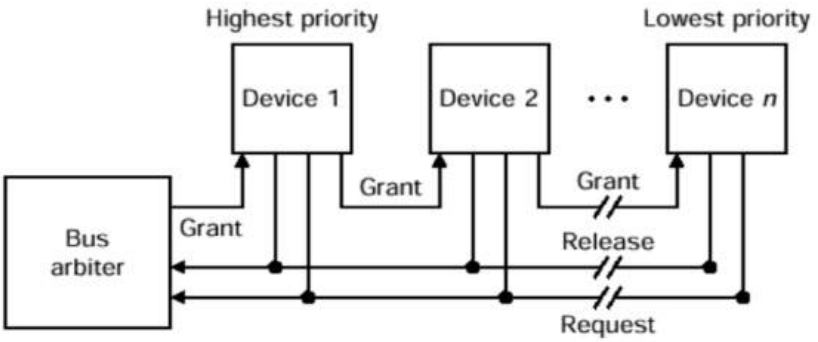
- **Bus dati** → Trasporta i dati tra CPU, memoria e dispositivi. Più è largo (ad es. 32 o 64 bit), più dati può trasferire in un ciclo.
- **Bus indirizzi** → Indica a quale posizione della memoria o quale dispositivo deve ricevere i dati.
- **Bus di controllo** → Trasmette segnali che coordinano il funzionamento del bus (es. lettura/scrittura, sincronizzazione).

Bus clocking

- **Bus sincroni**
    - Tutti i dispositivi collegati al bus condividono lo **stesso Clock** del bus
    - Il clock sincronizza la comunicazione tra i dispositivi.
    - **Contro:** *clock skew*. Dispositivi diversi ricevono il clock in tempi diversi dato dall'implementazione fisica del circuito stampato.
- **Bus asincroni**
    - Il bus non ha un clock, quindi niente clock skew
    - Viene utilizzato un **handshaking protocol** per sincronizzare i dispositivi quindi rende le comunicazioni più lente
    - Esempio:
- 1. UIO1 writes a READ (WRITE) request ReadReq (WriteReq) in the control line along with the address in the data line
  2. UIO2 reads the address and writes an ACK (acknowledge) into the control line to UIO1
  3. UIO2 writes the data on the bus and signal DataReady on the control line to notify UIO1
  4. UIO1 reads the data and sends an ACK into the control line to UIO2

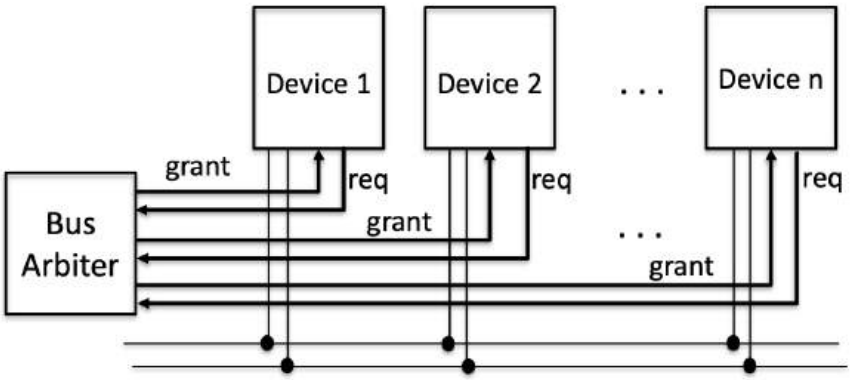
Le comunicazioni attraverso il BUS devono essere coordinate da un **protocollo di comunicazione** gestito dal **BUS ARBITER**. Abbiamo quindi il concetto di BUS MASTER che coordina le richieste dei dispositivi fisici e decide chi può trasmettere informazioni attraverso il BUS. Possiamo implementare questo modello in 2 modi differenti:

1. **Centralizzato**: un dispositivo dedicato colleziona le richieste da parte dei dispositivi fisici e decide a quale assegnare il bus.
    - **Daisy-chain**



2. Nella Daisy chain i dispositivi fisici sono ordinati per priorità quindi ottiene il BUS il dispositivo con priorità maggiore. Quando qualcuno fa richiesta l'arbitro scrive dentro alla linea grant dando possibilità di scrittura al primo dispositivo, se questo non ha fatto richiesta forwarda il segnale al secondo dispositivo e così via finchè non si incontra il dispositivo che voleva utilizzare il bus
   PRO:
   - Facile da implementare
3. **CONTRO**:
   - Lento e non assicura equità.

- **Linee indipendenti per request/reply**



Nell'implementazione con linee indipendenti invece ogni dispositivo fisico comunica attraverso una linea di bus indipendente col bus arbiter, quest'ultimo decide a chi dare il BUS attraverso

- **priorità**, quindi le richieste vengono classificate per priorità
- **algoritmo Round Robin**, che assicura equità.

- ***Distribuito***: sono i dispositivi stessi a decidere chi utilizza il bus.

I/O management

1. Come la cpu invia i comandi ai dispositivi di I/O?
2. Come fa la cpu a determinare quando il device ha completato la richiesta?
3. Come fa il dispositivo di input/output a trasferire i dati?

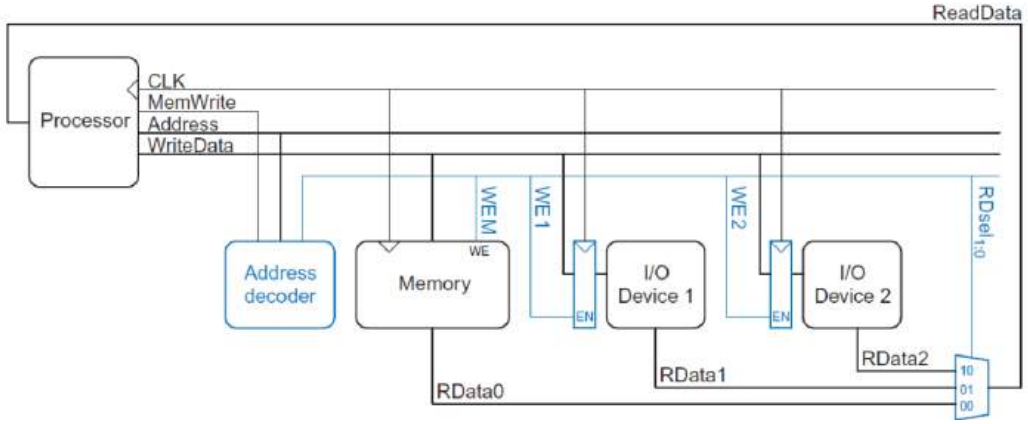
1. *Esistono due metodi per inviare comandi dalla cpu ai dispositivi di I/O*

- **Istruzioni I/O**

Istruzioni speciali definite dall'architettura che permettono di accedere ai registri dell'I/O

- **Memory mapped I/O**

Le comunicazioni agli I/O devices avviene assegnando ad ognuno di loro un indirizzo in memoria fisica che viene reindirizzato verso il registro del device con il quale vogliamo comunicare e ci permette di scrivere sul registro.



Read
print
..

Come fa la CPU a sapere se la richiesta di operazione è stata completata?

- **Programmed I/O**

Attraverso l'operazione di **polling** invia continuamente richieste di lettura del registro all' I/O device, rimane quindi in attesa finché l'I/O non ha terminato le operazioni.
(operazione *sincrona*)
Pro: facile da implementare
Contro: perdita di tempo della cpu
Utile per dispositivi a bassa banda (mouse, tastiera)

- **Interrupt-driven I/O**

Nella **Interrupt-Driven I/O**, invece di interrogare continuamente il dispositivo, la CPU lo lascia lavorare in autonomia e viene notificata tramite un **interrupt** quando il dispositivo è pronto.
La CPU **non è bloccata** in attesa dell'I/O, ma continua a eseguire altre istruzioni.
Quando il dispositivo è pronto, invia un **interrupt** alla CPU, che sospende l'esecuzione corrente per gestire l'I/O. Opera in maniera asincrona.
Più efficiente della Programmed I/O perché **riduce il tempo sprecato**.

Come fa il dispositivo i/o a trasferire i dati?

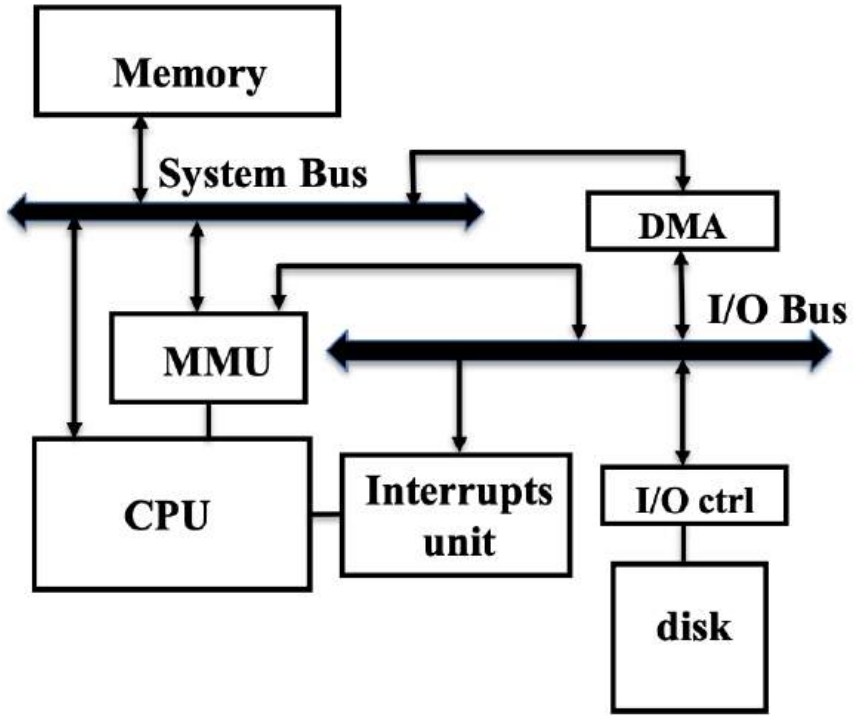
DMA
Il **Direct Memory Access (DMA)** è una tecnica che permette a un dispositivo di I/O di trasferire dati direttamente nella memoria senza coinvolgere continuamente la CPU

- Il **DMA** viene implementato con un **controller specializzato**.
    - Per eseguire un trasferimento **DMA**, un dispositivo di **I/O** viene collegato a un **controller DMA**.
    - Più dispositivi possono essere connessi allo stesso controller.

    - Il **controller DMA** si occupa dell'**arbitraggio del bus** (diverso comunque dal controller I/O) e del **trasferimento dei dati**
    - Il **controller DMA** e la **CPU** competono per l'accesso al **bus di memoria** (o **bus di sistema**).

Ci sono tre fasi in un trasferimento **DMA**:

1. La **CPU configura il DMA**, fornendo l'identità del dispositivo, l'operazione da eseguire, l'indirizzo di memoria e il numero di byte da trasferire.
2. Il **DMA avvia l'operazione** sul dispositivo e gestisce l'arbitraggio per la connessione. Trasferisce i dati dal dispositivo alla memoria (o viceversa).
3. Una volta completato il trasferimento **DMA**, il DMA controller invia un **interrupt** alla CPU per notificare il completamento.

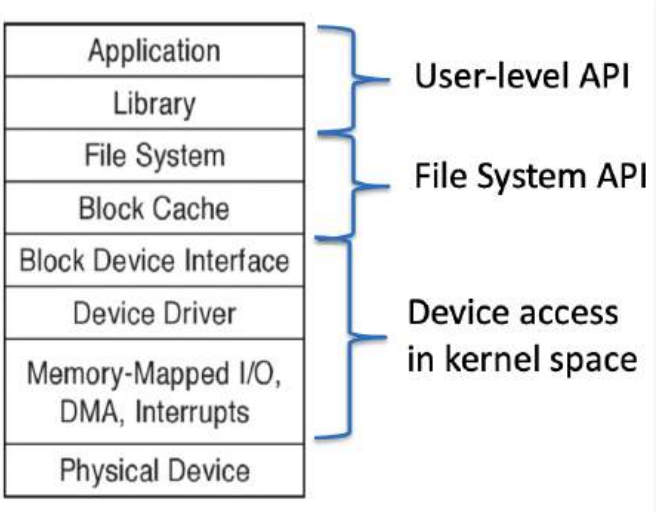


DMA e GERARCHIA DI MEMORIA

- **Virtual DMA:**
    - Il controller DMA ragiona con **indirizzi virtuali** deve fare la traduzione degli indirizzi internamente utilizzando una piccola cache (**TLB**) inizializzata dal sistema operativo quando richiede un trasferimento I/O.
    - Complesso ma flessibile (ad esempio, grandi trasferimenti tra pagine).
- **Physical DMA:**
    - Il controller DMA funziona con **indirizzi fisici**.
    - Più semplice, ma meno flessibile.
- **DMA e CACHE WRITE BACK (scrive in memoria sola o quando il dato viene scaricato, uso del dirty bit)**

Problema di coerenza dei dati, dato che la DMA scrive e legge direttamente nella memoria principale se ad esempio utilizziamo una cache write back potremmo **leggere** in memoria un dato non aggiornato alla versione corrente. Per risolvere questo problema possiamo utilizzare il **flushing** o l'invalidazione dei dati in cache.

Device Access
Un device access è la struttura del software attraverso il quale il sistema operativo comunica e controlla un device.



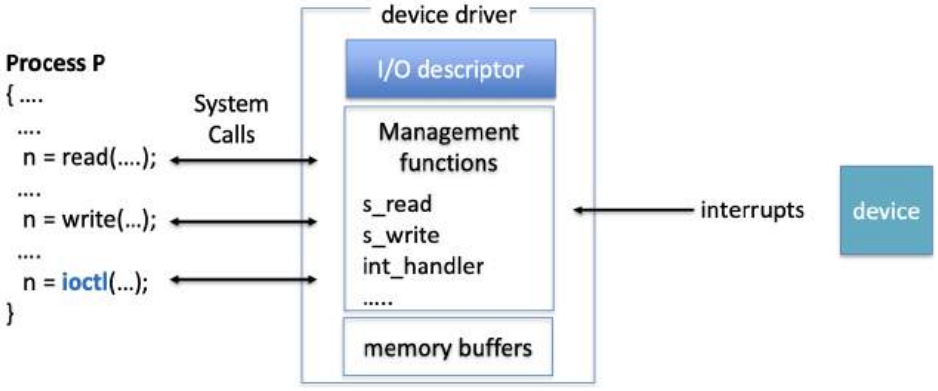
Dove i livelli più alti sono caratterizzati dall'indipendenza dall'hardware mentre il livello più basso è hardware dipendente dato che implementa le specifiche operazioni del device come l'**accesso ai registri** e i **protocolli** utilizzati.

L'obiettivo di un **device driver layer** è quello di nascondere le differenze tra i vari device e permettere al SO di gestire diversi device nel solito modo.

Ad esempio il file system viene usato per memorizzare e organizzare i dati nel computer. A livello del sistema operativo vengono implementate le politiche di caching e tutte le operazioni logiche rimanendo indipendente dall'hardware utilizzato.

Il **device driver** lavora nello spazio del kernel, si interfaccia direttamente col device quindi è **dipendente dall'hardware**, gestisce la sincronizzazione col dispositivo, il trasferimento dei dati (attraverso l'interfaccia del DMA) e i fallimenti dei device.

Le interazioni di un device driver sono definite da un **I/O descriptor** che contiene tutte le informazioni per interagire con il device specifico.



In questo modo tutte le applicazioni possono mandare comandi personalizzati al device driver che "conosce" il dispositivo e riesce a compiere tutte le operazioni.

Possiamo vedere il device driver come un tramite con il SO, che sa come funziona il dispositivo e le operazioni che può fare.

 OS

OS Intro + Kernel

Introduzione

Un SO è un programma che gestisce tutte le risorse di un computer per l'utente e le applicazioni.
I compiti più importanti del SO sono:

- La gestione delle richieste alle risorse
- La gestione della memoria virtuale
- La gestione dei processi e la loro comunicazione

Il SO ha 3 tipi di ruoli:

- Arbitro: gestisce le risorse tra utenti e applicazioni, isola le memorie dei vari utenti e quella delle varie applicazioni.
- Illusionista: ad ogni applicazione fa credere di avere tutte le risorse del computer anche se non è così.
- Collante: semplifica lo sviluppo di applicazioni, in generale ci permette di mettere le cose insieme e di farle funzionare.

Example: File Systems
• Referee
– Prevent users from accessing each other's files
without permission
– Even after a file is deleted and its space re-used
• Illusionist
– Files can grow (nearly) arbitrarily large
– Files persist even when the machine crashes in the
middle of a save
• Glue
– Common services: named directories, printf, …

SO design patterns
Il sistema operativo viene utilizzato in contesti molto diversi tra loro come Cloud, Web services, DBs, Game engine. Per questo vengono raggruppate delle proprietà comuni che il SO deve saper soddisfare.

Criteri del SO

- **Affidabilità**: il SO deve fare quello per cui è stato progettato ed essere sempre disponibile.
- **Sicurezza**: la privacy dei dati deve essere mantenuta, i dati devono rimanere inalterati e custoditi bene.
- **Portabilità**: significa poter usare lo stesso oggetto su macchine di diverso tipo
    - Per i programmi: vogliamo che le applicazioni che girano sul SO, girino senza problemi anche quando cambio l'hardware che ci sta sotto.
    - Per il SO stesso: deve poter girare su macchine con differente hardware.
- **Performance**: mantenere un overhead basso
- **Fairness & Predictability** Devo garantire equità per l'accesso alle risorse e devo sapere che un certa operazione finisce in un certo tempo.

Questi criteri non possono essere sempre tutti rispettati e questo porta a dei compromessi di progettazione dove si favoriscono maggiormente alcune proprietà rispetto alle altre.

SO struttura
Il sistema operativo è composto da moduli ognuno dei quali con una specifica funzione, questi moduli devono poter comunicare tra loro e possono essere strutturati in modi diversi. Questa struttura può essere progettata in 3 modi:

- **Monolithic kernel**
    - Tutte i moduli (le funzioni) fanno parte del SO. Questo approccio si usa per rendere il mio SO efficiente, sacrificando la modularità e la portabilità. Il SO è tutto insieme in un unico blocco del kernel.
    - L'astrazione viene garantita dai HAL (**Hardware Abstraction Layer**).
- **Micro Kernel**
    - In questo caso nel blocco del kernel vengono implementate poche funzioni, tipo lo schedulatore dei processi, la memoria virtuale e poco altro tutto il resto viene implementato al di sopra del microKernel. Sono affidabili e più facilmente mantenibili.
- **Modello ibrido**
    - Combina le tue pratiche

I primi SO

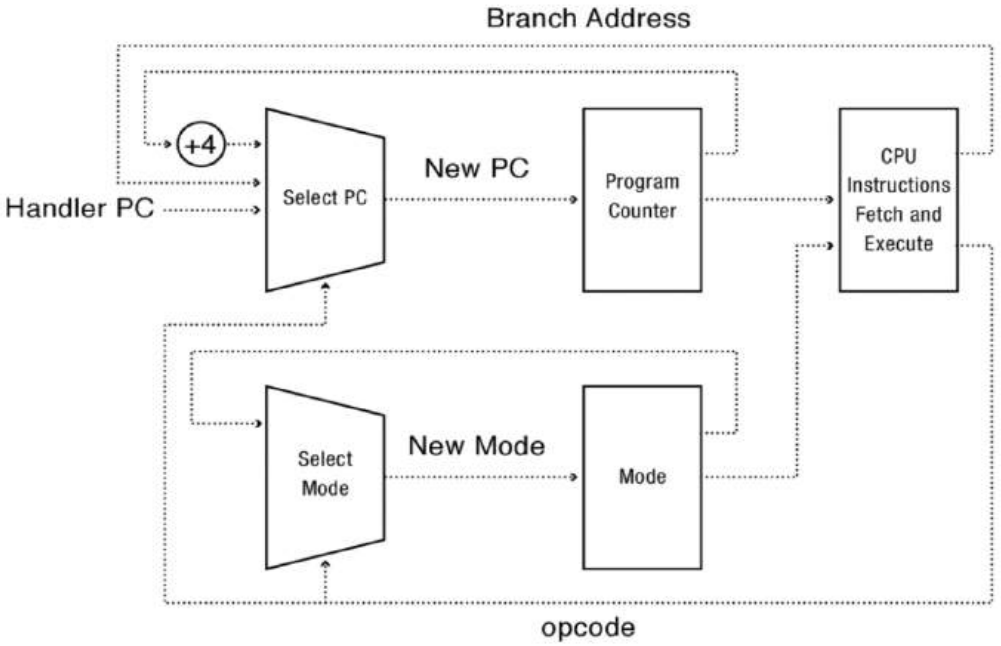
- Completavano un processo alla volta.
- I sistemi batch hanno invece una coda di task.

Kernel

È stato introdotto **il processo** cioè **un'astrazione del SO per eseguire un programma con privilegi limitati**.

Nel SO si introducono (almeno) due modalità:
User mode: non ho tutti i privilegi, alcune cose **non** sono permesse come il controllo di un I/O. Ha solo i privilegi che gli vengono assegnati dal SO.
Kernel mode: nessuna restrizione, il SO stesso esegue in kernel mode.

A CPU with Dual-Mode operation



Queste due modalità vengono differenziate dai bit nel registro **CPSR** (Current Program Status Register) per ARM. Questi bit vengono modificati solo dalla kernel mode.

Come implementare queste due modalità. Con focus sulla **protezione**, ovvero l'**accesso alla memoria**.

Concetto di processo
Programma sorgente => compilo => eseguibile (entità statica) → SO carica in memoria a partire da un certo indirizzo, questa entità che il mio SO esegue è il processo.

Programma => entità **statica**
Processo => entità **dinamica**

Quando il processo viene caricato i privilegi sono limitati, il processo quando viene caricato gli viene assegnato un **PCB** - **Process Control Block** - ovvero una struttura dati che il SO tiene per tenere traccia del processo.
Process table: contiene tutti i PCB
Gli elementi che caratterizzano un processo sono:

- La sua memoria (separata dagli altri processi).
- I thread al suo interno (processo leggero) condividono la memoria tra loro.

Cosa contiene il PCB

- **PID** process identifier index in process table
- Stato del processo: stato di attesa, esecuzione, terminato, waiting
- Registri CPU e informazioni sullo scheduling
- Puntatori ai thread cioè ai TCB (Thread control block)
- Memoria assegnata quindi dove inizia e dove finisce la memoria

- Puntatori alle risorse assegnate

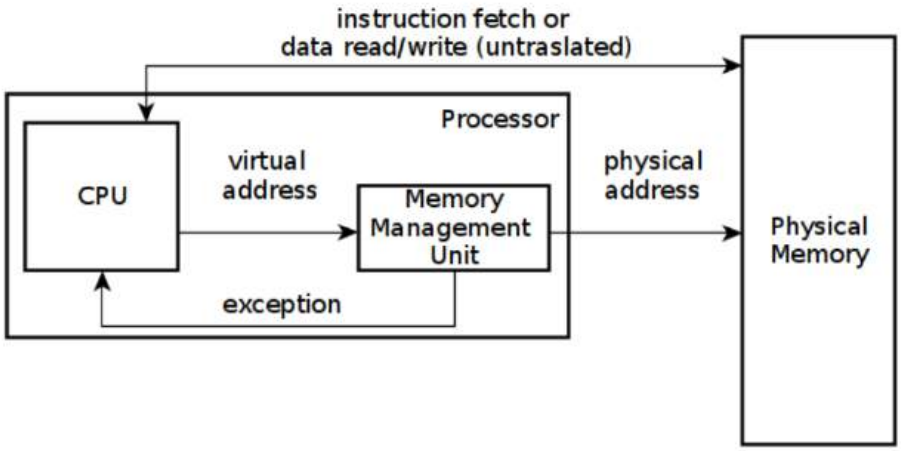
Per realizzare le due modalità abbiamo bisogno del **supporto hardware** per :

1. **Istruzioni privilegiate** disponibili solo per la kernel mode (cambio di utente, timer, accesso ai dispositivi I/O, **SVC**…)
2. Limitare l'accesso in memoria, viene fatto tramite l'astrazione del processo a cui viene assegnato dall' HW un indirizzo base e uno bound.
3. **Timer** che fa riprendere il controllo al kernel per interrompere in caso di loop while (true).
4. Permettere di **switchare** tra user e kernel mode, settando i bit del CPSR.

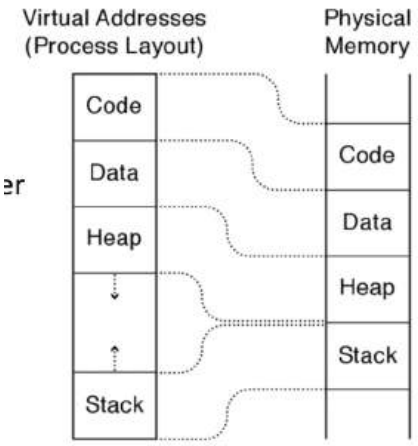
1. **Istruzioni privilegiate**
   - Disabilitare le interruzioni (ad esempio quelle del timer)
   - Setting the timer
   - Cambiare i bit al **CPSR** (current program status register) per fare lo switch tra le modalità

Se vengono eseguite in **user mode** le istruzioni privilegiate vengono lanciate le **eccezioni**.

2. **Protezione della memoria BASE BOUND**
   Ogni processo ha due registri Base e Bound, questo mi permette di controllare se un certo processo va fuori dalla memoria che gli è stata allocata. Quindi supponendo di lavorare con degli indirizzi fisici, possiamo controllare tutte le op di load e di store.
   Però a noi piacciono gli indirizzi di memoria virtuali o anche detti logici che ci permettono di estendere la memoria allocata per quel processo, di condividere memoria tra processi.
   Quindi ci piace il concetto di indirizzi virtuali:
   Vuol dire che c'è qualcosa che traduce l'indirizzo virtuale in un indirizzo fisico. Questo lo possiamo fare utilizzando una tabella **MMU** (Memory Management Unit).



Questo porta il processo ad essere rappresentato attraverso indirizzi virtuali che poi vengono tradotti in memoria da indirizzi fisici.



3. **Hardware timer**
    - Ritorna il controllo al kernel
    - La frequenza delle interruzioni viene decisa dal kernel
    - Le interruzioni possono essere modificate. Questo è cruciale per l'implementazione della mutua esclusione.

4. **Mode Switch**

Lo switch USER MODE a KERNEL MODE può avvenire per:

- **Interrupts** : lanciate da I/O o timer
- **Exception**: triggered da comportamenti anomali del programma
- **System call**: richiesta da parte del programma per un'operazione che viene richiesta al SO.

Lo switch da KERNEL MODE a USER MODE può avvenire per:

- Ritornare dalle interrupt, sys call, eccezioni
- Quando viene lanciato un nuovo programma o thread e devo effettuare il CONTEXT SWITCH
- **User level upcall**

Context switch: quando termino un processo o thread devo eliminare i dati dai registri e devo caricare il nuovo **PCB** nei registri.

Come effettuare l'operazione di INTERRUPT in maniera sicura, implementando una struttura dati riesco a gestire le interruzioni.

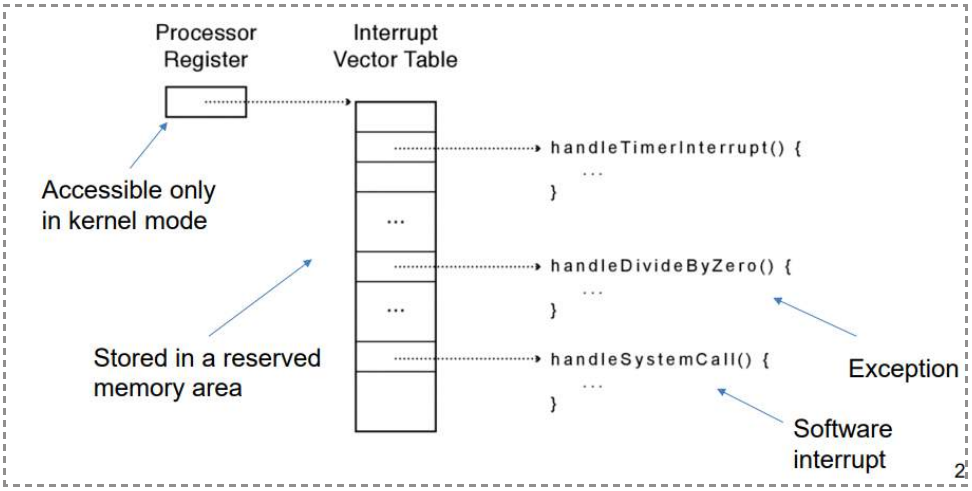
L'Interrupt Vector Table (IVT) è una **tabella di puntatori a funzione** memorizzata in una parte fissa della memoria dove:

- Ogni **voce** nella tabella corrisponde a un **interrupt specifico**.
    - Contiene **l'indirizzo dell'interrupt handler** che deve essere eseguito quando si verifica l'interrupt.
    - La CPU utilizza questa tabella per **trovare rapidamente il codice giusto da eseguire**.
- esempio INTERRUPT VECTOR TABLE**:**
  1) Un dispositivo (es. tastiera, timer, scheda di rete) genera un **interrupt**.
  2) La CPU sospende il programma attuale e consulta la **IVT**.

3) La CPU cerca l'**indirizzo dell'interrupt handler** corrispondente.
4) La CPU esegue l'**interrupt handler** per gestire l'evento.
5) Dopo la gestione dell'interrupt, la CPU **torna al programma originale**.

Kernel interrupt stack

Quando viene avviato il gestore di un interruzione, lo stack utilizzato non deve essere quello del processo che stava venendo eseguito fino a quel momento, dato che potrebbe in qualche modo corrompere il gestore. Per questo viene usato lo **stack del kernel**.



Interrupt masking

Tecnica che permette al kernel mode di disabilitare temporaneamente le interruzioni per evitare interruzioni durante operazioni sensibili.

Es. quando sto gestendo una interrupt non accetto altre interrupt.

Interrupt handler

Per evitare di disabilitare per troppo tempo il gestore delle interruzioni divido ogni interrupt handler in due parti:

- Parte inferiore (Bottom Half): contenente le operazioni più rapide che vengono effettuate mascherando il gestore delle interruzioni.
- Parte superiore (Top Half): contenente le operazioni più costose le quali vengono eseguite **non bloccando completamente** le funzionalità del kernel.

Esempio

- **Bottom Half** → Un interrupt della scheda di rete salva un pacchetto ricevuto in memoria.
- **Top Half** → Il sistema processa il pacchetto e lo passa al browser.

Atomic transfer of control (cosa succede quando avviene un'interrupt)

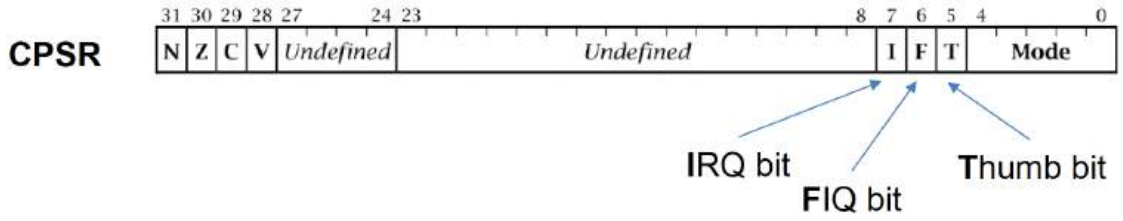
Esempio in x86

Quando andiamo a switchare dalla user mode alla kernel mode, SP e PC puntano rispettivamente al kernel stack e al codice dell'interrupt handler. Viene salvato nello stack il contesto del processo in user mode (PC, SP, PSW). Quando l'operazione di handler viene completata possiamo ritornare al processo in user level. Ripristinando PS,PC, SP, PSW, switch to user mode e riabilitando le interruzioni.

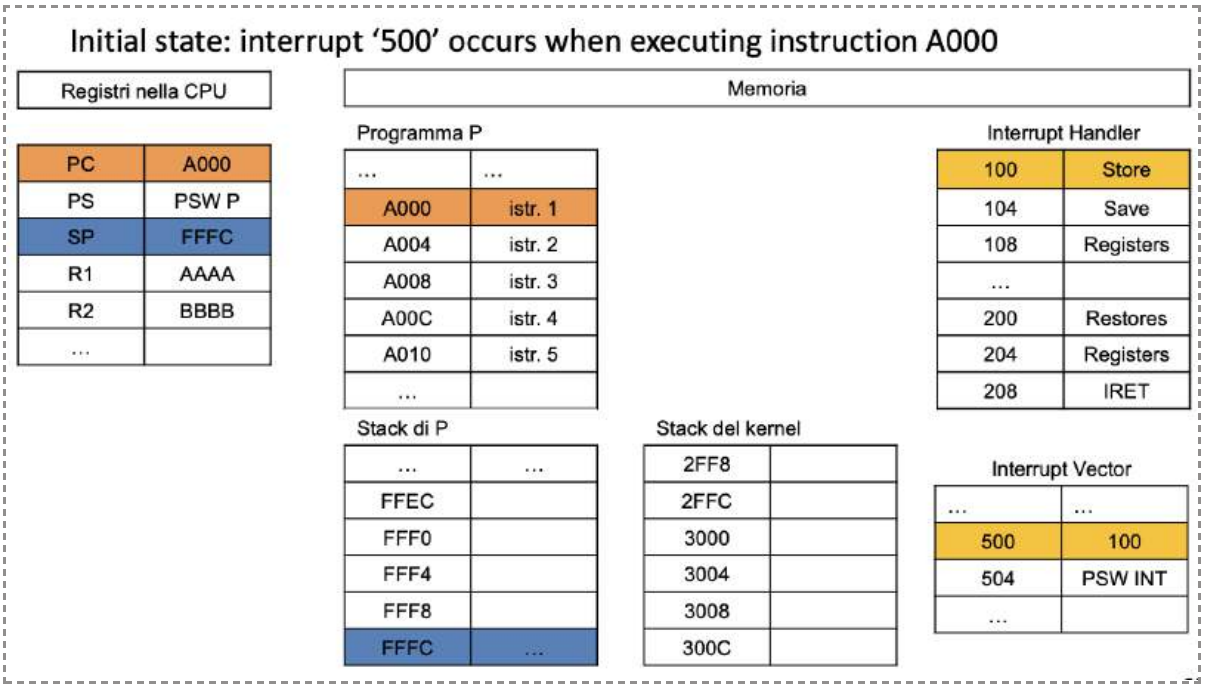
Gestione Interrupt

Stato iniziale in user mode dove viene lanciata l'interruzione
PSW = include il **CPSR**, **bit di controllo** (per modalità utente/kernel), e **flag di interrupt** (per abilitare/disabilitare gli interrupt).
Nella situazione rappresentata sotto, **PS = PSW P** vuol dire che il processore è attualmente in modalità
Programma P (user mode), e il contenuto del PS riflette questa modalità.
IRQ =
FIQ =
THUMB =

L'istruzione **IRET** (**I**nterrupt **RET**urn) serve per il ritorno dal gestore di interrupt. Espelle PC, SP e PSW dallo stack del kernel e li ripristina atomicamente! Questo ci permette di tornare al contesto pre-interruzione.

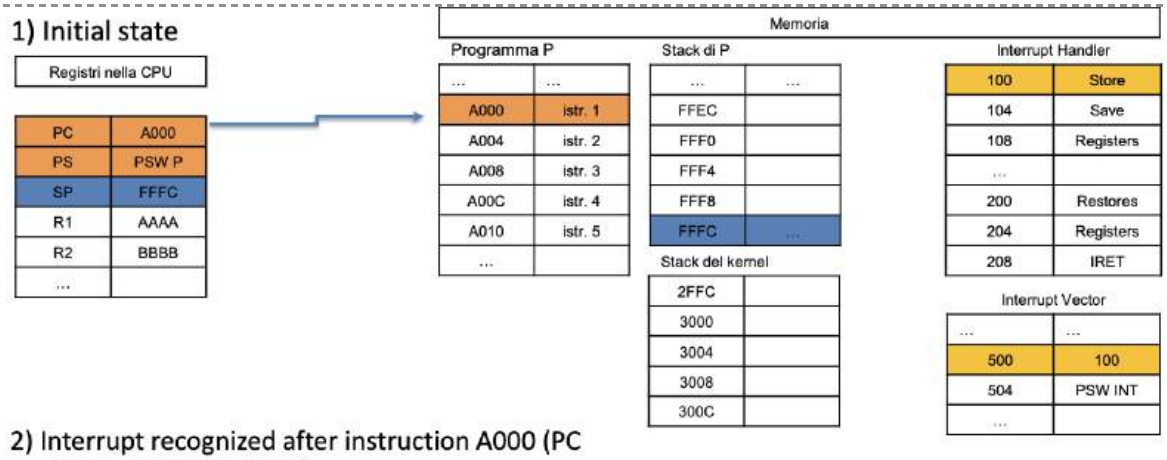


Esempio

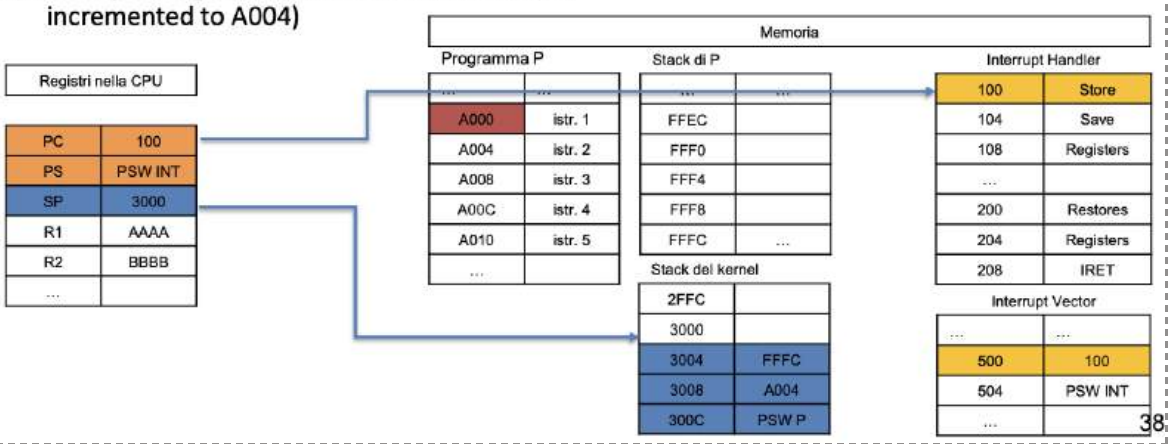


Switch in kernel mode, in questo modo inizio ad eseguire l'interrupt handler indirizzato dall'interrupt vector.

