Architettura degli elaboratori semplice (per davvero)

Rovesti Gabriel

Attenzione

Il file non ha alcuna pretesa di correttezza; di fatto, è una riscrittura attenta di appunti, slide, materiale sparso in rete, approfondimenti personali dettagliati al meglio delle mie capacità. Credo comunque che, per scopo didattico e di piacere di imparare (sì, io studio per quello e non solo per l'esame) questo file possa essere utile. Semplice si pone, per davvero ci prova.

Thank me sometimes, it won't kill you that much.

Gabriel

Sommario

Introduzione

Architettura: caratteristiche visibili al programmatore
- Istruzioni
- Spazi (numero bit) usato per rappresentare i dati
- Tecniche di indirizzamento della memoria

Organizzazione: unità operative e loro connessioni
- Interfacce tra calcolatore e periferiche
- Tecnologia per le memorie

Esempio: Istruzione per la moltiplicazione:
- Decidere se è disponibile, è una decisione architetturale
- Come implementarla (circuito per la moltiplicazione o somme ripetute) è una decisione di organizzazione (costo, velocità, ...)

Modelli diversi della stessa marca: stessa architettura, organizzazione diversa
- Esempio: architettura dell'IBM 370 (dal 1970)
    - Fino ad oggi per calcolatori mainframe
    - Varie organizzazioni con costo e prestazioni diverse

Struttura e funzione

Calcolatore:
- Insieme di componenti connesse tra loro

Visione gerarchica
- Insieme di sottosistemi correlati
- Ogni sistema ad un livello si basa sulla descrizione astratta del livello successivo

Ad ogni livello
- Struttura: come sono correlati i componenti
- Funzione: cosa fa ciascun componente

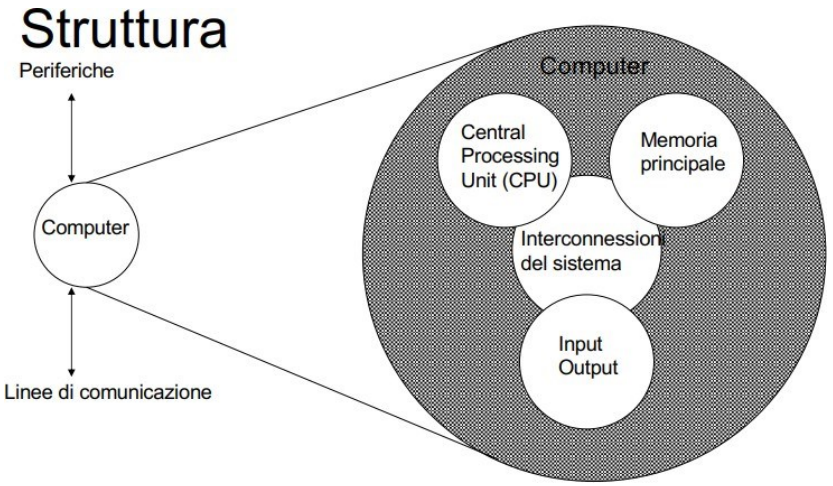
Descrizione top-down:
- da componenti principali a sottocomponenti, fino a una descrizione completa dei dettagli

Funzioni basilari di un calcolatore (livello più alto della gerarchia)

- Elaborazione dati
- Memorizzazione dati
- Trasmissione dati
    - Input/output o verso un dispositivo remoto
- Controllo
    - Delle tre funzioni sopra

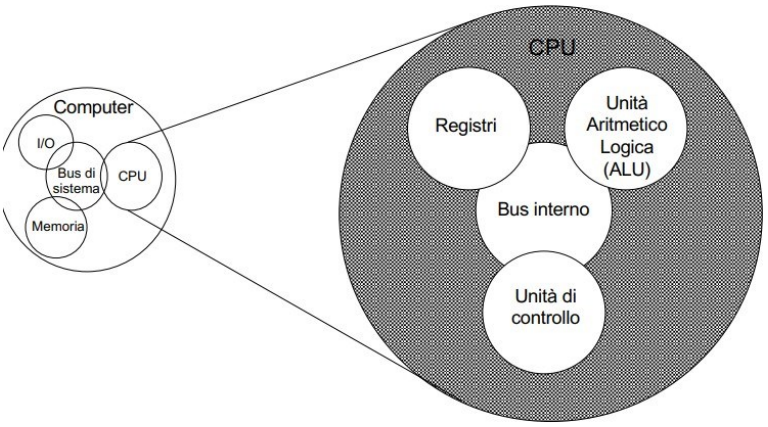
Struttura (livello più alto della gerarchia)
Quattro componenti principali:

- Unità centrale di elaborazione (CPU)
    - Esegue le funzioni di elaborazione dati
- Memoria centrale
    - Per immagazzinare i dati
- I/O (input/output)
    - Per trasferire i dati tra calcolatore ed esterno
- Interconnessioni
    - Per far comunicare CPU, memoria centrale, e I/O



Central Processing Unit (Unità Centrale di Elaborazione)

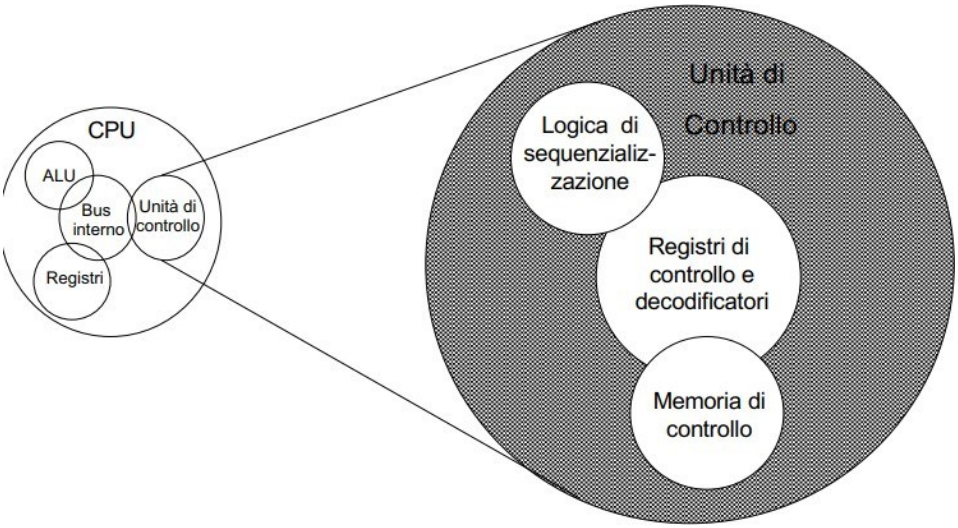
- *Unità di controllo*
    - *Controlla la sequenza di operazioni*
- *Unità aritmetico-logica (ALU)*
    - *Elaborazione dati*
- *Registri*
    - *Memoria interna della CPU*
- *Interconnessioni*
    - *Comunicazione tra unità di controllo, ALU e registri*



Dentro la CPU vediamo l'unità di controllo, è un componente della CPU (Central Processing Unit) di un computer. L'unità di controllo è conosciuta anche con la sigla CU (Control Unit). È il dispositivo della CPU a cui spettano le funzioni di controllo. L'unità di controllo coordina il flusso di dati tra il processore e gli altri componenti del computer, legge ed esegue le istruzioni nella memoria centrale. Il funzionamento dell'unità di controllo del computer si basa sul processo fetch-execute (lettura-esecuzione).

- Fetch. Nella fase fetch (lettura) l'unità di controllo legge dalla memoria centrale del computer le istruzioni da eseguire, scrivendole nel registro di istruzione. Le istruzioni sono registrate in particolari locazioni di memoria consecutive, al fine di agevolare la successiva operazione di interpretazione ed esecuzione sequenziale delle istruzioni mediante la locazione del contatore di programma (program counter).
- Execute. Nella fase execute (esecuzione) l'unità di controllo interpreta ed esegue in modo sequenziale le istruzioni situate nel registro di istruzione. Nel caso di calcoli tra dati numerici, l'unità di controllo trasferisce i dati dalla memoria centrale del computer all'unità aritmetico-logica (ALU) e attende il risultato.

L'unità di controllo legge le istruzioni da eseguire dalla memoria centrale e le scrive nel registro di istruzioni, al fine di poterle interpretare ed eseguire in modo sequenziale, inviando segnali di controllo alle unità del computer interessate.



Perché studiare l'architettura dei calcolatori?

- Capire i compromessi costo-prestazioni
  - Esempio: scegliere il calcolatore migliore a parità di costo
    - spesa maggiore ma memoria più grande o frequenza di clock più alta e quindi maggiore velocità
- Supporto ai linguaggi di programmazione
  - Diverso a seconda delle architetture

Evoluzione dei calcolatori

Nel corso del tempo, l'evoluzione dei calcolatori ha portato:

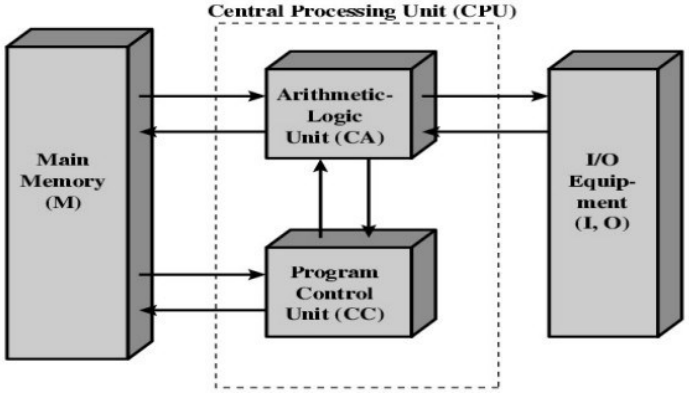
- Processori sempre più veloci
- Componenti sempre più piccoli → più vicini → elaborazione più veloce
  - Ma la velocità è derivata anche da nuove tecniche (pipeline, parallelismo, ecc.) che tengono occupato il processore il più possibile
- Memoria sempre più grande
- Capacità e velocità di I/O sempre maggiore
- Tecniche per bilanciare velocità diverse di processore e memoria
  - Memoria cache, ecc.

Questi, infatti, si sono adattati ad eseguire compiti ripetitivi e complessi e quindi automatizzabili. Ciò che a noi interessa in questa lezione è il modello della macchina di Von Neumann.

- Programma memorizzabile come i dati
- Istruzioni in memoria: decidere il programma specificando una porzione di memoria
- Idea di John von Neumann (consulente ENIAC, uno dei primi calcolatori)

La struttura di Von Neumann è così formata:

- Memoria, contiene dati e istruzioni
- Molte operazioni di aritmetica, quindi dispositivi specializzati per eseguirle è unità aritmetico-logica (dati binari)
- Organo centrale per il controllo della sequenza delle operazioni, generico è unità di controllo
- Organi di ingresso e uscita



La struttura di Von Neumann è nota anche come computer IAS.

- Memoria:
  - 1000 locazioni (parole), numerate da 0 a 999 (indirizzo)
  - Ogni parola: 40 cifre binarie (0 o 1, bit)
- Dati e istruzioni in memoria:
  - numeri in forma binaria: bit di segno + 39 bit per il numero
  - istruzioni con codice binario:
    - due in ogni parola
    - 8 bit per codice istruzione, 12 bit per indirizzo parola di memoria
- Unità di controllo: preleva le istruzioni dalla memoria e le esegue una alla volta

La struttura è quella riportata a lato.

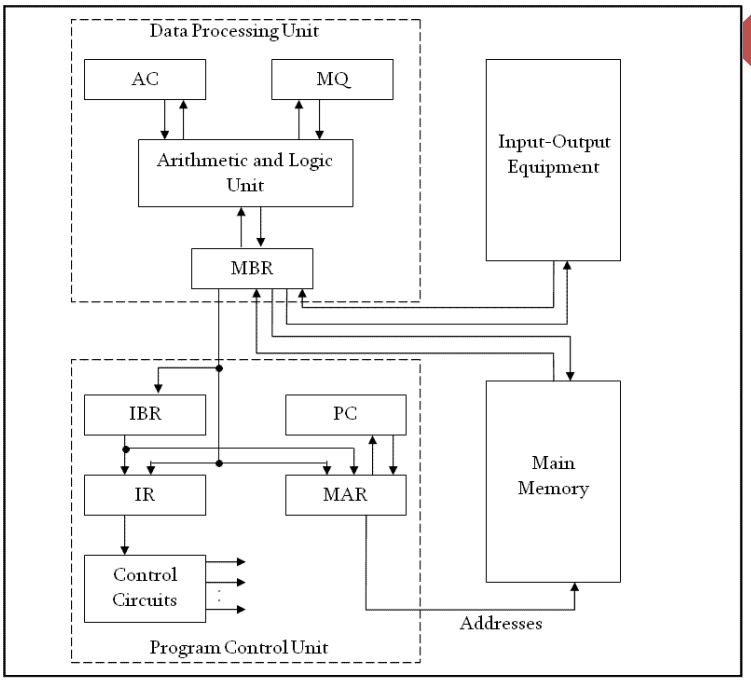


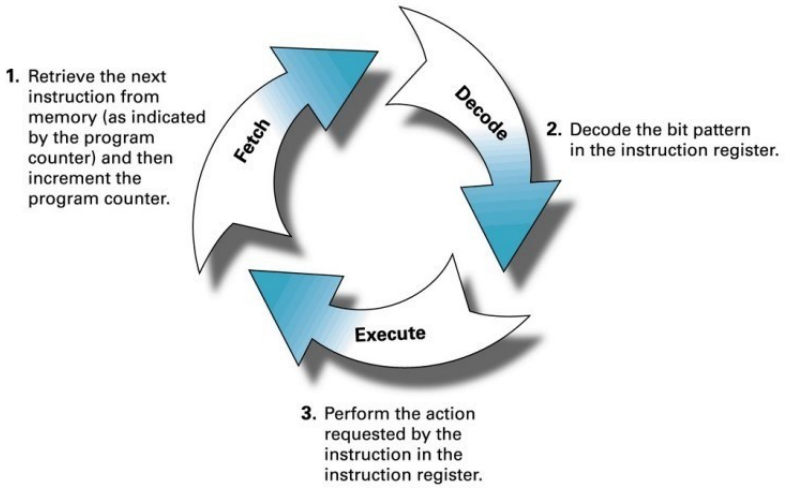
Figure: Expanded structure of Von Neumann Architecture or IAS computer

I registri IAS invece sono:

- MBR (memory buffer register)
  - Contiene una parola da immagazzinare in memoria, o da leggere dalla memoria
- MAR (memory address register)
  - Contiene un indirizzo di una parola di memoria (dove scrivere il contenuto di MBR o da trasferire in MBR)
- IR (instruction register)
  - Contiene 8 bit per il codice operativo dell'istruzione in corso
- IBR (instruction buffer register)
  - Contiene temporaneamente l'istruzione destra di una parola
- PC (program counter)
  - Indirizzo della prossima coppia di istruzioni da prendere dalla memoria
- AC (accumulator) e MQ (multiplier quotient)
  - Temporaneamente, operandi e risultati parziali delle operazioni della ALU

La CPU esegue un programma memorizzato in memoria prendendo ad una ad una le istruzioni, nell'ordine in cui sono memorizzate. Il ciclo della CPU è composto da (definito anche sopra nelle due fasi):

- Prelievo dell'istruzione (fetch):
  - Istruzione letta da IBR o dalla memoria tramite MBR, IBR, IR e MAR
  - Carica il codice dell'istruzione successiva nell'IR e indirizzo in MAR
- Esecuzione dell'istruzione:
  - Attiva i circuiti necessari per l'operazione da eseguire



Le istruzioni IAS sono 21 in totale, comprendendo trasferimento dati tra i registri, *salti* con o senza condizione tra i vari pezzi (approfondito il concetto di salto successivamente), aritmetiche, modifica di indirizzo (da mettere in memoria). I primi 8 bit sono uno dei 21 codici e i successivi 12 sono le celle di memoria coinvolte.
Esempi di istruzioni IAS:

- LOAD MQ, M(X)
  - Trasferisce il contenuto della cella di memoria di indirizzo X in MQ
- STOR M(X)
  - Trasferisce il contenuto dell'accumulatore nella locazione X della memoria
- JUMP M(X,0:19)
  - Carica l'istruzione dalla metà sinistra di M(X)
- ADD M(X)
  - Somma M(X) ad AC e mette il risultato in AC

La grande innovazione fu certamente l'invenzione dei transistor.
Il transistor è un dispositivo elettronico a tre terminali che amplifica o commuta segnali elettronici. I suoi componenti essenziali sono due materiali semiconduttori, tipicamente il silicio, con proprietà opposte, noti come tipo p e tipo n.
Quando i due materiali vengono accostati, formano una barriera a strato di deplezione. Questo strato agisce come un *interruttore*, permettendo alla corrente elettrica di passare o non passare, a seconda della tensione applicata al terzo terminale, noto come *gate*.
I transistor sono presenti in quasi tutti i dispositivi elettronici e sono componenti fondamentali dei circuiti integrati o chip. Inventati nel 1947 nei Bell Laboratories, i transistor hanno rivoluzionato l'elettronica rendendo possibili dispositivi più piccoli, più economici e più affidabili.

Ulteriore grande invenzione: il circuito integrato.
Esso è un unico pezzo di silicio per molti componenti e le loro connessioni. Col tempo, sempre più componenti in un circuito integrato. Altre definizioni utili:

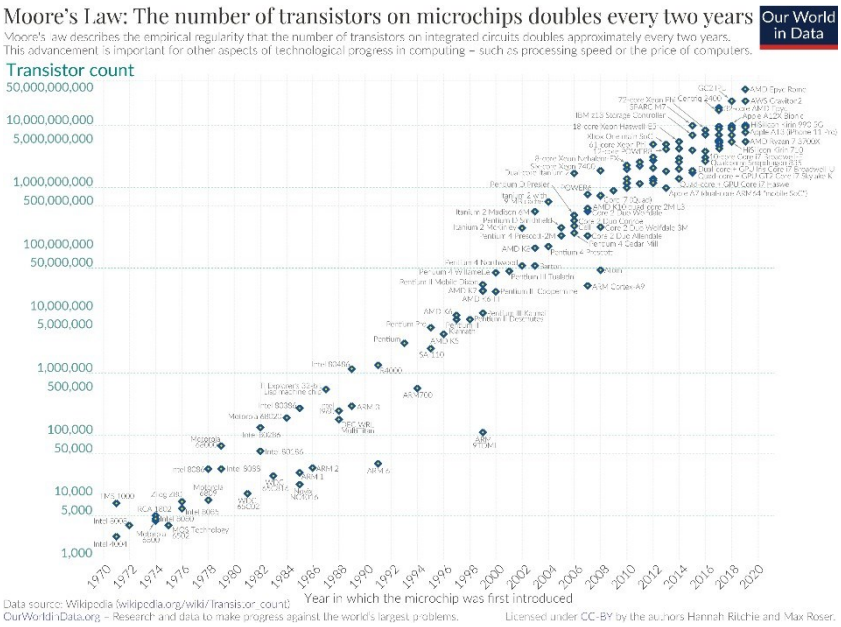
- Porta logica
  - Dispositivo che esegue una semplice funzione logica
  - Esempio: se A e B sono veri allora C è vero (porta AND)
- Cella di memoria: dispositivo in grado di memorizzare un bit (due stati possibili)
- Calcolatore: numero grandissimo di porte logiche e celle di M (M=memoria)

Le sue funzioni:

- Memorizzazione dati
  - celle di memoria
- Elaborazione dati
  - porte logiche
- Trasferimento dati
  - tra memoria e memoria, direttamente o attraverso porte logiche
- Controllo
  - segnali di controllo per attivare le porte logiche o leggere/scrivere una cella di memoria

Ultima grande innovazione: dispositivi time-sharing, cioè un approccio all'uso interattivo dei computer in cui un singolo computer viene utilizzato contemporaneamente da più utenti, ciascuno con un proprio terminale. Similare a questa l'elaborazione batch, altro modo di interagire con un mainframe, ma associato per tutto il tempo ad un singolo utente.

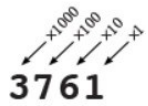
Altra definizione fondamentale in informatica: la legge di Moore, termine usato per riferirsi all'osservazione fatta da Gordon Moore nel 1965, secondo cui il numero di transistor in un circuito integrato (IC) denso raddoppia ogni due anni circa.
La Legge di Moore non è morta, è ancora valida. Sebbene sia vero che la densità dei chip non raddoppia più ogni due anni (quindi la Legge di Moore non si sta più verificando secondo la sua definizione più rigorosa), essa continua a fornire miglioramenti esponenziali, anche se a un ritmo più lento.



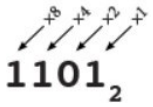
Notazione binaria, ottale, esadecimale. Algebra di Boole

Sebbene le persone lavorino con i numeri utilizzando il sistema numerico in base 10 (decimale), altri sistemi sono rilevanti per l'informatica, tra cui quello binario (base 2) e quello esadecimale (base 16). I computer gestiscono i dati sotto forma di sequenze di bit (cifre binarie), che sono tutti zeri o uno. Le persone hanno maggiore familiarità con la base 10, quindi scriviamo un software che consente di utilizzare la base 10 per comunicare con il computer.

In base 10 ci sono dieci cifre (0-9) e ogni posto vale dieci volte il posto alla sua destra.



In binario, base 2, ci sono solo due cifre (0 e 1) e ogni posto vale due volte il posto alla sua destra.



Il pedice 2 su 11012 indica che 1101 è in base 2. I numeri sono normalmente scritti in base 10, quindi il pedice 10 viene utilizzato solo se necessario per chiarezza.

Nella notazione in base 10, ogni valore di posto rappresenta una *potenza di dieci*: il posto delle unità ($10^0 = 1$), il posto delle decine ($10^1 = 10$), il posto delle centinaia ($10^2 = 100$), il posto delle migliaia ($10^3 = 1000$), ecc. Quindi, ad esempio:

$$9827 = 9 \times 10^3 + 8 \times 10^2 + 2 \times 10^1 + 7 \times 10^0$$

Nella notazione in base 2 si utilizza la stessa idea, ma con potenze di due invece che di dieci. I valori dei luoghi binari rappresentano le unità ($2^0 = 1$), i due ($2^1 = 2$), i quattro ($2^2 = 4$), gli otto ($2^3 = 8$), i sei ($2^4 = 16$), ecc. Quindi, ad esempio:

$$10010_2 = \mathbf{1 \times 2^4} + 0 \times 2^3 + 0 \times 2^2 + \mathbf{1 \times 2^1} + 0 \times 2^0 = 16 + 2 = 18_{10}$$

Altro esempio (delle slide):

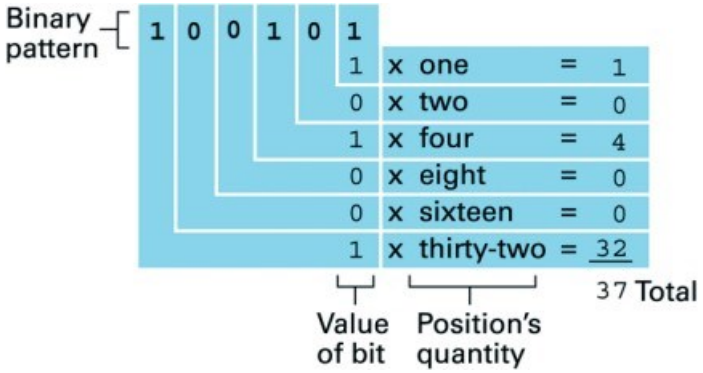
• Esempio:

$0101101_2 = 1 \cdot 2^5 + 1 \cdot 2^3 + 1 \cdot 2^2 + 1 \cdot 2^0$

$= 32 + 8 + 4 + 1$

$= 45_{10}$

Valore di una rappresentazione binaria



Valore minimo → 0^{10}
Valore massimo → $2^n - 1$

▪Da 0 a 8 (su 4 bit):

0 1 2 3 4 5 6 7 8

0000, 0001, 0010, 0011, 0100, 0101, 0110, 0111, 1000

Ricordiamo che si parte dal cosiddetto *bit* (0 oppure 1). 8 bit formano un *byte*.
Abbiamo poi le scale:

- Kilo (10^3) → 1024 B/Byte formano 1 KB (Kilobyte)
- Mega (10^6) → 1024 KB formano 1 MB (Megabyte)
- Giga (10^9) → 1024 GB formano 1 MB (Gigabyte)
- Tera (10^{12}) → 1024 GB formano 1 TB (Terabyte)
- Peta (10^{15}) → 1024 TB formano 1 PB (Petabyte)

Per rappresentare i numeri rappresentiamo varie notazioni, ad esempio:

- Notazione ottale (base 8)

Essa è formata da 8 simboli, rappresentando ogni gruppo di 3 cifre binarie a numeri binari. Di solito hanno una lunghezza multipla di 3.

Es.: 101101010011 (binario)

101 101 010 011

5 5 2 3 → 5523 (ottale)

Tabella di conversione

binario	ottale
000	0
001	1
010	2
011	3
100	4
101	5
110	6
111	7

$(010\ 110\ 000\ 101)_2$ → $(2\ 6\ 0\ 5)_8$

- Notazione esadecimale (base 16)

Essa è formata da 16 simboli, rappresentando ogni gruppo di 4 cifre binarie a corrispettive traduzioni. Di solito hanno una lunghezza multipla di 4.

• Es.: 101101010011 diventa B53

Bit pattern	Hexadecimal representation
0000	0
0001	1
0010	2
0011	3
0100	4
0101	5
0110	6
0111	7
1000	8
1001	9
1010	A
1011	B
1100	C
1101	D
1110	E
1111	F

Tabella di conversione

binario	esadecimale	binario	esadecimale
0000	0	1000	8
0001	1	1001	9
0010	2	1010	A
0011	3	1011	B
0100	4	1100	C
0101	5	1101	D
0110	6	1110	E
0111	7	1111	F

$(0100\ 1110\ 1100\ 1000)_2$
↓ ↓ ↓ ↓
$(4\ E\ C\ 8)_{16}$

Avviene quindi la manipolazione logica di bit, tramite la cosiddetta algebra di Boole, usando apposite variabili logiche (binarie) che possono assumere valore 0 (falso) ed 1 (vero).
Le operazioni logiche di base sono come segue:

- OR (somma)
- AND (prodotto)
- NOT (opposto)
- NAND (opposto di AND)
- XOR (1 se diversi, 0 se uguali)
- XNOR (0 se diversi, 1 se uguali)

Function	x	0	0	1	1
	y	0	1	0	1
Constant 0	0	0	0	0	0
And	$x \cdot y$	0	0	0	1
x And Not y	$x \cdot \bar{y}$	0	0	1	0
x	x	0	0	1	1
Not x And y	$\bar{x} \cdot y$	0	1	0	0
y	y	0	1	0	1
Xor	$x \cdot \bar{y} + \bar{x} \cdot y$	0	1	1	0
Or	$x + y$	0	1	1	1
Nor	$\overline{x + y}$	1	0	0	0
Equivalence	$x \cdot y + \bar{x} \cdot \bar{y}$	1	0	0	1
Not y	$\bar{y}$	1	0	1	0
If y then x	$x + \bar{y}$	1	0	1	1
Not x	$\bar{x}$	1	1	0	0
If x then y	$\bar{x} + y$	1	1	0	1
Nand	$\overline{x \cdot y}$	1	1	1	0
Constant 1	1	1	1	1	1

Altre tabelle utili:

Operatori booleani su due variabili

P	Q	NOT P ($\overline{P}$)	P AND Q ($P \cdot Q$)	P OR Q ($P + Q$)	P NAND Q ($\overline{P \cdot Q}$)	P NOR Q ($\overline{P + Q}$)	P XOR Q ($P \oplus Q$)
0	0	1	0	0	1	1	0
0	1	1	0	1	1	0	1
1	0	0	0	1	1	0	1
1	1	0	1	1	0	0	0

Algebra booleana: postulati e identità

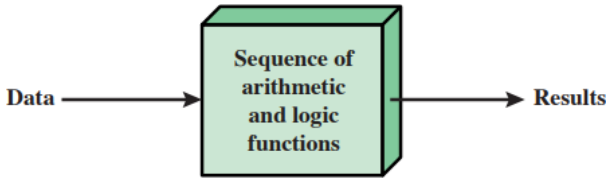
Basic Postulates		
$A \cdot B = B \cdot A$	$A + B = B + A$	Commutative Laws
$A \cdot (B + C) = (A \cdot B) + (A \cdot C)$	$A + (B \cdot C) = (A + B) \cdot (A + C)$	Distributive Laws
$1 \cdot A = A$	$0 + A = A$	Identity Elements
$A \cdot \overline{A} = 0$	$A + \overline{A} = 1$	Inverse Elements
Other Identities		
$0 \cdot A = 0$	$1 + A = 1$	
$A \cdot A = A$	$A + A = A$	
$A \cdot (B \cdot C) = (A \cdot B) \cdot C$	$A + (B + C) = (A + B) + C$	Associative Laws
$\overline{A \cdot B} = \overline{A} + \overline{B}$	$\overline{A + B} = \overline{A} \cdot \overline{B}$	DeMorgan's Theorem

Componenti e connessioni

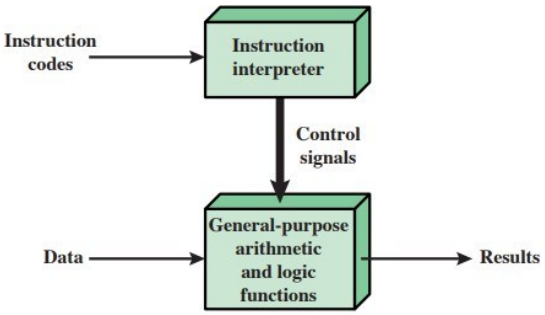
I componenti principali sono:
- La CPU
- La memoria
- Sistemi di I/O
- Connessioni tra loro

Avendo descritto l'architettura di Von Neumann, diciamo che per eseguire un programma, possiamo costruire i componenti logici in modo che il risultato sia quello voluto. Questo è un modo di costruire il programma "cablato", cioè in forma hardware, che non può essere modificato.

È un sistema non flessibile, che può eseguire solo le operazioni predeterminate (accetta dati e produce risultati). Con circuiti generici, accetta dati e segnali di controllo che dicono cosa eseguire, e produce risultati. In generale, per ogni nuovo programma, basta dare i giusti segnali di controllo.



(a) Programming in hardware

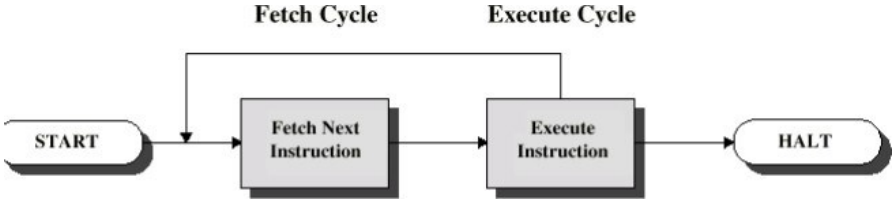


(b) Programming in software

Definiamo quindi un programma: esso non è altro che un insieme di passo in cui, ad ognuno, si ha una operazione logica o aritmetica. Per ogni operazione, si ha un diverso insieme di segnali di controllo.
Con un hardware generico che preleva il codice delle istruzioni e genera appositi segnali di controllo, parte la programmazione software.
La CPU è quindi sia interprete delle istruzioni che modulo per operazioni aritmetico-logiche.

Per la memoria principale:
- Possibilità di salti oltre che esecuzione sequenziale
- Operazioni che richiedono accesso a più dati in memoria
- Immagazzinare temporaneamente sia istruzioni che dati

Come detto, al di là delle varie componenti della CPU, i passi principali del ciclo di esecuzione sono:
- Fetch (reperimento dell'istruzione)
- Execute (esecuzione dell'istruzione)



Importante in particolare è il registro PC (Program Counter): salva l'indirizzo della cella di memoria principale contenente la prossima istruzione. In generale preleva dalla memoria e poi si incrementa.
Esempio:
- parole di M con 16 bit
- PC contiene 300
- CPU preleva l'istruzione nella cella 300, poi 301, poi 302, ... n

Scritto da Gabriel

L'istruzione prelevata viene messa in IR (Instruction Register), poi l'operazione corrispondente viene eseguita.

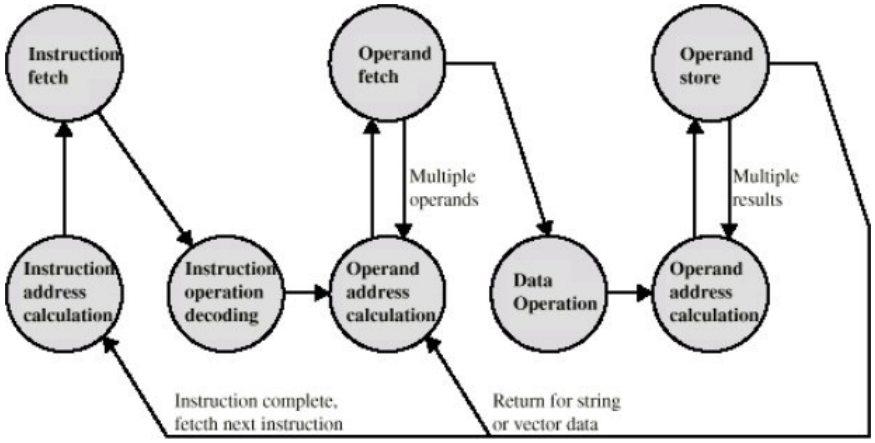
Operazioni di 4 tipi per la CPU

1) Processore-memoria
    a. Trasferimento dati tra la CPU e la M
2) Processore-I/O
    a. Trasferimento dati tra CPU e I/O
3) Elaborazione dati
    a. Operazione logica o aritmetica sui dati n
4) Controllo
    a. Può alterare la sequenza delle istruzioni
    b. Esempio: prelievo istruzione dalla cella 149, che dice che la prossima istruzione è nella cella 182.

Altro esempio:

- Somma di cella 940 e 941 e memorizzazione del risultato nella cella 941
- Tre istruzioni
- All'inizio PC contiene 300
- Celle di M in esadecimale

Il ciclo di esecuzione è così strutturato:



1) Prelievo delle istruzioni
2) Decodifica dell'operazione
3) Calcolo dell'indirizzo dell'operando (ritorna una stringa o un vettore di dati)
4) Prelievo degli operandi e calcolo se necessario per molteplici operandi
5) Operazioni sui dati
6) Calcolo dell'indirizzo dell'operando (con molteplici risultati)
7) Memorizzazione del risultato
8) Istruzione completata e prelievo successiva istruzione

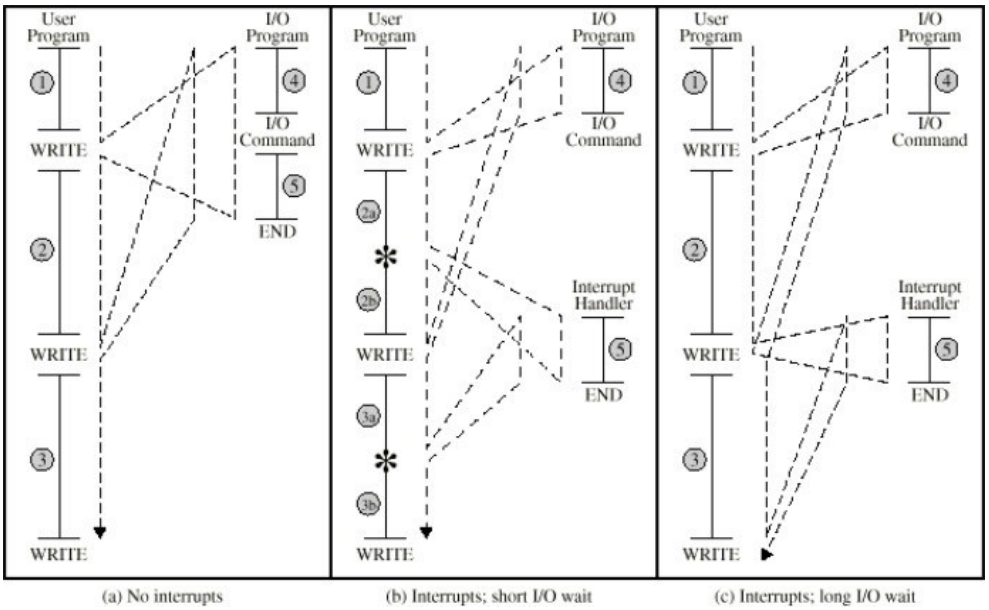
Esiste un meccanismo con cui i moduli (per esempio di I/O) possono interrompere la normale sequenza du esecuzione e queste sono le interruzioni.

Tipiche interruzioni

- Program
    - Esempio: overflow, division by zero
- Timer
    - Generata da un timer interno alla CPU

- I/O
    - Per segnalare la fine di un'operazione di I/O
- Guasto hardware
    - Esempio: mancanza di alimentazione

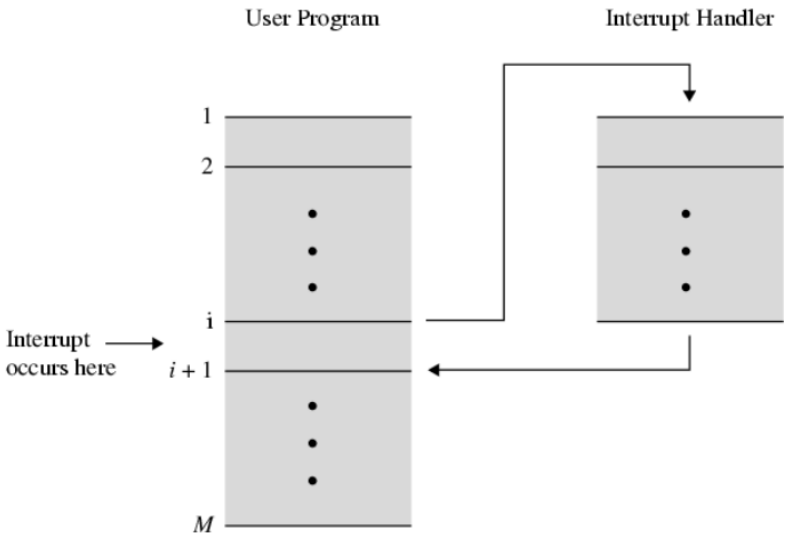
Le interruzioni vengono fatte per migliorare l'efficienza dell'elaborazione (vari dispositivi sono pià lenti del processore e anche per evitare che la CPU attenda la fine di un'operazione di I/O). Qui sotto un esempio:



Trasferimento del controllo per una interruzione

```
USER PROGRAM

⟨statement⟩
⟨statement⟩      Code segment 1
    :
⟨statement⟩

WRITE

⟨statement⟩
⟨statement⟩      Code segment 2
    :
⟨statement⟩

WRITE

⟨statement⟩
⟨statement⟩      Code segment 3
    :
⟨statement⟩
```

```
I/O PROGRAM

⟨statement⟩
⟨statement⟩      Code segment 4
    :
⟨statement⟩

I/O command

⟨statement⟩
⟨statement⟩      Code segment 5
    :
⟨statement⟩
```



Come si nota, esiste l'*interrupt handler* per gestire le interruzioni, è un blocco speciale di codice associato a una specifica condizione di interrupt. I gestori di interrupt vengono avviati da interrupt hardware, istruzioni di interrupt software o eccezioni software e sono utilizzati per implementare i driver dei dispositivi o le transizioni tra modalità operative protette, come le chiamate di sistema.

Ci sono due tipi di interrupt:

1) Interrupt hardware generati da dispositivi esterni alla CPU (periferiche), che hanno il compito di comunicare il verificarsi di eventi esterni, di solito dispositivi di Input/Output. Un interrupt hardware costringe il processore a memorizzare il suo stato di esecuzione fino all'arrivo dell'interrupt e ad iniziare l'esecuzione della subroutine (sottoprogramma) (commutazione di contesto) che esegue il compito richiesto dall'interrupt, terminato il quale il processore riprende l'esecuzione delle operazioni che stava precedentemente elaborando. Nella pratica, si nota un rallentamento del sistema ed un aumento dell'uso della CPU, che può arrivare ad essere impegnata al 100% e per lunghi periodi.
2) Interrupt software: sono delle istruzioni assembly, tipo INT xx o SYSCALL, che possono essere assimilate alle chiamate di sottoprogrammi, ma che sfruttano il meccanismo delle interruzioni per passare il controllo dal programma chiamante a quello chiamato, e viceversa; vengono utilizzati per accedere direttamente alle risorse del sistema operativo.

<u>Uso degli interrupt</u>

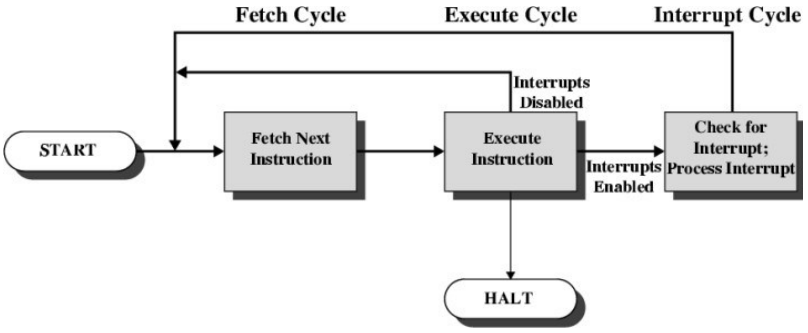
Gli interrupt vengono utilizzati principalmente quando:

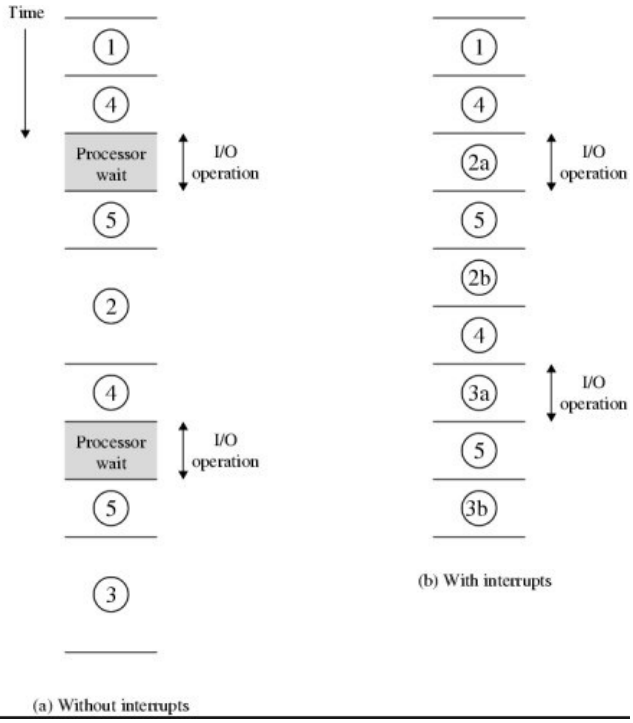
- un processo tenta di eseguire un'istruzione non valida, come una divisione per zero. In questi casi non è possibile proseguire con l'esecuzione del processo, perché genererebbe parecchi errori, corromperebbe i dati nei registri della CPU e/o porterebbe a un crash il sistema stesso. Quindi l'interrupt consente di informare il sistema operativo di quanto avvenuto in modo da permettere la corretta gestione del problema.
- un processo richiede un'operazione di I/O al sistema operativo. Le CPU moderne prevedono la possibilità di utilizzare diversi livelli di privilegi, per ragioni di sicurezza, che i processi in esecuzione possono ricevere. Solo il sistema operativo può effettuare alcune operazioni come accedere ad alcune aree di memoria protette o gestire le periferiche.
- un dispositivo di I/O informa la CPU che è disponibile a ricevere o fornire dati. In questo caso viene avviata un'opportuna procedura del sistema operativo preposta ad occuparsi della relativa periferica. Questo tipo di interrupt necessita una gestione molto attenta, infatti è possibile che due dispositivi abbiano generato un interrupt durante l'esecuzione di un processo, ed è necessario disporre di meccanismi che evitino conflitti e la perdita di informazioni, ad esempio decidendo quale interrupt ha maggiore priorità e deve essere eseguito per primo e ponendo in coda il secondo, delegando il compito al Programmable Interrupt Controller.
- il tempo massimo a disposizione per tale processo è raggiunto e lo scheduler deve riassegnare la CPU ad un altro processo in coda.

- viene effettuato il debugging di un'applicazione. Durante la fase di sviluppo di un programma è frequente la necessità di testare il funzionamento di quanto creato per scoprire e risolvere l'origine dei malfunzionamenti. Il debugging consente di seguire l'evoluzione del programma istruzione per istruzione, dando la possibilità di interrompere il processo in qualunque momento per verificare il valore di ogni parametro. Per effettuare questo è necessario che il codice sorgente sia compilato in maniera apposita, in questo caso se il programma viene eseguito sotto il controllo di un apposito programma, il debugger, ad ogni istruzione viene eseguito un interrupt che consente di verificare se in tale punto è richiesta l'interruzione del processo.

<u>Ciclo di interruzione</u>

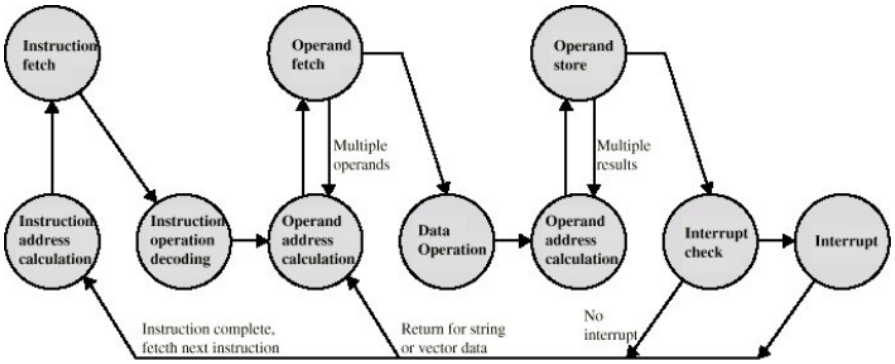
- Aggiunto al ciclo di esecuzione
- La CPU controlla se ci sono interruzioni pendenti
- Se no, prende la prossima istruzione
- Se sì:
    - Sospende l'esecuzione del programma corrente
    - Salva il contesto (es.: indirizzo prossima istruzione)
    - Imposta il PC all'indirizzo di inizio del programma di gestione dell'interruzione
    - Esegue il programma di gestione dell'interruzione
    - Rimette il contesto al suo posto e continua il programma interrotto





Le interruzioni eliminano le attese durante le operazioni di I/O, specie se queste sono molto lunghe. Per esempio, è tipicamente il caso delle operazioni di scrittura WRITE, come segue a lato:

Il completamento dell'esecuzione, graficamente, aggiungendo anche gli interrupt diventa:



Nella gestione delle interruzioni multiple

L'interrupt multiplo è un evento di interrupt che può verificarsi mentre il processore sta gestendo un interrupt precedente.

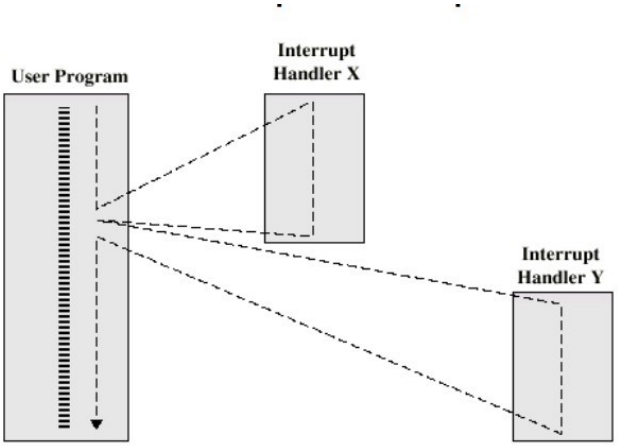
Ad esempio, se un programma riceve dati da una linea di comunicazione e stampa i risultati, è possibile che si verifichi un interrupt di comunicazione mentre viene elaborato l'interrupt della stampante.

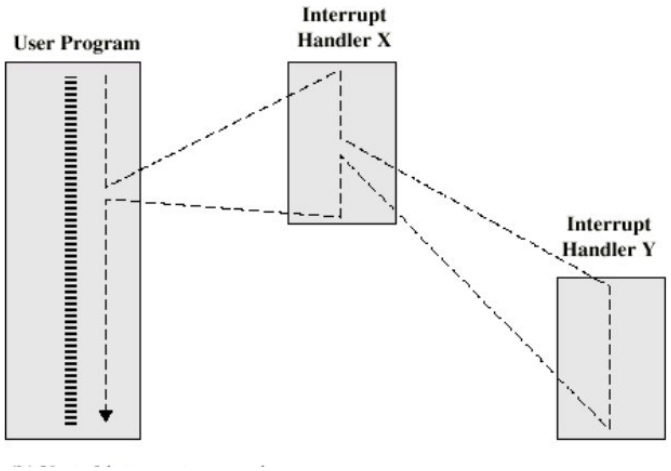
Modalità di gestione degli interrupt
Il processore può gestire gli interrupt in due modi

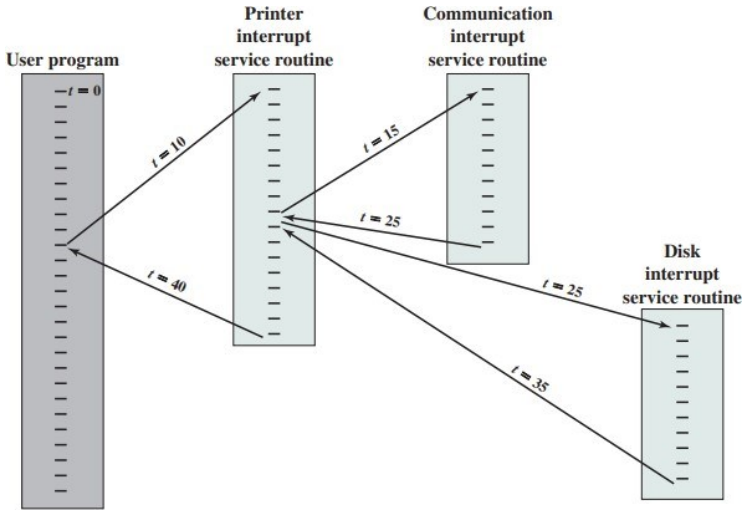
1) Disabilitazione dell'interrupt - Il processore ignora ulteriori interrupt mentre ne sta elaborando uno. Le interruzioni rimangono in sospeso e vengono controllate dopo che il primo interrupt è stato gestito. In questo modo gli interrupt vengono gestiti in sequenza.

2) Definire le priorità - In questo metodo gli interrupt a bassa priorità possono essere interrotti da quelli a priorità più alta. In questo caso, l'interrupt ad alta priorità viene gestito e il processore torna all'interrupt precedente su cui stava lavorando.

Le immagini riportano una gestione delle interruzioni multiple sequenzialmente (sx) e annidate (dx)







Qui invece un esempio reale:
- Programma utente in esecuzione
- Essa viene passata al servizio di stampa che interrompe
- Poi si passa alla routine di comunicazione di interruzione
- Si passa anche attraverso il disco per gestirla

In merito invece alle connessioni, tutte le componenti di un calcolatore devono essere connesse e ci sono tipi diversi di connessione per diversi tipi di componente.
Esse sono fondamentali:

- Nella memoria permettono di leggere/scrivere indirizzi e gestire dati, dando in output dati
- Nei moduli I/O si gestiscono dati interni ed esterni in lettura/scrittura ed indirizzi, dando nuovi dati e segnali di interruzione
- Per la CPU vi sono dati, segnali di interruzione e istruzioni in input e in output dati, indirizzi e segnali di controllo

Connessioni per la memoria

- Riceve e spedisce dati (scrittura e lettura)
- Riceve indirizzi (di locazioni di M)
- Riceve segnali di controllo
    - Lettura
    - Scrittura

Connessioni dell' Input/Output

- Modulo di I/O: simile ad una memoria dal punto di vista della CPU
- Operazioni di Output
    - Riceve dati dalla CPU
    - Manda dati alle periferiche
- Operazioni di Input
    - Riceve dati dalle periferiche
    - Manda dati alla CPU
- Riceve segnali di controllo dalla CPU
- Manda segnali di controllo alle periferiche
- Riceve indirizzi dalla CPU (n.ro di porta per identificare una periferica)
- Manda segnali di interruzione

Connessioni:

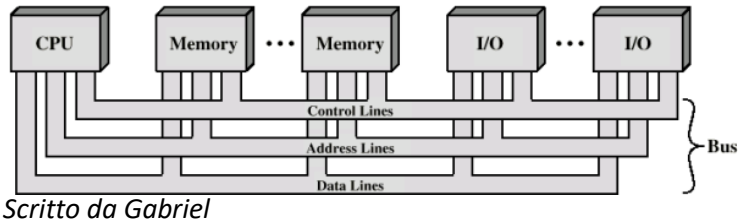
- Da M a CPU: la CPU legge un'istruzione o un dato dalla M
- Da CPU a M: la CPU scrive un dato in M
- Dall'I/O alla CPU: la CPU legge i dati da una periferica
- Dalla CPU all'I/O: la CPU invia dati ad una periferica
- Dall'I/O alla M o viceversa: accesso diretto alla M da parte di un dispositivo di I/O

Connessioni per la CPU

- Legge istruzioni e dati
- Scrive dati (dopo l'elaborazione)
- Manda segnali di controllo alle altre unità
- Riceve segnali di interruzione

Bus

La funzione principale del BUS è quella di interconnettere due o più dispositivi, quali la CPU, la memoria centrale e le interfacce verso dispositivi periferici (I/O, memoria di massa, etc.). Di fatto, esso collega due unità funzionali alla volta: una trasmette e l'altra riceve. Il trasferimento avviene sotto il controllo della CPU. Un segnale trasmesso da uno dei dispositivi collegati ad un bus è disponibile a tutti gli altri.
Solo un dispositivo alla volta può trasmettere, altrimenti i segnali si sovrappongono. Normalmente, in più linee di comunicazione, ogni linea trasmette uno 0 o un 1 n Insieme, più linee trasmettono in parallelo numeri binari (esempio: dato da 8 bit tramesso in parallelo da un bus a 8 bit).
Qui lo schema di interconnessione dei bus:



Scritto da Gabriel

Esistono vari tipi di bus:

Bus di sistema
- Connette CPU, I/O, M
- Da 50 a qualche centinaio di linee (ampiezza del bus)
- Tre gruppi di linee
    - Dati: su cui viaggiano i dati (bus dati)
    - Indirizzi
    - Controllo

Bus dati
- Trasporta i dati (o le istruzioni)
- L'ampiezza è importante per l'efficienza del sistema
    - Se poche linee, più accessi in M per prendere un dato

Bus indirizzi
- Indica la sorgente o la destinazione dei dati
    - Es.: la CPU vuole leggere un dato dalla M
- L'ampiezza determina la massima quantità di M indirizzabile

Bus di controllo
- Per controllare accesso e uso delle linee dati e indirizzi
    - M write: scrittura dei dati sul bus alla locazione di M
    - M read: mette sul bus i dati della locazione di M
    - Richiesta bus: un modulo vuole il controllo del bus
    - Bus grant: è stato concesso il controllo ad un modulo
    - Interrupt request: c'è una interruzione pendente
    - Clock: per sincronizzare le operazioni

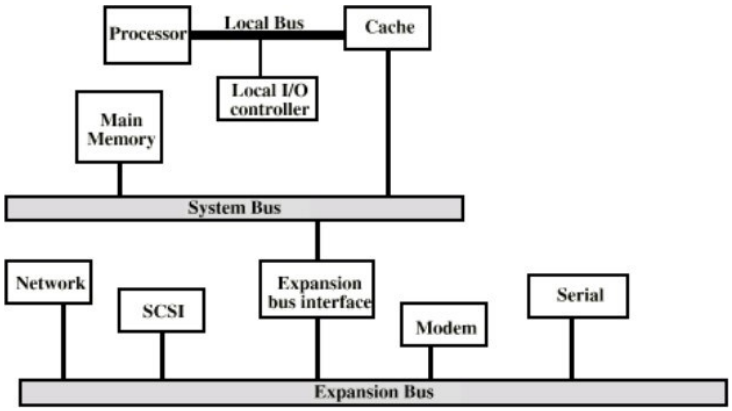
Uso del bus
- Se un modulo vuole inviare dati ad un altro, deve:
    - Ottenere l'uso del bus
    - Trasferire i dati sul bus
- Se un modulo vuole ricevere dati da un altro modulo, deve:
    - Ottenere l'uso del bus
    - Trasferire una richiesta all'altro modulo sulle linee di controllo
    - Attendere l'invio dei dati

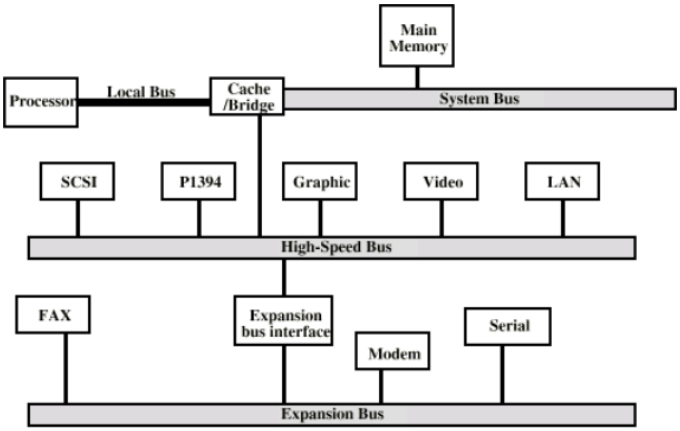
Se esiste un solo bus, si può avere ritardo e congestione. Normalmente, ovviamente, si cerca di usare bus multipli per risolvere i vari problemi.
In un'architettura a bus singolo, tutti i componenti, tra cui l'unità di elaborazione centrale, la memoria e le periferiche, condividono un bus comune. Quando molti dispositivi hanno bisogno del bus allo stesso tempo, si crea uno stato di conflitto chiamato contesa del bus: alcuni aspettano il bus mentre un altro ne ha il controllo. L'attesa fa perdere tempo e rallenta il computer, come spiega Engineering 360. I bus multipli permettono a più dispositivi di lavorare contemporaneamente, riducendo i tempi di attesa e migliorando la velocità del computer. I miglioramenti delle prestazioni sono la ragione principale per la presenza di più bus nel progetto di un computer.