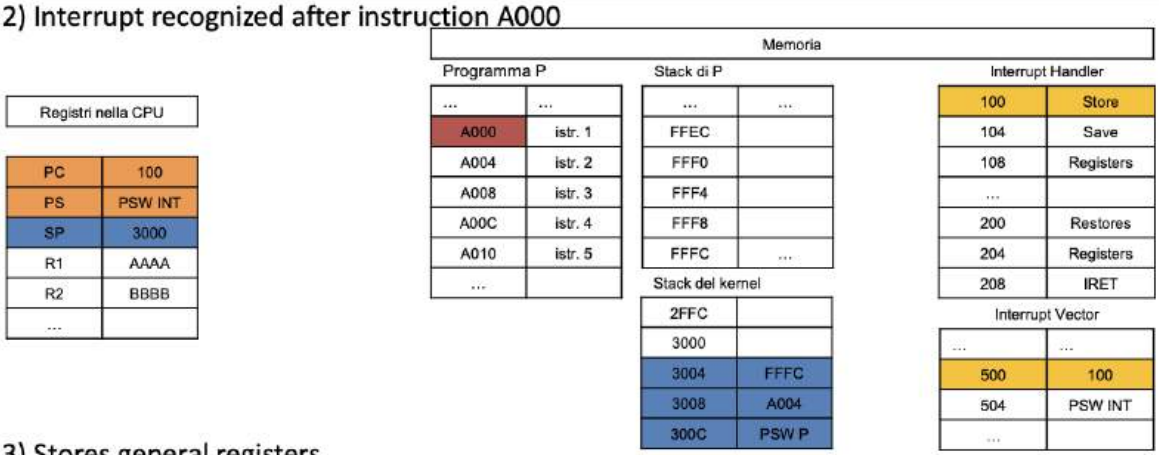
1) Initial state



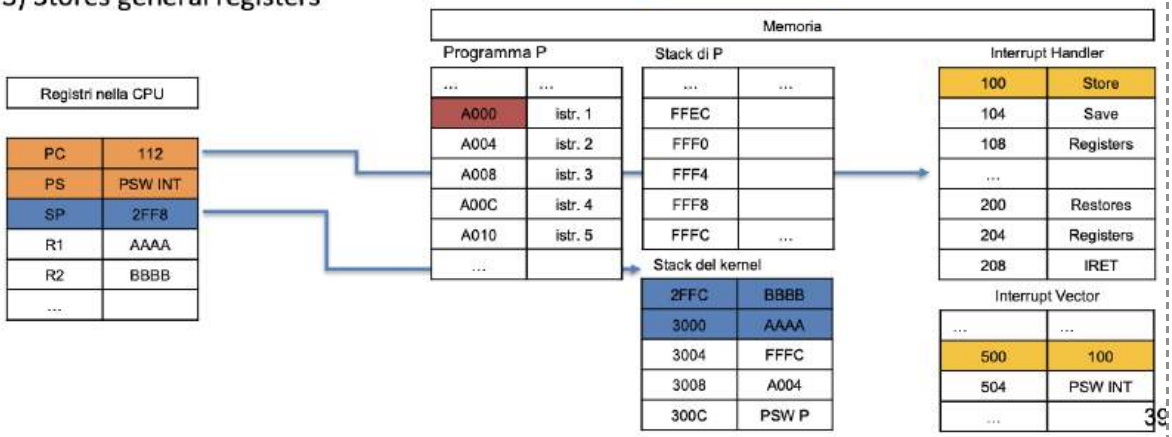
2) Interrupt recognized after instruction A000 (PC incremented to A004)



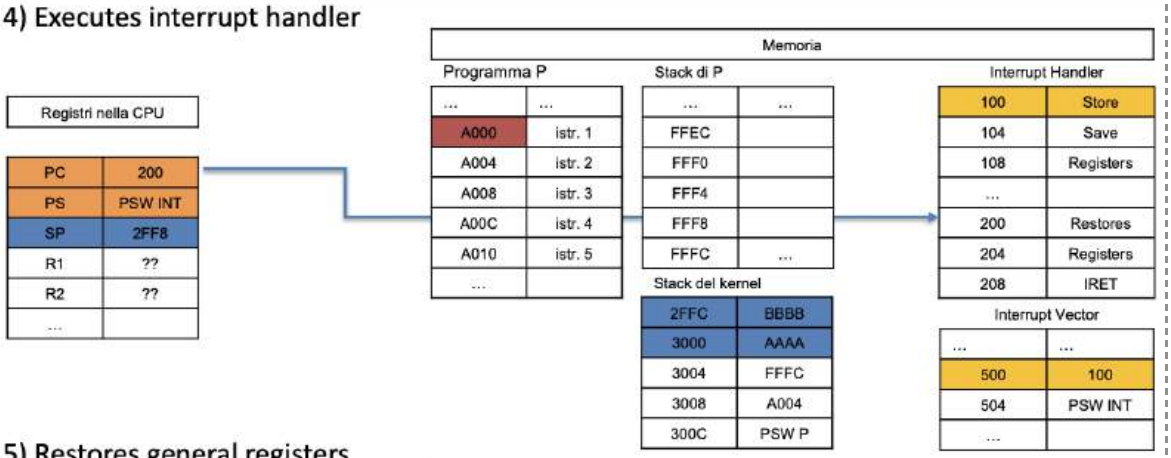
2) Interrupt recognized after instruction A000



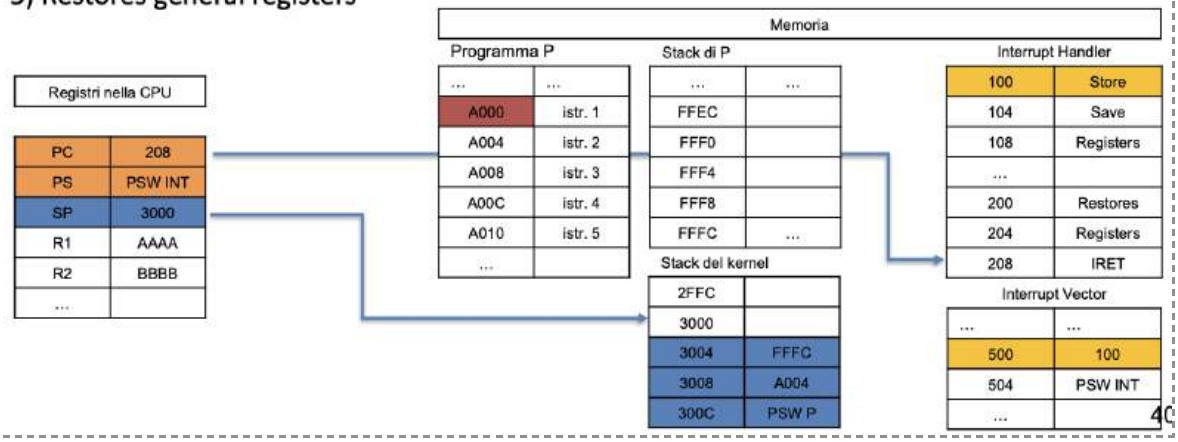
3) Stores general registers



4) Executes interrupt handler

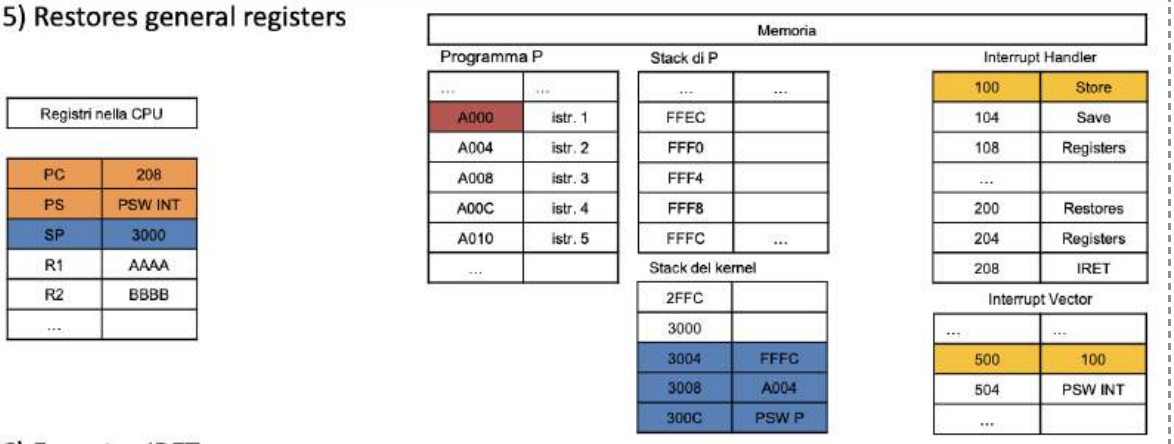


5) Restores general registers

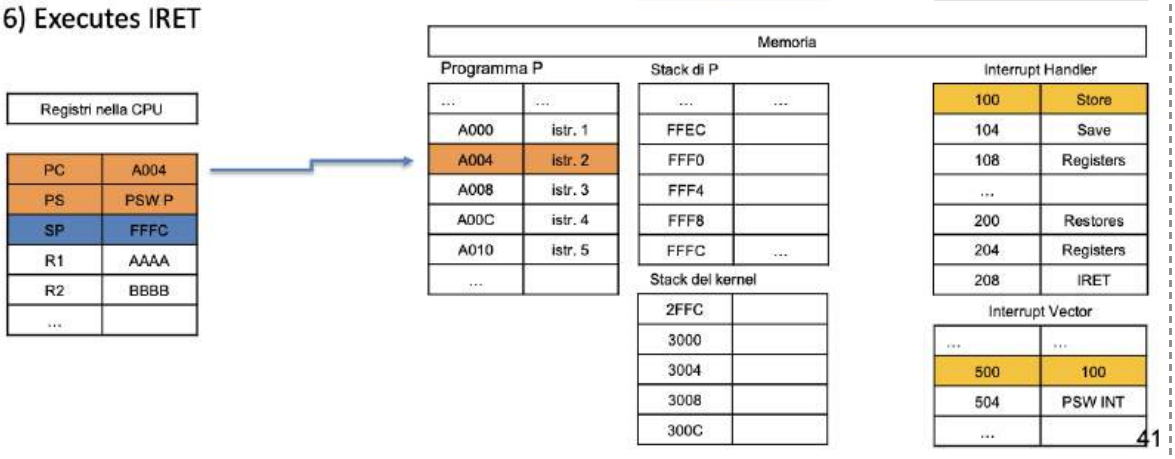


40

5) Restores general registers



6) Executes IRET



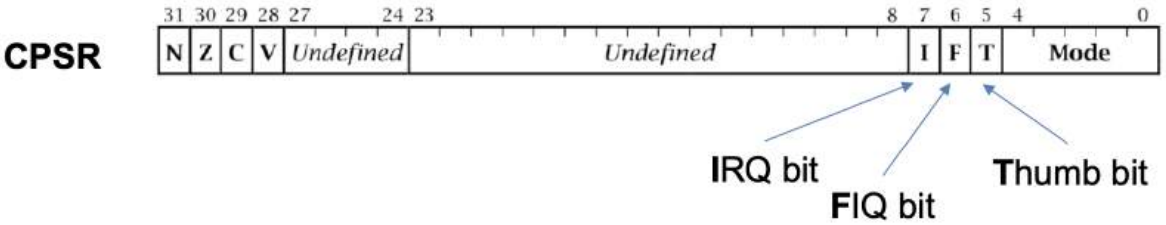
41

Inizialmente siamo nella modalità USER MODE mentre sta eseguendo un processo, all'istruzione A000 viene lanciata un'interruzione. Quindi si entra in modalità KERNEL MODE andando a modificare il PC (personal computer) che viene indirizzato dall' Interrupt Vector verso il codice da eseguire dell'interrupt vector nel nostro caso indirizzo = 100, e SP (stack pointer) che indirizza lo stack del kernel. Poi vengono salvati i registri generali del processo che era in esecuzione nello stack del kernel**.** Viene eseguito l'handler che al termine della gestione dell'interruzione ricarica tutti i registri generali e infine esegue l'istruzione di IRET (Interrupt return) che ripristina i registri speciali ovvero PC, SP, PSW (CPSR (current program status register)). Quindi possiamo tornare al contesto pre-interruzione.

Gestione delle modalità in ARM

Esistono varie modalità tra le quali switchare, tra le quali User, Supervisor, Interrupt, Fast Interrupt. Nell'immagine vediamo i corrispondenti bit del CPSR.

Mode	$CPSR_{4:0}$
User	10000
Supervisor	10011
Abort	10111
Undefined	11011
Interrupt (IRQ)	10010
Fast Interrupt (FIQ)	10001



Bit	Nome	Posizione in CPSR	Cosa fa
I	IRQ-mask	bit 7	Quando = 1 disabilita le **IRQ** (Interrupt Requests) normali; quando = 0 le abilita.
F	FIQ-mask	bit 6	Quando = 1 disabilita le **FIQ** (Fast Interrupt Requests); quando = 0 le abilita.

T	Thumb	bit 5	Quando = 1 la CPU decodifica ed esegue istruzioni in **Thumb/Thumb-2** mode; = 0 usa il set di istruzioni ARM a 32 bit.

IRQ vs FIQ

- **IRQ (Interrupt Request)**
  - Sono le interruzioni “normali” usate per la maggior parte dei dispositivi (timer, periferiche I/O).
  - Hanno priorità inferiore rispetto alle FIQ.
  - Gestite dal bit **I**: se I=1, tutte le IRQ vengono ignorate finché non viene resettato il bit.
- **FIQ (Fast Interrupt Request)**
  - Interruzioni “veloci” destinate a eventi time-critical (es. DMA, controller ad alta velocità).
  - Hanno priorità più alta delle IRQ; dispongono di un bank di registri R8–R12 e R14 “privati” per rispondere più rapidamente.
  - Gestite dal bit **F**: se F=1, tutte le FIQ sono ignorate finché non viene resettato il bit.
  - Disabilita le Interruzioni normali e “fast” mettendo I = 1 e F = 1 (**Interrupt Masking**).

Al posto dell'interrupt vector in Arm abbiamo la Exception vector table

Exception	Address	Mode
Reset	0x00	Supervisor
Undefined Instruction	0x04	Undefined
Supervisor Call	0x08	Supervisor
Prefetch Abort (instruction fetch error)	0x0C	Abort
Data Abort (data load or store error)	0x10	Abort
Reserved	0x14	N/A
Interrupt	0x18	IRQ
Fast Interrupt	0x1C	FIQ

Per ogni entry è salvata un'operazione di salto o di caricamento del PC, in questo modo la nostra tabella ci indirizzerà verso l'handler dell'interrupt.

Registri
Finora i registri che abbiamo usato erano i registri da R0 a R15 dove R13 era lo SP, R14 il LR R15 pc e poi la PSW.

Le altre 5 modalità hanno dei registri in più. Questi sono dei registri aggiuntivi visibili solo nella modalità specificata, ad esempio nella modalità fast interrupt ci sono 8 registri aggiuntivi. In realtà aggiuntivi non è il termine giusto in quanto i registri da R8-R14 non sono più visibili ma vengono abilitati dei corrispondenti registri con cui lavorare così da non dover salvare i dati. I registri aggiuntivi cambiano a seconda della modalità in cui siamo. In tutto abbiamo non 17, ma 37 registri.

Banked registers

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	CPSR	CPSR	CPSR	CPSR
		SPSR_fiq	**SPSR_irq**	**SPSR_svc**	**SPSR_abt**	**SPSR_und**

Quando effettuiamo lo switch da una modalità all'altra (es. USER MODE to FIQ MODE) salvo l'attuale CPSR dentro a SPSR e nel registro CPSR carico il CPSR per la modalità in cui sono.
Salva l'attuale **CPSR** dentro **SPSR_fiq**.
Carica un nuovo CPSR costruito per FIQ (disabilita ulteriori FIQ, setta il mode bits, ecc.).

Gestione delle eccezioni (INTERRUPT) in ARM
Quando dobbiamo gestire un'eccezione avviene un cambio di contesto nella modalità in cui dobbiamo fare lo Switch. A seconda della modalità avremo a disposizione diversi registri. Le operazioni che vengono eseguite dipendono dalla modalità in cui entro (Reset, SVC =

Supervisor, IRQ o FIQ).

Reset:

```
R14_rst  = PC
SPSR_rst = CPSR
CPSR[4:0] = 0x10011 // supervisor mode
CPSR[5] = 0 // ARM state
CPSR[6] = 1 // Disable FIQ
CPSR[7] = 1 // Disable IRQ
PC = 0x00000000
```

SVC:

```
R14_svc  = PC+4  // next instruction
SPSR_svc = CPSR
CPSR[4:0] = 0x10011 // supervisor mode
CPSR[5] = 0 // ARM state
// CPSR[6] unchanged
CPSR[7] = 1 // Disable IRQ
PC = 0x00000008
```

IRQ:

```
R14_irq  = PC+4
SPSR_irq = CPSR
CPSR[4:0] = 0x10010 // IRQ mode
CPSR[5] = 0 // ARM state
// CPSR[6] unchanged
CPSR[7] = 1 // Disable IRQ
PC = 0x00000018
```

FIQ:

```
R14_fiq  = PC+4
SPSR_fiq = CPSR
CPSR[4:0] = 0x10001 // FIQ mode
CPSR[5] = 0 // ARM state
CPSR[6] = 1 // Disable FIQ
CPSR[7] = 1 // Disable IRQ
PC = 0x0000001C
```


(le **IRQ** vengono sempre disattivate)

Ritornare da un'eccezione (interruzioni sono sottoinsiemi delle eccezioni)
Gli step da fare sono:

1. ricaricare il CPSR dal SPSR (istruzione atomica) cambio della modalità a user.
2. Riabilitare le interruzioni
3. Ricaricare il PC dal LR (istruzione atomica)

Se cambiassi prima il PC (ritorno al vecchio indirizzo) ma non ancora il CPSR, la CPU riprenderebbe a prelevare istruzioni usando ancora i bit di mode/flag/Thumb di eccezione.

Se cambiassi prima il CPSR (riporto dei flag e del mode) ma non ancora il PC, continueresti a eseguire l'ISR sull'indirizzo di eccezione, ma ormai con i flag e la modalità utente: potresti perdere la corretta atomicità di disable/enable degli interrupt o violare requisiti di sicurezza.

Per questo serve impostare il vecchio PC e il vecchio CPSR in un'**unica istruzione atomica**.
ARM fa del ritorno da eccezione un'unica operazione atomica, **con S in fondo**, per modificare le flag del **CPSR**. In particolare vengono utilizzate le istruzioni MOVS, SUBS oppure la IRET (**Instruction RETurn**)

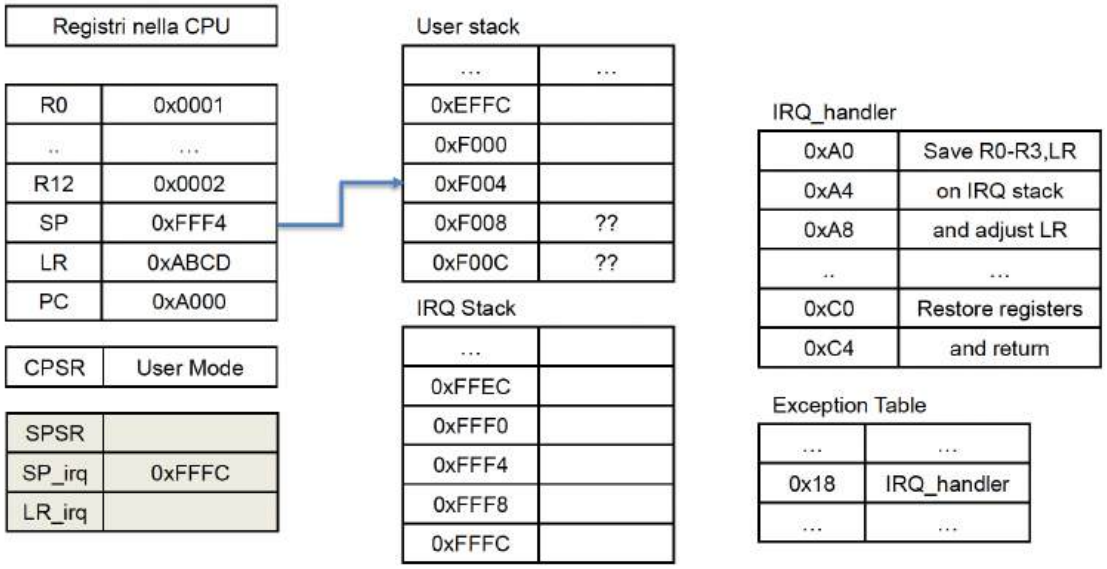
In particolare nell'immagine qui sotto possiamo trovare i comandi utilizzati da ogni modalità di ARM.

	Return instruction	ARM LR_x	THUMB LR_x
SWI	MOVS PC, LR_svc	PC+4	PC+2
UNDEF	MOVS PC, LR_und	PC+4	PC+2
FIQ	SUBS PC, LR_fig, #4	PC+4	PC+4
IRQ	SUBS PC, LR_irq, #4	PC+4	PC+4
PrefetchABT	SUBS PC, LR_abt, #4	PC+4	PC+4
DataABT	SUBS PC, LR_abt, #8	PC+8	PC+8
RESET	-	?	?

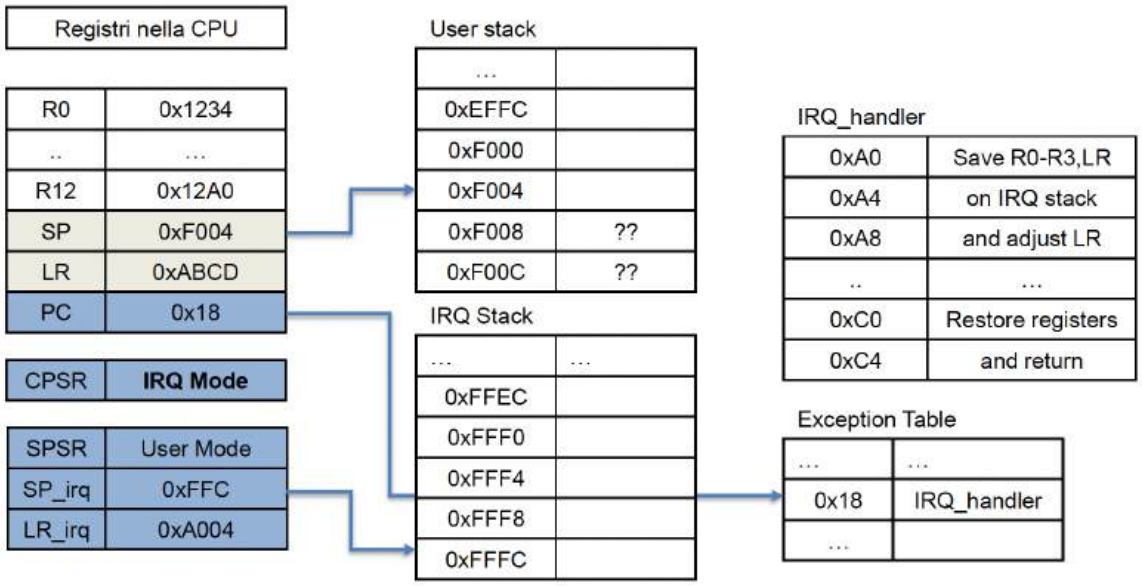
Esempio Gestione Eccezione ARM

Interrupt management example

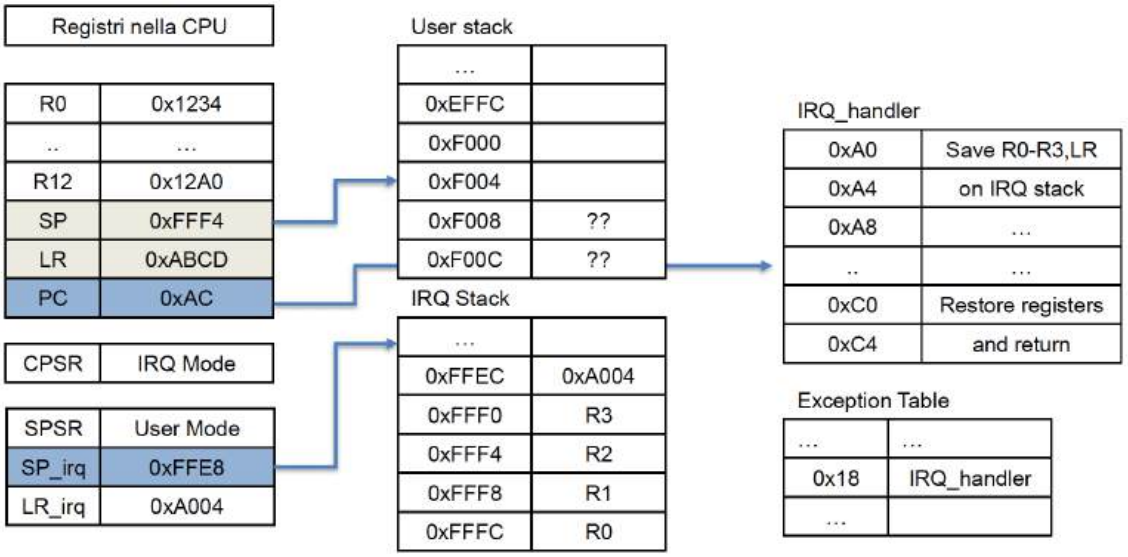
Step1: Initial state, IRQ occurs when executing instruction 0xA000



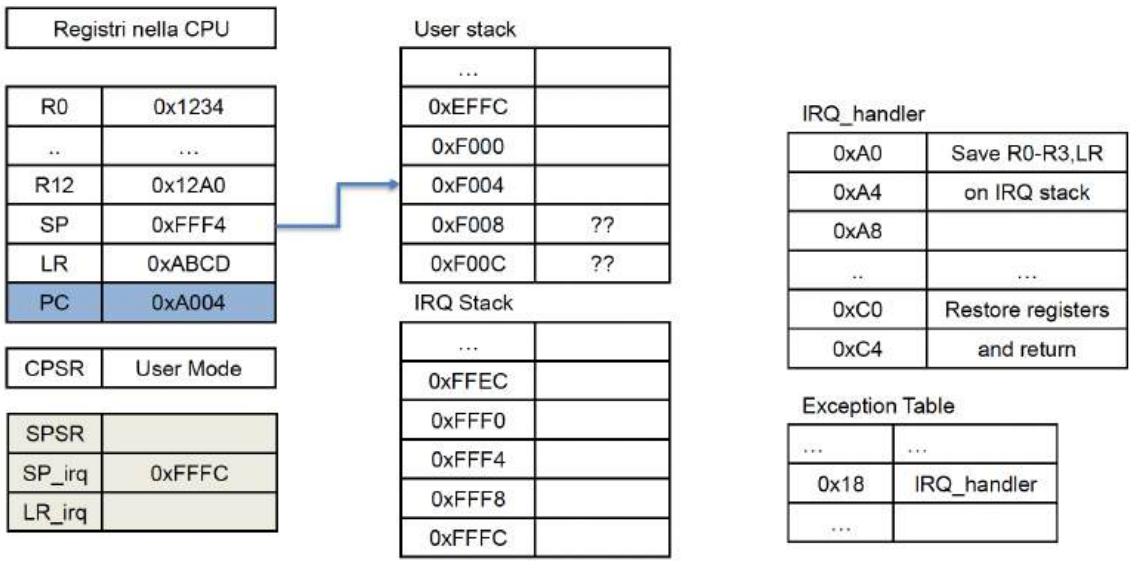
Step2: Switching to IRQ Mode, map to banked SP and LR and be prepared to execute the IRQ handler



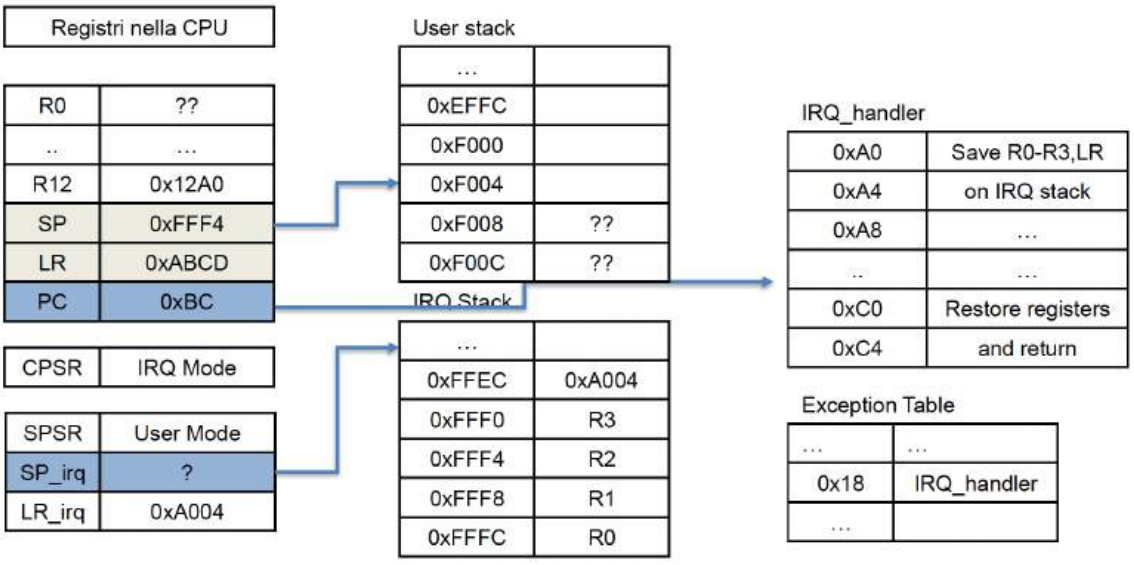
Step3: Start executing top-level handler for IRQ



Step4: Restore user registers and switch to User Mode, LR_irq (on the stack) will be copied into the PC register



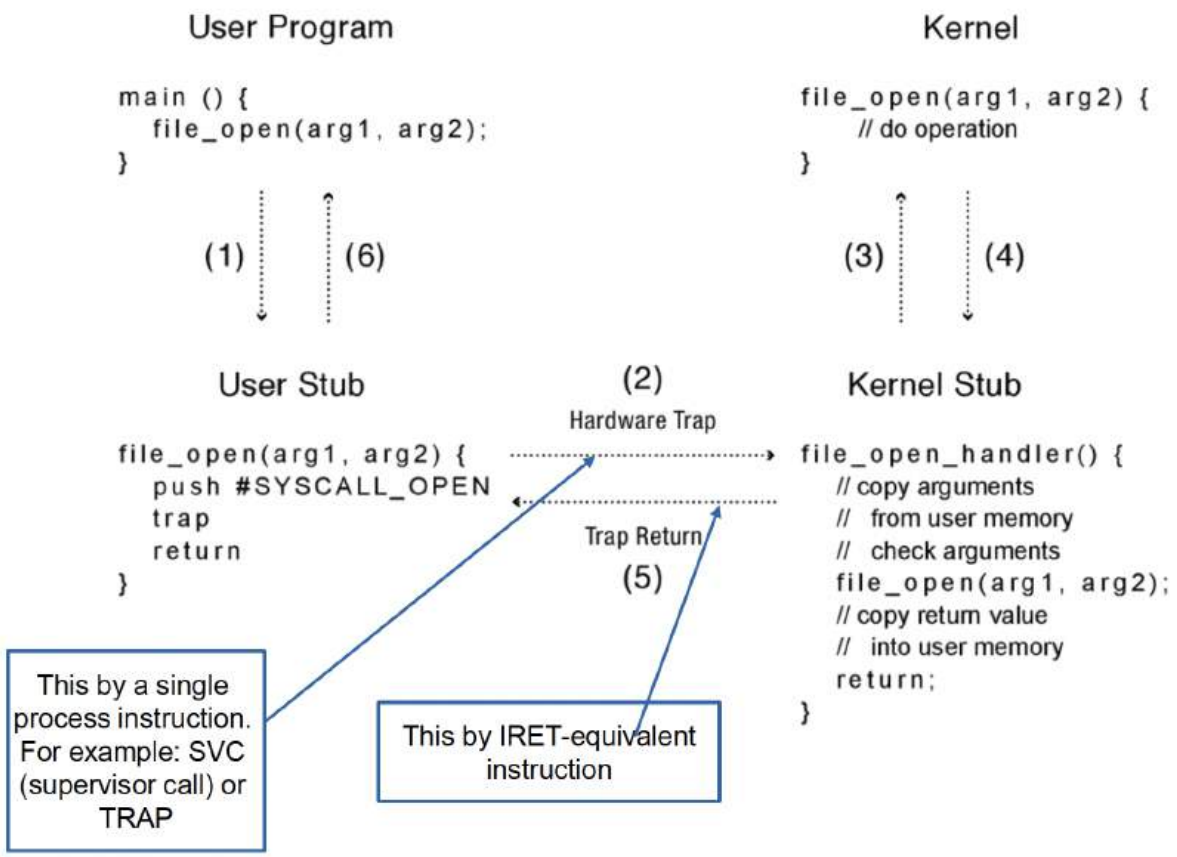
Step4: IRQ handled, now we have to restore the state before the IRQ



System call

Le System Call sono chiamate a funzioni che deve eseguire il kernel in uno spazio privilegiato (kernel mode). Queste sono un'interfaccia per eseguire funzioni del SO. (esempi: input da tastiera, scrivere/leggere un file ecc.). Una system call chiama una **Software Interrupts** (**SWIs**) e sospende il chiamante. Ritornano tipicamente un valore intero che indica se la System Call è andata a buon fine.

- < 0 => **errore**
- >=0 => **tutto ok!**



Descrizione immagine
Il codice chiama una System Call, che viene gestita da uno **user stub** che tramite l'istruzione di **trap** o **SVC** - **Supervisor Call** - chiama una Software Interrupt per passare alla kernel mode, effettua anche un primo controllo sui dati.
A questo punto vengono passati gli argomenti ad un intermediario del kernel ovvero il **kernel stub** che una volta identificata la System Call, effettua una **copia degli argomenti** nel Kernel-Stack, **controlla** gli argomenti e chiama la vera e propria funzione del sistema operativo. Infine copia il risultato nello user stack. Infine esegue la **Trap Return**, ovvero l'equivalente della IRET.

Chiamate per passare da USER MODE to KERNEL MODE
TRAP (=SVC in ArmV7)
Cosa succede quando si attiva una **trap**:

1) La CPU cambia da User Mode a Kernel Mode.
2) Salva lo stato della CPU nel **SPSR** (dove viene salvato il vecchio CPSR).
3) Usa la **Exception Vector Table** (EVT) per trovare l'indirizzo della syscall.

SVC
Una **SVC** è un'**istruzione speciale** eseguita da un programma utente per richiedere un servizio al kernel. è un'istruzione di TRAP di ArmV7.

SWI (Software Interrupts) handler

```
/* simplified schema of the top-level handler for handling SVCs */
SWI_Handler                     ; top-level handler
    STMFD   SP!, {R0-R12, LR}   ; pushes registers into the stack
    ADR R8, SYS_CALL_TABLE      ; load syscall table pointer in R8
    ; R7 contains the SC number
    ADD R7, R7, #_SYSCALL_BASE  ; OS entry of the sys_* routine
    ; sanity checks
    ;
    LDR PC [R8, R7, LSL #2]     ; call sys_* routing
    ; ...
    ; result of the SVC stored in R0
    ; ...
    ; restore user registers
    ;
    MOVS PC, LR                 ; return from handler restoring CPSR
```

Upcall a.k.a. UNIX signals
L'analogo delle System Call solo da parte del kernel verso la user-mode. Sono delle chiamate che possono essere effettuate da parte del kernel verso lo user. Questo può essere utile per semplificare le operazioni di **polling** notificando dal kernel quando i dati sono pronti.

Le **upcall** avvengono nelle seguenti **fasi**:

1. Il kernel **rileva un evento** (es. nuovi dati di rete).
2. Il kernel **attiva un'upcall** per notificare il processo utente.
3. Il processo utente **riceve la notifica e reagisce all'evento, salvando lo stato architetturale nel momento in cui si verifica l'evento nello *stack utente*.**
4. Viene eseguita in **User mode**.

Caratteristica	System Call	Upcall
Direzione	user → kernel	kernel → user
Iniziativa	presa dal processo utente	presa dal kernel (evento)
Blocco del chiamante	sì (sincrona)	no (asincrona)
Istruzione trap	`syscall`/`SVC`	trapframe + `iret` al handler
Ritorno al contesto	`iret` al punto dopo la syscall	`upcall_return()` al punto di crash originario
Uso tipico	I/O, gestione file, memoria, processi	page-fault user-level, notifiche I/O asincrono, segnali

Kernel booting

Il kernel è un software complesso e grande. Sta sul disco e dobbiamo caricarlo nella RAM, come
possiamo fare?

ROM => BIOS => BOOTLOADER => KERNEL => USER MODE

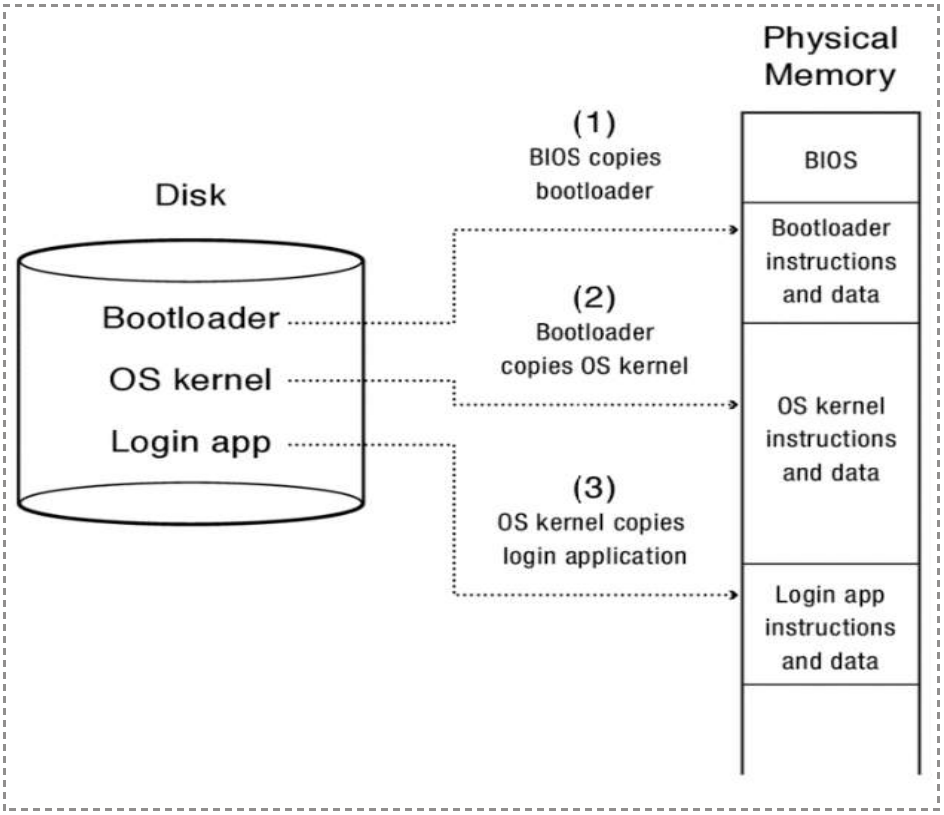
La **ROM** (Read-Only Memory) è una memoria **lenta** e **non modificabile**, che contiene il **bootloader** (o firmware come BIOS/UEFI).

1. Accensione – Esecuzione del firmware (BIOS/UEFI): Risiede In **ROM**. Inizializza l'hardware essenziale (RAM, tastiera, video, CPU, ecc.). Cerca un **dispositivo di boot** (es. hard disk, SSD, USB) da cui caricare il bootloader e carica il **bootloader** dalla **prima parte del disco** nella **RAM**. Trasferisce il controllo alla CPU.

2. Bootloader: Risiede su **disco**, in genere nel Master Boot Record (MBR) o nella partizione EFI. Mostra eventualmente un menu di scelta tra sistemi operativi. Carica in **RAM** l'immagine del **kernel**. Inizializza alcune strutture di memoria (come la **Interrupt Vector Table**). Passa il controllo al **kernel**.

3. Kernel: Inizializza dispositivi e driver. Monta il file system root (`/`). Crea il primo processo utente. Effettua il **passaggio da kernel mode a user mode**, (ad esempio con un'**istruzione come `IRET`**, dipende dall'architettura).

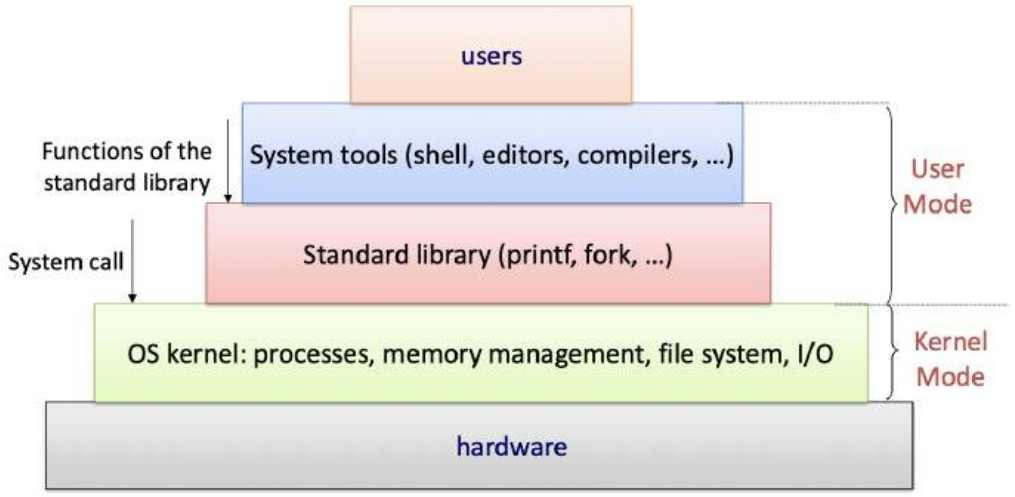
4. User space: Viene avviato il primo processo utente (per esempio il processo di **login** o la **shell**).


Physical Memory
(1)
BIOS copies bootloader
BIOS
Disk
Bootloader instructions and data
Bootloader
OS kernel
Login app
(2)
Bootloader copies OS kernel
OS kernel instructions and data
(3)
OS kernel copies login application
Login app instructions and data

Processi

Processi

Os interface
Il sistema operativo funziona come interfaccia per le applicazioni che comunicano verso il sistema hardware attraverso delle chiamate.
Unix interface



Shell

La shell permette all'utente di lanciare e gestire processi attraverso un'interfaccia grafica su linea di comando. Questo permette di avere una via di comunicazione diretta col SO. La shell è implementata da tutti i moderni SO (Mac OS, Windows, Linux).
esempio: compiliamo un programma in C

$> cc –c sourcefile1.c
$> cc –c sourcefile2.c
$> ln –o program sourcefile1.o sourcefile2.o

La shell utilizza fork, execve, waitpid, dup2, signal, etc… per eseguire un programma.

```
char *prog, **args;
int child_pid;

// Read and parse the input one line at a time
while (readAndParseCmdLine(&prog, &args)) {
  child_pid = fork();     // create a child process
   if (child_pid == 0) {
     exec(prog, args);      // I'm the child process.  Run the program
    // NOT REACHED
   } else {
     wait(child_pid);      // I'm the parent, wait for child
     // checking exit status
   }
}
```

See the example code provided for a working (simplified) implementation.

Windows create process

È una **chiamata di sistema** che permette di creare direttamente un nuovo processo che esegue un programma. E' però complicata da utilizzare, con 10 parametri in input.

1. Crea e inizializza un PCB (process control block) nel kernel
2. Crea e inizializza un nuovo spazio di indirizzi
3. Carica il programma nello spazio degli indirizzi
4. Copia gli argomenti in memoria nello spazio degli indirizzi
5. Inizializza il contesto hardware per avviare l'esecuzione a "start"
6. Informa lo scheduler che il nuovo processo è pronto per l'esecuzione

Unix process management

• ***Unix fork*** – chiamata di sistema per creare una copia del processo corrente e avviarlo in esecuzione – Nessun argomento!

• ***Unix exec*** – chiamata di sistema per cambiare il programma in esecuzione dal processo corrente

• ***Unix wait*** – chiamata di sistema per attendere il completamento di un processo

• ***Unix signal*** – chiamata di sistema per inviare notifiche tra i processi

Unix fork

```
pid_t fork(void);
```

- Non prende parametri in input

- Ritorna un intero **= 0** se è processo **figlio**; **> 0** se è processo **padre**, dove il valore è il **PID figlio**. Se ritorna un intero **<0**, errore: può avvenire nel caso **non ci sia spazio nella tabella dei processi**.
- La fork genera un processo figlio che condivide lo stesso codice del padre.
- **Getpid()** e **getppid()** restituiscono rispettivamente l'ID del processo del chiamante e del processo del genitore
- Ogni processo ha il suo PID (>0)
- Un processo padre contiene il PID del figlio, un processo figlio contiene 0.

Implementazione di una UNIX fork

1. Crea e inizializza il Process Control Block (PCB) nel kernel
2. Crea un nuovo spazio di indirizzi, dove copia (eredità) l'intero contesto di esecuzione del padre.
3. Informa lo scheduler del nuovo processo pronto a runnare (il processo si trova nello stato di pronto).

Unix exec

```
int execvp(const char *file, char *const argv[]);
```

- Rimpiazza il codice eseguito da un processo. Non crea un nuovo processo ma **eredita** il PCB
- Inizializza l'HW context (setta i registri) per eseguire la prima istruzione
- Rimpiazza i dati e copia gli argomenti nello spazio degli indirizzi
- Se fallisce ritorna un errore (es: percorso non valido, problemi di autorizzazione...), altrimenti non ritorna nulla.
- Dopo exec si mantiene tutto uguale (PID, PCB, reset pending signals, kernel stack e risorse assegnate) viene cambiato il riferimento al codice
- Nel caso vada a buon fine viene caricato al posto del vecchio codice quello nuovo. Nello spazio degli indirizzi dedicato al codice da eseguire del processo.
- L'unico caso in cui viene eseguito il codice originario dopo la exec è quando la exec non viene eseguita correttamente tornando errore

Dopo una *fork()* seguita da una *exec()* il processo padre e il processo figlio **condividono l'intera tabella dei file aperti.**

Esempio

Esercizio 1
In seguito alla chiamata di sistema *fork*, quali informazioni del processo figlio **sono identiche** a quelle del processo padre tra quelle indicate in tabella ?

Dato	**SI/NO**
Stack	NO
Codice	SI
Heap	SI
Handler dei segnali (signal handler array)	SI
Informazioni sui segnali mascherati (signal mask)	SI
Informazioni sui segnali pendenti (pending signal bitmap)	NO
Program counter	SI
Stack pointer	SI
Registri generali e PSW	SI
PID del processo	NO
PID del padre del processo	NO

Esercizio 2
In seguito alla chiamata di sistema *exec*, quali informazioni del processo **vengono modificate** se la exec ha successo?

Dato	**SI/NO**
Stack	SI
Codice	SI
Heap	SI
Handler dei segnali (signal handler array)	SI
Informazioni sui segnali mascherati (signal mask)	NO
Informazioni sui segnali pendenti (pending signal bitmap)	NO
Program counter	SI
Stack pointer	SI
Registri generali e PSW	SI
PID del processo	NO
PID del padre del processo	NO

Terminazione di un processo

Un processo può terminare a causa di una **eccezione** o invocando la **exit()**. Il processo terminato ritorna un valore di uscita o il padre. Il padre riceve un valore dalla **wait()**, ovvero il valore di ritorno della exit del figlio. Quando un processo termina *correttamente* viene eliminato dal SO tutto lo spazio allocato per il processo e viene cancellato dalla **process table** (tabella nel Kernel contenente tutti i processi).

Un **processo zombie** è un processo terminato, il quale mantiene il suo **PCB** nella **process table** il SO non può eliminarlo perché il padre non ha ancora chiamato `wait()` per raccoglierne lo stato di uscita.
Soluzioni per eliminare un processo zombie:

1. **Chiamare `wait()` nel padre**: il metodo più semplice per raccogliere l'uscita del figlio.
2. **Usare `SIGCHLD` e `waitpid()`**: il padre può intercettare il segnale `SIGCHLD` e rimuovere automaticamente i figli terminati.
3. **Terminare il processo padre**: se il padre muore, il sistema assegna il processo zombie a `init` (o `systemd`), che lo rimuove automaticamente.

Unix exit

```
void exit (int status);
```

- Status è il codice di terminazione
- Exit non ritorna niente
- Libera la memoria e rilascia le risorse
- Se switcha allo stato di **zombie**, mantiene il PCB (non lo elimina)

Unix wait

```
Int wait (int *status)
```

- Status è l'indirizzo in memoria in cui viene scritto il PID del processo terminato o del codice di errore, serve anche per risolvere lo stato di zombie di un processo figlio. Permette di sincronizzare padre e figlio.
- Può ritornare immediatamente nel caso il processo figlio sia già terminato.

Unix pipe

```
int pipe(int pipefd[2]);
```

a. Scopo

- **pipe**: Crea un canale di comunicazione **unidirezionale** tra processi, consentendo la trasmissione di dati da un processo (scrittore) a un altro (lettore).

b. Cosa accade internamente

1. **Allocazione della Struttura del Pipe:**
   Il kernel alloca una struttura dati interna che fungerà da buffer circolare. Questa struttura include:
   - Un'area di memoria dedicata per il buffering.
   - Variabili per tenere traccia della quantità di dati presenti, degli indici di lettura e scrittura, e dello stato del pipe (ad esempio, se il lato di scrittura è ancora aperto).
2. **Creazione di Due File Descriptor:**
   La chiamata a pipe restituisce due file descriptor:
   - **fd[0]:** Per la lettura.
   - **fd[1]:** Per la scrittura.
     Questi descrittori puntano alla stessa struttura del kernel, consentendo la comunicazione tra processi.
3. **Gestione della Concorrenza:**
   Il kernel gestisce in modo sincronizzato l'accesso al buffer:
   - Quando un processo scrive, se il buffer è pieno, il processo può essere bloccato fino a quando il lato di lettura non libera spazio.
   - Allo stesso modo, se il processo di lettura tenta di leggere da un buffer vuoto, può essere messo in attesa fino a quando nuovi dati non vengono scritti.
4. **Chiusura e Cleanup:**
   Quando tutti i descrittori di lettura o scrittura sono chiusi, il kernel libera la

struttura del pipe e notifica eventuali processi in attesa (ad esempio, *un tentativo di scrivere su un pipe senza lettore restituisce un errore*).

Se io leggo e lo **scrittore** ha **chiuso** => **aspetta la terminazione del lettore**
Se io scrivo e il **lettore interrompe** => **errore**, termina

Unix read

```
ssize_t read(int fd, void *buf, size_t count);
```

read() è la system call Unix per **leggere dati** da un **file descriptor** (file, pipe, socket, ecc.) in un buffer utente.

Parametri
fd = Il file descriptor aperto in lettura (es. risultato di open(), pipe(), socket()).
buf = Puntatore al buffer in memoria utente dove copierà i byte letti.
count = Numero massimo di byte da leggere.

Comportamento
Se ci sono dati pronti (file non-blocking o pipe con writer), copia fino a *count* byte nel buffer.
Se non ci sono dati il chiamante viene sospeso finché arrivano dati o EOF.

Valore di ritorno
- ≥ 1 : numero di byte effettivamente letti.
- 0 : **EOF** (nessun altro dato disponibile).
- -1 : errore (se *errno* è settato, es. EINTR, EAGAIN, EBADF).

Unix write

```
ssize_t write(int fd, const void *buf, size_t count);
```

Parametri
fd = file descriptor aperto in modalità scrittura.
buf = puntatore del buffer in memoria che contiene i byte da scrivere.
count = numero massimo di byte da scrivere dal buffer.

Valore di ritorno

≥ 0 : numero di byte effettivamente scritti (può essere < `count`).

`-1`: errore, e `errno` è settato a uno dei codici seguenti.

Unix open

```
int open(const char *pathname, int flags, mode_t mode);
```

Parametri
pathname: percorso del file da aprire o creare.

flags (bit-wise OR):
- access mode: O_RDONLY, O_WRONLY, O_RDWR
- creation: O_CREAT (usa mode), O_EXCL (con O_CREAT, error se esiste), O_TRUNC (tronca se esiste)
- other: O_APPEND, O_NONBLOCK, O_CLOEXEC, …

mode: PERMESSI UNIX (es. 0644) usati solo se O_CREAT è specificato; altrimenti ignorato.

Valore di ritorno
≥ 0: nuovo file descriptor.
–1: errore, con errno settato

UNIX File System Interface

• **UNIX open è un Coltellino Svizzero:**
Attraverso i flags posso personalizzare il comportamento della chiamata OPEN.
– Open the file, return file descriptor
– Options:
• if file doesn't exist, return an error
• If file doesn't exist, create file and open it
• If file does exist, return an error
• If file does exist, open file
• If file exists but isn't empty, nix it then open
• If file exists but isn't empty, return an error

Interface design question

```
if (!exists(name))
create(name); // can create fail?
fd = open(name); // does the file exist?
```

La open come scelta di design ha quella di avere al suo interno altre chiamate di sistema. Questo logicamente può sembrare una scelta azzardata e si potrebbe pensare di renderla indipendente da altre chiamate di sistema per renderla più modulare. Invece si preferisce questo approccio per **evitare race condition** e **rendere l'operazione atomica** all'interno del kernel.
Se fai exists() prima di fare la open allora si crea una "finestra" tra la *condizione dell'if* e *create* in cui un altro processo può creare/rimuovere il file.

Unix close

```
int close(int fd);
```

fd: file descriptor da chiudere (ottenuto con open, pipe, dup, …).

Valore di ritorno
0: successo.

–1: errore, con errno.

Unix dup2

```
int dup2(int oldfd, int newfd);
```

Parametri
oldfd: file descriptor esistente da duplicare.
newfd: target descriptor. Se è aperto, viene chiuso prima di riutilizzarlo; se uguale a oldfd, non fa nulla e ritorna newfd.

Valore di ritorno
≥ 0: il valore di newfd.
–1: errore, con errno:

Esempio di utilizzo *fork, wait, exec*

```
char *prog, **args;
int child_pid;

// Read and parse the input one line at a time
while (readAndParseCmdLine(&prog, &args)) {
  child_pid = fork();     // create a child process
   if (child_pid == 0) {
     exec(prog, args);      // I'm the child process.  Run the program
    // NOT REACHED
   } else {
     wait(child_pid);      // I'm the parent, wait for child
     // checking exit status
   }
}
```

See the example code provided for a working (simplified) implementation.

Unix I/O

• **Uniformità** – il modo in cui un programma interagisce con un **file su disco**, un **terminale**, una **connessione di rete** (socket), una **pipe** o qualsiasi altro tipo di **"file"** (nel senso lato di Unix) è fondamentalmente lo stesso. Si utilizzano le stesse **system call** (viste sopra).

• **Apri prima dell'uso** – Prima di poter leggere o scrivere su un file o interagire con un dispositivo, un processo deve prima "aprirlo" utilizzando la system call `open()`. Questa chiamata al kernel esegue diverse operazioni, come verificare i permessi di accesso, allocare risorse e creare una rappresentazione interna del file/dispositivo per il processo.

• **Orientato ai byte:** l'I/O in Unix è trattato come un flusso sequenziale di byte. Non c'è una nozione intrinseca di "record" o "blocchi" a livello delle system call `read()` e `write()`.

L'interpretazione di questi byte (come caratteri, numeri interi, strutture dati, ecc.) è **responsabilità del programma utente**.

• **Lettura/scrittura** con **buffer** nella memoria del kernel.

• **Chiusura esplicita** – a differenza di altri sistemi o linguaggi di programmazione con garbage collection automatica per le risorse di I/O, in Unix è **responsabilità del programmatore** chiudere esplicitamente i file descriptor aperti utilizzando la system call `close()`.

Concorrenza

Concorrenza

La concorrenza serve a gestire più cose alla volta, possiamo mandare più processi in background o thread per eseguire più operazioni sfruttando il parallelismo, **MTAO** multiple things at once.

Processi

Nei processi non esiste la memoria condivisa, i processi cooperano comunicando attraverso **inter-process communication** (IPC) ovvero meccanismi offerti dal sistema operativo. Quindi la comunicazione passa attraverso il kernel. I processi comunicano effettuando chiamate di sistema e questo è molto dispendioso in termini di prestazioni introduciamo quindi i thread.

Thread abstraction

Un thread è una singola esecuzione di codice processato in maniera sequenziale che il SO può sospendere o interrompere in qualsiasi momento* (se cooperative-multithreading NO). I thread condividono memoria e strutture dati tra thread appartenenti allo stesso processo.
Quando si programma usando i thread il sistema dà l'impressione al programmatore di avere N processori per eseguire N thread, ma questa non è la realtà. Infatti l'esecuzione dei thread viene interrotta e ripresa di continuo alternando i vari thread (**interleaving)**.
PRO:

- L'uso dei thread migliora la struttura del codice
- Incrementa le performance sfruttando architetture multiprocessore
- Ogni processo ha i suoi thread e la relativa **thread table** contenente tutti i **TCB** (Thread Control Block).

Kernel mode Multi-process

- Processi multipli a **thread singolo**, con **chiamate di sistema** utilizzate per accedere alle strutture dati condivise dal kernel in grado di usare le istruzioni privilegiate.

Kernel mode multi-thread

- Thread multipli, condivisione di strutture di dati del kernel dato che i thread hanno memoria condivisa e in grado di utilizzare istruzioni privilegiate

User mode a thread singolo

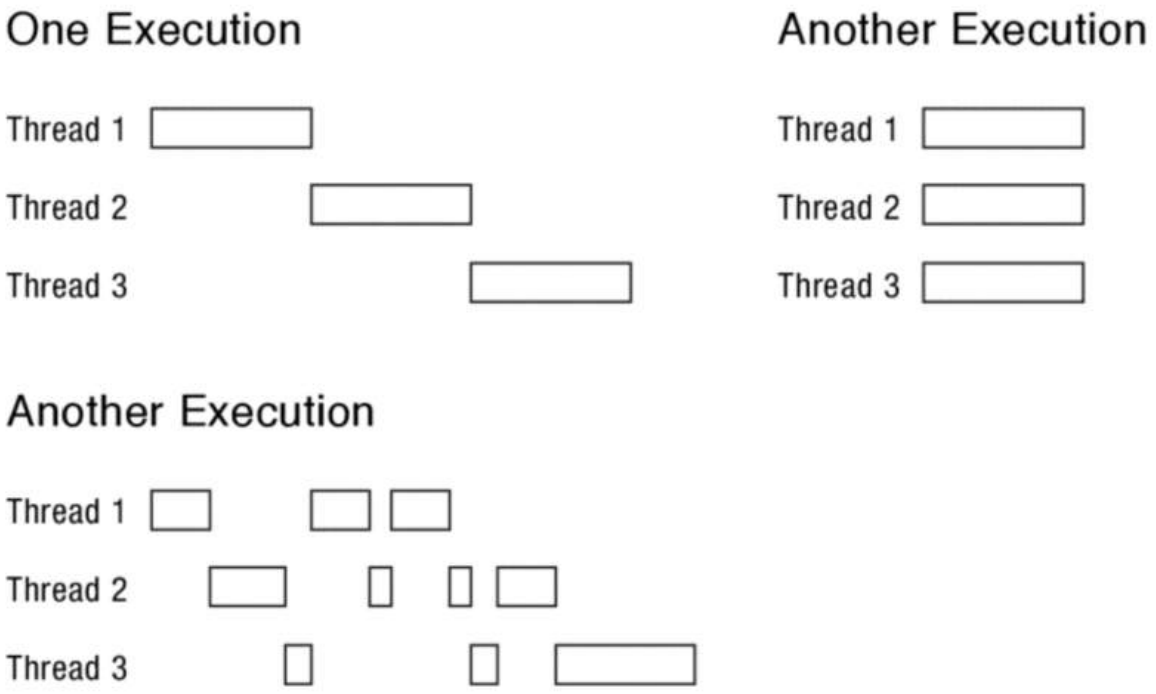
- Un thread all'interno del processo utente (processo=thread).

User mode multi-thread

- Thread multipli all'interno del processo utente, condividono strutture dati e sono isolati da altri processi utente

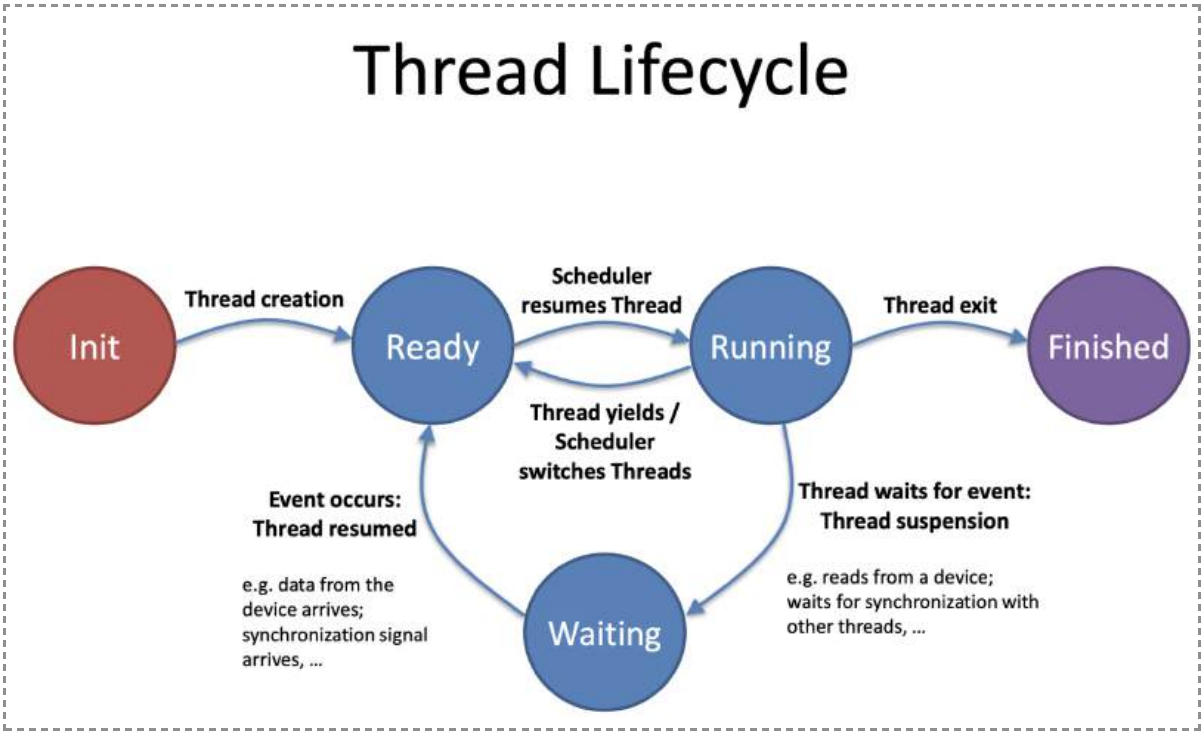
Possibili esecuzioni di un programma multi-thread, non conosciamo a priori l’esecuzione dei thread questo porta alla **race condition** ovvero non l’output dipende dall’ordine di esecuzione dei thread.

Esempio delle possibili esecuzioni



Implementazione dei thread

1. Thread Control Block - **TCB** - struttura dati che memorizza informazioni sul thread (Thread ID, stato, contesto…)
2. Un insieme di **operazioni** sui thread (creazione, terminazione, blocco…)
3. Uno scheduler in grado di pianificare thread
4. Una funzione del sistema operativo che assegna i processori ai thread



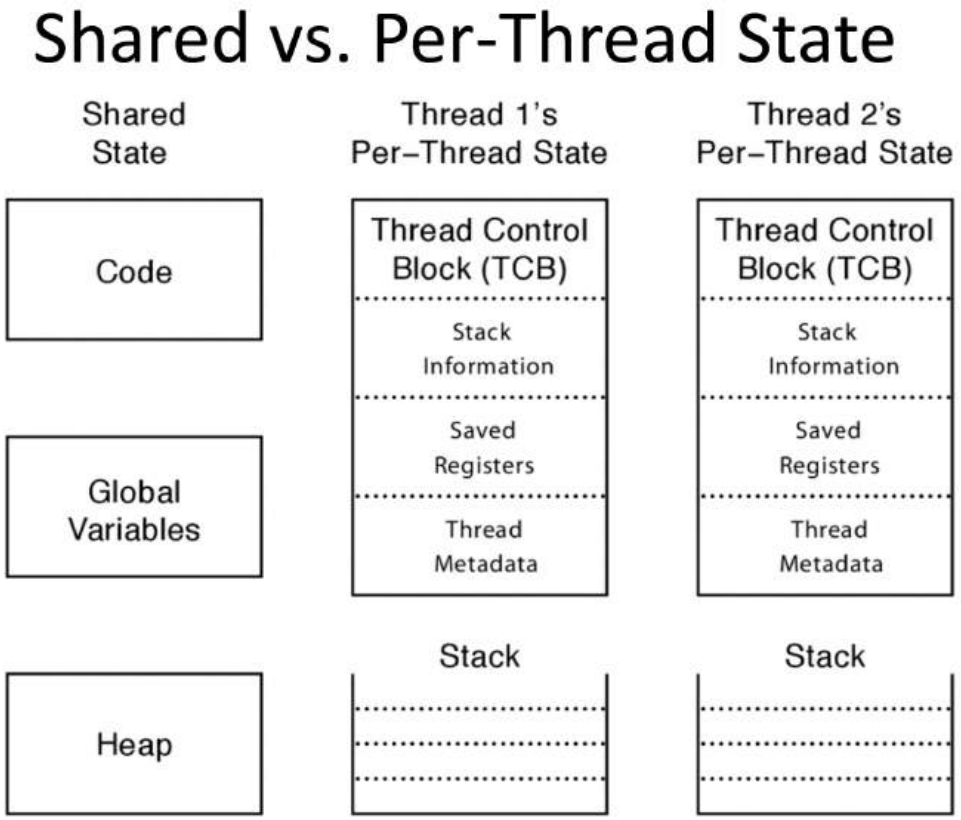
State of thread	Location of TCB	Location of registers
INIT	Being created	TCB
READY	Ready List	TCB
RUNNING	Running List	Processor
WAITING	Synchronization Variable's Waiting List	TCB
FINISHED	Finished List, then Deleted	TCB

Rappresentazione per-thread

Ogni thread è rappresentato da una struttura dati chiamata **Thread Control Block** (TCB) composta da:

1. **Stato di computazione del thread**:
    a. Stack
    b. Copia dei registri del processore
2. **Metadata**:
    a. Info sul thread (priorità, stato di esecuzione, ecc.)

Ogni thread ha un ***per-thread state*** che appartiene al singolo thread e uno **stato condiviso** che comprende il *codice*, le *variabili globali* e lo *heap*.



Differenza tra il **PCB** (Process Control Block) e il **TCB** (Thread Control Block)

- **PCB**:
  - Process name (PID)
  - Assigned memory
  - Other resources
    - Devices, open files, …
  - List of theads
    - References to TCBs
  - Accounting info
  - Upcall handlers
  - …
- **TCB**:
  - Thread ID
  - State
  - Context of the thread
  - Scheduling parameters
    - Priority, affinity, …
  - Reference to the stack(s)
  - Thread local storage
  - …

Cooperative vs Pre-emptive multi-threading

Cooperative multi-threading

Un thread si può bloccare solo se esegue un comando di **interruzione** esplicita come lo **yield** o attraverso la **wait**. **Non implementa lo scheduler** e può prendere possesso del processore senza essere controllato.

Pros:

- Maggiore controllo nell'intervallare i thread.

Cons:

- A lungo andare si potrebbe monopolizzare il processore.
- Questo tipo di approccio non è più utilizzato nei SO di oggi.

Pre-emptive multi-threading

Il SO controlla e gestisce il cambio di thread/processo come vuole grazie a un **timer**.

Pros:

- Migliore approccio per i moderni sistemi a multicore e per il mantenimento responsivo
- Largamente usato nei moderno SO

Cons:

- Maggior costo di context switch.

Implementazione User-level threads e Kernel threads

User-level threads

- Implementazione dei thread tramite **libreria**
- **Scheduling** dei thread implementati dal Run Time Support (**RTS**)
- La **thread table** risiede in user space all'interno del **processo**, significa che il SO vede il singolo processo e non sa dei thread che risiedono all'interno.
- Le operazioni di **create**, **yield**, **wait**, **exit** vengono effettuate dalla libreria e quindi **non** sono chiamate di sistema.

Pro	**Contro**
Context switch **veloce**	non sfruttano realmente più processori
Portabile anche su macchine che non sono multiprocessore	Una **system call** di un thread **blocca** tutti gli altri thread sotto al solito processo.

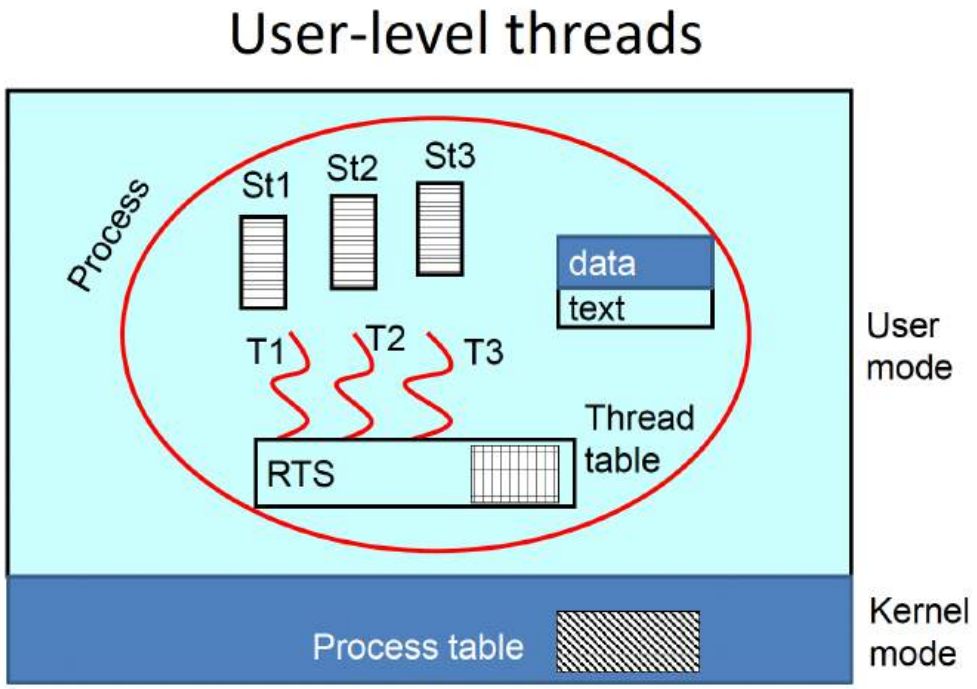
Kernel-level threads

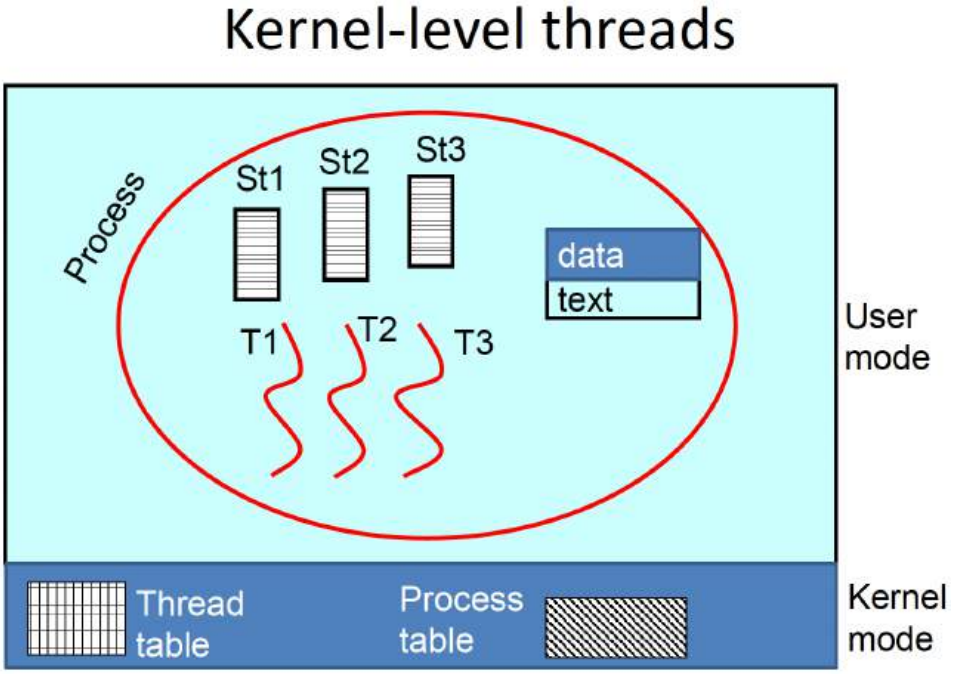
- La gestione dei thread è **responsabilità del SO**
- **Scheduling** effettuato dal **Kernel**.

- La **thread table** risiede nel **kernel,** che ha tutti i thread in Kernel space.
- Le operazioni di **create, yield, wait, exit** vengono effettuate tramite **System Call**.

Pro	**Contro**
Sfrutta le architetture multiprocessore	Context Switch lento
Una system call non blocca tutto il sistema	Deve essere supportato fisicamente dalla macchina

Kernel-level Thread vs User level thread immagine





Funzioni per la gestione dei Thread

thread_create(th, func, args)

1. Alloca il TCB (Thread Control Block)
2. Alloca lo stack
3. Inserisce dentro allo stack lo **stub** che serve per far partire il thread. Chiama **stub(func, args).**

4. Mette func, args sullo stack, cioè permette di eseguire func (args) cioè la funzione che deve eseguire il thread con i parametri in ingresso
5. inserisci il TCB nella **lista pronti**

thread_exit(exit_status)

1. Rimuovi il thread dall'elenco pronto
2. Libera lo stato **per-thread-state** composto da TCB e stack

Quando il thread sta uscendo

- Sposta lo stato su **FINISHED**.
- Sposta il **TCB** dall'elenco pronto all'elenco finito.

thread_join(th, retval)

Parametri
thread_da_aspettare: il thread di cui aspettiamo la terminazione
&retval: locazione di memoria dove vado a raccogliere il dato ritornato dalla *exit()*.

Quando chiami `join()` su un thread:

1. Il thread chiamante **si blocca** cioè viene messo in stato di attesa fino a quando il thread specificato termina.
2. Dopo la terminazione del thread, l'esecuzione del thread chiamante riprende normalmente.
3. Assicura che il thread venga **ripulito correttamente** dal sistema.

thread_yieald()

Serve per **cedere volontariamente il processore** da parte del thread chiamante, in modo che il sistema possa far eseguire un altro thread. E' essenziale nel caso di **cooperative multi-threading** per evitare che altri non vengano mai eseguiti. Si usa anche in sistemi **preemptive** per migliorare **l'equità** dell'esecuzione.

- **User-level threads** sono gestiti da una **libreria in spazio utente** (es. Pthreads). Il **SO non è consapevole** dell'esistenza dei singoli thread. Lo **scheduler è nella libreria**, che decide quale altro thread eseguire (non il kernel), questo ha il vantaggio di velocizzare il context switch. Di solito usa **cooperative multi-threading** (un thread che non chiama *yield()* può monopolizzare gli altri).

- **Kernel-level threads** sono gestiti direttamente dal **SO**. Il kernel **vede ogni thread** e li può schedulare in modo indipendente. Usa **preemptive multi-threading**. Quando chiami yield(), si fa una **chiamata di sistema** per dire al kernel: "Questo thread è pronto, ma può aspettare". Più **costoso** il context switch.

Quando viene fatta a livello utente, viene richiamato il library scheduler che viene gestito dal Run Time Support. Solitamente a livello utente viene usato il. Mentre quando si

implementano a livello del Kernel vengono usate le chiamate di sistema, è il SO a schedulare i thread e a pensare a tutto. Il context switch ha minore costo implementato a livello utente.

La chiamata yield prerilascia il processore, il thread chiamante rientra nella lista pronti e lascia che qualche altro thread entri al posto suo. Utile per eseguire + thread che hanno la stessa priorità, viene molto utilizzata nella cooperative-multithreading per eseguire tutti i thread nella lista pronti.

Thread in a process

I thread all'interno di un processo aumentano il parallelismo nascondono ritardi dovuti a grandi operazioni di calcolo o I/O, possono essere implementati in maniera diversa, qui sotto riporto alcuni esempi.

Early Java, implementazione user-level

Gestione dei thread in User space tramite una libreria che si occupa di effettuare la gestione dei thread: creazione, attesa, terminazione e context switch. Per le operazioni sui thread non viene fatta alcuna chiamata al SO. Questo aumenta portabilità su computer diversi e velocità di context-switch. In questo modo però non si ha il pre-emptive multithreading, e se viene effettuata una chiamata su un thread viene bloccato tutto.

Linux, MacOS implementazione kernel-level

La gestione dei thread avviene tramite sistema operativo. (Vedi implementazione in kernel space).

Windows, Scheduler Activations

é un meccanismo che permette di combinare i vantaggi dell'implementazione in user space e kernel space dei thread.
Il kernel assegna i processori (**activations**) alla libreria a livello utente, in questo modo posso effettivamente **sfruttare il parallelismo** utilizzando + processori.
La libreria thread implementa il context switch e decide quale thread eseguire successivamente. Quindi la gestione dei thread viene **affidata alla libreria**.
Il kernel esegue una chiamata di **upcall** ogni volta che necessita di una decisione di scheduling cioè o quando si libera un processore o quando un thread esegue un'operazione di I/O e il thread deve essere rimpiazzato. Questo richiede una complessità di implementazione molto alta.

Threads + Asynchronous I/O

Come funziona il modello asincrono

Meccanismo: il processo invoca `aio_read()` → gestita dal kernel che avvia l'I/O e ritorna immediatamente, il processo continua a runnare. Al completamento dell'operazione di I/O il **Kernel** riceve un **interrupt** dalla componente, quindi invia un **upcall** di ritorno contenente i dati.
Flusso temporale:

1. Process: `aio_read()` syscall → Kernel programma I/O, ritorna subito.
2. Process esegue altre operazioni.
3. Disk effettua lettura.
4. Al termine, OS riceve interrupt dalla I/O e prepara il risultato.
5. Process chiama `aio_return()` per ottenere i dati.

thread + modello asincrono:

Obiettivo: combinare threading e I/O per minimizzare blocchi e massimizzare parallelismo.
Implementazione tipica: un pool di worker thread che lanciano I/O e attendono completion via callback o event queue.

Flusso:Main thread sottomette più `aio_read()`. Worker threads elaborano completions (ricevono segnali o eventi). I/O e calcolo procedono in parallelo senza blocchi.

Thread context switch

1. Salva i registri (contesto) del vecchio thread dallo stack del kernel nel TCB
2. Sposta il TCB del vecchio thread nell'elenco pronto o in una lista d'attesa
3. Seleziona un nuovo thread dall'elenco pronto
4. Ripristina i registri del nuovo thread da TCB allo stack del kernel
5. Metti il TCB del nuovo thread nello stato in esecuzione

Possono avvenire per 2 cause:

- **Volontariamente**
  - **yield** (pre rilascio forzato del processore)
  - **join** (se il thread che aspetto non è ancora terminato)
- **Involontariamente**
  - Interrupt (Premi un tasto sulla tastiera.)
  - Eccezione (Un programma cerca di dividere per zero)

User-level thread / kernel-level thread:

1. Salvo i registri del vecchio TCB
2. Switcho al nuovo stack e al nuovo TCB
3. Carico i nuovi registri dal TCB
4. Return (**IRET o MOVS/SUBS in kernel mode**)

Thread switch on an interrupt

Lo switch tramite interrupt può avvenire tramite **timer** o **I/O interrupt**, notifico al SO che qualche altro thread dovrebbe runnare.

Simple version

1. Interrupt handler salva i registri nello stack del kernel
2. Alla fine dell'interrupt handler chiamo **switch_threads** per scegliere un nuovo thread da eseguire.
3. Se lo scheduler decide di switchare ad un thread differente allora caricherò il nuovo thread e aggiorno il TCB del vecchio thread.
4. Quando l'interrupt handler ritorna, riprende nel nuovo contesto del thread.

Faster version

1. **L'handler dell'interrupt salva** direttamente **lo stato del thread nel suo TCB** (senza passare per lo stack del kernel).

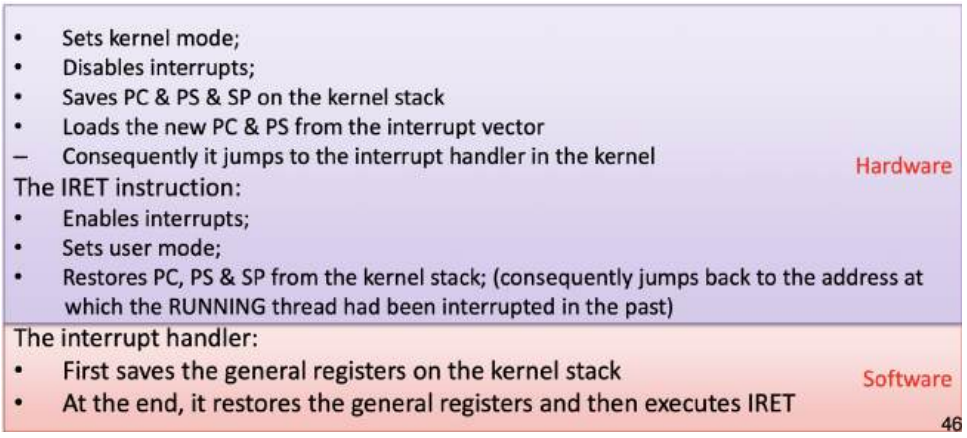
2. **Carica il nuovo contesto direttamente dai registri salvati nel TCB** del thread **nei registri della CPU**. Questo viene fatto **dall'interrupt handler**

3. **Quando l'interrupt termina, il nuovo thread riprende direttamente l'esecuzione**, senza bisogno di ulteriori passaggi nello stack del kernel.

Thread switch - overhead

• A causa del salvataggio e del ripristino dei registri
• A causa della gestione delle code TCB
• Potenziale invalidazione della cache
• Se il thread proviene dallo stesso processo, l'invalidazione della cache potrebbe non verificarsi
• Operazioni indotte sul gestore di memoria – Eccezioni di indirizzo – Difetti di pagina – Invalidazione MMU/TLB

Context switch - example



1) Initial situation during the execution of SVC instruction (**USER MODE**)

TCB T1	
State	Running
PC	????
PS	16F2
SP	????
R1	????
R2	????

TCB T2	
State	Ready
PC	A12C
PS	16F2
SP	A275
R1	25CC
R2	F012

Kernel stack	
0FFF	
0FFE	
0FFD	
0FFC	
0FFB	
0FFA	

Registers	
PC	1880
PS	16F2
SP	2880
R1	4500
R2	CD31

address	5000
PS	AA45
Interrupt vector	

Base kernel SP	0FFF

2) After interrupt (**KERNEL MODE**)

TCB T1	
State	Running
PC	????
PS	16F2
SP	????
R1	????
R2	????

TCB T2	
State	Ready
PC	A12C
PS	16F2
SP	A275
R1	25CC
R2	F012

Kernel stack	
0FFF	1880
0FFE	16F2
0FFD	2880
0FFC	
0FFB	
0FFA	

Registers	
PC	5000
PS	AA45
SP	0FFC
R1	4500
R2	CD31

address	5000
PS	AA45
Interrupt vector	

Base kernel SP	0FFF

3) After temporary storage of registers (**KERNEL MODE**)

TCB T1	
State	Running
PC	????
PS	16F2
SP	????
R1	????
R2	????

TCB T2	
State	Ready
PC	A12C
PS	16F2
SP	A275
R1	25CC
R2	F012

Kernel stack	
0FFF	1880
0FFE	16F2
0FFD	2880
0FFC	4500
0FFB	CD31
0FFA	

Registers	
PC	5000 + ?
PS	AA45
SP	0FFA
R1	??
R2	??

4) After storage of registers of T1 and restore of registers of T2 (**KERNEL MODE**)

TCB T1	
State	Waiting
PC	1880
PS	16F2
SP	2880
R1	4500
R2	CD31

TCB T2	
State	Running
PC	A12C
PS	16F2
SP	A275
R1	25CC
R2	F012

Kernel stack	
0FFF	A12C
0FFE	16F2
0FFD	A275
0FFC	25CC
0FFB	F012
0FFA	

Registers	
PC	5000+?
PS	AA45
SP	0FFA
R1	??
R2	??

5) During execution of IRET at address 5100 (**KERNEL MODE**)

TCB T1	
State	Waiting
PC	1880
PS	16F2
SP	2880
R1	4500
R2	CD31

TCB T2	
State	Running
PC	A12C
PS	16F2
SP	A275
R1	25CC
R2	F012

Kernel stack	
0FFF	A12C
0FFE	16F2
0FFD	A275
0FFC	
0FFB	
0FFA	

Registers	
PC	5100
PS	AA45
SP	0FFC
R1	25CC
R2	F012

6) At the end of IRET (**USER MODE**)

TCB T1	
State	Waiting
PC	1880
PS	16F2
SP	2880
R1	4500
R2	CD31

TCB T2	
State	Running
PC	A12C
PS	16F2
SP	A275
R1	25CC
R2	F012

Kernel stack	
0FFF	
0FFE	
0FFD	
0FFC	
0FFB	
0FFA	

Registers	
PC	A12C
PS	16F2
SP	A275
R1	25CC
R2	F012

Synchronization

Synchronization

Quando i thread accedono in lettura/scrittura di variabili in memoria condivise l'esecuzione dipende da come si alternano il processore e i thread questo causa race condition.