La disponibilità di più bus offre una maggiore scelta per il collegamento dei dispositivi al computer, poiché i produttori di hardware possono offrire lo stesso componente per più di un tipo di bus. Come sottolinea Digital Trends, la maggior parte dei PC desktop utilizza l'interfaccia Serial Advanced Technology Attachment per le unità disco interne, ma molte unità disco esterne e unità flash si collegano tramite USB. Se le connessioni

SATA del computer sono tutte utilizzate, l'interfaccia USB consente di collegare ulteriori dispositivi di archiviazione.

Come per tutti i componenti di un computer, i design dei bus si evolvono, con l'introduzione di nuovi tipi ogni pochi anni. Ad esempio, il bus PCI che supporta le schede video, di rete e altre schede di espansione è precedente alla più recente interfaccia PCIe, mentre l'USB è stato sottoposto a diverse revisioni. La presenza di più bus che supportano apparecchiature di epoche diverse consente di mantenere le apparecchiature tradizionali, come le stampanti e i dischi rigidi più vecchi, e di aggiungere anche dispositivi più recenti.





Chiaramente, gli eventi sui bus vanno coordinati per mezzo della temporizzazione.

- Coordinazione degli eventi su un bus
- Sincrona
    - Eventi determinati da un clock
    - Una linea di clock su cui viene spedita una sequenza alternata di 0 e 1 di uguale durata
    - Una singola sequenza 1-0 è un ciclo di clock
    - Tutti i dispositivi connessi al bus possono leggere la linea di clock
    - Tutti gli eventi partono dall'inizio di un ciclo di clock

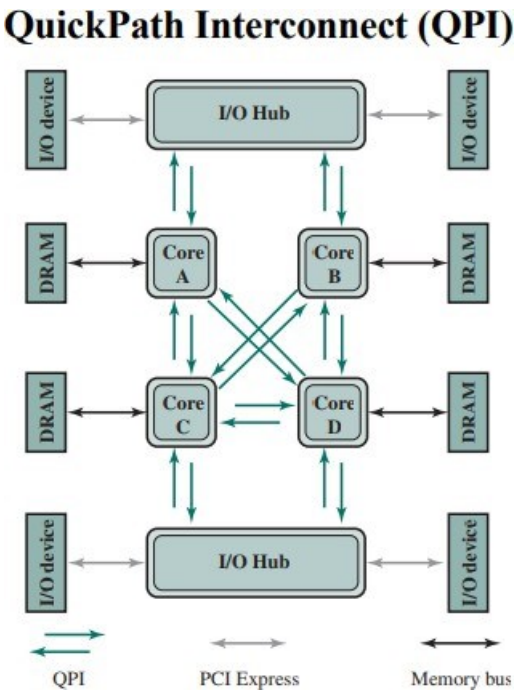
QuickPath Interconnect (QPI)

Anche se l'architettura Core è stata straordinariamente efficiente, alcuni dettagli della progettazione hanno iniziato a mostrare la loro età, primo fra tutti il Front Side Bus (FSB). Questo bus, che connette i processori al Northbridge, è stato la nota dolente di un'architettura moderna. Il difetto era maggiormente visibile in configurazione multiprocessore, dove l'architettura faticava nella gestione dei crescenti carichi di lavoro. I processori dovevano condividere il bus, non solo per accedere alla memoria, ma anche per assicurare la coerenza dei dati contenuti nelle rispettive memorie cache.

In questo tipo di situazione, il flusso di transazioni poteva velocemente saturare il bus. Per lungo tempo Intel ha semplicemente lavorato attorno al problema usando un bus più veloce o memorie cache più ampie, ma Nehalem è l'opportunità di risolvere il problema alla radice, rivedendo completamente il modo in cui i processori comunicano con la memoria e i componenti esterni.

La soluzione scelta da Intel – chiamata QuickPath Interconnect (QPI) – non è nuova; un controller di memoria integrato è un bus seriale punto-punto veramente veloce.
Da un punto di vista tecnico, il collegamento QPI è bidirezionale e ha due collegamenti a 20 bit – uno in ogni direzione – di cui 16 sono riservati per i dati; gli altri quattro sono usati per i codici di error detection o funzioni di protocollo. Il bus QPI gestisce un massimo di 6.4 GT/s (miliardi di trasferimenti al secondo) o un bandwidth di 12.8 GB/s, sia in lettura che in scrittura.

Un collegamento QPI ha perciò un bandwidth teorico fino a due volte maggiore, fermo restando letture e scritture ben bilanciate.



- Connessioni dirette multiple: più componenti all'interno del sistema godono di connessioni dirette a coppie con altri componenti
- Architettura di protocollo a strati/livelli (layer): come si trova negli ambienti di rete
- Trasferimento dati a pacchetto: i dati non vengono inviati come flusso di bit non elaborato ma inviati come una sequenza di pacchetti, ognuno dei quali include intestazioni di controllo e codici di controllo degli errori.

<u>Livelli QPI</u>

- Fisico: è costituito dai cavi, circuiti e logica per supportare le funzioni ausiliarie per trasmissione e ricezione bit. L'unità di trasferimento a livello fisico è di 20 bit, chiamata Phit (unità fisica).
- Link: responsabile della trasmissione affidabile e del controllo del flusso. L'unità di trasferimento è un Flit (unità di controllo del flusso) a 80 bit.
- Routing: fornisce la struttura per dirigere i pacchetti attraverso la struttura.
- Protocollo: l'insieme di regole di alto livello per lo scambio di pacchetti di dati tra dispositivi. Un pacchetto è composto da un numero intero di 104 Flits.

In particolare:

- Il livello fisico è composto da una serie di percorsi di trasmissione e ricezione, tale che si realizzi una distribuzione multilinea
- Il livello link, che usa un protocollo con pacchetti da 72 (dati/messaggi) + 8 (codice correzione di errore) bit
    - Due funzioni:
        - controllo del flusso: evita che il mittente invii più dati di quanti il destinatario possa ricevere (sistema a crediti)
        - controllo dell'errore: 8 bit sono utilizzati per rilevare errori di trasmissione sui 72 bit di dati/messaggi (vedremo in seguito come funziona); in caso di errore il mittente deve re-inviare il pacchetto con l'errore (e altri successivamente inviati)
- Il livello routing (instradamento), che determina il percorso che un pacchetto deve seguire all'interno del sistema. Supportato da:
    - Tabelle di instradamento:
        - definite dal firmware;
        - descrivono i possibili percorsi che un pacchetto può seguire;
        - utile soprattutto in sistemi di dimensione maggiore;
- Il livello protocollo, con un pacchetto definito come unità di trasferimento
    - Caratteristiche:
        - definizione contenuto pacchetto flessibile, in modo da coprire esigenze diverse;
        - supporta protocollo di coerenza della cache, in modo da garantire coerenza fra i contenuti delle cache dei core e la memoria principale

Gerarchie di memoria

Le caratteristiche principali delle memorie sono così condensate:

- Locazione: processore, interna (principale), esterna (secondaria)
- Capacità: dimensione parola, numero di parole
- Unità di trasferimento: parola, blocco
- Metodo di accesso: sequenziale, diretto, casuale, associativo
- Prestazioni: tempo di accesso, tempo di ciclo, velocità trasferimento
- Modello fisico: a semiconduttore, magnetico, ottico, magneticoottico
- Caratteristiche fisiche: volatile/non volatile, riscrivibile/non riscrivibile
- Organizzazione

L'ideale in termini di memoria è disporre di una memoria molto ampia, molto veloce e molto economica. Graficamente, si ha questo confronto:

Tecnologia
registro
cache
SRAM
DRAM
disco
CD/DVD-ROM [meno capace di disco!]
nastro

Le CPU hanno avuto un aumento di prestazioni notevole, dovuto ad innovazioni tecnologiche ed architetturali. Le memorie sono migliorate solo grazie agli avanzamenti tecnologici; chiaramente, con esse sono aumentate sempre di pià le esigenze di memoria da parte dei singoli programmi.
Essi hanno le seguenti proprietà:

- Proprietà statiche (dal file sorgente)
- Proprietà dinamiche (dall'esecuzione)
    - Linearità dei riferimenti
        - Gli indirizzi acceduti sono spesso consecutivi
- Località dei riferimenti
    - Località spaziale
        - Gli accessi ad indirizzi contigui sono più probabili
    - Località temporale
        - La zona di accesso più recente è quella di permanenza più probabile

Da qui, *la congettura 90/10,* per cui "Un programma impiega mediamente il 90% del suo tempo di esecuzione alle prese con un numero di istruzioni pari a circa il 10% di tutte quelle che lo compongono".

Conviene quindi organizzare la memoria su più livelli gerarchici:

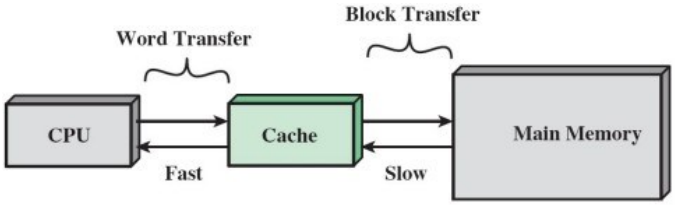
- Livello 1 (cache): molto veloce e molto costosa → dimensioni ridotte, per i dati ad accesso più

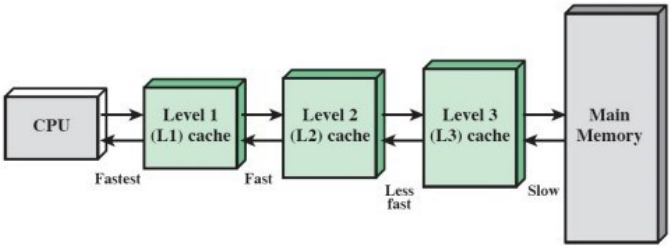
probabile [anche più livelli di cache]

- Livello 2 (memoria centrale): molto ampia e lenta → costo contenuto, per tutti i dati del

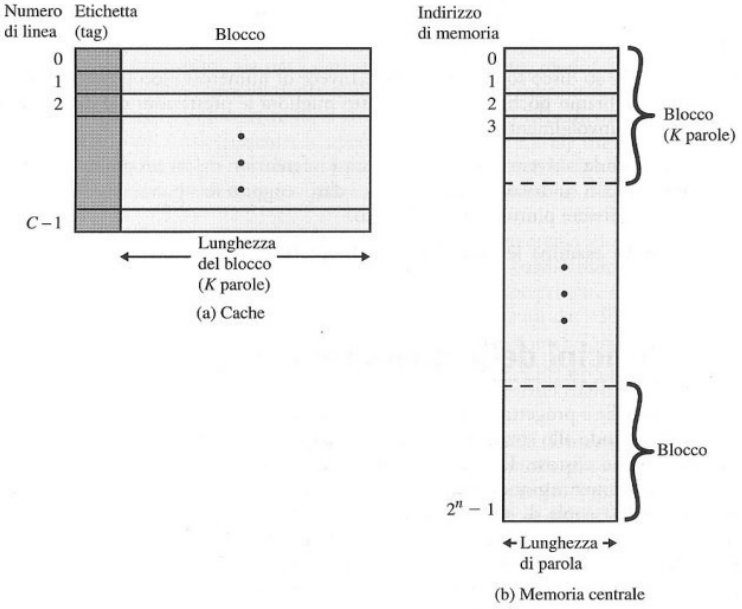
programma

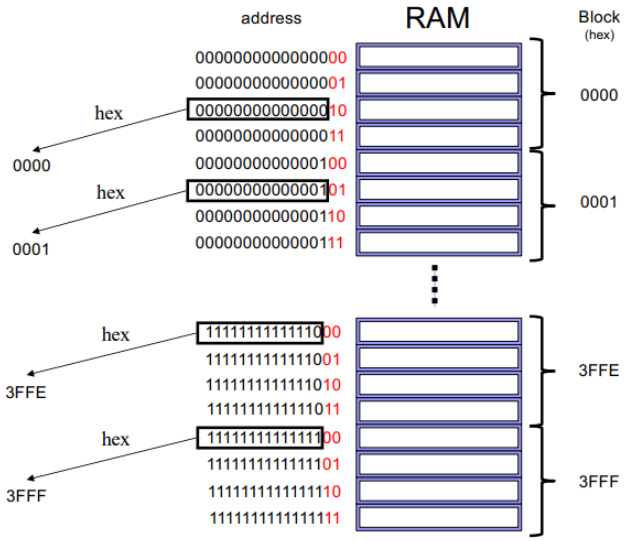
Al livello più basso (inferiore) stanno i "supporti di memoria" più capaci, più lenti e meno costosi, mentre ai livelli più alti (superiori) si pongono supporti più veloci, più costosi e meno capaci. La CPU usa direttamente il livello più alto ed ogni livello inferiore deve contenere tutti i dati presenti ai livelli superiori (ed altri).





Per realizzare un'organizzazione gerarchica conviene suddividere la memoria in <u>blocchi</u>. La *dimensione* di un blocco è la quantità minima indivisibile di dati che occorre prelevare (copiare) dal livello inferiore. *L'indirizzo* di un dato diviene l'indirizzo del blocco che lo contiene sommato alla posizione del dato all'interno del blocco. Due esempi, rispettivamente per cache e per RAM, per cui normalmente si cerca di avere una memoria ad accesso veloce (cache) che, se non dispone dei dati utili, se li cerca nel disco e memorizza continuativamente una serie di dati scambiati direttamente a stretto contatto con la RAM.





Un dato richiesto dalla CPU può essere presente in cache (<u>hit</u>) oppure mancante (<u>miss</u>)

- Un *hit*, successo, deve essere molto probabile (>90%) se si vuole guadagnare efficienza prestazionale
- Un *miss*, fallimento, richiede l'avvio di una procedura di scambio dati (swap) con il livello inferiore

Il tempo medio di accesso è così misurato:

T_a : Tempo medio di accesso ad un dato in memoria

$$T_a = T_h \times P_h + T_m \times (1 - P_h)$$

T_h = tempo di accesso ad un dato **presente** in cache

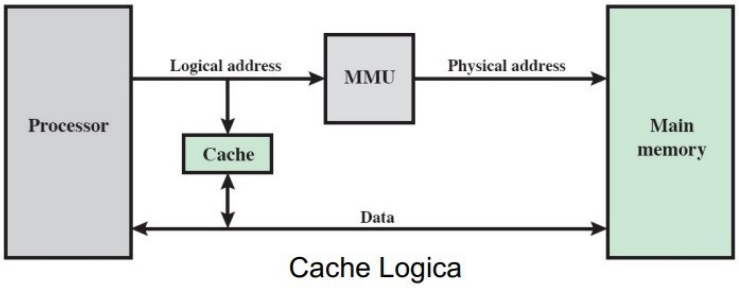
T_m = tempo medio di accesso ad un dato **non** in cache
(funzione della dimensione del blocco)

P_h = probabilità di **hit**
(funzione della dimensione del blocco **e** della politica di gestione)

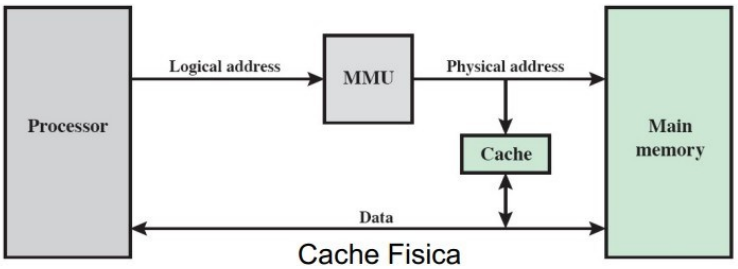
La tecnica generale, comunque:

- Suddivisione della memoria centrale in blocchi logici
- Dimensionamento della cache in multiplo di blocchi
- Per ogni indirizzo emesso dalla CPU
  - Hit → Il dato richiesto viene fornito immediatamente alla CPU
  - Miss → La cache richiede il dato al livello inferiore, il blocco contenente il dato viene posto in cache ed il dato richiesto viene fornito alla CPU

La memoria cache può essere collocata su entrambi i lati di un'unità di gestione della memoria e utilizzare indirizzi fisici o logici come dati di tag. In termini di prestazioni, la posizione della cache può influire notevolmente sulle prestazioni del sistema. In una cache *logica*, le informazioni di tag si riferiscono agli indirizzi logici attualmente in uso dal task in esecuzione.



Cache Logica



Cache Fisica

Se l'attività viene interrotta durante un cambio di contesto, i tag della cache non sono più validi e la cache, insieme ai suoi dati spesso conquistati con fatica, deve essere svuotata e cancellata. Il processore deve andare alla memoria principale per recuperare le prime istruzioni e attendere la seconda iterazione prima di ottenere qualsiasi beneficio dalla cache.

Le cache *fisiche* utilizzano indirizzi fisici, non necessitano di flush su un cambio di contesto e quindi i dati vengono conservati all'interno della cache. Lo svantaggio è che tutti gli accessi devono passare attraverso

l'unità di gestione della memoria, con conseguenti ritardi. Occorre prestare particolare attenzione anche quando le pagine vengono scambiate da e verso il disco.
Se il processore non invalida le voci della cache associate, il contenuto della cache sarà diverso da quello della memoria principale a causa della nuova pagina che è stata scambiata.

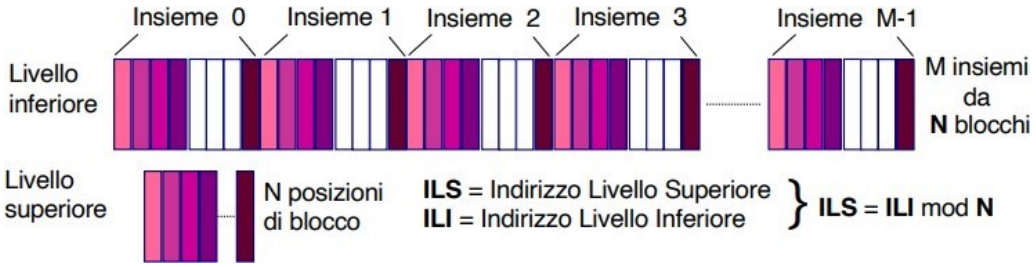
Problematiche
- Organizzazione della cache e tecniche di allocazione
- Individuazione di hit o miss
- Politica di rimpiazzo dei blocchi
- Congruenza dei blocchi

Memorizzazione ed organizzazione (Mapping della cache)

Esistono tre diversi tipi di mappatura utilizzati per la memoria cache.

1) Mappatura diretta/Direct Mapping

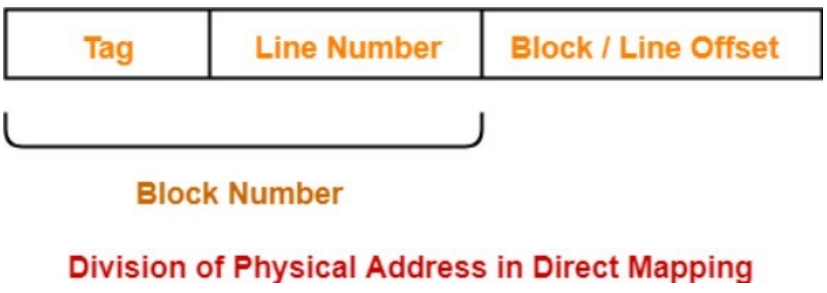
La tecnica più semplice, nota come mappatura diretta, mappa ogni blocco di memoria principale in una sola possibile linea di cache. Nella mappatura diretta, ogni blocco di memoria viene assegnato a una linea specifica della cache. Se una riga è stata precedentemente occupata da un blocco di memoria, quando deve essere caricato un nuovo blocco, il vecchio blocco viene cestinato. Uno spazio di indirizzi è diviso in due parti: un campo indice e un campo tag. La cache viene utilizzata per memorizzare il campo tag, mentre il resto viene memorizzato nella memoria principale. Le prestazioni della mappatura diretta sono direttamente proporzionali alla possibilità di avere una Hit.



Il numero di riga della cache in cui può essere mappato un determinato blocco è dato da:

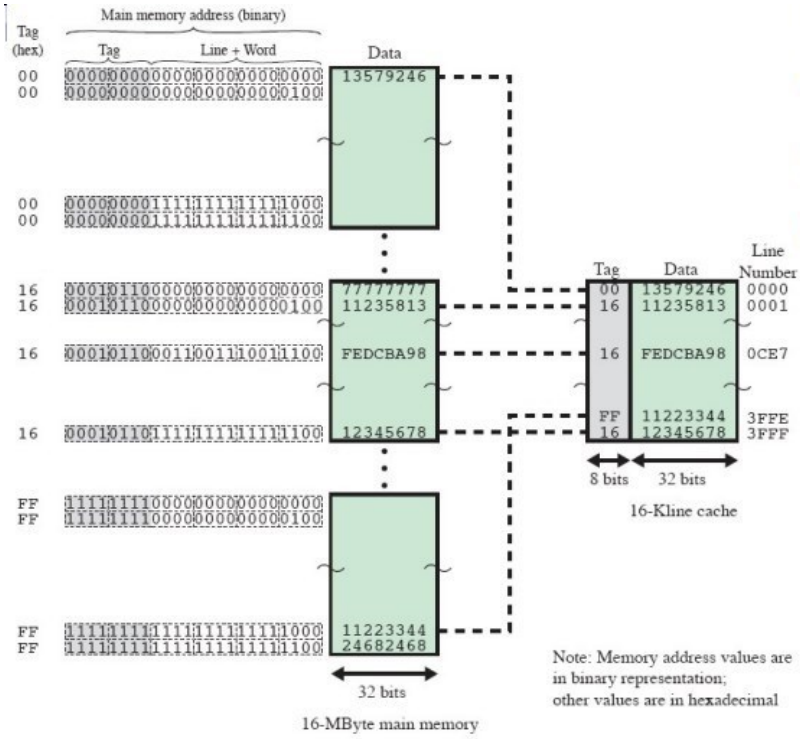
$$Numero\ di\ linea\ della\ cache = \\ (\ Indirizzo\ del\ blocco\ di\ memoria\ principale\ )\ mod\ (Numero\ di\ righe\ nella\ cache)$$

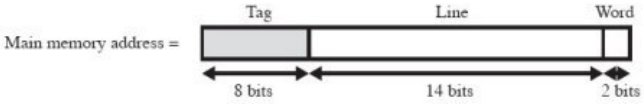
Ai fini dell'accesso alla cache, ogni indirizzo di memoria principale può essere considerato come composto da tre campi. I bit w meno significativi identificano una parola o un byte unico all'interno di un blocco di memoria principale. Nella maggior parte delle macchine moderne, l'indirizzo è a livello di byte. I restanti bit s specificano uno dei 2^s blocchi di memoria principale. La logica della cache interpreta questi bit s come un tag di $s - r$ bit (porzione più significativa) e un campo di riga di r bit. Quest'ultimo campo identifica una delle $m = 2r$ linee della cache.

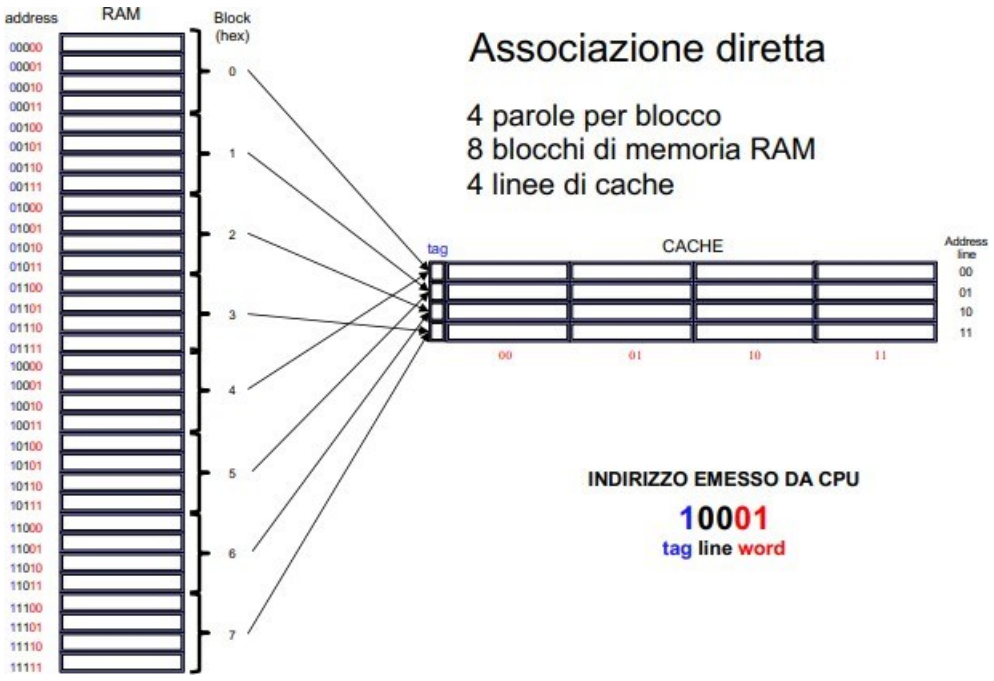


Division of Physical Address in Direct Mapping

Esempio di associazione diretta







I passaggi seguenti spiegano il funzionamento della cache a mappatura diretta.
La CPU genera una richiesta di memoria:

- Il campo del numero di riga dell'indirizzo viene utilizzato per accedere a una particolare riga della cache.
- Il campo tag dell'indirizzo della CPU viene quindi confrontato con il tag della riga.
- Se i due tag corrispondono, si verifica una cache hit e la parola desiderata viene trovata nella cache.
- Se i due tag non corrispondono, si verifica una cache miss.
- In caso di cache miss, la parola desiderata deve essere prelevata dalla memoria principale.
- Viene quindi memorizzata nella cache insieme al nuovo tag che sostituisce quello precedente.

Vantaggi

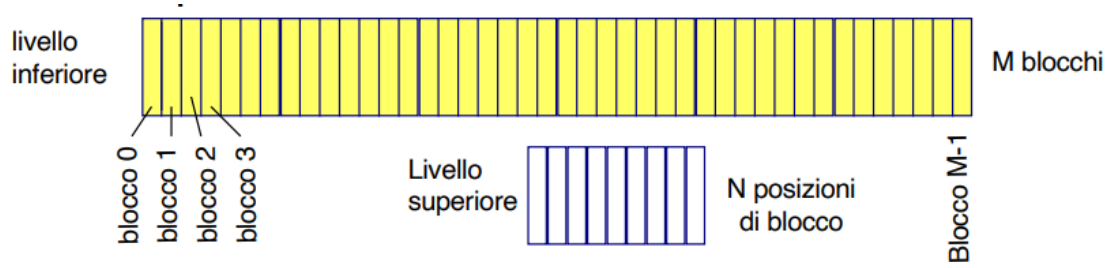
- Semplicità di traduzione da indirizzo ILI (memoria) ad indirizzo ILS (cache)
- Determinazione veloce di hit o miss

Svantaggi

- Necessità di contraddistinguere il blocco presente in ILS (introduzione di un'etichetta, 'tag')
- Swap frequenti per accesso a dati di blocchi adiacenti

2) Associative mapping/Mapping associativo

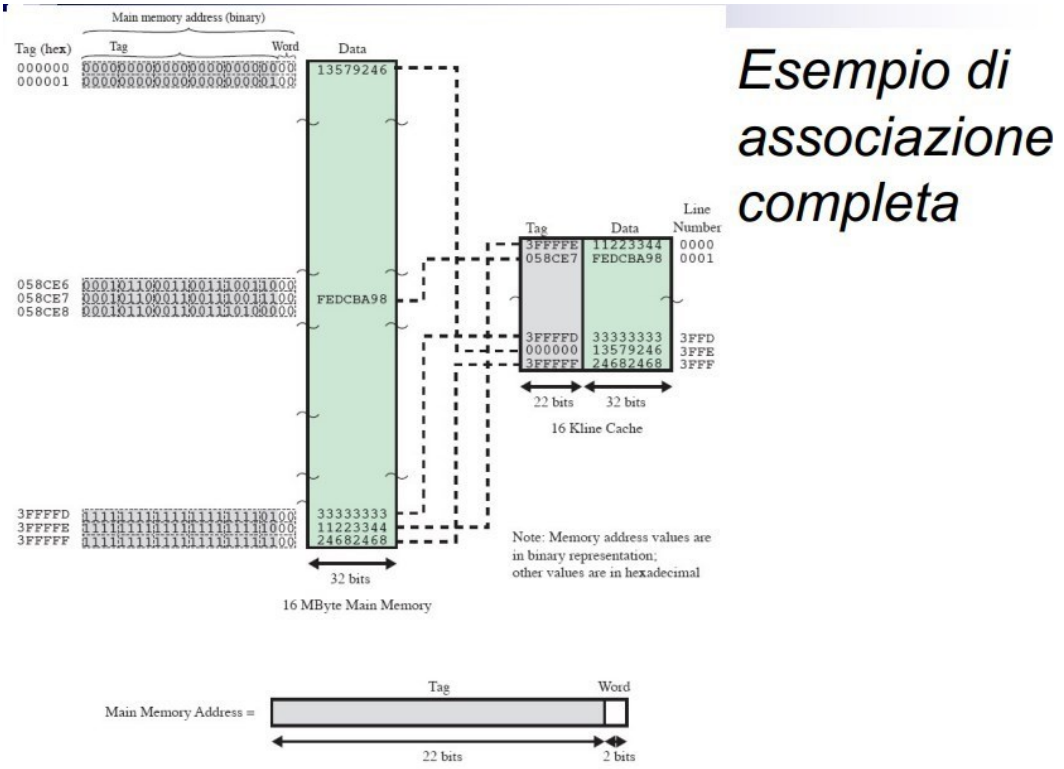
In questo tipo di mappatura, la memoria associativa viene utilizzata per memorizzare il contenuto e gli indirizzi delle parole di memoria. Qualsiasi blocco può essere inserito in qualsiasi riga della cache. Ciò significa che i bit dell'id della parola vengono utilizzati per identificare quale parola del blocco è necessaria, ma il tag diventa tutti i bit rimanenti. Ciò consente di collocare qualsiasi parola in qualsiasi punto della memoria cache. È considerata la forma di mappatura più veloce e flessibile.
Ogni blocco del livello inferiore può essere posto in qualunque posizione del livello superiore.



Questo metodo di ricerca di un blocco all'interno della cache sembra essere un processo lento, ma non è così. Ogni riga della cache ha il suo circuito di confronto, che può analizzare rapidamente se il blocco è contenuto o meno in quella riga. Poiché tutte le linee eseguono questo processo di confronto in parallelo, la linea corretta viene identificata rapidamente.

Questa tecnica di mappatura è stata progettata per risolvere un problema che esiste con la mappatura diretta, in cui due blocchi di memoria attivi possono essere mappati sulla stessa riga della cache. In questo caso, nessuno dei due blocchi di memoria può rimanere nella cache perché viene sostituito rapidamente dal blocco concorrente. Questo porta a una condizione che viene definita *thrashing*.

Nel thrashing, una riga della cache va avanti e indietro tra due o più blocchi, di solito sostituendo un blocco prima ancora che il processore lo abbia superato. Il thrashing può essere evitato consentendo la mappatura di un blocco di memoria su qualsiasi linea della cache. È necessario un algoritmo di sostituzione per sostituire un blocco se la cache è piena.



Alla cache capace di N blocchi viene associata una tabella di N posizioni, contenenti il numero di blocco effettivo (tag) in essa contenuto.

Vantaggi

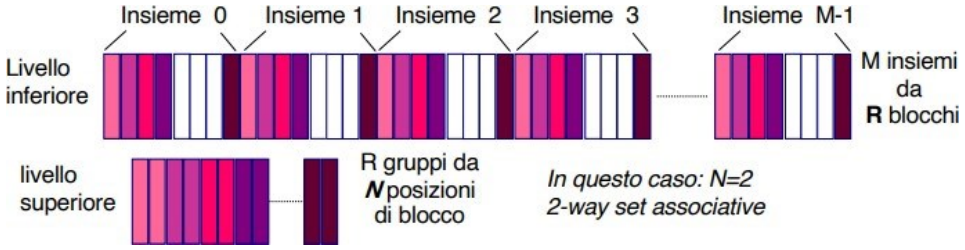
- Massima efficienza di allocazione

Svantaggi

- Determinazione onerosa della corrispondenza ILS-ILI e della verifica di hit/mis

3) <u>Associazione a gruppi / N-way set associative</u>

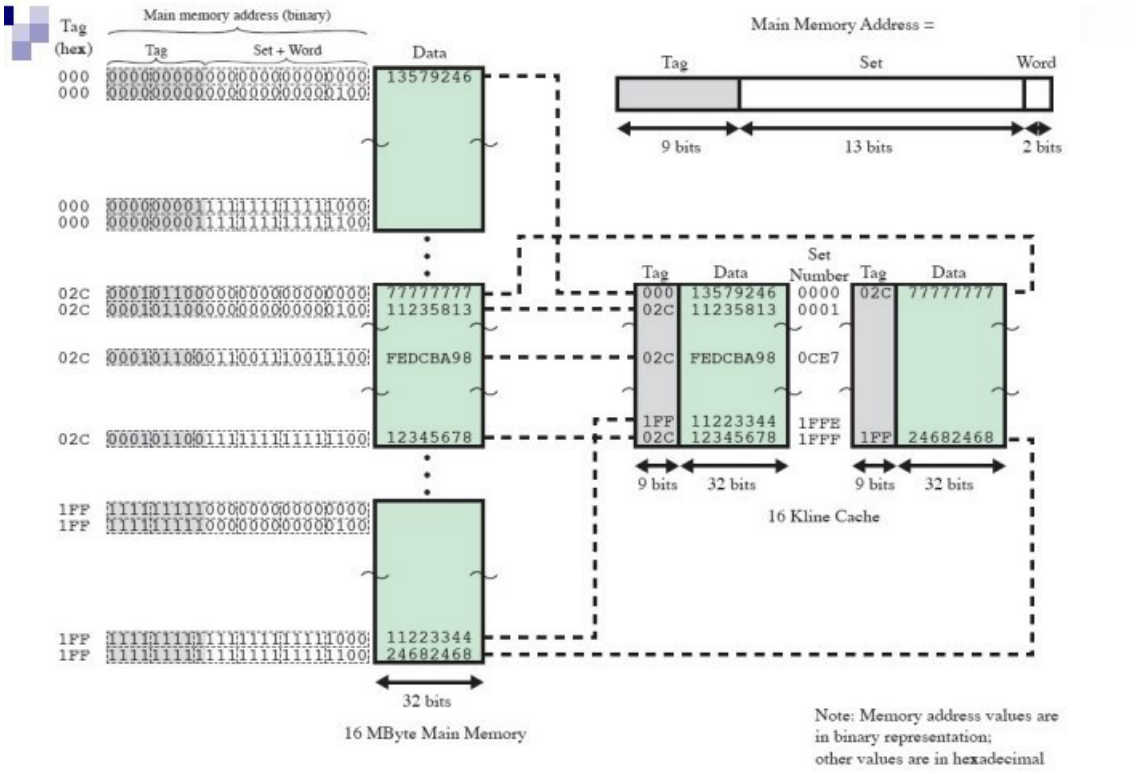
Ogni blocco di un certo insieme di blocchi del livello inferiore può essere allocato liberamente in uno specifico gruppo di blocchi del livello superiore. Questa forma di mappatura è una forma migliorata di mappatura diretta, in cui gli svantaggi della mappatura diretta vengono eliminati. Il set associativo risolve il problema del possibile thrashing del metodo di mappatura diretta. A tal fine, invece di avere esattamente una riga a cui un blocco può fare da mappatura nella cache, si raggruppano alcune righe creando un insieme. La mappatura associativa a set permette che ogni parola presente nella cache possa avere due o più parole nella memoria principale per lo stesso indirizzo di indice. La mappatura associativa a set combina il meglio delle tecniche di mappatura diretta e associativa della cache.



Un indirizzo di cache può essere suddiviso in 3 parti.

- l'offset all'interno del blocco
- l'indice che identifica l'insieme
- il tag che identifica il blocco nell'insieme.

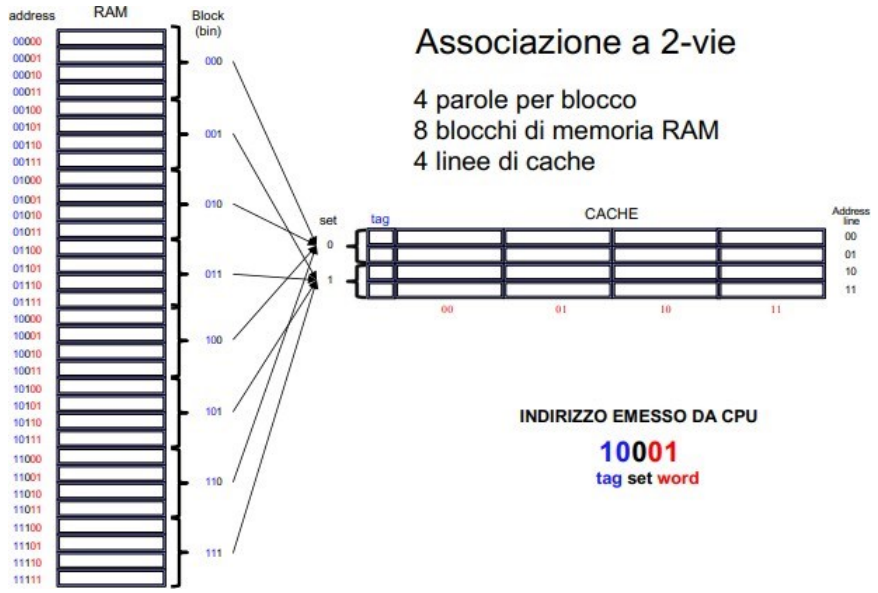
Quando arriva una richiesta, viene calcolato l'indice per identificare l'insieme. Poi vengono controllati i tag di tutti i blocchi dell'insieme. Quando viene trovato un blocco con un tag corrispondente, vengono restituiti i byte giusti in base all'offset.
Una cache a mappatura diretta è di fatto una cache associativa a 1 via.
Il grande vantaggio di una cache associativa a n vie rispetto a una cache a mappatura diretta è che quest'ultima può contenere un solo blocco per un insieme di indirizzi, mentre la prima può contenere più blocchi per un insieme di indirizzi.

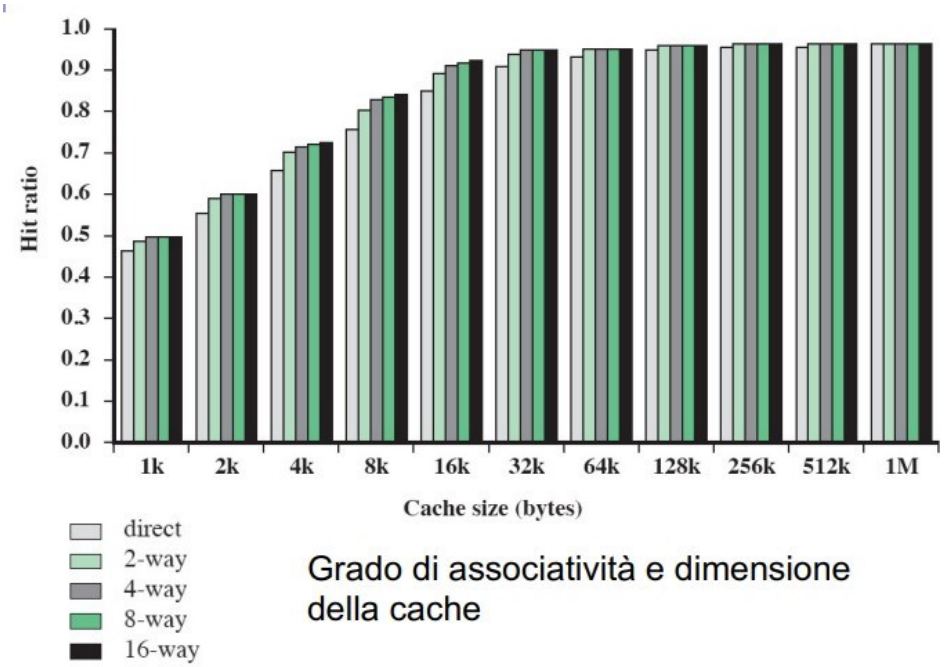


Esempio di associazione a gruppi (N=2)

Alla cache, composta da R gruppi di N posizioni di blocco ciascuno, si affiancano R tabelle di N elementi, contenenti le etichette (tag) che designano i blocchi effettivi posti nelle posizioni corrispondenti

- Valutazione: buona efficienza di allocazione a fronte di una sopportabile complessità di ricerca





Politiche di rimpiazzo dei blocchi

Quale blocco conviene sostituire in cache per effettuare uno swap?
(Penalità di miss)

- Casuale, per occupazione omogenea dello spazio
- First-In-First-Out (FIFO), per sostituire il blocco rimasto più a lungo in cache
- Least Frequently Used (LFU), per sostituire il blocco con meno accessi

P(miss)		rimpiazzo casuale			rimpiazzo LRU		
	N-way	2	4	8	2	4	8
Ampiezza cache	16 KB	5,69	5,29	4,96	5,18	4,67	4,39
	64 KB	2,01	1,66	1,53	1,88	1,54	1,39
	256 KB	1,17	1,13	1,12	1,15	1,13	1,12

Scritto da Gabriel

- Least Recently Used (LRU), per preservare località temporale

Problema della scrittura

La scrittura dei dati determina incoerenza tra il blocco in cache e quello nei livelli inferiori.

- 'Write through'
  - Scrittura contemporanea in cache e nel livello di memoria inferiore
  - Aumento di traffico per frequenti scritture nel medesimo blocco, ma i dati sono sempre coerenti tra i livelli
  - Si ricorre a buffer di scrittura asincroni (differiti) verso la memoria
- 'Write back'
  - Scrittura in memoria inferiore differita al rimpiazzo del blocco di cache corrispondente
  - Occorre ricordare se sono avvenute operazioni di scrittura nel blocco
  - Consente ottimizzazione del traffico tra livelli
  - Causa periodi di incoerenza (problemi con moduli di I/O e multiprocessori con cache locale)

Scenario particolarmente problematico: più dispositivi (es. processori) connessi allo stesso bus con cache locale e memoria centrale condivisa

Possibili soluzioni

- Monitoraggio del bus con write through
  - Controllori cache intercettano modifiche locazioni condivise
- Trasparenza hardware
  - Hardware aggiuntivo: modifica a M ® modifica tutte cache
- Memoria noncacheable
  - Solo una porzione di M è condivisa e noncacheable (accessi a M condivisa generano miss)
- Modifica dati in una cache
  - invalida la parola corrispondente in memoria centrale
  - invalida la parola corrispondente nelle altre cache che la contengono
  - write through non risolve il problema (risolve solo l'inconsistenza della memoria centrale)

Il problema dei 'miss'

- Miss di primo accesso, inevitabile e non riducibile
- Miss per capacità insufficiente, quando la cache non può contenere tutti i blocchi necessari all'esecuzione del programma
- Miss per conflitto, quando più blocchi possono essere mappati (con associazione diretta o a gruppi) su uno stesso gruppo

Possibili soluzioni

- Maggior dimensione di blocco
  - Buona per fruire di località spaziale
  - Causa incremento di miss per conflitto (meno blocchi disponibili)
  - Maggiore associatività
- Causa incremento del tempo di localizzazione in gruppo (hit)
- Soggetta alla 'regola del 2:1'
  - Una cache ad N blocchi con associazione diretta ha una probabilità di miss pressoché uguale ad una cache di dimensione N/2 con associazione a 2 vie
- Altre tecniche:
  - Cache multilivello (cache on-chip L1 e/o L2 e/o L3)
  - Separazione tra cache dati e cache istruzioni
  - Ottimizzazione degli accessi mediante compilatori
    - Posizionamento accurato delle procedure ripetitive

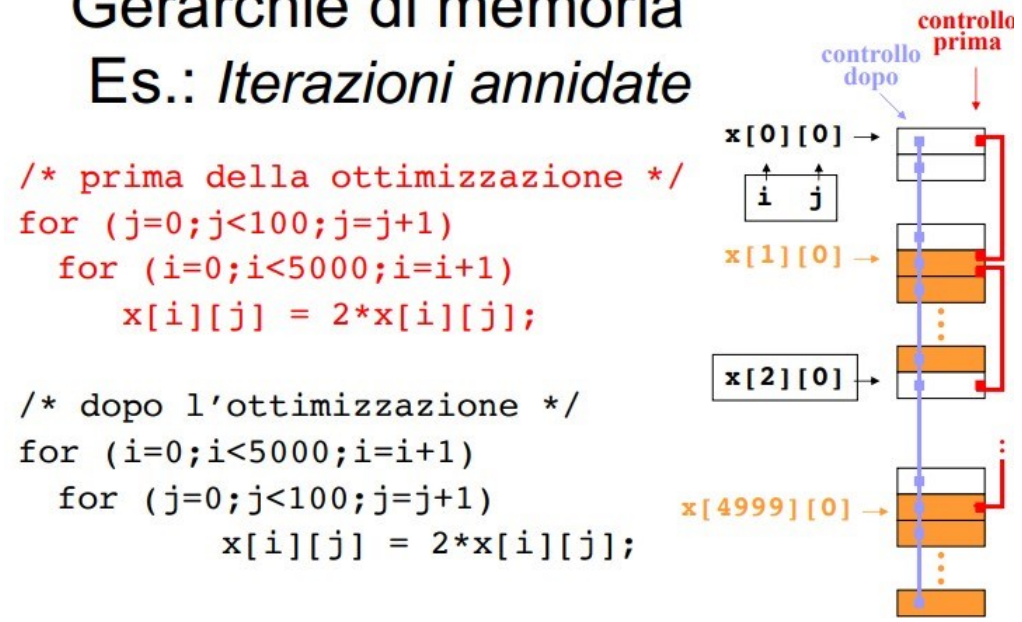
- Fusione di vettori in strutture (località spaziale)
- Trasformazioni di iterazioni annidate (località spaziale)

Gerarchie di memoria

Es.: *Fusione di vettori in strutture*

Gerarchie di memoria

Es.: *Iterazioni annidate*



Esercizi cache 1

Esercizi Cache 

organizzazione e tecniche di allocazione

Es1: Si consideri una cache di 4KB con associazione a gruppi a 8 vie (8-way set associative) in congiunzione con una memoria centrale di 1MB.

Supponendo che un blocco sia di dimensione 64B, e che la dimensione di parola sia di un singolo byte, si dica come un indirizzo di memoria è suddiviso in campi e a quanto ammonta la dimensione di ogni campo.

Soluz.:

• trattandosi di una cache con associazione a gruppi, l'indirizzo di memoria centrale deve essere suddiviso nei campi tag, set, e parola.

• la memoria centrale è di 1MB, cioè 2^{20} byte; pertanto un indirizzo di memoria centrale è espresso in 20 bit.

• la dimensione del campo parola è individuato univocamente dalla dimensione del blocco, che è di 64B, cioè 2^6 byte; pertanto il campo parola è di 6 bit.

• una cache di 4KB possiede 2^{12} byte; ogni linea deve contenere un blocco e quindi impegna 2^6 byte; quindi la cache contiene $2^{12} / 2^6 = 2^6$ linee. Poiché un insieme deve contenere 8 linee, il numero di insiemi della cache è pari a $2^6/2^3 = 2^3$. Pertanto il campo set è di 3 bit.

• la dimensione del campo tag sarà dunque: 20 -3 -6 = 11 bit

Esercizi Cache 

organizzazione e tecniche di allocazione

Es2: Si consideri una cache di 16KB con associazione a gruppi a 4 vie (4-way set associative) e dimensione di linea di 32B.

Supponendo che il campo tag sia di 12 bit, , e che la dimensione di parola sia di un singolo byte, si dica quale è la dimensione massima (in byte) di memoria principale che la cache è in grado di gestire, assumendo il singolo byte come unità di indirizzamento della memoria.

Soluz.:

• per calcolare la quantità massima di memoria principale gestibile, bisogna calcolare il numero di bit totali che esprimono una generica locazione di memoria.

• trattandosi di una cache con associazione a gruppi, l'indirizzo di memoria centrale deve essere suddiviso nei campi tag, set, e parola.

• sappiamo che il campo tag è di 12 bit; quindi occorre calcolare la dimensione dei campi set e parola.

• la dimensione del campo parola è individuato univocamente dalla dimensione del blocco, che è di 32B, cioè 2^5 byte; pertanto il campo parola è di 5 bit.

• una cache di 16KB possiede 2^{14} byte; ogni linea deve contenere un blocco e quindi impegna 2^5 byte; quindi la cache contiene $2^{14}/2^5 = 2^9$ linee. Poiché un insieme deve contenere 4 linee, il numero di insiemi della cache è pari a $2^9/2^2 = 2^7$. Pertanto il campo set è di 7 bit.

• quindi la dimensione massima di memoria gestibile è: 2^{12+7+5}, cioè 16MB

Esercizi Cache

organizzazione e tecniche di allocazione

Es3: Si consideri una cache di 4KB con associazione a gruppi a 4 vie (4-way set associative) in congiunzione con una memoria centrale di 256KB.

Supponendo che un blocco sia di dimensione 64B, e che la dimensione di parola sia di un singolo byte, si dica:

a) se le locazioni di memoria con indirizzi (in esadecimale) **30E5C** e **17A87** hanno la possibilità di essere caricate all'interno dello stesso set di linee;

b) se in cache è presente la locazione con indirizzo **05ABC**, quali altre locazioni sono sicuramente presenti nella cache.

Soluz.:

a) procedendo come visto negli esercizi precedenti, abbiamo che un indirizzo di memoria è decomponibile in un campo parola di 6 bit, un campo set di 4 bit, ed un campo tag di 8 bit.

• le due locazioni di indirizzo **30E5C** e **17A87** possono trovarsi nello stesso insieme se il loro campo set è identico. Quindi basta controllare se i bit da 10 a 7 (a partire da destra) sono identici:

30E5C = (su 18 bit) 110000111001011100

17A87 = (su 18 bit) 010111101010000111

• non essendo identici, la risposta è no.

b) le altre locazioni che necessariamente saranno presenti con la locazione di indirizzo **05ABC** sono quelle all'interno del medesimo blocco.

• poiché **05ABC** = (su 18 bit) 000101101010111100 , tutte le locazioni con indirizzo da 000101101010000000 (hex **05A80**) a 000101101010111111 (hex **05ABF**) si troveranno simultaneamente in cache.

Es4: Sia data la seguente sequenza di indirizzi in lettura (l) o scrittura (s) emessi dalla CPU:

	Indirizzo	l/s	dato scritto (in esadecimale)
1	000100000000	l	
2	000100001000	l	
3	000100001100	s	B1
4	000100001100	l	
5	000100010000	s	B4
6	000100010000	l	
7	000100010100	s	B7

Si assuma che la dimensione di parola coincida con un byte, e la presenza di una cache di ampiezza 16B, dimensione di blocco 4B, inizialmente vuota, e ad associazione a 2 vie (con politica di rimpiazzo LRU e politica di scrittura write-through). Si assuma che la memoria abbia il contenuto esadecimale mostrato di seguito:

ind	byte	ind	byte	ind	byte	ind	byte
100	0C	101	00	102	07	103	02
104	00	105	00	106	00	107	00
108	AE	109	13	10A	A1	10B	23
10C	A1	10D	42	10E	90	10F	75
110	B9	111	16	112	00	113	00
114	0A	115	07	116	03	117	71

ind = indirizzo

Si mostri come sia il contenuto della cache che il contenuto della memoria cambia.

Soluzione:

Poiché un blocco è costituito da 4B, e la cache è di 16B, si avranno in cache $2^4/2^2 = 2^2$ linee.

Essendo l'associatività a due linee (2 vie), la cache sarà costituita da due insiemi (set 0 e set 1) ognuno di 2 linee.

Quindi i 12 bit di indirizzo saranno suddivisi nel seguente modo:

- i 2 bit meno significativi individueranno il byte all'interno del blocco;
- il terzo bit da destra individuerà l'insieme (set 0 o set 1);
- i restanti bit costituiranno il campo tag.

Mostriamo di seguito l'evoluzione del contenuto della cache e della memoria.

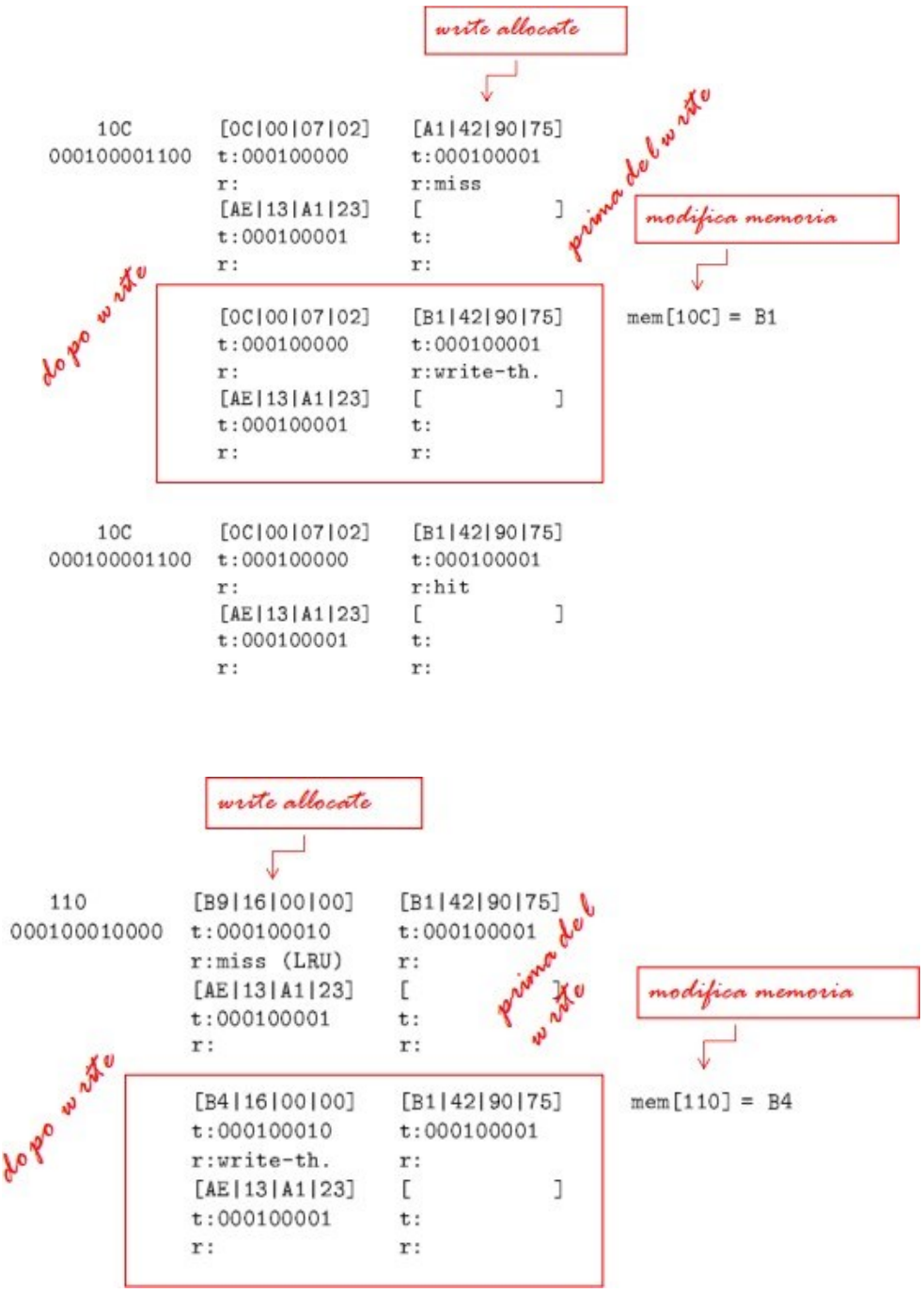
Per la cache, nel caso in cui tutte e due le linee di un insieme (set) siano libere, si sceglie la linea con indirizzo minore per la allocazione (scelta arbitraria: si poteva usare un criterio diverso).

In caso di miss per una operazione di scrittura, si assume la politica "write allocate", cioè si porta prima in cache il blocco che contiene la parola da scrivere e poi si effettua la scrittura.