Motivazioni

- La schedulazione dei thread **non avviene in maniera deterministica**, le interruzioni possono avvenire in qualsiasi momento.
- Il programma potrebbe non essere eseguito nell'ordine in cui è stato scritto a causa del riordino eseguito dal compilatore
- Le istruzioni non sono tutte atomiche e quindi potrebbe interrompersi l'esecuzione del thread anche durante una di queste.

Reordering del compilatore

- Il compilatore per ottimizzare il codice riordina le istruzioni per usare la pipeline in maniera più efficiente.
- Il riordino delle istruzioni non altera il comportamento del programma a single-thread, mentre potrebbe dare un comportamento inaspettato per applicazioni multi-thread a variabili condivise perché queste ultime potrebbero cambiare in qualsiasi momento.
- Questo può essere sistemato attraverso l'uso di strumenti di sincronizzazione che rendono le ops esclusive.

Race-condition: Si verifica quando **l'output dipende dall'ordine in cui vengono eseguite le istruzioni**, si verifica quando **almeno una** di queste è **di scrittura**.

Tecniche di sincronizzazione

- **Barriera di memoria**: All ops before barrier complete before barrier returns, No op after barrier starts until barrier returns. La barriera di memoria **si assicura** che **le istruzioni non vengano eseguite in ordine diverso rispetto a quello con cui sono state scritte**. Ci assicurano quindi il funzionamento di programmi multi-thread a memoria condivisa.

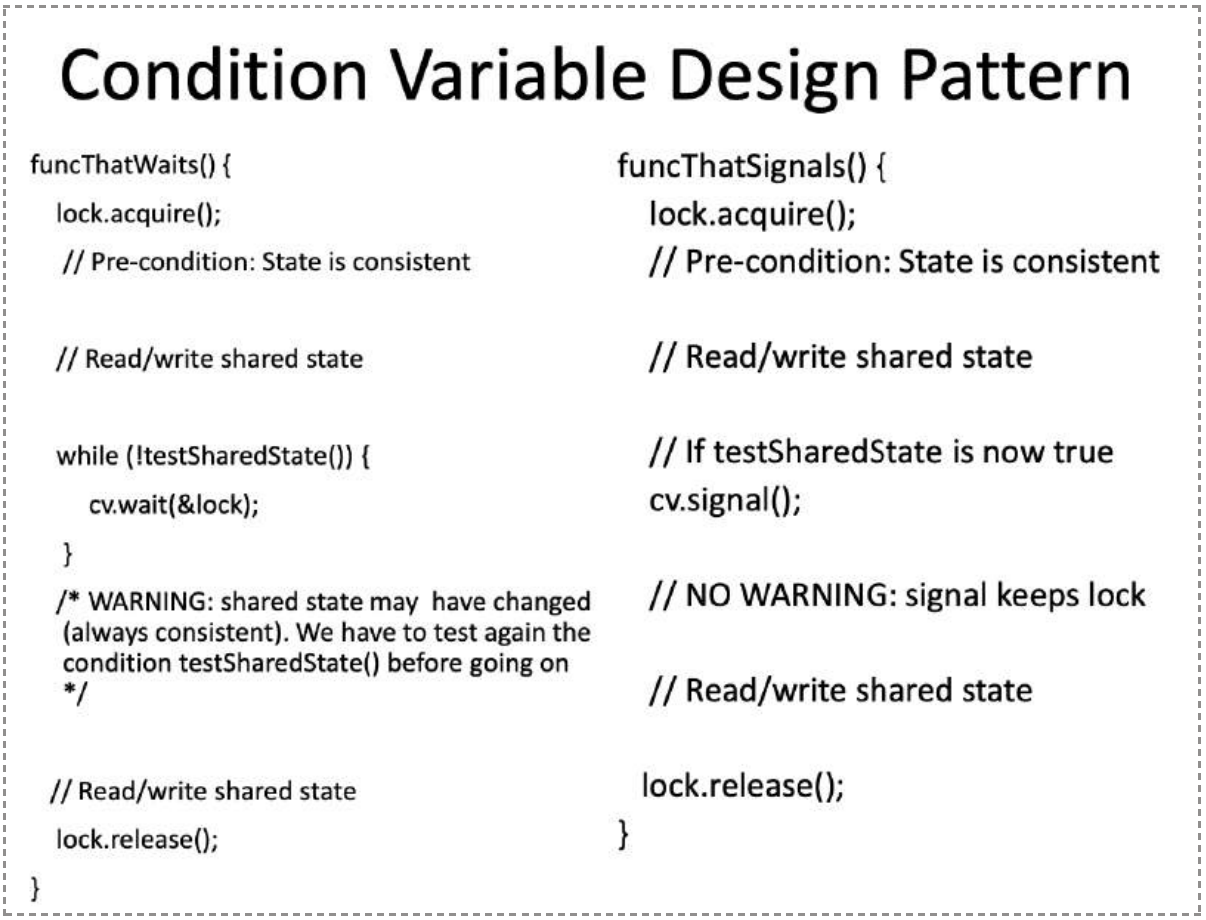
- **Mutual exclusion:** solo un thread può eseguire un preciso frammento di codice alla volta.

- **Lock:** variabile di sincronizzazione che garantisce la mutua esclusione, blocca quando accede alla sezione critica, rilascia quando lascia la sezione critica, aspetta se quella sezione è già bloccata.

- **Variabili di condizione**: una variabile di condizione è un oggetto di sincronizzazione che consente a un thread di attendere in modo efficiente una modifica allo stato condiviso protetto da un blocco. Deve **sempre** essere **utilizzata** con un **Mutex**. Una variabile di condizione ha tre metodi:
    1. **Wait** : Questa chiamata rilascia atomicamente il blocco (rilascio del mutex) e sospende l'esecuzione del thread chiamante, posizionando il thread

chiamante nella lista d'attesa della variabile di condizione. Successivamente, quando il thread di chiamata viene riattivato, riacquista il blocco (aspetta il mutex) prima di tornare dalla chiamata di attesa.

2. **Signal**: Questa chiamata prende un thread dalla lista di attesa della variabile di condizione e la contrassegna come idonea all'esecuzione (cioè, inserisce il thread nella lista di pronte del pianificatore). Se nessun thread è nella lista d'attesa, il segnale non ha alcun effetto. (notify)
3. **Broadcast**: Questa chiamata toglie tutti i thread dalla lista d'attesa della variabile di condizione e li contrassegna come idonei all'esecuzione. Se nessun thread è in lista d'attesa, la trasmissione non ha alcun effetto. (notifyall)



Gli **spurious wakeup** sono risvegli casuali che possono avvenire in `cnd_wait()` anche se nessun thread ha effettivamente chiamato `cnd_signal()`. Per evitarli, bisogna **sempre** verificare la condizione dentro un ciclo `while`:

```
while(needToWait()) {condition.Wait(lock)};
```

Mesa VS Hoare semantics

Mesa:

- Ampiamente utilizzata
- **Signal**: mette un thread in attesa nella lista dei thread pronti senza rilasciare il lock. **Non assicura al thread l'esecuzione.**

Hoare:

- Più semplice dimostrare la condizione di liveness
- **Signal:** prende il thread dallo stato di attesa e lo manda in esecuzione dandogli lock e processore. Quando il waiter ha terminato riprende il thread che ha chiamato la *signal()*, che era stato messo in *stato di attesa*.
- La wait viene fatta all'interno di un if.

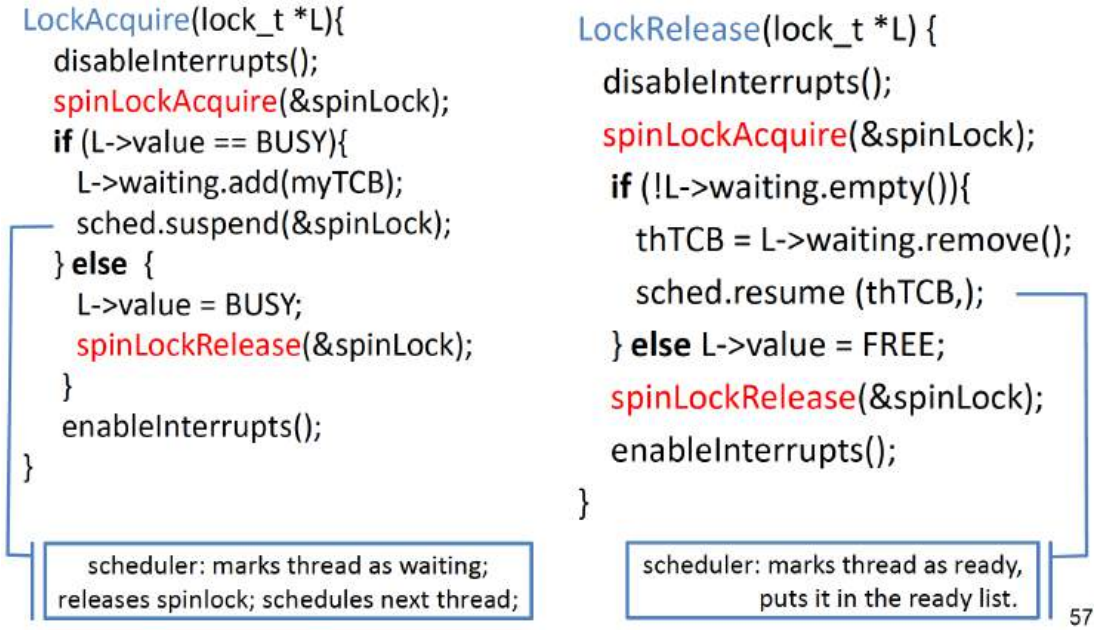
Implementazione Uniprocessor lock

Su un uniprocessore possiamo fare read-modify-write *atomicamente* **disabilitando le interruzioni**.
Interrupt non controllato: Se le interruzioni non vengono disabilitate, è possibile che un interrupt hardware o un cambio di contesto avvenga proprio mentre il thread sta accedendo a una risorsa condivisa.
❌ **Problema**: Non funziona bene su multiprocessori, perché altri core possono comunquSe accedere alla risorsa.

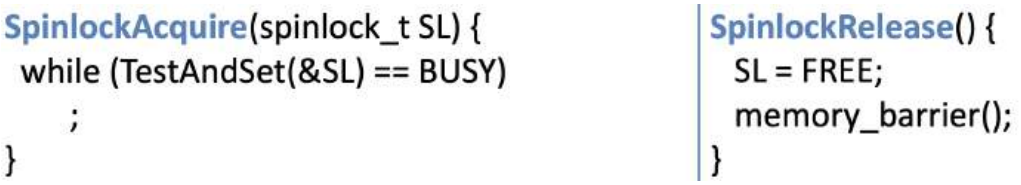
```
class Lock {
  private:
    int value = FREE;
    Queue waiting;
  public:
    void acquire();
    void release();
}

Lock::acquire() {
    TCB *chosenTCB;

    disableInterrupts();
    if (value == BUSY) {
        waiting.add(runningThread);
        runningThread->state = WAITING;
        next = readyList.remove();
        thread_switch(runningThread,
                      chosenTCB);
        runningThread->state = RUNNING;
    } else {
        value = BUSY;
    }
    enableInterrupts();
}
```

```
Lock::release() {
// next thread to hold lock
    TCB *next;

    disableInterrupts();
    if (waiting.notEmpty()) {
    // move one TCB from waiting
    // to ready
        next = waiting.remove();
        next->state = READY;
        readyList.add(next);
    } else {
        value = FREE;
    }
    enableInterrupts();
}
```

Implementazione Multiprocessore lock



Problema: disabilitare le interruzioni non è abbastanza, perché se le disabilito su un core gli altri possono comunque operare sugli oggetti condivisi.
Risoluzione: implementazione delle **spin lock**
La **spin lock** è un meccanismo di lock nella quale il processore aspetta all'interno di un loop (**polling**) che la risorsa torni libera. Assume che la risorsa sia detenuta per un *breve periodo di tempo*.
È usata dal sistema operativo per proteggere la lista pronti, *non andando in wait il thread non vi è alcuna sospensione e non c'è nessun context switch*.



Per l'implementazione delle spinlock si utilizzano istruzioni speciali fornite dalla microarchitettura cioè disposte dall' HW.

- **Test and set** -> funzioni **atomiche** che verificano lo spinlock di una locazione in memoria.
- **Compare and swap (CAS)**

I semafori

I semafori sono una struttura dati composta da un intero + una coda
I semafori sono definiti come segue:

- Un semaforo ha un **valore non negativo**.

- Quando viene creato un semaforo, il suo valore può essere inizializzato a qualsiasi numero intero non negativo

Le uniche operazioni permesse sono (operazioni atomiche):

- ***P()*** attende che il valore sia positivo. Quindi, decresce atomicamente il valore di 1 e restituisce.

- ***V()*** incrementa atomicamente il valore di 1. Se ci sono dei thread in attesa in P, uno è abilitato, in modo che la sua chiamata a P riesca a diminuire il valore e restituisce.

- Non sono consentite altre operazioni su un semaforo; in particolare, nessun thread può leggere direttamente il valore corrente del semaforo.

I semafori **permettono l'implementazione delle variabili di condizione**, inoltre nei linguaggi moderni abbiamo dei **monitor** ovvero funzioni che esclusive e che si escludono tra loro quindi solo una funzione monitor alla volta può essere eseguita da al più un thread.

Implementazione multiprocessore

```
P(sem){
  disableInterrupts();
  spinLockAcquire(&spinLock);
  if (sem->value == 0){
    sem->waiting.add(myTCB);
    suspend(&spinLock);
  } else {
     sem->value --;
     spinLockRelease(&spinLock);
  }
  enableInterrupts();
}
```

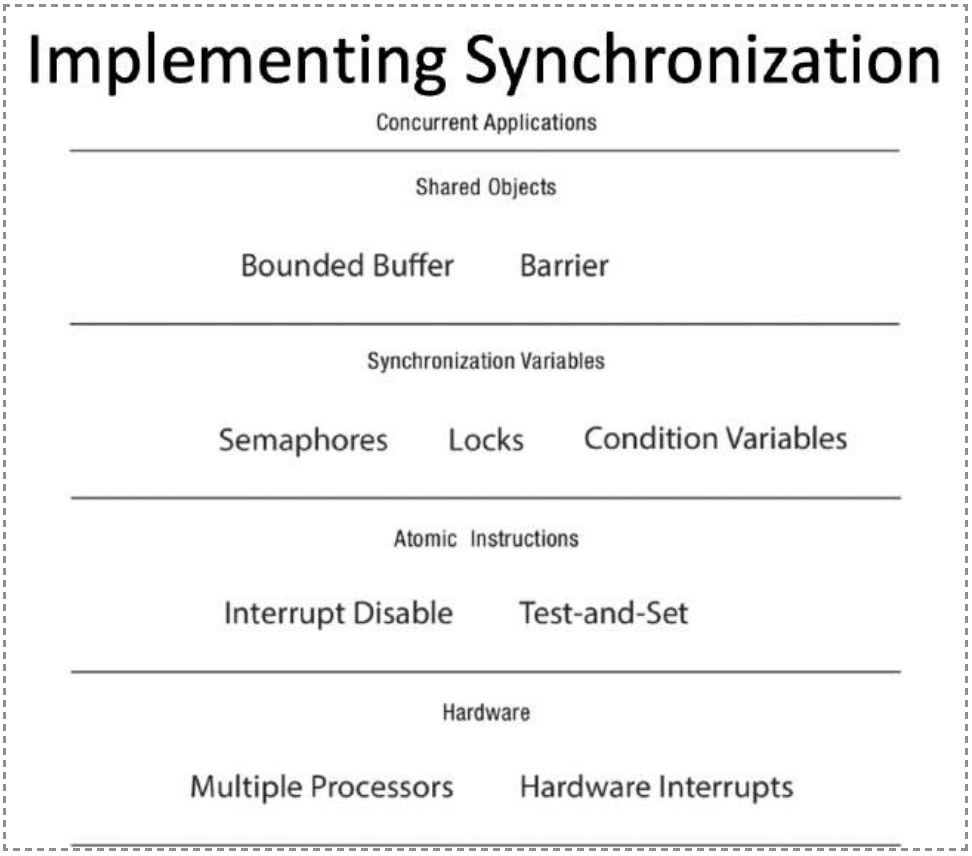
Invokes scheduler; context switch; enables interrupts

```
V(sem) {
  disableInterrupts();
  spinLockAcquire(&spinLock);
  if (!sem->waiting.empty()){
     thTCB = sem->waiting.remove();
     readyList.Append(thTCB);
  }
  else sem->value ++;
  spinLockRelease(&spinLock);
  enableInterrupts();
}
```

Throws semaphore to the awakened thread!

61

Implementing Synchronization



Multi-Object Synchronization

Multi-Object Synchronization

Per programmi molto grandi, per ragioni prestazionali, dobbiamo usare più mutex. Per evitare di avere sezioni critiche molto grandi, voglio ridurle a piccole sezioni che riguardano il solo oggetto condiviso.
Più oggetti implica più mutex da dover utilizzare. Cerchiamo di rendere le sezioni critiche più brevi possibili.

PROBLEMA: Il pericolo di usare + mutex è che c'è il rischio di entrare in deadlock.

In questo capitolo affronteremo cos'è la **deadlock** e come evitarlo.

L'aspetto prestazionale è molto importante nella programmazione multi-threading. Il fatto di avere tanti thread non comporta un aumento della velocità. Ad esempio se tanti thread contendono grandi sezioni critiche si perde tanto tempo nel contendersi i lock e questo implica un degrado delle prestazioni.

Problema false-sharing ovvero thread che scrivono in memoria su parole diverse che però appartengono alla stessa linea di cache che viene invalidata, questo porta ad un degrado delle prestazioni. (Vedi Memory Hierarchy).

Risorsa: qualunque oggetto passivo di cui un thread può richiedere l'accesso in maniera esclusiva (buffer, lock, ecc.).

Una risorsa può essere:

- **Preemptive** (Prerilasciabile): può essere ceduta in qualsiasi momento, questo tipo di risorsa **NON può creare deadlock** (CPU..)
- **Non Preemptive** (Non prerilasciabile): non può essere prerilasciata, queste sono quelle gestite dai LOCK. Questo tipo di risorse può causare deadlock

DEADLOCK vs STARVATION
Nel **deadlock** non ho possibilità di uscita, sono bloccato, faccio prima a spegnere e riaccendere.
Nella **starvation** ho un'**attesa indefinita** nella quale però posso uscire grazie ad un evento che potrebbe accadere anche se non so quando.
Si parla di deadlock quando un insieme di processi (non tutti) che sono tutti in attesa e non hanno modo di svegliarsi. Se c'è deadlock => starvation ma non vale il contrario.

Condizioni necessarie per la deadlock

1. **Risorse limitate:** ho un # di thread maggiore rispetto al # di risorse
2. **Non Preemptive:** una volta acquisite non posso rilasciarle
3. **Wait while holding:** aspettare per una risorsa mentre se ne detiene un'altra
4. **Attesa circolare** di richieste: guarda il problema dei filosofi a cena

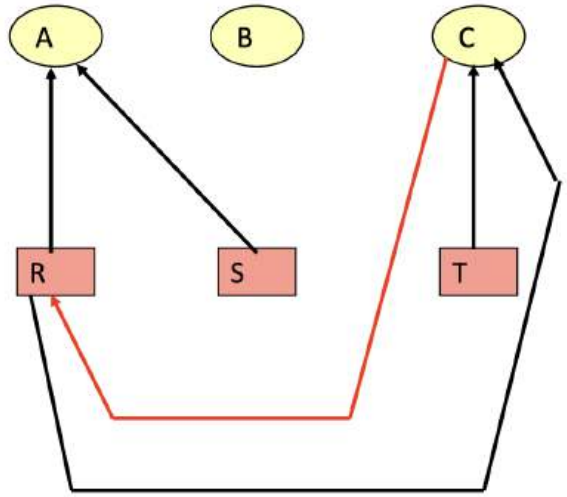
Quindi basterà invalidare una di queste condizioni per non entrare in deadlock.

Per descrivere le attese circolari basta disegnare il grafo delle risorse, dove ogni nodo è un processo/thread o una risorsa, mentre gli archi:

- l'arco parte dal processo verso la risorsa: rappresenta la richiesta della risorsa da parte del processo
- l'arco parte dalla risorsa verso il processo: rappresenta che la risorsa è stata acquisita dal processo

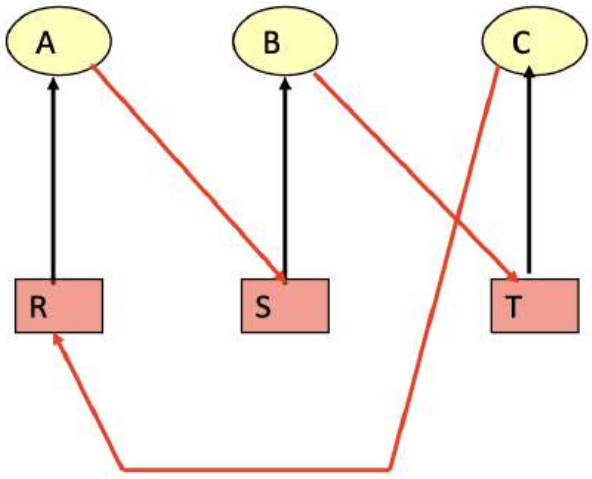
Se questo grafo **contiene un ciclo** (tra più processi, non tra un processo e una risorsa che gli viene assegnata) allora abbiamo 2 scenari:
– Risorse per **istanze singole**: un ciclo implica una condizione di deadlock
– Risorse per **istanze multiple**: un ciclo implica un possibile deadlock noi non consideriamo questo caso.

In questo caso **NON** si verifica la **DEADLOCK**



(risorsa = quadrato, processo = cerchio)

In questo caso si verifica la **DEADLOCK**



Come gestire il deadlock

1. **Detect and fix** : la faccio succedere e lo sistemo.
2. **Static prevention** : analisi statiche.
3. **Dynamic prevention**: analisi dinamica, es. algoritmo del banchiere.

Soluzione #1 Detect and fix

Scansiono il grafo delle richieste e qualora si creino questi cicli, ci metto una pezza. Come?
Sopprimo il thread:

- Un modo può essere quello di killare uno dei thread del ciclo e assegnare le sue risorse ad un altro thread. Questo è un approccio rischioso, molto costoso però in generale è difficile e complicato.
- Killo il thread che ha il minimo impatto quando viene ucciso un thread il SO pensa a rendere nuovamente disponibili le risorse.

Operazione di rollback:

- Rollback, *come nei DB*, si torna ad uno stato precedente e far ricominciare l'esecuzione sperando in uno scheduling diverso che faccia partire l'applicazione. Però non c'è determinismo nello scheduling dei thread quindi non è sicuro. Complicato da usare perché devo gestire i check point e assicurarmi di poter sempre tornare indietro.

Solitamente si preferisce quello di sopprimere il thread i SO adottano **la politica dello struzzo**:
nei SO Unix vengono allocate delle risorse per permettere all'amministratore di loggarsi, aprire una shell e killare i processi anche se si dovesse creare la deadlock.

Soluzione #2: Deadlock prevention (tecnica di prevenzione statica)

Invalidare una delle 4 condizioni di deadlock

1. **Risorse limitate**: virtualizzarle o spool per renderle non limitate
   Lo **spooling** è una tecnica che usa una memoria intermedia (spool) per gestire l'accesso alle periferiche, permettendo ai processi di continuare l'esecuzione senza attendere la periferica. Un esempio classico è la stampa: i documenti vengono accodati e stampati uno alla volta, senza bloccare il sistema.
2. **No preemption:** consentire al sistema di prendere le risorse con la forza.
3. **Wait while holding**: o riesco a prendere tutte le risorse insieme o non ne prendo nessuna. devo sapere in anticipo tutte le risorse che andrò ad utilizzare, questo non è sempre possibile.
4. **Circular waiting**: *Lock ordering*: quando lockiamo o rilasciamo le risorse lo facciamo seguendo un ordine ben preciso.

No wait while holding

- Un processo dovrebbe richiedere tutte le risorse in anticipo
- Rilasciare le lock quando si termina l'utilizzo della risorsa
- **Free lock programming,** utilizzo di algoritmi senza load e store con test and set e CAS. .

Soluzione #3: Deadlock prevention (tecnica di prevenzione dinamica)

Algoritmo del banchiere

Nuove assunzioni:
Molteplicità delle risorse cioè una risorsa può essere acquisita da più entità. Fino ad ora abbiamo sempre considerato le risorse di disponibilità 1 che implicano la mutua esclusione.

Molteplicità: numero di risorse di un dato tipo
Disponibilità : numero di risorse attualmente disponibili

Possiamo immaginarci le risorse come un vettore di lunghezza uguale alla molteplicità, mentre i valori sono 0 1 se già assegnati o no a qualcuno

Si chiama algoritmo del banchiere perché cerca di mimare il comportamento di un banchiere che presta soldi sapendo che a chi li ha prestati può restituirglieli. I soldi sono le risorse e i processi sono le persone, l'algoritmo valuta se prestare le risorse in quanto deve sapere se queste gli torneranno indietro per evitare la deadlock.

Questo è un algoritmo dinamico che quindi viene eseguito in run-time. L'algoritmo si comporta come un gestore di risorse.

Per far funzionare l'algoritmo del banchiere:

- Ogni processo deve dichiarare in anticipo le risorse che andrà ad utilizzare*
- Alloca le risorse dinamicamente
- L'assegnazione della risorsa viene concessa se posso soddisfare anche le altre richieste.

Quindi l'algoritmo del banchiere si assicura quando assegna le risorse di essere sempre in uno stato sicuro che mantiene lo stato di sicurezza ogni volta che assegna una risorsa in questo modo non andiamo mai in uno stato non sicuro nel quale rischiamo di andare in deadlock. Quindi per garantire lo stato sicuro non assegna necessariamente le risorse ai processi ma li mette in attesa.

Stato sicuro:
– esiste almeno una sequenza di allocazione delle risorse per la quale sicuramente non entro in deadlock.

Stato non sicuro:
– non esiste alcuna sequenza di esecuzione che garantisca il completamento di tutti i processi senza deadlock.
– Tuttavia, lo stallo non è certo.

Stato compromesso:
– Indipendentemente da come vengono allocate le risorse da questo punto in poi, lo stallo è inevitabile
– Tutte le possibili sequenze portano allo stallo.

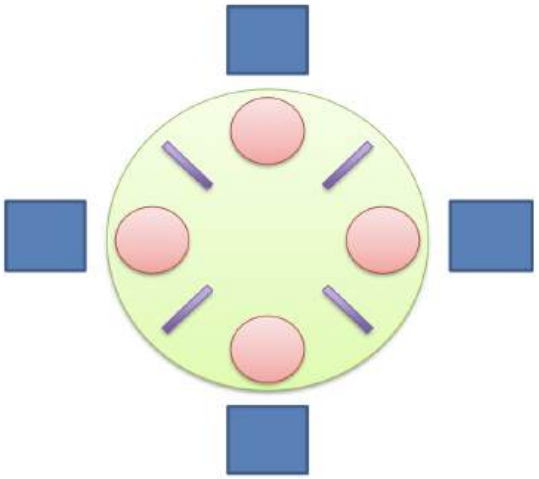
Principio chiave: l'algoritmo **simula di assegnare la risorsa** ad un certo processo e se rimane in uno stato sicuro la assegna veramente. L'algoritmo del banchiere quindi non fa altro che modificare la disponibilità delle risorse, ogni volta che assegna le risorse fa una simulazione e verifica che questo non porti ad uno stato non-safe

Nell'algoritmo del banchiere ogni risorsa è rappresentata da un vettore di length = molteplicità che ne misura la disponibilità. Mentre per ogni processo abbiamo un vettore che rappresenta le risorse assegnate e un vettore delle risorse che gli devono essere ancora assegnate. Termino quando ho marcato tutte le risorse da assegnare.

Questo è un algoritmo molto **costoso** che deve sapere in ogni momento tutte le risorse assegnate ad ogni processo e quelle ancora da assegnare, con l'utilizzo di questo algoritmo siamo sicuri che non ci sia neanche la possibilità di entrare in deadlock.

- Stato sicuro non implica che non si possa andare in stallo.
- Stato non sicuro non implica che ci sia stallo.
- Se un processo è in stallo non può essere nella coda pronti.

Filosofi a cena



N filosofi si incontrano per cena ogni filosofo medita e mangia. Per mangiare ogni filosofo ha bisogno di due bacchette solo che le bacchette sono solo N e non possono quindi soddisfare tutti i filosofi allo stesso tempo.

CODICE DEL FILOSOFO

```
while (true) {
    penso();   // il filosofo pensa
    // il filosofo di indice i decide di mangiare
    lockBastoncino[i].Acquire();   // acquisisce il bastoncino di destra
    // il filosofo si sospende se non può acquisire il bastoncino alla sua sinistra
    lockBastoncino[(i+ 1) mod N].Acquire();
    mangia();  // il filosofo di indice i mangia
    // rilascia i bastoncini
    lockBastoncino[(i+ 1) mod N].Release();
    lockBastoncino[i].Release();
}
```

Dal codice notiamo che ogni filosofo tenta di prendere prima la bacchetta alla sua sinistra e successivamente quella alla sua destra. Si potrebbe creare deadlock in quanto tutti successivamente tentano di acquisire la bacchetta alla loro destra e se quello accanto l'ha già acquisita rimangono in attesa infinita, deadlock.
Come evitare lo stallo:

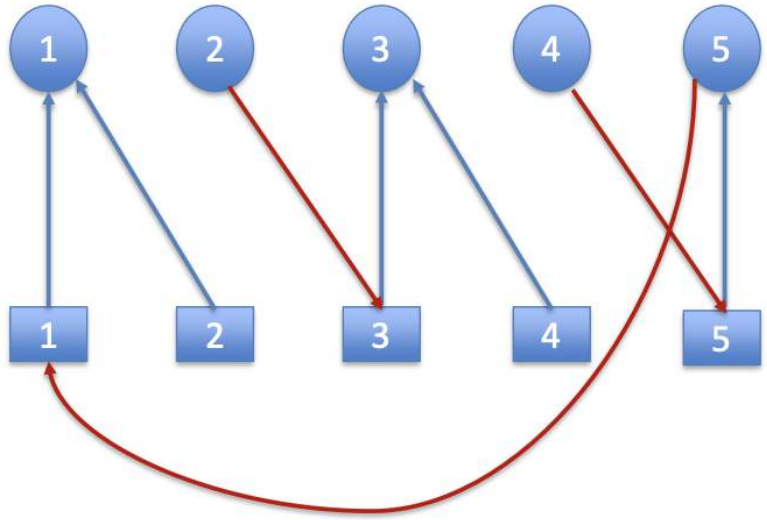
1. Usiamo una soluzione asimmetrica in cui differenziamo due tipi di thread, quelli di *indice pari* e quelli di *indice dispari* oppure fissiamo un indice tra le bacchette e diciamo che i filosofi devono acquisire prima quelle che hanno un indice maggiore. **In questo modo imponiamo un ordinamento nell'acquisizione delle bacchette per i filosofi**.
2. Un filosofo o acquisisce le due bacchette insieme oppure attende che si liberino, in questo modo invalidiamo la condizione necessaria alla deadlock wait while holding.
3. Si usa una soluzione basata sul concetto di monitor in cui, per l'acquisizione dei bastoncini, si tiene di conto dello stato dei filosofi vicini e si agisce di conseguenza.

Soluzione #1
Nella prima soluzione del problema dei filosofi andiamo a differenziare i comportamenti dei filosofi di indice pari e quelli di indice dispari. In questo modo andiamo a dire ai filosofi di indice pari di richiedere la bacchetta alla loro sinistra e quelli di indice dispari quella a destra. In questo modo i filosofi adiacenti andranno a prendere la stessa bacchetta e se non riescono ad acquisirla si bloccano subito. Questa tecnica **non elimina completamente la possibilità di starvation**, cioè che un filosofo aspetti per tanto tempo, ma **evita blocchi globali**. Si spezza la circolarità del blocco.

Implementazione della soluzione

```
while (true) {
    penso();
    if (i % 2) { // filosofo con indice dispari
       // prima acquisisce la bacchetta di sinistra e poi quella di destra
       lockBastoncino[i].Acquire();   lockBastoncino[(i+ 1) mod N].Acquire();
       mangia();  // il filosofo di indice i mangia
       lockBastoncino[(i+ 1) mod N].Release();    lockBastoncino[i].Release();
    } else  { // filosofo con indice pari
       // prima acquisisce la bacchetta di destra e poi quella di sinistra
       lockBastoncino[(i+ 1) mod N].Acquire();    lockBastoncino[i].Acquire();
       mangia();  // il filosofo di indice i mangia
       lockBastoncino[i].Release();   lockBastoncino[(i+ 1) mod N].Release();
```

Possibile grafo delle risorse



Soluzione #2

In questa soluzione andiamo a rompere la condizione di deadlock *wait while holding* implementando un' attesa attiva ovvero i filosofi tentano di acquisire entrambe le bacchette insieme e se non ci riescono ritentano. Garantisce che non avvenga ***deadlock*** ma potrebbe accadere ***starvation***.

```
while (true) {
    penso();   // il filosofo pensa
    // il filosofo di indice «i» decide di mangiare
    PrendiBastoncini(&lockBastoncino[i], &lockBastoncino[[(i+ 1) mod N);
    mangia();  // il filosofo di indice «i» mangia
    RilasciaBastoncini(&lockBastoncino[i], &lockBastoncino[[(i+ 1) mod N);
}
```

```
PrendiBasoncini(lock1, lock2) {
  while(true) {
    lock1.Acquire();
    if (lock2.tryAcquire()) return;  // successo
    lock1.Release();      // rilascio il bastoncino e ....
    swap(lock1, lock2); // .... provo nell'ordine contrario
  }
}
```

```
RilasciaBastoncini(lock1, lock2) {
    lock1.Release();
    lock2.Release();
}
```

Rimuovo la condizione "wait while holding"

Soluzione #3

Questa soluzione utilizza i monitor, i filosofi in questa soluzione hanno uno stato che può essere:

1. "**fame**" => richiede i bastoncini
2. "**mangia**" => possiede i 2 bastoncini
3. "**pensa**" => senza bastoncini

Si utilizzano inoltre le seguenti variabili:

- Una variabile mutex per la mutua esclusione sul vettore stato
- Attesa filosofo[N] vettore di variabili di condizione utilizzate per la sospensione dei filosofi. La variabile attesaFilosofo[i] è usata per l'attesa del filosofo di indice i.

I monitor sono meccanismi di sincronizzazione di più alto livello rispetto ai semafori ed alla lock. Possiamo vederli come una collezione di procedure, variabili (di stato e di condizione), strutture dati, confinate all'interno di un modulo. I processi possono chiamare le procedure del monitor (API) senza avere accesso alle sue strutture dati. I monitor sono strutture del linguaggio. Il C non ha i monitor come costrutto nativo ma si possono emulare. Java si: synchronized.

Protocollo per mangiare

```
prendiBastoncini(i) {   // metodo del monitor
    // il filosofo di indice i ha deciso di di mangiare
    mutex.Acquire();
    stato[i]= HaFame;
    while  (stato[(i- 1) mod N] == Mangia) || (stato[(i+ 1) mod N] == Mangia )  {
        attesaFilosofo[i].wait(&mutex); // devo attendere che siano liberi entrambi
    }
    stato[i]= Mangia;      // ha ottenuto entrambi i bastoncini
    mutex.Release();
}
```

Protocollo per pensare

```
rilasciaBastoncini(i) {      // metodo del monitor
    mutex.Acquire();
    stato[i]=Pensa;
    if (stato[(i - 1) mod N]== HaFame) && (stato[(i - 2) mod N] != Mangia) {
        // riattiva il filosofo (i-1) mod N se può ottenere entrambi i bastoncini
        stato[(i - 1) mod N = Mangia;
        attesaFilosofo[(i- 1) mod N].signal();
    }
    if (stato[(i + 1) mod N]== HaFame) && (stato[(i + 2) mod N] != Mangia) {
        // riattiva il filosofo (i+1) mod N se può ottenere entrambi i bastoncini
        stato[(i + 1) mod N = Mangia;
        attesaFilosofo[(i + 1) mod N].signal();
    }
    mutex.Release();
}
```

Scheduling

Scheduling

Intro

Lo scheduling dei processi consiste nell'operazione di assegnare in un certo ordine i processi alla CPU.

Terminologie

• **Task**. Una richiesta dell'utente. Un compito è spesso chiamato anche lavoro. Un compito può essere di qualsiasi dimensione, dal semplice ridisegnare lo schermo per mostrare il movimento del cursore del mouse al calcolo della forma di una proteina appena scoperta. Quando si discute di pianificazione, usiamo il termine **compito**, piuttosto che thread o processo, perché un singolo thread o processo può essere responsabile di più richieste o attività degli utenti. Ad esempio, in un elaboratore di testi, ogni carattere digitato è una singola richiesta dell'utente di aggiungere quel carattere al file e visualizzare il risultato sullo schermo.

Response time Il tempo percepito dall'utente per svolgere un'attività.

Predictability Bassa varianza nei tempi di risposta per le richieste ripetute.

Throughput La velocità con cui le attività vengono completate.

Scheduling overhead Il tempo per passare da un compito all'altro.

Fairness Uguaglianza nel numero e nella tempestività delle risorse assegnate a ciascun compito.

Starvation La mancanza di progressi per un compito, a causa delle risorse asserite a un compito di priorità più alta.

Workload = set di task

Scheduling algortithm

1. prende un carico di lavoro come input.
2. decide quali attività fare per prime.
3. metrica delle prestazioni (throughput, latenza) come output.

Uniprocessor scheduling

Priority-based scheduling
Ad ogni task viene assegnata una **priorità**

- Lo scheduler della CPU seleziona sempre il processo nella coda pronta con la massima priorità – di cui FIFO e SJF sono casi speciali di pianificazione prioritaria
- La pianificazione basata sulla priorità può essere preemptive e non preemptive – Se il nuovo processo ha una priorità più alta rispetto a quello in esecuzione è anticipato

Può verificarsi fame: un processo a bassa priorità potrebbe non essere mai eseguito – Può essere utilizzata la tecnica di invecchiamento: aumenta gradualmente la priorità dei processi che attendono nel sistema per lungo tempo.

FIFO (First in First out)
La politica FIFO prevede di schedulare i processi per ordine di arrivo. In questo modo viene garantita equità, tutti i processi vengono eseguiti ma spesso il response time è grande poiché potrebbero arrivare prima processi grandi che vengono eseguiti prima di quelli più piccoli aumentando la latenza. Se i task sono molto variabili in termini di lunghezza fa schifo altrimenti se sono tutti uguali è ammodo.
PROBLEMA -> **Response-time quando arrivano prima task grandi e dopo quelli + piccoli**

SJF (Shortest Job First)

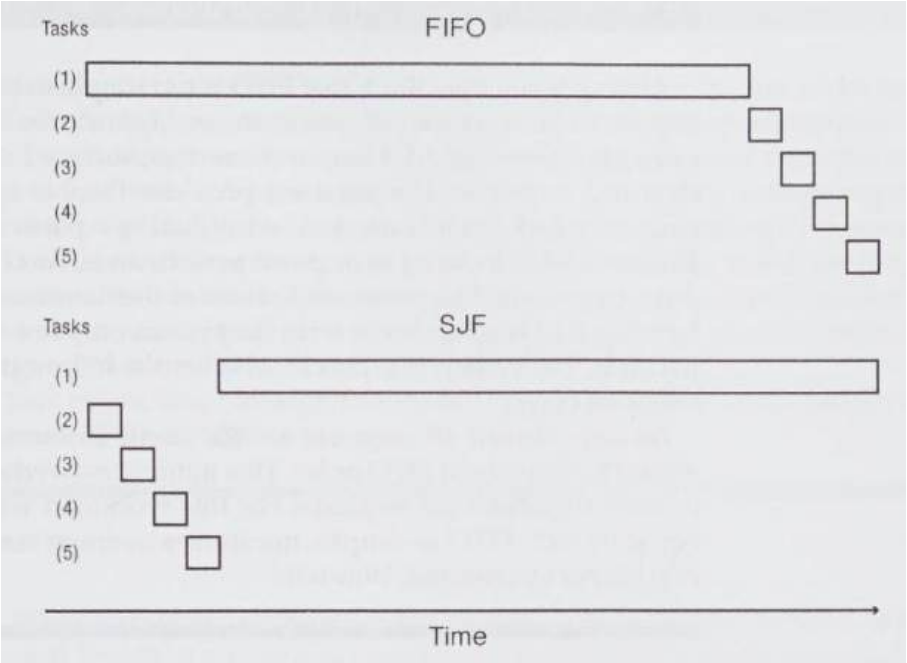
La politica SJF prevede di schedulare i processi in ordine di quanto tempo hanno bisogno per essere processati.
NOTA: nella realtà non è noto a priori quanto tempo richiede il processo alla CPU.
Preemptive se arriva un processo che richiede meno tempo di quello attuale viene fatto lo switch **politica SRTF (Shortest Remaining Time First).**

SRTF (Shortest Remaining Time First)

Questo algoritmo di schedulazione si comporta come un **SJF preemptive**, se arriva un processo che richiede meno tempo rispetto a quello che sta eseguendo attualmente viene switchato.

Versione 1: aspetto che tutti i processi siano arrivati e li schedulo
Versione 2: li schedulo appena arrivano.

PROBLEMA: Starvation, se continuano ad arrivare processi "corti" **i processi più lunghi entrano in starvation** aumentando di molto la latenza. Anche qua se abbiamo job molto varianti in termini di lunghezza avremo un aumento dei tempi di risposta.



Round Robin

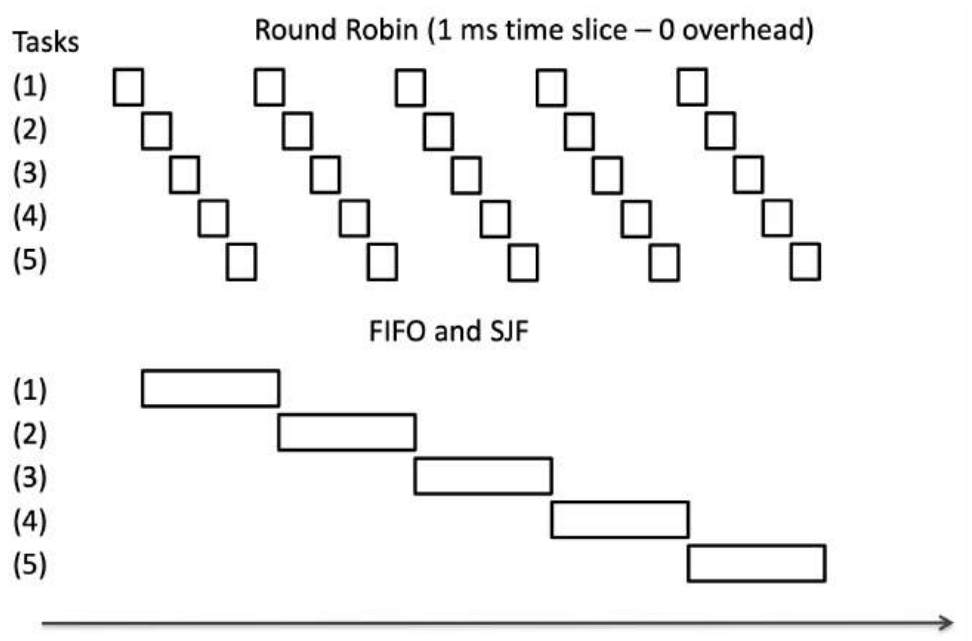
Il Round Robin (**RR**) è un algoritmo di scheduling **preemptive**. Funziona **assegnando a ogni processo un quanto di tempo fisso**. Se il processo non termina entro il quanto, viene interrotto e messo in coda, permettendo al processo successivo di eseguire. I processi vengono eseguiti tutti per un certo tempo nell'ordine di arrivo (**FIFO**).

- Se il quanto di tempo è troppo piccolo perdiamo troppo tempo per overhead dovuto al context switching.
- Se invece è troppo grande diventa come la politica FIFO.

CPU burst → periodo di uso attivo della CPU
I/O burst → È un periodo in cui un processo è in attesa di operazioni o eventi I/O
CPU bound task → usano quanta più CPU viene assegnata. Ridotto I/O burst.
I/O bound tasks → usano poca CPU, ma frequentemente.

PROBLEMA
Abbiamo un problema di **Fairness** quando il **workload** è misto cioè ho sia task CPU bound che I/O bound questo mi porta a diminuire il tempo di reattività dei task I/O bound poiché per un piccolo uso della CPU devono aspettare il loro turno tra processi che hanno bisogno di un quanto di tempo + grande.
Esempi: leggere da un file, stampare a schermo, ricevere dati dalla rete. Di solito fanno un piccolo lavoro sulla CPU e poi **aspettano** che l'I/O finisca.

Abbiamo un problema di **response time** quando dobbiamo eseguire + task della stessa lunghezza a causa di overhead dovuto al context switching e anche senza overhead.



Per risolvere il problema di equità:

Max-Min Fairness

Massimizzare la minima allocazione → Dare priorità al task che ha ricevuto meno CPU.

Come funziona:

1. Se un task ha bisogno di **meno** della quota uguale → **schedulo lui prima**.
2. Il **tempo rimanente** si divide secondo la logica max-min (dare di più a chi ha ricevuto meno).
3. Se tutti i task rimanenti hanno bisogno **almeno della quota uguale** → divido equamente.

Supponendo di avere task 1, 2, 3 dando 100 di budget
Task 1 -> 20
Task 2 -> 50
Task 3 -> 50
Allora assegno rispettivamente 20, 40, 40
Viene usata un'approssimazione di questo approccio per la risoluzione del problema.
Questo però porta ad un costo elevato per un overhead quindi non viene usato ma viene usato come benchmark.

Un algoritmo di condivisione delle risorse è **max–min fair** se:

1. Assegna risorse in modo tale che si massimizzi il minimo assegnamento tra gli utenti.
2. Non è possibile incrementare la risorsa di un utente senza ridurre quella di un altro che abbia pari o minore allocazione.

In altre parole, prima si soddisfano le richieste più piccole, poi si distribuisce equamente il resto.

Problemi: difficoltà di implementazione efficiente in sistemi dinamici a causa della necessità di conoscenza globale, ricalcoli frequenti e potenziali inefficienze (ad esempio, se un task con un fabbisogno di risorse molto elevato è bloccato o in attesa, l'algoritmo potrebbe ritardare l'esecuzione di altri task meno esigenti per non "favorirli" troppo, anche se questi ultimi potrebbero essere completati rapidamente).

MFQ (MultiLevel feedback Queue)

MFQ (Multilevel Feedback Queue) è un'estensione del Round Robin. Invece di avere una sola coda, MFQ ne prevede più di una, ciascuna con un diverso livello di priorità e un diverso quanto di tempo.
I task di priorità più alta precedono quelli di priorità inferiore, mentre i task sullo stesso livello vengono schedulati in Round Robin. Inoltre, i livelli di priorità superiore hanno quanti di tempo più brevi rispetto a quelli inferiori.

Un nuovo task entra sempre al livello di massima priorità. Ogni volta che un task esaurisce il suo quanto, scende di un livello; *ogni volta che cede volontariamente la CPU perché è in attesa di I/O* (task I/O bound)*, rimane allo stesso livello* (o viene promosso di un livello); quando il task termina, esce dal sistema.

Problema
Quando un processo I/O-bound tiene sempre libera la CPU (cedendo per I/O), nel MFQ "base" può capitare che rimanga sempre nei livelli più alti, mentre un CPU-bound che consuma tutto il suo quanto scende sempre verso la coda di priorità inferiore e rischia di non avere mai più occasioni di essere eseguito dalla CPU, dato che potrebbero arrivare sempre nuovi task che entrano alla massima priorità.

Soluzione di Linux al problema
Per evitarlo Linux aggiunge un meccanismo di fair--share che prevede:

- la suddivisione di ogni coda in 2 code distinte.
- Una quota di tempo di CPU ottenuta dividendo equamente il tempo tra tutti i task (della coda).

Le due code per ciascun livello sono:

- Coda "*regolare*": contiene i task che non hanno ancora raggiunto la loro quota di CPU (**coda veloce**).
- Coda " *di riserva*": contiene i task che hanno già superato la loro quota di CPU (**coda lenta**).

Quando lo scheduler sceglie un task da un dato livello, svuota prima la coda regolare; solo se quella è vuota passa alla coda di riserva.

Ogni intervallo di tempo (es. ogni 100 ms) il kernel calcola per ciascun processo quanta CPU ha consumato rispetto alla "quota ideale" (CPU totale divisa per numero di processi). Se un processo ha usato meno della sua quota allora i suoi task vengono **promossi** (o lasciati) nella coda regolare. Se ha usato più della sua quota i suoi task vengono spostati nella coda di riserva (o a questo punto declassati come in MFQ base).

Effetto pratico

- Gli I/O-bound (che cedono spesso) restano in coda regolare e continuano a essere serviti rapidamente, ma non possono usare più di quanto gli spetti in totale.
- I CPU-bound, anche se finiscono in riserva dopo aver consumato molto, torneranno in regolare non appena avranno ricevuto meno CPU della loro quota nel periodo successivo.

Priority inversion problem
L'inversione di priorità si verifica quando un processo ad alta priorità è costretto ad attendere una risorsa (come un blocco o un semaforo) che è attualmente detenuta da un processo a priorità inferiore.
• Le attività a priorità più alta non dovrebbero essere bloccate da attività a priorità inferiore per molto tempo.
• Può accadere che un processo a priorità intermedia precetti continuamente la CPU e il processo a bassa priorità non venga mai eseguito.
• Di conseguenza, il processo ad alta priorità viene effettivamente "invertito" alla priorità più bassa, poiché non può fare progressi.

Multiprocessor scheduling

PROBLEMA MFQ

1. **Cache coherence:** eseguendo i thread su processori diversi introduco il problema di coerenza. Dato che eseguirò uno stesso thread su processori diversi il processore deve riuscire a recuperare il TCB più aggiornato da un qualche core in giro per il mondo. L'obiettivo è di riusare la cache con i dati già caricati per non causare overhead.
2. **Lock centralizzato:** Se pensiamo di avere una singola MFQ per tutti i core dovremmo implementare un lock centralizzato che potrebbe creare un rallentamento a **collo di bottiglia** dovuto al processore che detiene il lock sulle liste MFQ.

Cerchiamo quindi di legare ogni job ad un core. Abbiamo due soluzioni per aumentare la cache coherence 1) Per-processor MFQ
2) Affinity scheduling

Per-Processor MFQ

A ogni processore viene assegnato un set di job organizzato nella sua struttura dati MFQ e una **spinlock** associata. Una volta che un processo si sospende riprenderà l'esecuzione sullo stesso processore (**cache coherence**), per massimizzare l’utilizzo della cache.
Questo approccio riduce significativamente la contesa rispetto a un'unica coda globale e un'unica spinlock.
In caso di disparità di tempo per l’esecuzione dei job che vengono assegnati ai processori, potrei avere che solo un processore lavora mentre gli altri hanno già finito. Questo introduce un problema di bilanciamento. Risolto dall’Affinity scheduling.

Affinity scheduling

Ogni processore ha la sua lista di job implementata con MFQ dove tutte le liste sono su ogni processore, protette da spinlock. Il job tende ad eseguire sullo stesso processore per sfruttare le cache, quindi in qualche modo leghiamo il job al core. L'obiettivo è proprio quello di "legare" il job al processore massimizzando l'affinità, per sfruttare la cache coherence. La cache coherence **riduce l'overhead** dovuto al caricamento dei dati di un job da qualche altro core.

Quando un core finisce il suo lavoro ruba il lavoro (**steal work**) ad altri core . Piuttosto che non fare nulla meglio pagare l'overhead dovuta allo switch. Questa operazione di steal work permette di **bilanciare** il lavoro tra i processori.

Viene implementata una spinlock per ogni coda. Dato che quando rubo il lavoro ad un altro job ho bisogno di sincronizzare le liste tra i core, più core accedono concorrentemente alla solita lista MFQ durante l'operazione di steal work.

Problema dei JOB PARALLELI
Nelle architetture multiprocessore abbiamo :

Job sequenziali -> job che devono essere schedulati seguendo un certo ordine (t1 -> t2 -> t3)
Job paralleli -> job che possono e dovrebbero essere schedulati in maniera parallela per sfruttare il parallelismo (t1, t2,t3 -> t4,t5,t6). Non li devo eseguire sulla stessa CPU poiché non sfrutterei il parallelismo ma li eseguirei in maniera sequenziale.

Cercando di legare un job (thread) ad un core per sfruttare la cache coherence (ci permette di riusare i dati caricati nella cache "privata" del singolo core). Però per thread paralleli non è una buona idea, infatti N thread di uno stesso programma collaborano tra loro (lock, barriere, strutture dati condivise…).

Problema:
Se sospendo un singolo thread "A" mentre gli altri continuano, questi ultimi possono incappare in un punto di sincronizzazione (es. barriere o lock) e rimanere bloccati in attesa che "A" faccia la sua parte.
Nel frattempo il sistema potrebbe aver schedulato altri thread di un differente programma, così quei miei thread in attesa restano fermi più del necessario.

Scheduling Oblivious (Scheduling ignaro, inconsapevole)

Se eseguo lo **scheduling oblivious** ovvero non mi frego se i job sono sequenziali o paralleli, avrò uno scheduling arbitrario. Questo però non ci assicura di sfruttare il parallelismo.

Gang Scheduling

Una possibile soluzione è di mandare tutti i job di un'applicazione su tutti i processori. Ogni volta che dobbiamo fare context switch lo faccio con tutti i job.

Questo non è sempre fattibile ad esempio potrei avere più thread dei core che ho a disposizione, potrebbe succedere che solo pochi dei miei processori sono disponibili per eseguire i task quindi il programma si blocca. Oppure potrei necessitare di meno core di quelli che ho a disposizione, sprecando risorse.

Space Sharing

Lo space sharing **divide i processori** tra **le applicazioni** parallele in esecuzione. A ciascuna applicazione viene assegnato un sottoinsieme dei processori disponibili. Lo scheduler esegue queste applicazioni contemporaneamente, ciascuna all'interno del suo "spazio" allocato (insieme di processori). All'interno di ciascuno spazio allocato, i thread dell'applicazione possono essere schedulati utilizzando qualsiasi metodo (incluso gang scheduling).

1. Più applicazioni parallele possono progredire anche se non ci sono abbastanza processori per eseguire tutti i thread di ogni applicazione contemporaneamente.
2. Uso efficiente delle risorse quando le applicazioni hanno diversi livelli di parallelismo (alcune hanno bisogno di meno processori di altre).

Allocando i processori alle applicazioni, fornisce un certo grado di controllo su come i thread di una determinata applicazione vengono schedulati rispetto agli altri.

Address Translation

Address Translation

Come viene mappata logicamente la RAM?

Virtual memory
Virtual memory è una separazione tra la memoria logica dell'utente e quella fisica. Questo livello di astrazione garantisce che il SO dia l'illusione ai processi di avere più memoria rispetto a quella che realmente hanno (cioè quella fisica).

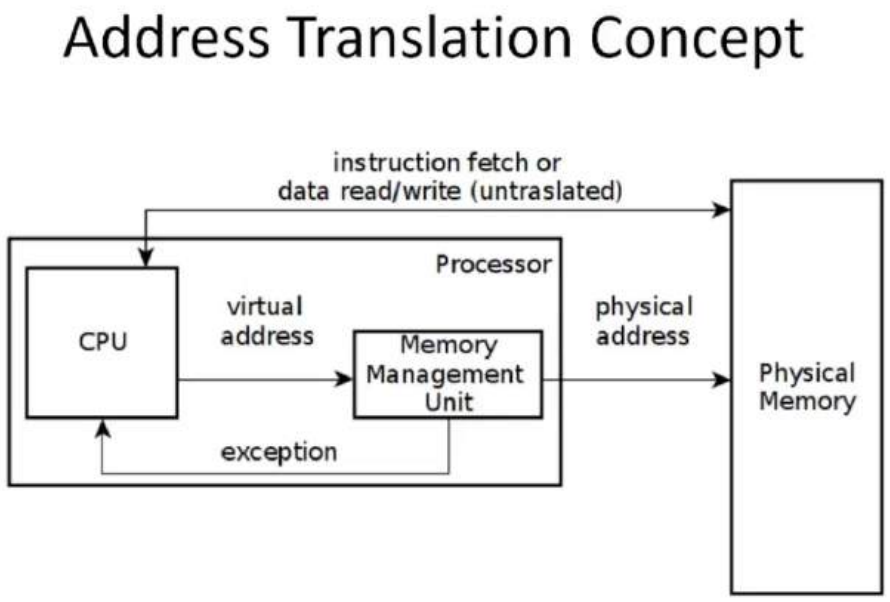
Punti principali
Convertire gli indirizzi logici con indirizzi fisici.
Rendere la conversione flessibile ed efficiente.

Obiettivi principali
Proteggere la memoria, limitandone l'accesso
Condividere la memoria tra processi
Il sistema di conversione deve essere molto flessibile per permettere al SO di allocare dove vuole la memoria.
Deve essere portabile in architetture diverse.

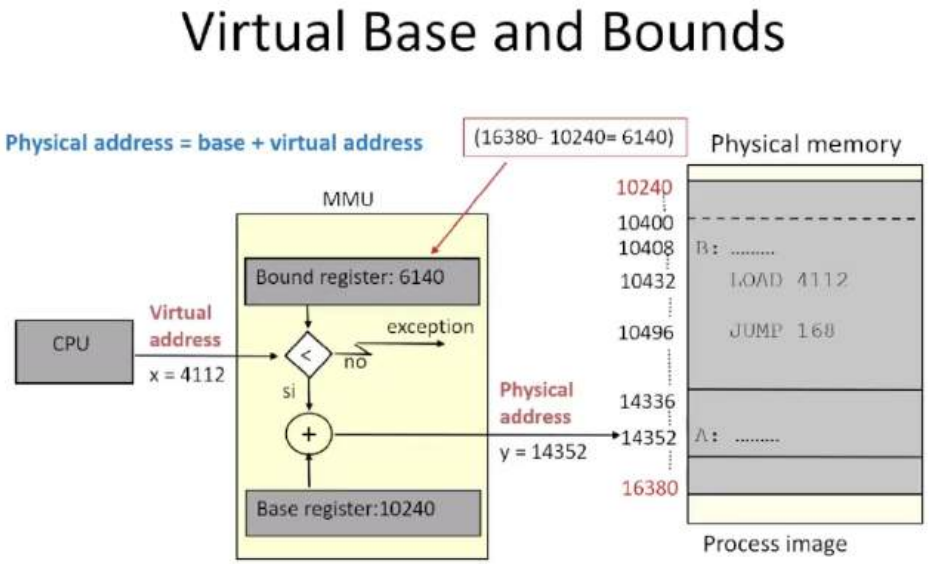
Indirizzo logico o virtuale = quello usato a livello di SO.
Indirizzo fisico = quello usato dal sottosistema di memoria.

La traduzione degli indirizzi serve per isolare i processi per rendere efficiente la comunicazione tra processi e condividere segmenti tra differenti programmi. Per la program inizialization.

Da adesso in poi faremo riferimento a questo tipo di struttura.



Virtual base and bound



La traduzione viene effettuata aggiungendo all'indirizzo virtuale l'indirizzo base. Questi registri possono essere modificati solo dal SO altrimenti il processo violerebbe la segmentazione della memoria assegnata. Questo effettua un controllo se l'indirizzo finisce fuori dalla segmentazione assegnata al processo.

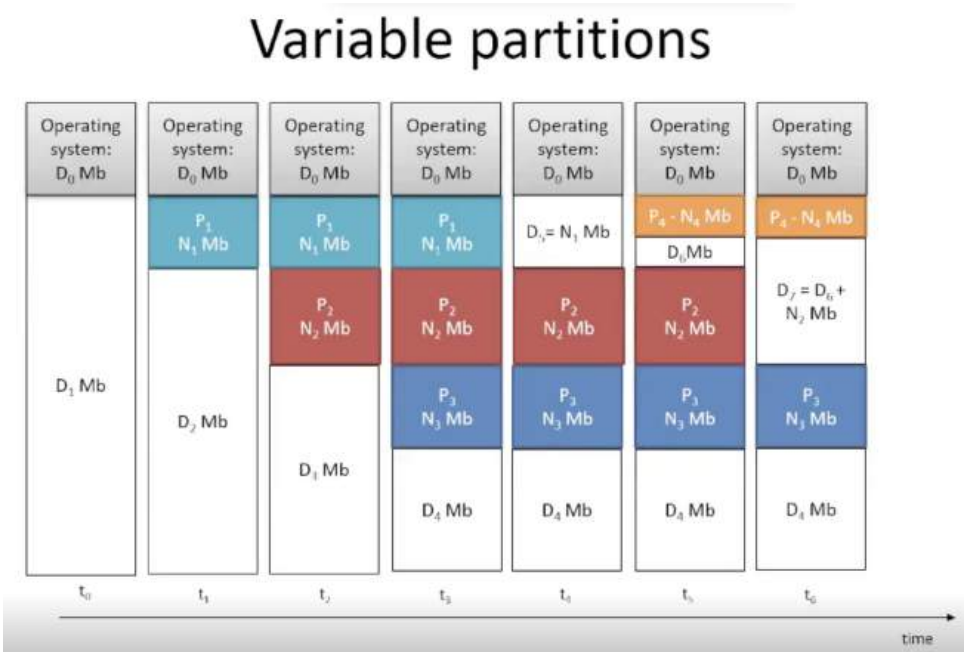
Pro:

- Semplice
- Veloce (2 registri, adder, comparatore)
- Permette di riallocare la memoria senza modificare il processo

Contro:

- Non è flessibile, niente condivisione di codice e dati
- Una volta che ho allocato un segmento non posso ingrandirlo o rimpicciolirlo quindi ho meno flessibilità per il SO.
- Potenzialmente ci potrebbero essere delle riscritture del codice che risiede tutto nel solo segmento in cui il processo ha accesso in lettura e scrittura.

Problema delle segmentazioni variabili

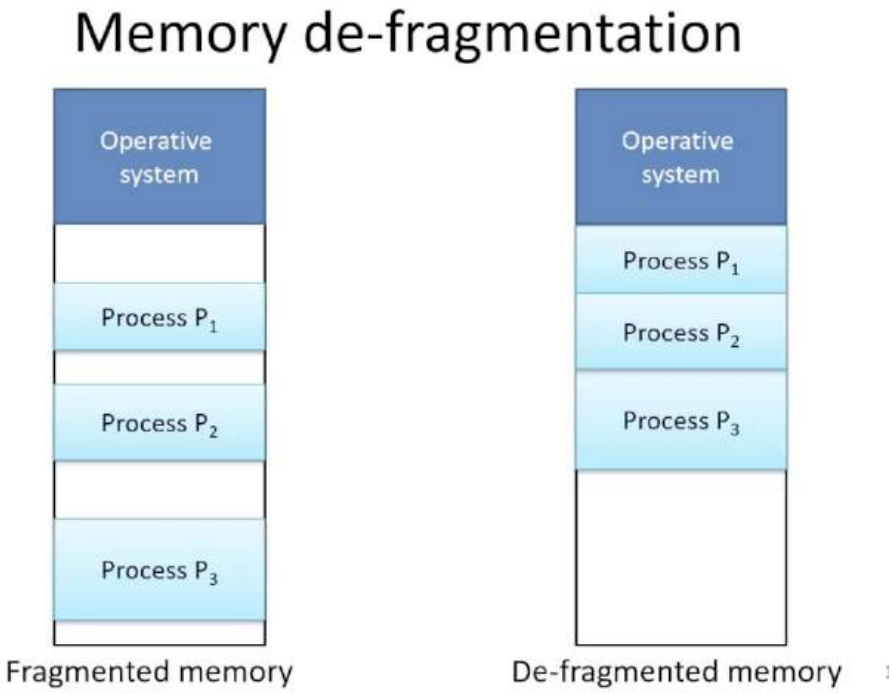


Questa immagine descrive l'allocazione dinamica dei segmenti di memoria ai processi. Nel tempo i processi nascono e muoiono e questo causa frammentazione della memoria. Quindi potrei avere la quantità di memoria disponibile ma frammentata in tanti "buchi". **Problema di frammentazione esterna della memoria**.

Frammentazione esterna della memoria => quando devo allocare un nuovo spazio in memoria e complessivamente avrei abbastanza spazio libero ma frammentato in piccole partizioni non contigue che le rendono inutilizzabili. Questo problema viene causato dall'allocazione di **segmenti di memoria variabili**.

Frammentazione interna della memoria => Nel caso dell'allocazione di blocchi di lunghezza fissata, è che devo prendere la lunghezza del segmento più grande questo significa che quando andrò ad allocare quel blocco ad un processo che non ha bisogno di tutta quella memoria avrò uno spazio inutilizzato **interno** al blocco stesso di memoria.

Possibili soluzioni



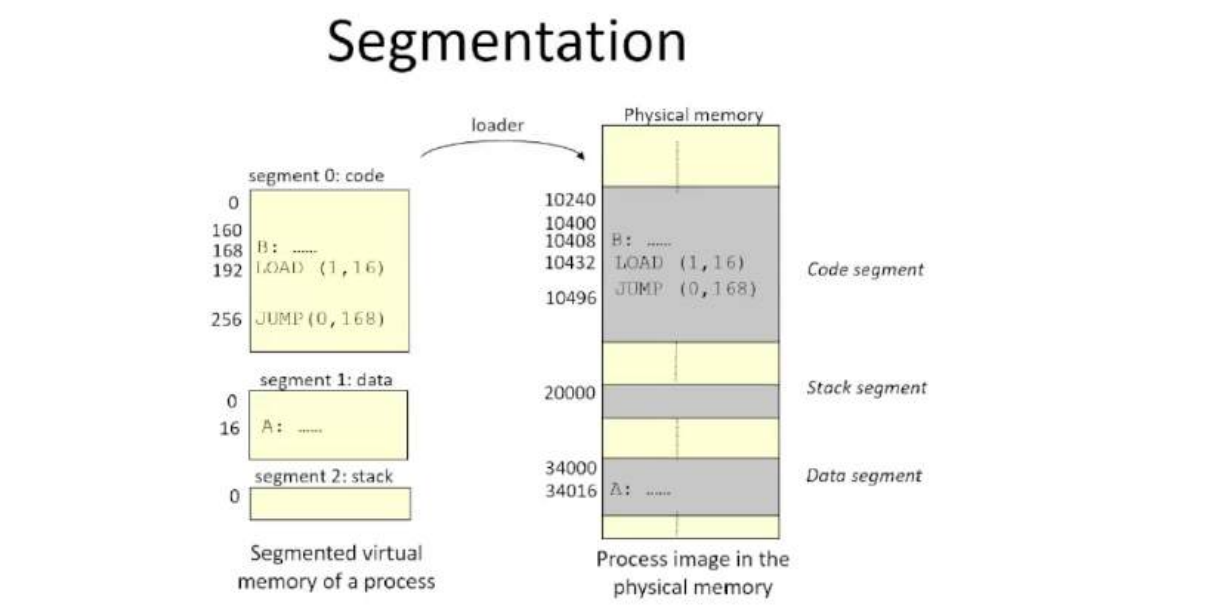
Molto costoso a livello di cicli di clock per copiare i dati da una parte all'altra non si può sempre fare.

Se invece accetto che ci sia la frammentazione della memoria allora posso fare un altro tipo di scelta.
Ottimizzazione del problema
Firs-fit prendo il primo blocco che sia abbastanza grande per contenere il mio segmento di indirizzi.
Best-fit prendo il più piccolo blocco che mi contiene il segmento di memoria

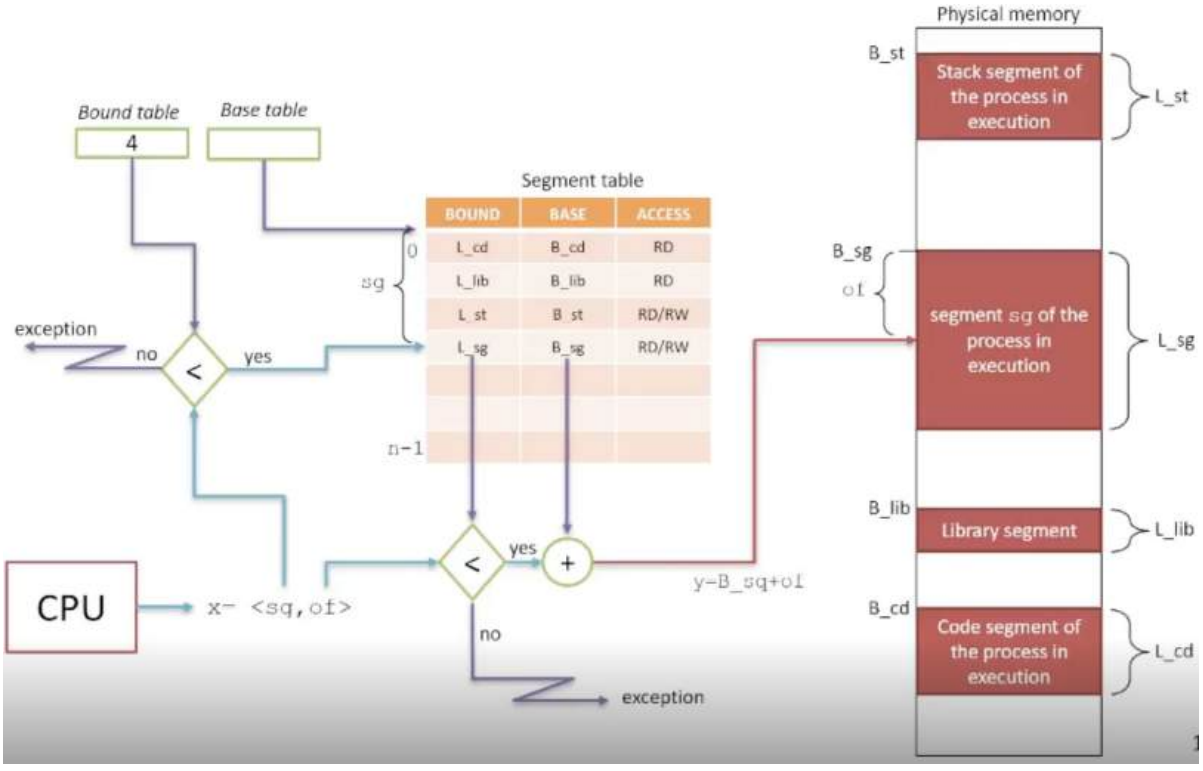
Segmentation

Un segmento è una regione contigua di memoria
Ogni processo ha una tabella di segmenti cioè un array dove i valori sono segmenti.
Un segmento può essere allocato in qualsiasi posizione in memoria, per il processo logicamente sono contigui ma fisicamente sparsi.
Posso assegnare degli speciali bit di permessi per descrivere i privilegi del processo.

I processi possono condividere i segmenti posso duplicare l'entry del segmento su processi diversi per avere il segmento disponibile su processi diversi.



- The virtual address is represented as a pair of values:
  virtual address = <segment index, segment offset>



La tabella della segmentazione se piccola risiede direttamente nel PCB altrimenti in memoria. Funziona come il Base and Bound solo che ho molteplici segmenti e quindi molteplici Base and bound. Posso generare un numero massimo di segmenti diminuendo la frammentazione della memoria.

Ottimizzazione della fork **COPY&WRITE**

Copio la segmentation table nella tabella del figlio, termina la chiamata della fork subito. Marca read-only i segmenti padre-figlio.
Se il figlio scrive in un segmento del padre genera una eccezione trap in to kernel che però viene gestita in maniera differente, cioè crea una copia di quel solo segmento che riscrivo. Miglioro tantissimo le prestazioni.

Ad esempio quando uso fork ed exec evito di copiare e scrivere tutti i segmenti per il figlio che verrebbero totalmente cancellati successivamente dalla exec.
Exec riinizializza la tabella dei segmenti.

Zero on reference:
Posso avere bisogno di allocare memoria sullo heap o sullo stack e quindi genero una speciale eccezione che non segnala un comportamento sbagliato ma semplicemente voglio allocare nuova memoria nello stack o heap. A quel punto posso dare memoria azzerata cioè zone di memoria che sono state "**imbiancate**" per evitare di lasciare traccia di informazioni sensibili. Operazione costosa che viene fatta solo se necessaria.

Pro:
- Posso condividere dati e codice tra processi.
- Posso proteggere le riscritture del codice.
- Posso aumentare o decrementare i segmenti di memoria, **memoria flessibile**
- Uso ottimizzazione della **copy and write**, **zero on reference**.

Contro:
- Gestione complessa della memoria
- frammentazione esterna
- Ogni tanto c'è bisogno di risistemare la memoria cioè fare l'operazione di **deframmentazione**.

Come possiamo risolvere il problema della frammentazione?

L'idea è di usare segmenti di dimensione fissa diamo un nome a questi segmenti di dimensioni fissa li chiamiamo **pagine**, in questo modo eliminiamo il problema della frammentazione esterna.

Page Translation

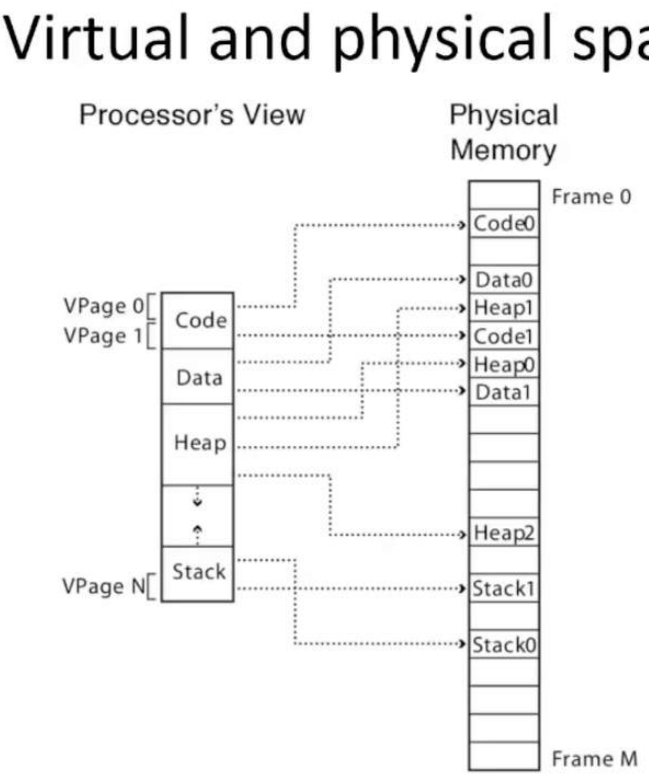
La memoria viene segmentata in pagine tutte della stessa dimensione, a questo punto trovare una pagina libera è facile, però potrei avere il problema della **frammentazione interna**.

Però se la dimensione è piccola la frammentazione interna è limitata, spreco poca memoria. Quindi l'idea è di avere tanti segmenti di piccola dimensione. In questo caso si parla di page table (milioni di entry).

Questa tabella delle pagine è molto più grande della segmentation table quindi non è possibile farla risiedere nella MMU come nella tabella dei segmenti.

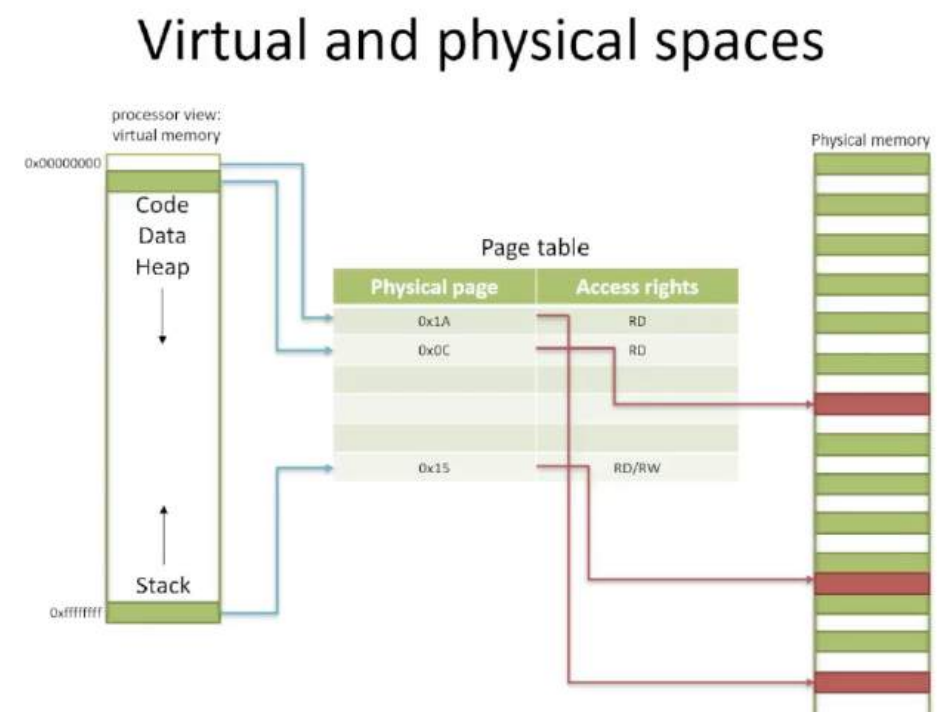
Quindi la tabella delle pagine va tenuta in memoria fisica e nel MMU viene tenuta solo l'indirizzo alla tabella della nella RAM (page frame). Per fare la traduzione degli indirizzi devo accedere alla tabella nella memoria fisica che richiederebbe una nuova conversione degli indirizzi che verrà risolta in seguito.

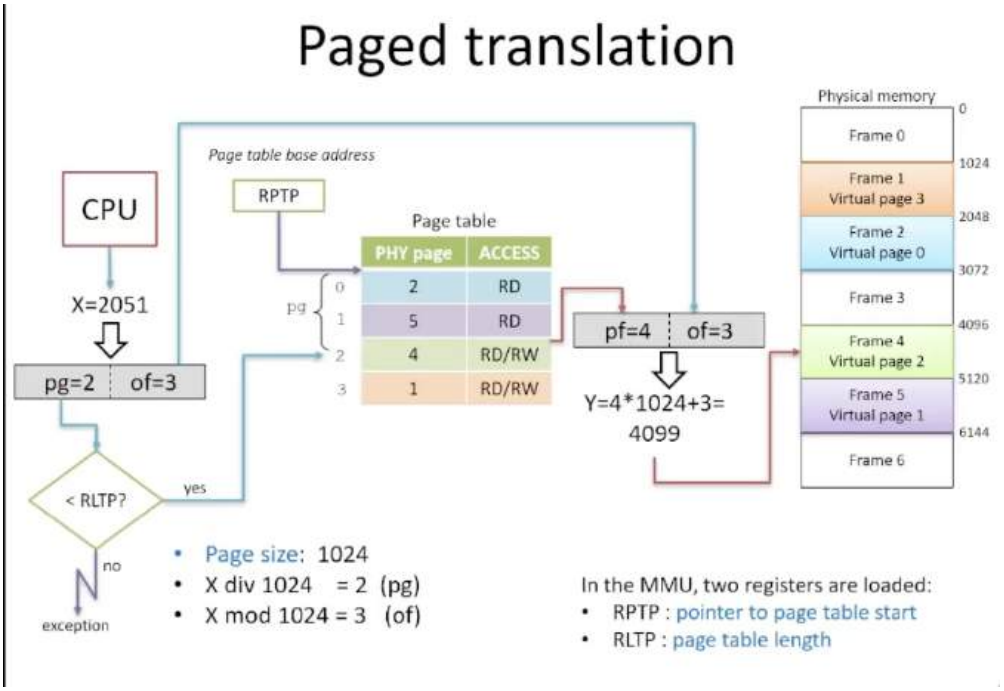
Di base bisogna sapere l'indirizzo in memoria fisica della page table e queste informazioni saranno contenute nel PCB.



Aumento la flessibilità tra memoria logica e fisica.
La frammentazione interna esiste ma è un problema marginale, mentre se avessi una dimensione grande per page frame questo diventerebbe un serio problema, riduco però la dimensione della tabella.

Dim tabella = dim memoria / dim pagina.





Cache TLB nel MMU per tenere traccia degli indirizzi delle page table per sfruttare la località spaziale.

La tabella delle pagine è una per processo, il processo ha l'indirizzo base della tabella e la sua lunghezza.

Domanda
Cosa dobbiamo memorizzare in caso di context switch?
Dobbiamo aggiornare l'indirizzo base della tabella delle pagine e della sua lunghezza.

Mentre la **page table** la dovremo caricare in memoria.

Abbiamo page sharing e Copy on write.

Con la paginazione possiamo Fast program start, posso far partire un programma non caricando tutte le pagine necessariamente.

Cosa succede se gli indirizzi fisici sono messi in maniera sparsa in memoria ?
Per diminuire la dimensione della tabella delle pagine abbiamo 3 tecniche che la ottimizzano nel caso medio.

Paged segmentation

La memoria del processo è segmentata, quindi unisce sia la tecnica della segmentazione a quella dell'impaginazione.

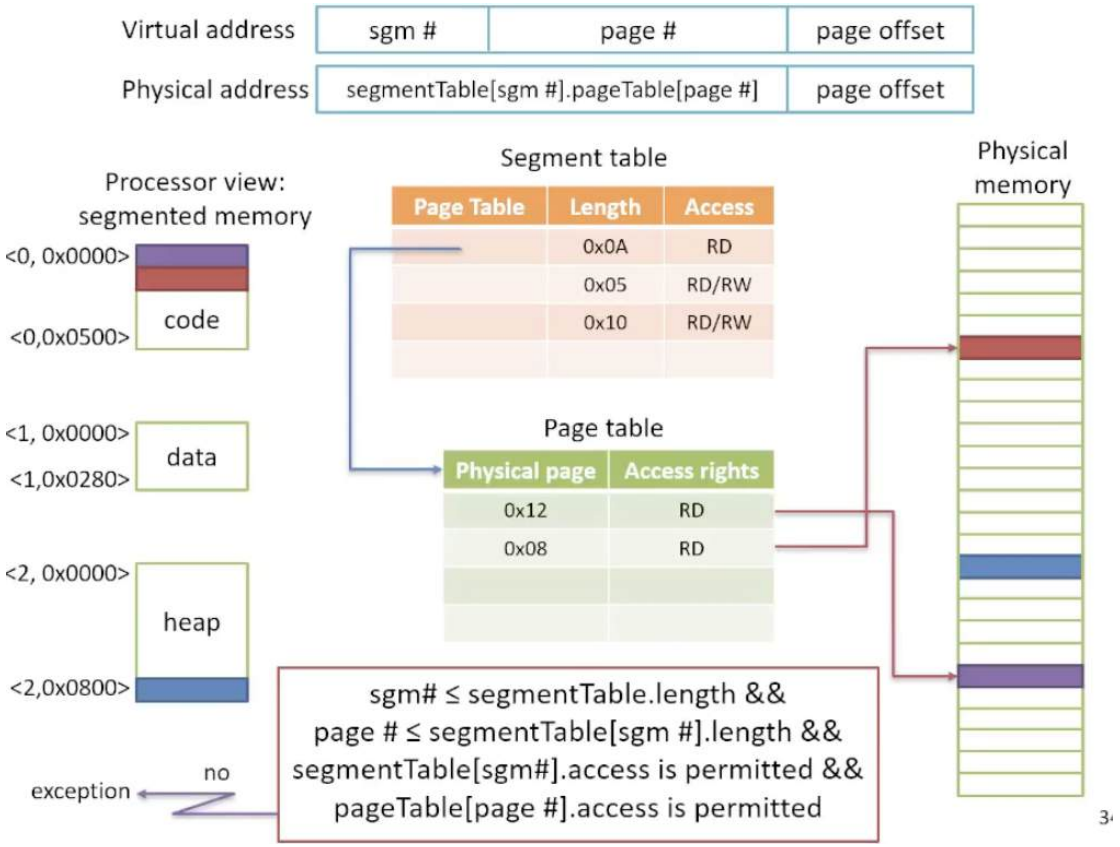
Ogni entry della segment table indirizza:

- Un puntatore alla page table
- Lunghezza della page table
- Permessi di accesso

Le entry della tabella delle pagine sono:

- Page frame
- Permessi di accesso

Segment = lunghezza della tabella del segmento.