Codifica della soluzione

```
ind. rif.              cache dati                  modifica memoria
memoria          set  0          set  1            mem[ind.] = cont.
   hex         [  linea  0 ]   [  linea  2 ]
 binario       t: tag          t: tag
               r: rif.         r: rif.
               [  linea  1 ]   [  linea  3 ]
               t: tag          t: tag
               r: rif.         r: rif.
```

```
    100        [0C|00|07|02]
000100000000   t:000100000
               r:miss
               [           ]
               t:
               r:

    108        [0C|00|07|02]
000100001000   t:000100000
               r:
               [AE|13|A1|23]
               t:000100001
               r:miss
```


write allocate
10C
000100001100
[0C|00|07|02]
t:000100000
r:
[AE|13|A1|23]
t:000100001
r:
[A1|42|90|75]
t:000100001
r:miss
[ ]
t:
r:
prima della write
modifica memoria
mem[10C] = B1
dopo write
[0C|00|07|02]
t:000100000
r:
[AE|13|A1|23]
t:000100001
r:
[B1|42|90|75]
t:000100001
r:write-th.
[ ]
t:
r:
10C
000100001100
[0C|00|07|02]
t:000100000
r:
[AE|13|A1|23]
t:000100001
r:
[B1|42|90|75]
t:000100001
r:hit
[ ]
t:
r:
write allocate
110
000100010000
[B9|16|00|00]
t:000100010
r:miss (LRU)
[AE|13|A1|23]
t:000100001
r:
[B1|42|90|75]
t:000100001
r:
[ ]
t:
r:
prima della write
modifica memoria
mem[110] = B4
dopo write
[B4|16|00|00]
t:000100010
r:write-th.
[AE|13|A1|23]
t:000100001
r:
[B1|42|90|75]
t:000100001
r:
[ ]
t:
r:

Memorie interne

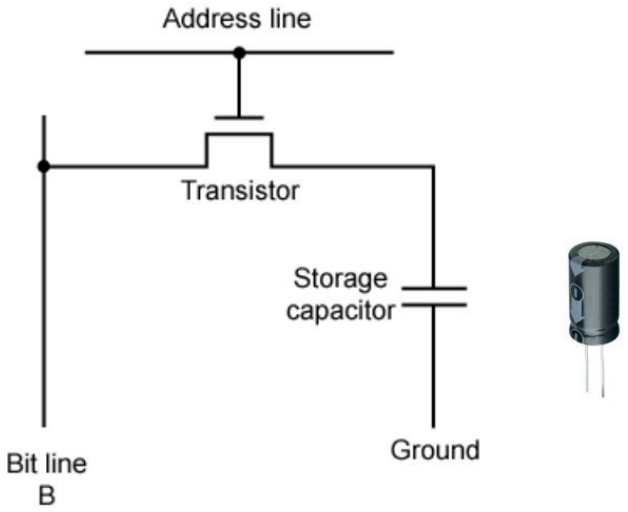
La memoria a semiconduttori è una memoria informatica in cui le informazioni e i dati sono registrati mediante le tecnologie dei semiconduttori, (transistor, circuiti integrati, chip, ecc). È una delle principali componenti della memoria centrale del computer. La memoria a semiconduttori è composta da celle elementari, ognuna delle quali può contenere un'informazione (bit) sotto forma di carica elettrica di un condensatore, che compongono una matrice simile a una bacheca/lavagna digitale. Le memorie a semiconduttori i sono prevalentemente delle memorie volatili, i dati sono mantenuti in memoria fin quando il dispositivo è alimentato dalla corrente elettrica. Al momento dello spegnimento l'informazione viene perduta. Sono utilizzate nella memoria centrale del computer.

Le memorie sono classificare in:

- Memoria RAM (Random Access Memory), ad accesso casual, memoria volatile a memorizzazione temporanea (quindi allo spegnimento del computer i dati sono automaticamente cancellati). Esiste una distinzione ulteriore, in particolare può essere:

1) RAM Dinamica (Dynamic RAM)
    - Bit memorizzati come cariche in condensatori
    - Decadimento delle cariche con il tempo
    - Necessitano di refresh delle cariche, anche durante l'alimentazione
    - Costruzione più semplice
    - Un condensatore per bit
    - Meno costose
    - Necessitano di circuiti per il refresh
    - Più lente
    - Usate per la memoria principale
    - In essenza operano in modo analogico
        - Il livello di carica determina il valore digitale

La memoria dinamica ad accesso casuale (DRAM) è un tipo di memoria a semiconduttore utilizzata in genere per i dati o il codice di programma necessari al funzionamento di un processore di computer. La DRAM è un tipo comune di memoria ad accesso casuale (RAM) utilizzata nei personal computer (PC), nelle workstation e nei server. L'accesso casuale consente al processore del PC di accedere direttamente a qualsiasi parte della memoria, anziché procedere in sequenza da un punto di partenza. La RAM si trova vicino al processore del computer e consente un accesso più rapido ai dati rispetto ai supporti di memorizzazione come i dischi rigidi e le unità a stato solido.



Come funziona la DRAM?
La memoria è costituita da bit di dati o codice di programma disposti in una griglia bidimensionale. La DRAM memorizza i bit di dati in una cosiddetta cella di memoria, composta da un condensatore e da un transistor. Le celle di memoria sono tipicamente organizzate in una configurazione rettangolare. Quando una carica viene inviata attraverso una colonna, il transistor sulla colonna viene attivato. Una cella di memoria DRAM è dinamica, il che significa che deve essere rinfrescata o ricevere una nuova carica elettronica ogni pochi millisecondi per compensare le perdite di carica dal condensatore.

Le celle di memoria funzionano con altri circuiti che possono essere utilizzati per identificare le righe e le colonne, seguire il processo di aggiornamento, indicare a una cella se accettare o meno una carica e leggere o ripristinare i dati da una cella.

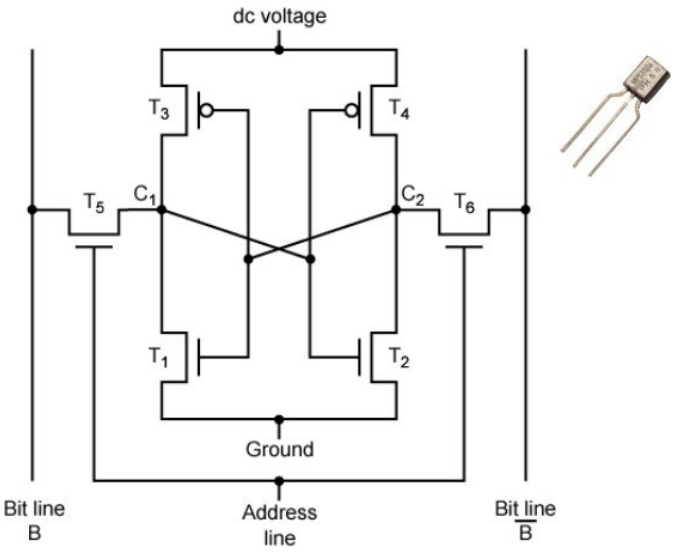
La DRAM è una delle opzioni di memoria a semiconduttore che il progettista di un sistema può utilizzare nella costruzione di un computer. Altre opzioni di memoria sono la RAM statica (SRAM), la memoria di sola lettura programmabile e cancellabile elettricamente (EEPROM), la memoria flash NOR e la memoria flash NAND. Molti sistemi utilizzano più di un tipo di memoria.

<u>Funzionamento DRAM</u>

- Linea indirizzo attivata quando si deve scrivere o leggere un bit
  - Transistor “chiuso” (la corrente fluisce)
- Write
  - Si applica tensione alla linea di bit
    - Tensione alta indica valore 1; tensione bassa indica valore 0
  - Poi si applica un segnale alla linea indirizzo
    - Trasferisce la carica al condensatore
- Read
  - Si seleziona la linea indirizzo
    - Il transistor si accende
- La carica del condensatore fluisce attraverso la linea di bit verso un amplificatore
  - Valore di carica comparato con un segnale di riferimento per stabilire se vale 0 o 1
- La carica del condensatore deve essere ristabilita (refresh)

RAM Statica

- Bit memorizzati tramite porte logiche
- Nessuna perdita di carica
- Nessuna necessità di refresh
- Costruzione più complessa
- Più elementi per bit
- Più costosa
- Non ha bisogno di circuiti di refresh
- Più veloci
- Usate per la cache
- Digitale
    - usa flip-flop



La SRAM o Static Random Access Memory (memoria statica ad accesso casuale) è una forma di memoria a semiconduttore ampiamente utilizzata nell'elettronica, nei microprocessori e nelle applicazioni informatiche in generale.
Questa forma di memoria per computer deve il suo nome al fatto che i dati sono conservati nel chip di memoria in modo statico e non devono essere aggiornati dinamicamente come nel caso della memoria DRAM.

Sebbene i dati nella memoria SRAM non debbano essere aggiornati dinamicamente, sono comunque volatili, il che significa che quando l'alimentazione viene rimossa dal dispositivo di memoria, i dati non vengono conservati e scompaiono.
La S-RAM ha il vantaggio di offrire prestazioni migliori rispetto alla DRAM, perché quest'ultima deve essere aggiornata periodicamente quando è in uso, mentre la SRAM non lo fa. Tuttavia, la SRAM è più costosa e meno densa della DRAM, per cui le dimensioni della SRAM sono di ordini di grandezza inferiori rispetto alla DRAM.

Nozioni di base sulle SRAM
Le caratteristiche principali della SRAM (Static Random Access Memory) sono due e la distinguono dagli altri tipi di memoria disponibili:

- I dati sono conservati in modo statico: Ciò significa che i dati sono conservati nella memoria a semiconduttore senza bisogno di essere aggiornati finché la memoria è alimentata.
- La memoria SRAM è una forma di memoria ad accesso casuale: Una memoria ad accesso casuale è una memoria in cui le posizioni nella memoria a semiconduttore possono essere scritte o lette in qualsiasi ordine, indipendentemente dall'ultima posizione di memoria a cui si è acceduto.

Il circuito di una singola cella di memoria SRAM comprende in genere quattro transistor configurati come due invertitori accoppiati a croce. In questo formato il circuito ha due stati stabili, che equivalgono agli stati logici "0" e "1".

I transistor sono MOSFET (metal–oxide–semiconductor field-effect transistor) perché la quantità di energia consumata da un circuito MOS è notevolmente inferiore a quella della tecnologia dei transistor bipolari, che è l'altra opzione praticabile, ma consuma molto di più.

L'uso della tecnologia bipolare limita il livello di integrazione perché la questione della rimozione del calore diventa un problema importante. Per questo motivo la tecnologia bipolare viene utilizzata raramente.

Vantaggi

I principali vantaggi della DRAM sono i seguenti:

- Il suo design è semplice e richiede un solo transistor.
- Il costo è basso rispetto a tipi di memoria alternativi come la SRAM.
- Offre livelli di densità più elevati.
- Con la DRAM si possono memorizzare più dati.
- La memoria può essere aggiornata e cancellata durante l'esecuzione di un programma.

Svantaggi

I principali svantaggi della DRAM sono i seguenti:

- La memoria è volatile.
- Il consumo di energia è elevato rispetto ad altre opzioni.
- La produzione è complessa.
- I dati nelle celle di memoria devono essere aggiornati.
- È più lenta della SRAM.

Oltre ai quattro transistor della cella di memoria di base, sono necessari altri due transistor per controllare l'accesso alla cella di memoria durante le operazioni di lettura e scrittura.
In questo modo si arriva a un totale di sei transistor, che danno vita alla cosiddetta cella di memoria a 6T, che, sebbene più complessa in termini di numero di componenti, presenta una serie di vantaggi.

Il vantaggio principale del circuito SRAM a sei transistor è la riduzione della potenza statica. Nella versione a quattro transistor, c'è un flusso di corrente costante attraverso l'uno o l'altro dei resistori di pull down e questo aumenta il consumo energetico complessivo del chip. Ciò può limitare il livello di integrazione e aumentare i problemi di progettazione dei circuiti a causa della maggiore dissipazione di potenza.

Vale la pena di ricordare che la cella di memoria SRAM a quattro transistor offre alcuni vantaggi in termini di densità, ma ciò comporta una maggiore complessità di produzione, in quanto è necessario fabbricare le resistenze e ciò richiede una lavorazione aggiuntiva. Inoltre, i resistori devono avere dimensioni ridotte e valori elevati per soddisfare i requisiti della cella.

Funzionamento RAM Statica

- La disposizione dei transistor garantisce stati stabili
- Stato 1
    - C1 alto, C2 basso
    - T1 T4 “spenti”, T2 T3 “accesi”,
- Stato 0
    - C2 alto, C1 basso
    - T2 T3 “spenti”, T1 T4 “accesi”,
- La linea indirizzo controlla i transistor T5 T6 (accesi con presenza di segnale)
- Write – si applica il valore da scrivere alla linea B ed il complemento del valore alla linea B
- Read – il valore viene letto tramite la linea B

SRAM	DRAM
La SRAM ha un tempo di accesso inferiore, più veloce rispetto alla DRAM.	La DRAM ha un tempo di accesso più elevato. È più lenta della SRAM.
La SRAM è più costosa della DRAM.	Il costo della DRAM è inferiore a quello della SRAM.
La SRAM ha bisogno di un'alimentazione costante, ma consuma meno energia.	La DRAM richiede un consumo maggiore di energia, poiché le informazioni sono memorizzate nel condensatore.
La SRAM offre una bassa densità di packaging.	La DRAM offre un'elevata densità di packaging.
Utilizza transistor e latches.	Utilizza condensatori e pochissimi transistor.
Le unità cache L2 e L3 della CPU sono alcune applicazioni generali di una SRAM.	La DRAM è la memoria principale dei computer.
La capacità di memorizzazione della SRAM va da 1MB a 16MB.	La capacità di memoria della DRAM va da 1 GB a 16 GB.
La SRAM si presenta sotto forma di memoria on-chip.	La DRAM ha le caratteristiche di una memoria off-chip.
La SRAM è ampiamente utilizzata sul processore o collocata tra la memoria principale e il processore del computer.	La DRAM è collocata sulla scheda madre.
La SRAM è di dimensioni più piccole.	La DRAM è disponibile con una capacità di memoria maggiore.
Questo tipo di RAM funziona in base al principio di cambiare la direzione della corrente attraverso degli interruttori.	Questo tipo di RAM funziona con il mantenimento delle cariche.

Read Only Memory (ROM)

- Memorizzazione permanente
    - Non volatili
- Usate per memorizzare:
    - Microprogrammi (Nella progettazione dei processori, il microcodice è una tecnica che interpone uno strato di organizzazione informatica tra l'hardware dell'unità di elaborazione centrale (CPU) e l'architettura del set di istruzioni visibile al programmatore di un computer. Il microcodice è uno strato di istruzioni a livello hardware che implementa istruzioni di codice macchina di livello superiore o sequenze interne di macchine a stati finiti in molti elementi di elaborazione digitale. Il microcodice è utilizzato nelle unità di elaborazione centrale di uso generale, anche se nelle CPU desktop attuali è solo un percorso di ripiego per i casi che l'unità di controllo cablata più veloce non è in grado di gestire. La scrittura di microcodice è spesso chiamata microprogrammazione e il microcodice di una particolare implementazione del processore è talvolta chiamato microprogramma.)

    - subroutine di libreria (Le subroutine vengono memorizzate in librerie per risparmiare spazio e rendere più efficiente il processo di collegamento dei programmi. Una libreria è un file di dati che contiene copie di un certo numero di file individuali e informazioni di controllo che consentono di accedervi singolarmente).

    - programmi di sistema (BIOS) (BIOS, in inglese Basic Input/Output System, programma per computer tipicamente memorizzato in EPROM e utilizzato dalla CPU per eseguire le procedure di avvio all'accensione del computer. Le sue due procedure principali sono la determinazione dei dispositivi periferici (tastiera, mouse, unità disco, stampanti, schede video, ecc.) disponibili e il caricamento del sistema operativo (OS) nella memoria principale)

La memoria di sola lettura (ROM) è un tipo di supporto di archiviazione che memorizza in modo permanente i dati nei personal computer (PC) e in altri dispositivi elettronici.
Contiene la programmazione necessaria per avviare un PC, essenziale per l'avvio; esegue le principali operazioni di input/output e contiene programmi o istruzioni software. Questo tipo di memoria viene spesso definito "firmware" e la sua modifica è stata una fonte di considerazione per la progettazione nel corso dell'evoluzione del computer moderno.

Tipi di ROM

- Scritte in produzione
    - Molto costoso per pochi “pezzi”
- Programmabili (una sola volta)
    - PROM
    - Necessitano di strumentazione speciale per la programmazione
- Principalmente di lettura (Read “mostly”)
    - Erasable Programmable (EPROM)
        - Si cancellano (per intero) tramite raggi ultravioletti
- Electrically Erasable (EEPROM)
    - Impiegano molto più tempo per la scrittura che per la lettura
- Memorie Flash
    - Cancellazione elettrica di blocchi di memoria

Ecco una panoramica dei diversi tipi di ROM, dai più semplici ai più versatili.

- ROM: i chip ROM classici o "programmati a maschera" contengono circuiti integrati. Un chip ROM invia una corrente attraverso uno specifico percorso di input-output determinato dalla posizione dei fusibili tra le righe e le colonne del chip. La corrente può viaggiare solo lungo un percorso abilitato dai fusibili e quindi può tornare solo attraverso l'uscita scelta dal produttore. Il ricablaggio è funzionalmente impossibile e quindi non c'è modo di modificare questi tipi di chip ROM. Mentre la produzione di un modello per un chip ROM originale è laboriosa, i chip realizzati secondo un modello esistente possono essere molto più convenienti.

- *PROM*: la ROM programmabile, o PROM, è essenzialmente una versione vuota della ROM che può essere acquistata e programmata una volta con l'aiuto di uno strumento speciale chiamato programmatore. Un chip PROM vuoto permette alla corrente di passare attraverso tutti i percorsi possibili; il programmatore sceglie un percorso per la corrente inviando un'alta tensione attraverso i fusibili indesiderati per "bruciarli". L'elettricità statica può creare accidentalmente lo stesso effetto, quindi le PROM sono più vulnerabili ai danni rispetto alle ROM convenzionali.

- *EPROM*: i chip ROM programmabili e cancellabili consentono di scrivere e riscrivere più volte. Questi chip sono dotati di una finestra di quarzo attraverso la quale un programmatore EPROM specializzato emette una specifica frequenza di luce ultravioletta. Questa luce brucia tutte le piccole cariche presenti nella EPROM per riaprirne i circuiti. Questa esposizione rende il chip nuovamente vuoto, dopodiché è possibile riprogrammarlo secondo lo stesso processo di una PROM. I chip EPROM finiscono per consumarsi, ma spesso hanno una durata di vita superiore a 1000 cancellazioni.

- *EEPROM*: per modificare un chip ROM programmabile elettricamente cancellabile, si applicano campi elettrici localizzati per cancellare e riscrivere i dati. Le EEPROM presentano diversi vantaggi rispetto ad altri tipi di ROM. A differenza delle forme precedenti, è possibile riscrivere la EEPROM senza apparecchiature dedicate, senza rimuoverla dall'hardware e in incrementi specificamente designati. Non è necessario cancellare e riscrivere tutto per effettuare una singola modifica.

- *Memorie flash*: La memoria flash è un tipo di EEPROM progettata per l'alta velocità e la densità di memoria. Pertanto, le unità flash basati su questa tecnologia possono memorizzare molti gigabyte di dati su una chiavetta USB più piccola di un pollice. Sviluppando ulteriormente questo design compatto, le schede micro SD sono grandi come un'unghia e possono comunemente memorizzare decine o addirittura centinaia di gigabyte di informazioni. Su una scala più ampia, si potrebbero stipare ancora più dati su questo tipo di memoria, forse abbastanza da fungere da metodo di memorizzazione primaria del computer. Le schede sono più o meno delle dimensioni di un'unghia e possono comunemente memorizzare decine o addirittura centinaia di gigabyte di informazioni. Su una scala più ampia, si potrebbero stipare ancora più dati su questo tipo di memoria, forse abbastanza da fungere da metodo di memorizzazione primaria del computer.

I guasti possono essere di due tipi:
- Guasti Hardware (Hard Failure)ù
    - Guasti permanenti
- Errori Software (Soft Error)
    - Random, non-distruttivi

  Danni alla memoria non permanenti
- Errori rilevati ed eventualmente corretti usando, ad esempio, codici correttori di Hamming.

Una sequenza di n bit composta da m bit di dati e di r bit di controllo, con $n = m + r$, viene chiamata parola di codice (codeword) su n bit.
Il numero di bit diversi tra due parole di codice viene detto distanza di Hamming; si può calcolare il numero di bit diversi facendo l'or esclusivo delle due stringhe e contando il numero di bit 1 del risultato; se due parole hanno distanza d significa che servono d errori per trasformare una nell'altra.

In genere non tutte le possibili stringhe di n bit (2n) sono legali (anche se quelle con m bit di dati lo sono); la distanza minima tra le parole legali del codice è la distanza di Hamming del codice.
La distanza di Hamming indica quanti errori si possono rilevare e quanti se ne possono correggere.
Per rilevare d errori serve una distanza di $d + 1$; per correggere d errori serve una distanza di $2d + 1$ (la parola originale è la più vicina valida).

Il codice di Hamming è un codice che permette di aggiungere un certo numero di bit ai bit di dati in modo da comporre parole con distanza 3, in grado di rilevare e correggere errori su un singolo bit. Il numero di bit da aggiungere aumenta all'aumentare del numero dei bit di dati.
I bit aggiunti sono bit di parità calcolati su sottoinsiemi di bit della parola di codice; numerando a partire da 1 a sinistra i bit che compongono la parola di codice, i bit di parità vengono inseriti nelle posizioni che sono potenze di 2 (1, 2, 4, 8, 16 …); gli altri bit sono i bit di dati.

Ogni bit di parità viene calcolato su un sottoinsieme di bit; ogni bit di dati può essere incluso in diversi sottoinsiemi e influire su diversi bit di parità. Per sapere su quali bit di parità influisce il bit di dati k, basta riscrivere k come somma di potenze di 2 (per esempio 11 = 1 + 2 + 8); ogni bit di dati è controllato da tutti e soli i bit di parità che appartengono alla sua espansione (il bit 11 è controllato dai bit 1, 2 e 8).
Al momento del controllo, per ogni parola di codice vengono ricalcolati i bit di parità per ogni posizione k (con k = 1, 2, 4, 8, 16…). Se la parità non è corretta viene aggiunto k a un contatore inizializzato a 0; al termine il valore del contatore indica la posizione del bit errato (se non sono corretti i bit di parità 1, 2 e 8 il bit errato è quello in posizione 11).

Partendo dalla sequenza:

$$0\ 1\ 1\ 0$$
$$(1\ 2\ 3\ 4)$$

si calcolano e inseriscono i bit di parità nelle posizioni 1, 2 e 4:

_ _ 0 _ 1 1 0
(1 2 3 4 5 6 7)

quindi i bit di dati occupano le posizioni 3, 5, 6 e 7.

Il bit 3 influenza i bit di parità 1 e 2 (3=1+2).

Il bit 5 influenza i bit di parità 1 e 4 (5=1+4).

Il bit 6 influenza i bit di parità 2 e 4 (6=2+4).

Il bit 7 influenza i bit di parità 1, 2 e 4 (7=1+2+4).

Il bit di parità 1 è calcolato sui bit di dati 3, 5 e 7 (0 1 0) e quindi vale 1.

Il bit di parità 2 è calcolato sui bit di dati 3, 6 e 7 (0 1 0) e quindi vale 1.

Il bit di parità 4 è calcolato sui bit di dati 5, 6 e 7 (1 1 0) e quindi vale 0.

Perciò si ottiene la sequenza:

1 1 0 0 1 1 0

Se un errore modifica la sequenza in:

1 1 0 0 1 0 0
(1 2 3 4 5 6 7)

ricalcolando i bit di parità si ottiene:

bit di parità 1 calcolato sui bit di dati 3, 5 e 7 (0 1 0): 1

bit di parità 2 calcolato sui bit di dati 3, 6 e 7 (0 0 0): 0

bit di parità 4 calcolato sui bit di dati 5, 6 e 7 (1 0 0): 1

Confrontando i bit della sequenza originale con quelli calcolati viene incrementato il contatore k:

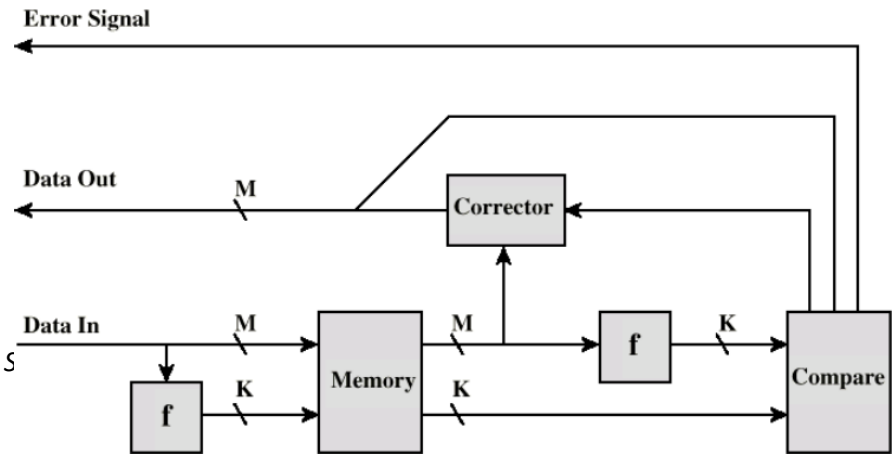
il bit di parità 1 è uguale: k=0;

il bit di parità 2 è diverso: k=2;

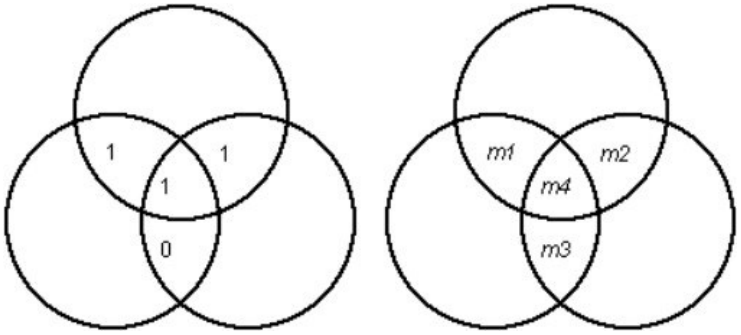
il bit di parità 4 è diverso: k=6.

Pertanto, si può stabilire che c'è un bit errato nella posizione 6 e lo si può correggere riottenendo la sequenza 1 1 0 0 1 1 0.

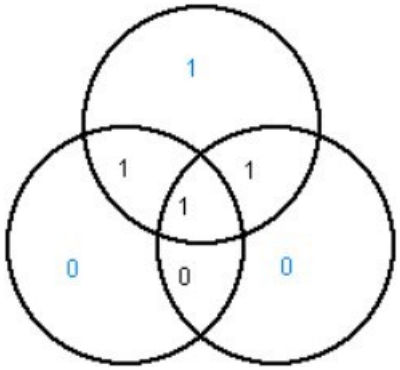
Lo schema del funzionamento del codice a correzione di errore:



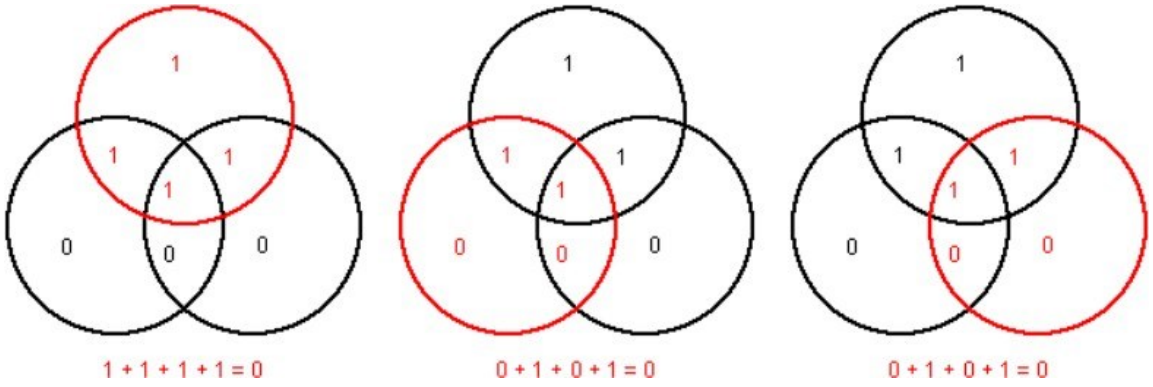
Possiamo anche visualizzare il funzionamento con i *diagrammi di Venn*.
A titolo di esempio, si supponga di voler inviare il messaggio 1101. Associamo ciascuno dei quattro bit del messaggio a una specifica regione di intersezione di tre cerchi sovrapposti, come illustrato di seguito:



Il codice di Hamming aggiunge tre bit di parità in modo che ogni cerchio abbia una parità pari.

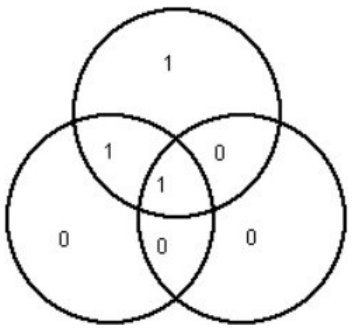


Cioè, la somma dei quattro bit di ciascun cerchio è ora pari:

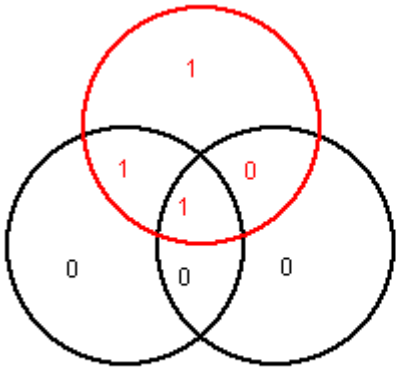


In questo caso si invia 1101100 poiché i tre bit di parità sono 1 (in alto), 0 (a sinistra) e 0 (a destra).

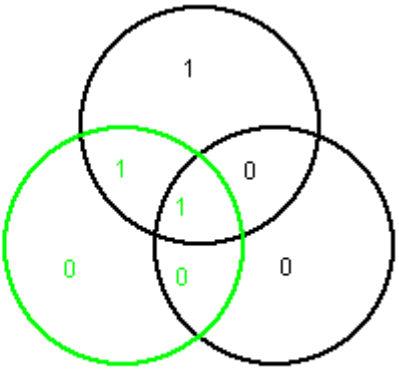
Immaginiamo ora che questa immagine venga trasmessa via modem su un canale di comunicazione rumoroso e che un bit venga corrotto in modo tale che alla stazione ricevente arrivi la seguente immagine (corrispondente a 1001100):



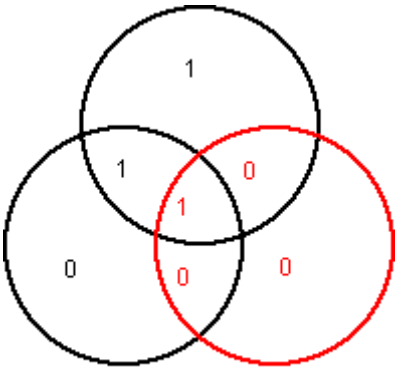
Il ricevitore scopre che si è verificato un errore controllando la parità dei tre cerchi. Inoltre, il ricevitore può persino determinare dove si è verificato l'errore (il secondo bit) e recuperare i quattro bit del messaggio originale!



1 + 0 + 1 + 1 = 1 NOT OK!



0 + 1 + 0 + 1 = 0 OK!



0 + 1 + 0 + 0 = 1 NOT OK!

Poiché il controllo di parità per il cerchio superiore e per quello destro non è andato a buon fine, mentre per quello sinistro era tutto a posto, c'è solo un bit che potrebbe essere responsabile, ovvero m2. Se il bit centrale, m4, è danneggiato, tutti e tre i controlli di parità falliranno. Se uno stesso bit di parità è danneggiato, solo un controllo di parità fallirà. Se il collegamento dati è così rumoroso che due o più bit sono corrotti contemporaneamente, il nostro schema non funziona. Riuscite a capire perché? Codici di correzione degli errori più sofisticati sono in grado di gestire situazioni di questo tipo.

Correzione degli errori: disposizione bit

Quanti bit di controllo servono ? $2^K-1 \geq M + K$

Bit di dati	Bit di controllo	% incremento
8	4	50
16	5	31,25
32	6	18,75
64	7	10,94

Bit Position	12	11	10	9	8	7	6	5	4	3	2	1
Position Number	1100	1011	1010	1001	1000	0111	0110	0101	0100	0011	0010	0001
Data Bit	D8	D7	D6	D5		D4	D3	D2		D1		
Check Bit					C8				C4		C2	C1

Esempio generazione bit di controllo

12 1100	11 1011	10 1010	9 1001	8 1000	7 0111	6 0110	5 0101	4 0100	3 0011	2 0010	1 0001
D8	D7	D6	D5		D4	D3	D2		D1		
				C8				C4		C2	C1
1	1	0	0		0	0	1		0		

$$C1 = D1 \oplus D2 \oplus D4 \oplus D5 \oplus D7$$
$$C2 = D1 \oplus D3 \oplus D4 \oplus D6 \oplus D7$$
$$C4 = D2 \oplus D3 \oplus D4 \oplus D8$$
$$C8 = D5 \oplus D6 \oplus D7 \oplus D8$$

si ha

$$C1 = 0 \oplus 1 \oplus 0 \oplus 0 \oplus 1 = 0$$
$$C2 = 0 \oplus 0 \oplus 0 \oplus 0 \oplus 1 = 1$$
$$C4 = 1 \oplus 0 \oplus 0 \oplus 1 = 0$$
$$C8 = 0 \oplus 0 \oplus 1 \oplus 1 = 0$$

Correzione degli errori: disposizione bit

Bit position	12	11	10	9	8	7	6	5	4	3	2	1
Position number	1100	1011	1010	1001	1000	0111	0110	0101	0100	0011	0010	0001
Data bit	D8	D7	D6	D5		D4	D3	D2		D1		
Check bit					C8				C4		C2	C1
Word stored as	0	0	1	1	0	1	0	0	1	1	1	1
Word fetched as	0	0	1	1	0	1	1	0	1	1	1	1
Position Number	1100	1011	1010	1001	1000	0111	0110	0101	0100	0011	0010	0001
Check Bit					0				0		0	1

XOR XOR XOR XOR

risultato XOR 0 1 1 0

es8

Sia data la seguente sequenza di indirizzi in lettura (l) o scrittura (s) emessi dalla CPU e che la memoria abbia il contenuto esadecimale mostrato di seguito:

#	indirizzo (binario)	l/s	byte scritto (HEX)
1	000100001000	s	43
2	000100001100	s	3F
3	000100001111	l	
4	000100001101	l	
5	000100010100	l	
6	000100011111	s	AE
7	000100000111	s	CD
8	000100100110	l	

ind	byte	ind	byte	ind	byte	ind	byte
100	08	101	D0	102	07	103	02
104	00	105	00	106	00	107	00
108	AE	109	13	10A	A1	10B	23
10C	A1	10D	42	10E	90	10F	75
110	BB	111	16	112	00	113	00
114	0A	115	87	116	03	117	71
118	3E	119	13	11A	A1	11B	23
11C	A1	11D	82	11E	90	11F	15
120	F9	121	86	122	A0	123	00
124	E9	125	16	126	05	127	00

Si assuma che la dimensione di parola coincida con un byte, e la presenza di una cache di ampiezza 16B, dimensione di blocco 2B, inizialmente vuota, e ad associazione a 2 vie (politica di rimpiazzo LRU, politica di scrittura write-back e gestione dei miss in scrittura con la politica write allocate).

Si mostri come sia il contenuto della cache che il contenuto della memoria cambia.

Soluzione (da compilare)

- Indicare di seguito in quali campi (e la loro dimensione) gli indirizzi emessi dalla CPU sono suddivisi:
  tag (o etichetta) da 9 bit, set (o insieme) da 2 bit, word (o parola) da 1 bit
- Indicare di seguito in quante linee/set la cache è suddivisa:
  La cache è costituita da 4 set, ognuno di 2 linee da 2B

Indicare l'evoluzione della cache e della modifica della memoria nello schema sottostante:

Indirizzo	hit/ miss	Cache *(per ogni linea di cache indicare il contenuto del campo tag)*				Modifica memoria *M[ind.] = contenuto*
		set 00	set 01	set 10	set 11	
108_{HEX} 000100001000	miss	*linea* 0 [$AE13$] *write allocate* ⇓ *linea* 0 [4313]* *tag*:000100001				
$10C_{HEX}$ 000100001100	miss	*linea* 0 [4313]* *tag*:000100001		*linea* 0 [$A142$] *write allocate* ⇓ *linea* 0 [$3F42$]* *tag*:000100001		

continuare nella pagina seguente

Indirizzo	hit/ miss	Cache *(per ogni linea di cache indicare il contenuto del campo tag)*				Modifica memoria *M[ind.] = contenuto*
		set 00	set 01	set 10	set 11	
$10F_{HEX}$ 000100001111	miss	*linea* 0 [4313]* *tag*:000100001		*linea* 0 [$3F42$]* *tag*:000100001	*linea* 0 [9075] *tag*:000100001	
$10D_{HEX}$ 000100001101	hit	*linea* 0 [4313]* *tag*:000100001		*linea* 0 [$3F42$]* *tag*:000100001	*linea* 0 [9075] *tag*:000100001	
114_{HEX} 000100010100	miss	*linea* 0 [4313]* *tag*:000100001		*linea* 0 [$3F42$]* *tag*:000100001 *linea* 1 [$0A87$] *tag*:000100010	*linea* 0 [9075] *tag*:000100001	
$11F_{HEX}$ 000100011111	miss	*linea* 0 [4313]* *tag*:000100001		*linea* 0 [$3F42$]* *tag*:000100001 *linea* 1 [$0A87$] *tag*:000100010	*linea* 0 [9075] *tag*:000100001 *linea* 1 [9015] *write allocate* ⇓ *linea* 1 [$90AE$]* *tag*:000100011	

107_{HEX} 000100000111	miss	*linea 0* [4313]* *tag*:000100001	*linea 0* [3F42]* *tag*:000100001	*linea 0 [LRU]* [0000] *write allocate*	
			linea 1 [0A87] *tag*:000100010	*linea 1* [90AE]* *tag*:000100011	
				⇓	
		linea 0 [4313]* *tag*:000100001	*linea 0* [3F42]* *tag*:000100001	*linea 0 [LRU]* [00CD]* *tag*:000100000	
			linea 1 [0A87] *tag*:000100010	*linea 1* [90AE]* *tag*:000100011	
126_{HEX} 000100100110	miss	*linea 0* [4313]* *tag*:000100001	*linea 0* [3F42]* *tag*:000100001	*linea 0* [00CD]* *tag*:000100000	M[11E] = 90 M[11F] = AE
			linea 1 [0A87] *tag*:000100010	*linea 1 [LRU]* [0500] *tag*:000100100	

* indica linea sporca a causa della politica write-back

Esercizi codice correzione Hamming

Es1: Si supponga che una parola di dati da 8 bit memorizzata sia

11001010

Adottando l'algoritmo di Hamming, determinare quali bit di controllo verrebbero immagazzinati in memoria insieme alla parola di dati.

Es2: Per la parola

00111001

i bit di controllo memorizzati sono 0111. Si supponga che, quando la parola viene letta dalla memoria, i bit di controllo siano calcolati per essere 1101.
Quale parola di dati è letta dalla memoria ?

Es3: Quanti bit di controllo sono necessari se il codice a correzione di errore di Hamming viene usato per rilevare errori di bit singoli in una parola di dati a 1024 bit ?

Es4: Sviluppare un codice SEC per una parola di dati a 16 bit. Generate il codice per la parola dati

0101000000111001

Esercizi codice correzione Hamming

Soluz. es2: I dati e bit di controllo scritti in memoria sono:

12	11	10	9	8	7	6	5	4	3	2	1
D8	D7	D6	D5		D4	D3	D2		D1		
				C8				C4		C2	C1
0	0	1	1	0	1	0	0	1	1	1	1

I bit di controllo calcolati dai dati letti da memoria sono **1101**, pertanto la ***parola sindrome*** è:

$$(0\ 1\ 1\ 1) \oplus (1\ 1\ 0\ 1) = 1\ 0\ 1\ 0$$

quindi il bit errato è quello in posizione 10 (bit dati D6). Ne consegue che la parola letta dalla memoria è

00011001

Esercizi codice correzione Hamming

Soluz. es3: Bisogna trovare il valore minimo di K tale che:

$$2^K\text{-}1 \geq M + K$$

con $M = 1024 = 2^{10}$

$$2^K\text{-}1 \geq 2^{10} + K$$

Pertanto sicuramente K deve essere maggiore di 10. Verifichiamo se K=11 soddisfa la disuguaglianza di sopra:

$$2047 = 2^{11}\text{-}1 \geq 2^{10} + 11 = 1035$$ **VERO!**

Soluz. es4: Poiché $M = 16 = 2^4$, K deve essere >4, K=5 va bene:

$$31 = 2^5\text{-}1 \geq 2^4 + 5 = 21$$

Pertanto dobbiamo avere M + K = 16 + 5 = 21 bit disposti secondo lo schema che assegna ai bit di controllo le posizioni che sono potenze di 2 (C1, C2, C4, C8, C16) e alle altre posizioni i bit dati (da D1 a D16):

21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1
D16	D15	D14	D13	D12		D11	D10	D9	D8	D7	D6	D5		D4	D3	D2		D1		
					C16								C8				C4		C2	C1
0	1	0	1	0		0	0	0	0	0	1	1		1	0	0		1		

Esercizi codice correzione Hamming

Per capire quali bit dati contribuiscono a formare il valore per i vari bit di controllo bisogna rappresentare tutte le posizioni tramite i 5 bit di controllo e selezionare per ogni bit di controllo i bit dati che corrispondono ad un valore di 1 per quel bit:

C16	C8	C4	C2	C1	posizione	bit dati
0	0	0	0	1	1	
0	0	0	1	0	2	
0	0	0	1	1	3	D1
0	0	1	0	0	4	
0	0	1	0	1	5	D2
0	0	1	1	0	6	D3
0	0	1	1	1	7	D4
0	1	0	0	0	8	
0	1	0	0	1	9	D5
0	1	0	1	0	10	D6
0	1	0	1	1	11	D7
0	1	1	0	0	12	D8
0	1	1	0	1	13	D9
0	1	1	1	0	14	D10
0	1	1	1	1	15	D11
1	0	0	0	0	16	
1	0	0	0	1	17	D12
Continuare in modo analogo...						

Risultato finale:

$$C1 = D1 \oplus D2 \oplus D4 \oplus D5 \oplus D7 \oplus D9 \oplus D11 \oplus D12 \oplus D14 \oplus D16$$

$$C2 = D1 \oplus D3 \oplus D4 \oplus D6 \oplus D7 \oplus D10 \oplus D11 \oplus D13 \oplus D14$$

$$C4 = D2 \oplus D3 \oplus D4 \oplus D8 \oplus D9 \oplus D10 \oplus D11 \oplus D15 \oplus D16$$

$$C8 = D5 \oplus D6 \oplus D7 \oplus D8 \oplus D9 \oplus D10 \oplus D11$$

$$C16 = D12 \oplus D13 \oplus D14 \oplus D15 \oplus D16$$

Esercizi codice correzione Hamming

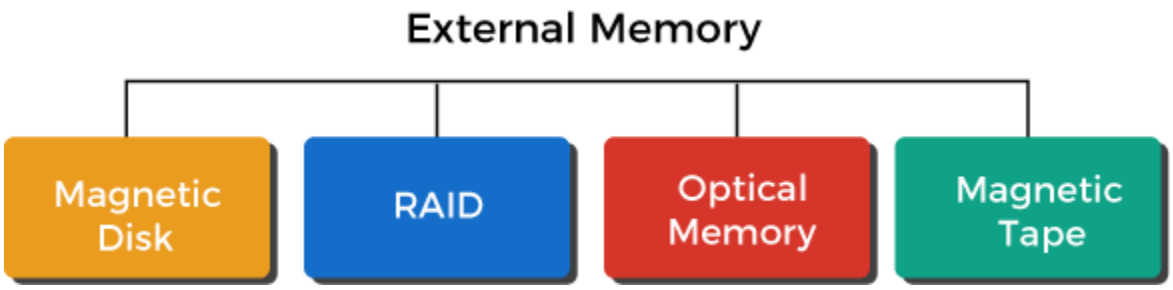
Quindi in memoria si deve immagazzinare la seguente configurazione binaria

21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1
D16	D15	D14	D13	D12		D11	D10	D9	D8	D7	D6	D5		D4	D3	D2		D1		
					C16								C8				C4		C2	C1
0	1	0	1	0	0	0	0	0	0	0	1	1	0	1	0	0	0	1	0	1

Memorie esterne

La memoria esterna è nota anche come memoria secondaria. Viene utilizzata per memorizzare un'enorme quantità di dati perché ha un'enorme capacità. Attualmente può misurare i dati in centinaia di megabyte o addirittura in gigabyte. L'importante proprietà della memoria esterna è che, in caso di spegnimento del computer, le informazioni memorizzate non andranno perse. La memoria esterna può essere suddivisa in quattro parti:

1. Disco magnetico
2. Raid
3. Memoria ottica
4. Nastro magnetico



Scritto da Gabriel

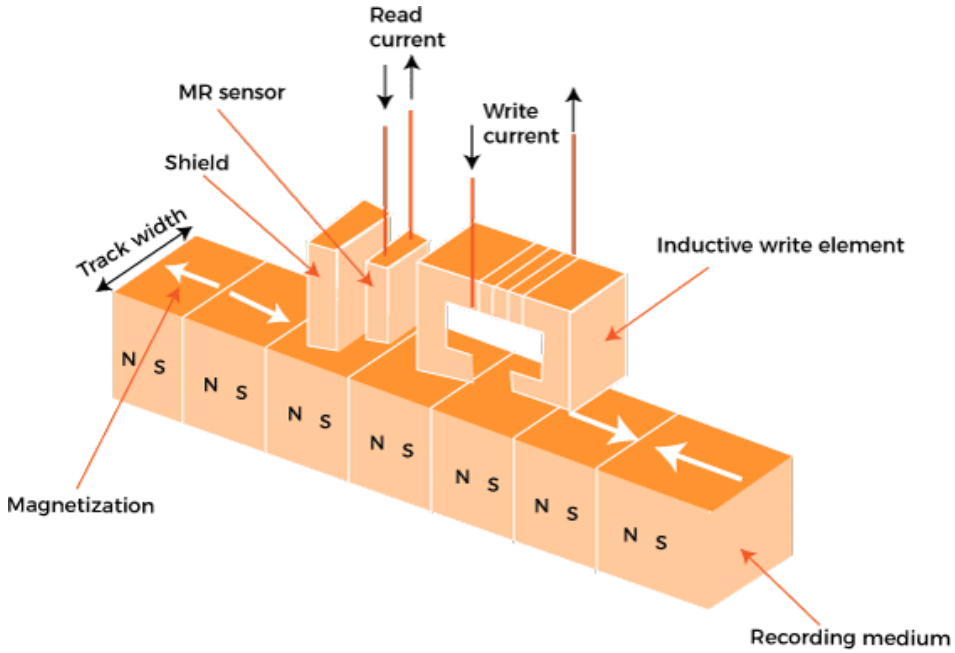
Dischi magnetici

Un disco è un tipo di piatto circolare costruito da un materiale non magnetico, noto come substrato. È ricoperto da un rivestimento magnetico utilizzato per contenere le informazioni. Il substrato è tradizionalmente costituito da alluminio o da una lega di alluminio. Recentemente, però, è stato introdotto un altro materiale, noto come substrato di vetro. I substrati di vetro offrono diversi vantaggi, descritti di seguito:

o Può aumentare l'affidabilità del disco migliorando l'uniformità della superficie del film magnetico.
o Viene utilizzato per ridurre gli errori di lettura-scrittura grazie a una significativa riduzione dei difetti superficiali complessivi.
o Ha una migliore rigidità, che contribuisce a ridurre la dinamica del disco. Ha la grande capacità di resistere agli urti e ai danni.

Memoria magnetica di lettura e scrittura

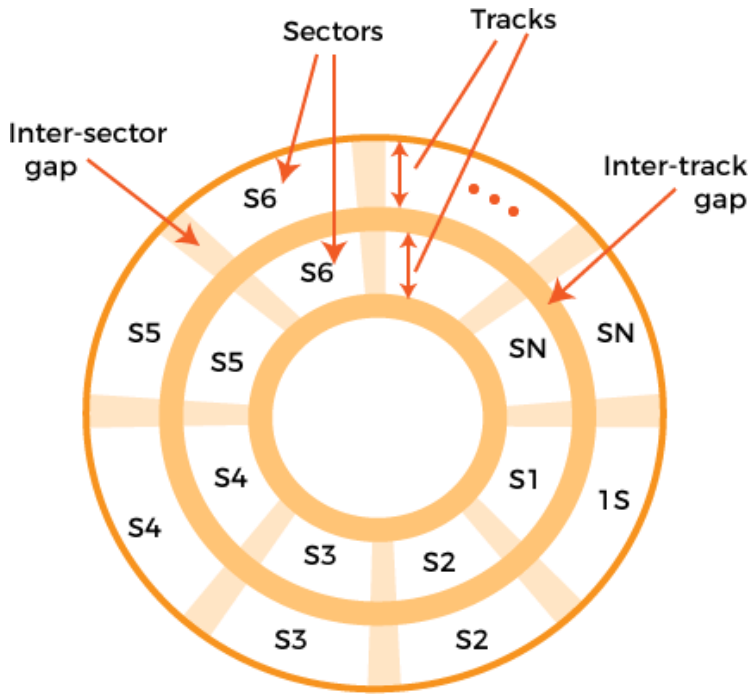
Il componente più importante della memoria esterna sono ancora i dischi magnetici. Molti sistemi, come supercomputer, personal computer e mainframe, contengono dischi rigidi sia rimovibili che fissi. È possibile condurre una bobina, denominata testina, in modo da poter recuperare i dati su e in un secondo momento e quindi recuperarli dal disco. Molti sistemi contengono due testine: una di lettura e una di scrittura. Durante le operazioni di lettura e scrittura, il piatto ruota mentre la testina è ferma.



Se l'elettricità scorre attraverso la bobina, il meccanismo di scrittura sfrutta il fatto che la bobina genera un campo magnetico. La testina di scrittura riceverà gli impulsi elettrici e la superficie sottostante registrerà il modello magnetico risultante. La registrazione avverrà secondo schemi diversi per correnti negative e positive. Se l'elettricità scorre attraverso la bobina, il meccanismo di lettura sfrutterà il fatto che genererà una corrente elettrica nella bobina. Quando la superficie del disco passa sotto la testina, produrrà una corrente con la stessa polarità di quella già registrata.
In questo caso, la struttura della testina è la stessa per la lettura e la scrittura. Pertanto, è possibile utilizzare la stessa testina per entrambe. Questi tipi di testine singole possono essere utilizzate nei vecchi sistemi di dischi rigidi e nei sistemi di dischi floppy. Nella testina di lettura è presente un tipo di sensore magneto-resistivo (MR) parzialmente schermato. La resistenza elettrica è contenuta nel materiale MR, che dipende dalla direzione di magnetizzazione del mezzo che si muove sotto di esso.

Organizzazione e formattazione dei dati

La testina è un piccolo dispositivo in grado di leggere o scrivere sulla porzione di piatto che ruota sotto di essa. La larghezza di ogni traccia è uguale a quella della testina. Ci sono migliaia di tracce per ogni superficie. Gli spazi vuoti vengono utilizzati per indicare la separazione delle tracce adiacenti. In questo modo è possibile prevenire o ridurre al minimo l'errore generato dall'interferenza dei campi magnetici o dal disallineamento della testina. I settori sono utilizzati per trasferire i dati da e verso i dischi.



I settori a lunghezza fissa saranno utilizzati nei sistemi più moderni con 512 byte, una dimensione quasi universale. Gli spazi intersettoriali separano i settori adiacenti in modo da evitare di imporre ai sistemi requisiti di precisione irragionevoli. Allo stesso tempo, possiamo scansionare le informazioni con l'aiuto della rotazione del disco a una velocità fissa, chiamata velocità angolare costante (CAV).
I dischi possono essere suddivisi in vari modi. Possono essere suddivisi in una serie di tracce concentriche e in diversi settori a forma di torta. La CAV ha il vantaggio che le tracce e i settori possono indirizzare direttamente i dati con l'aiuto della CAV. Il CAV ha anche uno svantaggio: la quantità di dati memorizzati sulle tracce interne corte e sulle tracce esterne lunghe è la stessa.

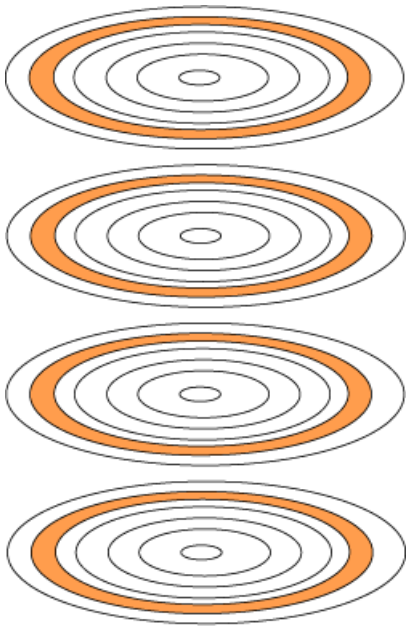
Constant angular velocity

I moderni dischi rigidi introducono una tecnica per aumentare la densità, chiamata registrazione a zone multiple. Con questa tecnica, la superficie può essere suddivisa in diverse zone concentriche, che in genere sono pari a 16, il che significa 16 zone. Il numero di bit per traccia è costante all'interno di una zona. Le zone più vicine al centro hanno una quantità inferiore di bit o settori rispetto alle zone più lontane dal centro.

Multiple zoned recording

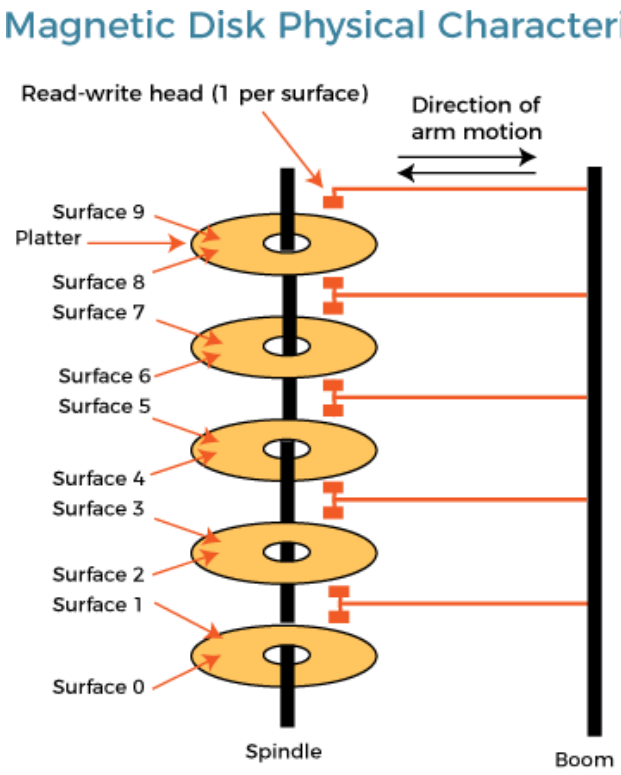
Caratteristiche fisiche

Se si tratta di un disco a testine fisse, allora conterrà una testina di lettura-scrittura per traccia. Tutte queste testine sono montate su un braccio rigido, che può estendersi su tutte le tracce. Se il disco è a testine mobili, conterrà una sola testina di lettura-scrittura. Anche in questo caso la testina è montata sul braccio. La testina può posizionarsi sopra qualsiasi traccia. A questo scopo, il braccio può essere retratto o esteso.
L'unità disco contiene sempre o permanentemente un disco non rimovibile. Ad esempio, nei personal computer, il disco rigido non può mai essere rimosso, o possiamo dire che è un disco non rimovibile. Il disco rimovibile è un tipo di disco che può essere rimosso e sostituito con altri dischi. Entrambi i lati del piatto contengono il rivestimento magnetizzabile per la maggior parte dei dischi, che sarà anche indicato come doppio lato. I dischi a lato singolo sono utilizzati in alcuni sistemi di dischi meno costosi.



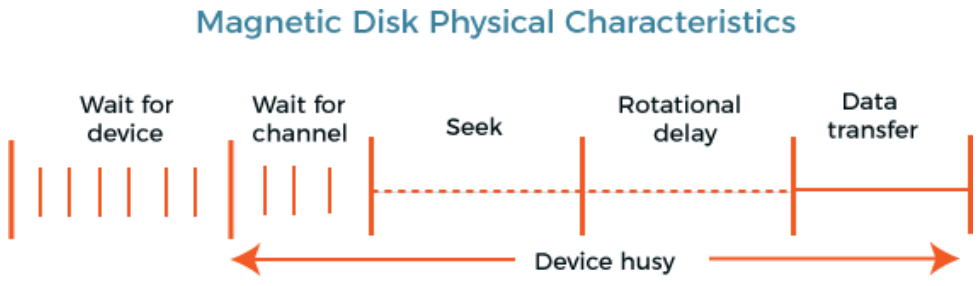
Tracks and Cylinders

I dischi a piatti multipli impiegano una testina mobile con una testina di lettura-scrittura per ogni superficie del piatto. Al centro del disco, tutte le testine hanno la stessa distanza e si muovono insieme perché tutte le testine sono fissate meccanicamente. Nel piatto, un insieme di tutte le tracce nella stessa posizione relativa è noto come cilindro.



Components of a Disk Drive

Questo tipo di meccanismo è utilizzato soprattutto nei floppy disk. Questo tipo di disco è il meno costoso, è piccolo e contiene anche un piatto flessibile. I gruppi di unità sigillati sono quasi privi di contaminanti e contengono le testine Winchester. IBM utilizza il termine Winchester come nome in codice, ed è stato utilizzato per il modello di disco 3340 prima del suo annuncio in IBM. Le workstation e i personal computer contengono comunemente un disco incorporato, noto come disco Winchester. Questo disco viene anche chiamato disco rigido.