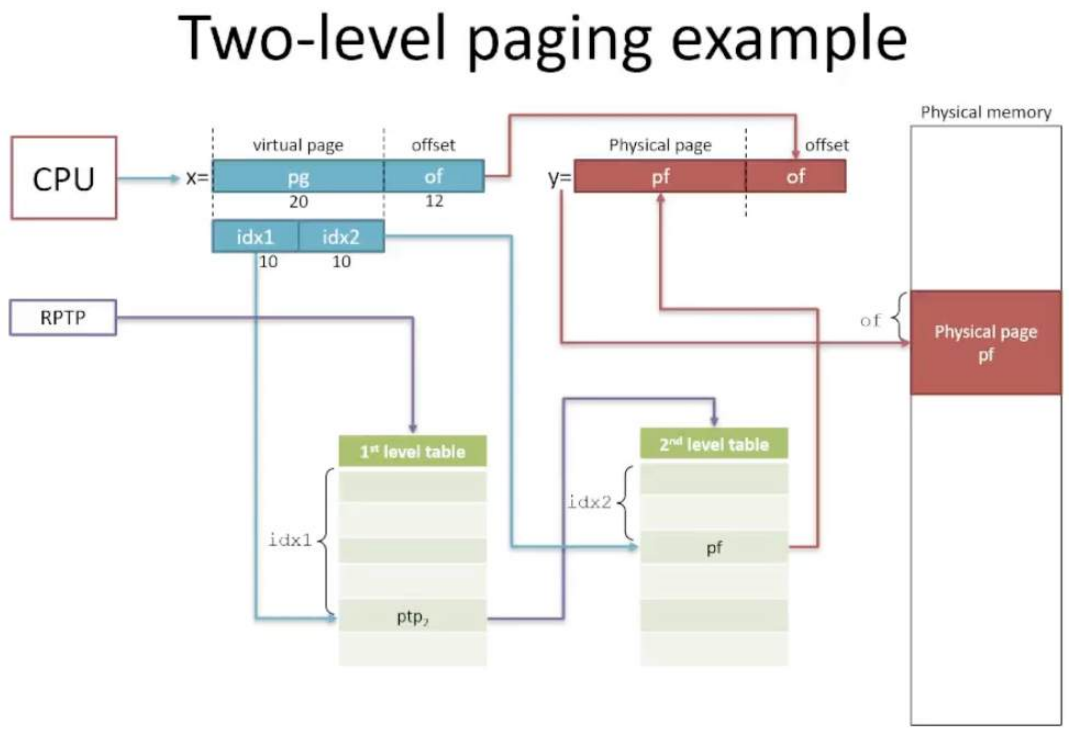


Cosa salvare durante il context switch? Il **SO** salva i **riferimenti alla tabella dei segmenti**.

Multilevel paging

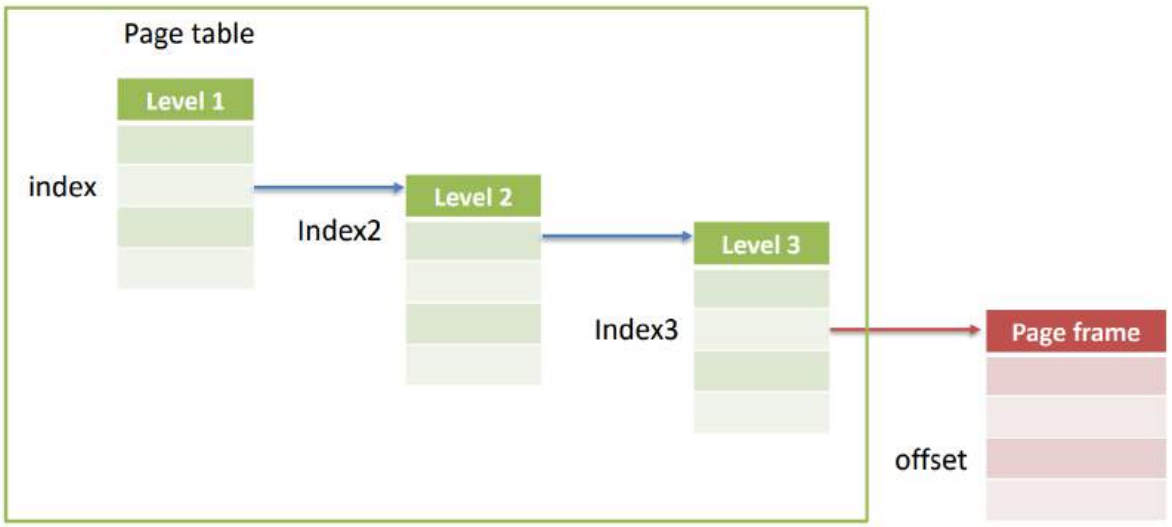
Con questa tecnica avrò più livelli prima di arrivare alla page frame, mentre prima avevo segment table -> page table -> page frame con un indirizzo logico diviso in [index segment table - index page table - offset page frame] ora posso suddividere il mio indirizzo logico in N indici di pagina e infine l'offset della page frame in questo andrò a visitare le page table dove

ogni entry mi porta alla page table successiva fino ad arrivare al page frame.



In questo caso non ho una singola tabella per tutte le entry, ma **più livelli di page table** che non necessariamente devo caricare tutti in memoria. Questo mi permette di risparmiare spazio in memoria ma aumentando però il costo computazionale per la visita delle liste. Durante la visita dell'albero delle pagine dovrò caricare dinamicamente le page table che mi servono.
Generalmente avrò caricato solo un sottoinsieme di tabelle rispetto a quelle che ho.

physical address=pageTable[index].pageTable[index2].pageTable[index3]|page offset



Comparing page table size

- Hypothesis:
  - 32 bit address;
  - logical and physical pages of 4K (→ 12 bit for the offset)
  - Page table entry of 4 byte
    - 3 bytes for block address
    - 1 byte for the flags
- Page table size one single page level:
  - Number of table entries: 2^{20}
  - Space in bytes: $2^{20} * 4 = 2^{22}$ (4MB)
- Max physical memory size:
  - 3 bytes for the block address means 2^{24} page frames each storing 4K bytes → 64GB

Comparing page table size (cont.)

- Page table size two-level paging (same hypothesis):
  - Of the 20 bits, let's use 10 bits for index1 and 10 bits for index2
  - We have 2^{10} second-level tables
  - First and second level page table size is $2^{10} * 4 = 2^{12}$ (4K)
  - Best case: only 1 table in the second level → size = 8 K
  - Worst case: all tables in the second level → size = 4K + 4M
  - In most cases, 4K + n*4K where n is the number of second-level tables we need ($0 \leq n \leq 2^{10}$)
- Max physical memory size:
  - 3 bytes for the block address means 2^{24} pages → 64GB

Pro:

- Allochiamo in memoria solo le tabelle che ci servono.
- Possiamo fare tutto quello che facevamo prima.
- Controlli per i permessi di pagina.

Contro:

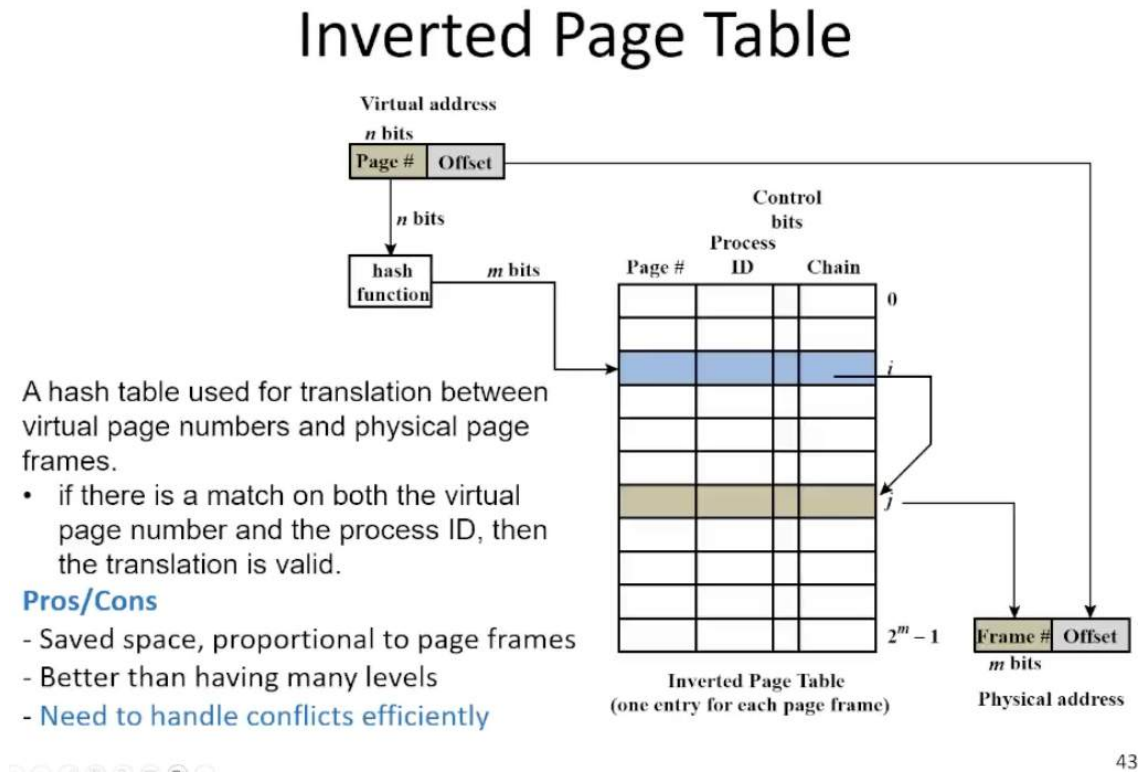
- lookup in memoria + accessi in memoria => **costoso**.
- Assurdo che devo fare più accessi in memoria per tradurre un indirizzo logico in fisico
- C'è un ulteriore livello che ci permette di risolvere questo problema.

Portabilità

Le tecniche viste in precedenza ci dicono come può essere implementata la traduzione degli indirizzi a livello HW. Anche il SO ha bisogno della traduzione degli indirizzi ad esempio per le OP di context switch ecc... Solo che non vogliamo che il SO funzioni solo per un certo tipo di architetture, per questo vogliamo un sistema SW per la gestione degli indirizzi virtuali che funzioni per qualsiasi tipo di architettura.

Inverted Page Table (Software)

Il SO usa una **core map** che permette tra le altre cose di tradurre da fisico a logico e da logico a fisico. Questa core map usa un certo numero N di bit dell'indirizzo virtuale. Dato l'indice di pagine ci effettua il calcolo di una funzione hash che indirizzerà a una entry della Page Table.



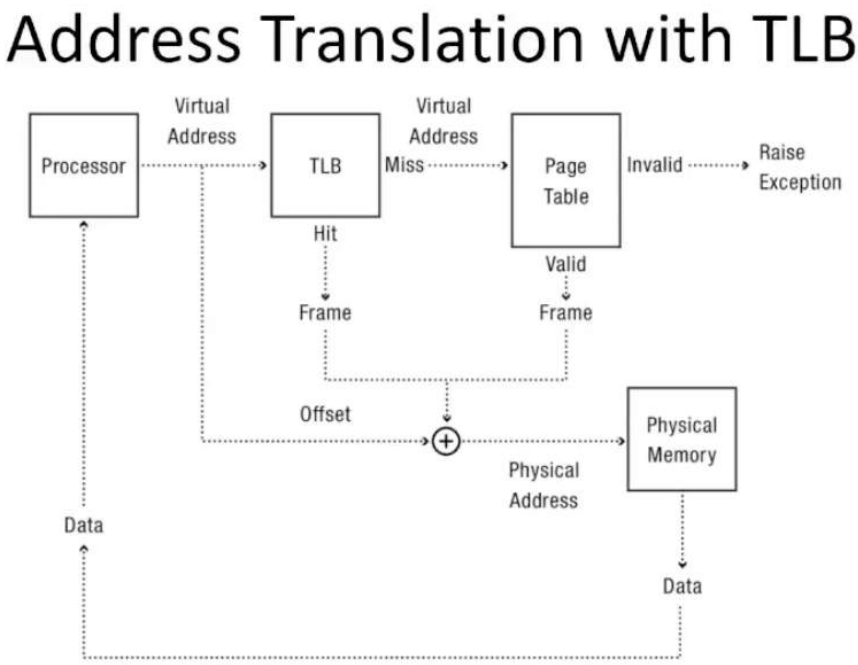
Non possiamo implementare una tabella hash ad HW poiché è troppo complicato, quindi viene implementata a **software**. Non possiamo neanche fare tutto a livello SW dato che dovremmo effettuare in continuazione chiamate di sistema che ridurrebbero drasticamente le prestazioni.

TLB (Translation Lookaside Buffer)

Problema: dobbiamo rendere più efficiente la traduzione degli indirizzi tramite page table, evitando di ripetere troppi lookup in memoria per consultare la Page Table. Per questo **adottiamo una cache,** che si chiama **TLB,** per memorizzare le traduzioni già fatte sfruttando il principio di **località spaziale e temporale**. Questa è una cache **pienamente associativa**, con b=1, abbastanza piccola in modo da evitare conflitti.

Accedo alla cache con un indirizzo logico e spero che la traduzione sia già memorizzata. Nel caso di cache miss comunque dopo carico anche le pagine vicine e dopo mi aspetto di fare cache hit.
Questa cache risiede all'interno dell'MMU.

Ogni entry è una coppia (logico, fisico).
Il miss-rate è molto basso quindi vale la pena costruire la cache.
Possiamo avere + di una TLB per ogni core.

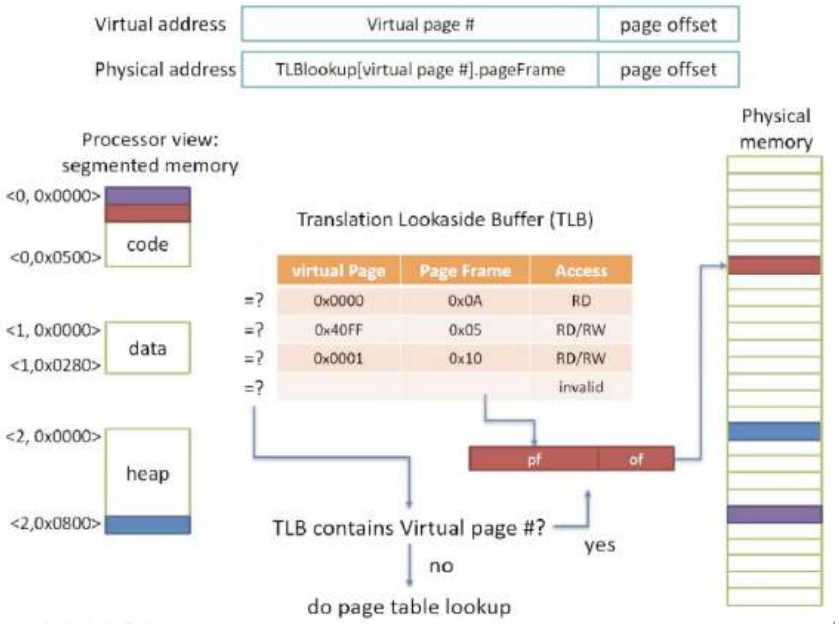


Costo è della traduzione degli indirizzi è l'**AMAT** relativo al TLB.
T = TLB-hit + TLB-miss * CostFULLTranslation.

In caso di context switch potrei avere che due processi diversi indirizzano allo stesso indirizzo logico che dovrei tradurre in due indirizzi fisici differenti. Il problema è che se la traduzione viene memorizzata nel TLB potrei erroneamente usare la traduzione già fatta per il vecchio processo e andare all'indirizzo di memoria sbagliato. Per evitare questo tipo di problema posso usare uno schema **TLB tagged** ovvero dove ogni entry del TLB ha un **campo PID** che ci permette di confrontare anche il processo corrente con le traduzioni fatte.

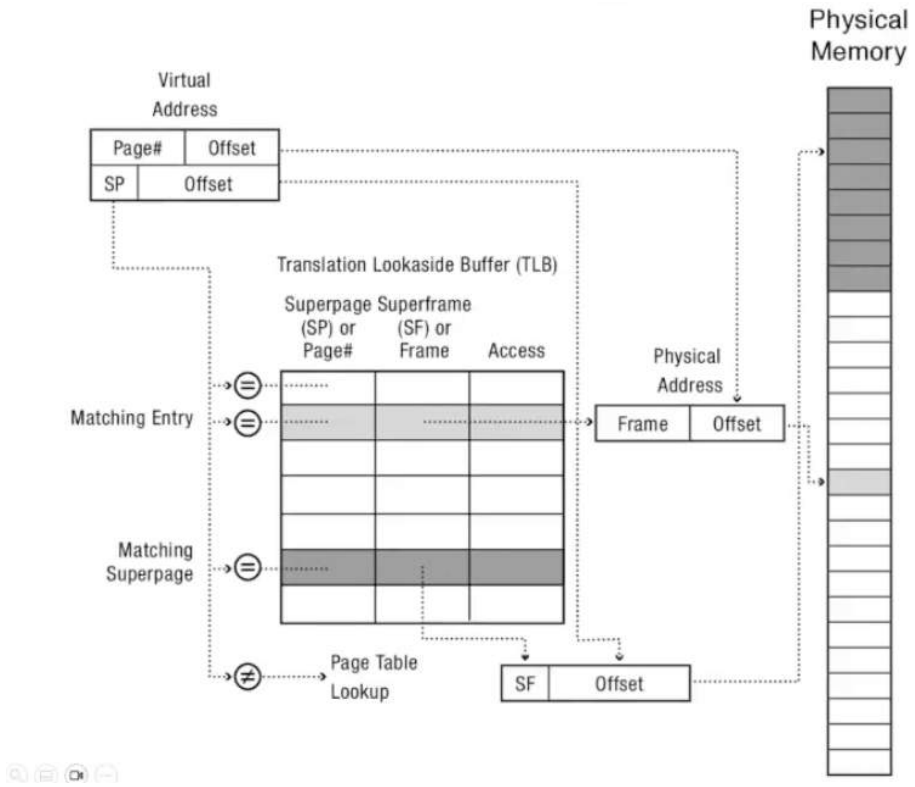
Su multicore abbiamo il fenomeno **TLB shootdown**. Il SO deve quindi aggiornare i dati su tutti i processori per la coerenza dei dati. La coerenza funziona automaticamente ad HW e deve funzionare anche per tutte le TLB. Quando aggiorniamo la tabella delle pagine, ad esempio aggiorno il page frame (indirizzo fisico) di una certa pagina, la traduzione fatta dalla TLB diventa invalida e deve essere fatta.

Il fatto di avere un TLB ci dà un guadagno, ma ci aggiunge anche degli overhead di gestione.



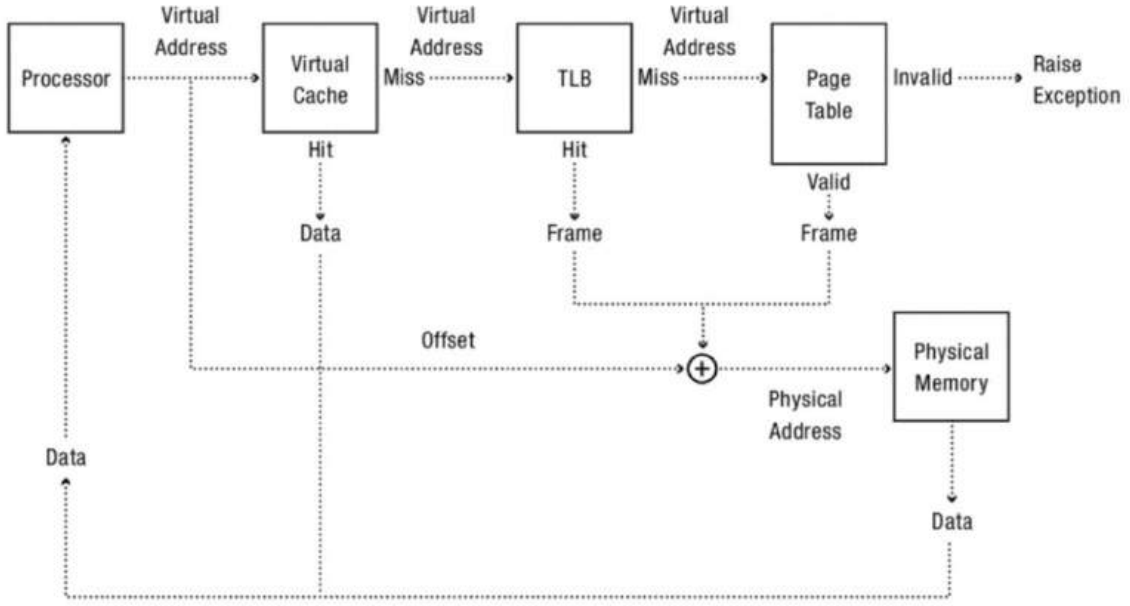
Superpagine

Usare una superpagina che occupa tanta memoria che raggruppa al suo interno le pagine normali, di modo da effettuare meno traduzioni. Avrò però un Offset più grande.
Super pagine usate per diminuire i cache miss. Devo poter discriminare le super pagine dalle pagine normali. In questo modo posso raggruppare le entry del TLB risparmiando spazio. Questo può avvenire quando su richiesta il SO comunica con l'MMU e il TLB che una certa pagina è una super pagina che raggruppa tante piccole pagine.

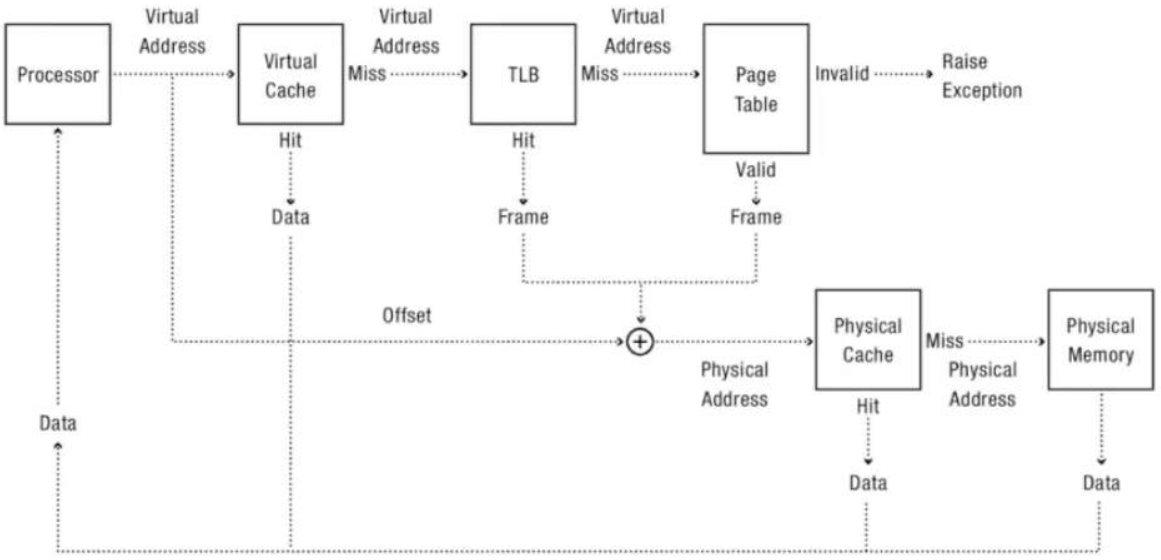


Esistono due tipi diversi di schemi con TLB una con una cache virtuale e una con una cache fisica.

Cache con indirizzi virtuali
In questo caso la cache fa un associazione al dato con l'indirizzo virtuale questo performa meglio dato che non devo aspettare la traduzione dell'indirizzo, devo però stare attento che più processo non abbiamo lo stesso indirizzamento logico per questo salvo anche l'id del processo per ogni entry.



Cache con indirizzi fisici
In questo caso la cache verrà consultata solo dopo aver effettuato la traduzione dell'indirizzo questo ci porta a una semplificazione della gestione della cache ma bisogna sempre tradurre.



Demand-Paged Virtual Memory and Page Caching

Demand-Paged Virtual Memory and Page Caching

Virtual memory

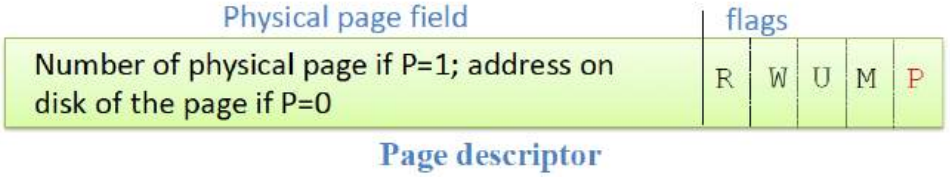
Virtual memory è una separazione tra la memoria logica dell'utente e quella fisica. Questo livello di castrazione garantisce che il SO dia l'illusione ai processi di avere più memoria rispetto a quella che realmente hanno (cioè quella fisica).

- flessibilità
- accesso controllato (lettura scrittura)

Questa astrazione viene implementata **considerando la memoria principale come una cache per il disco**.

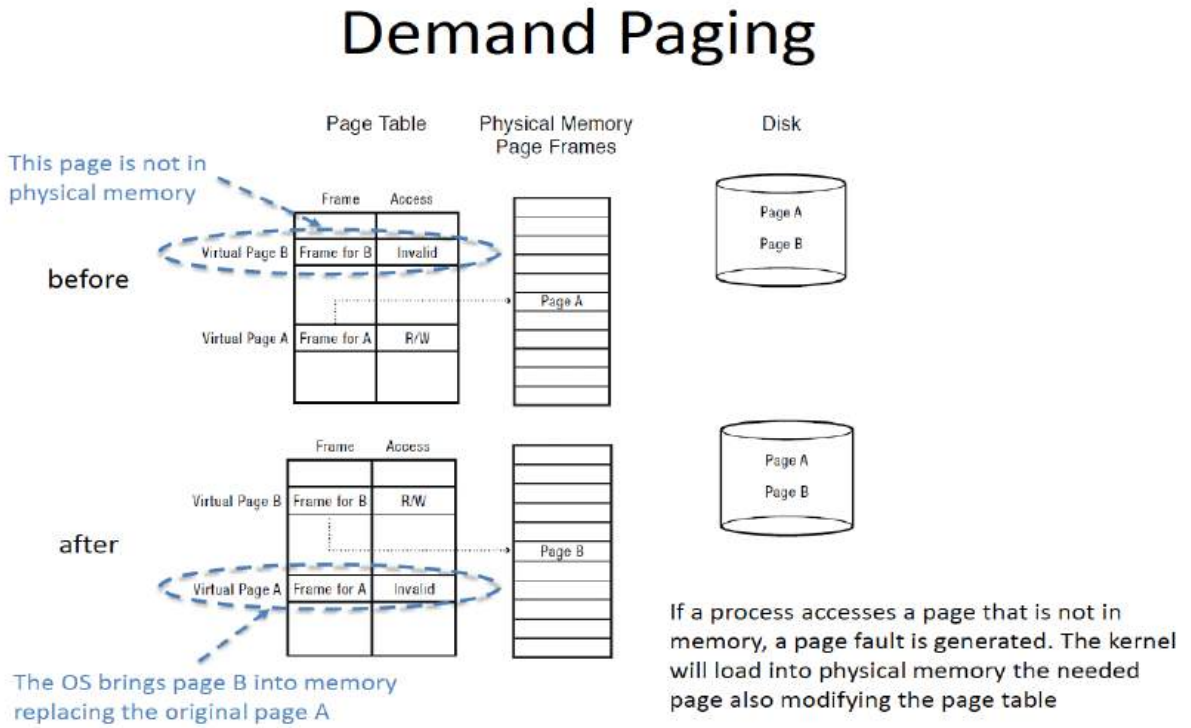
Quindi nelle entry delle nostre page table avremo anche un campo, un bit, che ci indica se quella pagina è valida o non è valida. Le pagine non valide sono quelle che non risiedono in memoria e che quindi devono essere caricate dal disco mentre quelle valide sono quelle presenti in memoria. Quando la pagina non è caricata in RAM, nella page table non è presente alcun indirizzo fisico, viene mandata una eccezione al kernel che tramite ALTRE STRUTTURE sa dove andare a cercare quella pagina sul disco.

Quando una pagina non valida deve essere caricata in memoria si dice che viene caricata **on-demand,** cioè su richiesta da parte del SO. Questo processo è chiamato **demand paging**.

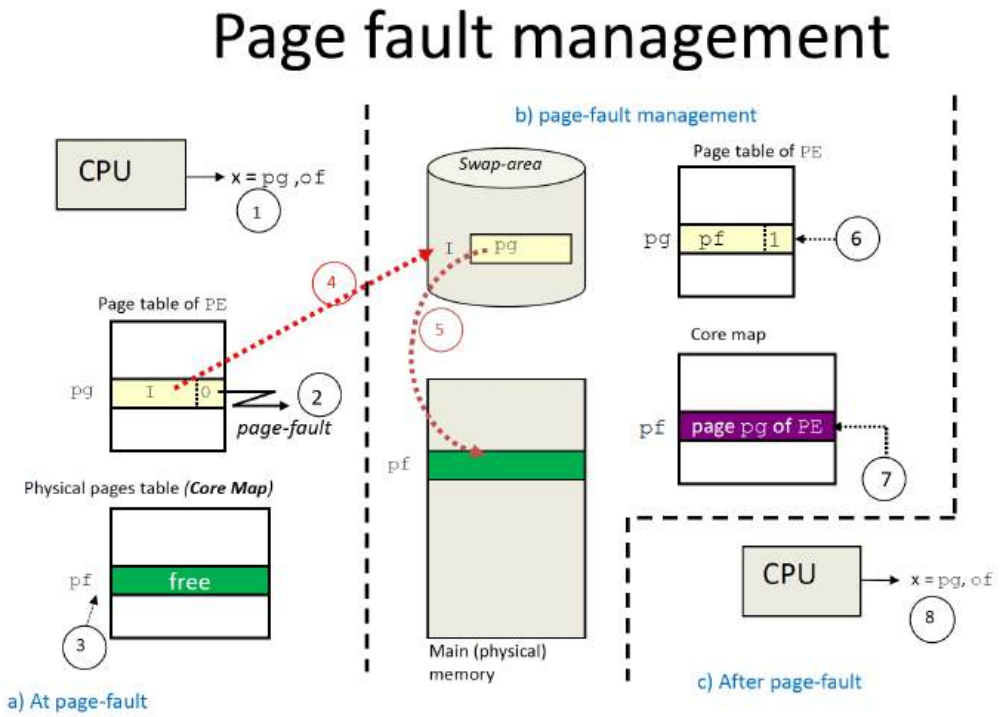


Page descriptor

R,**W** => read/write access
M,**U** => modified/use bits (per il replacement algorithms)
P => presence bit (se 0 -> page non presente in memoria, 1 -> presente in memoria)



In questa immagine possiamo vedere come avviene il processo di validazione e invalidazione delle page nella page table. Quando viene effettuata una richiesta per una pagina non valida viene caricata dal disco. Se la RAM è piena viene prima eliminata una page dalla memoria. Questo processo di eliminazione viene fatto con un criterio, ma lo vedremo dopo.



Per leggere l'immagine segui l'ordine dei numeri!!!

La **swap area** è uno spazio su disco (generalmente una partizione dedicata o un file speciale) che il sistema operativo utilizza come estensione della memoria principale per memorizzare temporaneamente le pagine di processo che vengono “espulse” dalla RAM.

La **Core Map** è la tabella che tiene traccia di **chi sta usando ogni pagina fisica** della RAM, se è libera, condivisa, modificata o meno.

Quando la pagina non risulta valida viene mandata una eccezione che viene gestita dal SO. Il page fault viene mandato quando nella nostra page table viene visualizzata una pagina che non è valida, nella core map corrisponderà uno spazio vuoto. Per la gestione del fault viene caricata dal disco (dalla swap area) la pagina che viene poi messa nella memoria principale. A quel punto viene validata la pagina all'interno della page table. Mentre nella core map vengono inserite le informazioni relative alla pagina. A questo punto l'eccezione è stata gestita correttamente e possiamo rimandare la richiesta della pagina dalla CPU.

Se ci fosse il TLB non cambia molto possiamo descrivere i passaggi logici:

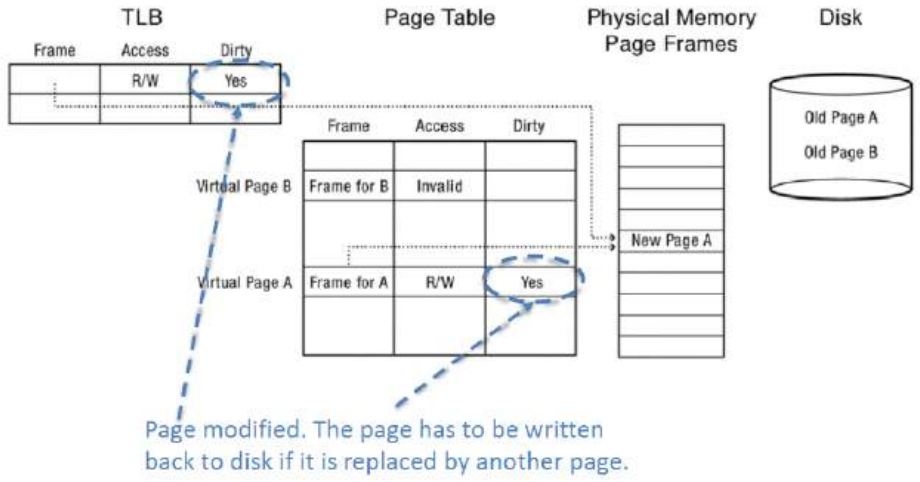
1. TLB miss
2. Page table walk
3. Page fault (page invalid in page table)
4. Trap to kernel
5. Convert address to file + offset
6. Allocate page frame
   - Evict page if needed
7. Initiate disk block read into page frame
8. Disk interrupt when DMA complete
9. Mark page as valid
10. Resume process at faulting instruction
11. TLB miss
12. Page table walk to fetch translation

Nella core map, la tabella fisica delle pagine, potrebbe non esserci spazio quindi potremmo doverlo liberare. Dobbiamo selezionare una **pagina vittima** che andrà tolta. Possibilmente una che non sia stata modificata, perché se lo fosse dovrei riscrivere sul disco e il SO spende più tempo.
Quando devo selezionare una pagina vittima mi conviene prendere una pagina che non uso. Una volta selezionata devo invalidare la pagina in tutti i TLB di tutti core contenenti eventuali riferimenti a quella pagina.

Come faccio a sapere se una pagina è stata modificata?
Esiste un bit che ci dice se la pagina è stata modifcata, e un bit di uso che ci dice se la pagina è attualmente in uso.

Questi Flag possono essere settati ad HW. Queste entry possono essere modificati dall HW, come il bit di presenza che viene gestito interamente dall'HW.

Tenere traccia delle modifiche delle pagine



Basta che almeno un processo che condivide la pagina l'abbia modificato affinché il bit vada settato a modificato. Non tutte le architetture possono tenere traccia delle modifiche allora alcuni sistemi cercano di emulare quei bit.

Se questo fenomeno di scaricare e caricare (**swap out**) viene effettuato troppo spesso si va incontro al fenomeno del **trashing**. Quindi dobbiamo trovare un modo per evitare questo fenomeno. Se la memoria fisica è troppo piccola rispetto a quella che mi serve, entro in questo fenomeno quindi devo trovare degli **algoritmi che selezionino le pagine giuste da eliminare**.

A simple policy

Random => rimpiazziamo una random page
FIFO => rimpiazzo con la prima che è stata caricata in memoria.

Cosa può andare storto con queste politiche?

FIFO in Action

reference	A	B	C	D	E	A	B	C	D	E	A	B	C	D	E
1	A				E				D				C		
2		B				A				E				D	
3			C				B				A				E
4				D				C				B			

Worst case for FIFO is when program strides through memory
that is larger than the main memory
- working set larger than cache capacity

Problema : non tiene conto della località degli accessi. Produce un sacco di fault di memoria.

MIN (ideal, optimal)
Questo algoritmo prevede di conoscere il futuro cioè elimino quello che userò tra più tempo.
Impossibile da implementare.

LRU (Least Recently Used) approssima MIN
Elimina la pagina che ho utilizzato meno recentemente. Troppo costoso da implementare.

NRU (Not Recently Used) approssima LRU
Elimina una tra quelle che non ho usato recentemente.

LRU/MIN for Sequential Scan

LRU

reference	A	B	C	D	E	A	B	C	D	E	A	B	C	D	E
1	A				E				D				C		
2		B				A				E				D	
3			C				B				A				E
4				D				C				B			

MIN

reference	A	B	C	D	E	A	B	C	D	E	A	B	C	D	E
1	A					+					+				
2		B					+					+	C		
3			C					+	D					+	
4				D	E					+					+

Nel caso ci sia località, si vede che MIN produce meno miss in assoluto, FIFO è una via di mezzo mentre LRU è simile a MIN. Mentre FIFO in presenza di località non minimizza il numero di miss.

LRU

reference	A	B	A	C	B	D	A	D	E	D	A	E	B	A	C
1	A		+				+				+			+	
2		B			+								+		
3				C					E			+			
4						D		+		+					C

FIFO

reference	A	B	A	C	B	D	A	D	E	D	A	E	B	A	C
1	A		+				+		E			+			
2		B			+						A			+	
3				C									B		
4						D		+		+					C

MIN

reference	A	B	A	C	B	D	A	D	E	D	A	E	B	A	C
1	A		+				+				+			+	
2		B			+								+		
3				C					E			+			
4						D		+		+					C

Anomalia di Belady

Per FIFO uno potrebbe pensare che aumentando il numero di *page frame* (numero di pagine in memoria) si potrebbe ridurre i fault, ma in realtà non è vero. Nel caso di algoritmi come FIFO potrebbero addirittura aumentare.

Belady's Anomaly (Cont.)

FIFO (3 slots)

reference	A	B	C	D	A	B	E	A	B	C	D	E
1	A			D			E					+
2		B			A			+		C		
3			C			B			+		D	

9 page faults Miss Rate= 9/12

FIFO (4 slots)

reference	A	B	C	D	A	B	E	A	B	C	D	E
1	A				+		E				D	
2		B				+		A				E
3			C						B			
4				D						C		

10 page faults Miss Rate= 10/12

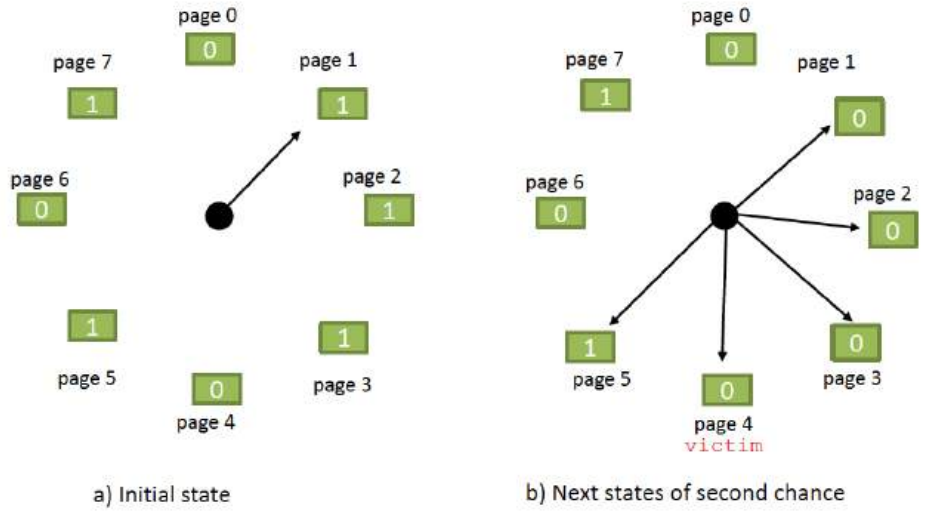
Quindi FIFO fa schifo. Dobbiamo usare un'approssimazione: LRU.

Algoritmi LRU

Second chance
Tutte le pagine vengono organizzate in una lista, con in testa la più vecchia. Quando ho bisogno di una pagina vedo la testa della lista e se ha il bit a 0, cioè non l'ho riferita di recente, allora la posso eliminare. Altrimenti se è ad 1 lo metto ad 0 e la metto in coda. Questo processo va avanti finchè non si trova una pagina con bit riferito a 0.
Problema: troppi movimenti in questa lista.

Negli esercizi non per forza ordini la lista dal meno recente.

Clock algorithm
Scorro una **lista circolare** e ogni volta che consulto la lista **riparto da dove ero arrivato la volta precedente**. Per il resto il funzionamento è uguale.