Timing of a Disk I/O Transfer

In un sistema mobile, ci sarà un tempo di ricerca che può essere definito come il tempo necessario per posizionare la testina sulla traccia. Ci sarà anche una latenza di rotazione o ritardo di rotazione, che può essere definita come il tempo impiegato dall'inizio del settore per raggiungere la testina. Il tempo necessario per raggiungere la posizione di scrittura o lettura è noto come tempo di accesso, pari alla somma del ritardo di rotazione e dell'eventuale tempo di ricerca.
Una volta che la testina ha raggiunto la sua posizione, possiamo eseguire l'operazione di lettura o scrittura mentre il settore si muove sotto la testina. Questo processo può essere chiamato la parte di trasferimento dei dati dell'operazione e il tempo impiegato per trasferire i dati è noto come tempo di trasferimento.

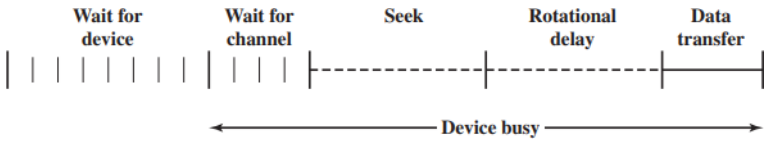
Scritto da Gabriel

Prestazionalmente parlando, quindi con relativa temporizzazione:

- Tempo di posizionamento (seek time)
  - spostamento della testina sulla giusta traccia
    5-20 ms, difficilmente riducibile
- Latenza [rotazionale] (latency)
  - attesa che il settore di interesse cada sotto la testina
  - dipende dalla velocità di rotazione
    Esempio
    RPM=3600 ⇒ RPS=60 ⇒ 1 rotazione ≈16.7ms ⇒ T_L=8.35ms
- Tempo di accesso = (seek + latency)
- Tempo di trasferimento:

$$T = \frac{b}{rN}$$

b	#byte da trasferire
N	#byte per traccia
r	velocità rotazione (in rotazioni per sec.)

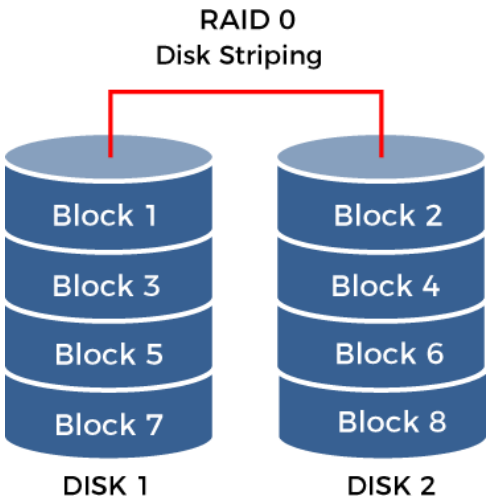


Dischi RAID

RAID è una tecnologia utilizzata per aumentare le prestazioni e/o l'affidabilità dell'archiviazione dei dati. L'abbreviazione sta per Redundant Array of Independent Drives o Redundant Array of Inexpensive Disks, più vecchio e meno utilizzato. Un sistema RAID è costituito da due o più unità che lavorano in parallelo. Possono essere dischi rigidi, ma si sta diffondendo la tendenza a utilizzare questa tecnologia anche per le unità SSD (Solid State Drives). Esistono diversi livelli RAID, ciascuno ottimizzato per una situazione specifica. Questi non sono standardizzati da un gruppo industriale o da un comitato di standardizzazione. Questo spiega perché a volte le aziende propongono numeri e implementazioni uniche.

- RAID 0

Il RAID 0 può essere chiamato anche striping del disco. Nella tecnica RAID 0, i dati vengono suddivisi in modo uniforme su due o più dispositivi di archiviazione come HDD o SDD. In questa tecnica, i dati vengono organizzati in modo tale che gli utenti possano leggere o scrivere i file più velocemente. Grazie a questo processo, le prestazioni si velocizzano. Se si dispone di un gran numero di applicazioni e di dati enormi, la soluzione migliore è lo stripping del disco.



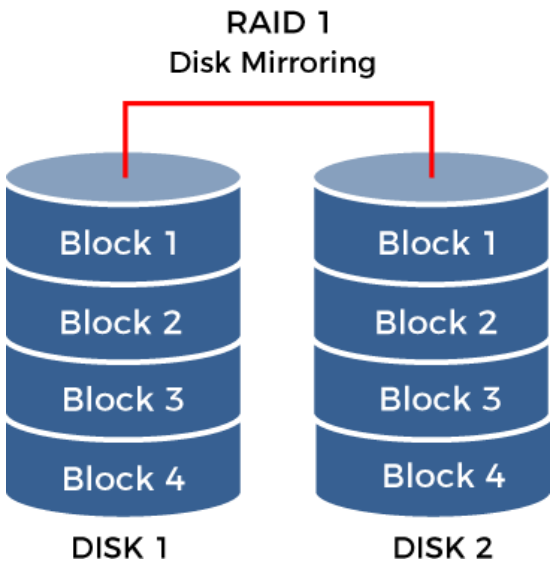
La configurazione di RAID 0 è molto semplice. Può anche essere definito il tipo più conveniente di organizzazione ridondante dei dischi. Tuttavia, questo tipo di organizzazione non è in gradodi gestire i guasti o gli errori e non può essere utilizzato per gestire i dati critici. Questo perché scrive il primo blocco nel primo disco, il secondo nel disco successivo e così via. Questo processo viene ripetuto fino a raggiungere tutti i dischi. Infine, torna al

primo disco. Ciò significa che tutti i dischi lavorano in parallelo e che siamo in grado di vedere tutte le prestazioni dei nostri dischi.
Il rovescio della medaglia è che non c'è ridondanza e quindi se un disco si rompe, perdiamo tutti i dati su tutti i dischi. Quindi RAID 0 offre prestazioni elevate ed espansione dello storage, ma in realtà è meno affidabile rispetto a un singolo disco.

Vantaggi del RAID 0
o Nelle operazioni di lettura e scrittura, offre grandi prestazioni.
o Non ci sarà alcun overhead perché RAID 0 utilizza tutta la capacità di archiviazione.
o Con RAID 0 è possibile implementare facilmente la tecnologia.
Svantaggi del RAID 0
o Il RAID 0 non può essere utilizzato nei sistemi critici perché non è in grado di tollerare i guasti.
o Se un disco si guasta in RAID 0, anche tutti i dati degli altri dischi vanno persi.

- RAID 1

RAID 1 può essere chiamato anche Mirroring. Prende tutti i dati da un disco e li scrive su un secondo disco, che è parallelo al primo. In RAID 1, la ridondanza è molto elevata perché ogni disco contiene la copia esatta dei dati presenti su un altro disco. Per funzionare ha bisogno di almeno due dischi. La configurazione di RAID 1 fornisce una protezione contro la perdita di dati, o possiamo dire che ha una capacità di tolleranza ai guasti. Se un disco si guasta, la copia di quel disco fornisce i dati necessari.
In questo caso, i sistemi possono leggere i dati da entrambi i dischi contemporaneamente. Grazie a questa caratteristica, il sistema è in grado di accelerare le prestazioni e la disponibilità. Tuttavia, le prestazioni dell'operazione di scrittura sono inalterate.



Richiede più tempo rispetto all'operazione di lettura perché il RAID 1 contiene due dischi che scrivono in parallelo e l'operazione di scrittura utilizza la capacità di un disco e deve scrivere gli stessi dati due volte. In RAID 1, lo svantaggio dei dischi è rappresentato dai costi elevati, perché un disco deve costruire il doppio della capacità effettivamente necessaria a questo livello.

Vantaggi del RAID 1
- Rispetto al disco singolo, il RAID 1 offre un'eccellente velocità di lettura e scrittura.
- Ha la capacità di tolleranza ai guasti. Se un disco si guasta, non è necessario ricostruire i dati e basta copiare i dati nel disco sostitutivo.
- È una tecnologia molto semplice e anche l'implementazione del RAID 1 è molto semplice.

Svantaggi del RAID 1

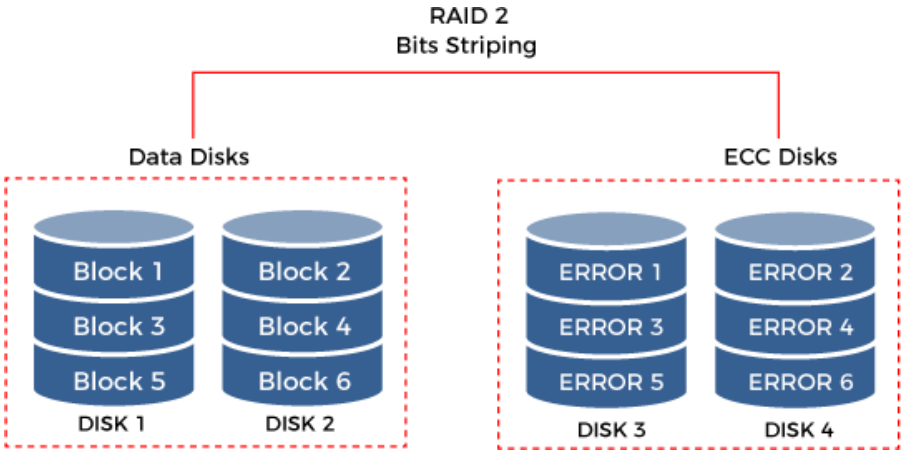
- Nel RAID 1, i dati devono essere scritti due volte. Per questo motivo, la capacità di archiviazione effettiva è solo la metà della capacità totale del disco, e questo è il principale svantaggio del RAID 1.
- Il RAID 1 è più costoso rispetto al RAID 0 perché richiede due dischi per il mirroring dei dati.
- Il software RAID 1 non consente sempre la sostituzione a caldo dei dischi guasti. Quando si spegne il computer attraverso il quale è stato attaccato il disco guasto, quest'ultimo può essere solo sostituito.

Scritto da Gabriel

- Molte persone utilizzano contemporaneamente i server e questo processo di spegnimento potrebbe non essere accettato. Per questo motivo, questi tipi di sistemi utilizzano in genere controller hardware che supportano la sostituzione a caldo dei dischi.

RAID 2

Il RAID 2 può essere chiamato anche striping a livello di bit. In RAID 2, si effettua lo striping dei bit sui dischi anziché lo striping dei blocchi sui dischi. A questo livello, sono necessari due gruppi di dischi. Il primo gruppo di dischi verrà utilizzato per scrivere i dati, mentre il secondo gruppo di dischi verrà utilizzato per scrivere i codici di correzione degli errori.
In questo livello, utilizzeremo il codice di correzione degli errori di Hamming (ECC) e poi useremo i dischi di ridondanza per memorizzare le informazioni del codice ECC. Il codice di Hamming è un tipo di codice di correzione degli errori lineare, in grado di rilevare fino a (d - 1) errori di bit e di correggere (d -1)/2 errori di bit. Dove d è un tipo di parola di codice dato dalla minima distanza di Hamming tra tutte le coppie. Se d è maggiore o uguale alla distanza di Hamming tra il modello di bit trasmesso e quello ricevuto, solo allora sarà possibile una comunicazione affidabile. Al contrario, un semplice codice di parità è in grado di rilevare solo un numero dispari di errori e non può correggere l'errore.
Quando scriviamo i dati sui dischi, il codice ECC (codice di correzione degli errori) per i dati viene valutato al volo. Successivamente, i bit dei dati vengono spogliati sui dischi di dati e, infine, il codice ECC viene scritto sui dischi di ridondanza. Quando si leggono i dati dai dischi, si utilizzano i dischi di ridondanza per leggere il codice ECC corrispondente. A questo punto verifica se i dati sono coerenti. Se necessario, esegue le correzioni appropriate al volo.
Questo processo utilizza molti dischi. Verrà configurato in varie configurazioni di dischi. Ora il RAID 2 non è più utile perché è costoso e l'implementazione del RAID 2 nel controller RAID è difficile. Ora anche l'ECC è ridondante perché i dischi rigidi sono in grado di svolgere da soli il lavoro dell'ECC.



Vantaggi di RAID 2

o Il RAID 2 utilizza il codice Hamming per la correzione degli errori.
o Può memorizzare la parità con l'aiuto di un'unità designata.

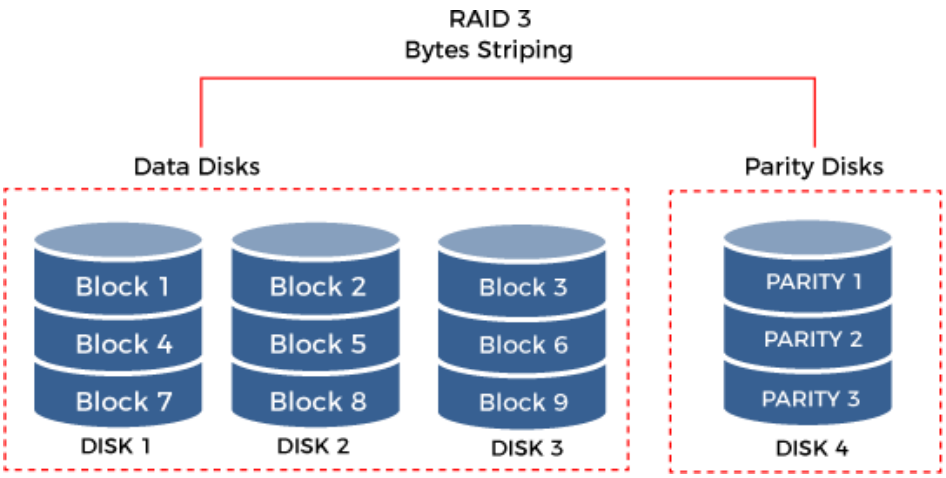
Svantaggi del RAID 2

o Il RAID 2 necessita di un'unità aggiuntiva per il rilevamento degli errori.
o Contiene un'unità aggiuntiva. Per questo motivo è costoso e ha una struttura complessa.

RAID 3

Il RAID 3 può essere chiamato anche Byte level stripping. Il funzionamento di RAID 3 è identico a quello di RAID 0, in quanto utilizza lo stripping a livello di byte, ma necessita di un disco aggiuntivo nell'array. Il RAID 3 è utilizzato per supportare un tipo speciale di processore nei calcoli del codice di parità, che può essere chiamato "disco di parità". In RAID 3, si esegue lo striping dei byte sui dischi anziché lo striping dei blocchi sui dischi. A questo livello, sono necessari più dischi di dati e un disco dedicato per memorizzare la parità. Nel processo di configurazione di RAID 3, i dati vengono suddivisi in singoli byte e quindi salvati su un disco. Per ogni riga di dati, viene determinato il disco di parità e quindi viene salvato nel disco di parità indicato. In caso di guasto, il sistema è in grado di recuperare i dati con l'aiuto dei byte di parità corrispondenti e con il calcolo appropriato dei byte rimanenti.

Sebbene questo livello sia raramente utilizzato nella pratica, presenta numerosi vantaggi: in primo luogo, è in grado di resistere in caso di danneggiamento del disco durante l'installazione. In secondo luogo, ha una velocità di lettura molto elevata. Purtroppo, il RAID 3 presenta anche molti svantaggi. In primo luogo, rispetto alla velocità di lettura, la velocità di scrittura è molto lenta a causa della necessità di calcolare il checksum. (Anche i controller hardware RAID non sono in grado di risolvere questo problema). Il secondo problema è che in caso di guasto di un disco, l'intero sistema funzionerà molto lentamente. Il RAID 3 ha la capacità di resistere ai guasti, il che significa che se un disco dell'array si guasta, sostituirà il disco danneggiato, ma il processo di sostituzione è molto costoso. Il terzo problema è che il disco viene utilizzato per calcolare le checksum, il che rappresenta il collo di bottiglia delle prestazioni dell'intero array. Nonostante la descrizione di cui sopra, RAID 3 non è in grado di mostrare una soluzione valida, affidabile ed economica. Per questo motivo il RAID 3 viene utilizzato raramente nella pratica. I sistemi basati su RAID 3 sono utilizzati per lo più per scopi di implementazione in cui file molto grandi sono riferiti da un numero ridotto di utenti.



Vantaggi del RAID 3

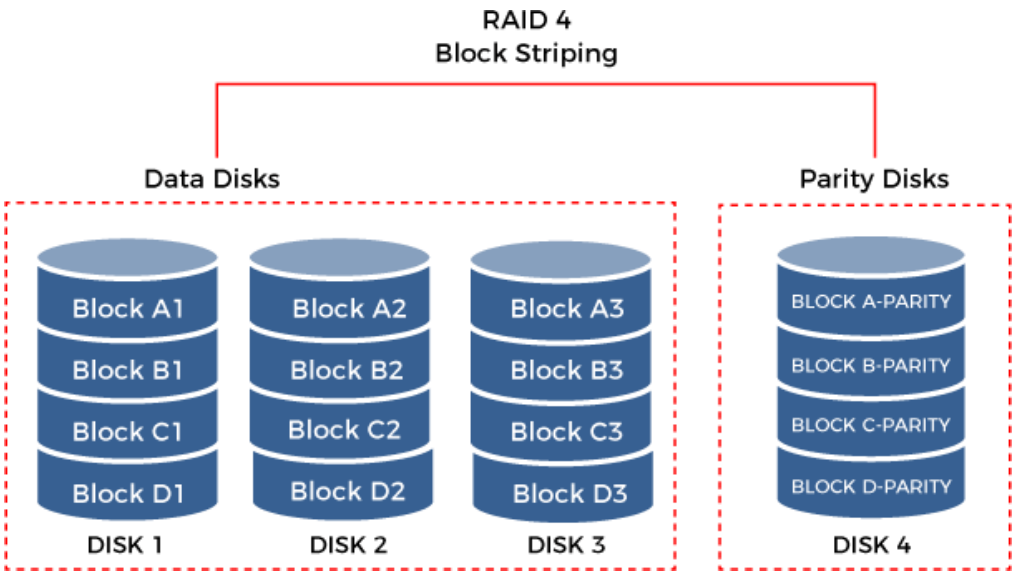
- Il RAID 3 offre un'elevata velocità di trasferimento dei dati di grandi dimensioni.
- Risolve il principale svantaggio del RAID 2, ossia la resistenza ai guasti e ai guasti del disco.

Svantaggi del RAID 3

- Se si deve trasferire solo un file di piccole dimensioni, la configurazione potrebbe essere eccessiva.
- Se si verifica un guasto del disco, la velocità di trasferimento diminuisce in modo significativo.

RAID 4

Il RAID 4 è noto come striping a livello di blocco. Il funzionamento di RAID 4 è identico a quello di RAID 3. La differenza principale è il processo di condivisione dei dati. I dati vengono suddivisi in blocchi da 16, 32, 64 o 128 GB. Come il RAID 0, i dati vengono scritti sul disco. Per ogni riga di dati scritti, viene utilizzato un disco di parità per scrivere qualsiasi blocco registrato. Ciò significa che questo livello utilizza lo striping dei dati a livello di blocco anziché a livello di byte. RAID 5 e RAID 4 hanno molte somiglianze, ma RAID 4 confina tutti i dati di parità su un singolo disco. Si può quindi dire che non utilizza la parità distributiva.
In RAID 4 è possibile completare l'implementazione e la configurazione con l'aiuto di almeno tre dischi. Il RAID 4 richiede anche un supporto hardware per eseguire i calcoli di parità. Per questo motivo, siamo in grado di recuperare i dati con l'aiuto di operazioni matematiche appropriate.



Vantaggi del RAID 4
- RAID 4 consente lo striping a livello di blocco, che permette di inviare simultaneamente le richieste di I/O.
- Fornisce un basso overhead di archiviazione. Se si aggiungono vari dischi, l'overhead diventa più basso.
- Questo livello non necessita di un controller sincronizzato o di mandrini.

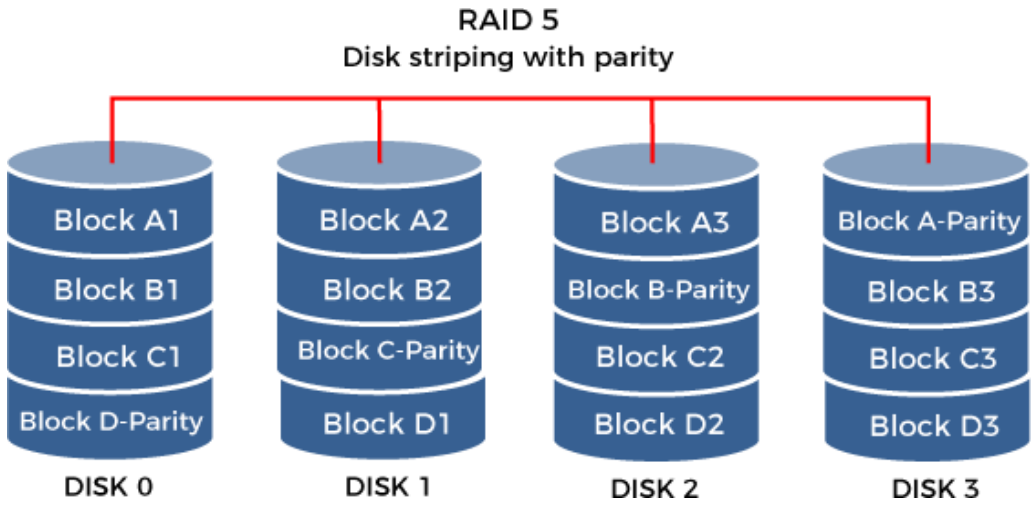
Svantaggi del RAID 4
- Contiene le unità di parità, il che può portare a un collo di bottiglia.
- Se si cerca di eseguire un'operazione di scrittura simultanea, l'operazione sarà più lenta perché le informazioni di parità vengono scritte su un solo disco.

RAID 5

RAID 5 può essere chiamato Stripping con parità. Utilizza il livello di blocco per lo striping dei dati e utilizza anche la parità distributiva. Il RAID 5 richiede un minimo di tre dischi, ma può funzionare fino a 16 dischi. È il livello RAID più sicuro. La parità è un tipo di dati binari grezzi. Il sistema RAID calcola i valori di parità e, utilizzando questi valori, crea un blocco di parità. Se un disco si guasta nel sistema RAID, il blocco di parità viene utilizzato per recuperare i dati a strisce. La maggior parte dei sistemi RAID con funzione di parità utilizza l'array per memorizzare i blocchi di parità nei dischi. A questo livello, i blocchi di dati sono suddivisi in strisce tra le unità. La somma di controllo di parità di tutti i blocchi di dati viene scritta solo su un'unità. La somma di controllo di parità non utilizza un'unità fissa, ma viene distribuita su tutte le unità. Se i dati di un blocco di dati non sono più disponibili, con l'aiuto dei dati di parità il computer può ricalcolare i dati. Ciò significa che in caso di guasto di una singola unità, il RAID 5 è in grado di resistere al guasto di qualsiasi disco dell'array senza accedere ai dati o perderli.

Scritto da Gabriel

Anche se è possibile utilizzare RAID 5 nel software, il controller consigliato è quello hardware. Questi controller possono migliorare le prestazioni di scrittura utilizzando spesso la memoria cache extra. In questo livello, le prestazioni del RAID 0 sono combinate con la ridondanza del RAID 1, ma questo processo richiede un'enorme quantità di spazio di archiviazione, che può essere circa un terzo della capacità utilizzabile. Nell'array, tutte le unità servono per le richieste di scrittura simultaneamente. Ecco perché questo livello aumenta le prestazioni di scrittura. Tuttavia, l'implicazione della scrittura può influire sulle prestazioni dell'intero disco, poiché è necessario eseguire più passaggi e ricalcoli se si apportano piccole modifiche alle strisce.
In breve, possiamo dire che RAID 5 offre affidabilità e prestazioni elevate. Ha la capacità di bilanciare le letture e le scritture ed è anche sicuro. RAID 5 memorizza la parità utilizzando lo spazio dell'intero disco e riduce anche la quantità aggregata di dati che gli utenti possono salvare. RAID 5 è un tipo di sistema eccellente a tutto tondo, che viene utilizzato per fornire prestazioni decenti e combinare un'archiviazione efficiente con un'eccellente sicurezza. È utilizzato principalmente per i server di file e applicazioni che contengono un numero limitato di unità di dati.



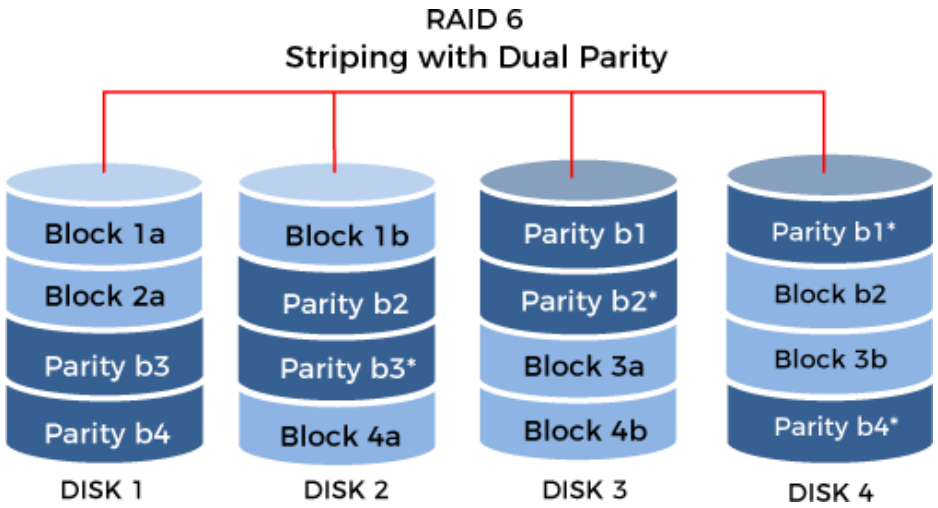
Vantaggi del RAID 5
- In RAID 5, le transazioni di scrittura dei dati sono lente a causa del calcolo della parità, mentre le transazioni di lettura dei dati sono molto veloci.
- In caso di guasto di un disco nel RAID 5, è ancora possibile accedere a tutti i dati, anche se il disco guasto viene sostituito e i dati vengono ricostruiti dal controller di archiviazione su un nuovo disco.

Svantaggi del RAID 5
- Il guasto di un disco influisce sul throughput, ma il processo è ancora accettabile.
- RAID 5 è una tecnologia complessa. Supponiamo che un disco da 4 TB nell'array di vari dischi si guasti. In questo caso, la sostituzione e il ripristino dei dati del disco guasto possono richiedere un giorno o più, in base alla velocità del controller e al carico dell'array. A questo punto, se un disco si guasta, i dati saranno persi per sempre.

RAID 6

Il RAID 6 può anche essere chiamato Striping con doppia parità. Il funzionamento del RAID 5 è identico a quello del RAID 6, con la differenza che nel RAID 6 il sistema memorizza un blocco di parità aggiuntivo su ciascun banco. Per questo motivo, viene attivata una configurazione in cui prima che l'array non sia disponibile, i due dischi possono essere guasti. Ha bisogno di due set diversi per i calcoli di parità e ha la capacità di ricostruire un array anche se due unità si guastano contemporaneamente. RAID 6 necessita di un minimo di quattro dischi e può sopportare due dischi che si guastano contemporaneamente. I due dischi saranno utilizzati per i dati, mentre i due dischi rimanenti saranno utilizzati per le informazioni di parità. Se

il numero di dischi aumenta, aumentano le possibilità di guasti multipli e la complessità della ricostruzione del set di dischi.
Rispetto al RAID 5, offre una ridondanza maggiore e aumenta anche le prestazioni di lettura. In caso di operazioni di scrittura intensive, anche questo livello soffrirà dello stesso overhead di prestazioni del server. Le prestazioni dipendono dall'architettura del sistema RAID, cioè dal software o dall'hardware. Se il sistema esegue il calcolo della parità ad alte prestazioni con l'aiuto di un software di elaborazione incluso, e se si trova nel firmware, le prestazioni ne risentiranno.
In RAID 6, le possibilità che due dischi si guastino contemporaneamente sono molto ridotte. Nel sistema RAID 5, se un disco si guasta, ci vorranno ore, giorni o più per sostituirlo con un nuovo disco. A quel punto, se un altro disco si guasta, perderemo tutti i nostri dati per sempre. Nel sistema RAID 6, invece, l'array RAID sopravvive anche al secondo guasto.



Vantaggi del RAID 6

- In RAID 6, le transazioni di lettura dei dati sono molto veloci, proprio come in RAID 5.
- È più sicuro del RAID 5 perché se due dischi si guastano, siamo in grado di accedere a tutti i nostri dati anche quando il sistema viene sostituito con i dischi guasti.

Svantaggi del RAID 6

- Nel RAID 6 è necessario calcolare la parità aggiuntiva. Per questo motivo, le transazioni di scrittura dei dati nel RAID 6 sono più lente rispetto al RAID 5. Possono essere più lente del 20% rispetto al RAID 5.
- Se si verifica un guasto del disco, il throughput ne risentirà, ma il processo è comunque accettabile.
- RAID 6 è una tecnologia complessa. In caso di guasto di un disco in un array RAID, la ricostruzione dell'array può richiedere molto tempo.

Un paragone finale:

Level	Advantages	Disadvantages	Applications
0	I/O performance is greatly improved by spreading the I/O load across many channels and drives No parity calculation overhead is involved Very simple design Easy to implement	The failure of just one drive will result in all data in an array being lost	Video production and editing Image Editing Pre-press applications Any application requiring high bandwidth
1	100% redundancy of data means no rebuild is necessary in case of a disk failure, just a copy to the replacement disk Under certain circumstances, RAID 1 can sustain multiple simultaneous drive failures Simplest RAID storage subsystem design	Highest disk overhead of all RAID types (100%)—inefficient	Accounting Payroll Financial Any application requiring very high availability
2	Extremely high data transfer rates possible The higher the data transfer rate required, the better the ratio of data disks to ECC disks Relatively simple controller design compared to RAID levels 3, 4, & 5	Very high ratio of ECC disks to data disks with smaller word sizes—inefficient Entry level cost very high—requires very high transfer rate requirement to justify	No commercial implementations exist/not commercially viable
3	Very high read data transfer rate Very high write data transfer rate Disk failure has an insignificant impact on throughput Low ratio of ECC (parity) disks to data disks means high efficiency	Transaction rate equal to that of a single disk drive at best (if spindles are synchronized) Controller design is fairly complex	Video production and live streaming Image editing Video editing Prepress applications Any application requiring high throughput
4	Very high Read data transaction rate Low ratio of ECC (parity) disks to data disks means high efficiency	Quite complex controller design Worst write transaction rate and Write aggregate transfer rate Difficult and inefficient data rebuild in the event of disk failure	No commercial implementations exist/not commercially viable
5	Highest Read data transaction rate Low ratio of ECC (parity) disks to data disks means high efficiency Good aggregate transfer rate	Most complex controller design Difficult to rebuild in the event of a disk failure (as compared to RAID level 1)	File and application servers Database servers Web, e-mail, and news servers Intranet servers Most versatile RAID level
6	Provides for an extremely high data fault tolerance and can sustain multiple simultaneous drive failures	More complex controller design Controller overhead to compute parity addresses is extremely high	Perfect solution for mission critical applications

Esercizi su dischi magnetici

Es1: Si supponga di sapere che per trasferire 64KB di dati da un dato disco rigido occorra un tempo totale di circa 9,728571 ms (senza contare l'attesa che il dispositivo ed uno dei suoi canali sia libero). Sapendo che:

- il disco possiede 524288 tracce,
- ogni settore memorizza 512B,
- il tempo medio di posizionamento della testina è 0,8 ms,
- la velocità di rotazione del disco è di 4200 rpm

si calcoli il numero totale di byte che il disco può memorizzare.

Soluzione: Sappiamo che

$$T_S = 0{,}8 \text{ ms} \quad \text{e} \quad T_L = (1000/\underset{\text{millesimi secondo}}{} \ (4200/60)\ )/2 \approx 7{,}142857 \ \underset{\text{rotazioni al secondo}}{} \ \text{ms}$$

e che il tempo totale di trasferimento è dato da

$$T = T_S + T_L + T_t$$

dove il tempo di trasferimento (in millisecondi) è dato dalla formula

$$T_t = \frac{b}{rN} \times 1000$$

b #byte da trasferire
N #byte per traccia
r velocità rotazione (in rotazioni per sec.)

Si può risalire al numero di byte per traccia con la seguente formula

$$N = \frac{b}{rT_t} \times 1000 \quad \text{dove} \quad T_t = T - T_S - T_L \approx 9{,}728571 - 0{,}8 - 7{,}142857 = 1{,}785714$$

Aiuto:

Ricordarsi che il tempo di trasferimento (in millisecondi) è dato dalla formula

$$T_t = \frac{b}{rN} \times 1000$$

b #byte da trasferire
N #byte per traccia
r velocità rotazione (in rotazioni per sec.)

Quindi

$$N = \frac{b}{rT_t} \times 1000 = 524288$$

Poiché il disco possiede 524288 tracce, si ha una capacità totale di memorizzazione di:

524288*524288 = 268435456KB, equivalenti a 256GB

Es2:
Sia dato un disco rigido con le seguenti caratteristiche:

- capacità di 512GB;
- 4 piatti (8 facce);
- 524288 tracce per faccia e 1024 settori per traccia;
- velocità di rotazione di 10000 rpm;
- tempo medio di posizionamento della testina di 1,4 ms.

Si calcoli il tempo totale medio di trasferimento (in millisecondi, e senza contare l'attesa che il dispositivo ed uno dei suoi canali sia libero; sul libro riferito come tempo di accesso) che occorre per trasferire 32KB, assumendo che i byte da trasferire siano memorizzati:

a) in settori contigui di una singola traccia;

b) in settori contigui di un cilindro.

Soluzione a): Sappiamo che

$$\boldsymbol{T_S} = 1{,}4 \text{ ms e } \boldsymbol{T_L} = (1000/\,(10000/60)\,)\,/\,2 \approx 3{,}0 \text{ ms}$$

e che il tempo totale di trasferimento è dato da

$$\boldsymbol{T} = \boldsymbol{T_S} + \boldsymbol{T_L} + \boldsymbol{T_t}$$

dove il tempo di trasferimento (in millisecondi) è dato dalla formula

$$T_t = \frac{b}{rN} \times 1000$$

b	*#byte da trasferire*
N	*#byte per traccia*
r	*velocità rotazione (in rotazioni per sec.)*

Il numero di byte per faccia sarà dato dalla capacità totale del disco diviso il numero di facce

$$512\text{GB} / 8 = 2^{39} / 2^3 = 2^{36}$$

Il numero di byte per traccia N sarà dato dalla capacità totale di una faccia diviso il numero di tracce ($524288 = 2^{19}$)

$$N = 2^{36} / 2^{19} = 2^{17}$$

Quindi

$$\begin{aligned} T_t &= [1000 \times 32KB] / [(10000/60) \times 2^{17}] \\ &= [1000 \times 2^{15}] / [(10000/60) \times 2^{17}] \\ &= 1{,}5 \text{ ms} \end{aligned}$$

Pertanto il tempo totale di accesso è

$$T = 1{,}4 + 3{,}0 + 1{,}5 = 5{,}9 \text{ ms}$$

Soluzione b): come nel caso a), però essendo i settori memorizzati in un cilindro, si possono leggere simultaneamente i settori posti su tracce collocate nella medesima posizione di facce diverse. Pertanto il tempo di trasferimento dei 32KB deve essere diviso per 8 (numero facce):

$$T = 1{,}4 + 3{,}0 + 1{,}5/8 = 4{,}5875 \text{ ms}$$

Es3:

Sia dato un disco rigido con le seguenti caratteristiche:

- capacità di 128GB;
- 2 piatti (4 facce);
- 65536 tracce per faccia e 2048 settori per traccia;
- velocità di rotazione di 4200 rpm;
- tempo medio di posizionamento della testina di 2,8 ms.

Sapendo che il tempo totale medio di trasferimento (in millisecondi, e senza contare l'attesa che il dispositivo ed uno dei suoi canali sia libero; sul libro riferito come tempo di accesso) che occorre per trasferire x byte (assumendo che i byte da trasferire siano memorizzati in settori contigui di una singola traccia) è di 11,728571 ms, si dica:

a) quanti byte x sono stati trasferiti;
b) quanti settori sono coinvolti nel trasferimento.

Soluzione a): Sappiamo che

$$T_S = 2{,}8 \text{ ms e } T_L = (1000/(4200/60)) / 2 \approx 7{,}142857 \text{ ms}$$

e che il tempo totale di trasferimento è dato da

$$T = T_S + T_L + T_t = 11{,}728571 \text{ ms}$$

dove il tempo di trasferimento (in millisecondi) è dato dalla formula

$$T_t = \frac{b}{rN} \times 1000$$

b	*#byte da trasferire*
N	*#byte per traccia*
r	*velocità rotazione (in rotazioni per sec.)*

Bisogna risalire al valore di b.
Il numero di byte per faccia sarà dato dalla capacità totale del disco diviso il numero di facce

$$128GB / 4 = 2^{37} / 2^2 = 2^{35}$$

Il numero di byte per traccia N sarà dato dalla capacità totale di una faccia diviso il numero di tracce ($65536 = 2^{16}$)

$$N = 2^{35} / 2^{16} = 2^{19}$$

Quindi

$$\begin{aligned} b &= \boldsymbol{T_t} \times [(4200/60) \times 2^{19}] /1000 \\ &= [11{,}728571 - 2{,}8 - 7{,}142857] \times [(4200/60) \times 2^{19}] /1000 \\ &= 65536 \text{ (arrotondando alla potenza di 2 più vicina)} \\ &= 64\text{KB} \end{aligned}$$

Soluzione b): il numero di settori coinvolti nel trasferimento può essere stabilito andando a calcolare la dimensione di un singolo settore:

$$\begin{aligned} \textit{dimensione settore (in byte)} &= N/(\textit{numero settori per traccia}) \\ &= 2^{19} / 2048 = 2^8 \end{aligned}$$

Quindi il numero di settori trasferiti è dato da:

$$b/(\textit{dimensione settore}) = 2^{16} / 2^8 = 2^8 = 256$$

Es4: La struttura dell'informazione memorizzata su un disco è organizzata in cilindri e settori. Si considerino i seguenti tre principali algoritmi di selezione della prossima ricerca di cilindro:

- ***First-Come First-Served***:
  le richieste di posizionamento sono servite nell'ordine di arrivo, senza alcun riordinamento.
- ***Shortest Seek First***:
  la prossima richiesta da servire è la più vicina al cilindro corrente tra quelle in attesa.
- ***Elevator Algorithm***:
  la testina avanza o retrocede verso il cilindro più vicino senza mai cambiare direzione fin quando esistano richieste pendenti in quella direzione.

Sia data una sequenza di richieste di lettura/scrittura per i cilindri:

10, 20, 15, 5, 40, 8, 35

pervenute nell'ordine mostrato.

Assumendo:

- un costo temporale di 5 millisecondi per lo spostamento della testina dal cilindro su cui si trova ad uno dei cilindri adiacenti
- che la testina, in posizione iniziale, sia sul cilindro 15

si determini il costo complessivo di posizionamento al termine della sequenza data per i 3 algoritmi indicati, illustrando anche l'ordine di selezione corrispondente.

Soluzione: L'algoritmo FCFS effettuerà la seguente scansione, con l'associato costo di posizionamento:

$$15 \rightarrow_5 10 \rightarrow_{10} 20 \rightarrow_5 15 \rightarrow_{10} 5 \rightarrow_{35} 40 \rightarrow_{32} 8 \rightarrow_{27} 35,$$

che comporta un onere complessivo di 124 spostamenti di cilindro, pari a 620 millisecondi.

L'algoritmo SSF selezionerà invece il seguente ordine di posizionamento, che è quello di maggior efficacia tra quelli compatibili con la logica dell'algoritmo:

$$15 \rightarrow_0 15 \rightarrow_5 10 \rightarrow_2 8 \rightarrow_3 5 \rightarrow_{15} 20 \rightarrow_{15} 35 \rightarrow_5 40,$$

con un costo complessivo di 45 spostamenti di cilindro, pari a 225 millisecondi.
La sequenza alternativa è:

$$15 \rightarrow_0 15 \rightarrow_5 20 \rightarrow_{10} 10 \rightarrow_2 8 \rightarrow_3 5 \rightarrow_{30} 35 \rightarrow_5 40,$$

con un costo complessivo di 55 spostamenti di cilindro, pari a 275 millisecondi.
L'algoritmo EA, invece, si comporterà in maniera diversa a seconda della direzione di movimento iniziale. Assumendo che essa sia verso l'alto, ossia verso i cilindri di posizione uguale o superiore a 15, otterremo la seguente sequenza:

$$15 \rightarrow_0 15 \rightarrow_5 20 \rightarrow_{15} 35 \rightarrow_5 40 \rightarrow_{30} 10 \rightarrow_2 8 \rightarrow_3 5,$$

con un costo complessivo di 60 spostamenti di cilindro, pari a 300 millisecondi. A fronte di una direzione iniziale in senso discendente otterremo, invece, la medesima sequenza selezionata dall'algoritmo SSF con 45 spostamenti complessivi.

Es5:

Sia dato un disco rigido con le seguenti caratteristiche:

- capacità di 128GB;
- 4 piatti (8 facce);
- 65536 tracce per faccia e 1024 settori per traccia;
- velocità di rotazione di 7200 rpm;
- tempo medio di posizionamento della testina di 8,5 ms.

Si calcoli il tempo totale medio di trasferimento (in millisecondi, e senza contare l'attesa che il dispositivo ed uno dei suoi canali sia libero; sul libro riferito come tempo di accesso) che occorre per trasferire 64KB, assumendo che i byte da trasferire siano memorizzati:

a) in settori contigui di una singola traccia;

b) in settori contigui di un cilindro.

Soluzione a): Sappiamo che

$$\boldsymbol{T_S} = 8{,}5 \text{ ms e } \boldsymbol{T_L} = (1000/\,(7200/60)\,)\,/\,2 \approx 4{,}166 \text{ ms}$$

e che il tempo totale di trasferimento è dato da

$$\boldsymbol{T} = \boldsymbol{T_S} + \boldsymbol{T_L} + \boldsymbol{T_t}$$

dove il tempo di trasferimento (in millisecondi) è dato dalla formula

$$T_t = \frac{b}{rN} \times 1000$$

b	*#byte da trasferire*
N	*#byte per traccia*
r	*velocità rotazione (in rotazioni per sec.)*

Il numero di byte per faccia sarà dato dalla capacità totale del disco diviso il numero di facce

$$128\text{GB} / 8 = 2^{37} / 2^3 = 2^{34}$$

Il numero di byte per traccia N sarà dato dalla capacità totale di una faccia diviso il numero di tracce ($65536 = 2^{16}$)

$$N = 2^{34} / 2^{16} = 2^{18}$$

Quindi

$$\begin{aligned} T_t &= [1000 \times 64\text{KB}] / [(7200/60) \times 2^{18}] \\ &= [1000 \times 2^{16}] / [(7200/60) \times 2^{18}] \\ &= 2{,}0833 \text{ ms} \end{aligned}$$

Pertanto il tempo totale di accesso è

$$T = 8{,}5 + 4{,}166 + 2{,}0833 = 14{,}75 \text{ ms}$$

Soluzione b): come nel caso a), però essendo i settori memorizzati in un cilindro, si possono leggere simultaneamente i settori posti su tracce collocate nella medesima posizione di facce diverse. Pertanto il tempo di trasferimento dei 64KB deve essere diviso per 8 (numero facce):

$$T = = 8{,}5 + 4{,}166 + 2{,}0833/8 = 12{,}927 \text{ ms}$$

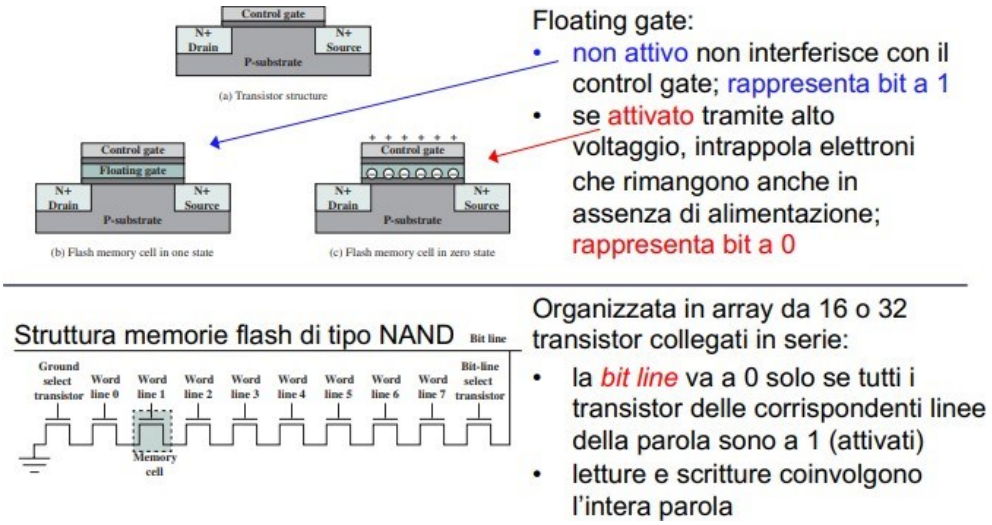
Dischi SSD (Solid State Drives)/Dischi a stato solido

Le unità a stato solido sono dispositivi di memorizzazione non volatili in grado di contenere grandi quantità di dati. Utilizzano memorie flash NAND, che offrono il vantaggio di non avere parti meccaniche in movimento e quindi un accesso immediato ai dati.
Sebbene le unità SSD svolgano la stessa funzione delle unità disco, i loro componenti interni sono molto diversi. A differenza dei dischi rigidi, le unità SSD non hanno parti in movimento (per questo sono chiamate unità allo stato solido). Invece di memorizzare i dati su piatti magnetici, le unità SSD li memorizzano utilizzando la memoria flash. Poiché le unità SSD non hanno parti in movimento, non devono "girare" durante lo stato di sospensione e non devono spostare la testina dell'unità in parti diverse dell'unità per accedere ai dati. Pertanto, le unità SSD possono accedere ai dati più velocemente delle unità HDD.

Le unità SSD presentano anche altri vantaggi rispetto alle unità disco. Ad esempio, le prestazioni di lettura di un disco rigido diminuiscono quando i dati vengono frammentati, ovvero suddivisi in più posizioni del disco. Le prestazioni di lettura di un'unità SSD non diminuiscono in base alla posizione in cui i dati sono memorizzati sul disco. Pertanto, la deframmentazione di un'unità SSD non è necessaria. Poiché le unità SSD non memorizzano i dati in modo magnetico, non sono soggette a perdite di dati dovute a forti campi magnetici in prossimità dell'unità. Inoltre, poiché le unità SSD non hanno parti in movimento, le possibilità di guasti meccanici sono molto minori. Le unità SSD sono anche più leggere, più silenziose e consumano meno energia rispetto alle unità disco. Per questo motivo le unità SSD sono diventate una scelta popolare per i computer portatili.

Sebbene le unità SSD presentino molti vantaggi rispetto alle unità disco, hanno anche alcuni svantaggi. Poiché la tecnologia delle unità SSD è molto più recente di quella dei dischi rigidi tradizionali, il prezzo delle unità SSD è sostanzialmente più alto. All'inizio del 2011, le unità SSD costavano per gigabyte circa 10 volte di più di un'unità disco. Pertanto, la maggior parte delle unità SSD vendute oggi ha una capacità molto inferiore rispetto a quella di unità disco paragonabili. Inoltre, hanno un numero limitato di cicli di scrittura, il che può causare un deterioramento delle prestazioni nel tempo. Fortunatamente, le unità SSD più recenti hanno migliorato l'affidabilità e dovrebbero durare diversi anni prima che si noti una riduzione delle prestazioni. Con il miglioramento della tecnologia SSD e il continuo calo dei prezzi, è probabile che le unità allo stato solido comincino a sostituire i dischi rigidi per la maggior parte degli scopi.

In particolare, essendo di tipo NAND:



Le unità SSD si basano su tecnologie di memoria flash che consentono di scrivere, leggere e cancellare i dati più volte. La memoria flash è disponibile in due varianti: NOR e NAND. Sebbene ognuna di esse offra vantaggi e svantaggi (una discussione che esula dallo scopo di questo articolo), la NAND è emersa come la tecnologia preferita perché offre tempi di cancellazione e scrittura più rapidi. La maggior parte delle unità SSD contemporanee si basa sulla tecnologia NAND flash, che è il motivo per cui è oggetto di questo articolo.

Un'unità SSD aziendale contiene più chip NAND flash per la memorizzazione dei dati. Ogni chip contiene uno o più die e ogni die contiene uno o più piani. Un piano è diviso in blocchi e un blocco è diviso in pagine. I blocchi e le pagine sono i più importanti, non perché si configurino o si manipolino direttamente, ma per il modo in cui i dati vengono scritti, letti e cancellati su un chip NAND. I dati vengono letti e scritti a livello di pagina, ma cancellati a livello di blocco, come illustrato nella Figura 1.

In questo caso, ogni pagina è di 4 kibibyte (KiB) e ogni blocco è di 256 KiB, il che equivale a 64 pagine per blocco. (Un kibibyte è pari a 1024 byte. I kibibyte vengono talvolta utilizzati al posto dei kilobyte perché sono più precisi. Un kilobyte può corrispondere a 1000 byte o a 1024 byte, a seconda dell'uso che se ne fa). Ogni volta che l'unità SSD legge o scrive dati, lo fa in pezzi da 4-KiB, ma ogni volta che l'unità cancella dati, esegue un'operazione da 256-KiB. Questa differenza di scrittura/cancellazione ha serie conseguenze quando si aggiornano i dati, come si vedrà più avanti nell'articolo.

All'interno della cella NAND

Una pagina è composta da più celle che contengono ciascuna uno o più bit di dati. Un bit di dati è rappresentato da uno stato di carica elettrica, determinato dagli elettroni intrappolati tra gli strati isolanti della cella. Ogni bit viene registrato come carico (0) o non carico (1), fornendo la formula binaria necessaria per rappresentare i dati.
Gli attuali chip NAND flash utilizzano celle a gate flottante o celle a trappola di carica. Fino a poco tempo fa la maggior parte delle memorie NAND si basava su tecnologie a gate flottante, in cui gli elettroni sono intrappolati tra due strati di ossido in una regione chiamata gate flottante. Lo strato di ossido inferiore è abbastanza sottile da consentire il passaggio degli elettroni quando viene applicata una tensione al substrato sottostante. Gli elettroni entrano nel floating gate durante un'operazione di scrittura ed escono dal floating gate durante un'operazione di cancellazione.

Il problema dell'approccio a gate flottante è che ogni volta che viene applicata la tensione e gli elettroni passano attraverso lo strato di ossido, quest'ultimo si degrada leggermente. Più sono le operazioni di scrittura e cancellazione, maggiore è il degrado, fino a quando la cella potrebbe non essere più utilizzabile. Si tenga presente, tuttavia, che le tecnologie delle unità SSD hanno fatto passi da gigante, rendendole più affidabili e durevoli, oltre che in grado di fornire maggiori prestazioni e memorizzare più dati. Allo stesso tempo, il loro prezzo continua a scendere, rendendole molto più competitive.

I produttori continuano a esplorare nuove tecnologie per migliorare le unità SSD. Ad esempio, molti produttori stanno adottando tecnologie a trappola di carica per le loro celle NAND. Le celle a trappola di carica sono simili alle celle a gate flottante, ma utilizzano materiali isolanti e metodologie diverse per intrappolare gli elettroni, ottenendo celle meno soggette a usura. Tuttavia, le tecnologie a trappola di carica comportano problemi di affidabilità, per cui nessuno dei due approcci è ideale.

Naturalmente le tecnologie floating gate e charge gate sono molto più complesse, ma questo dovrebbe darvi un'idea di cosa sta succedendo, nel caso in cui vi imbattiate in questi termini. Ma sappiate anche che le tecnologie dei gate sono solo una parte dell'equazione quando si tratta di capire la struttura delle celle NAND.
In effetti, la preoccupazione maggiore quando si valutano le unità SSD è il numero di bit memorizzati in ogni cella. Le attuali unità SSD accettano da uno a quattro bit per cella, con un numero correlato di stati di carica per cella, come mostrato nella tabella seguente. Si noti che i produttori stanno lavorando anche su flash con celle a cinque bit, denominate penta-level cell (PLC), ma la giuria non è ancora convinta di questa tecnologia.

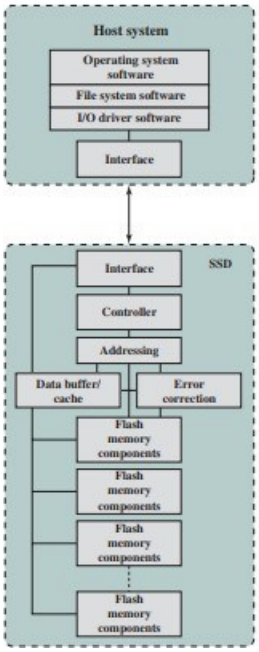
Dischi SSD: vantaggi

- Operazioni di I/O ad alte prestazioni al secondo (IOPS): aumenta notevolmente le prestazioni dei sottosistemi di I/O
- Durata: meno suscettibile a urti e vibrazioni
- Maggiore durata: gli SSD non sono soggetti a usura meccanica
- Consumo energetico inferiore: gli SSD consumano molta meno energia rispetto agli HDD di dimensioni comparabili
- Funzionalità più silenziose e più fredde: meno spazio richiesto, costi energetici inferiori e più ecologici
- Tempi di accesso e tassi di latenza inferiori: oltre 10 volte più veloci dei dischi rotanti in un HDD

	NAND Flash Drives	Seagate Laptop Internal HDD
File copy/write speed	200–550 Mbps	50–120 Mbps
Power draw/battery life	Less power draw, averages 2–3 watts, resulting in 30+ minute battery boost	More power draw, averages 6–7 watts and therefore uses more battery
Storage capacity	Typically not larger than 512 GB for notebook size drives; 1 TB max for desktops	Typically around 500 GB and 2 TB max for notebook size drives; 4 TB max for desktops
Cost	Approx. $0.50 per GB for a 1-TB drive	Approx. $0.15 per GB for a 4-TB drive

HDD godono di un vantaggio in termini di costo per bit e di capacità, ma queste differenze si stanno via via riducendo

Dischi SSD: organizzazione



Sistema Host:

- per accedere ai dati sul disco, il s.o. richiama il software del *file system*, che a sua volta, richiama il software del driver di I/O, che fornisce l'accesso host al particolare prodotto SSD
- il componente di interfaccia si riferisce all'interfaccia fisica ed elettrica tra il processore host e l'SSD

SSD:

- *Controller*: fornisce l'interfacciamento a livello di dispositivo SSD e l'esecuzione del firmware
- *Indirizzamento*: logica che esegue la funzione di selezione tra i componenti della memoria flash
- *Buffer/cache dati*: RAM ad alta velocità per compensare velocità e aumentare il *throughput* dei dati
- *Correzione degli errori*: logica per il rilevamento e la correzione degli errori
- *Componenti della memoria flash*: singoli chip flash NAND

I dischi SSD hanno due problemi:

- performance che decadono con l'uso
  - file memorizzati in pagine di 4KB, tipicamente non in pagine contigue
  - grandezza del blocco della memoria flash: 512 KB (128 pagine)
  - scrittura in una pagina:
    1. l'intero blocco che contiene la pagina deve essere letto dalla memoria flash e inserito in un buffer RAM, dove la pagina è aggiornata
    2. prima che il blocco possa essere riscritto nella memoria flash, è necessario cancellare l'intero blocco della memoria flash
    3. l'intero blocco dal buffer viene riscritto nella memoria flash
  - con l'uso i file si frammentano (pagine memorizzate su blocchi diversi) e le prestazioni decadono

<u>Soluzioni</u>:

- over-provisioning (non è altro che uno spazio del disco SSD non allocato, funzionale a garantire un numero adeguato di celle sostituibili a quelle che raggiungono il limite del ciclo di programmazione e cancellazione, così da prolungare la vita dello stesso SSD, cancellazione pagine inattive)
- comando TRIM (Il comando SSD TRIM contrassegna semplicemente i dati non validi e indica all'unità SSD di ignorarli durante il processo di garbage collection. L'SSD deve quindi spostare meno pagine durante la garbage collection, riducendo così il numero totale di cicli di programmazione/cancellazione (cicli P/E) del supporto NAND flash e prolungando la durata dell'SSD)

- Numero limitato di scritture: intorno a 100.000 scritture

<u>Soluzioni</u>:

- cache front-ending (tenere copie dei metadati e dei dati stessi, tali da poterli mantenere in locale)
- distribuzione scritture
- gestione blocchi esauriti, RAID, stima lunghezza vita blocchi

Esercizio

es8

Sia data la seguente sequenza di indirizzi in lettura (l) o scrittura (s) emessi dalla CPU e che la memoria abbia il contenuto esadecimale mostrato di seguito:

#	indirizzo (binario)	l/s	parola scritta (HEX)
1	000100001000	s	34A2239E
2	000100001100	l	
3	000100001100	l	
4	000100011000	s	9F33ABC1
5	000100011000	l	
6	000100011100	s	91DD39FA
7	000100000100	l	
8	000100000000	l	

ind	byte	ind	byte	ind	byte	ind	byte	
100	08	101	0A	102	D7	103	02	blocco
104	1F	105	00	106	80	107	E0	
108	AE	109	73	10A	AF	10B	23	blocco
10C	A1	10D	42	10E	90	10F	75	
110	B9	111	16	112	FD	113	D0	blocco
114	0A	115	07	116	03	117	71	
118	3E	119	D3	11A	71	11B	23	blocco
11C	A1	11D	8A	11E	90	11F	15	
120	F9	121	86	122	A0	123	00	blocco
124	E9	125	16	126	05	127	00	

Si assuma che la dimensione di parola sia di 4 byte (con memoria indirizzata al byte) e la presenza di una cache di ampiezza 32B, dimensione di blocco 8B, inizialmente vuota, e ad associazione a 2 vie (politica di rimpiazzo LRU, politica di scrittura write-back e gestione dei miss in scrittura con la politica write allocate).