Variante di questo algoritmo: **N-chance**
Invece che avere un solo bit, le pagine hanno più bit: quando vedo che non è stata usata incremento il bit. Finchè la pagina non è arrivata ad un certo valore non la elimino, dando più chance alla stessa pagina.
Si tratta di una generalizzazione del clock algorithm.

Quando vengono usati questi algoritmi di rimpiazzamento ?

- Quando ho bisogno di selezionare una pagina da eliminare, in modo sincrono quindi devo aspettare.

- In maniera asincrona, mandando un **demone** (thread che esegue in background) che in continuazione tiene traccia di quale pagina deve essere eliminata.

Prendo le pagine di tutte le pagine dei processi o del singolo processo (globale o locale?)

- **Contro globale**: se ci fossero processi che riferiscono le pagine poco frequentemente (dispositivo I/O) vengono messi spesso in attesa, e le pagine di quel processo vengono scelte più spesso come vittime.
- **Contro locale**: potrei selezionare una pagina da me processo poco riferita, ma da qualcun altro molto riferita.

T: time of last reference

a)

	T
A0	10
A1	7
A2	5
B0	9
B1	6
C0	12
C1	4
C2	3

b)

	T
A0	10
A1	7
A2	**5**
B0	9
B1	6
C0	12
C1	4
C2	3

c)

	T
A0	10
A1	7
A2	5
B0	9
B1	6
C0	12
C1	4
C2	**3**

a) Initial configuration, the process A2 needs a page frame
b) Page replacement with a local policy (WS, LRU, SC)
c) Page replacement with a global policy (LRU, SC)

- With a global replacement algorithm, a process cannot control its own page fault rate
- With a local replacement algorithm, a process does not have the opportunity of using other less used frames
- With a global strategy, throughput is usually higher, and is commonly used

Quella globale è meglio. E lo faccio in un sottoinsieme di pagine attraverso una core map.
Un approccio diverso è quello di working set.

Working Set Model

Per la paginazione abbiamo che un WS = pagine riferite in un periodo t. Per un processo, un WS cresce e dopo un po' nel tempo si stabilizza. Questi algoritmi tentano di approssimare tutti i comportamenti che possono avere i processi. Scegliere il giusto lasso di tempo T è difficile.

Il WS model è intrinsecamente di tipo locale con le sue problematiche. Ogni processo ha il suo WS. Viene reso globale considerando il Resident Set.

Resident Set = insieme delle pagine che sono attualmente in memoria. Approssima il Working Set globale del processo anche se non c'è una stretta relazione. Così si può far lavorare l'algoritmo del working set in maniera **globale**. Spesso l'algoritmo viene applicato a questo Resident Set.

Come funziona l'algoritmo del WS?

T = lasso di tempo
R = 1 se la pagina è stata riferita nel lasso di tempo t, 0 altrimenti
TLR (Time of the Last Reference) = tempo di ultimo riferimento
Age = tempo trascorso dall'ultima referenza [*current_time* - TLR]

Se R = 0:
Si setta l'age = Current Time - Time Last Reference.
Se R = 1:
TLR = current time, age = 0 and R = 0

Andiamo a considerare tutte le pagine che hanno Age < T, quelle non le posso eliminare, perchè sono quelle riferite nell'ultimo lasso di tempo T.
Posso eliminare le altre che sono relativamente vecchie, cioè che non riferisco da tanto tempo.

Page table

age	R	...
2084	1	
2003	0	
1980	1	
1213	0	
2014	1	
2020	1	
1604	0	

[age: current_time – TLR]

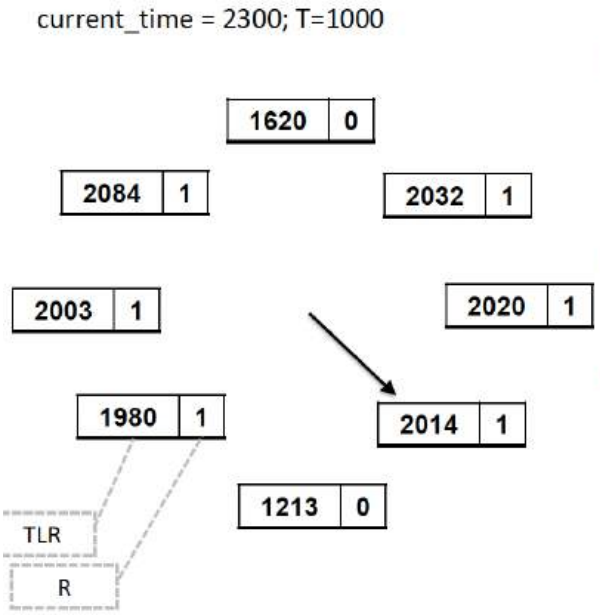
Current virtual time: 2204

```
For each page: {
  if (R==0)
     age = current_time – TLR;
  else if (R==1) {
     TLR= current_time; R=0; age=0;
  }
  if ( (age>T)
     removes the page
}
----------------------------------

if (age<=T for each page)
  removes the page with higher age
```

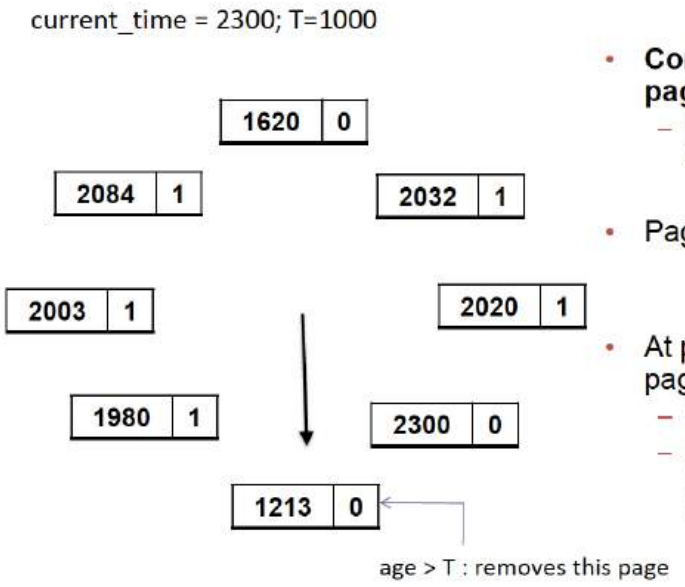
Problema: a scansionare tutti questi WS e settare tutti i parametri perdo tanto tempo, per questo dobbiamo trovare un'approssimazione.

Si considerano solo le pagine in main memory. La lista è una **lista circolare di pagine**, per ogni pagina tengo il bit **R** e l' **Age** (o il **TLR**). Tengo in memoria solo le pagine che hanno un'età minore di 1000.

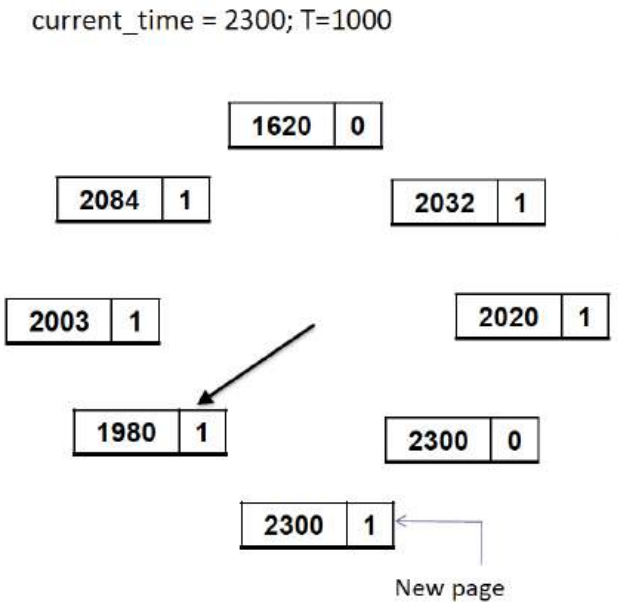
Fattore critico.
Se T è troppo grande, tengo in memoria troppe pagine. Se invece è troppo piccolo rischio di non approssimare bene il WS.



- **Considers only the pages in main memory**
  - More efficient than scanning the page table
  - It scans the core map
- Pages in a circular list
- At page fault looks for a page out of the WS
  - Better if not "dirty"
  - If selects a dirty page, the page is saved before its actual removal



- **Considers only the pages in main memory**
  - More efficient than scanning the page table
- Pages in a circular list
- At page fault looks for a page out of the WS
  - Better if not "dirty"
  - If selects a dirty page, the page is saved before its actual removal



- Considers only the pages in main memory
  - More efficient than scanning the page table
- Pages in a circular list
- At page fault looks for a page out of the WS
  - Better if not "dirty"
  - If selects a dirty page, the page is saved before its actual removal

Paging and WS algorithm

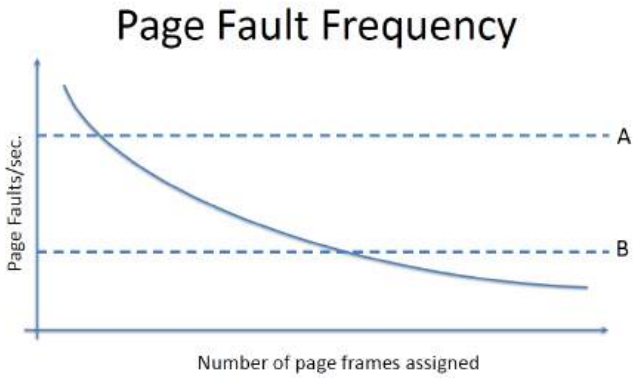
On-demand paging
Inizialmente nessuna pagina viene caricata per il processo, permettendo di far partire il processo istantaneamente. La page table viene caricata. Ogni volta che serve una pagina viene caricata su richiesta. All'inizio ci sono tanti fault.

Prepaging
Quando viene caricato un nuovo processo viene caricato il suo WS. Per questo abbiamo bisogno di predire il futuro e di ritardare l'inizio di un processo.

Quale dovrebbe essere il numero di pagine per processo?

SI segue un approccio dinamico -> **PFF** (Page Fault Frequency): algoritmo che tiene conto della frequenza di fault di un processo: se ha **troppi fault** deve avere **più pagine in memoria** mentre se ne ha **troppi pochi** significa che ha **troppe pagine in memoria**.

Page fault frequency
Cerca di trovare delle soglie per cui i fault sono troppo bassi e se sono troppo alti mantenendo pagine in memoria a seconda dei fault tenuti.



Cosa succede alle performance se abbiamo molti processi ?
Il sistema andrà facilmente in **trashing**. Crollo delle prestazioni da parte del SO alto, occorre fare tanti swap, il **throughput** cresce.

Quindi ti viene data una soglia di blocchi minima da avere libera. Ad ogni iterazione il demone esegue l'algoritmo Second Chance e scarica un tot di pagine che gli servono per rispettare le soglie previste.
Nel caso peggiore in cui non ho blocchi di memoria liberi devo effettuare lo **swapout di un processo** (SWAPPARE sul disco tutte le pagine di un set di processi, facendoli terminare e liberando spazio), con un criterio, ad esempio selezionare il primo in ordine alfabetico.
In gergo *"Far diminuire il grado di multiprogrammazione"* .

Sposto le pagine sul disco, ma dove finiscono ?
Code segment => code portion of executable
Data heap stack segments => swap area
Shared libraries => code file e temp data file

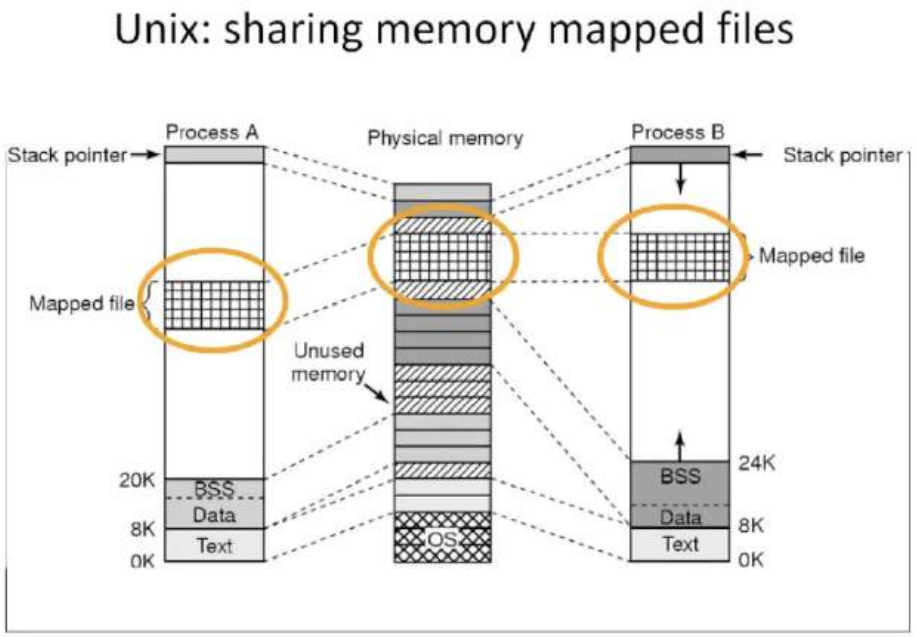
Il SO sa dove scaricare i dati quando deve mettere in pausa un processo e liberare la memoria.

ZIPF Model
Migliori per la gestione di server che devono caricare pagine web. Le pagine più accesse o più popolari sono messe in cima alla coda così ci si accede molto velocemente. Qui ha più senso parlare di popolarità che di WS.

UNIX
Organizzazione della memoria in Unix: Unix utilizza una combinazione di segmentazione e paginazione per la gestione della memoria virtuale. Questo significa che la memoria è

divisa in segmenti logici (come codice, dati, stack) e ulteriormente suddivisa in pagine di dimensioni fisse.
La memoria virtuale si basa sul **on-demand paging** (le pagine vengono caricate in memoria fisica solo quando sono necessarie).

Unix utilizza la **core map** per tenere traccia dell'allocazione dei blocchi fisici di memoria.



Modelli per I/O su File: I programmi possono eseguire operazioni di I/O su file in due modi principali:

Con le read/write tradizionali:
Il processo chiama read(fd, buf, n): il kernel copia n byte dal file in un buffer nel suo spazio utente.
L'applicazione fa le sue elaborazioni sul buffer.
Quando ha finito, invoca write(fd, buf, n): il kernel ricopia i dati dal buffer utente al file.

Con i file mappati in memoria (**mmap**):
Il processo apre il file e chiama *mmap()*, ottenendo un'area di memoria "alias" del file.
Ogni load/store su quell'area agisce direttamente sul file — senza copie esplicite intermedie.
Se accedi a una pagina non ancora caricata, scatta un page fault: il kernel preleva il blocco dal disco, lo mette in memoria, aggiorna la tabella delle pagine e riprende l'esecuzione.
Quando modifichi le pagine, queste vengono scritte sul file in background (dirty pages).

Vantaggi dei file mmap:

- **Semplicità di programmazione:** Soprattutto per file di grandi dimensioni, è più semplice operare direttamente sul file in memoria piuttosto che copiare dati da e verso il kernel.
- **Pipelining:** Un processo può iniziare a lavorare sui dati non appena alcune pagine del file sono state caricate in memoria.
- **Comunicazione** tra processi **efficiente**.

1) Core Map + Tabelle delle pagine

SO	SO	SO	SO	SO	SO		A,1 2	B,0 10	C,1 3		B,6 5	C,7 8		C,3 6	A,5 9	C,5 7	B,2 1	A,7 4
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18

Processo, pagina
Tempo ultimo riferimento
Blocco

Pagina	Blocco
0	-
1	7
2	-
3	-
4	-
5	15
6	-
7	18

Processo A

Pagina	Blocco
0	8
1	-
2	17
3	-
4	-
5	-
6	11
7	-

Processo B

Pagina	Blocco
0	-
1	9
2	-
3	14
4	-
5	16
6	-
7	12

Processo C

Tabelle delle pagine indicizzate dall'indice di pagina

- l'indice di pagina virtuale non è contenuto nella tabella

2) Core Map = Tabella delle pagine inversa

SO	SO	SO	SO	SO	SO		A,1 2	B,0 10	C,1 3		B,6 5	C,7 8		C,3 6	A,5 9	C,5 7	B,2 1	A,7 4
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18

Processo, pagina
Tempo ultimo riferimento
Blocco

Core Map indicizzata dall'indice di blocco fisico ottenuto applicando una funzione hash all'indirizzo logico

In entrambi i casi: il vettore circolare dell'algoritmo Clock algorithm of WSClock realizzato su *Core Map,* con i soli descrittori di blocchi assegnati ai processi

Algoritmo BSD

È una variante degli algoritmi visti, il sistema utilizza un deamon chiamato **page deamon** che utilizza 3 parametri: *lotsFree*, *minFree* e *desFree*.
MinFree sta per minime pagine libere, *DesFree* per desiderate e lotsFree per la quantità che abbiamo di pagine libere.

PageDaemon algorithm (**sketch**):

- ***if*** (*#freeblocks* ≥ *lotsfree*) return *//no operation required*
- ***if*** (*minfree* ≤ *#freeblocks* < *lotsfree*) ***or***
  (*#freeblocks* < *minfree* ***and***
  Average[*#freeblocks*, Δ*t*] ≥ *desfree*))
  replace pages until *#freeblocks* = *lotsfree* + *k* (with *k*>0)
- ***if*** ((*#freeblocks* < *minfree* ***and***
  Average[*#freeblocks*, Δ*t*] < *desfree*))
  swapout processes (all pages of a set of processes)

Quando non abbiamo il minimo dei blocchi liberi da tenere dobbiamo **eliminare** tutti i **processi necessari** per ristabilire il numero di blocchi liberi definito da *lotsFree*.

Su Windows

Windows segue un algoritmo di working set e non su approssimazione di LRU. L'algoritmo si basa su soglie di fault dinamiche quindi soglie che nel tempo cambiano. Se un certo processo supera la soglia di pagine allora verrà rimosso dal working set manager. Quali pagine saranno swappate? Quelle che non sono usate di recente con l'algoritmo del working set clock.
Windows usa due manager che sono due demoni:

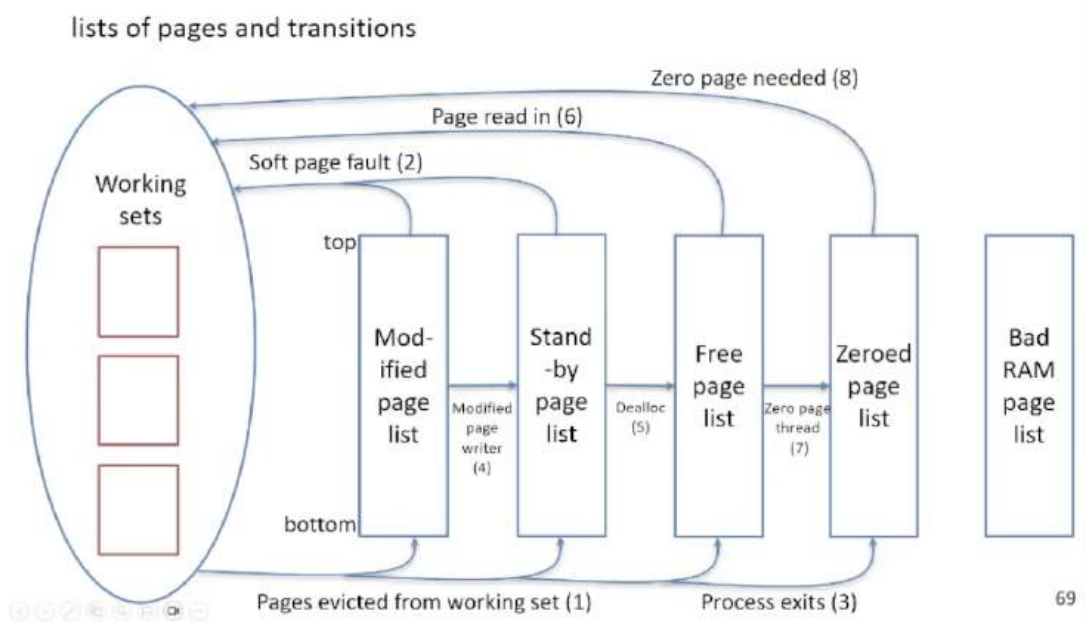
Balance set manager: scarica le pagine sul disco chiamando il ws manager
Working set manager: implementa l'algoritmo di rimpiazzamento

Windows: Working set manager
Scorre tutti i processi che hanno un valore di pagine sopra la soglia prefissata. Se R = 1 allora resetta R e il count mentre se R = 0 incrementa il contatore. Questo contatore dice da quanto tempo la pagina non è stata riferita.

A questo punto ordina le pagine sulla base del contatore. Seleziona x - max pagine da scaricare sul disco. Cercando in questo modo di rientrare nel range.

1. Fase 1 : aggiorno i contatori o li resetto.
2. Fase 2: elimino le pagine in ordine decrescente del contatore.

Windows: management of pages
Quando una pagina viene scaricata non viene realmente scaricata poiché è un'operazione che richiede tanto tempo. Non viene perciò scritta subito sul disco ma viene trasferita su una lista fino poi ad essere scritta sul disco. Non viene fatto immediatamente dato che una pagina potrebbe essere scaricata e dopo riusata.



File Systems

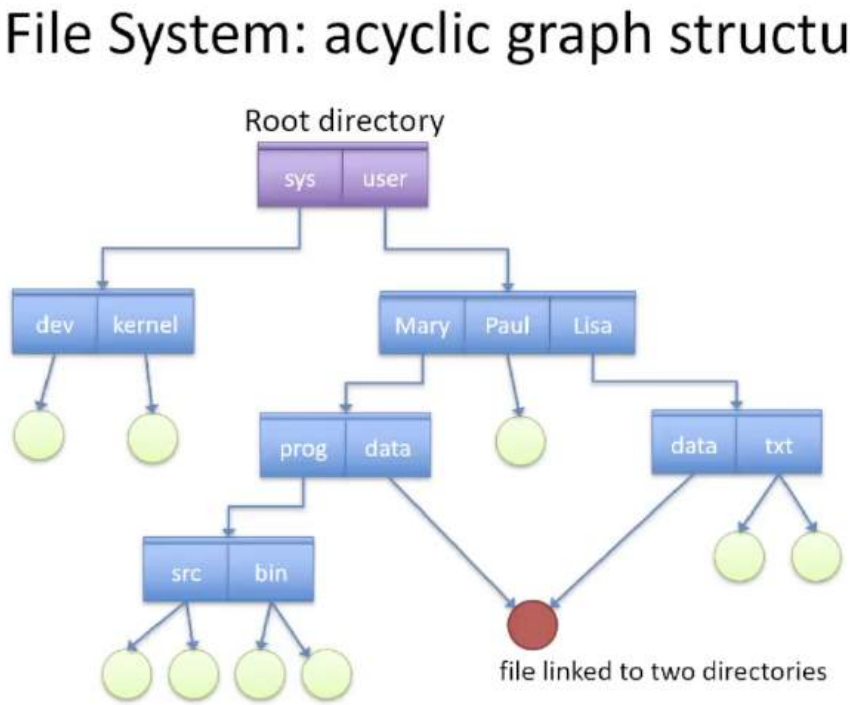
File Systems

Parte più vicina all'utente. Il file system è un astrazione del disco. Il ruolo del file system è quella di fornire un illusione per garantire la persistenza dei dati. Deve garantire alte prestazioni e politiche di caching di file, garantire protezione e privacy.

Il file system è una gerarchia di directory e sotto-directory. Quindi devo garantire che una certa directory e sottodirectory non siano viste da tutti. Deve mantenere una certa tolleranza ai guasti e mantenere salvi i dati.

Path = percorso relativo del path.
Path assoluto path che parte dalla radice.
Hard link
Soft link

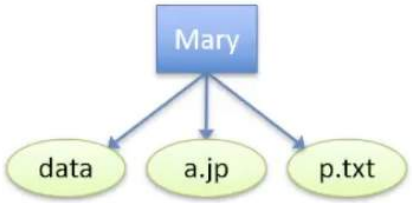
Mount point una directory che è usata per far partire un altro file system per estendere un file system e quindi usare un altro disco e usarlo sotto il mio file system. Lo posso fare anche con un server remoto.



File System Abstraction

Disk storage	FS abstractions for users
block oriented	byte oriented
physical sector	file name
no protection	file-based protection
if the system crashes, data data might be corrupted in some sectors	robust to system crashes

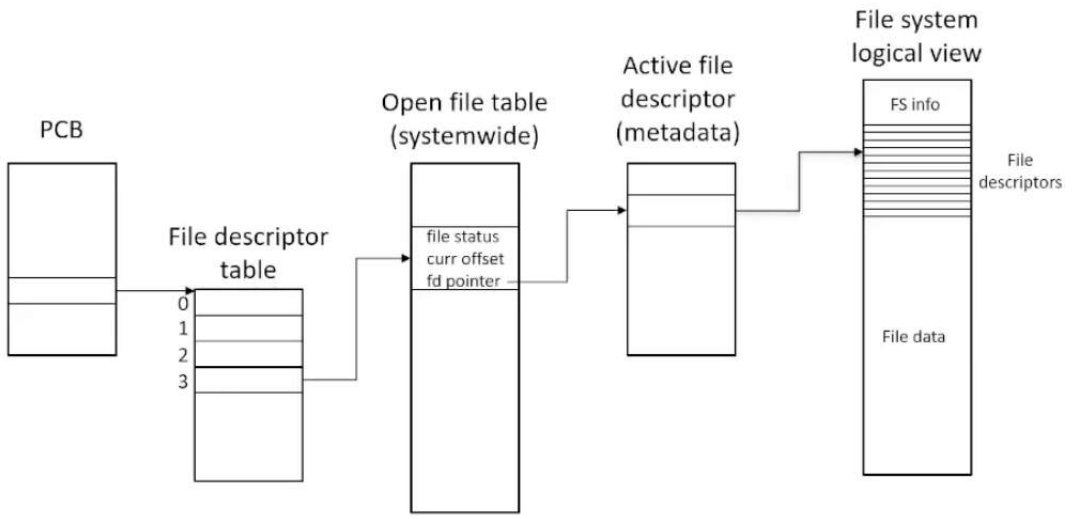
In una directory facciamo associazione nome file e attributi, sostanzialmente una directory è una tabella.



name	descriptor
data	Attribute: type, address, time, etc.
a.jp	Attribute: type, address, time, etc.
p.txt	Attribute: type, address, time, etc.

Implementation of directory Mary

Implementazione delle directory su Unix
Ogni entry della tabella ha il nome del file e il pointer al file descriptor.



Ogni processo ha sempre dei file descriptor aperti (input, output, error) e altri file che posso aprire durante l'esecuzione del processo la system wide tiene traccia di tutti i file aperti a livello globale, questo permette anche la shared memory e in questo modo più processi referenziano allo stesso file con indici diversi ma in questo modo possono comunicare. Vengono tenuti due offset di scrittura e lettura. Il file viene caricato in memoria e così lo posso leggere e scrivere. La ricerca di metadati avviene solo all'apertura e alla chiusura.

Accesso al file

- Sequenziale accedo ad una certa locazione e poi leggo in sequenza.
- Diretto sposto il cursore sul punto che voglio leggere o scrivere.

Accesso sequenziale ogni volta che viene fatta una operazione sul file sposto il cursore al successivo dato del record.
Accesso diretto: dico l'indice a cui voglio far riferimento e ogni volta che voglio fare un'operazione specifico l'offset su dove voglio operare.

UNIX File System API

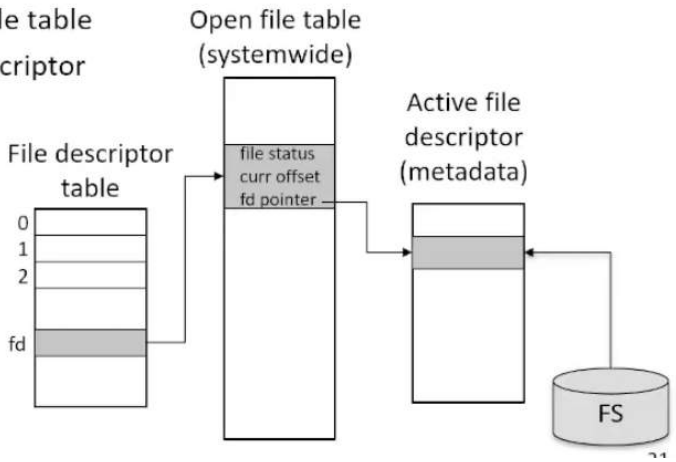
- create, link, unlink, mkdir, rmdir
  - Create file, link to file, remove link
  - Create directory, remove directory
- open, close, read, write, seek (mmap, munmap)
  - Open/close a file for reading/writing
  - Seek resets current position
- fsync
  - File modifications can be cached
  - fsync forces modifications to disk (like a memory barrier)

Fsync serve per rimuovere tutte le informazioni su un file in cache. La close implica una flush.

Open a File

- File name lookup and checks
- Copy the file descriptor into the in-memory data structure (if it is not in yet)
- Create an entry in the open file table
- Create an entry in the file descriptor table
- Return the index of the file descriptor table (fd)

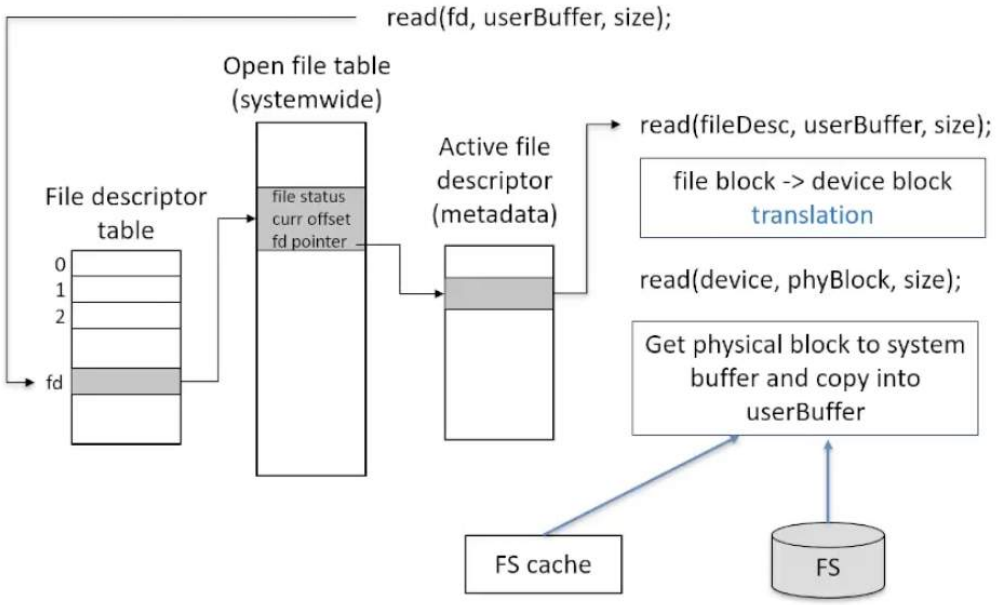
fd = open(filename, access);



User vs OS operations

La granularità minima del disco è un blocco. Quando il SO legge un file legge sempre un blocco non c'è un'unità più piccola. L'idea è che si lavora sempre con i blocchi che possono essere di centinaia di byte.

Read a block



Si può evitare la copia di file se mappo in una zona di memoria un file come quando apro file in condivisione per i file.

Workload del file system
Accediamo più spesso a file grandi o piccoli ?
Quindi il file system deve saper gestire in modo ottimale al caso più frequente.

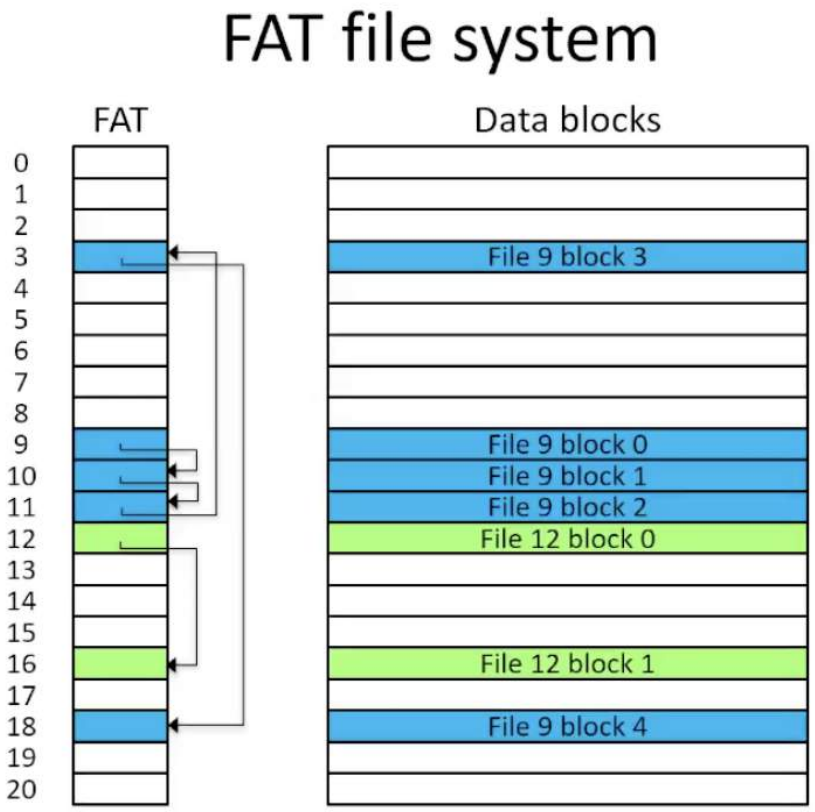
Data blocks e records
I dati sono organizzati in blocchi (blocco size >> record size) mentre quando vengono presi i dati sono organizzati in record. (In unix un record è un singolo byte).

File System Design Options

	FAT	FFS	NTFS
Index structure	Linked list	Tree (fixed, asym.)	Tree (dynamic)
granularity	block	block	extent
free space allocation	FAT array	Bitmap (fixed location)	Bitmap (file)
Locality	defragmentation	Block groups + reserve space	Extents Best fit defrag

FAT (File Allocation Table)

Struttura a lista linkata quindi un punto di ingresso è dato dalla directory del file. Dove ogni posizione del vettore rappresenta un blocco del disco.



I blocchi sono di dimensione fissa quindi non ho la frammentazione esterna. In questo caso però ho il problema della sparsità dei blocchi. Quindi per deframmentazione intendiamo

rimettere a posto i blocchi. **L'array della FAT deve essere caricato in memoria** quindi la dimensione di questo array non può essere troppo grande.

Pro:

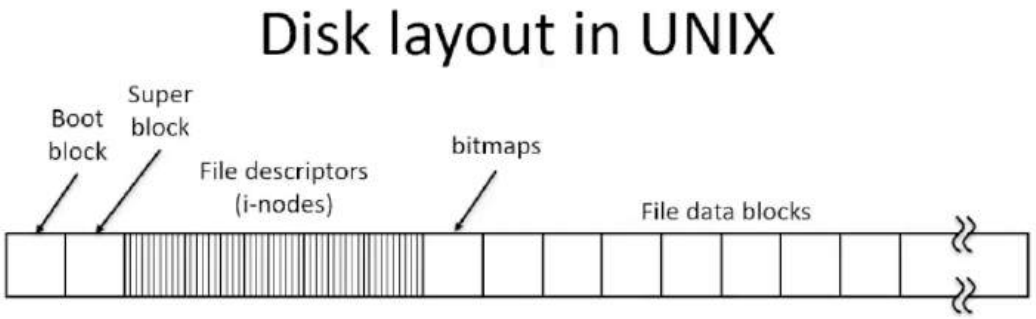
1. Semplice trovare un blocco libero
2. Facile appende un file
3. Facile eliminare un file

Contro:

1. FAT size:
    a. Deve essere caricato in memoria
    b. Limitazione alla grandezza del file system
2. Accesso diretto in lettura o scrittura non è ottimo dato che devo scorrere la lista.
3. Metadati limitati e senza protezione
4. Accesso randomico è lento
5. Problema di frammentazione.

FFS (Fast File System)

- Struttura dati: una lista di nodi. Ogni file ha il suo i-node, viene caricato in memoria solo l'i-node attivo.
- La i-node table **non viene caricata in RAM**. Vengono caricati solo gli i-node necessari per il funzionamento del file system e quelli dei file che andiamo ad utilizzare. In questo modo risolviamo il problema della FAT che ha un grandezza limitata, dato che la i-node table non va in memoria.
- Un i-node che ha dimensione fissa contiene:
    - Metadata: proprietario del file, permessi, accessi, file size, link count ecc.
    - Un set di puntatori ai blocchi di dati (struttura ad albero che permette di gestire meglio tutti i blocchi).



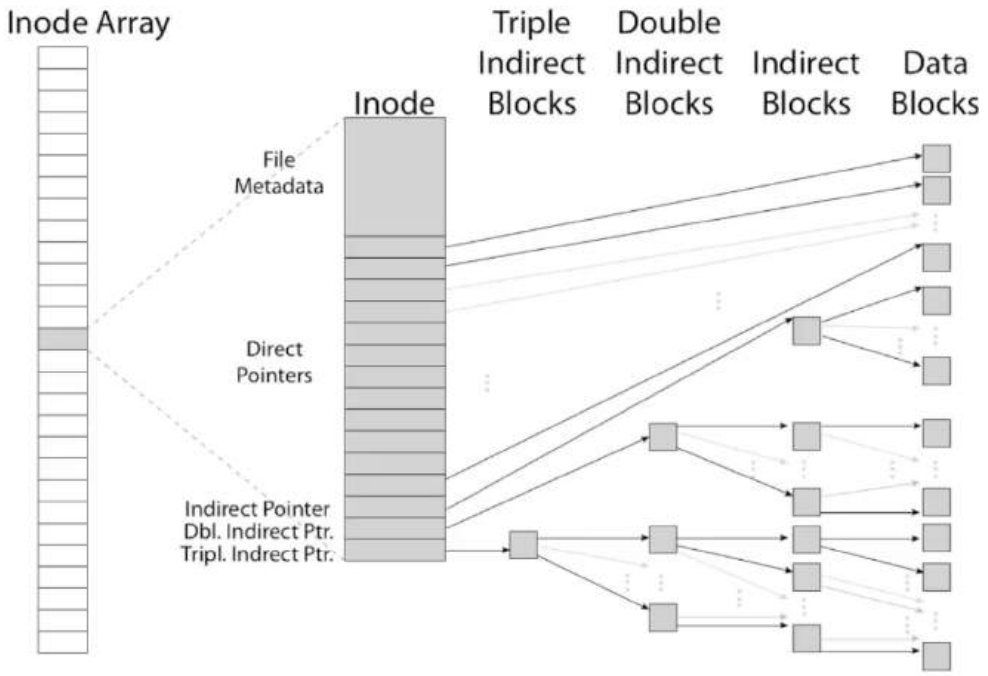
Super blocco composto dalle informazioni utili al FS per funzionare:

- Size del filesystem
- Size della i node area
- Pointers della bitmap
- Locazione degli i-node del
- la root directory

FFS inode

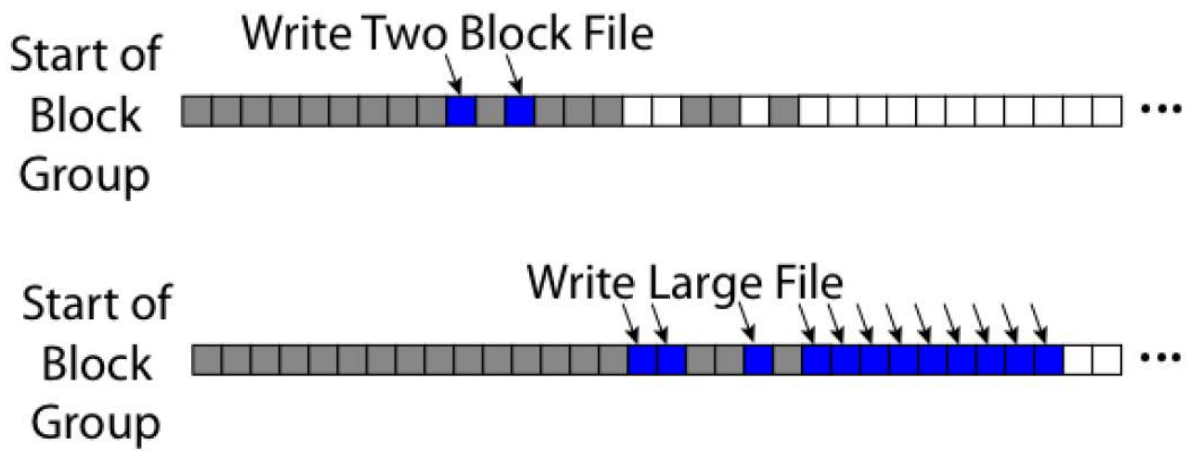
- Metadata
  - File owner, access permissions, access times, size, link count, i-node number, … (see 'man 2 stat' for more info)
- Set of 12 direct data pointers (e.g., 32-bit pointers)
  - With 4KB blocks => max size of 48KB
- 1 indirect pointer
  - pointer to disk block of data pointers
  - 4KB block size => 1K pointers to data blocks => 4MB
- 1 doubly indirect pointer
  - Doubly indirect block => 1K indirect blocks
  - 4GB (+ 4MB + 48KB)
- 1 triply indirect pointer
  - Triply indirect block => 1K doubly indirect blocks
  - 4TB (+ 4GB + 4MB + 48KB)

I 12 puntatori sono accessi diretti al blocco del disco, quindi sono in grado di indicizzare una massima dimensione del file data da 12 * dim. blocco, questo mi permette di ottimizzare l'accesso al disco per i file di piccole dimensioni. Se ho file più grandi devo usare un puntatore indiretto: esistono 3 livelli di indirezione singola, doppia e tripla.
Se il file ha dimensione minore di 48k per esempio riusciamo ad indirizzarlo tutto, altrimenti non mi bastano e dovrò utilizzare un puntatore a blocco indiretto. Avremo un puntatore (ad esempio il 13°) che punta ad un blocco che non contiene dati, ma che contiene puntatori a blocchi dati. E' una indirezione
semplice, conterrà *n* puntatori a seconda della dimensione del blocco, se il blocco era 4k e supponendo di avere puntatori da 4 byte (dimensione standard di un puntatore), ci saranno 1024 (4k / 4 Byte) puntatori a blocchi in questo blocco puntato. Avremo quindi 1024 nuovi blocchi dati ognuno da 4k. Con questi puntatori indiretti singoli riusciamo ad indicizzare file grandi fino a 4M+48k.
Se sono più grandi dobbiamo passare al puntatore indiretto doppio, abbiamo un puntatore che punta ad un blocco dati, dove ci sono 1024 puntatori a blocchi indiretti singoli, in cui ognuno di essi contiene 1024 puntatori a blocchi dati.
E così via fino ad arrivare al puntatore indiretto triplo, dove ho 3 livelli di indirezione, e alla fine ho blocchi dati.

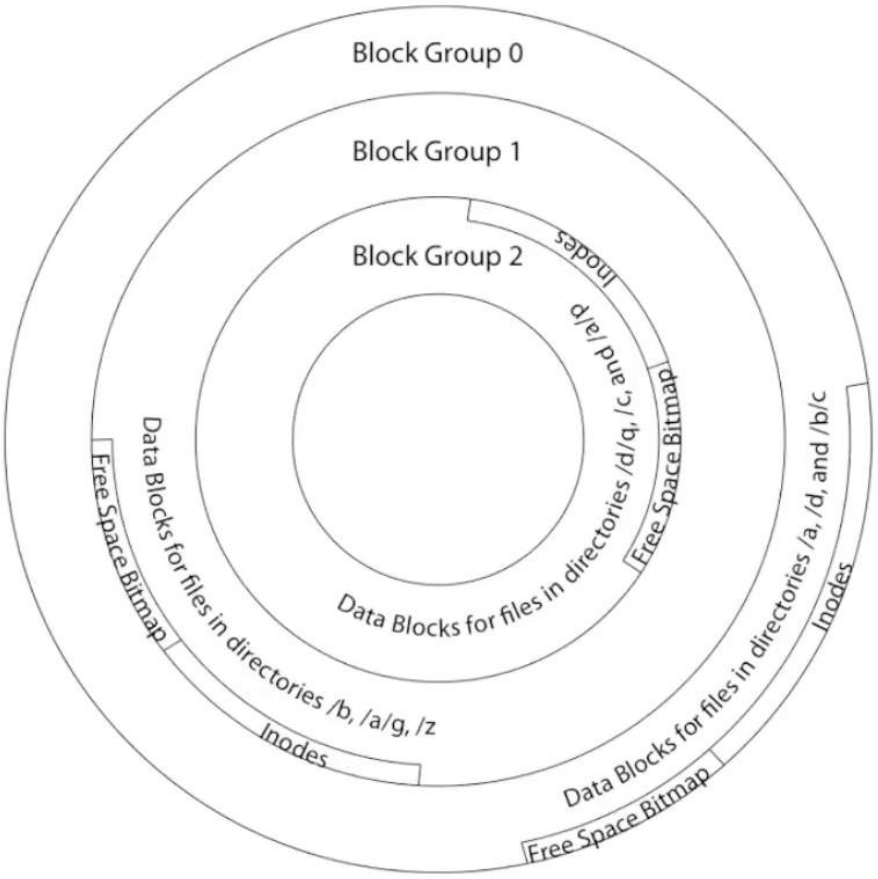


FFS asymmetric tree
Gestisce bene i file di piccole dimensioni ma non troppo piccoli per evitare frammentazione interna.

FFS località
Block group
Cerco di allocare i blocchi di uno stesso file vicini nel disco. Organizzando i blocchi in gruppi di blocchi. In questo modo si cerca di guadagnare località.
FFS usa la **first fit block allocation** cioè il primo blocco libero lo uso. Questo potrebbe portare alla frammentazioni di piccoli file ma non di quelli grandi che mantengono comunque una buona località spaziale.



Visione logica di un disco meccanico HDD.



Nei dischi SSD non ci sono problemi di località.

Pro

- Efficiente per file piccoli e grandi
- Buona località per metadati e dati
- Buona località per file di grandi e piccole dimensioni

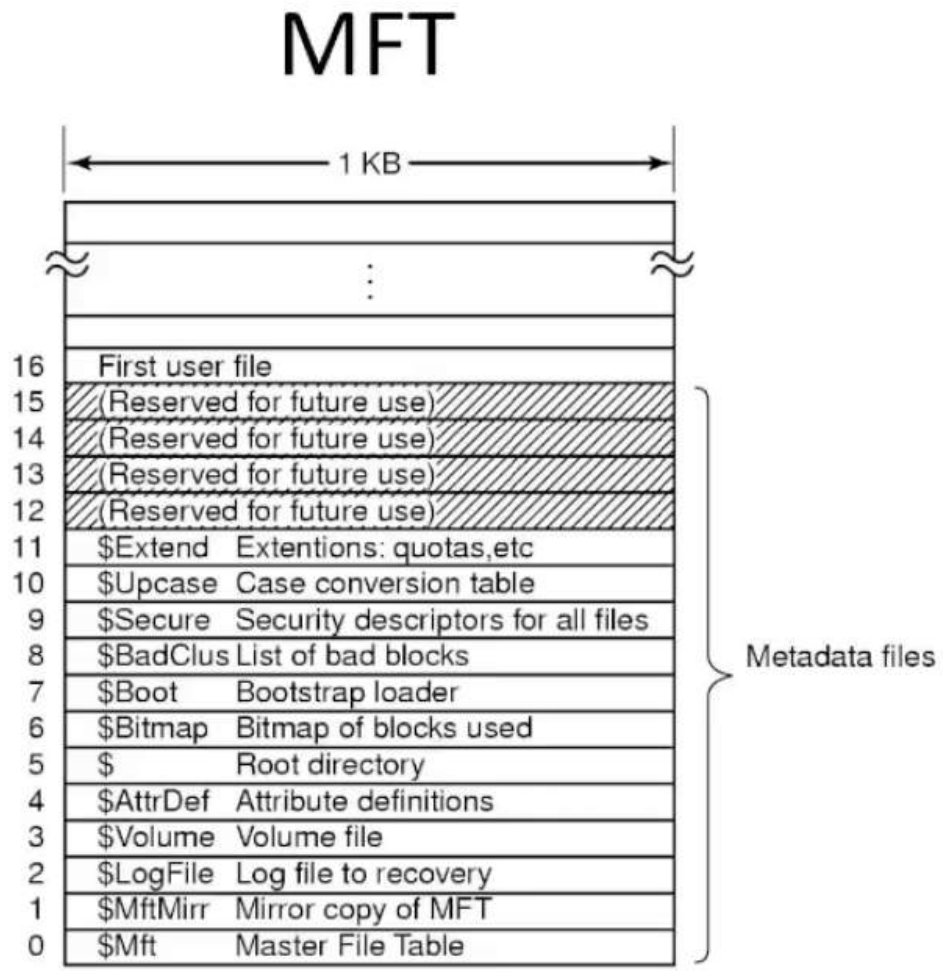
Contro

- Non è efficiente per file minuscoli a causa della rappresentazione con i-node.
- Inefficiente per codificare, quando il file è molto contiguo non sfrutto la cosa. Risolvo con NTFS
- Le tecniche di questo sistema funzionano bene per allocare gruppi di blocchi e per prevenire la frammentazione ma sprecano tra il 5-15% di memoria.

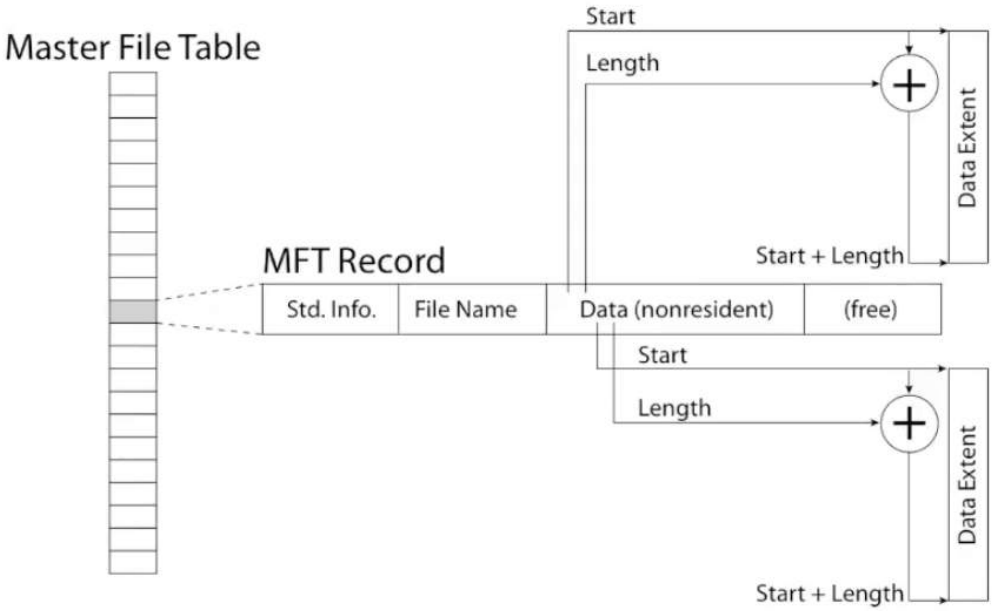
NTFS (New Technology File System)

Una sorta di evoluzione di FFS, questo sistema si basa su una struttura dati: **Master File Table**, cioè un file che memorizza dati e metadati in una singola entry, senza spreco di spazio o almeno molto marginale, dato che li potrei avere per file minuscoli (grandezza < 1KByte).
Si va in oltre a perdere il concetto di blocchi quindi di granularità a blocchi.

In questo tipo di file system non si avranno indirizzi logici dei blocchi che poi vengono mappati in indirizzi del blocco fisico, ma si avranno dei blocchi di lunghezza variabile: gli **extents.** La master file table **non risiede in RAM**, ma sul disco.

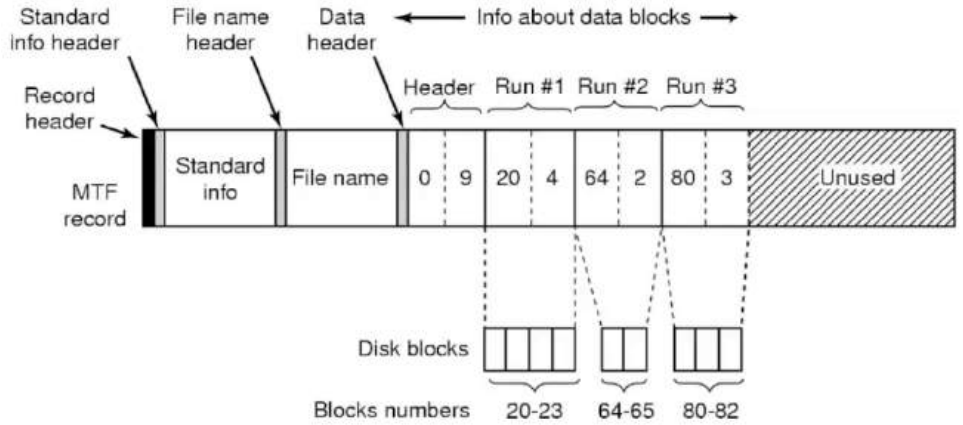


NTFS Medium File

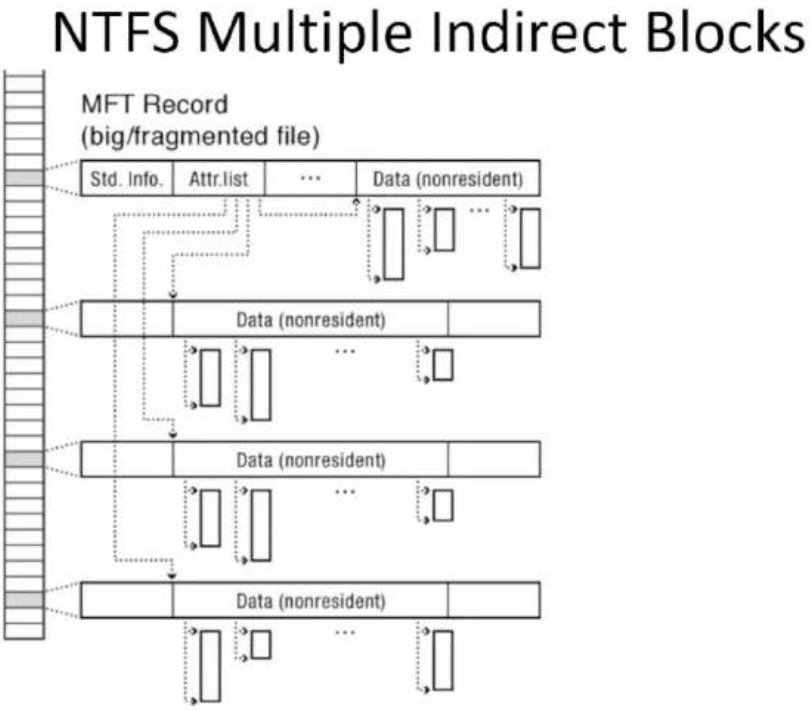


Nella **MFT** abbiamo una tabella dove le entry rappresentano i nostri file e contengono tutti i metadati del file e un numero contenuto di dati del file stesso.
Per file di grandi dimensioni, dove superiamo la grandezza della nostra entry, abbiamo dei **puntatori ad extends** che rappresentano un blocco di lunghezza variabile. Il puntatore all'extent mi dice da dove partire e la lunghezza dell'extent. Si tratta di un insieme di blocchi di cardinalità variabile.
Anche in questo FS si usa la politica **first fit**.
Rappresentazione di un file.

NTFS Medium File



Possiamo avere **più entry per un singolo file** andando a linkare, invece che blocchi del disco, altre entry della master file table.



Implementa un sistema di journaling per il recovery del file system, di modo che in caso di guasti vengano salvati i log per sapere tutte le operazioni che abbiamo fatto col file system.
Journaling
Il journaling tiene traccia delle operazioni che il file system sta per eseguire **in un'area speciale chiamata *journal*** (una sorta di diario).
Prima di modificare realmente i dati o le strutture del file system, il sistema scrive un *log* dell'operazione nel journal.

Rimane il file system migliore.
Contro: frammentazione esterna dovuta alla politica first fit quando alloco l'extent. L'ultimo blocco di ogni file avrà spazio in più.

Directory
Hard link: è un alias di un file che punta direttamente all'i-node. Quindi rappresenta una entry in più nella tabella delle directory, ma non costa di più a livello di spazio. Posso creare **quanti hard link voglio** , ed ognuno **punta allo stesso i-node.** Quando i riferimenti al file diventano 0 elimino il file, dato che non può più essere referenziato.

Soft link: file speciale che **contiene il percorso al file originale**. Questo significa che oltre a rappresentare una entry in più all'interno della mia tabella della directory **devo creare un nuovo i-node, dove al posto del link al file ho un link al nome del file**. Questo causa un overhead nella inode table.

Le directory inizialmente venivano implementate come liste concatenate il problema di questa implementazione è che per directory contenenti un grande numero di file, la ricerca di

un file all'interno diventa inefficiente e dispendiosa. Per evitare di scorrere tutti i file, la directory è stata implementata come albero, dove ogni per ogni file viene calcolato un hash e rappresentato come nodo. Migliorando le prestazioni di ricerca di un file a **O(log(n))**.

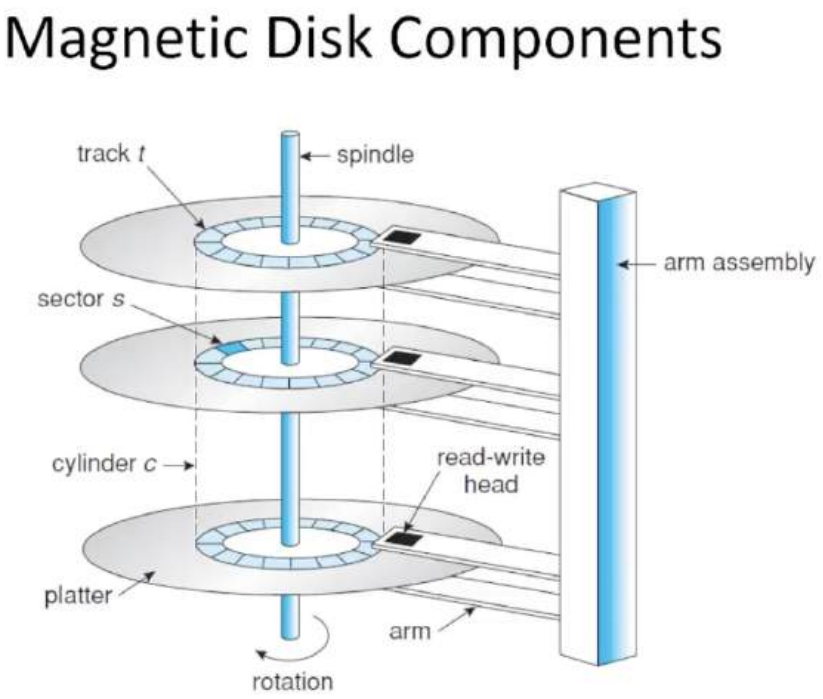
Storage System

Storage System

Punti principali :

- Dischi magnetici
- Flash memory
- Raid storage overview

HDD



Il valore letto è il settore. Ogni settore è identificato da 3 coordinate: cilindro traccia e settore. Quindi quando vado a leggere sul disco non leggo un blocco ma un settore e dai settori faccio la traduzione in blocchi.
Seek = movimento della **testina** dell'HDD per raggiungere la **traccia** (il cerchio di dati) giusta sul piatto.

- Quando il computer vuole leggere o scrivere un dato, l'HDD deve **trovare** la posizione esatta del dato.
- La **testina** si **sposta radialmente** (verso il centro o verso l'esterno del disco) per posizionarsi sulla **traccia** corretta.

Questo movimento si chiama proprio **operazione di seek** (o **ricerca**).

Blocchi per cilindro = #facce * #settori

Cilindro di un blocco = parte intera inferiore (#bloccoFisico / #blocchi per cilindro)
#settore = #bloccoFisico mod #settori

Disco
 ├── Platter (piatti)
│   ├── Tracce (cerchi)
│   │   ├── Settori (spicchi delle tracce)
│   │   └── Cilindri (tracce allineate sui piatti)
└── Testina (lettura/scrittura)

Sectors <-> blocks

Given a sector number b, and a triple $< c, f, s >$:

$$b = c(\#faces \cdot \#sectors) + f(\#sectors) + s$$

* assumption 1 block corresponds to 1 sector

- $\#faces$ is the number of faces in the disk
- $\#sectors$ is the number of sectors per track

Consequently:

$$c = b \; div \; (\#faces \cdot \#sectors)$$
$$f = \big(b \; mod \; (\#faces \cdot \#sectors)\big) div \; \#sectors$$
$$s = \big(b \; mod \; (\#faces \cdot \#sectors)\big) \; mod \; \#sectors$$

Example two plates and 512 sectors per track:
<2,0,3> b= 2*(2*512) + 0*512 + 3 = 2051

Queste formule servono per la traduzione tra blocchi usati dal file system e settori su cui ragiona il disco rigido.

Le prestazioni di un **HDD** dipendono dalla latenza = seek time + rotation time + transfer time
Seek time = tempo di spostamento del braccio per arrivare al cilindro desiderato. 1-20 ms
Tempo di rotazioni = a che velocità ruotano i dischi (+ giri + prezzo). 4-15 ms
Transfer rate = dipende con cosa faccio il trasferimento, che tipo di collegamento, e quanta informazione trasferisco. 50-100 MB/s.

Quando **sfruttiamo** la **località spaziale** aumentiamo le prestazioni di lettura e scrittura sul disco dato che vado a risparmiare il tempo di seek.

SSD

Nessuna parte meccanica.
Lettura e scrittura sono fatte elettronicamente.
Gli accessi non hanno delay dovuti alla meccanica della lettura nel disco.

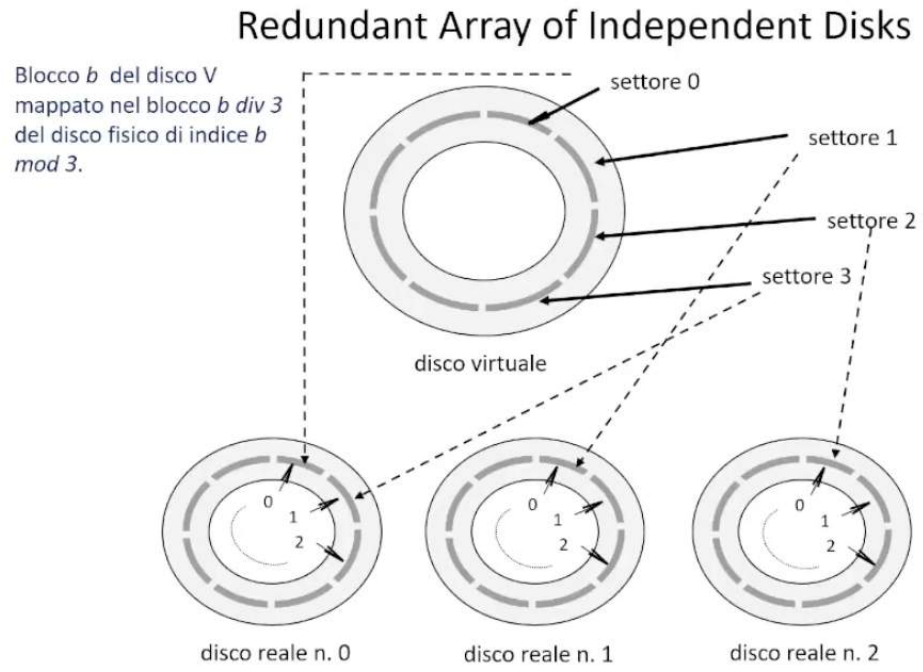
Sono maggiormente resistenti agli urti.
Hanno un problema legato alle scritture, in quanto possono avvenire solo in zone mai state scritte. Quando scrivo nuove pagine non c'è nessun problema, mentre se devo aggiornare qualcosa devo trovare una nuova pagina in memoria per riscrivere gli stessi dati aggiornati. L'informazione sul SSD che devo cancellare non posso cancellarla localmente ma devo cancellare una intera pagina composta da molte più informazioni. Quindi l'SSD cerca di avere sempre pagine bianche disponibili. La gestione dell'SSD è rimandata al suo controller.

Hanno anche una durata limitata dovuta alle scritture che non posso riscrivere andando a perdere memoria.

SSD e HDD coesistono perché se ho enormi quantità di dati ho bisogno degli HDD, mentre uso l'SSD come cache.

Flash translation layer
Si occupa di far trovare sempre la pagina per poter permettere le scritture. Associa ad un indirizzo logico il suo indirizzo fisico. Siccome il controller sposta di continuo le pagine, non sappiamo precisamente dove queste risiedono, quindi non dobbiamo fare la traduzione da indirizzo fisico a logico. Lo sa solo il controller dell'SSD dove si trova un certo blocco.

RAID



Dischi RAID

Livello 0: Dischi asincroni, nessuna ridondanza (striping)

- Si possono effettuare contemporaneamente operazioni indipendenti
- Anche detto JBOD (just a bunch of disks)

Livello 1: Dischi asincroni, disco con copie ridondanti *(mirror)*

- Si possono effettuare contemporaneamente operazioni indipendenti e correggere errori

Livello 2: Dischi sincroni, i dischi ridondanti contengono codici per la correzione degli errori

- non si possono effettuare contemporaneamente operazioni indipendenti e correggere errori

Livello 3: Dischi sincroni, un solo disco ridondante

- contiene la parità del contenuto degli altri dischi
- non si possono effettuare contemporaneamente operazioni indipendenti e correggere errori

Livello 4: Dischi asincroni; 1 disco ridondante

- contiene la parità del contenuto degli altri dischi
- si possono effettuare contemporaneamente operazioni indipendenti e correggere errori
- Il disco ridondante è sovraccarico nei piccoli aggiornamenti

Livello 5: Dischi asincroni, con parità distribuita tra tutti i dischi

- permette un miglior bilanciamento del carico tra i dischi
- minimo 3 dischi

Livello 6: Come livello precedente, ma parità doppia distribuita tra tutti i dischi

- minimo 4 dischi, permette di tollerare fino al fallimento di 2 dischi

Livello 10: Dischi asincroni, Mirror di stripes

Livello 01: Dischi asincroni, Stripe di mirror

Dischi **asincroni**, vengono distribuite le *stripe*

Livello 4: Ridondanza con *strip* di parità

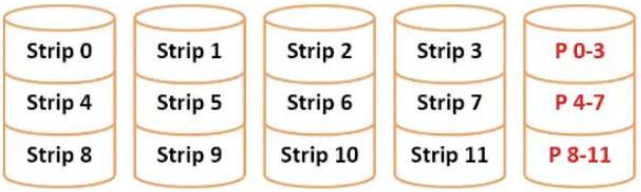
Esempio: disco 4 contiene le strip di parità delle strips omologhe dei dischi 0, 1, 2, 3

- esecuzione contemporanea di operazioni indipendenti e *correzione di crash faults singoli*

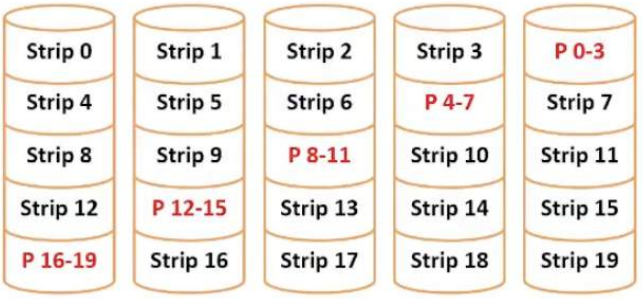
Livello 5: Come livello 4, ma strip di parità distribuite nei vari dischi

- migliore bilanciamento del carico tra i dischi

Raid level 4



Raid level 5



Domande Esame

Esame 1 :

sistemi operativi:

- parlare di FFS. Cosa c'è nei metadati? e nei blocchi dei puntatori indiretti?
- cos'è la core map? quante entry ha? (spoiler: quante le pagine fisiche)
- gestione della memoria con la paginazione, parlare della TLB.
- se ho pagine da 1K, quanti bit mi servono di offset (spoiler: 10)
- funzionamento scheduling FIFO e SJF. quando sono convenienti?

architetture:

- funzione che dato un array di caratteri e un carattere di input, restituisce la posizione della prima ricorrenza del carattere
- quante varianti della load si hanno
- differenza tra single cycle e multi cycle
- quanti cicli di clk si necessitano per la branch e per la store?
- cos'è il forwarding e quando non basta?

Esame 2 :

sistemi operativi:

- come funziona la paginazione dinamica?
- multi level paging
- se ho indirizzi a 32 bit e pagine da 4k quanto è grande la tabella? (2^20)
- cosa c'è in una entry della page table?
- perché durante la fork è giusto copiare le pagine mentre con la exec no?
- parlare della FFS. Dov'è contenuto il nome del file? cos'è una directory?
- quando è rilevante che il FFS sfrutti la località usando i gruppi? (spoiler: con HHD) spiegare i gruppi
- difetti principali del FFS
- PTHREAD YELD, quando si usa maggiormente ? (spoiler: user level thread)
- come funzionano i semafori? sono equivalenti alle c.v.? (spoiler: no, confrontare V() e signal())
- quando ha senso usare attesa attiva? indicare gli svantaggi -> bassa banda I/O

architetture:

- Load in multicycle. parlare del ritardo sia in load che nella store.
- differenza tra multicycle e singlecycle
- come viene realizzato il salto nel multicycle?
- tecnica del forwarding: scrivere un pezzettino di codice in cui c'è dipendenza e risolverla
- a livello di rete, cosa va controllato per fare il forwarding?
- (di tale domanda chiede l'idea) LDR R0, [R1,R2,LSL 2] in linguaggio macchina
- che tipo di shift ci sono, la lsl quanti bit richiede? differenza shift logica e aritmetica
- cosa sono push e pop
- tipo di stack in ARM

Esame 3 :

sistemi operativi:

- Sistemi RAID (descrive i livelli 0,1,4 e 5)
- cos'è il DMA? come si sa che la trasmissione è terminata? (spoiler: interruzioni)
- interrupt driven I/O
- fs di tipo FAT. se ho un disco da 1M con blocchi da 1k quante entry ho? (spoiler: 1000)
- pro e contro FAT, con cosa funziona meglio? (spoiler: accessi sequenziali)
- perché non ci sono questi problemi nel FFS? come capisco che puntatori usare? (spoiler: dimensione del file)
- ho 2K blocchi e puntatori da 4 byte. quanti puntatori metto in un blocco? (2^10 / 2^2 = 2^8)
- differenza tra HDD e SSD? ha senso parlare di località negli SSD?
- traduzione degli indirizzi con paginazione a 2 livelli, con pagine da 4K e indice di primo livello a 8 bit. come funzione?
- parla dei semafori

architetture:
- che cos'è una rete sequenziale?
- ritardo di propagazione e contaminazione
- thold e tsetup nei registri
- cos'è LATCH SR? disegnalo. LATCH D? cosa risolve?
- descrivere il multicycle. Quando è conveniente? descrivere load e store.
- (ha indicato un percorso) quando viene usato? (salto)
- quando serve lo stallo nel pipeline? come si realizza nel data path?

Esame 4 :

sistemi operativi:
- come funziona il meccanismi di fork, exec e wait?
- cos'è un processo zombie?
- cos'è e dove si usa il copy on write?
- cos'è la TLB? quali sono i possibili problemi? TLB **shootdown**, associazione processo indirizzo virtuale
- parla dell MFQ
- da quale punto di vista SSF è ottimo?
- semantica Mesa e Hore

architetture:
- differenza tra single cycle e multi cycle.
- quanti cicli di clk per la branch e per la store?
- nel pipeline cos'è il forwarding? quando non basta?
- funzione che prende un vettore, size e un carattere, restituisco 0 se c'è 1 altrimenti
- spiega il pipeline? quanti cicli al max per istruzione?
- come si gestiscono i salti nel pipeline?
- altri modi per fare salto oltre alla branch

Esame 5 :

sistemi operativi:
- spinlock come funziona, che tipo di istruzione serve (spoiler: read modify write)
- memory barrier
- pseudocodice lock nel multi processore
- politica di scheduling Round Robin.
- come funzionano le interruzioni (dual mode operation)
- politica di scheduling MLFQ
- elencare le altre politiche di scheduling
- descrivere uno scheduling per multiprocessore

architetture:
- cos'è una rete sequenziale e sapendo delle varie parti che la compongono, come si può stimare il tempo di clk?
- che tipi di rete sequenziale conosciamo? (Mealy e Moore)
- calcolare il ciclo di clk in una rete
- differenza tra memoria modulare sequenziale e interlacciata
- se ho un modulo da 1k con porte di read w di write, ne voglio costruire una da 2k. come le devo collegare?
- cosa vuol dire avere una dipendenza sui dati in un'architettura pipeline, fammi un esempio. come lo risolvo (forwarding)
- in che casi serve lo stallo in aggiunta al forwarding?
- cosa succede con il salto?

Esame 6 :

sistemi operativi:
- parlare dei semafori (come funzionano, che valori possono avere)
- cosa fanno P() eV()
- come si implementa la mutua esclusione con i semafori
- differenza tra V() e signal
- SJF
- differenza starvation e deadlock. Si può uscire dallo stallo?
- condizioni necessarie che portano allo stallo
- algoritmo del banchiere
- come funziona il file system FAT

architetture:
- differenza memoria interlacciata e sequenziale.
- come realizzo una memoria modulare interlacciata da 2k con moduli da 1k? disegno
- scrivere una funzione che prende come parametro un puntatore a vettore di interi, la size e un intero x. torna l'indice di x se no 0
- dove si trovano per convenzione i parametri di ingresso? (spoiler: r0, r1, r2 e il risultato in r0)
- quanti tipi di load ci sono?
- caratterizzazione dell'architettura single cycle
- qual è l'istruzione più lunga? (LDR)

- a cosa servono i commutatori prima del RF? perché in un'entrata si ha 15? quale istruzione la
  usa (branch)
- come fa la control unit a capire che c'è un salto?
- differenza single cycle e multi cycle
- cos'è una rete sequenziale?

Lista delle domande (possono esserci doppioni):

- Cosa succede se elimino la pc da una rete sequenziale
- Spiegare il multithreading interleaving e blocked
- Spiegare le dipendenze iu-eu, eu-eu e qualche nozione generale sul processore superscalare
- Memoria
- Implementare un arbitro a richieste indipendenti senza l'utilizzo del registro turno
- Vettore interruzioni, e visto che avevo fatto confusione tutta la gestione dell'interruzione
- domanda per il vecchio ordinamento : implementare la receive (praticamente scena muta)
- dipendenze EU-EU (come funzionano, cosa fare per ridurne il degrado)
- multithreading blocked e interleaving
- condizioni di Bernstein (quali sono con esempi di microcodice)
- memoria: come funzionano le cache di primo e secondo livello (in generale e in caso di fault)
- trattamento delle interruzioni in maniera molto dettagliata (perché il mprogramma è fatto da 3 istruzioni, se si poteva fare con meno, ecc)
- processore superscalare in generale e come si modificano IM, IU, DM, EU rispetto al processore pipeline
- spiegare il multithreading blocked e interleaving
- controllo residuo con esempio, condizioni di bernstein con piccolo codice esempio
- realizzazione tramite componenti logici delle interfacce a transizione di livello per la sincronizzazione (RDY e ACK)
- varie domande sulle gerarchie di memoria (come funzionano le cache, come si possono realizzare ecc)
- cooperazione tra processi ed implementazione di send e receive (sono classe 26 altrimenti il capitolo sui processi non farebbe parte del programma).
- Multithread e supporto firmware per attuarlo
- Superscalare e supporto firmware per attuarlo
- Differenze fra le due architetture, con alcuni esempi di possibile funzionamento
- Casi di Riuso e Località istruzioni assembly
- Schedulazione e commutazione di contesto di processi (in particolare nel caso del quanto di tempo e dei processi con priorità)
- Condizioni di Bernstein: spiegazioni in termini del ciclo di clock (con esempio di codice)
- Eu parallela: in che modo si sfrutta il modello pipeline e il modello a replicazione funzionale, dipendenze EU-EU
- Cosa comporta in termini di unità di tempo l'utilizzo della EU slave e che cosa si deve considerare nel rapporto con la IU e nella gestione dei registri interni. (domanda per lode)
- PC con componenti standard
- Cache: Write Back e Write Through

- Tempo di completamento effettivo e ideale su processore monolitico e pipeline (con esempio)
- Tempo necessario all'esecuzioni di due istruzioni LOAD consecutive su processore monolitico
- Numero di stati in un atoma di Mealy/Moore necessari a rappresentare un dato numero di stati
- Automa di Mealy e di Moore per contare la parità di una stringa binaria (e realizzazione di quello di Moore)
- Come realizzare un processore superscalare a 4 vie (gestione dei registri nella IU)
- Calcolo del numero di stati / tempi di una rete sequenziale che ha scritto lui sul momento
- Processo di sintesi formale, con disegno e tutto. Per capirci, alla compito.
- Perche', nel calcolo del tempo di ciclo di clock, insieme a omegaPC+sigmaPO, confrontiamo anche sigmaPC, nel massimo ? Tradotto, puo' essere sigmaPC > omegaPC+sigmaPO ?
- Differenza tra rete Mealy e rete Moore.
- Parallelismo Farm
- Aggiungere nell'interprete fw una istruzione di MOVI che copia 32bit in un registro dalla memoria. Dopo averla scritta, aumenta il ciclo di clock?
- Come si risolvono le dipendenze logiche? risp, allontanandole tra di loro e inserendo nel mezzo istruzioni non in dipendenza.
- Come fa il compilatore a capire quali istruzioni inserire? non sapevo cosa rispondere, ho provato spiegando i semafori dei registri, ma ha detto che il compilatore non li vede, rimane tutt'ora una domanda senza risposta per me.
- Come ridurre le probabilità di salto? Loop unfolding e espansione di macro e di funzioni e poi ho spiegato anche il delayed branch.
- Fai il microcodice di un arbitro, poi fallo fair e poi fai la pc dell'arbitro.
- Metodo set associative, divisione dei bit dell'indirizzo (offset, set, tag), poi mi ha chiesto come si fa a vedere se è nell'insieme o meno (ho spiegato che si confrontano i tag, voleva sapere nello specifico come, ma non mi ricordavo bene).
- Come funziona la MMU e la traduzione di indirizzi (nel dettaglio, con IPL-IPF e offset, tab.ril, schema dell'MMU)
- Superscalare a 2 vie: condizioni per mettere due istruzioni in contemporanea, problemi con LOAD/LOAD (soluzione standard, poi mi ha chiesto se si riesce a potenziare l'accesso alla cache per leggere due indirizzi diversi insieme)
- Rete di Mealy, ha scritto un automa sul momento e mi ha chiesto di fare le tabelle di verità di ω e σ)
- (domanda difficile) cambiare la gestione delle interruzioni nel processore sequenziale: cambio l'interfaccia con UINT (l'arbitro che sceglie di quale periferica prendere l'interruzione) per ricevere oltre al segnale anche i due valori (quelli che di solito arrivano tramite la MMU) in registri. Vantaggi, svantaggi e modifiche da fare al processore e alle unità FW esterne che mandano le interruzioni.
- Definizione di dipendenze logiche, dove si trovano nel processore sequenziale e dove in quello pipeline (spoiler: nel sequenziale non ci sono)
- Condizioni di Bernstein, quali sono e quali si considerano nel microcodice
- Vantaggi e svantaggi di costruire un commutatore a 4 ingressi a partire da una rete di commutatori da 2 vs implementazione ad hoc con tabella di verità. E se avessi voluto un commutatore a 2 ingressi da 32 bit a partire da una rete di commutatori a 2 ingressi da 32 bit? E se lo avessi fatto con una tabella di verità?

- Cosa succede ad un indirizzo da quando esce dal processore a quando arriva alla cache set-associativa, in cui è presente? (Quindi paginazione, funzionamento della MMU, funzionamento della cache set-associativa)
- Implementa un'istruzione ADDM Ra Rb Rc che prende due parole alle locazioni Ra ed Rb dalla memoria, le somma e mette il risultato in Rc. Descrivi i pro e i contro dell'implementazione a livello firmware rispetto all'implementazione come istruzione derivata.
- In quali casi conviene cambiare o tau o k nel tempo di completamento di un programma T = tau * k (ricordo che k è la somma del numero di istruzioni per ogni operazione, ognuna moltiplicata per la probabilità che venga richiesta), cosa succede a tau se cambio k e viceversa, in pratica cosa succede al microcodice e cosa succede alla parte operativa; controllo residuo.
- Politica write through e write back, quando conviene usare una o l'altra, e di cosa mi devo assicurare in un programma prima di usare la w. through.
- Processore pipeline superscalare con multithread simmetrico, caratteristiche implementative (soprattutto IM e DM), vantaggi e svantaggi.
- Alcune domande per verificare che fossi l'autore del progetto Verilog
- In Verilog come scriveresti il commutatore a 8 vie a N bit usando piu' commutatori a 8 vie 1 bit? Altre domande sui ritardi, sulle tabelle di verita' in Verilog
- Date le unita' A, B e una unita' intermedia tra A e B, scrivere un micro-programma tale per cui l'unita' intermedia mandi un dato da A a B
- Come funziona la commutazione di contesto, in particolare chi se ne occupa, come funziona la start_process, come deve essere realizzata la MMU per riconoscere una commutazione di contesto, come deve essere memorizzata le Tabril dei processi in memoria principale
- Mi viene proposto un codice DRISC con un po' di istruzioni, dire il Tempo di Completamento ideale (#istruzioni * t), il Tempo di completamento non ideale (con le bolle sulla IU), Tempo di Completamento effettivo (con la presenza di una memoria cache), quindi per quest'ultimo bisognava anche calcolare il #Fault, dire poi in quale caso avrei avuto 2 fault invece che 1 (elementi B e B[i+1] su linee diverse della cache)
- Come avviene la traduzione di un indirizzo (IPL.OFF -> IPF.OFF)
- Come si scompone l'indirizzo per indirizzare una Cache Set Associativa (TAG, #SET, OFF)
- Cosa succede se la MMU trova il bit di presenza a 0 nella tabella di rilocazione (in memoria principale)
  R: La MMU setta il bit di Esito (quello relativo al "FAULT DI PAGINA") a 1, il processore quindi andrà al trattamento eccezioni.
- Tecnica Out-of-order nel Pipeline, se nel mio programma ho bolle che durano solamente 1, il tempo di completamento migliora o peggiora?
  R: Rimane lo stesso perchè il tempo risparmiato eliminando la bolla da 1 viene comunque speso per rieseguire l'istruzione successivamente.
- Differenza tra Tempo di Servizio e Tempo di Completamento nel Pipeline
- La comunicazione a livelli è asincrona o sincrona? Perchè? Come renderla sincrona?
- Descrivi le forme di parallelismo nel processore e i problemi nei vari casi. Da cosa sono causati? Come possono essere risolti?
- Come funziona la paginazione? Quante pagine sono necessarie per immagazzinare la tabella di rilocazione ipotizzando 4K indirizzi per pagina?