Si mostri come sia il contenuto della cache che il contenuto della memoria cambia.

Soluzione (da compilare)

- Indicare di seguito in quali campi (e la loro dimensione) gli indirizzi emessi dalla CPU sono suddivisi:
  3 campi: etichetta, insieme, parola. Si osserva che i 2 bit più a destra indicano la posizione del byte all'interno della parola, il campo parola sarà costituito da 1 bit poiché un blocco è costituito da 2 parole (8B/4B), il campo set da 1 bit in quanto la cache contiene 32B/8B = 4 linee, raggruppate in 2 gruppi da 2 linee, e il campo etichetta dai restanti 8 bit più a sinistra.
- Indicare di seguito in quante linee/set la cache è suddivisa:

  2 set, ognuno composto da 2 linee

Indicare l'evoluzione della cache e della modifica della memoria nello schema sottostante:

Indirizzo	hit/ miss	Cache *(per ogni linea di cache indicare il contenuto del campo tag)*	Modifica memoria *M[ind.] = contenuto*
108_{HEX} 000100001000	miss	set 0 / set 1 *linea 0* [AE 73 AF 23 A1 42 90 75] etichetta: 10_{HEX} ↓ w.a. *linea 0* [34 A2 23 9E A1 42 90 75]* etichetta: 10_{HEX}	
$10C_{HEX}$ 000100001100	hit	*linea 0* [34 A2 23 9E A1 42 90 75]* etichetta: 10_{HEX}	

Indirizzo	hit/ miss	Cache *(per ogni linea di cache indicare il contenuto del campo tag)*	Modifica memoria *M[ind.] = contenuto*
$10C_{HEX}$ 000100001100	hit	*linea 0* [34 A2 23 9E A1 42 90 75]* etichetta: 10_{HEX}	
118_{HEX} 000100011000	miss	*linea 0* [34 A2 23 9E A1 42 90 75]* etichetta: 10_{HEX} *linea 1* [3E D3 71 23 A1 8A 90 15] etichetta: 11_{HEX} ↓ w.a. *linea 1* [9F 33 AB C1 A1 8A 90 15]* etichetta: 11_{HEX}	
118_{HEX} 000100011000	hit	*linea 0* [34 A2 23 9E A1 42 90 75]* etichetta: 10_{HEX} *linea 1* [9F 33 AB C1 A1 8A 90 15]* etichetta: 11_{HEX}	
$11C_{HEX}$ 000100011100	hit	*linea 0* [34 A2 23 9E A1 42 90 75]* etichetta: 10_{HEX} *linea 1* [9F 33 AB C1 91 DD 39 FA]* etichetta: 11_{HEX}	
104_{HEX} 000100000100	miss	*linea 0* [08 0A D7 02 1F 00 80 E0] etichetta: 10_{HEX}	*linea 0* [34 A2 23 9E A1 42 90 75]* etichetta: 10_{HEX} *linea 1* [9F 33 AB C1 91 DD 39 FA]* etichetta: 11_{HEX}
100_{HEX} 000100000000	hit	*linea 0* [08 0A D7 02 1F 00 80 E0] etichetta: 10_{HEX}	*linea 0* [34 A2 23 9E A1 42 90 75]* etichetta: 10_{HEX} *linea 1* [9F 33 AB C1 91 DD 39 FA]* etichetta: 11_{HEX}

Memorie esterne: CD-ROM

Il CD-ROM (Compact disc read-only memory) è un dispositivo di memorizzazione che può essere letto ma non scritto.
Il CD-ROM è stata una convenzione comune per la distribuzione di audio e altri dati nel corso degli anni, prima che le piccole unità flash a stato solido e altri dispositivi iniziassero a prendere il sopravvento.

Così come il nastro magnetico aveva sostituito il vinile, il compact disc ha sostituito il nastro magnetico come mezzo durevole e semplice per memorizzare le informazioni.
Per molti versi, il CD-ROM è stato l'ultimo metodo di archiviazione fisica dei dati, coincidente con l'uso dei floppy disk per i computer. Al contrario, oggi l'archiviazione e la trasmissione dei dati sono per lo più "completamente digitali", nel senso che piccoli pezzi di hardware possono gestire le informazioni che sarebbero state inserite in decine di singoli compact disc o floppy disk.

Poiché i compact disc sono diventati un formato di dati comune sia per la musica che per altri tipi di dati, i CD scrivibili consentono agli utenti di scaricare i dati dai loro computer per utilizzarli in altri dispositivi, ad esempio per replicare le canzoni e le playlist da utilizzare negli impianti stereo dotati di funzionalità CD.

Scritto da Gabriel

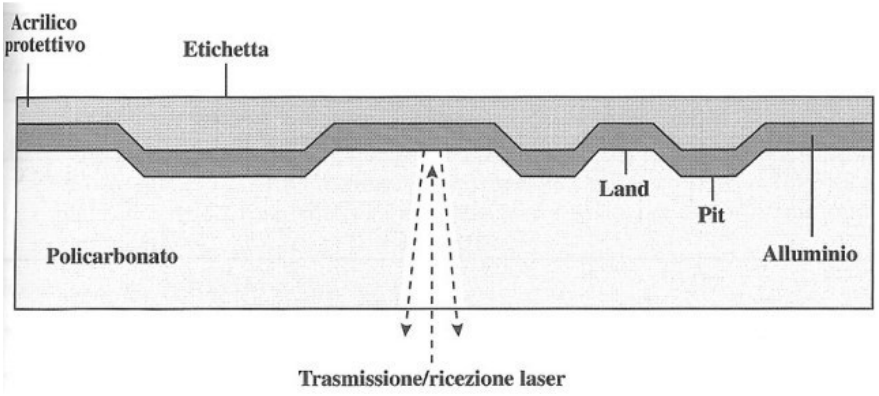
Quando i compact disc sono diventati utili per memorizzare e distribuire software oltre alla musica, le aziende hanno lavorato su protocolli tecnici specifici per i diversi tipi di dati digitali scritti sui prodotti CD-ROM. Questi protocolli continuano ad aiutare a gestire video, singoli file e diversi tipi di dati che possono essere presenti su un compact disc.

Caratteristiche principali

- Concepiti originariamente per dati audio
- 650 MB memorizzano più di 70 minuti audio
- Dischi di policarbonato rivestiti con materiale altamente riflettente (di solito alluminio)
- Dati memorizzati come microscopici pozzetti (pit)
- Lettura tramite laser
- Densità di memorizzazione costante
- Velocità lineare costante

Questi passaggi riguardano la lettura di un CD-ROM:

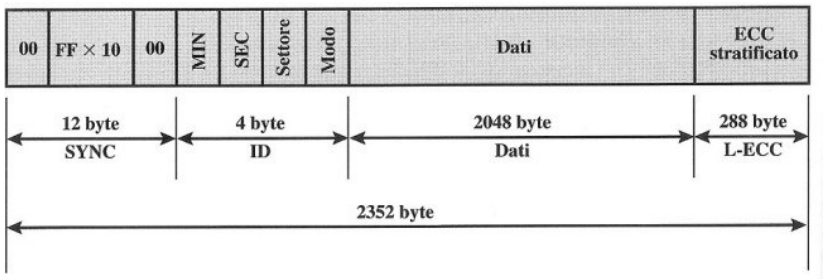
- Una singola traccia scorre a spirale dal centro del disco verso l'esterno. La traccia è costituita da terre e buche che rappresentano gli uni e gli zeri dei dati binari.
- Un laser a bassa potenza viene puntato su una superficie metallica e la riflessione viene catturata da un sensore a fotodiodo; le terre si riflettono in modo diverso rispetto ai pit, il che significa che è in grado di distinguere uno 0 e un 1.
- Il disco gira e i laser seguono la traccia.
- I dati binari vengono messi insieme e il CD-ROM viene letto.



Velocità lettore CD-ROM

- Audio: singola velocità
  - Velocità lineare costante
- 1.2 ms^{-1}
- Traccia (a spirale) lunga 5.27 km
- memorizza 4391 secondi = 73.2 minuti
- Altre velocità sono riferite come multipli
- Per esempio: 24x
- La velocità dichiarata è quella massima che il lettore può raggiungere

Formato dati CD-ROM



- Modo 0=campo dati vuoto
- Modo 1=2048 byte dati+correzione errori
- Modo 2=2336 byte dati

Con riferimento ad un tipo di CD-ROM, chiamati Yellowbook, per spiegare le singole modalità:

Le specifiche dello Yellow-Book

Le specifiche dello **Yellow-Book** sono strettamente legate a quelle del **Red-Book**. Vengono conservati la dimensione di **2352** byte utili per settore, il **Lead-in**, la **Program area**, il **Lead-out** e il primo livello di correzione dell'errore . Già dal livello successivo, lo **Yellow-Book** può aggiungere un ulteriore livello per il rilevamento e la correzione degli errori. I settori **Yellow-Book** hanno **12** byte di sincronismo e **4** byte di **Header**. Per usare lo spazio rimanente, sono inoltre definiti due modi, detti **mode 1** e il **mode 2**; la scelta tra i due è codificata nel campo Header. Il **mode 1** caratterizza il **CD-ROM** con codice a correzione di errore, le informazioni aggiuntive per la correzione degli errori, **288** byte per settore, riducono la capacità per i dati utenti a **2.048** byte (che essendo potenza di due è un buon numero per gli informatici), il mode **2** contiene più dati ed è usato per le informazioni musicali e grafiche (più tolleranti agli errori.) nei **CD** multimediali. In entrambi i casi l'indirizzo di un settore è scritto nel settore stesso, questo accorgimento risolve completamente il problema di un accesso casuale ai dati in quanto, una volta letti anche in ordine sparso, questi possono essere riordinati.

Accesso casuale su CD-ROM

- Difficile a causa della velocità lineare costante
- Spostare la testina in posizione approssimata
- Configurare la giusta velocità di rotazione
- Leggere l'indirizzo
- Altri aggiustamenti per spostarsi sul settore richiesto

Pro e contro CD-ROM

- Capacità (?, ormai non più…)
- Facili da produrre su grande scala
- Rimovibile
- Robusto
- Costoso per piccole quantità
- Lento
- Solo lettura

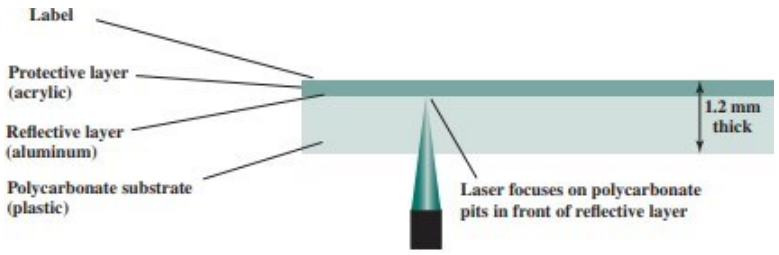
Esistono altri formati di CD:

- Un Compact Disc Recordable (CD-R) è un disco di tipo Write Once Read Multiple (WORM). Questi dischi possono registrare solo una volta i dati, che diventano permanenti sul disco. Non è possibile registrare altri dati. Può quindi essere letto come un CD-ROM standard. Dopo la scrittura su un CD-R, il disco diventa un CD-ROM.
- Un Compact disc Re-Writable (CD-RW) è un disco cancellabile che può essere riutilizzato. I dati su un disco CD-RW possono essere cancellati e registrati in più riprese.

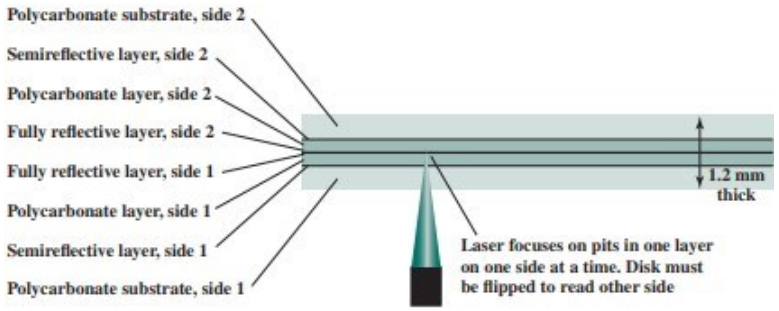
Similmente, abbiamo anche i DVD (Digital Video Disk, per riprodurre film) o Digital Versatile Disk (lettore dati e video per computer).

I DVD hanno lo stesso diametro e spessore dei CD e sono realizzati con alcuni degli stessi materiali e metodi di produzione. Come i CD, i dati su un DVD sono codificati sotto forma di piccoli fori e protuberanze nella traccia del disco.
Un DVD è composto da diversi strati di plastica, per un totale di circa 1,2 millimetri di spessore. Ogni strato è creato mediante stampaggio a iniezione di policarbonato. Questo processo forma un disco con microscopiche protuberanze disposte come un'unica, continua e lunghissima traccia a spirale di dati. Per saperne di più sulle protuberanze, si veda più avanti.

Una volta formati i pezzi trasparenti di policarbonato, un sottile strato riflettente viene spruzzato sul disco, coprendo le protuberanze. Per gli strati interni si usa l'alluminio, mentre per gli strati esterni si usa uno strato d'oro semi-riflettente, che permette al laser di focalizzarsi attraverso gli strati esterni e su quelli interni. Dopo aver realizzato tutti gli strati, ciascuno di essi viene rivestito di lacca, compresso e polimerizzato sotto la luce infrarossa. Per i dischi monofacciali, l'etichetta viene serigrafata sul lato non leggibile. I dischi a doppia faccia sono stampati solo sull'area non leggibile vicino al foro centrale.
Ogni strato scrivibile di un DVD ha una traccia di dati a spirale. Nei DVD a singolo strato, la traccia gira sempre dall'interno del disco verso l'esterno. Il fatto che la traccia a spirale inizi al centro significa che un DVD a singolo strato può essere più piccolo di 12 centimetri, se lo si desidera.

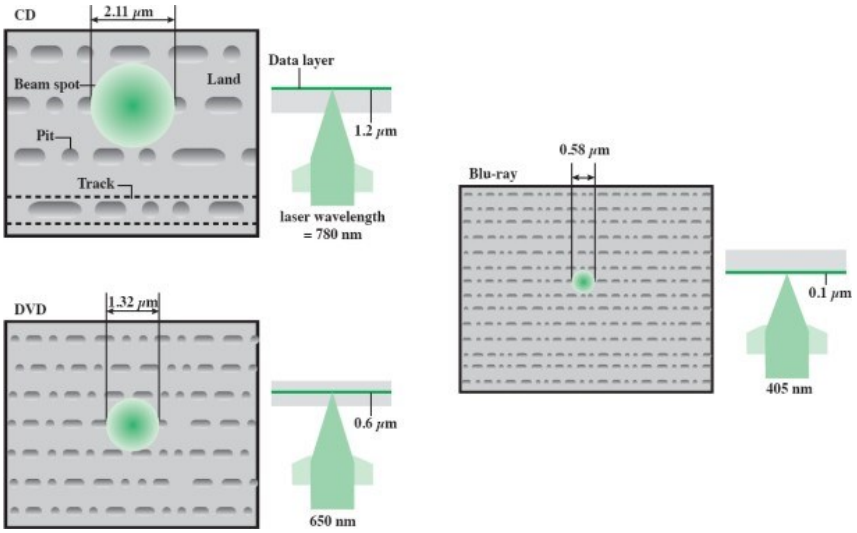


(a) CD-ROM–Capacity 682 MB



(b) DVD-ROM, double-sided, dual-layer–Capacity 17 GB

I vari strati riflettenti, a seconda delle loro proprietà, riescono a trasmettere meglio la luce incrementando e migliorando la memorizzazione.



Quello che l'immagine a sinistra non riesce a far capire è quanto sia incredibilmente piccola la traccia dei dati: solo 740 nanometri separano una traccia dall'altra (un nanometro è un miliardesimo di metro). Le protuberanze allungate che compongono la traccia sono larghe 320 nanometri ciascuna, lunghe almeno 400 nanometri e alte 120 nanometri. La figura seguente illustra come osservare le protuberanze attraverso lo strato di policarbonato.
Spesso si legge di "buche" su un DVD anziché di protuberanze. Sul lato dell'alluminio appaiono come pozzetti, ma sul lato da cui il laser legge sono protuberanze.

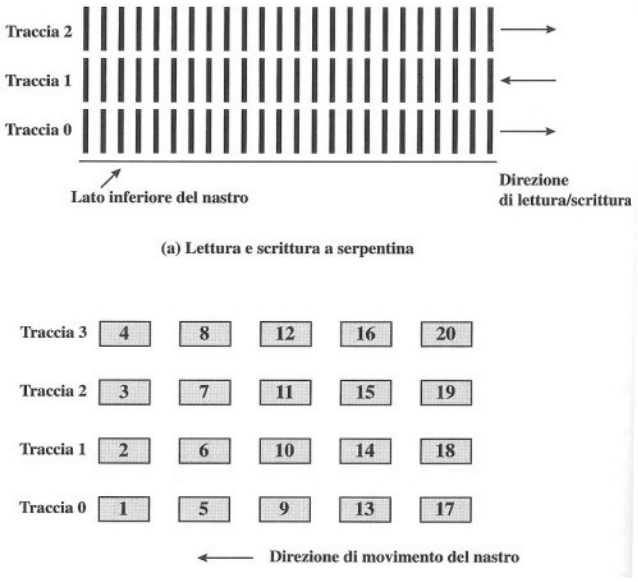
Le dimensioni microscopiche delle protuberanze rendono la traccia a spirale di un DVD estremamente lunga. Se si potesse sollevare la traccia dati da un singolo strato di un DVD e allungarla in linea retta, sarebbe lunga quasi 7,5 miglia! Ciò significa che un DVD a doppio lato e doppio strato conterrebbe 30 miglia (48 km) di dati!

Infine, il nastro magnetico.
Esso è un nastro di plastica con rivestimento magnetico. È un supporto di memorizzazione su una grande bobina aperta o in una cartuccia o cassetta più piccola (come una cassetta musicale). I nastri magnetici sono supporti di memorizzazione più economici. Sono durevoli, possono essere scritti, cancellati e riscritti. Data la loro grande lentezza, sono usati principalmente per backup di dati.
Sebbene sia ottimo per un uso a breve termine, il nastro magnetico è altamente incline alla disintegrazione. A seconda dell'ambiente, questo processo può iniziare dopo 10-20 anni.

Con il passare del tempo, i nastri magnetici prodotti negli anni '70 e '80 possono soffrire di un tipo di deterioramento chiamato "sindrome della macchia appiccicosa". È causata dall'idrolisi del legante del nastro e può rendere il nastro inutilizzabile.

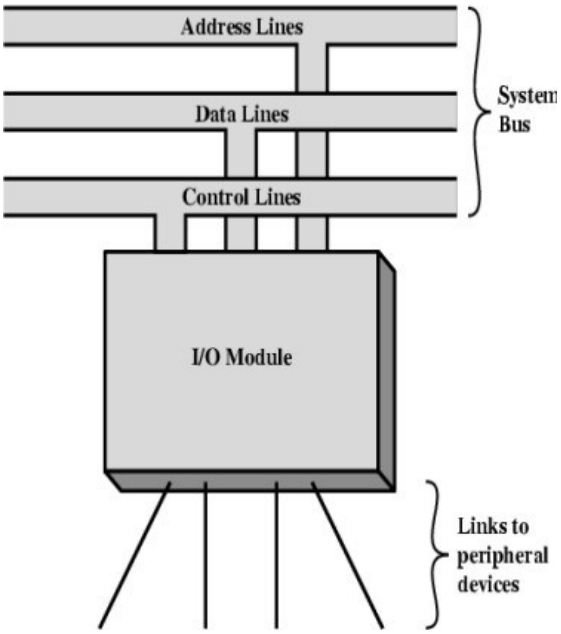
Come funzionano

I nastri magnetici convertono i segnali audio elettrici in energia magnetica, costituendo così la base del loro principio di funzionamento. La conversione imprime tali segnali sul nastro; quando il nastro si sposta sulle testine magnetiche, le particelle magnetiche fanno sì che gli impulsi si allineino in schemi, provocando la produzione del suono.
Il nastro viene spostato a una velocità costante grazie a una macchina a nastro. Se la velocità cambia per qualsiasi motivo, che si tratti di motori difettosi o di impostazioni non corrette, i dati potrebbero corrompersi. In generale, ha una direzione di scorrimento a serpentina, tale da salvare i dati simultaneamente su più tracce.



Scritto da Gabriel

Gestione I/O



Esistono molte periferiche da gestire con quantità di dati differenti e a diverse velocità e formati. Normalmente i vari dispositivi sono più lenti di CPU e RAM e si ha necessità di avere moduli di I/O.

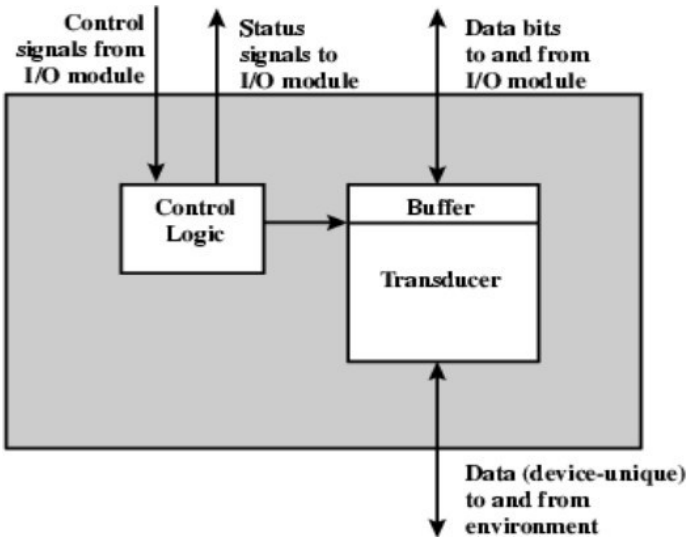
I moduli di ingresso/uscita (moduli I/O) fungono da mediatori tra il processore e i dispositivi di ingresso/uscita. I moduli di ingresso ricevono i segnali dagli interruttori o dai sensori e li inviano al processore, mentre i moduli di uscita riportano i segnali del processore ai dispositivi di controllo, come i relè o gli avviatori di motori.
Un modulo di I/O svolge un ruolo fondamentale nel colmare il divario tra un sistema informatico e i dispositivi esterni. Può svolgere diverse funzioni, tra cui:

- Comunicazione con i dispositivi: Un modulo I/O può svolgere diverse funzioni di comunicazione con i dispositivi, come la segnalazione dello stato, i comandi e il trasferimento di informazioni.
- Interfaccia diretta con CPU/Memoria e periferiche
- Controllo e temporizzazione: Per gestire correttamente il flusso di informazioni tra un sistema informatico e un dispositivo esterno, un modulo I/O utilizza le risorse interne di un computer per eseguire la gestione del tempo.
- Comunicazione con il processore: Per completare la comunicazione con il processore è necessario che il modulo I/O svolga dei compiti, tra cui la decodifica e l'accettazione dei comandi, la segnalazione degli aggiornamenti di stato e il riconoscimento del proprio indirizzo.
- Rilevamento degli errori: Un modulo I/O può rilevare una serie di problemi diversi tra il suo sistema e un dispositivo esterno. Ciò include errori meccanici, come l'inceppamento della carta in una stampante, e problemi basati sui dati durante la trasmissione.
- Buffering dei dati: Una delle funzionalità più importanti di un modulo I/O è quella di gestire la velocità di trasferimento tra la memoria, il processore e le altre periferiche collegate.
- I moduli I/O migliorano anche le capacità di più sistemi. Sono una parte necessaria di qualsiasi sistema informatico che coinvolga un dispositivo esterno. Un modulo di ingresso può portare a un lavoro più efficiente e preciso grazie alle sue capacità multifunzionali.

I vantaggi dei moduli I/O
Esistono diversi tipi di moduli I/O e tutti svolgono un ruolo necessario in qualsiasi sistema operativo efficace. I moduli I/O possono essere utili in un controllore logico programmabile (PLC) e in altri sistemi operativi perché possono:

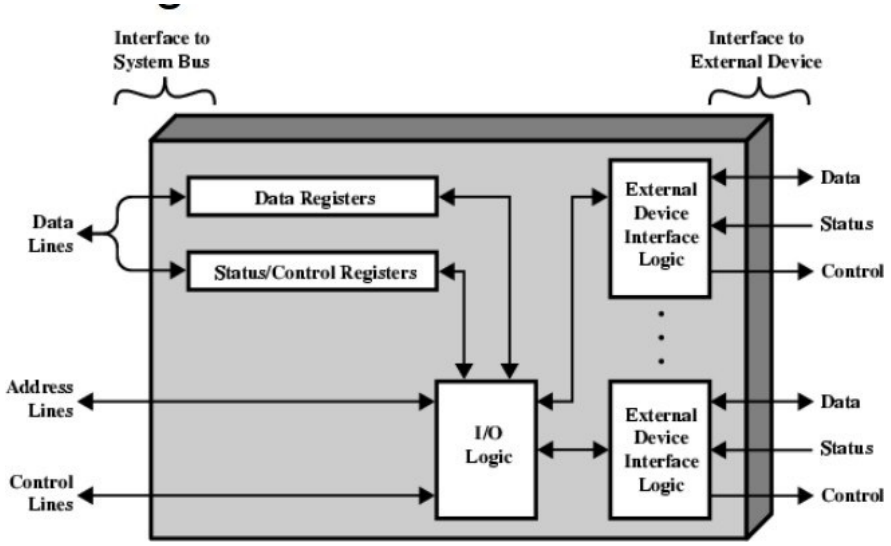
- Ridurre le spese per l'hardware.
- Risparmiare spazio nell'armadio di controllo.
- Possono essere installati in remoto.
- Semplificare il cablaggio e la configurazione dei cavi.
- Risparmio evidente sull'hardware.
- Aumentare l'organizzazione

Similmente vi sono i vari dispositivi esterni, ciascuno comprensibile all'uomo (video, stampante, tastiera), comprensibili dalla macchina (monitoraggio e controllo) di comunicazione (modem/rete (NIC)).
Tutto ciò che può essere collegato a un computer dall'esterno può essere considerato un dispositivo esterno. Qualsiasi dispositivo periferico non alloggiato all'interno dell'armadio del computer. Monitor, tastiere, mouse e stampanti sono intrinsecamente dispositivi esterni; tuttavia, anche i drive, gli adattatori di rete e i modem possono essere esterni.



Come si vede, hanno dei bit dati che vengono elaborati da un'apposita logica di controllo da moduli e buffer elaboratori, andando poi a tradurre per ogni singolo dispositivo ed elaborando (buffering) i dati, rilevando possibili errori e comunicando con la CPU e i singoli dispositivi.
In generale, i passi di I/O sono così descritti:

- CPU interroga il modulo I/O sullo stato del dispositivo connesso
- Il modulo I/O restituisce lo stato del dispositivo
- Se dispositivo pronto a trasmettere, CPU richiede il trasferimento dei dati, tramite comando a modulo I/O
- Il modulo I/O ottiene una unità di dati dal dispositivo esterno
- Il modulo I/O trasferisce i dati alla CPU



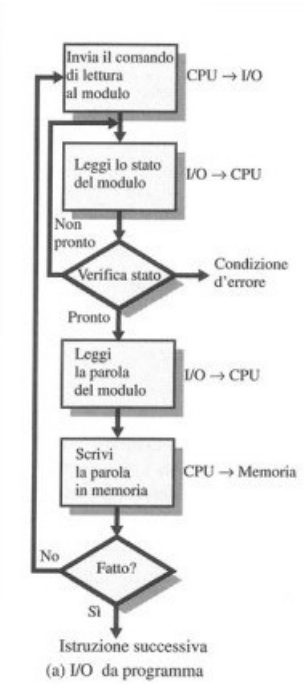
Essi permettono di nascondere/rivelare le proprietà di un dispositivo alla CPU, supportando dispositivi singoli/multipli, controllando funzioni del dispositivo (o lasciandolo alla CPU) e, naturalmente, adattarsi allo specifico sistema operativo.

Esistono 3 tecniche di gestione dell'I/O:

1) I/O da programma (I/O programmed)
   L'I/O programmabile è una tecnica di I/O diversa dall'I/O a interrupt e dall'accesso diretto alla memoria (DMA). L'I/O programmato è il tipo più semplice di tecnica di I/O per lo scambio di dati o qualsiasi tipo di comunicazione tra il processore e i dispositivi esterni. Con l'I/O programmato, i dati vengono scambiati tra il processore e il modulo di I/O. Il processore esegue un programma che fornisce un'informazione sul funzionamento del modulo. Il processore esegue un programma che gli dà il controllo diretto delle operazioni di I/O, compreso il rilevamento dello stato del dispositivo, l'invio di un comando di lettura o scrittura e il trasferimento dei dati. Quando il processore invia un

comando al modulo I/O, deve attendere che l'operazione di I/O sia completata. Se il processore è più veloce del modulo di I/O, ciò comporta uno spreco di tempo per il processore. Il funzionamento complessivo dell'I/O programmato può essere riassunto come segue:

- Il processore sta eseguendo un programma e incontra un'istruzione relativa a un'operazione di I/O. Il processore esegue l'istruzione.
- Il processore esegue l'istruzione emettendo un comando al modulo di I/O appropriato.
- Il modulo di I/O esegue l'azione richiesta in base al comando di I/O impartito dal processore (READ/WRITE) e imposta i bit appropriati nel registro di stato dell'I/O. Il processore controlla periodicamente lo stato del modulo.
- Il processore controlla periodicamente lo stato del modulo di I/O fino a quando non constata che l'operazione è stata completata (sprecando tempo di CPU, dato che la stessa deve aspettare la fine dell'operazione da parte dei moduli di I/O).



(a) I/O da programma

Nel dettaglio:

- CPU richiede operazione I/O
- Modulo I/O esegue operazione
- Modulo I/O setta bit di stato
- CPU controlla bit di stato periodicamente
- Modulo I/O non informa direttamente CPU
- Modulo I/O non interrompe CPU
- CPU può attendere o fare altro e controllare più tardi

La CPU, a sua volta, a ha una serie di comandi:

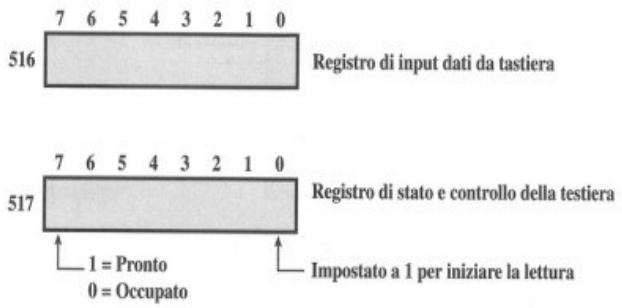
- CPU invia indirizzo
    a. che identifica modulo (& dispositivo se >1 per modulo)
- CPU invia commando
- di controllo – dire al modulo cosa fare
    a. ad esempio, dare velocità al disco
- di test – controlla lo stato
    a. ad esempio, alimentazione? errore?
- di lettura/scrittura
    a. il modulo trasferisce i dati tramite buffer dal/verso il dispositivo

Nell'I/O da programma il trasferimento dati è molto simile all'accesso alla memoria (dal punto di vista della CPU). Ad ogni dispositivo viene assegnato un identificatore unico ed i comandi di CPU riferiscono tale identificatore (indirizzo).
Il mapping I/O è fatto in due modi:

1) I/O memory-mapped, esiste un unico spazio di indirizzi per le locazioni di memoria e i dispositivi di I/O e il processore tratta i registri di stato e di dati dei moduli di I/O come locazioni di memoria e utilizza le stesse istruzioni macchina per accedere sia alla memoria che ai dispositivi di I/O. Quindi, ad esempio, con 10 linee di indirizzo, è possibile supportare un totale di = 1024 locazioni di memoria e indirizzi di I/O, in qualsiasi combinazione. Con l'I/O mappato in memoria, sul bus sono necessarie una singola linea di lettura e una singola linea di scrittura.
2) I/O isolated, Con l'I/O isolato, il bus può essere dotato di linee di comando di lettura e scrittura della memoria e di linee di comando di ingresso e uscita. Ora, la linea di comando specifica se l'indirizzo si riferisce a una posizione di memoria o a un dispositivo di I/O. L'intera gamma di indirizzi può essere disponibile per entrambi. L'intera gamma di indirizzi può essere disponibile per entrambi. Ancora una volta, con 10 linee di indirizzo, il sistema può ora supportare sia 1024 locazioni di memoria che 1024 indirizzi di I/O.

Scritto da Gabriel

Confronto tra I/O memory mapped e separato



INDIRIZZO	ISTRUZIONE	OPERANDO	COMMENTO
200	Load AC	"1"	Carica nell'accumulatore
	Store AC	517	Inizia la lettura della tastiera
202	Load AC	517	Legge il byte di stato
	Branch if Sign = 0	202	Sta in loop fino a quando è pronto
	Load AC	516	Carica un byte di dati

(a) I/O memory mapped

INDIRIZZO	ISTRUZIONE	OPERANDO	COMMENTO
200	Load I/O	5	Inizia la lettura della tastiera
201	Test I/O	5	Controlla il completamento
	Branch Not Ready	201	Sta in loop fino al completamento
	In	5	Carica un byte di dati

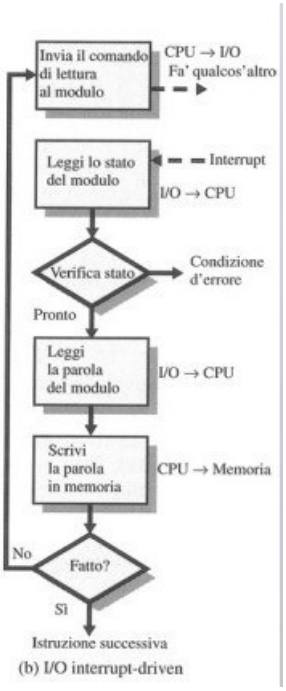
(b) I/O isolato

La tecnica memory-mapped I/O ha diversi vantaggi
- Non necessita di istruzioni speciali
  - Le istruzioni che accedono alla memoria "normale" accedono anche alle aree di I/O
  - Il software di controllo di dispositivo può essere scritto interamente in linguaggi ad alto livello
- Consente una più agevole protezione
- è sufficiente nascondere le aree di I/O allo spazio di indirizzamento dell'utente (privilegi)
- Con la tecnica della memoria segmentata, più aree di I/O possono mappare sul medesimo spazio di indirizzamento fisico

Tuttavia, anche alcuni svantaggi:
- Non si presta all'uso di cache
- Il dato rilevante è sempre e solo nella memoria del dispositivo
- Occorre disabilitare selettivamente la cache
- Non è compatibile con architetture a bus multipli
- I dispositivi di I/O non possono rispondere ad indirizzi emessi su bus non connessi
- Occorre filtrare gli indirizzi emessi dalla CPU ed instradarli sul bus appropriato
  - Filtraggio a sorgente piuttosto che a destinazione

Un interrupt I/O è un processo di trasferimento di dati in cui un dispositivo esterno o una periferica informa la CPU di essere pronta per la comunicazione e richiede l'attenzione della CPU.

L'I/O guidato da interrupt è un approccio per trasferire dati tra la 'memoria' e i 'dispositivi di I/O' attraverso il 'processore'. Le altre due tecniche sono l'I/O programmato e l'accesso diretto alla memoria (DMA). L'I/O guidato da interrupt comporta l'uso di interrupt per lo scambio di dati tra I/O e memoria.
Per migliorare le prestazioni del sistema si può utilizzare un approccio alternativo in cui, dopo aver impartito il comando di I/O al modulo di I/O, il processore può essere impegnato in altre operazioni. In questo modo, il tempo prezioso del processore può essere utilizzato.



(b) I/O interrupt-driven

Funzionamento dell'I/O guidato da interrupt
In questa sezione studieremo come con l'approccio dell'I/O guidato da interrupt i dati vengono scambiati tra la memoria e l'I/O attraverso il processore. Vedremo l'intero scenario prima dal punto di vista del modulo I/O e poi dal punto di vista del processore.
Consideriamo che i dati devono essere memorizzati nella memoria principale dal modulo di I/O come input dal punto di vista del modulo di I/O.

1) A questo scopo, il processore invia un comando READ I/O al modulo I/O corrispondente e procede con altre operazioni utili. Non attende che il modulo I/O sia pronto con i dati desiderati.

2) Il modulo I/O elabora il comando READ I/O e legge i dati dalla periferica indirizzata. Il modulo I/O memorizza i dati letti nel suo registro dati ed emette un segnale di interruzione al processore

Scritto da Gabriel

attraverso la linea di controllo del bus di sistema. Inviando il segnale di interruzione, il modulo I/O indica al processore che ora è pronto a trasmettere i dati. Tuttavia, il modulo I/O deve aspettare che il processore richieda i dati al modulo I/O.

3) Quando il processore richiede i dati al modulo di I/O, li trasferisce sulla linea dati del bus di sistema. Una volta che il modulo di I/O trasferisce i dati al processore, si predispone per un altro trasferimento di I/O.

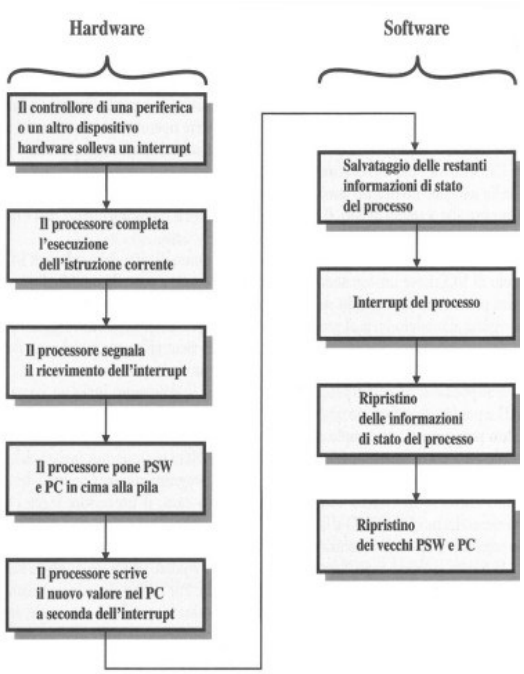
Discutiamo ora questo trasferimento di dati tra processore e I/O dal punto di vista del processore.

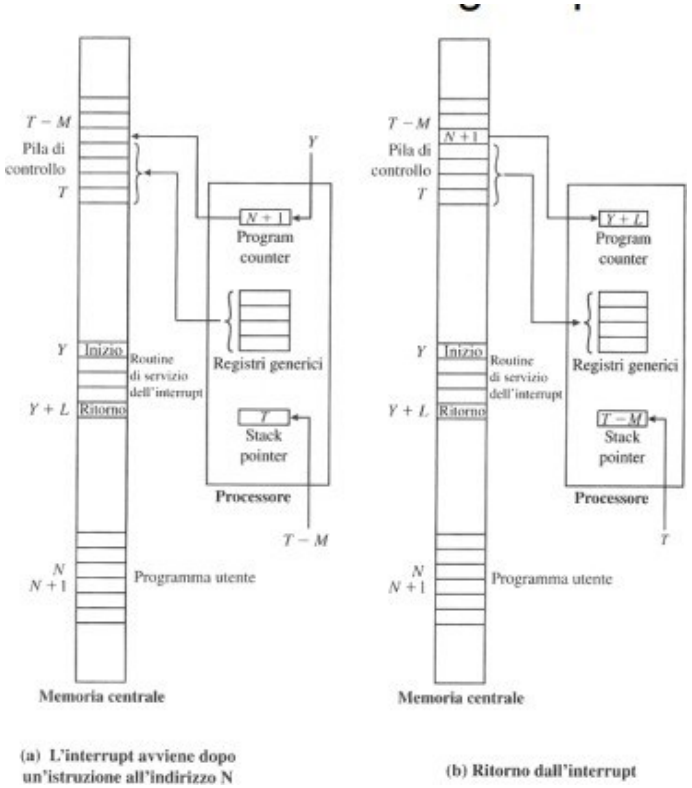
1) Per recuperare i dati dal modulo di I/O, il processore emette un comando READ e procede a fare qualcos'altro. Ad esempio, inizia l'esecuzione di un altro programma, perché potrebbe lavorare su più programmi alla volta.

2) Come sappiamo, ogni volta che il processore esegue un programma, dopo ogni ciclo di istruzioni controlla se si sono verificati degli interrupt. Se trova un'interruzione in corso, risponde e serve l'interruzione verificatasi.

3) Nel momento in cui il processore trova un'interruzione da parte del modulo di I/O, sospende l'esecuzione in corso e salva il contesto (ad esempio, il contatore del programma, il registro del processore) per servire l'interruzione.

4) A questo punto il processore richiede i dati al modulo di I/O e accetta la parola di dati sulla linea dati. Il processore salva questi dati nella memoria e ripristina il contesto del programma su cui stava lavorando e riprende l'esecuzione.





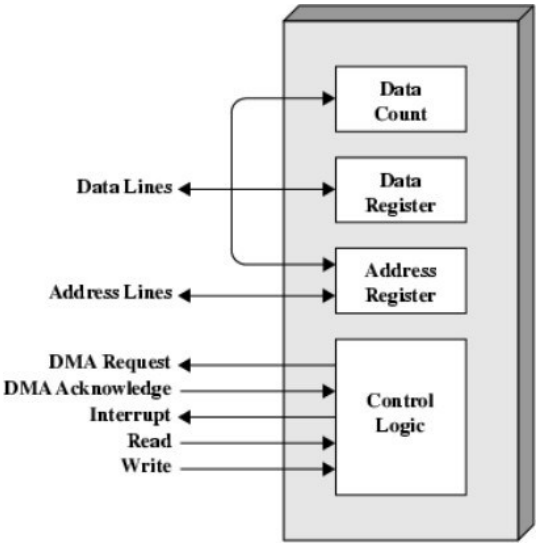
Le operazioni base, come da slide:
- CPU rilascia comando di lettura
- Modulo I/O ottiene i dati dalla periferica mentre la CPU svolge altro lavoro
- Modulo I/O interrompe la CPU
- CPU richiede i dati al modulo I/O
- Modulo I/O trasferisce i dati alla CPU

La CPU, infatti, rilascia un comando di lettura e nel frattempo esegue altro lavoro; controlla se c'è interruzione alla fine di ogni ciclo di istruzione (ciclo fetch/execute con trattamento delle interruzioni). Se interruzione presente:
- Salva contesto (PC e registri)
- Interruzione del processo corrente ed elaborazione interrupt
- Lettura dati da modulo I/O e scrittura in memoria

Esiste inoltre il DMA (Direct Memory Access), che è una funzionalità fornita da alcune architetture di bus per computer che consente di inviare dati direttamente da un dispositivo collegato (ad esempio un'unità disco) alla memoria sulla scheda madre del computer. Il microprocessore non è coinvolto nel trasferimento dei dati, accelerando così il funzionamento complessivo del computer.
Gli strumenti delle risorse di sistema di un computer sono utilizzati per la comunicazione tra hardware e software. I quattro tipi di risorse di sistema sono:

- Indirizzi di I/O.
- Indirizzi di memoria.
- Numeri di richiesta di interruzione (IRQ).
- Canali di accesso diretto alla memoria (DMA).

I canali DMA sono utilizzati per comunicare i dati tra la periferica e la memoria di sistema. Tutte e quattro le risorse di sistema si basano su alcune linee del bus. Alcune linee del bus sono utilizzate per gli IRQ, altre per gli indirizzi (gli indirizzi di I/O e l'indirizzo della memoria) e altre ancora per i canali DMA.
Un canale DMA consente a un dispositivo di trasferire dati senza esporre la CPU a un sovraccarico di lavoro. Senza i canali DMA, la CPU copia ogni dato utilizzando un bus periferico dal dispositivo di I/O. L'utilizzo di un bus periferico occupa la CPU. L'utilizzo di un bus periferico occupa la CPU durante il processo di lettura/scrittura e non consente di eseguire altre operazioni fino al completamento dell'operazione.



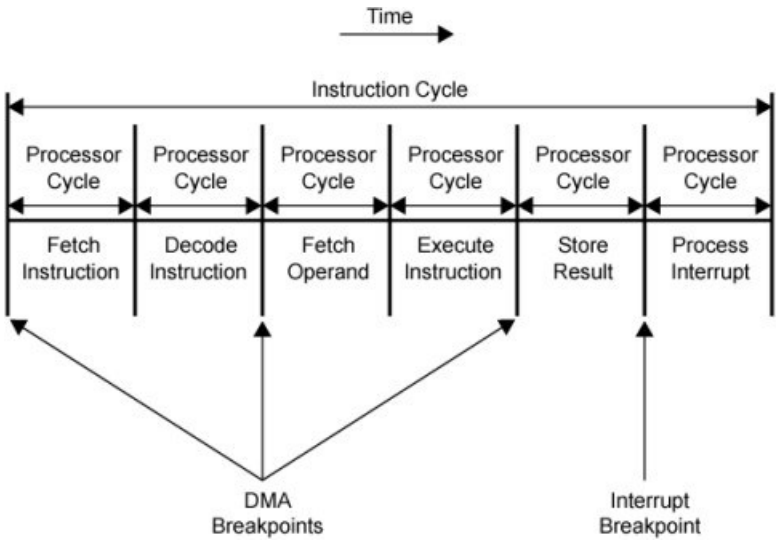
Con il DMA, la CPU può eseguire altre operazioni mentre viene eseguito il trasferimento dei dati. Il trasferimento dei dati viene innanzitutto avviato dalla CPU. Il blocco di dati può essere trasferito da e verso la memoria dal DMAC in tre modi.
In modalità burst, il bus di sistema viene rilasciato solo al termine del trasferimento dei dati. In modalità cycle stealing, durante il trasferimento dei dati tra il canale DMA e il dispositivo di I/O, il bus di sistema viene ceduto per alcuni cicli di clock in modo che la CPU possa eseguire altre operazioni. Al termine del trasferimento dei dati, la CPU riceve una richiesta di interrupt dal controller DMA. In modalità trasparente, il DMAC può occupare il bus di sistema solo quando non è richiesto dal processore.

Tuttavia, l'uso di un controller DMA può causare problemi di coerenza della cache. I dati memorizzati nella RAM a cui accede il controller DMA potrebbero non essere aggiornati con i dati corretti della cache se la CPU utilizza una memoria esterna.
Le soluzioni comprendono il lavaggio delle linee della cache prima di avviare i trasferimenti DMA in uscita, oppure l'esecuzione di un'invalidazione della cache sui trasferimenti DMA in entrata quando le scritture esterne vengono segnalate al controller della cache.

Il controller DMA può accedere al canale:

- Una parola alla volta, sottraendo di tanto in tanto alla CPU il controllo sul canale (cycle stealing). All'interno di questo:
    - Il controllore DMA prende possesso del bus per un ciclo
    - Trasferisce una parola (word) di dati
    - Non è una interruzione (la CPU non cambia contesto)
    - La CPU rimane "sospesa" proprio nel momento prima che acceda al bus
    - Ad esempio, prima del caricamento di un dato e/o operando o di una scrittura
    - Rallenta la CPU ma non così tanto come nel caso in cui sia la CPU stessa ad occuparsi del trasferimento dati

- Per blocchi, prendendo possesso del canale per una serie di trasferimenti (burst mode)
- La CPU è bloccata in entrambi i casi, ma il burst mode è più efficace perché l'acquisizione del canale è onerosa

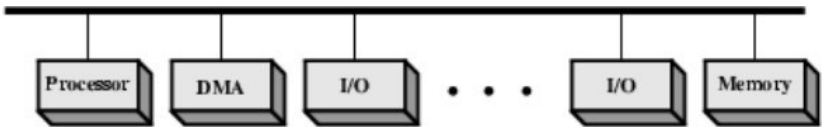


L'interfaccia DMA trasferisce un blocco completo di dati, una parola alla volta, direttamente alla o dalla memoria senza passare per il processore. Al termine del trasferimento, l'interfaccia DMA trasmette un segnale di interrupt al processore. Quindi, in DMA il coinvolgimento del processore può essere limitato all'inizio e alla fine del trasferimento, come mostrato nella figura precedente. Tuttavia, ci si chiede quando il DMA debba prendere il controllo del bus.

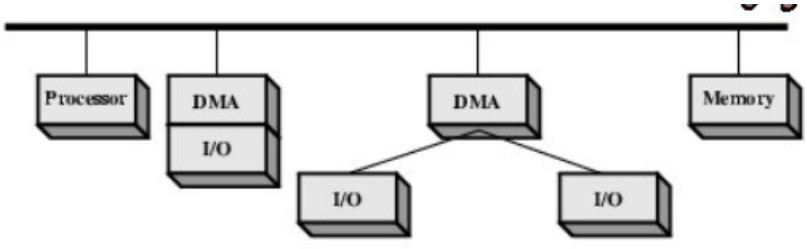
A questo proposito, ricordiamo il fenomeno dell'esecuzione di un'istruzione da parte del processore. La figura seguente mostra i cinque cicli di esecuzione di un'istruzione. La figura mostra anche cinque punti in cui è possibile rispondere a una richiesta DMA e un punto in cui è possibile rispondere a una richiesta di interrupt. Si noti che una richiesta di interrupt viene riconosciuta solo in un punto di un ciclo di istruzioni, ossia nel ciclo di interrupt.

Le configurazioni del DMA possono essere:

- A bus singolo, controller DMA isolato. Ogni trasferimento usa il bus due volte da I/O a DMA e poi da DMA alla memoria. La CPU perde il possesso del bus due volte

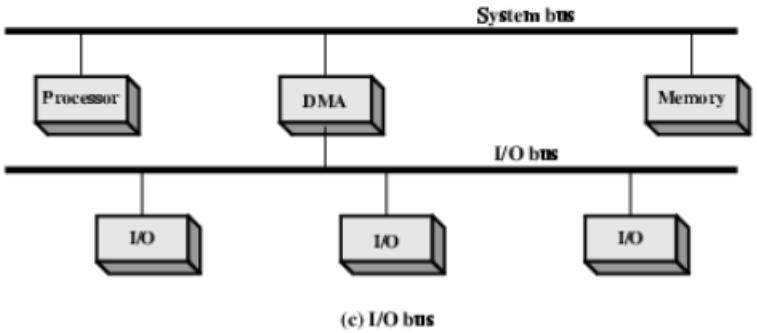


- A bus singolo, controllore DMA integrato con I/O. Può controllare più di un dispositivo ed ogni trasferimento usa il bus una volta da DMA a memoria. La CPU perde il controllo del bus una sola volta.



(b) Single-bus, Integrated DMA-I/O

- Bus di I/O separato, DMA necessita di una sola interfaccia I/O. Ogni trasferimento usa il bus di sistema una sola volta da DMA a memoria. La CPU perde il controllo del bus una sola volta



(c) I/O bus

Esistono inoltre gli I/O Channels/Canali di I/O, che sono un'estensione del concetto di DMA.
Sono in grado di eseguire le istruzioni di I/O utilizzando un processore speciale sul canale di I/O e di controllare completamente le operazioni di I/O. Il processore non esegue direttamente le istruzioni di I/O. Il processore non esegue le istruzioni di I/O da solo. Il processore avvia il trasferimento di I/O ordinando al canale di I/O di eseguire un programma in memoria.

Il programma specifica: dispositivo o dispositivi, area o aree di memoria, priorità e azioni in condizioni di errore.
Tipi di canali di I/O :

1) Canale selettore:

Il canale selettore controlla più dispositivi ad alta velocità. È dedicato al trasferimento di dati con uno dei dispositivi. Nel canale selettore, ogni dispositivo è gestito da un controllore o da un modulo di I/O. Il canale selettore controlla i controllori di I/O mostrati nella figura.

2) Canale multiplexer :

Il canale multiplexer è un controller DMA che può gestire più dispositivi contemporaneamente. Può eseguire trasferimenti a blocchi per più dispositivi contemporaneamente.

In questo canale vengono utilizzati due tipi di multiplexer:

1) Multiplexer di byte

È utilizzato per i dispositivi a bassa velocità. Trasmette o accetta caratteri. Intercalare i byte da diversi dispositivi.

2) Multiplexer a blocchi

Accetta o trasmette blocchi di caratteri. Intercetta blocchi di byte da diversi dispositivi. Utilizzato per dispositivi ad alta velocità.

Esercizi sulla prima parte

Domande a risposta multipla

es2

Si consideri un codice di correzione di Hamming su 16 bit. Dire quale sequenza di bit è memorizzata in memoria se si devono memorizzare i seguenti 16 bit 1001001010001110 di dati:

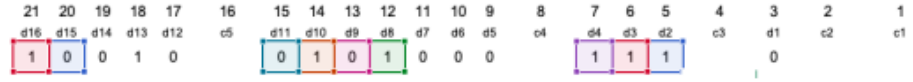
- [a] 0000110010010100011110
- [b] 1101010101000011111001
- [c] 1001000101000011110001
- [d] 1001001010001110000001
- [e] nessuna delle risposte precedenti è corretta

Spiegazione:

c1=1

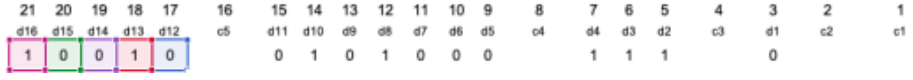
c2=0

c3=0



c4=0

c5=0



es3

Sia dato un disco rigido con le seguenti caratteristiche:

- capacità di 1TB;
- 2 piatti (4 facce);
- 65536 tracce per faccia e 8192 settori per traccia;
- velocità di rotazione di 10000 rpm;
- tempo medio di posizionamento della testina di 6,5 ms.

Il tempo totale medio per trasferire (tempo di accesso totale medio, secondo il libro) 256KB memorizzati in una stessa traccia è di circa

- [a] 9,6875
- [b] 9,875 ms
- [c] 9,59375 ms
- [d] 10,25 ms
- [e] nessuna delle risposte precedenti è corretta

Spiegazione:

Sappiamo che

$$T_S = 6{,}5 \text{ ms e } T_L = (1000/\,(10000/60)\,)\,/\,2 \approx 3{,}0 \text{ ms}$$

e che il tempo totale di trasferimento è dato da

$$T = T_S + T_L + T_t$$

dove il tempo di trasferimento (in millisecondi) è dato dalla formula

$$T_t = \frac{b}{rN} \times 1000$$

b #byte da trasferire
N #byte per traccia
r velocità rotazione (in rotazioni per sec.)

Il numero di byte per faccia sarà dato dalla capacità totale del disco diviso il numero di facce

$$1\text{TB} / 4 = 2^{40} / 2^2 = 2^{38}$$

Il numero di byte per traccia N sarà dato dalla capacità totale di una faccia diviso il numero di tracce ($65536 = 2^{16}$)

$$N = 2^{38} / 2^{16} = 2^{22}$$

Quindi

$$\begin{aligned} T_t &= [1000 \times 256\text{KB}] / [(10000/60) \times 2^{22}] \\ &= [1000 \times 2^{18}\,] / [(10000/60) \times 2^{22}] \\ &= 0{,}375 \text{ ms} \end{aligned}$$

Pertanto il tempo totale di accesso è

$$T = 6{,}5 + 3{,}0 + 0{,}375 = 9{,}875 \text{ ms}$$

Esercizio

es8

Sia data la seguente sequenza di indirizzi in lettura (l) o scrittura (s) emessi dalla CPU e che la memoria abbia il contenuto esadecimale mostrato di seguito:

#	indirizzo (binario)	l/s	byte scritto (HEX)
1	000100001000	s	D4
2	000100001111	l	
3	000100001110	s	DC
4	000100011101	s	9F
5	000100011001	l	
6	000100011111	l	
7	000100000001	s	FF
8	000100000010	l	

ind	byte	ind	byte	ind	byte	ind	byte
100	08	101	0A	102	D7	103	02
104	1F	105	00	106	80	107	E0
108	AE	109	73	10A	AF	10B	23
10C	A1	10D	42	10E	90	10F	75
110	B9	111	16	112	FD	113	D0
114	0A	115	07	116	03	117	71
118	3E	119	D3	11A	71	11B	23
11C	A1	11D	8A	11E	90	11F	15
120	F9	121	86	122	A0	123	00
124	E9	125	16	126	05	127	00

Si assuma che la dimensione di parola coincida con un byte, e la presenza di una cache di ampiezza 32B, dimensione di blocco 4B, inizialmente vuota, e ad associazione a 2 vie (politica di rimpiazzo LRU, politica di scrittura write-back e gestione dei miss in scrittura con la politica write allocate).

Si mostri come sia il contenuto della cache che il contenuto della memoria cambia.

Si assuma che la dimensione di parola coincida con un byte, e la presenza di una cache di ampiezza 32B, dimensione di blocco 4B, inizialmente vuota, e ad associazione a 2 vie (politica di rimpiazzo LRU, politica di scrittura write-back e gestione dei miss in scrittura con la politica write allocate).

Si mostri come sia il contenuto della cache che il contenuto della memoria cambia.

Soluzione (da compilare)

- Indicare di seguito in quali campi (e la loro dimensione) gli indirizzi emessi dalla CPU sono suddivisi:

  tag: 8 bit, set: 2 bit, word: 2

- Indicare di seguito in quante linee/set la cache è suddivisa:

  8 linee suddivise in 4 set

Indicare l'evoluzione della cache e della modifica della memoria nello schema sottostante:

Indirizzo	hit/ miss	Cache *(per ogni linea di cache indicare il contenuto del campo tag)*	Modifica memoria *M[ind.] = contenuto*
0001 0000 1000 1 0 8	m	SET 10 [AE 73 AF 23] linea 0 tag:00010000 | w.a. v [D4 73 AF 23]*	
0001 0000 1111 1 0 F	m	SET 11 [A1 42 90 75] linea 0 tag:00010000	

Indirizzo	hit/ miss	Cache *(per ogni linea di cache indicare il contenuto del campo tag)*	Modifica memoria *M[ind.] = contenuto*
0001 0000 1110 1 0 E	h	SET 11 [A1 42 DC 75]* linea 0 tag:00010000	
0001 0001 1101 1 1 D	m	SET 11 [A1 8A 90 15] linea 1 tag:00010001 | w.a. v [A1 9F 90 15]*	
0001 0001 1001 1 1 9	m	SET 10 [3E D3 71 23] linea 1 tag:00010001	
0001 0001 1111 1 1 F	h	SET 11 [A1 9F 90 15]* linea 1 tag:00010001	
0001 0000 0001 1 0 1	m	SET 00 [08 0A D7 02] linea 0 tag:00010000 | w.a. v [08 FF D7 02]*	
0001 0000 0010 1 0 2	h	SET 00 [08 FF D7 02]* linea 0 tag:00010000	

es8bis

Sia data la seguente sequenza di indirizzi in lettura (l) o scrittura (s) emessi dalla CPU e che la memoria abbia il contenuto esadecimale mostrato di seguito:

#	indirizzo (binario)	l/s	parola scritta (HEX)
1	000100001000	s	D4FF032A
2	000100001100	l	
3	000100011100	s	DC3E1189
4	000100010100	s	9F9E9B9C
5	000100011000	l	
6	000100010000	l	
7	000100000000	s	FF112233
8	000100100100	l	

ind	byte	ind	byte	ind	byte	ind	byte
100	08	101	0A	102	D7	103	02
104	1F	105	00	106	80	107	E0
108	AE	109	73	10A	AF	10B	23
10C	A1	10D	42	10E	90	10F	75
110	B9	111	16	112	FD	113	D0
114	0A	115	07	116	03	117	71
118	3E	119	D3	11A	71	11B	23
11C	A1	11D	8A	11E	90	11F	15
120	F9	121	86	122	A0	123	00
124	E9	125	16	126	05	127	00

Si assuma che la dimensione di parola coincida con 4 byte, e la presenza di una cache di ampiezza 32B, dimensione di blocco di 2 parole, inizialmente vuota, e ad associazione a 2 vie (politica di rimpiazzo LRU, politica di scrittura write-back e gestione dei miss in scrittura con la politica write allocate).

Si mostri come sia il contenuto della cache che il contenuto della memoria cambia.

Soluzione (da compilare)

- Indicare di seguito in quali campi (e la loro dimensione) gli indirizzi emessi dalla CPU sono suddivisi:

  tag: 8 bit, set: 1 bit, word: 3 (di cui ultimi 2 indicano la posizione del byte nella parola)

- Indicare di seguito in quante linee/set la cache è suddivisa:

  4 linee suddivise in 2 set

Indicare l'evoluzione della cache e della modifica della memoria nello schema sottostante:

Indirizzo	hit/ miss	Cache *(per ogni linea di cache indicare il contenuto del campo tag)*	Modifica memoria *M[ind.] = contenuto*
0001 0000 1000 1 0 8	m	SET 1 [AE 73 AF 23 A1 42 90 75] linea 0 tag:00010000 l w.a. v [D4 FF 03 2A A1 42 90 75]*	
0001 0000 1100 1 0 C	h	SET 1 [D4 FF 03 2A A1 42 90 75]* linea 0 tag:00010000	

Indirizzo	hit/ miss	Cache *(per ogni linea di cache indicare il contenuto del campo tag)*	Modifica memoria *M[ind.] = contenuto*
0001 0001 1100 1 1 C	m	SET 1 [3E D3 71 23 A1 8A 90 15] linea 1 tag:00010001 l w.a. v [3E D3 71 23 DC 3E 11 89]*	
0001 0001 0100 1 1 4	m	SET 0 [B9 16 FD D0 0A 07 03 71] linea 0 tag:00010001 l w.a. v [B9 16 FD D0 9F 9E 9B 9C]*	
0001 0001 1000 1 1 8	h	SET 1 [3E D3 71 23 DC 3E 11 89]* linea 1 tag:00010001	
0001 0001 0000 1 1 0	h	SET 0 [B9 16 FD D0 9F 9E 9B 9C]* linea 0 tag:00010001	
0001 0000 0000 1 0 0	m	SET 0 [08 0A D7 02 1F 00 80 E0] linea 1 tag:00010000 l w.a. v [FF 11 22 33 1F 00 80 E0]*	
0001 0010 0100 1 2 4	m	SET 0 [F9 86 A0 00 E9 16 05 00] linea 0 (LRU) tag:00010010	M[100-107] = [B9 16 FD D0 9F 9E 9B 9C]

Domande a risposta multipla

es2

Si consideri un codice di correzione di Hamming su 16 bit. Dire quale sequenza di bit è memorizzata in memoria se si devono memorizzare i seguenti 16 bit 0100111000101011 di dati:

[a] 010011100010101111110
[b] 111110110100011010010
[c] 110101000111001001111
[d] 010010110001011011111
[e] nessuna delle risposte precedenti è corretta

Soluzione:
Di seguito si riporta la collocazione dei bit dati all'interno della tabella che permette di calcolare i bit di controllo per il codice di correzione di Hamming su 16 bit.

21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1
d16	d15	d14	d13	d12	**c5**	d11	d10	d9	d8	d7	d6	d5	**c4**	d4	d3	d2	**c3**	d1	**c2**	**c1**
0	1	0	0	1		1	1	0	0	0	1	0		1	0	1	1	1	1	1

Si ricorda che i bit di controllo necessari per $M = 16$ bit dati sono $K = 5$, in accordo con il valore minimo di K nell'equazione $2^K - 1 \geq M + K$. Inoltre, i bit di controllo sono collocati nelle posizioni che corrispondono a potenze di 2.

Di seguito si riporta il valore di ogni bit di controllo calcolato nel seguente modo:

- si rappresenta in binario (su 5 bit) la posizione di ogni bit dati;
- per il bit di controllo corrispondete alla posizione 2^i ($i \in \{0, 1, 2, 3, 4\}$) si selezionano i bit dati che hanno il bit i della rappresentazione binaria della posizione uguale a 1;
- il valore del bit di controllo è calcolato come il risultato dello XOR dei bit dati per lui selezionati.

21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1
d16	d15	d14	d13	d12	c5	d11	d10	d9	d8	d7	d6	d5	c4	d4	d3	d2	c3	d1	c2	c1
0	1	0	0	1	0	1	1	0	0	0	1	0	1	1	0	1	1	1	1	1

Pertanto la risposta corretta è d).

es3

Sia dato un disco rigido con le seguenti caratteristiche:

- capacità di 128GB;
- 1 piatto (2 facce);
- 16384 tracce per faccia e 8192 settori per traccia;
- velocità di rotazione di 10000 rpm;
- tempo medio di posizionamento della testina di 9,5 ms.

Il tempo totale medio per accedere a 128KB memorizzati in settori contigui su uno stesso cilindro è di circa

[a] 12,59375 ms

[b] 15,6875 ms

[c] 12,5875 ms

[d] 12,6875 ms

[e] nessuna delle risposte precedenti è corretta

Soluzione:
Sappiamo che

$$T_S = 9,5\ ms \text{ e } T_L = (1000/(10000/60))/2 = 3,0\ ms$$

e che il tempo totale di trasferimento è dato da

$$T = T_S + T_L + T_t,$$

con tempo di trasferimento (in millisecondi) dato dalla formula

$$T_t = \frac{b}{rN} \times 1000,$$

dove b è il numero di byte da trasferire, r è la velocità di rotazione al secondo del disco, N è il numero di byte contenuto in una traccia.

Il numero di byte per faccia sar dato dalla capacit totale del disco diviso il numero di facce

$$128GB/2 = 2^{37}/2^1 = 2^{36}.$$

Il numero di byte per traccia N sarà dato dalla capacità totale di una faccia diviso il numero di tracce ($16384 = 2^{14}$)

$$N = 2^{36}/2^{14} = 2^{22}.$$

Quindi

$$\begin{aligned} T_t &= [1000 \times 128KB]/[(10000/60) \times 2^{22}] \\ &= [1000 \times 2^{17}]/[(10000/60) \times 2^{22}] \\ &= 0,1875\ ms. \end{aligned}$$

Essendo i settori memorizzati in un cilindro, si possono leggere simultaneamente i settori posti su tracce collocate nella medesima posizione di facce diverse. Pertanto il tempo di trasferimento dei $128KB$ deve essere diviso per 2 (numero facce):

$$T = 9,5 + 3,0 + 0,1875/2 \approx 12,59375\ ms.$$

Pertanto la risposta corretta è a).

es8

Sia data la seguente sequenza di indirizzi in lettura (l) o scrittura (s) emessi dalla CPU e che la memoria abbia il contenuto esadecimale mostrato di seguito:

#	indirizzo (binario)	l/s	byte scritto (HEX)
1	000100001001	l	
2	000100001101	s	AB
3	000100001110	s	39
4	000100011100	l	
5	000100001000	s	D4
6	000100011110	l	
7	000100001010	s	98
8	000100100001	l	

ind	byte	ind	byte	ind	byte	ind	byte
100	08	101	00	102	07	103	02
104	00	105	00	106	00	107	00
108	AE	109	59	10A	AD	10B	23
10C	A1	10D	42	10E	90	10F	75
110	B9	111	16	112	00	113	00
114	0A	115	07	116	03	117	71
118	3E	119	13	11A	71	11B	23
11C	A1	11D	82	11E	90	11F	15
120	FF	121	C6	122	AD	123	00
124	E9	125	16	126	05	127	00

Si assuma che la dimensione di parola coincida con un byte, e la presenza di una cache di ampiezza 32B, dimensione di blocco 2B, inizialmente vuota, e ad associazione a 2 vie (politica di rimpiazzo LRU, politica di scrittura write-back e gestione dei miss in scrittura con la politica write allocate).

Si mostri come sia il contenuto della cache che il contenuto della memoria cambia.

Soluzione (da compilare)

- Indicare di seguito in quali campi (e la loro dimensione) gli indirizzi emessi dalla CPU sono suddivisi:

  tag da 8 bit, set da 3 bit, word da 1 bit

- Indicare di seguito in quante linee/set la cache è suddivisa:

  La cache è suddivisa in 4 set, ognuno di 2 linee da 2B

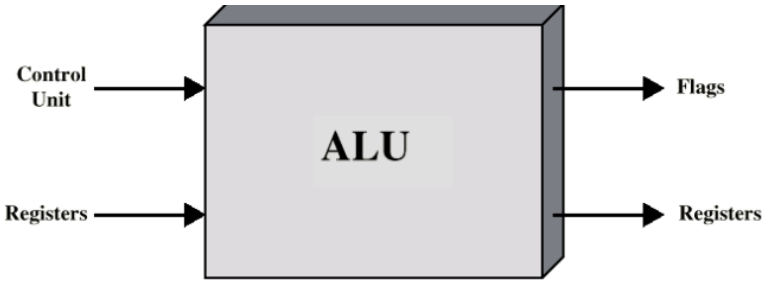
Indicare l'evoluzione della cache e della modifica della memoria nello schema sottostante:

Indirizzo	hit/ miss	Cache *(per ogni linea di cache indicare il contenuto del campo tag)*	Modifica memoria *M[ind.] = contenuto*
109hex 000100001001	miss	set 100 linea 0 [AE I 59] tag: 10hex	
10Dhex 000100001101	miss	set 110 linea 0 [A1 I 42] write allocate tag: 10hex ↓ [A1 I AB]* tag: 10hex	
10Ehex 000100001110	miss	set 111 linea 0 [90 I 75] write allocate tag: 10hex ↓ [39 I 75]* tag: 10hex	

continuare nella pagina seguente

Indirizzo	hit/ miss	Cache *(per ogni linea di cache indicare il contenuto del campo tag)*	Modifica memoria *M[ind.] = contenuto*
11Chex 000100011100	miss	set 110 linea 1 [A1 I 82] tag: 11hex	
108hex 000100001000	hit	set 100 linea 0 [D4 I 59]* tag: 10hex	
11Ehex 000100011110	miss	set 111 linea 1 [90 I 15] tag: 11hex	
10Ahex 000100001010	miss	set 101 linea 0 [AD I 23] write allocate tag: 10hex ↓ [98 I 23]* tag: 10hex	
121hex 000100100001	miss	set 000 linea 0 [FF I C6] tag: 12hex	

Aritmetica del calcolatore



Come dicevamo, nella CPU esista la ALU, in grado di eseguire le operazioni aritmetico-logiche. Ogni altra componente nel calcolatore serve questa unità, gestisce gli interi e riesce anche a gestire i numeri reali (oltre che operazioni logiche di controllo) .
La CPU richiede all'unità ALU di effettuare una particolare operazione logico-matematica sui dati. I dati (input) sono prelevati dai registri del processore. L'operazione e i dati sono elaborati in linguaggio binario. Una volta elaborata l'operazione, il risultato finale in output viene registrato nella locazione di memoria dell'unità aritmetico-logica, detta accumulatore, o nei registri di uscita del processore e restituito all'unità di controllo.

Per la r*appresentazione degli interi*, possiamo solo usare 0 e 1 per rappresentare tutto. I numeri positivi sono scritti in binario come sappiamo (es. $41 = 00101001$). Non c'è bisogno del segno.

Per la *rappresentazione in modulo e segno*, si considera come segno il bit più a sinistra (0 positivo, 1 negativo). Ad esempio: $+18 = 00010010$ $-18 = 10010010$
Problemi

- Per eseguire operazioni aritmetiche bisogna considerare sia i moduli che i segni
- Due rappresentazioni per lo 0: +0 *e* -0

Scritto da Gabriel

La rappresentazione in complemento a 2 è un tipo di rappresentazione in complemento dei numeri binari che agevola e semplifica le operazioni aritmetiche nel sistema binario da parte del computer. La rappresentazione in complemento a 2 si ottiene in modo simile alla rappresentazione in complemento tradizionale a 1.

- Numeri positivi. Nel caso dei numeri binari positivi la rappresentazione in complemento a 2 è uguale alla rappresentazione segno-grandezza. Vanno da 0 ad $2^{(n-1)} - 1$
- Numeri negativi. Nel caso dei numeri binari negativi la rappresentazione in complemento a 2 è uguale all'inversione di ogni cifra (bit) del numero binario in valore assoluto a cui viene addizionato il numero binario uno $(00000001)_2$. Il bit più significativo (a sinistra) è ad 1 ed i restanti $n - 1$ bit possono assumere $2^{(n-1)}$ configurazioni diverse

La seguente formula *definisce* il complemento a due:

Se sequenza di bit $a_{n-1}\ a_{n-2}\ \dots\ a_1\ a_0$,

$$\text{numero} = -2^{n-1}\, a_{n-1} + \sum_{(i=0,\ \dots,\ n-2)} 2^i\, a_i$$

- Bit più a sinistra è $-2n - 1$
- Per n bit: possiamo rappresentare tutti i numeri da $-2^{(n-1)}$ a $+2^{(n-1)} - 1$
- Per i numeri positivi, come per modulo e segno n zeri rappresentano lo 0, poi 1, 2, … in binario per rappresentare 1, 2, … positivi
- Per i numeri negativi, da n uni per il -1, andando indietro

Vediamo ad esempio il complemento a due su 3/4 bit:

Bit pattern	Value represented
011	3
010	2
001	1
000	0
111	−1
110	−2
101	−3
100	−4

Bit pattern	Value represented
0111	7
0110	6
0101	5
0100	4
0011	3
0010	2
0001	1
0000	0
1111	−1
1110	−2
1101	−3
1100	−4
1011	−5
1010	−6
1001	−7
1000	−8

Confrontiamo le rappresentazioni di k e $-k$

- da destra a sinistra, uguali fino al primo 1 incluso
- poi una il complemento dell'altra

Esempio (su 4 bit): $2 = 0010, -2 = 1110$

In generale, su 8/16 bit, si rappresentano come numeri:

- Complemento a 2 su 8 bit
  - Numero più grande: +127 = 01111111 = 2^7 -1
  - Numero più piccolo: -128 = 10000000 = -2^7
- Complemento a 2 su 16 bit
  - +32767 = 01111111 11111111 = 2^{15} - 1
  - -32768 = 10000000 00000000 = -2^{15}

La rappresentazione in complemento a 2 consente di effettuare l'operazione aritmetica dell'addizione tra due numeri binari indipendentemente dal segno dei due numeri. È sufficiente sommare i due numeri in complemento a 2 ignorando il riporto. Ad esempio, per sommare il numero negativo (-10)10 con il numero positivo $(+20)_{(10)}$ tramite la rappresentazione in complemento a 2 si procede nel seguente modo:

$(-10)_{10}$ ==> 10001010

$(+20)_{10}$ ==> 00010100

Il numero positivo $(+20)_{(10)}$resta invariato mentre il numero negativo $(-10)_{(10)}$deve essere elaborato mediante la regola della rappresentazione in complemento a due. In primo luogo, procediamo a calcolare il valore assoluto (senza segno) del numero binario.

$|10|_{10}$ ==> 00001010

Utilizziamo il valore assoluto del numero binario per il calcolo del complemento invertendo tutte le cifre (bit) del numero da zero a uno e viceversa.

00001010

11110101

Infine, per ottenere il complemento a due si addiziona il numero ottenuto con il numero uno $(00000001)_2$.

11110101+

00000001=

==========

11110110

Una volta calcolata la rappresentazione in complemento a due del numero negativo si somma il numero binario $(11110110)_2$ con il numero positivo dell'operazione aritmetica $(00010100)_2$senza tenere in conto l'eventuale resto finale.

11110110+ (complemento a due del numero -10)

00010100= (numero decimale +20)

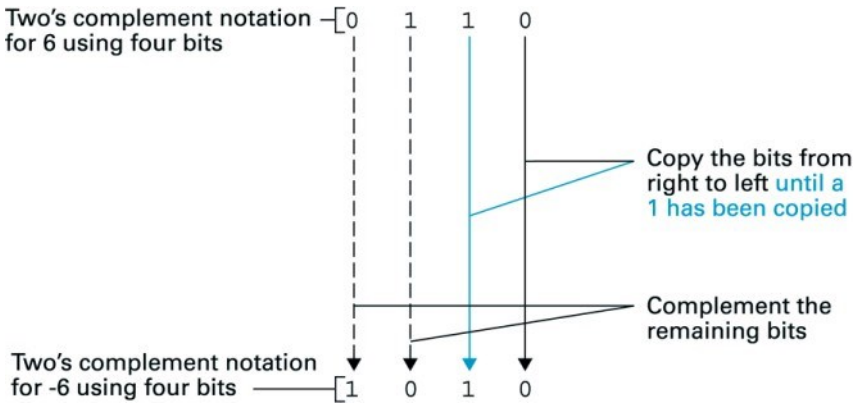
==========

00001010

Decodifica veloce

- Se bit più a sinistra =0 è positivo, altrimenti negativo
- Se positivo, basta leggere gli altri bit
- Se negativo, scrivere gli stessi bit da destra a sinistra fino al primo 1, poi complementare, e poi leggere
- Es.: 1010 è sicuramente negativo (bit più a sinistra vale $-2^{(n-1)}$), rappresenta 0110 (6), quindi -6

Da k a $-k$, quindi:



Altra decodifica veloce

- Data la rappresentazione di k (positivo), $-k$ si può anche ottenere così:
  - Complemento bit a bit della rappresentazione di k
  - Somma di 1 al risultato
- Esempio:
  - $2 = 0010$
  - Complemento: 1101
  - $1101 + 1 = 1110$
  - $-2 = 111$

Benefici

- Una sola rappresentazione dello zero
- Le operazioni aritmetiche sono facili
- La negazione è facile
  - $3 = 00000011$
  - Complemento Booleano 11111100
  - Somma di 1 11111101

Esercizi

- Da complemento a 2 a base 10:
  - 00011 → 3
  - 01111 → 15
  - 11100 → -4
  - 11010 → 6
  - 00000 → 0
  - 10000 → -16
- Da base 10 a complemento a 2 su 8 bit:
  - 6 → 00000110
  - -6 → 11111010
  - 13 → 00001101
  - -1 → 11111111
  - 0 → 00000000
- Numero più grande e più piccolo per la notazione in complemento a 2 su:
  - 4 bit → $2^3 - 1$
  - 6 bit → $2^5 - 1$
  - 8 bit → $2^7 – 1$

Scritto da Gabriel

Conversione tra diverse lunghezze

- Da una rappresentazione su n bit ad una rappresentazione dello stesso numero su $m\ bit\ (m > n)$
- Modulo e segno: facile
  - Bit di segno nel bit più a sinistra
  - $m - n$ zeri aggiunti a sinistra
- Esempio (da 4 a 8 bit): 1001 è 10000001
- Complemento a 2: stessa cosa del modulo e segno per numeri positivi
- Per numeri negativi: replicare il bit più significativo dalla posizione attuale alla nuova
  Esempi:
  - $+18\ (8\ bit) = \ 00010010$
  - $+18\ (16\ bit) = \ 00000000\ 00010010$
  - $-18\ (8\ bit) = \ 11101110$
  - $-18\ (16\ bit) = \ 11111111\ 11101110$

Opposto su numeri a complemento a 2
Strutturato in due passi: Complemento e Somma 1.
Rappresentiamo il caso speciale 1, come segue:

0 =	00000000
Complemento:	11111111
Somma 1:	+1
Risultato:	1 00000000

L'uno più a sinistra è un overflow, ed è ignorato. Quindi - 0 = 0

Così come il caso speciale 2, come nuovamente segue:

-128 =	10000000
Complemento:	01111111
Somma 1:	+1
Risultato:	10000000

Quindi, -(-128) = -128 !

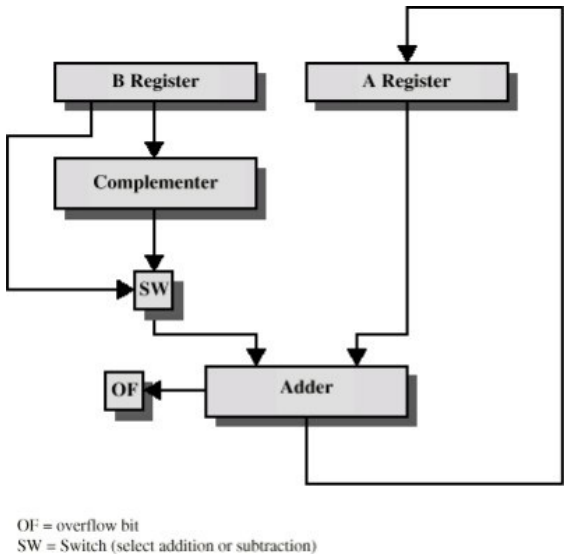
2^n stringhe su n bit, un numero positivo in più di quelli negativi: -2^n si può rappresentare, ma $+2^n$ no ➔ -2^n non può essere complementato

Per la somma, è una normale somma binaria (basta controllare il bit più significativo per l'overflow). Per la sottrazione, invece, basta avere i circuiti per somma e complemento.

- $(4\ bit)$: $7 - 5 = \ 7 + (-5) \ = \ 0111 \ + \ 1011 \ = \ 0010$
- $5 = \ 0101$ ➔ $-5 \ = \ 1011$

Problem in base ten		Problem in two's complement		Answer in base ten
3 + 2	→	0011 + 0010 = 0101	→	5
−3 + −2	→	1101 + 1110 = 1011	→	−5
7 + −5	→	0111 + 1011 = 0010	→	2

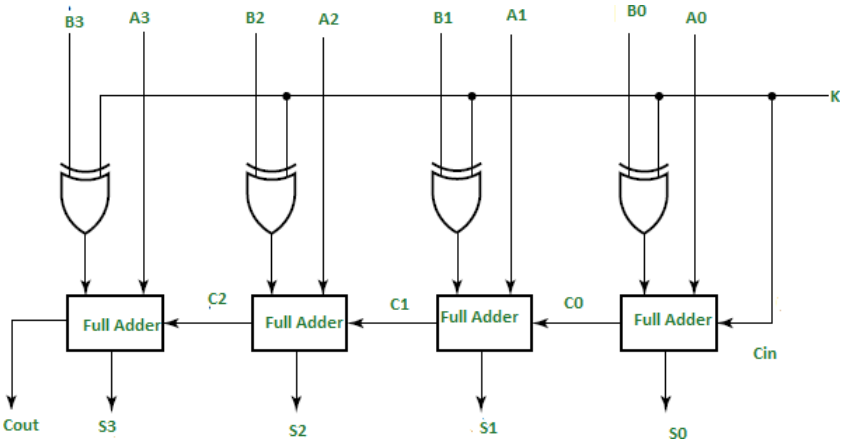
Ecco l'hardware necessario per somma e sottrazione:



OF = overflow bit
SW = Switch (select addition or subtraction)

Nei circuiti digitali, un sommatore-sottrattore binario è in grado di eseguire sia l'addizione che la sottrazione di numeri binari in un unico circuito. L'operazione eseguita dipende dal valore binario del segnale di controllo. È uno dei componenti dell'ALU (Unità Logica Aritmetica).

Consideriamo due numeri binari a 4 bit A e B come ingressi al Circuito Digitale per l'operazione con le cifre
$A0\ A1\ A2\ A3\ per\ A$
$B0\ B1\ B2\ B3\ per\ B$
Il circuito è composto da 4 sommatori completi, poiché stiamo eseguendo operazioni su numeri a 4 bit. C'è una linea di controllo K che contiene un valore binario di 0 o 1 che determina l'operazione di addizione o sottrazione.



Come mostrato in figura, il primo sommatore completo ha la linea di controllo direttamente come ingresso (input carry Cin), l'ingresso A0 (il bit meno significativo di A) è direttamente inserito nel sommatore completo. Il terzo ingresso è XOR di B0 e K. Le due uscite prodotte sono Somma/Differenza (S0) e Riporto/Carry (C0).

Se il valore di K (linea di controllo) è 1, l'uscita di B0(XOR)K=B0'(complemento B0). L'operazione sarebbe quindi A+(B0'). Ora la sottrazione del complemento a 2 per due numeri A e B è data da A+B'. Questo suggerisce che quando K=1, l'operazione eseguita sui numeri a quattro bit è la sottrazione.

Analogamente, se il valore di K=0, B0 (XOR) K=B0. L'operazione è A+B, ovvero una semplice addizione binaria. Questo suggerisce che quando K=0, l'operazione eseguita sui numeri a quattro bit è l'addizione.

Quindi C0 viene passato in serie al secondo sommatore completo come una delle sue uscite. La somma/differenza S0 viene registrata come il bit meno significativo della somma/differenza. A1, A2, A3 sono ingressi diretti al secondo, terzo e quarto sommatore completo. Il terzo ingresso è costituito da B1,

B2, B3, XORed (avendo eseguito lo XOR) con K, rispettivamente al secondo, terzo e quarto sommatore completo. I riporti C1 e C2 vengono passati in serie al successivo sommatore completo come uno degli ingressi. C3 diventa il riporto totale alla somma/differenza. S1, S2, S3 vengono registrati per formare il risultato con S0.
Per un sommatore-sottrattore binario a n bit, si utilizza un numero n di sommatori completi.

Può succedere anche un overflow: quando si sommano due numeri positivi tali che il risultato è maggiore del massimo numero positivo rappresentabile con i bit fissati (lo stesso per somma di due negativi). Se la somma dà overflow, il risultato non è corretto. Come si riconosce? Basta guardare il bit più significativo della risposta: se 0 (1) e i numeri sono entrambi negativi (positivi) è overflow. Su 4 bit, ad esempio:

- -4 (1100) + 4 (0100) = 10000 (0)
  - Riporto ma non overflow
- -4 (1100) - 1 (1111): 11011 (-5)
  - Riporto ma non overflow
- -7 (1001) -6 (1010) = 10011 (non è -13, ma 3)
  - Overflow
- + 7 (0111) + 7 (0111) = 1110 (non è 14, ma -2)
  - Overflow

Per la moltiplicazione, è più complessa; si calcola il prodotto parziale per ogni cifra e si sommano i prodotti parziali. Esempio:

```
   1011   Moltiplicando (11 decimale)
 x 1101   Moltiplicatore (13 decimale)
 ------
   1011   Prodotto parziale 1
  0000    Prodotto parziale 2
 1011     Prodotto parziale 3
1011      Prodotto parziale 4
 ------
10001111  Prodotto (143 decimale)
```

Nota: da due numeri di n bit potremmo generare un numero di 2n bit

La moltiplicazione di due numeri binari senza segno, X e Y, può essere eseguita con l'algoritmo longhand/a mano libera:

```
Y:    1011
X:   x 101
     ------
      1011
     0000
  + 1011
  --------
    110111
```

Si noti che i bit 0 in X contribuiscono con uno 0 al prodotto, mentre i bit 1 in X contribuiscono con Y spostato a sinistra per allinearsi con il bit corrispondente in X.

$$X = \sum_{i=0}^{n} x_i \cdot 2^i,$$

La base dell'algoritmo a mano lunga può essere compresa dalla rappresentazione binaria dei numeri interi senza segno. Siano X e Y numeri interi senza segno di n+1 bit:

$$Y = \sum_{j=0}^{n} y_j \cdot 2^j.$$

Il prodotto XY è dato da:

$$X \cdot Y = \left(\sum_{i=0}^{n} x_i \cdot 2^i\right) \cdot \left(\sum_{j=0}^{n} y_j \cdot 2^j\right)$$

$$= \sum_{i=0}^{n} \left(x_i \cdot 2^i \cdot \left(\sum_{j=0}^{n} y_j \cdot 2^j\right)\right)$$

$$= \sum_{i=0}^{n} x_i \cdot 2^i \cdot Y.$$

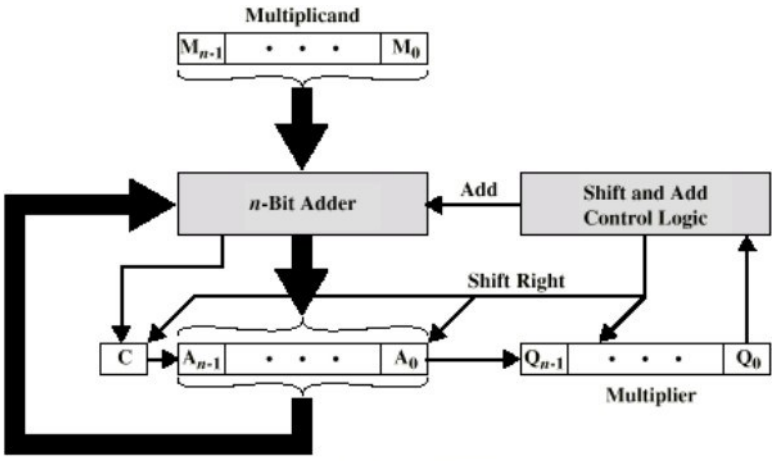
Ogni bit non nullo in X contribuisce con un termine costituito da Y moltiplicato per una potenza di 2.
Il prodotto di due numeri senza segno di n bit può richiedere fino a 2n bit poiché

$$(2^n - 1) \cdot (2^n - 1) = 2^{2n} - 2^{n+1} + 1 < 2^{2n}.$$

L'overflow si verifica quando il prodotto è maggiore di n bit.

L'algoritmo di moltiplicazione a mano per i numeri interi senza segno a n bit può essere implementato utilizzando l'addizione a n bit. Si noti che il bit meno significativo è determinato dal primo termine della somma. Questo bit può essere memorizzato e il termine può essere spostato di 1 bit a destra in preparazione dell'aggiunta del termine successivo. Questo processo viene ripetuto n volte per n bit.

In generale (spiegato nel dettaglio come qui, ma non chiesto) si ha da immagine l'hardware:



(a) Block Diagram

Implementazione

- Se $Q_0 = 0$, traslazione di C, A e Q
- Se $Q_0 = 1$, somma di A e M in A, overflow in C, poi traslazione di C, A, e Q
- Ripetere per ciascun bit di Q
- Prodotto (2n bit) in A e Q

Un esempio

C	A	Q	M		
0	0000	1101	1011	Initial Values	
0	1011	1101	1011	Add	First Cycle
0	0101	1110	1011	Shift	
0	0010	1111	1011	Shift	Second Cycle
0	1101	1111	1011	Add	Third Cycle
0	0110	1111	1011	Shift	
1	0001	1111	1011	Add	Fourth Cycle
0	1000	1111	1011	Shift	

Moltiplicare numeri in complemento a 2

- Per la somma, i numeri in complemento a 2 possono essere considerati come numeri senza segno

Esempio:

- $1001 + 0011 = 1100$
- Interi senza segno: $9 + 3 = 12$
- Complemento a 2: $-7 + 3 = -4$

Per la moltiplicazione questo non funziona! *Non funziona se almeno uno dei due numeri è negativo*.

Esempio: 11 (1011) x 13 (1101)

- Interi senza segno: 143 (10001111)
- Se interpretiamo come complemento a 2: -5 (1011) x -3 (1101) dovrebbe essere 15, invece otteniamo 10001111 (-113)

Bisogna usare la rappresentazione in complemento a due per i prodotti parziali:

1001 (9) ×0011 (3) 00001001 1001 × 2^0 00010010 1001 × 2^1 00011011 (27)	1001 (−7) ×0011 (3) 11111001 (−7) × 2^0 = (−7) 11110010 (−7) × 2^1 = (−14) 11101011 (−21)
(a) Unsigned integers	(b) Twos complement integers

Problemi anche se moltiplicatore negativo.
Una possibile soluzione:
1. convertire I fattori negativi in numeri positivi
2. effettuare la moltiplicazione
3. se necessario (-+ o +-), cambiare di segno il risultato

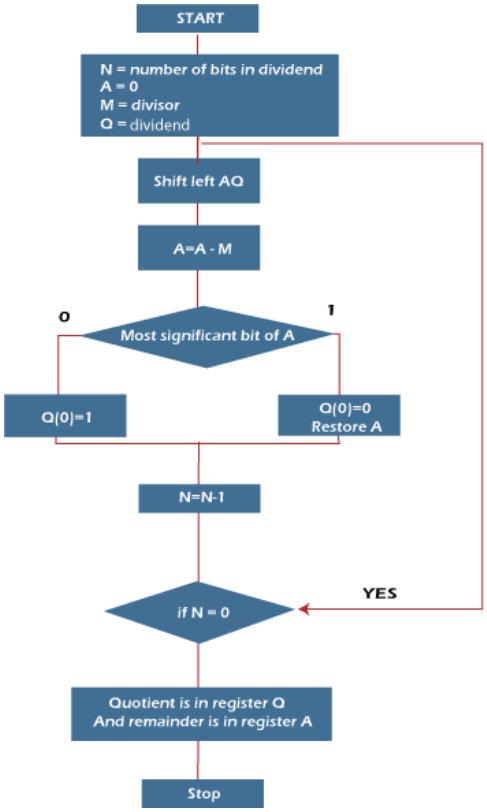
Soluzione usata → Algoritmo di Booth (approfondimento messo solo per completezza).

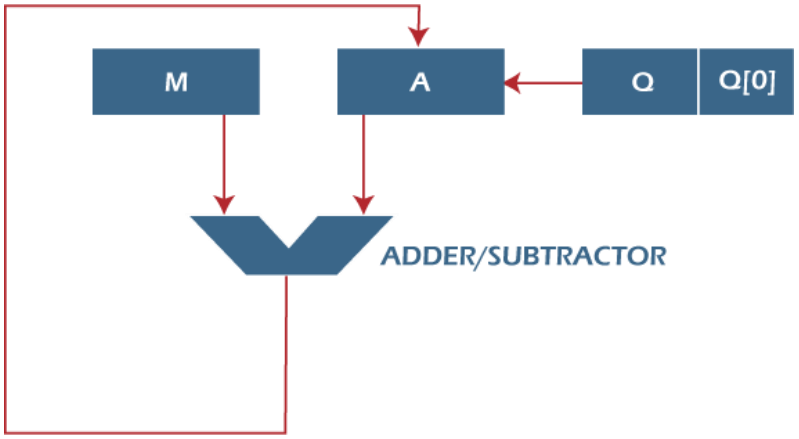
L'algoritmo di Booth fornisce una procedura per moltiplicare i numeri interi binari nella rappresentazione a complemento di 2 firmata in modo efficiente, cioè con un minor numero di addizioni/sottrazioni.
L'algoritmo si basa sul fatto che le stringhe di 0 nel moltiplicatore non richiedono alcuna addizione, ma solo uno spostamento, mentre una stringa di 1 nel moltiplicatore dal peso del bit 2^kal peso 2^m può essere trattata come $2^{(k+1)}$ a 2^m. Come in tutti gli schemi di moltiplicazione, l'algoritmo di stand richiede l'esame dei bit del moltiplicatore e lo spostamento del prodotto parziale. Prima dello spostamento, il moltiplicando può essere aggiunto al prodotto parziale, sottratto al prodotto parziale o lasciato invariato secondo le regole seguenti:

- Il moltiplicando viene sottratto dal prodotto parziale quando incontra il primo 1 meno significativo in una stringa di 1 nel moltiplicatore.
- Il moltiplicando viene aggiunto al prodotto parziale quando si incontra il primo 0 (a condizione che ci sia stato un precedente "1") in una stringa di 0 nel moltiplicatore.
- Il prodotto parziale non cambia quando il bit del moltiplicatore è identico al bit del moltiplicatore precedente.

Per la divisione, invece, si basa sugli stessi principi della moltiplicazione, ma usa traslazioni, somme e sottrazioni ripetute (altro approfondimento messo solo per completezza).

La divisione di ripristino viene solitamente eseguita sui numeri frazionari in virgola fissa. Quando si eseguono operazioni di divisione su due numeri, l'algoritmo di divisione fornisce due dati, ossia il quoziente e il resto. Questo algoritmo si basa sul presupposto che 0 < D < N. Con l'aiuto dell'insieme di cifre {0, 1}, nell'algoritmo di divisione ripristinante si forma la cifra quoziente q.





In questo caso, il registro Q viene utilizzato per contenere il quoziente e il registro A per contenere il resto. In questo caso, il divisore viene caricato nel registro M e il diviso a n bit viene caricato nel registro Q. 0 è il valore iniziale di un registro. I valori di questi tipi di registri vengono ripristinati al momento dell'iterazione. Per questo motivo si parla di ripristino.

Ora impareremo alcuni passaggi dell'algoritmo di divisione di ripristino, descritto come segue:

- Fase 1: in questa fase, i registri vengono inizializzati con il valore corrispondente, ossia il registro A conterrà il valore 0, il registro M conterrà il divisore, il registro Q conterrà il dividendo, mentre N viene utilizzato per specificare il numero di bit del dividendo.

- Fase 2: in questa fase, il registro A e il registro Q vengono trattati come una singola unità e il valore di entrambi i registri viene spostato a sinistra.

- Fase 3: Successivamente, il valore del registro M verrà sottratto dal registro A. Il risultato della sottrazione verrà memorizzato nel registro A.

Scritto da Gabriel

- Fase 4: ora si controlla il bit più significativo del registro A. Se questo bit del registro A è 0, allora il bit meno significativo del registro Q sarà impostato con il valore 1. Se il bit più significativo di A è 0, allora il bit meno significativo del registro Q sarà impostato con il valore 1. Se il bit più significativo di A è 1, allora il bit meno significativo del registro Q verrà impostato con il valore 0 e ripristinerà il valore di A, ovvero ripristinerà il valore del registro A prima della sottrazione con M.

- Fase 5: Successivamente, il valore di N viene decrementato. Qui n viene utilizzato come contatore.

- Fase 6: Se il valore di N è 0, il ciclo viene interrotto. In caso contrario, si deve tornare al passaggio 2.

- Passo 7: Questo è l'ultimo passo. In questo passaggio, il quoziente è contenuto nel registro Q e il resto è contenuto nel registro A.

Ad esempio:
In questo esempio, verrà eseguito un algoritmo di ripristino della divisione.
1. Dividendo = 11
2. Divisore = 3

N	M	A	Q	Operation
4	00011	00000	1011	Initialize
	00011	00001	011_	Shift left AQ
	00011	11110	011_	A = A - M
	00011	00001	0110	Q[0] = 0 And restore A
3	00011	00010	110_	Shift left AQ
	00011	11111	110_	A = A - M
	00011	00010	1100	Q[0] = 0
2	00011	00101	100_	Shift left AQ
	00011	00010	100_	A = A - M
	00011	00010	1001	Q[0] = 1
1	00011	00101	001_	Shift left AQ
	00011	00010	001_	A = A - M
	00011	00010	0011	Q[0] = 1

Non bisogna quindi dimenticare di ripristinare il valore del bit più significativo di A, che è 1. Quindi, il registro A contiene il resto 2, mentre il registro Q contiene il quoziente 3.

Rappresentazione e Aritmetica Numeri Reali

La rappresentazione dei numeri reali deve comprendere anche i numeri frazionari: essi possono anche essere rappresentati in binario (comprendendo la virgola), ad es:

Es.: $1001.1010 = 2^3 + 2^0 + 2^{-1} + 2^{-3} = 9.625$

Se numero fisso (senza virgola), la rappresentazione è limitante; se invece esiste la virgola, dobbiamo saper specificare dove si trova la virgola.

Conosciamo anche la *notazione scientifica* (decimale), per cui si eseguono arrotondamenti, rappresentando numeri molto grandi o molto piccoli con poche cifre.

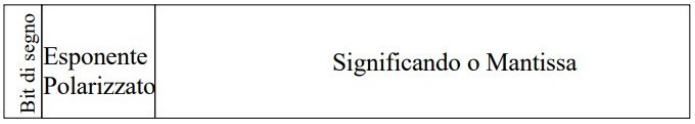
976.000.000.000.000 viene rappresentato come
$9{,}76 \times 10^{14}$

0,0000000000000976 viene rappresentato come
$9{,}76 \times 10^{-14}$

Lo stesso vale per i numeri binari, dove riconosciamo la struttura (poi descritta): $+/- \; SxB^{(+/-E)}$.

- S: significando o mantissa (la parte prima della virgola, parte intera)
- Si assume la virgola dopo una cifra della mantissa
- B: base

La struttura in FP/Floating Point/virgola mobile è come segue:



- Numero rappresentato:
  +/- 1.mantissa x $2^{esponente}$
- Esponente polarizzato: una valore fisso viene sottratto per ottenere il vero esponente
  k bit per esponente polarizzato → $2^{k-1} - 1$
  $e = ep - (2^{k-1} - 1)$

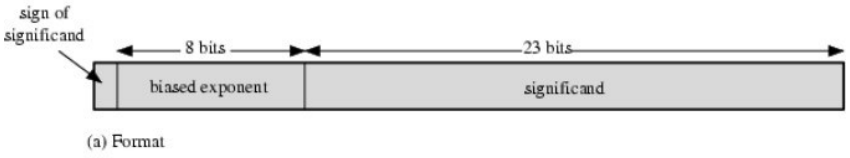
Es.: 8 bit → valori tra 0 e 255 → $2^7 - 1 = 127$ → esponente da -127 a +128

I numeri in virgola mobile, di solito, vanno *normalizzati*; quindi, l'esponente è aggiustato in modo che il bit più significativo della mantissa sia 1. Dato che è sempre 1 non c'è bisogno di specificarlo.

Numero: +/- 1.mantissa x $2^{esponente}$

L'1 non viene rappresentato nei bit a disposizione → se 23 bit per la mantissa, posso rappresentare numeri in [1,2). Se non normalizzato, aggiusto l'esponente: $0{,}1x2^0 = 1{,}0x2^{(-1)}$

Esempi



(a) Format

$1.1010001 \times 2^{10100} = 0\ 10010011\ 10100010000000000000000 = 1.638125 \times 2^{20}$
$-1.1010001 \times 2^{10100} = 1\ 10010011\ 10100010000000000000000 = -1.638125 \times 2^{20}$
$1.1010001 \times 2^{-10100} = 0\ 01101011\ 10100010000000000000000 = 1.638125 \times 2^{-20}$
$-1.1010001 \times 2^{-10100} = 1\ 01101011\ 10100010000000000000000 = -1.638125 \times 2^{-20}$

I numeri rappresentabili a 32 bit sono come segue:

Complemento a due: da -2^{31} a $+2^{31}-1$

Virgola mobile (con 8 bit per esponente):

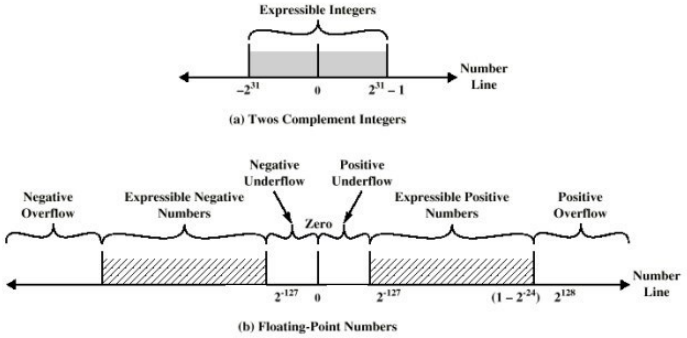
Esponente: da -127 (tutti 0) a 128 (tutti 1)

Mantissa: 1.0 (tutti 0) a $2-2^{-23}$ (tutti 1, cioè 1. 1...1, cioè $1 + 2^{-1} + 2^{-2} + ... + 2^{-23} = 2-2^{-23}$)

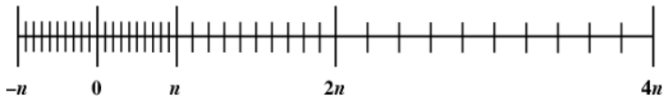
Negativi: Da $-2^{128} \times (2-2^{-23})$ a -2^{-127}

Positivi: da 2^{-127} a $2^{128} \times (2-2^{-23})$

Negativi minori di $-2^{128} \times (2-2^{-23})$ [overflow negativo]
Negativi maggiori di -2^{-127} [underflow negativo]
Positivi minori di 2^{-127} [underflow positivo]
Positivi maggiori di $2^{128} \times (2-2^{-23})$ [overflow positivo]
Non c'è una rappresentazione per lo 0
I numeri positivi e negativi molto piccoli (valore assoluto minore di 2^{-127}) possono essere approssimati con lo 0
Non rappresentiamo più numeri di 2^{32}, ma li abbiamo divisi in modo diverso tra positivi e negativi
I numeri rappresentati non sono equidistanti tra loro: più densi vicino allo 0 (errori di arrotondamento)



In merito invece alla *densità* ed alla *precisione*, si considera un esempio con 8 bit per l'esponente e 23 per la mantissa (1 di segno ovviamente). Se più bit per l'esponente (e meno per la mantissa), espandiamo l'intervallo rappresentabile, ma i numeri sono più distanti tra loro: si ha quindi minore precisione.
La precisione, infatti, aumenta solo aumentando il numero dei bit (a 32 bit si ha precisione singola, a 64 bit precisione doppia).



Per la densità:

Numero (positivo) = $1,m \times 2^e$

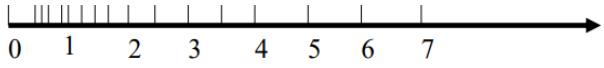
Fissato e, i numeri rappresentabili sono tra 2^e (mantissa tutta a 0) e $2^e \times (2-2^{-23})$

- Quanti numeri? 2^{23}

Consideriamo adesso e+1 ➔ i numeri rappresentabili sono tra $2x2^e$ e $2x2^e \times (2-2^{-23})$

- Intervallo grande il doppio
- Quanti numeri? Sempre 2^{23}

I numeri rappresentati con 2 bit di esponente e 2 di mantissa:



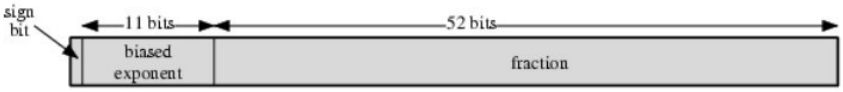
fe (due bit per esponente), ab (due bit per mantissa)

Numero= $2^{fx2+e-1}x(1+ax2^{-1}+bx2^{-2})$

f	e	a	b		numero	
0	0	0	0		0.5	
0	0	0	1		0.625	
0	0	1	0		0.75	
0	0	1	1		0.875	
0	1	0	0		1	
0	1	0	1		1.25	
0	1	1	0		1.5	
0	1	1	1		1.75	
1	0	0	0		2	
1	0	0	1		2.5	
1	0	1	0		3	
1	0	0	1		3.5	
1	1	0	0		4	
1	1	0	1		5	
1	1	1	0		6	
1	1	1	1		7	

Parliamo ora dello standard IEEE 754, che definisce un metodo per la rappresentazione dei numeri in virgola mobile, o floating point. Il numero reale viene dapprima rappresentato in binario, convertendo opportunamente la parte intera e la parte frazionaria.
Viene usato in formato singolo a 32 bit e a formato doppio a 64 bit, con un esponente con 8/11 bit, 1 implicito a sinistra della virgola e una serie di formati estesi (più bit per mantissa ed esponente) per risultati intermedi (più precisi → minore possibilità di risultato finale con eccessivo arrotondamento).

(a) Single format



(b) Double format

Alcune combinazioni (es.: valori estremi dell'esponente) sono interpretate in modo speciale.

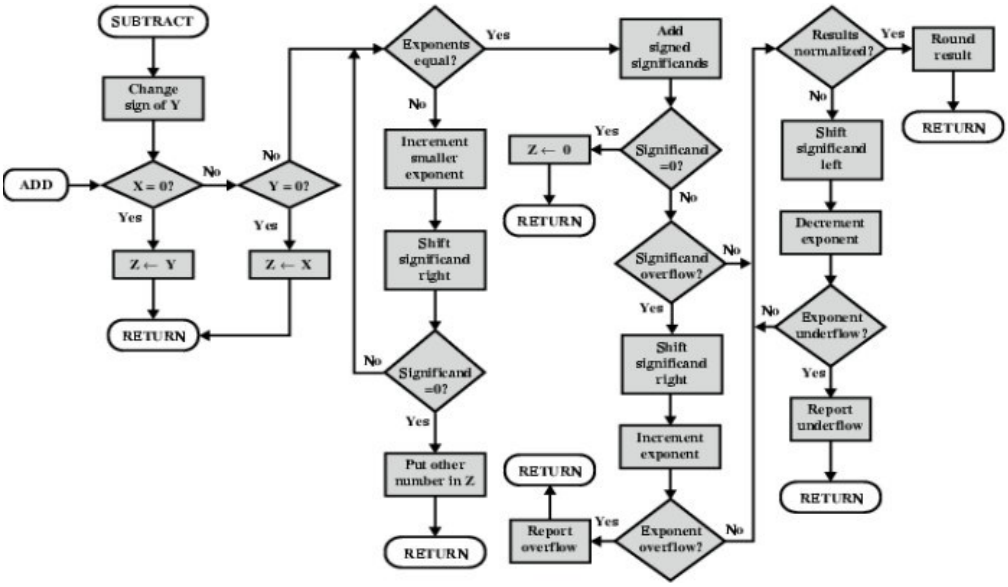
- Esponente polarizzato da 1 a 254 (cioè esponente da -126 a +127): numeri normalizzati non nulli in virgola mobile → $+/-2^{(e-127)}x1.f$.
- Esponente 0, mantissa (frazione) 0: rappresenta 0 positivo e negativo
- Esponente con tutti 1, mantissa 0: infinito positivo e negativo. L'overflow può essere errore o dare il valore infinito come risultato
- Esponente 0, mantissa non nulla: numero denormalizzato
    - Bit a sinistra della virgola: 0, vero esponente: -126
    - Positivo o negativo
    - Numero: $2\text{^}(-126)\ x\ 0.f$
- Esponente tutti 1, mantissa non nulla: errore (Not A Number)

Per le singole operazioni:

1) Somma e sottrazione
   Quattro fasi:

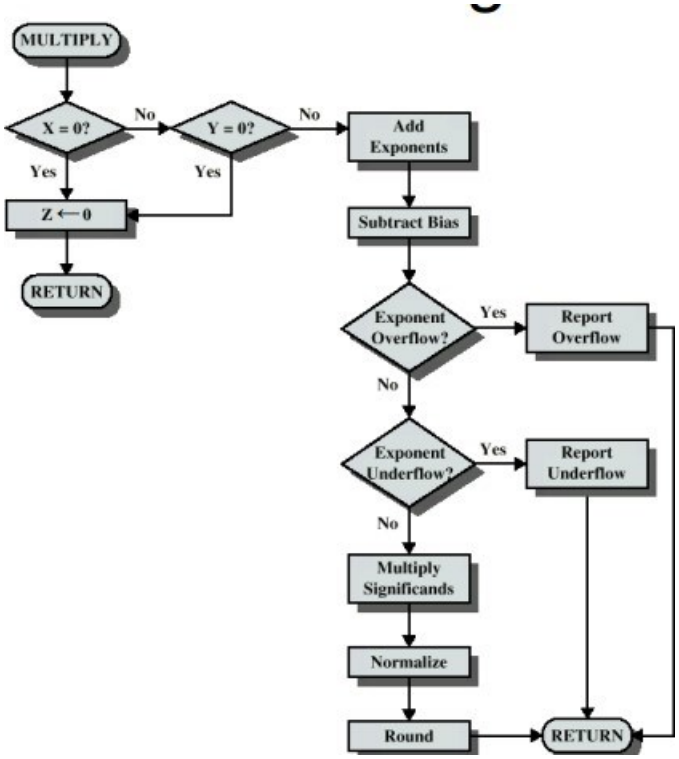
- Controllo dello zero
    - Se uno dei due è 0, il risultato è l'altro numero
- Allineamento delle mantisse
    - Rendere uguali gli esponenti
- Somma o sottrazione delle mantisse
- Normalizzazione del risultato
    - Traslare a sinistra finché la cifra più significativa è diversa da 0

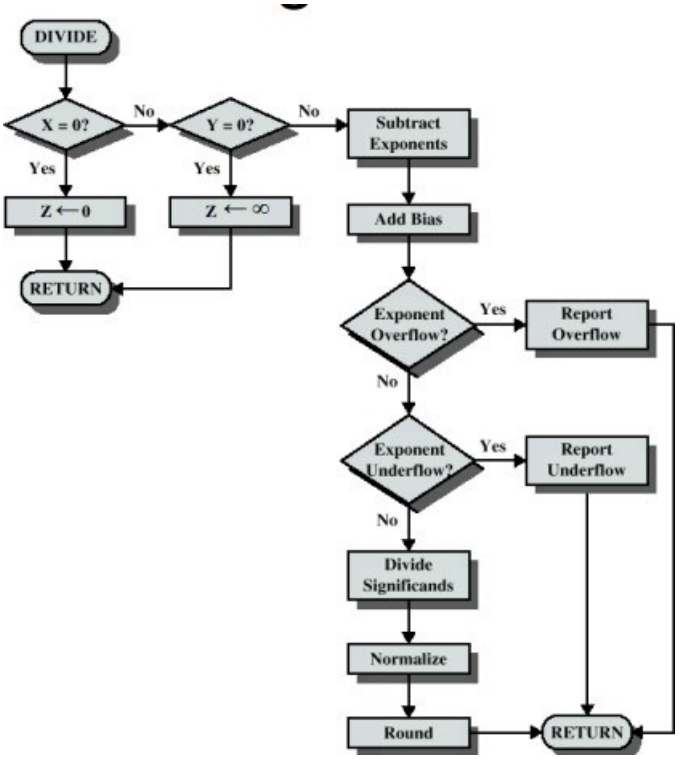
Esempio (in base 10):
$(123 \times 10^0) + (456 \times 10^{-2})$
123 + 4,56 → Non possiamo semplicemente sommare 123 a 456: il 4 deve essere allineato sotto il 3
Nuova rappresentazione: $(123 \times 10^0) + (4{,}56 \times 10^0)$
Adesso posso sommare le mantisse
(123 + 4,56 = 127,56)
Risultato: $127{,}56 \times 10^0$



2) Moltiplicazione e divisione
   Per la moltiplicazione, i significanti vengono moltiplicati, gli esponenti vengono sommati e il risultato viene arrotondato e normalizzato.
   Allo stesso modo, la divisione si ottiene sottraendo l'esponente del divisore dall'esponente del dividendo e dividendo il significante del dividendo per il significante del divisore.
   Non ci sono problemi di annullamento o di assorbimento con la moltiplicazione o la divisione, anche se piccoli errori possono accumularsi quando le operazioni vengono eseguite in successione.
   Può essere riassunto dei nei seguenti passi:

- Controllo dello zero
- Somma degli esponenti
- Sottrazione polarizzazione
- Moltiplicazione/divisione operandi
- Normalizzazione
- Arrotondamento





Per ottenere maggiore precisione nel risultato raggiunto, si usano i bit di guardia.
Di solito operandi nei registri della ALU, che hanno più bit di quelli necessari per la mantissa +1 → i bit più a destra sono messi a 0 e permettono di non perdere bit se i numeri vengono shiftati a destra.

Es.: X-Y, con $Y=1,11...11 \times 2^0$ e $X=1,00...00 \times 2^1$
Y va shiftato a destra di un bit, cioè diventa
$0,111...11 \times 2^1$ ➔ un 1 viene perso senza i bit di guardia
Risultato:

- Senza bit di guardia:
  $(1,0...0 - 0,1...1) \times 2 = 0,0...01 \times 2 = 1,0...0 \times 2^{-22}$
- Con bit di guardia:
  $(1,0...0 - 0,1...11) \times 2 = 0,0...001 \times 2 = 1,0...0 \times 2^{-23}$

Similmente, il risultato può essere arrotondato. Se il risultato è in un registro più lungo, quando lo si riporta nel formato in virgola mobile, bisogna arrotondarlo.

Quattro approcci:

- Arrotondamento al più vicino (default)
  - Bit aggiuntivi che iniziano con 1 ➔ sommo 1
    - Bit aggiuntivi 10...0 ➔ sommo 1 se l'ultimo bit è 1, altrimenti 0
  - Bit aggiuntivi che iniziano con 0 → elimino
- Arrotondamento (per eccesso) a $+\infty$ e arrotondamento (per difetto) a $-\infty$
  - Usati nell'aritmetica degli intervalli
- Arrotondamento a 0 (cioè troncamento dei bit in più)

Esercizi su virgola mobile

Es 1: Supponendo di avere a disposizione 3 bit per l'esponente e 4 bit per la mantissa:

a) dire quali numeri rappresentano le seguenti configurazioni di bit
   - 1 111 1111
   - 0 101 1011
   - 0 110 1111

b) dare la rappresentazione binaria in virgola mobile dei seguenti numeri reali
   - 6,5
   - 2,25
   - -7,3

Soluzione:

3 bit per esponente (quindi e-3), 4 bit per mantissa

a)

- 1 111 1111 → $-1{,}1111 \times 2^{7-3} = -11111 = -(16+8+4+2+1) = -31$
- 0 101 1011 → $+1{,}1011 \times 2^{5-3} = +110{,}11 = +(6+0{,}5+0{,}25) = +6{,}75$
- 0 110 1111 → $1{,}1111 \times 2^{6-3} = +1111{,}1 = +15{,}5$

b)

- 6,5 → 110,1 → $1{,}101 \times 2^{2}$ → $1{,}101 \times 2^{5-3}$ → 0 101 1010
- 2,25 → 10,01 → $1{,}001 \times 2$ → $1{,}001 \times 2^{4-3}$ → 0 100 0010
- -7,3 → -111,01001… → $-1{,}1101001\ldots \times 2^{2}$ → $-1{,}1101 \times 2^{5-3}$ → 1 101 1101 (approssimato)

Es 2: Supponendo di avere a disposizione 3 bit per l'esponente e 4 bit per la mantissa:

a) dire quali numeri rappresentano le seguenti configurazioni di bit
   - 0 111 1011
   - 1 100 1011
   - 0 101 1111

b) dare la rappresentazione binaria in virgola mobile dei seguenti numeri reali
   - 2,3
   - -0,25

Es 3: Supponendo di avere a disposizione 3 bit per l'esponente e 4 bit per la mantissa, dire quale è

- il numero più grande positivo rappresentabile
- il numero più piccolo positivo rappresentabile
- il numero più grande negativo rappresentabile
- il numero più piccolo negativo rappresentabile

Linguaggio macchina

Il linguaggio macchina è un linguaggio di basso livello composto da numeri o bit binari che un computer può comprendere. È noto anche come codice macchina o codice oggetto ed è estremamente difficile da comprendere. L'unico linguaggio che il computer comprende è il linguaggio macchina. Tutti i programmi e i linguaggi di programmazione, come Swift e C++, producono o eseguono programmi in linguaggio macchina prima di essere eseguiti su un computer. Quando si esegue un compito specifico, anche il più piccolo processo, il linguaggio macchina viene trasmesso al processore di sistema. I computer sono in grado di comprendere solo i dati binari in quanto sono dispositivi digitali.

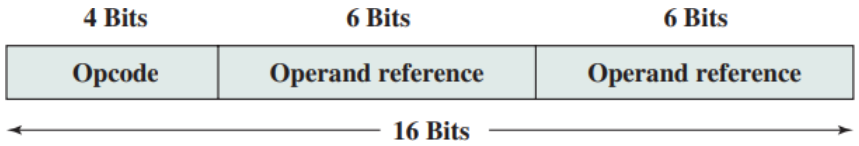
Gli elementi di un'istruzione macchina sono:

1) Codice operativo → Specifica l'operazione da eseguire
2) Riferimento all'operando sorgente → Specifica l'operando che rappresenta l'input dell'operazione
3) Riferimento all'operando risultato → Dove mettere il risultato
4) Riferimento all'istruzione successiva

Gli operandi possono essere in vari punti:

- Memoria centrale (o virtuale) → Si deve fornire l'indirizzo
- Registri della CPU → Ognuno ha un numero che lo identifica
- Dato immediato nella istruzione
- Dispositivi di I/O → Numero modulo o indirizzo di M

Le istruzioni vengono rappresentate come sequenze di bit (divise in campi), usando una rappresentazione simbolica delle configurazioni di bit (es.: ADD, SUB, LOAD). Anche gli operandi hanno una rappresentazione simbolica (es.: ADD A,B).
Ecco un esempio del formato di un'istruzione:



Similmente, ci sono vari tipi di istruzioni:

- Elaborazione dati → Istruzione aritmetiche e logiche, di solito sui registri della CPU
- Immagazzinamento dei dati in M o viceversa
- Trasferimento dei dati (I/O)
- Controllo del flusso del programma → Salto con o senza test

Per un'istruzione, sono necessari un certo numero di indirizzi:

- Un indirizzo per ogni operando (1 o 2)
- Uno per il risultato
- Indirizzo istruzione successiva

Quindi al massimo quattro indirizzi, tuttavia è molto raro, e sarebbe molto dispendioso. Di solito 1, 2 o 3 per gli operandi/risultati.

Per il numero di indirizzi, abbiamo varie situazioni:

- 1 indirizzo
  - il secondo indirizzo è implicito
  - di solito si tratta di un registro (accumulatore)
  - situazione tipica nei primi calcolatori
- 0 indirizzi
  - tutti gli indirizzi sono impliciti
  - utilizza una pila (stack)

n Ad esempio, $c = a + b$ è realizzato come segue
$push\ a \qquad push\ b\ add \qquad pop\ c$

Number of Addresses	Symbolic Representation	Interpretation
3	OP A, B, C	A ← B OP C
2	OP A, B	A ← A OP B
1	OP A	AC ← AC OP A
0	OP	T ← (T − 1) OP T

AC = accumulator
T = top of stack
(T − 1) = second element of stack
A, B, C = memory or register locations

Quindi:

- Meno indirizzi → istruzioni più elementari (e più corte), quindi CPU meno complessa. Però più istruzioni per lo stesso programma → tempo di esecuzione più lungo.
- Più indirizzi → istruzioni più complesse
- Indirizzo di M o registro: meno bit per indicare un registro

Instruction	Comment
SUB Y, A, B	Y ← A − B
MPY T, D, E	T ← D × E
ADD T, T, C	T ← T + C
DIV Y, Y, T	Y ← Y ÷ T

(a) Three-address instructions

Instruction	Comment
MOVE Y, A	Y ← A
SUB Y, B	Y ← Y − B
MOVE T, D	T ← D
MPY T, E	T ← T × E
ADD T, C	T ← T + C
DIV Y, T	Y ← Y ÷ T

(b) Two-address instructions

Instruction	Comment
LOAD D	AC ← D
MPY E	AC ← AC × E
ADD C	AC ← AC + C
STOR Y	Y ← AC
LOAD A	AC ← A
SUB B	AC ← AC − B
DIV Y	AC ← AC ÷ Y
STOR Y	Y ← AC

(c) One-address instructions

Figure 12.3 Programs to Execute $Y = \frac{A - B}{C + (D \times E)}$

Per progettare un set/insieme di istruzioni, si deve pensare ad un certo numero di cose:

- Repertorio → quante e quali operazioni
- Tipi di dato → su quali dati
- Formato → lunghezza, numero indirizzi, dimensione campi, ...
- Registri → numero dei registri della CPU indirizzabili dalle istruzioni
- Indirizzamento → modo di specificare gli indirizzi degli operandi

Gli operandi sono di vario tipo:
indirizzi (interi senza segno), numeri (limite al modulo o alla precisione), caratteri, dati logici.

I numeri, in particolare, sono interi a virgola fissa/virgola mobile e, nel caso di operazioni di I/O, si usano i decimali impaccati (packed decimals), che specifica un metodo di codifica dei numeri decimali utilizzando ogni byte per rappresentare due cifre decimali. La rappresentazione decimale a pacchetto memorizza i dati decimali con precisione esatta. La parte frazionaria del numero è determinata dal formato perché non ci sono mantissa ed esponente separati.

- Cifra decimale = 4 bit (0=0000, 1=0001, 2=0010, .. 8=1000, 9=1001)
- Inefficiente: solo 10 delle 16 configurazioni vengon usate
- Es.: 246 = 0010 0100 0110

In merito ai caratteri, si usa il classico standard del codice ASCII (American Standard Code for Information Exchange). Un carattere = 7 bit → 128 caratteri in totale.
Caratteri alfabetici + caratteri di controllo. Di solito 8 bit: un bit per controllo di errori di trasmissione (controllo di parità).

Settato in modo che il numero totale di bit a 1 sia sempre pari (o sempre dispari)
Es.: 0011100 → ottavo bit a 1
Se si ricevono 8 bit con n.ro dispari di 1, c'è stato un errore di trasmissione

I dati logici sono formati da n bit, invece che un singolo dato, per manipolare i bit separatamente.
Per l'architettura x86 (Intel), si hanno come caratteristiche per tipi di dati trattati:

8 (byte), 16 (parola), 32 (doppia parola), o 64 (quadword) bit, e 128 (quadword doppia)

L'indirizzamento è per unità di 8 bit

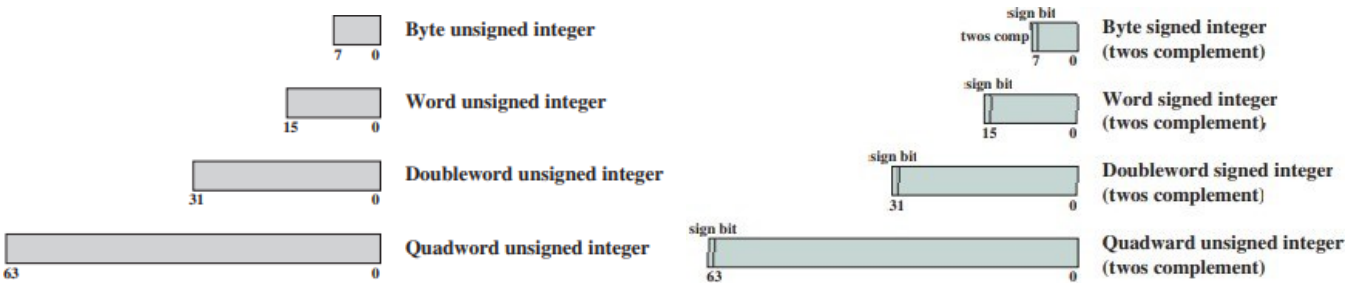
Una doppia parola di 32 bit inizia da un indirizzo divisibile per 4

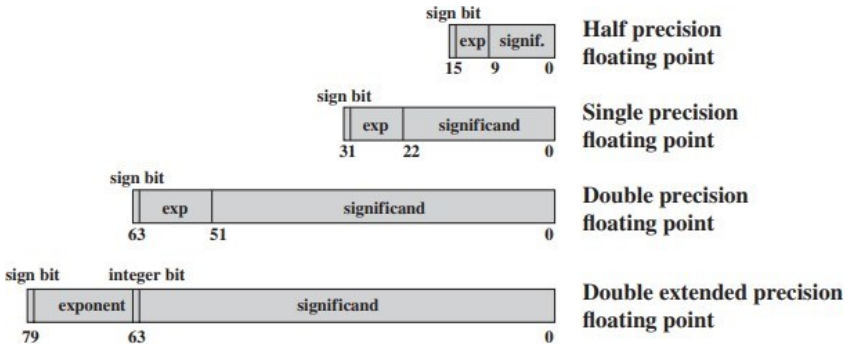
Non necessario allineamento indirizzi per le strutture dati in memoria

Allineamento per trasferimento dati (bus)

Più specificamente:

General	Byte, word (16 bits), doubleword (32 bits), quadword (64 bits), and double quadword (128 bits) locations with arbitrary binary contents.
Integer	A signed binary value contained in a byte, word, or doubleword, using twos complement representation.
Ordinal	An unsigned integer contained in a byte, word, or doubleword.
Unpacked binary coded decimal (BCD)	A representation of a BCD digit in the range 0 through 9, with one digit in each byte.
Packed BCD	Packed byte representation of two BCD digits; value in the range 0 to 99.
Near pointer	A 16-bit, 32-bit, or 64-bit effective address that represents the offset within a segment. Used for all pointers in a nonsegmented memory and for references within a segment in a segmented memory.
Far pointer	A logical address consisting of a 16-bit segment selector and an offset of 16, 32, or 64 bits. Far pointers are used for memory references in a segmented memory model where the identity of a segment being accessed must be specified explicitly.
Bit field	A contiguous sequence of bits in which the position of each bit is considered as an independent unit. A bit string can begin at any bit position of any byte and can contain up to 32 bits.
Bit string	A contiguous sequence of bits, containing from zero to $2^{23} - 1$ bits.
Byte string	A contiguous sequence of bytes, words, or doublewords, containing from zero to $2^{23} - 1$ bytes.
Floating point	See Figure 12.4.
Packed SIMD (single instruction, multiple data)	Packed 64-bit and 128-bit data types.





Ulteriormente, descriviamo i vari tipi di operazioni:

- Trasferimento Dati, per cui si deve specificare
  - Sorgente: dove è il dato da trasferire
  - Destinazione: dove va messo
  - Lunghezza del dato da trasferire.
    Diverse scelte:
  - Esempio: codici operativi diversi per trasferimenti diversi (L, LH, LR,LER, LE, LDR, LD in IBM 370) o stesso codice (MOV in VAX) ma specifica nell'operando
- Aritmetiche, per cui somma, sottrazione, moltiplicazione, divisione n Interi con segno sempre (spesso anche per numeri in virgola mobile). Possono includere anche: incremento, decremento, negazione, valore assoluto
- Logiche, quindi operazioni sui bit (AND, OR, NOT, XOR, EQUAL). Possono essere eseguite in parallelo su tutti i bit di un registro (And come maschera)

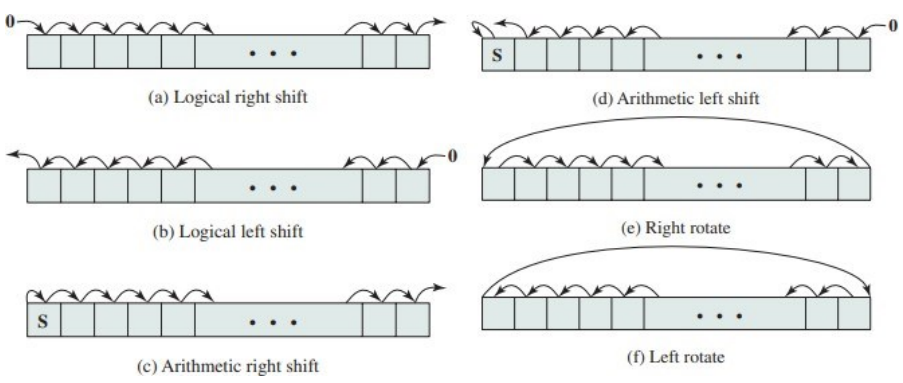
Esempio di operazione:

Parole da 16 bit con 2 caratteri (8 bit ciascuno)
Per inviare il carattere di sinistra a un modulo di I/O:

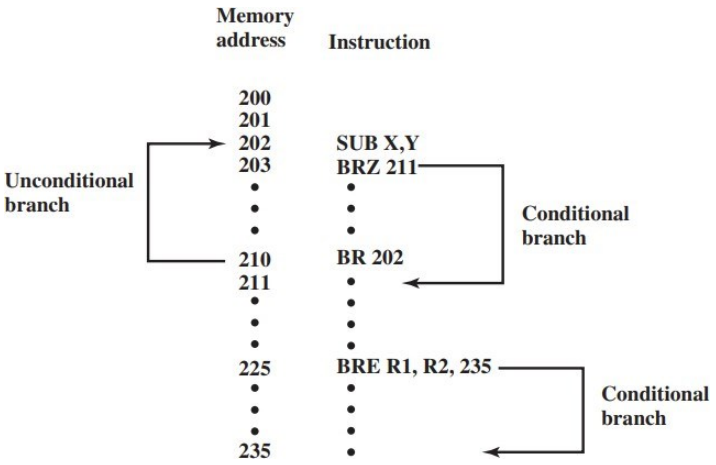
- Carico la parola in un registro a 16 bit
- AND del registro con 1111111100000000
- Traslo a destra per 8 volte
- Mando al modulo di I/O il registro (legge gli 8 bit più a destra)
- Per il carattere di destra, AND con 0000000011111111, e non serve la traslazione

Similmente, esistono operazioni di:

- Shift, quindi ci si sposta o verso destra o verso sinistra logicamente/aritmeticamente
- Rotazione, quindi una volta che si shifta, arrivando alla fine, si riparte dall'inizio o dalla fine



- Trasferimento del controllo:
  1) Salto condizionato (branch)
     - Es.: salta a x se il risultato è 0
     - Registro condizione o più operandi
       - Es.: BRE R1, R2, X
     - Perché saltare?
       - Istruzioni da eseguire varie volte
       - Decidere cosa fare sulla base del verificarsi di certe condizioni
       - Programmazione modulare



  2) Salto incondizionato (skip)
     - Scavalca un'istruzione e passa alla successiva
     - Non ha operandi
     - Per usare lo spazio operandi
       - es.: incrementa e salta se 0 (istruzione ISZ)

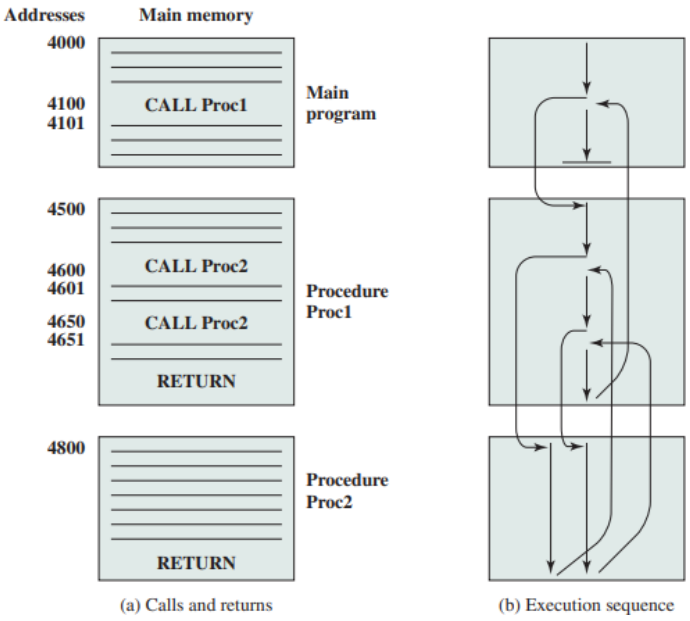
Introduciamo il concetto di procedura: pezzo di programma a cui si dà un nome, in modo da eseguirlo (chiamarlo) da qualunque punto di un programma indicando il suo nome. Si usano per due motivi:

- Risparmio codice: scrivo solo una volta un pezzo di codice
- Modularità: posso affidare la scrittura di una procedura ad un altro programmatore

Due istruzioni principali:

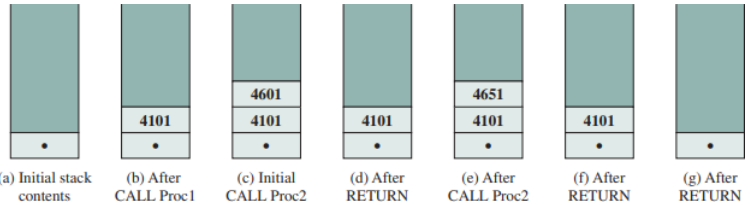
- Chiamata e ritorno (entrambe di salto)

Similmente, possono essere annidate (caso di esempio qui a fianco).



Luoghi per memorizzare l'indirizzo di ritorno

- Un registro
  - CALL X provoca:
    RN ← PC + D (D=lunghezza istruzione)
    PC ← X
- Inizio della procedura chiamata
  - CALL X provoca:
    X ← PC + D
    PC ← X+1
- Cima della pila: porzione di M dove le scritture/letture avvengono sempre in cima
  - Gli indirizzi di ritorno vengono memorizzati in cima alla pila, uno dopo l'altro, e vengono presi nell'ordine inverso alla chiusura delle procedure. Esempio di uso nell'immagine:



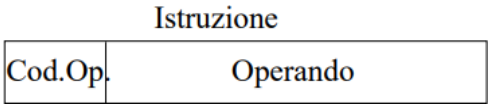
Parliamo di linguaggio assembly, un tipo di linguaggio di programmazione di basso livello destinato a comunicare direttamente con l'hardware di un computer. A differenza del linguaggio macchina, che consiste in caratteri binari ed esadecimali, i linguaggi assembly sono progettati per essere leggibili dall'uomo. I codici operativi sono dati da simboli e indirizzi numerici sono indirizzi simbolici (per operandi ed istruzioni). Per tradurre dall'linguaggio assembly a linguaggio macchina, vi è l'assembler/assemblatore.
I byte sono memorizzati in due modi (introducendo il concetto di *endianness*, un termine che descrive l'ordine in cui una sequenza di byte viene memorizzata nella memoria del computer)

- Big endian, che significa che i dati vengono archiviati prima big end. In più byte, il primo byte è il più grande o rappresenta il valore primario.
- Little endian, in cui i dati vengono memorizzati prima di tutto. In questo caso, con pezzi multibyte, è l'ultimo pezzo più grande o che ha il valore primario a cui vengono aggiunti o concatenati i valori successivi.

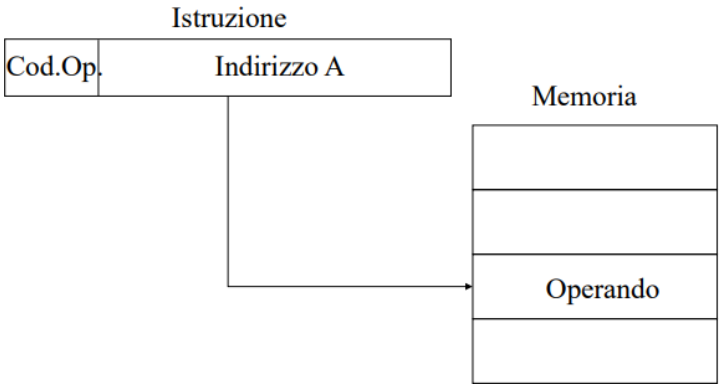
Modi di indirizzamernto

Vari modi di specificare l'indirizzo
degli operandi

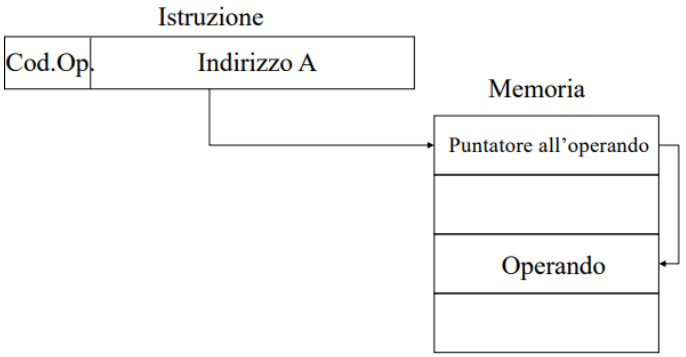
- Immediato, per cui l'operando è parte dell'istruzione (campo indirizzo)
  - Vantaggio: nessun accesso in M per prendere l'operando
  - Svantaggio: valore limitato dalla dimensione del campo indirizzo



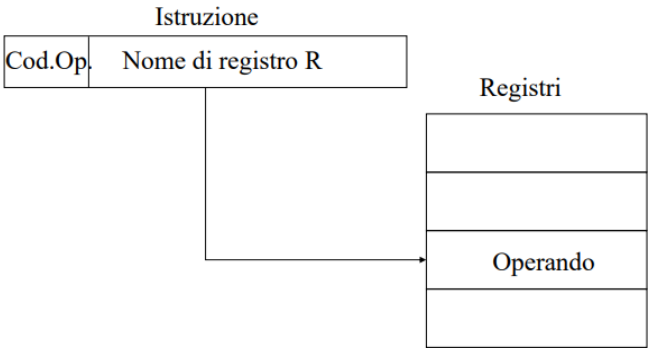
- Diretto, per cui il campo indirizzo = indirizzo dell'operando
  - Esempio: ADD A
    - Somma il contenuto della cella A all'accumulatore
    - Bisogna andare in M all'indirizzo A per trovare l'operando
  - Un singolo accesso in M per prendere l'operando
  - Spazio di indirizzamento limitato



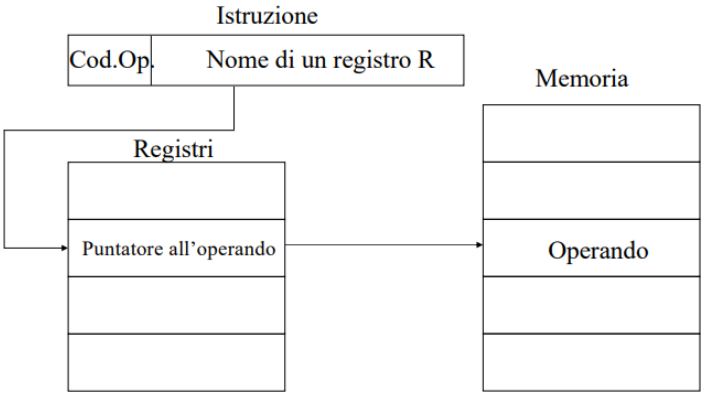
- Indiretto, per cio il campo indirizzo contiene l'indirizzo di una cella di M, che contiene l'indirizzo dell'operando
    - Vantaggio: parole di lunghezza N permettono di indirizzare 2^N entità diverse
        - In realtà 2^K, dove K è la lunghezza del campo indirizzo
    - Svantaggio: due accessi in M per ottenere l'operando.
    - Esempio: ADD A
        - Somma il contenuto della cella puntata dal contenuto di A all'accumulatore



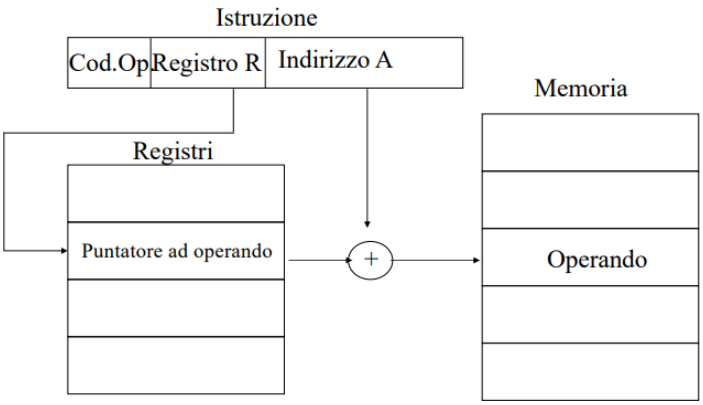
- Registro, per cui l'operando è in un registro indicato nel campo indirizzo. Si ha un numero limitato di registri e in generale pochi bit necessari per il campo indirizzo.
    - Istruzioni più corte
    - Fase di fetch più veloce (nessun accesso in M per prendere l'operando)



- Registro indiretto, con lo stesso principio dell'indirizzamento indiretto.
    - L'operando è in una cella di M puntata dal contenuto del registro R
    - Grande spazio di indirizzamento (2^n)
    - Un accesso in meno in M rispetto all'indirizzamento indiretto



- Spiazzamento, che è una combinazione di indirizzamento diretto e indirizzamento registro indiretto.
    - Il campo indirizzo ha due sottocampi
        - A = valore di base (diretto)
        - R = registro che contiene l'indirizzo di un valore da sommare ad A per ottenere l'indirizzo
        - o viceversa (R base e A spiazzamento)

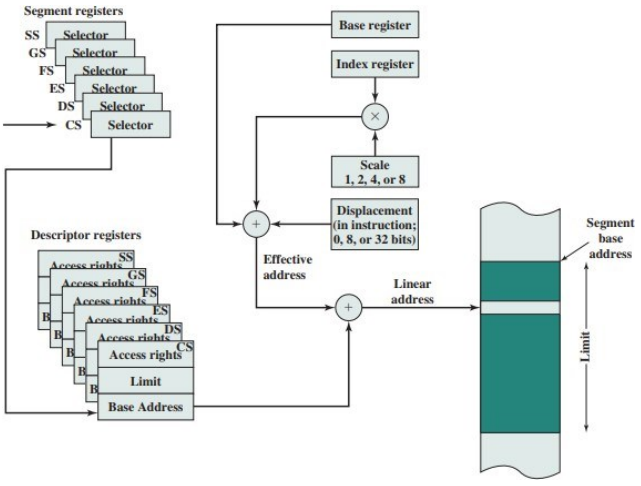


- Relativo, una versione dell'indirizzamento con spiazzamento
    - R = registro PC (program counter)
    - Indirizzo dell'operando = A + (PC)
        - A celle dalla cella di M puntata da PC
- Indirizzamento, registro base
    - A contiene lo spiazzamento
    - R contiene il puntatore all'indirizzo base
        - R può essere esplicito o implicito

L'indicizzazione usa infatti funziona in questo modo:

- A = base
- R = spiazzamento
- Esempio: elenco di dati memorizzati in M a partire da un indirizzo A
    - Per accedere a tutti, la sequenza di indirizzi è A, A+1, A+2, ...
    - A viene messo nel campo indirizzo e il registro (indice) contiene 0 all'inizio e viene incrementato di 1 dopo ogni accesso

- Indirizzamento a pila, come detto lo stack, sequenza lineare di locazioni riservate di M.
    - Puntatore (nel registro SP, stack pointer) con l'indirizzo della cima della pila
    - L'operando è sulla cima della pila
    - Quindi è un esempio di indirizzamento a registro indiretto



Mode	Algorithm
Immediate	Operand = A
Register Operand	LA = R
Displacement	LA = (SR) + A
Base	LA = (SR) + (B)
Base with Displacement	LA = (SR) + (B) + A
Scaled Index with Displacement	LA = (SR) + (I) × S + A
Base with Index and Displacement	LA = (SR) + (B) + (I) + A
Base with Scaled Index and Displacement	LA = (SR) + (I) × S + (B) + A
Relative	LA = (PC) + A

LA = linear address
(X) = contents of X
SR = segment register
PC = program counter
A = contents of an address field in the instruction
R = register
B = base register
I = index register
S = scaling factor

Le istruzioni, appunto, hanno un loro *formato*, quindi la struttura dei campi dell'istruzione. Include il codice operativo, in modo implicito o esplicito, uno o più operandi (avendo di solito più di un formato per un linguaggio macchina). La *lunghezza* delle istruzioni condiziona ed è condizionata da:

- Dimensione della M
- Organizzazione della M
- Struttura del bus
- Complessità della CPU
- Velocità della CPU

Si cerca quindi un compromesso tra repertorio di istruzioni potente e necessità di risparmiare spazio

Per quanto riguarda l'*allocazione* dei bit:

- Vari modi di indirizzamento
  - Vari numeri di operandi (di solito 1 o 2)
  - Registri verso M (di solito almeno 32 registri)
  - Banchi di registri (esempio: Pentium ha due banchi)
    - Due banchi da 8 registri ciascuno è solo 3 bit per indicare un registro (il codice operativo indica il banco)
  - Intervallo di indirizzi
  - Granularità degli indirizzi (es.: byte o parola)
    - L'indirizzamento di byte richiede più bit ma è utile (es. per manipolare caratteri)

Esempio del PDP-8, primo minicomputer commerciale della storia che abbia avuto successo:

Memory reference instructions

Opcode			D/I	Z/C	Displacement						
0		2	3	4	5						11

Input/output instructions

1	1	0	Device						Opcode		
0		2	3					8	9		11

Register reference instructions

Group 1 microinstructions

1	1	1	0	CLA	CLL	CMA	CML	RAR	RAL	BSW	IAC
0	1	2	3	4	5	6	7	8	9	10	11

Group 2 microinstructions

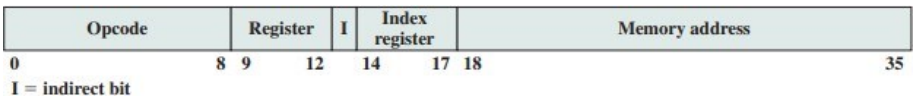
1	1	1	0	CLA	SMA	SZA	SNL	RSS	OSR	HLT	0
0	1	2	3	4	5	6	7	8	9	10	11

Group 3 microinstructions

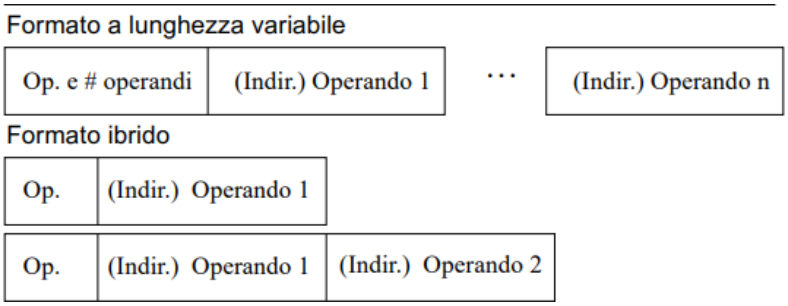
1	1	1	0	CLA	MQA	0	MQL	0	0	0	1
0	1	2	3	4	5	6	7	8	9	10	11

D/I = Direct/Indirect address
Z/C = Page 0 or Current page
CLA = Clear Accumulator
CLL = Clear Link
CMA = CoMplement Accumulator
CML = CoMplement Link
RAR = Rotate Accumulator Right
RAL = Rotate Accumulator Left
BSW = Byte SWap
IAC = Increment ACcumulator
SMA = Skip on Minus Accumulator
SZA = Skip on Zero Accumulator
SNL = Skip on Nonzero Link
RSS = Reverse Skip Sense
OSR = Or with Switch Register
HLT = HaLT
MQA = Multiplier Quotient into Accumulator
MQL = Multiplier Quotient Load

E similmente del PDP-10, versione successiva, condividendo la stessa lunghezza di parole a 36 bit e estendendo leggermente il set di istruzioni (ma con una migliore implementazione hardware). Alcuni aspetti del set di istruzioni sono unici, in particolare le istruzioni “byte”, che hanno operato su campi di bit di qualsiasi dimensione da 1 a 36 bit inclusi secondo la definizione più generale di un byte come una sequenza contigua di un numero fisso di bit.

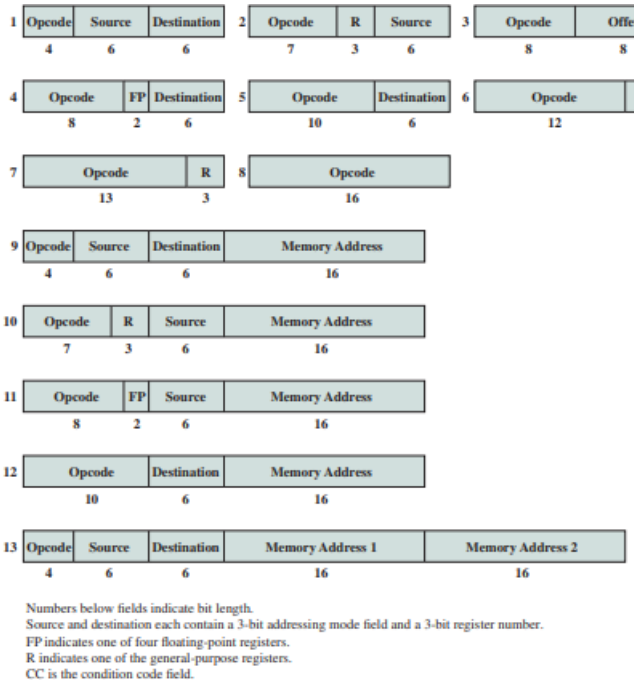


- Lunghezza variabile, che funziona con una serie di indirizzi per i singoli operandi. Questa è la base anche per il successivo sviluppo del formato ibrido:

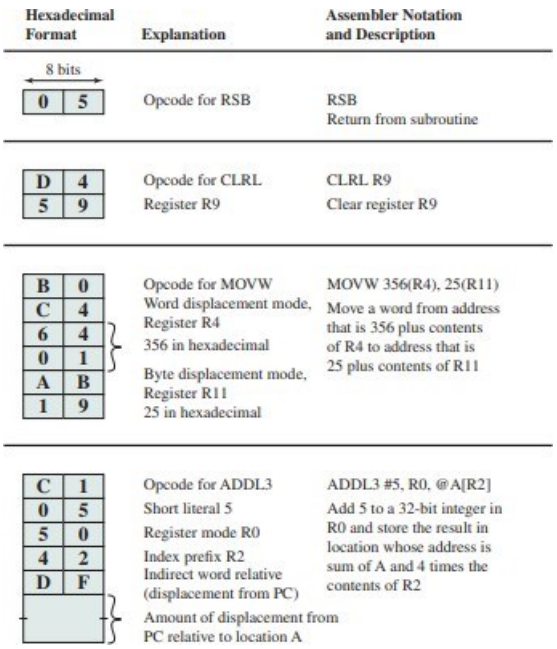


Similmente, ecco un formato che usa questa idea, quello del *PDP-11*

(Il PDP-11 includeva una serie di caratteristiche innovative nel suo set di istruzioni e registri generici aggiuntivi che lo rendevano molto più facile da programmare rispetto ai modelli precedenti della serie PDP. Inoltre, l'innovativo sistema Unibus consentiva di interfacciare facilmente dispositivi esterni al sistema utilizzando l'accesso diretto alla memoria, aprendo il sistema a un'ampia gamma di periferiche. Il PDP-11 sostituì il PDP-8 in molte applicazioni di calcolo in tempo reale, anche se le due linee di prodotti vissero in parallelo per oltre 10 anni. La facilità di programmazione del PDP-11 lo rese molto popolare anche per usi informatici generici.
Il design del PDP-11 ha ispirato la progettazione di microprocessori della fine degli anni '70, tra cui l'Intel x86[1] e il Motorola 68000)

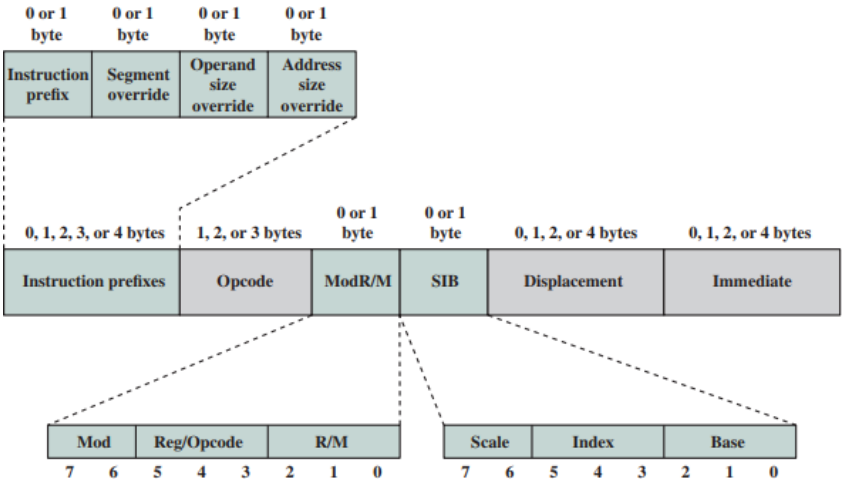


E del *VAX*
("VAX" era originariamente un acronimo per l'espressione in lingua inglese Virtual Address eXtension, in quanto VAX era visto come un'estensione a 32 bit della precedente architettura a 16 bit del PDP-11; le prime versioni di VAX implementavano una modalità di funzionamento "compatibile" che emulava molte delle istruzioni del PDP-11. Le versioni successive abbandonarono questa modalità e anche alcune delle istruzioni CISC meno utilizzate, a favore di un maggiore uso del microcodice o dell'emulazione da parte del software del sistema operativo.)



Similmente, il formato delle istruzioni x86:
(Un'istruzione x86-64 può essere lunga al massimo 15 byte. È costituita dai seguenti componenti nell'ordine indicato, dove i prefissi sono all'indirizzo meno significativo (più basso) in memoria:

- Prefissi legacy (1-4 byte, facoltativi)
- Opcode con prefissi (1-4 byte, obbligatorio)
- ModR/M (1 byte, se richiesto)
- SIB (1 byte, se richiesto)
- Spostamento (1, 2, 4 o 8 byte, se richiesto)
- Immediato (1, 2, 4 o 8 byte, se richiesto)

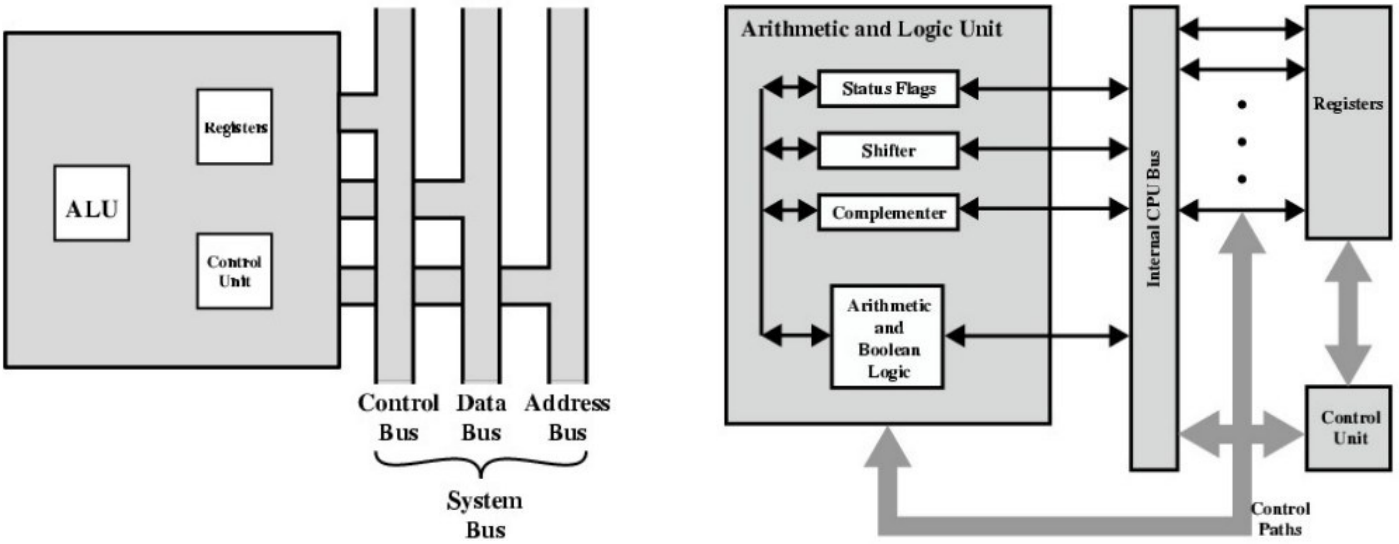


Struttura e funzione del processore

Compiti CPU:

- Prelevare istruzioni
- Interpretare istruzioni
- Prelevare dati
- Elaborare dati
- Scrivere (memorizzare) dati

Come abbiamo già visto, collabora con i bus di sistema (sx) con relativa struttura interna (dx):



Il livello più alto della gerarchia di memoria sono i registri, spazio per la CPU di memorizzazione dati e con un numero di funzioni svolte in base all'impianto progettuale della stessa CPU.
Sono di due tipi:

- Registri utente
  - usati dal "programmatore" per memorizzare internamente alla CPU i dati da elaborare
  - Usati per la memorizzazione di dati/indirizzi/codici di condizione
- Registri di controllo e di stato
  - usati dall'unità di controllo per monitorare le operazioni della CPU
  - usati dai programmi del Sistema Operativo (SO) per controllare l'esecuzione dei programmi

Normalmente, il "programmatore" è l'umano che programma codice che viene interpretato sotto forma di assembler e linker per istruzioni mnemoniche per la macchina e scritte in un linguaggio ad alto livello (C, C++, Java).

Parliamo anche dei registri ad uso generale, dedicati a varie funzioni ma anche per funzioni specifiche. Similmente, possono essere usati per contenere dati (es. accumulatore) ed indirizzi (es. indirizzo base di un segmento di memoria).
La memoria principale, infatti, può essere organizzata come insieme di segmenti o spazi di indirizzamento multipli:
– "visibili" al "programmatore", che riferisce logicamente una locazione di memoria riferendo il segmento e la posizione della locazione all'interno del segmento:
es. segmento 4, locazione 1024
– come supporto a questa "visione" della memoria, occorre poter indicare dove, all'interno della memoria fisica, inizia il segmento (base) e la sua lunghezza (limite) es. il segmento 4 ha base = 00EF9445hex e limite = 4MB
– quindi occorrono dei registri dove memorizzare tali informazioni

Scritto da Gabriel

I registri, infatti, sono:

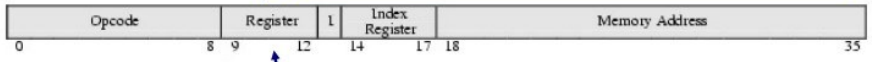
- Registri veramente ad uso generale
  - Aumentano la flessibilità e le opzioni disponibili al programmatore “a basso livello”
  - Aumentano la dimensione dell’istruzione e la sua complessità (perché ?)
- Registri specializzati
  - Istruzioni più piccole e più veloci
  - Meno flessibili

Perché aumenta dimensione e complessità?

– Facciamo l’esempio di istruzioni a formato fisso:

PDP-10

Campo non necessario se non avessi registri generali: *formato semplificato*



4 bit ➡ $2^4 = 16$ registri generali
Se si avessero 32 registri generali sarebbero necessari 5 bit di indirizzamento

Formato più lungo! 37 bit ⇦

Quanti registri generali?

– Tipicamente tra 8 e 32
– Meno di 8 = più riferimenti (accessi) alla memoria principale (perché ?)
– Più di 32 non riducono i riferimenti alla memoria ed occupano molto spazio nella CPU

Perché più accessi?

ESEMPIO: supponiamo di dover calcolare:

```
mem[4] = mem[0]+mem[1]+mem[2]+mem[3]
mem[5] = mem[0]*mem[1]*mem[2]*mem[3]
mem[6] = mem[5]-mem[4]
```

```
 4 registri
R0 ← mem[0];
R1 ← mem[1];
R2 ← mem[2];
R3 ← mem[3];
R0 ← R0+R1;
R0 ← R0+R2;
R0 ← R0+R3;
R1 ← R1*R2;
R1 ← R1*R3;
R2 ← mem[0];
R1 ← R1*R2;
R0 ← R1-R0;
```

7 op
5 mem

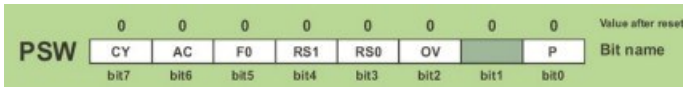
```
 6 registri
R0 ← mem[0];
R1 ← mem[1];
R2 ← mem[2];
R3 ← mem[3];
R4 ← R0+R1;
R4 ← R2+R4;
R4 ← R3+R4;
R5 ← R0*R1;
R5 ← R2*R5;
R5 ← R3*R5;
R0 ← R5-R4;
```

7 op
4 mem

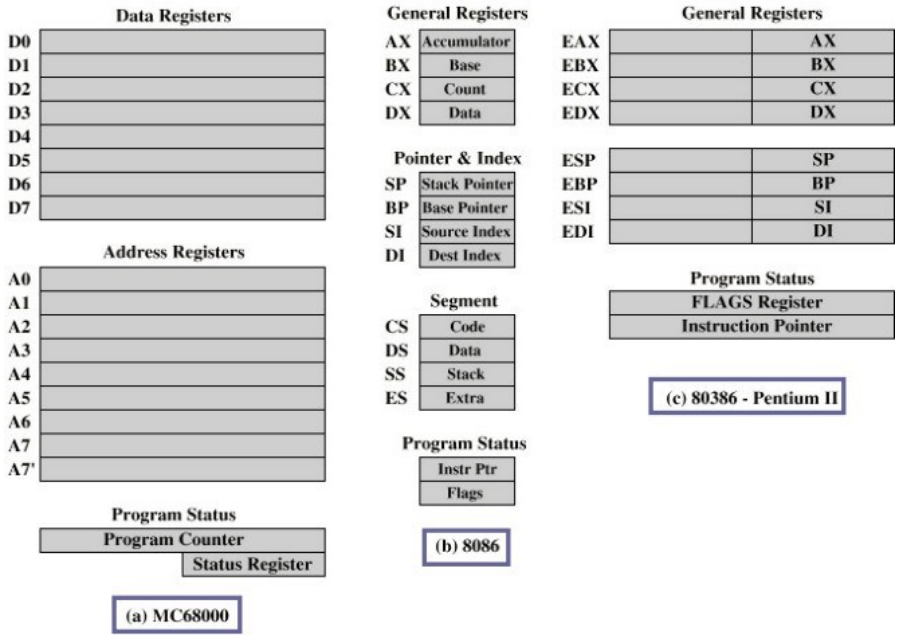
Quanto lunghi (in bit)?
- Abbastanza grandi da contenere un indirizzo della memoria principale
- Abbastanza grandi da contenere una “full word”
- è spesso possibile combinare due registri dati in modo da ottenerne uno di dimensione doppia
- Es.: programmazione in C
    - double int a;
    - long int a;

Similmente, esistono registri per la memorizzazione di codici di condizione, avendo insiemi di bit individuali (es. Il risultato dell’ultima operazione era zero). Possono essere letti (implicitamente) da programma (es. “Jump if zero” (salta se zero)). Non possono (tipicamente) essere impostati da programma.

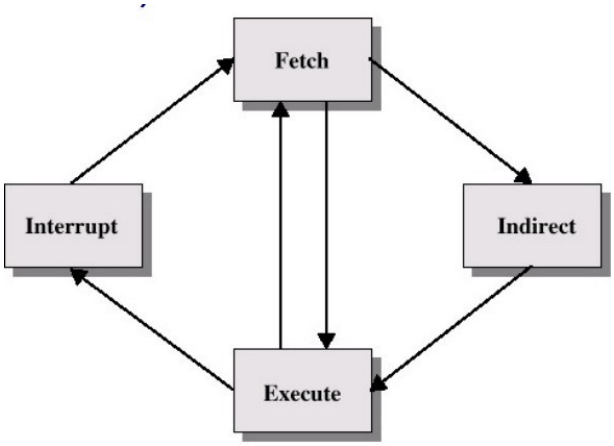
Questo insieme di codici di condizioni esiste nel *Program Status Word*, costituito da un insieme di bit per codici di condizione (segno dell’ultimo risultato, zero, riporto/carry, uguale, overflow, gestione interrupt, supervisore, ecc.)

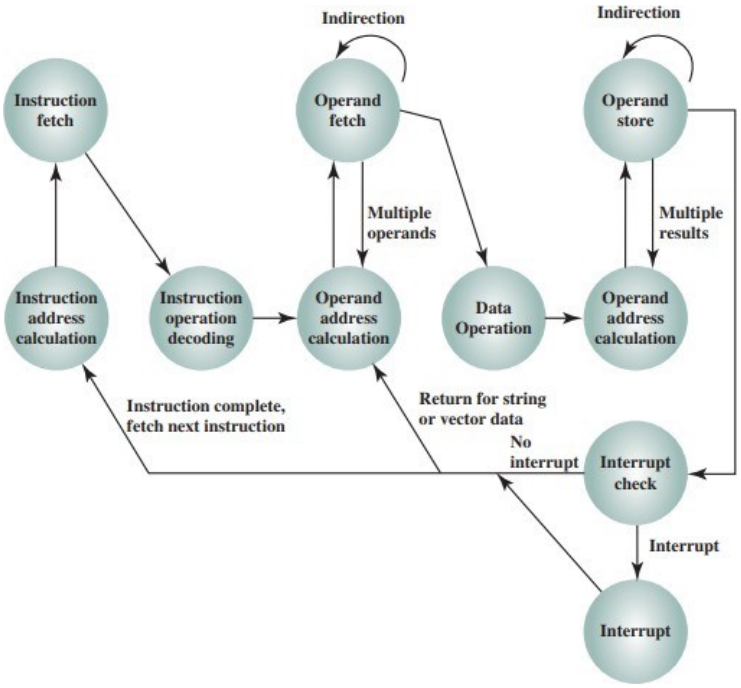


Quindi, per eseguire istruzioni privilegiate e agendo direttamente sul Kernel, quindi attuando modifiche a componenti critiche del sistema, si attiva la cosiddetta Superuser Mode/Supervisor Mode, come superutente, non disponibile al normale utente.
Ci possono essere registri che puntano a tabella delle pagine della memoria virtuale, blocchi di controllo dei processi, ecc.
Un esempio di organizzazione dei registri:



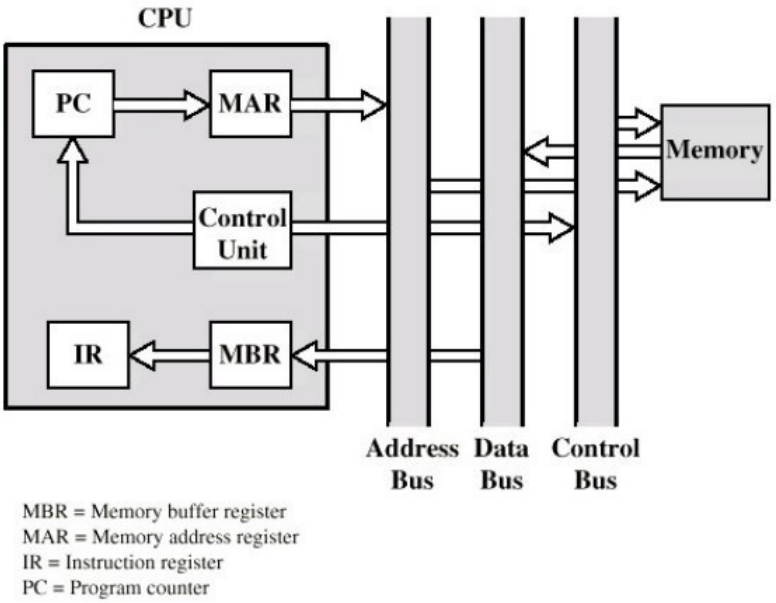
Recuperiamo il funzionamento del ciclo esecutivo (Fetch/Execute) delle istruzioni (già visto precedentemente) in una versione revisionata.
Per recuperare gli operandi di una istruzione può essere necessario accedere alla memoria. La modalità di indirizzamento indiretto per specificare la locazione in memoria degli operandi richiede più accessi in memoria. L'indirettezza si può considerare come un sottociclo del ciclo fetch/execute.





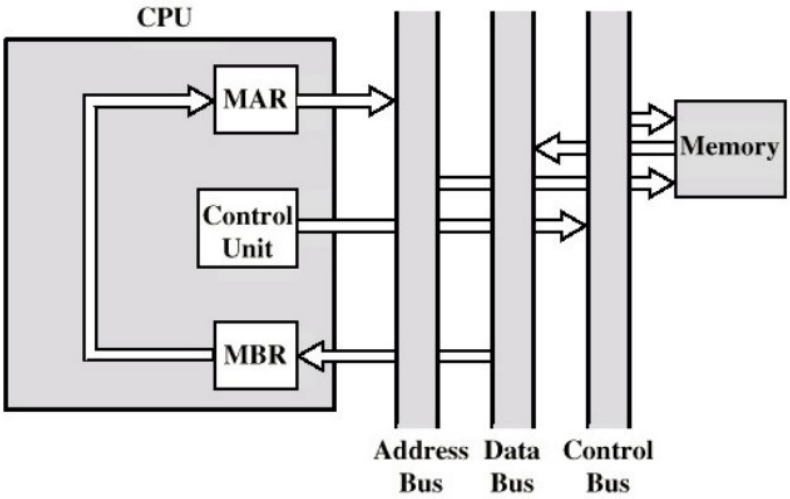
Il flusso dei dati, in generale, dipende dall'architettura della CPU e in generale:

- Fetch
  - PC contiene l'indirizzo della istruzione successiva
  - Tale indirizzo viene spostato in MAR
  - L'indirizzo viene emesso sul bus degli indirizzi
  - La unità di controllo richiede una lettura in memoria principale
  - Il risultato della lettura in memoria principale viene inviato nel bus dati, copiato in MBR, ed infine in IR
  - Contemporaneamente il PC viene incrementato



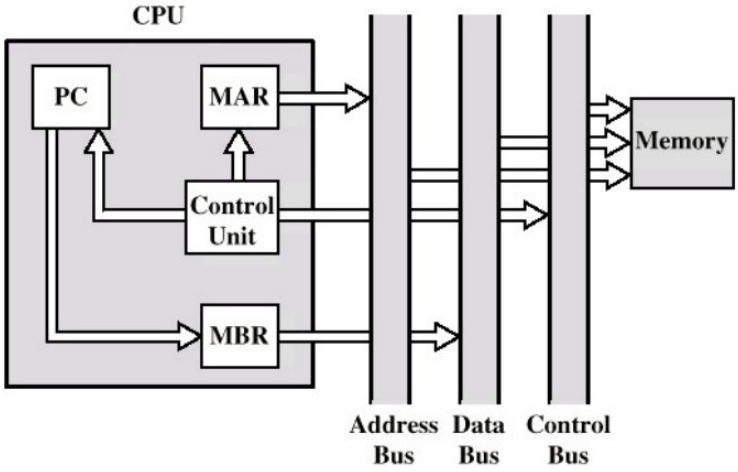
Il flusso dei dati (Data Fetch) esegue come operazioni questa serie:

- IR è esaminato
- Se il codice operativo della istruzione richiede un indirizzamento indiretto, si esegue il ciclo di indirettezza
    - gli N bit più a destra di MBR vengono trasferiti nel MAR
    - L'unità di controllo richiede la lettura dalla memoria principale
    - Il risultato della lettura (indirizzo dell'operando) viene trasferito in MBR



L'esecuzione, invece, assume molte forme e dipende dall'istruzione da eseguire.
Può includere:

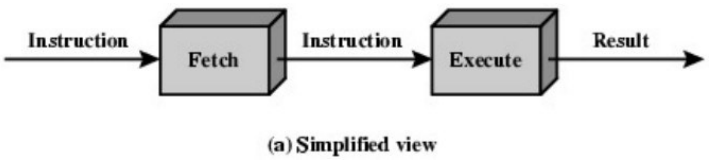
- lettura/scrittura della memoria
- Input/Output
- Trasferimento di dati fra registri e/o in registri
- Operazioni della ALU



L'interruzione invece è semplice e prevedibile:

- Contenuto corrente del PC deve essere salvato per permettere il ripristino della esecuzione dopo la gestione dell'interruzione
    - Contenuto PC copiato in MBR
    - Indirizzo di locazione di memoria speciale (es. stack pointer) caricato in MAR
    - Contenuto di MBR scritto in memoria
- PC caricato con l'indirizzo della prima istruzione della routine di gestione della interruzione
- Fetch della istruzione puntata da PC

La fase di *prelievo* della istruzione accede alla memoria principale mentre la fase di esecuzione di solito non lo fa. Si può prelevare l'istruzione successiva durante l'esecuzione della istruzione corrente. Questa operazione si chiama "instruction prefetch".



(a) Simplified view

Il prefetch non raddoppia le prestazioni:

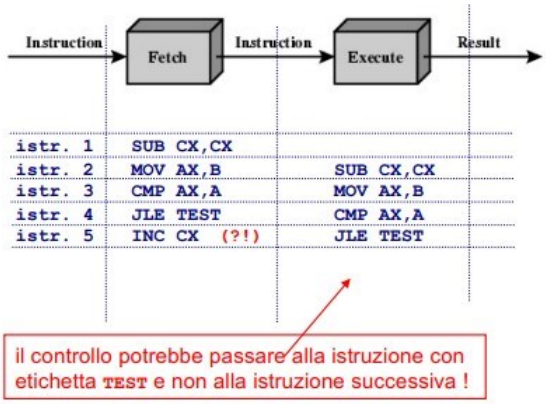
- L'esecuzione di istruzioni jump o branch possono rendere vano il prefetch
- La fase di prelievo è tipicamente più breve della fase di esecuzione
- Occorre aggiungere più fasi per migliorare le prestazioni
- Può essere inutile perché:

Per il seguente costrutto

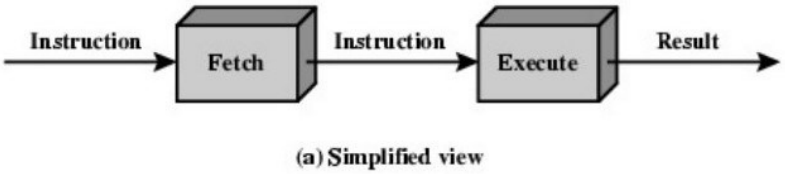
```
if (A > B) then
```

un compilatore potrebbe generare il seguente codice 80x86

```
      SUB CX,CX; CX ← 0
      MOV AX,B ; AX ← mem[B]
      CMP AX,A ; paragona [AX] con mem[A]
      JLE TEST ; salta se A ≤ B
      INC CX   ; CX ← CX+1
TEST  JCXZ OUT ; salta se [CX] è 0
THEN

OUT
```

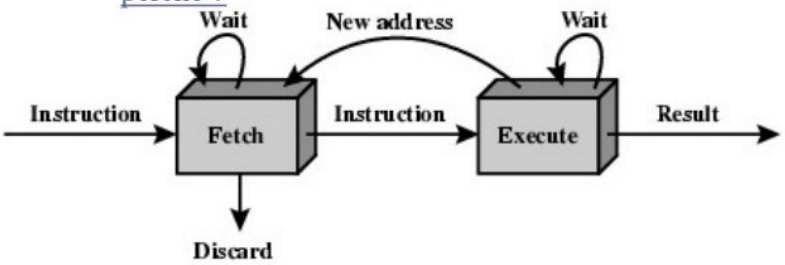


… si deve caricare una istruzione diversa dalla successiva !



(a) Simplified view

perché ?

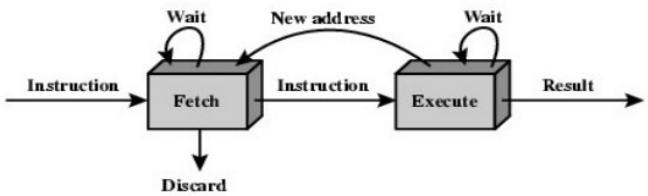


(b) Expanded view

perché…

su processore 286

	istruzione	cicli di clock impiegati dall'istr.
istr. 1	SUB CX,CX	2
istr. 2	MOV AX,B	5
istr. 3	CMP AX,A	6
istr. 4	JLE TEST	3 se non salta,>7 altrimenti
istr. 5	INC CX	2
istr. 6	JCXZ	4 se non salta,>8 altrimenti



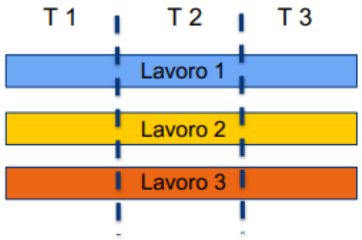
(b) Expanded view

Pipeline

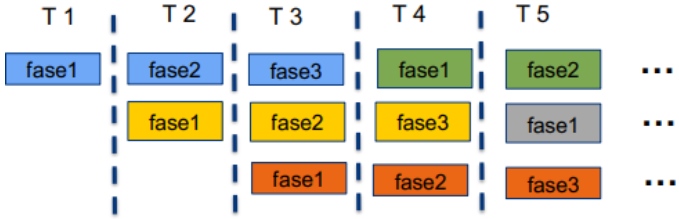
Il principio della *pipeline* permette di eseguire più attività contemporaneamente, completando il lavoro in meno tempo.

in 3 unità di Tempo
• 1 esecutore termina 1 solo lavoro
• 3 esecutori terminano 3 lavori

In generale, quindi otteniamo un parallelismo totale, eseguendo tante attività allo stesso tempo.
Tuttavia, possono esistere dipendenze funzionali tra le singole attività.

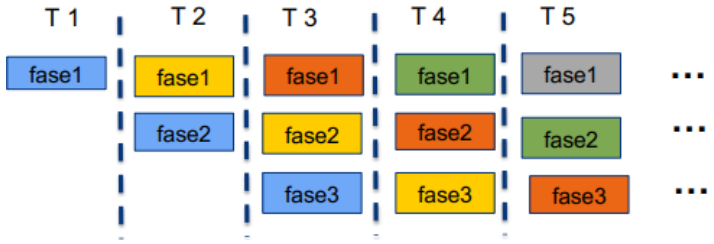


Dipendenza funzionale tra lavori successivi
• ogni lavoro è diviso in 3 fasi successive
• la fase 1 di Lavoro i deve essere eseguita dopo la fase 1 del precedente Lavoro i-1

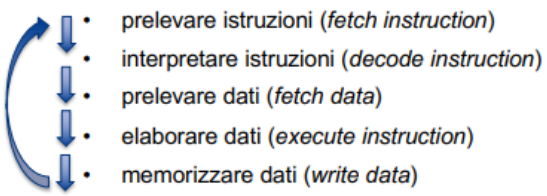


Esecutori generici
• ognuno esegue un lavoro completo
• a regime ha lo stesso throughput del parallelismo totale
• ognuno ha le risorse necessarie per ogni fase: sistema totalmente replicato

Esecutori specializzati
• ogni esecutore svolge sempre la stessa fase di ognuno dei lavori
• ogni esecutore ha solo le risorse per eseguire quella fase
• ogni lavoro passa da un esecutore all'altro
• a regime ha lo stesso throughput del parallelismo totale, ma usando meno risorse



Quindi:
• si decompone un lavoro in fasi successive
  - un prodotto deve passare una fase dopo l'altra
  - ogni fase è realizzata da un diverso operatore
  - nello stesso istante
    • prodotti diversi sono in fasi diverse (parallelismo)
  - l'istante successivo
    • ogni fase ripete lo stesso lavoro sul prodotto successivo,
• ogni lavoro avanza alla fase successiva



– operatori/fasi diverse usano risorse diverse evitando conflitti (se possibile)
Scritto da Gabriel

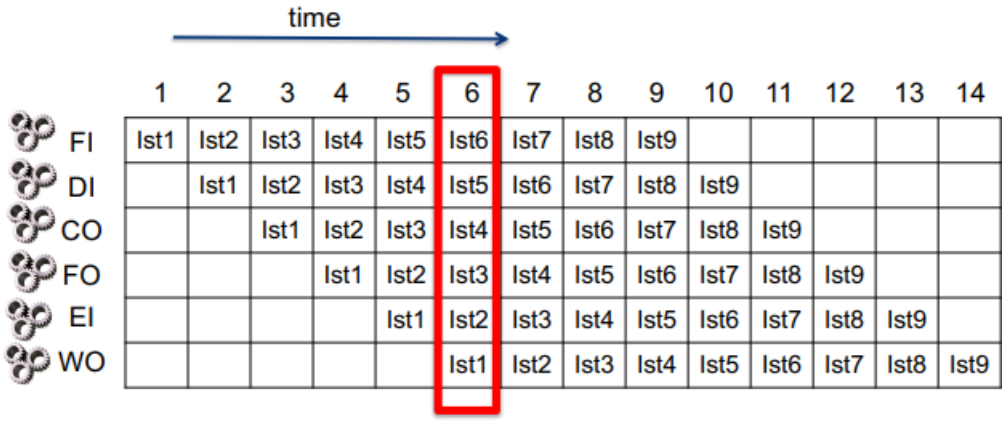
Nello stesso istante, istruzioni diverse sono in fasi diverse e, nell'istante successivo, ogni fase ripete lo stesso lavoro sull'istruzione avanza alla fase successiva.
Ogni fase è realizzata da una diversa unità funzionale della CPU ed operatori/fasi diverse usano risorse diverse evitando conflitti (se possibile). Tra due fasi successive si inseriscono dei buffer (registri) su cui si scrivono/leggono dati temporanei utili alla fase successiva.

Come detto, *il prefetch non raddoppia le prestazioni*:
• la fase di fetch è più breve, ma prima di poter iniziare il fetch successivo deve attendere che termini anche la fase di esecuzione
• se viene eseguito un jump o branch, la prossima istruzione da eseguire non è quella che è appena stata prelevata:

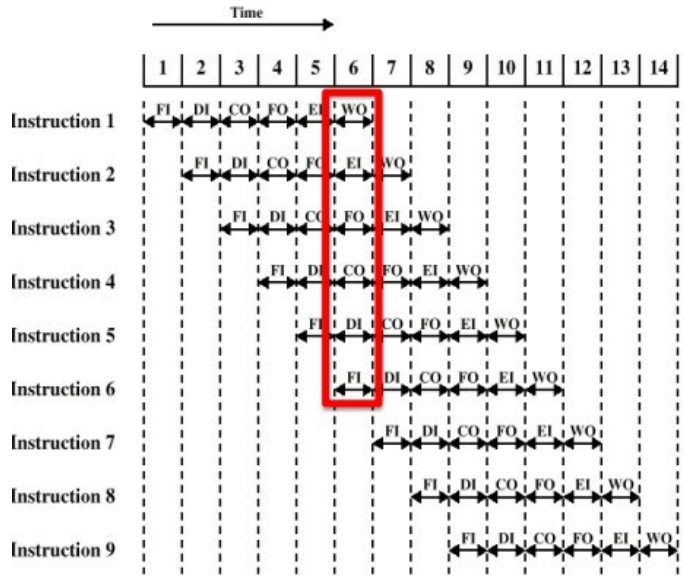
- la fase di fetch deve attendere che la fase execute le fornisca l'indirizzo a cui prelevare l'istruzione
- la successiva fase di execute deve attendere che sia prelevata l'istruzione, perche' quella pre-fetched non era valida

La suddivisione in fasi aggiunge overhead per spostare i dati nei buffer tra una fase e l'altra e per gestire il cambiamento di fase. Questo overhead potrebbe essere significativo quando:
– istruzioni successive dipendono logicamente da quelle precedenti,
– quando ci sono salti,
– quando ci sono conflitti negli accessi alla memoria/registri
La gestione logica e l'overhead aumentano con l'aumentare del numero di fasi della pipeline; occorre una progettazione accurata per ottenere risultati ottimali con una complessità ragionevole.
Per aumentare le prestazioni bisogna
• decomporre il lavoro in un maggior numero di fasi
• cercare di rendere le fasi più indipendenti e con una durata simile

L'evoluzione ideale della pipeline segue: in questo caso, esegue 9 istruzioni in 14 unità di tempo invece di 9x6=54.



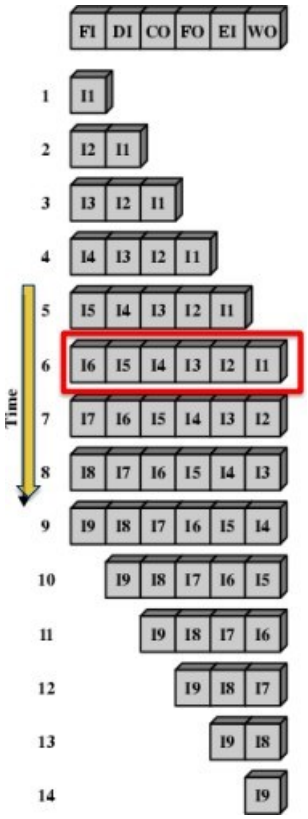
Si può vedere la temporizzazione a lato



esegue 9 istruzioni
in 14 unità di tempo
invece di 9x6=54

Assunzioni:

- ogni fase ha durata uguale
- ogni istruzione passa per tutte le fasi (e.g. LOAD non necessita WO)
- FI, FO, WO possono accedere alla memoria parallelamente senza fare conflitti
- non ci sono salti, né interrupt, né dipendenze



La performance della pipeline, quindi, viene così descritta:

Sia τ il **tempo di ciclo** di una pipeline

- cioè il tempo necessario per far **avanzare di uno stadio/fase** le istruzioni attraverso una pipeline
- può essere determinato come segue:

$$\tau = \max_i [\tau_i] + d = \tau_m + d \qquad 1 \le i \le k$$

k: numero di stadi nella pipeline

$\max_i [\tau_i]$: massimo ritardo di stadio (ritardo dello stadio più oneroso)

d: ritardo di commutazione di un registro, richiesto per l'avanzamento di segnali e dati da uno stadio al successivo

$\tau_m >> d$

Idealmente, le performance della pipeline:

Tempo totale richiesto da una pipeline con *k* stadi per eseguire *n* istruzioni
(approssimazione e assumendo no salti)

$$T_k = [k + (n-1)]\,\tau$$

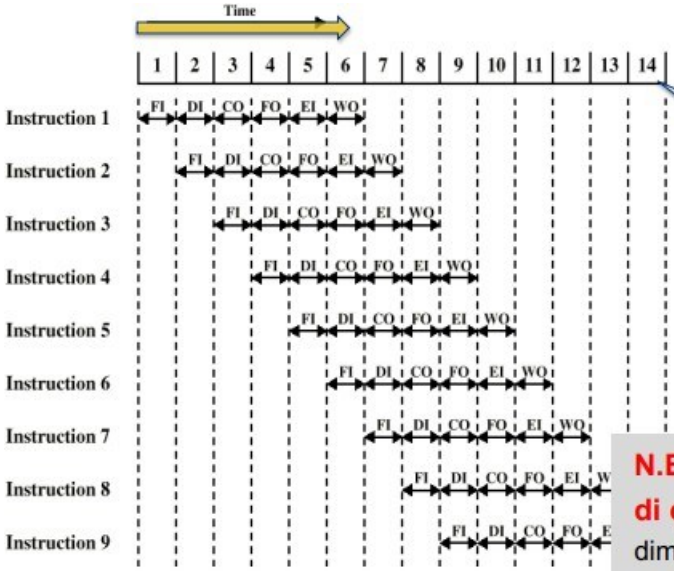
Infatti in k cicli si completa la prima istruzione
in altri n-1 cicli si completano le altre n-1 istruzioni (ogni istruzione finisce la sua pipeline 1 ciclo dopo la precedente)

Speedup (fattore di velocizzazione)

n istruzioni ***senza*** pipeline,
cioè 1 stadio di durata $k\,\tau$

$$S_k = \frac{T_1}{T_k} = \frac{nk\tau}{[k+(n-1)]\tau} = \frac{nk}{[k+(n-1)]}$$

Scritto da Gabriel



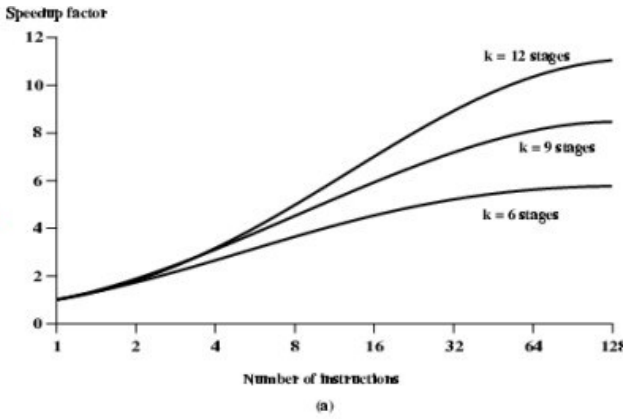
9 istruzioni
ognuna in 6 unità di tempo

14 = [6+ (9-1)]
invece di 6 * 9=54

N.B. non diminuisce il tempo di esecuzione di 1 istruzione!
diminuisce il tempo di esecuzione di **n** istruzioni:
aumenta il troughput

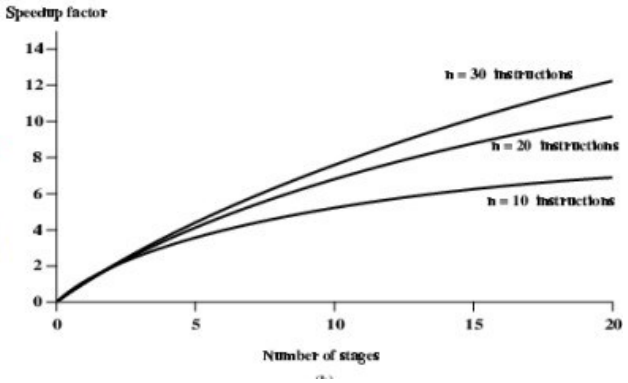
Speedup

Calcolato in funzione del numero di istruzioni



al crescere del numero di istruzioni l'incremento di velocità si avvicina al numero di stadi

Calcolato in funzione del numero di stadi



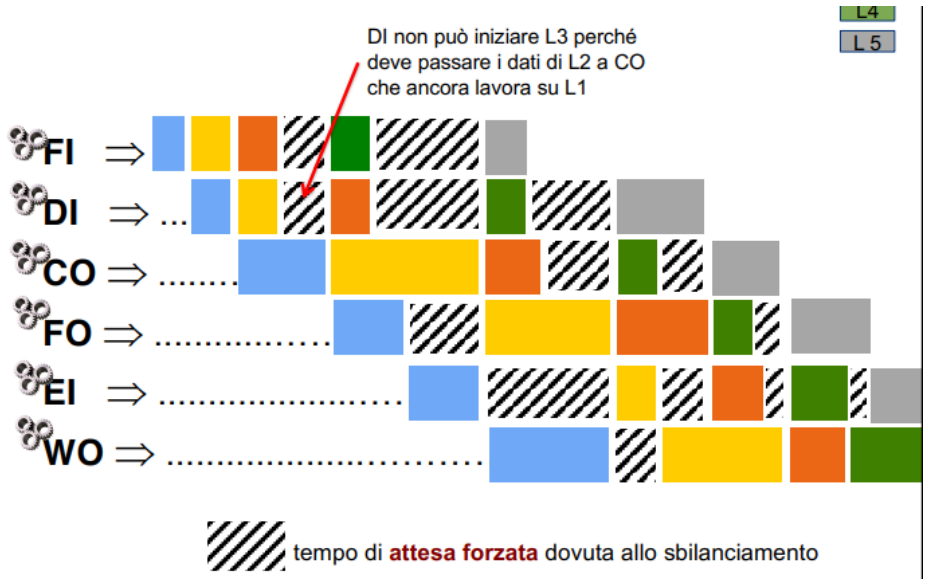
pipeline con più stadi aumentano il throughput, MA aggiungono overhead e criticità (es. con salti)

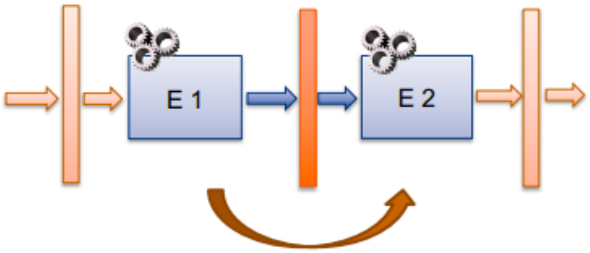
Ci sono varie situazioni in cui l'istruzione successiva non può essere eseguita nel ciclo di clock immediatamente successivo (stallo – pipeline bubble) non si raggiunge il parallelismo massimo, cosiddette pipeline hazards.

1. Sbilanciamento delle fasi

- Durate diverse per fase e per istruzione, e non tutte le fasi richiedono lo stesso tempo di esecuzione (es.: lettura di un operando tramite registro rispetto ad una mediante indirizzamento indiretto).

La suddivisione in fasi va fatta in base all'istruzione più onerosa e non tutte le istruzioni richiedono le stesse fasi e le stesse risorse.

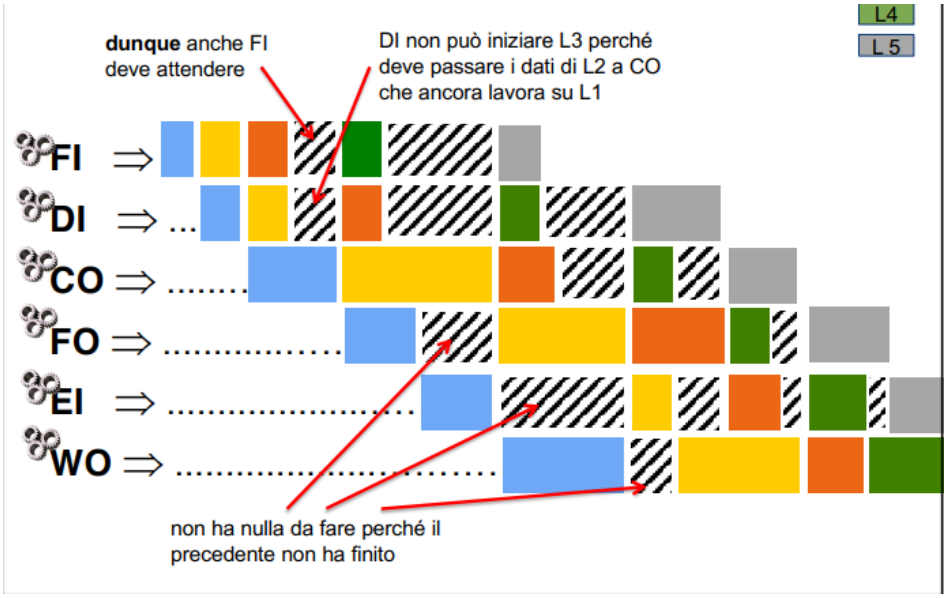




passare i dati significa che

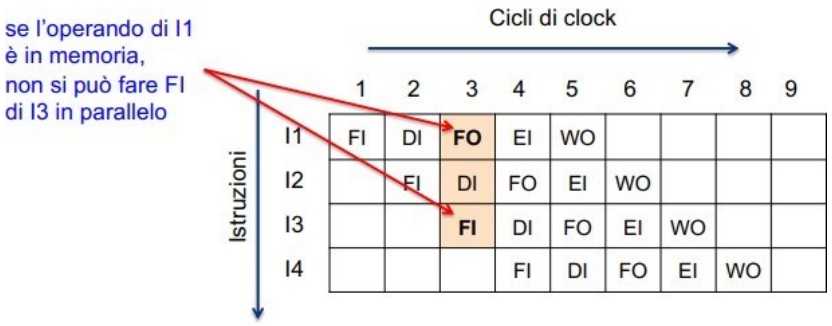
- l'esecutore E1 mentre lavora **scrive** sul registro intermendio
- l'esecutore E2 nel ciclo successivo **leggerà** questi dati

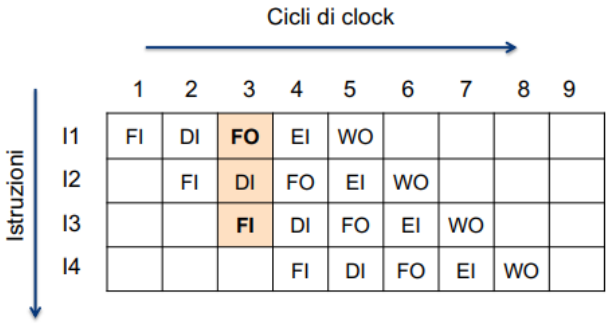
- se E1 comincia il lavoro successivo prima che anche E2 cominci il lavoro successivo, allora E1 può **sovrascrivere** i dati nel registro prima che E2 li abbia letti

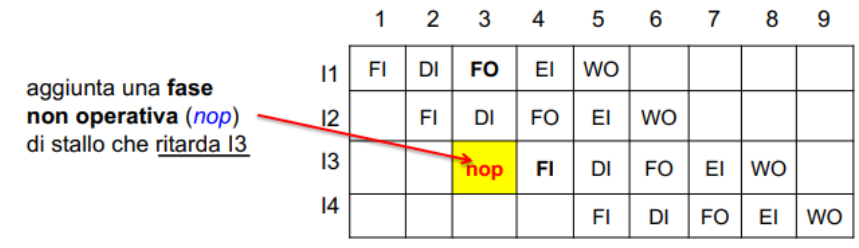


Possibili soluzioni:
• Decomporre fasi onerose in più sottofasi → Costo elevato e bassa utilizzazione
• Duplicare gli esecutori delle fasi più onerose e farli operare in parallelo → CPU moderne hanno una ALU in aritmetica intera ed una in aritmetica a virgola mobile

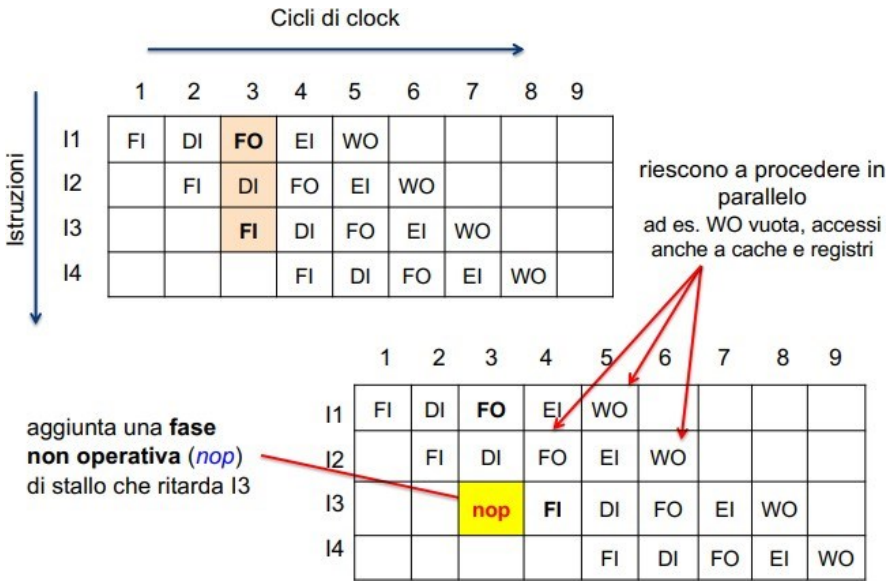
2. Problemi strutturali (structural hazards)
– Due fasi competono per usare la stessa risorsa, es. memoria in FI, FO, WO, dato che richiedono di accedere ad una stessa risorsa nello stesso ciclo di clock e quindi gli accessi devono avvenire in sequenza e non in parallelo.







Scritto da Gabriel



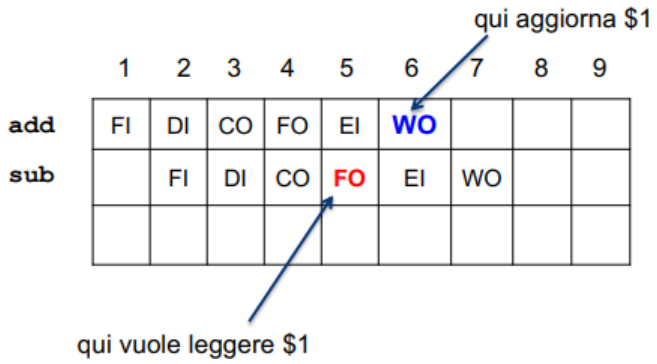
Possibili soluzioni:
• introdurre fasi non operative (nop)
• suddividere le memorie permettendo accessi paralleli: una memoria cache per le istruzioni e una per i dati

3. Dipendenza dai dati (data hazards)
– Un'istruzione dipende dal risultato di un'istruzione precedente ancora in pipeline. Una fase non può essere eseguita in un certo ciclo di clock perché i dati di cui ha bisogno non sono ancora disponibili e deve attendere che termini l'elaborazione di un'altra fase. Un dato modificato nell'esecuzione dell'istruzione corrente può dover essere utilizzato dalla fase FO dell'istruzione successiva.

```
add $1, $2, $3      R1 <- [R2] + [R3]
sub $4, $1, $5      R4 <- [R1] - [R5]
```

la seconda istruzione dipende dal risultato della prima, che si trova ancora all'interno della pipeline!

```
add $1, $2, $3      R1 <- [R2] + [R3]
sub $4, $1, $5      R4 <- [R1] - [R5]
```



```
add $1, $2, $3      R1 <- [R2] + [R3]
sub $4, $1, $5      R4 <- [R1] - [R5]
```

	1	2	3	4	5	6	7	8	9	10
add	FI	DI	CO	FO	EI	WO				
sub		FI	DI	CO	nop	nop	FO	EI	WO	

due cicli di stallo

```
add $1, $2, $3      R1 <- [R2] + [R3]
sub $4, $1, $5      R4 <- [R1] - [R5]
```

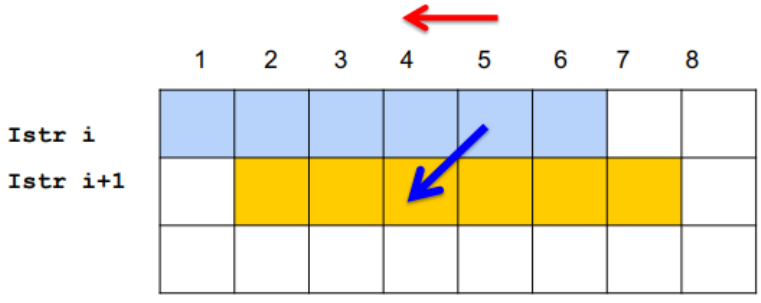
	1	2	3	4	5	6	7	8	9	10
add	FI	DI	CO	FO	EI	WO				
sub		FI	DI	CO	nop	nop	FO	EI	WO	
Istr3			FI	DI			CO	FO	EI	WO
Istr4				FI			DI	CO	FO	EI

due cicli di stallo per ***tutte*** le istruzioni

$istruzione\ i$
$istruzione\ i+1$

- Read after Write : “lettura dopo scrittura” (esempio di prima)
  - i+1 legge prima che i abbia scritto
- Write after Write : “scrittura dopo scrittura”
  - i+1 scrive prima che i abbia scritto
- Write after Read: “scrittura dopo lettura”
  - i+1 scrive prima che i abbia letto (caso raro in pipeline)

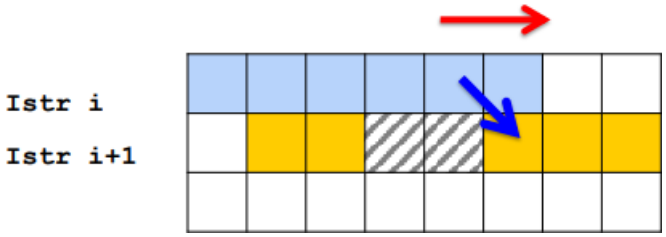
4. Dipendenza dal controllo (control hazards)
– Istruzioni che alterano la sequenzialità, es. salti (condizionati o no), chiamate e ritorni da procedure, interruzioni. L’istruzione successiva ha bisogno dei dati prima che la precedente li abbia prodotti. Dipende dall’architettura della pipeline: da come sono definiti i suoi stadi e come sono implementate le istruzioni



Scritto da Gabriel

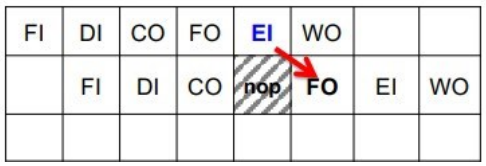
Possibili soluzioni:

1. Introduzione di fasi non operative (nop-stallo)



2. Propagazione in avanti del dato richiesto (data forwarding – bypassing)

1 solo ciclo di stallo



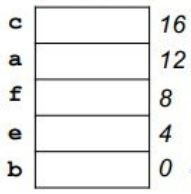
un circuito riconosce la dipendenza e **propaga in avanti** l'output della ALU

3. Riordino delle istruzioni

programma C con 5 variabili che si riferiscono a indirizzi di memoria

```
a = b + e;
c = b + f;
```

memoria indirizzata al byte (1 word=4 byte)



assumiamo corrisponda a ($t0) così usiamo offset

compilatore produce il codice assembler
- associando i registri alle variabili del programma
- e trasferendo i dati tra la memoria e i registri

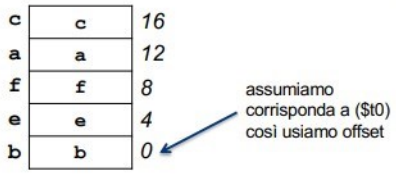
```
b - $1    e - $2    a - $3
f - $4    c -$5
```

```
lw  $1  0  ($t0)
lw  $2  4  ($t0)
add $3  $1  $2
sw  $3  12 ($t0)
lw  $4  8  ($t0)
add $5  $1  $4
sw  $5  16 ($t0)
```

programma C con 5 variabili
che si riferiscono a indirizzi di memoria

```
a = b + e;
c = b + f;
```

memoria indirizzata al byte (1 word=4 byte)




```
lw  $1  0  ($t0)
lw  $2  4  ($t0)
add $3  $1  $2
sw  $3  12 ($t0)
lw  $4  8  ($t0)
add $5  $1  $4
sw  $5  16 ($t0)
```

tutte dipendenze **Read after Write**

quindi servono degli **stalli**

(a seconda di come è definite la pipeline, qualche problema può risoversi con *data forwarding*)

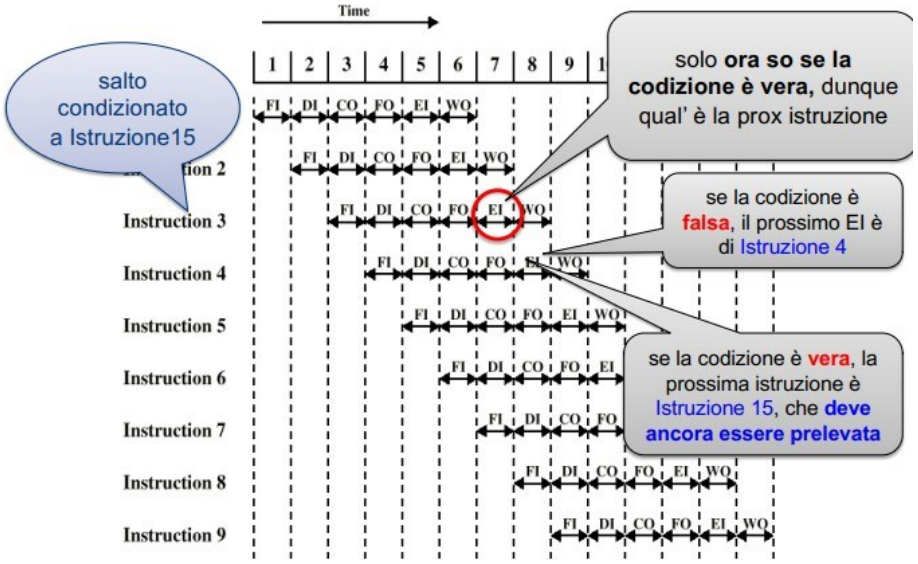
programma C con 5 variabili
che si riferiscono a indirizzi di memoria

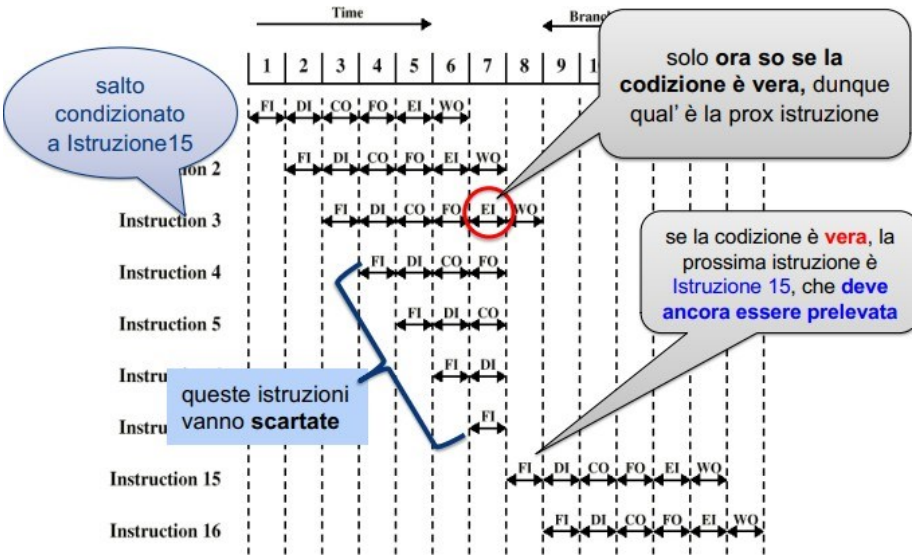
```
a = b + e;
c = b + f;
```

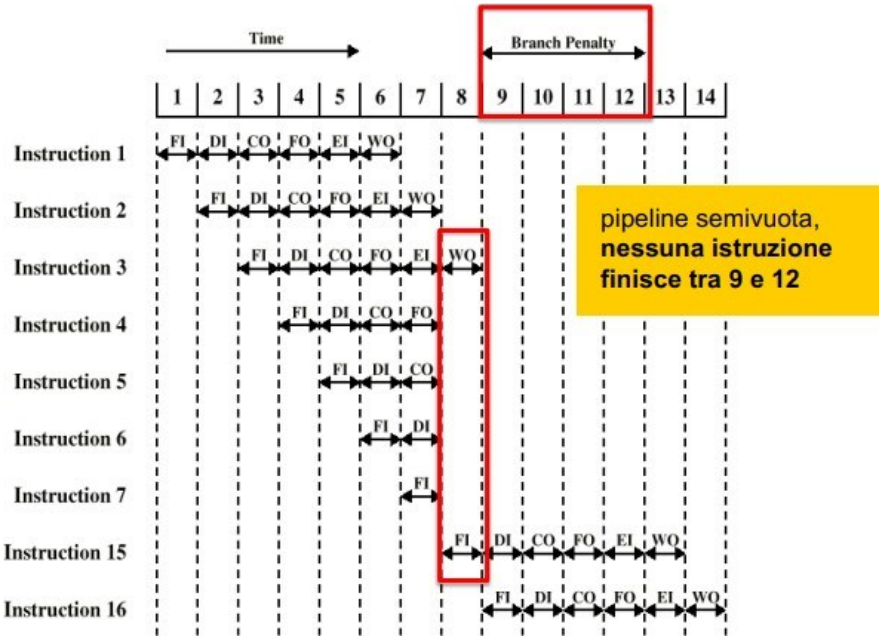
riordinando le istruzioni si sono "ridotte"
le dipendenze `lw - add`

```
lw  $1  0  ($t0)
lw  $2  4  ($t0)
add $3  $1  $2
sw  $3  12 ($t0)
lw  $4  8  ($t0)
add $5  $1  $4
sw  $5  16 ($t0)
```

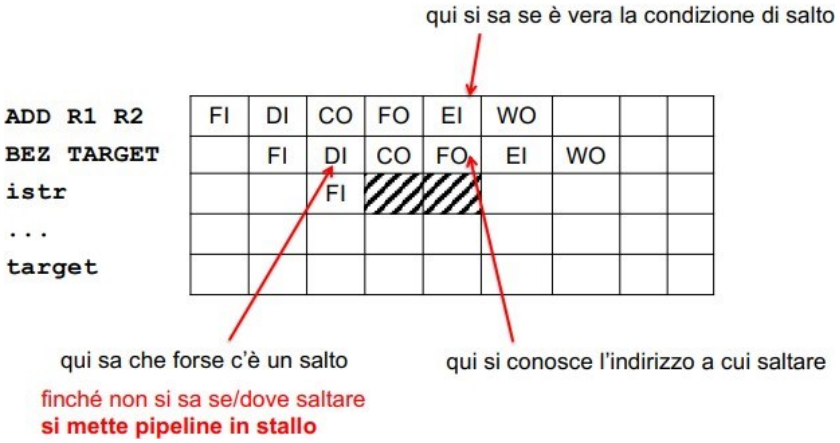
```
lw  $1  0  ($t0)
lw  $2  4  ($t0)
lw  $4  8  ($t0)
add $3  $1  $2
sw  $3  12 ($t0)
add $5  $1  $4
sw  $5  16 ($t0)
```

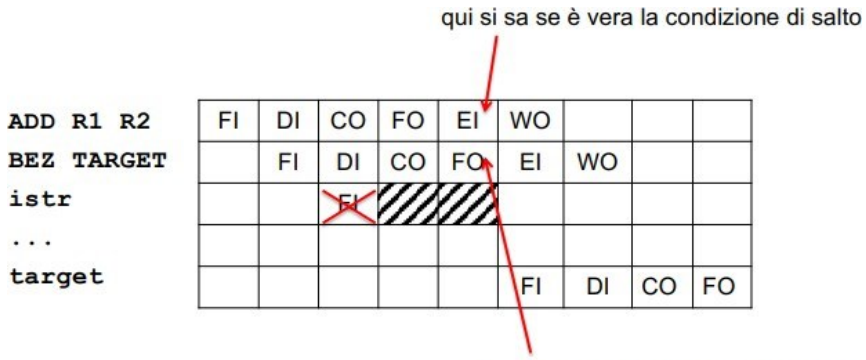




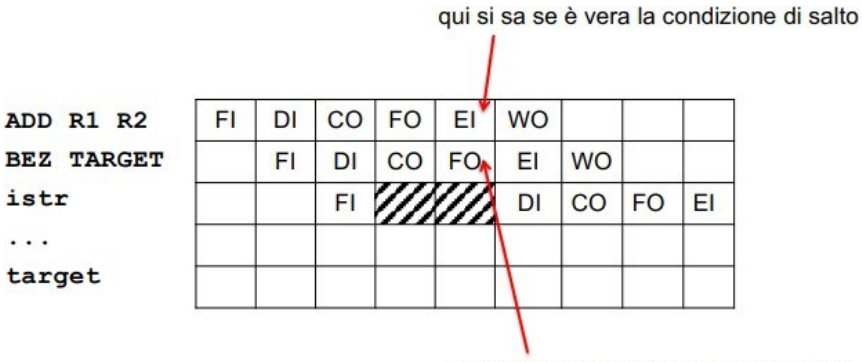


Esempio: salto condizionato





supponiamo **condizione vera**:
si scarta istruzione pre-fetched
e si ricomincia con istruzione `target`



supponiamo **condizione falsa**:
riprendo dopo lo stallo con
l'istruzione pre-fetched

supponiamo **condizione vera**:
si scarta istruzione pre-fetched
e si ricomincia con istruzione `target`

Esempio

salto incondizionato

qui si sa che ci sarà un salto

qui si conosce l'indirizzo a cui saltare

BR TARGET	FI	DI	CO	FO	EI	WO			
istr									
...									
target									
target+1									

salto incondizionato

qui si sa che ci sarà un salto

qui si conosce l'indirizzo a cui saltare

BR TARGET	FI	DI	CO	FO	EI	WO			
istr		~~FI~~							
...									
target									
target+1									

ormai ha prelevato l'istruzione errata,
quindi mette in stallo e scarta **istr**

salto incondizionato

qui si sa che ci sarà un salto

qui si conosce l'indirizzo a cui saltare

BR TARGET	FI	DI	CO	FO	EI	WO			
istr		~~FI~~							
...									
target					FI	DI	CO	EI	WO
target+1						FI	DI	CO	EI

ormai ha prelevato l'istruzione errata
quindi mette in stallo e scarta **istr**
ricomincia con istruzione **target**

Uno dei maggiori problemi della progettazione della pipeline è assicurare un flusso regolare di istruzioni
– violato da salti condizionati, salti non condizionati, chiamate e ritorni da procedure
– se la fase fetch ha caricato un'istruzione errata, va scartata
– queste istruzioni sono circa il 30% del totale medio di un programma
Possibili soluzioni:
• mettere in stallo la pipeline finché non si è calcolato l'indirizzo della prossima → istruzione semplice ma inefficiente
• individuare le istruzioni critiche e aggiungere un'apposita logica di controllo → si complica il compilatore e hardware specifico

Soluzioni per salti condizionati
1. flussi multipli (multiple streams)
– replica la prima parte della pipeline, EI esclusa, per entrambi i rami possibili

```
istr i ⟶ se cond
             istruzione n
         else
             istruzione i+1
```

inserisce nella pipeline sia `istruzione n` che `istruzione i+1`

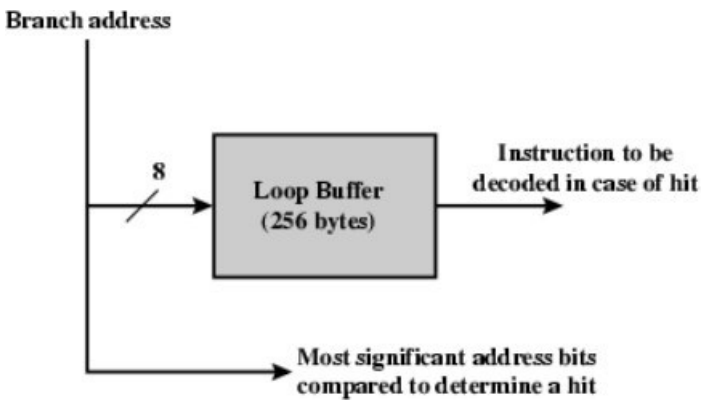
brute-force

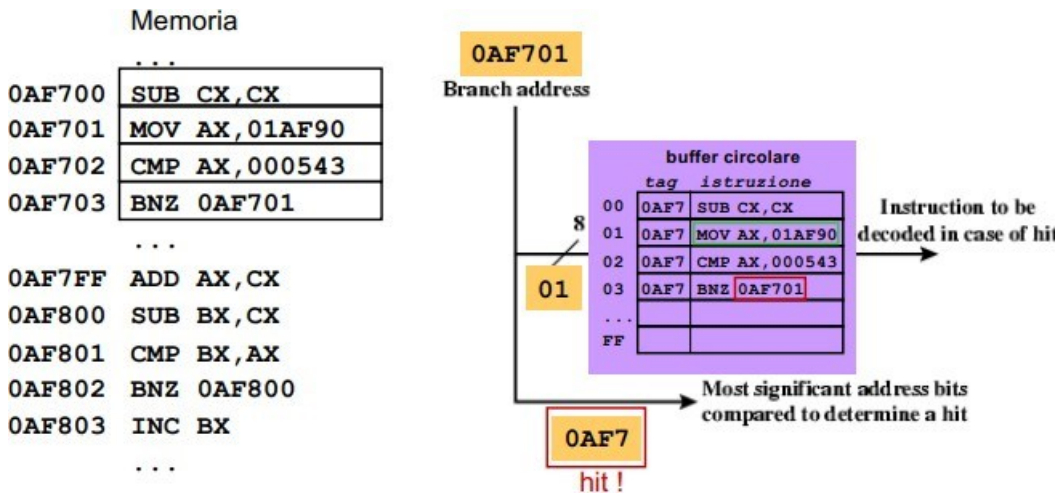
- conflitti di accesso alle risorse tra i due stream
- se istruzione n (o i+1) contiene un salto aggiunge ulteriori stream

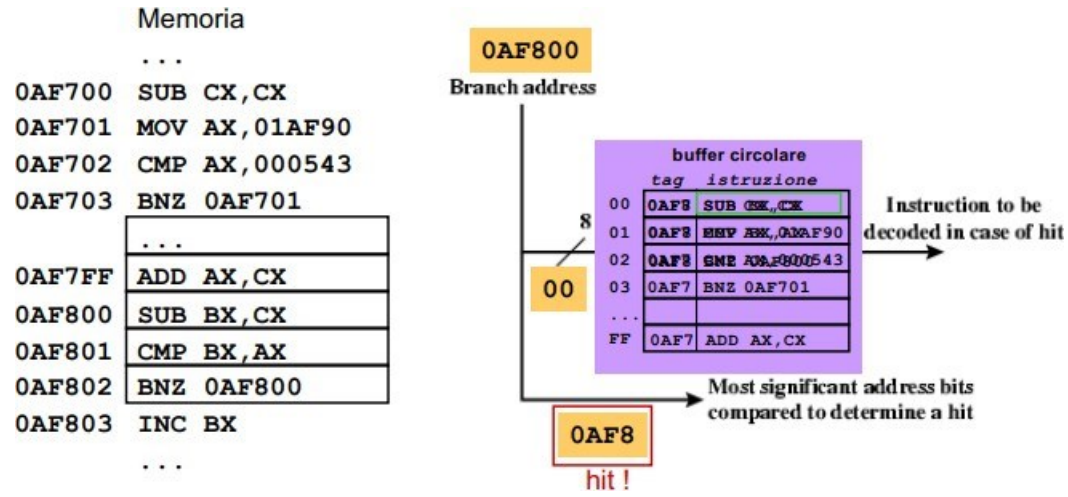
2. prefetch anche dell'istruzione target
– anticipa il fetch dell'istruzione target oltre a quella successiva al salto
– se il salto è preso, trova l'istruzione già caricata
– in ogni caso una parte della pipeline deve essere scartata

3. buffer circolare (loop buffer) → buffer senza prefetch, capienza 256 bytes, indirizzato a byte
– è una memoria piccola e molto veloce che mantiene le ultime n istruzioni prelevate
– in caso di salto l'hardware controlla se l'istruzione target è tra quelle già dentro il buffer, così da evitare il fetch
– utile in caso di loop, specie se il buffer contiene tutte le istruzioni nel loop, così vengono prelevate dalla memoria una sola volta
– può essere accoppiato al pre-fetch: riempio il buffer con un po' di istruzioni sequenzialmente successive alla corrente. Per molti if-then-else i due rami sono istruzioni vicine, quindi probabilmente entrambe già nel buffer

Dato l'indirizzo target di salto/branch, controllo se c'è nel buffer:
– gli 8 bit meno significativi sono usati come indice nel buffer
– gli altri bit più significativi si usano per controllare se la destinazione del salto sta già nel buffer







4. predizione dei salti – cerco di predire se il salto sarà intrapreso o no

Varie possibilità:

- previsione di saltare **sempre**
- previsone di non saltare **mai** *(molto usato)*
- previsione in base al codice operativo

} *approcci statici*

- bit *taken/not taken*
- tabella della storia dei salti

} *approcci dinamici*

Gli approcci dinamici di predizione cercano di migliorare la qualità della predizione sul salto memorizzando la storia delle istruzioni di salto condizionato di uno specifico programma. Ad ogni istruzione di salto condizionato associo 1 (o 2) bit per ricordare la storia recente dell'istruzione, i.e. se l'ultima (e la penultima) volta il salto è stato preso . I bit sono memorizzati in una locazione temporanea ad accesso molto veloce.

associo 1 bit ad ogni istruzione di salto

- ricorda come è andata l'ultima volta, predico di comportarsi nello stesso modo
- se bit è **1** predico di saltare
- se bit è **0** predico di non saltare
- se ho sbagliato predizione inverto il bit

esempio:

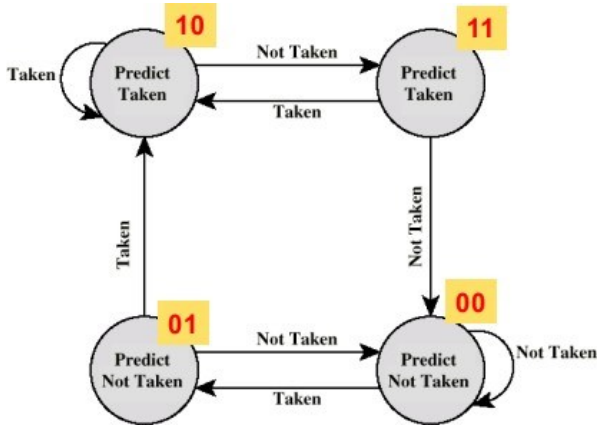
```
        ......
LOOP:   ......
        ......
        ......
        BNZ  LOOP
```

a regime:
2 errori per ciclo

- dopo la prima esecuzione del ciclo, **in uscita dal ciclo**, il bit assegnato a BNZ LOOP è **0** perché il salto **non è stato preso**
- quando si rientra nello stesso ciclo,
  - si avrà **un errore alla prima iterazione** (il bit era a 0, invece prendo il salto)
  - le successive predizioni saranno giuste (l'entrata ha portato il bit a 1)
  - quando **si esce dal ciclo si fa un ulteriore errore** di predizione (e si rimette il bit a 0)

Si ha una predizione dinamica:

1) con 2 bit, appunto usando un paio di bit per ricordare come è andata la predizione degli ultimi due salti. Per invertire la predizione ci vogliono 2 errori consecutivi e in questo modo a regime fa un solo errore per ciclo

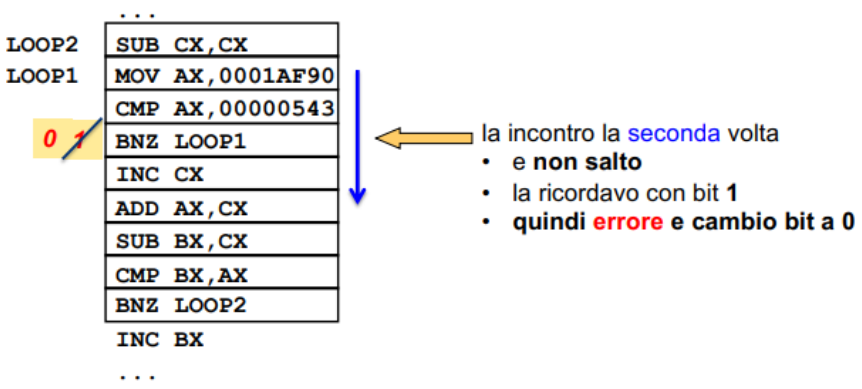


Per ogni istruzione di salto condizionato uso 1/2 bit
• per ricordare se l'ultima volta che ho eseguito quella stessa istruzione il salto è stato fatto o no
• se incontro di nuovo quell'istruzione e l'ultima volta aveva provocato il salto
• allora predico che salterà, quindi carico la pipeline con le istruzioni a partire dalla destinazione del salto
• se ho fatto la scelta sbagliata, le istruzioni caricate vengono eliminate

2) con 1 bit, due cicli innestati, supponiamo che per entrambi si iteri una sola volta

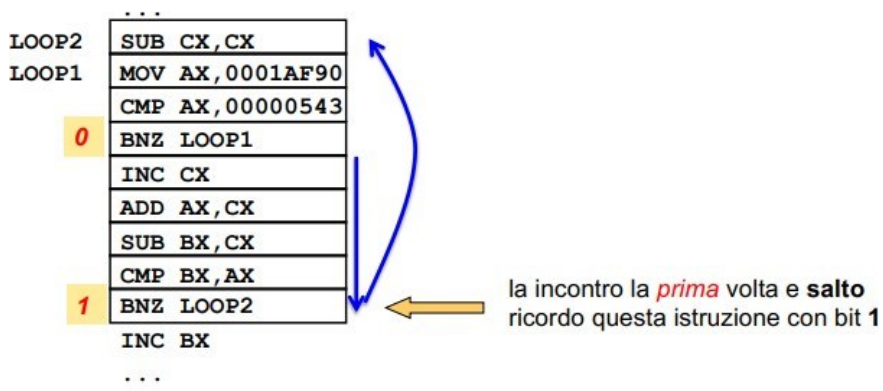
```
        ...
LOOP2   SUB  CX,CX
LOOP1   MOV  AX,0001AF90
        CMP  AX,00000543
        BNZ  LOOP1
        INC  CX
        ADD  AX,CX
        SUB  BX,CX
        CMP  BX,AX
        BNZ  LOOP2
        INC  BX
        ...
```

la incontro la prima volta e **salto,**
ricordo l'istruzione con bit **1**



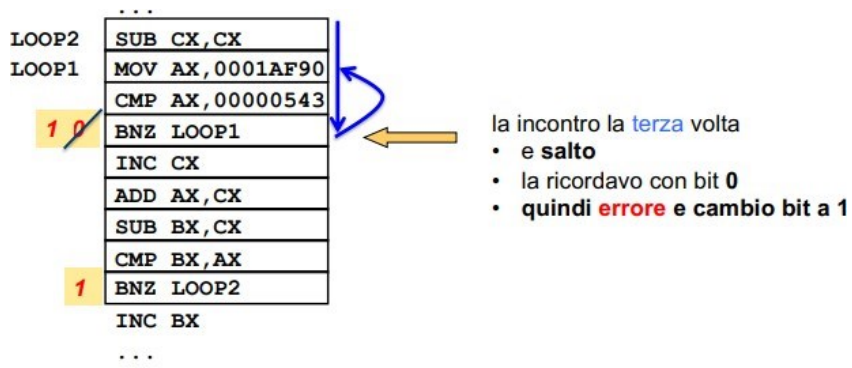
la incontro la seconda volta
- e **non salto**
- la ricordavo con bit **1**
- **quindi errore e cambio bit a 0**

errori totali = **1**



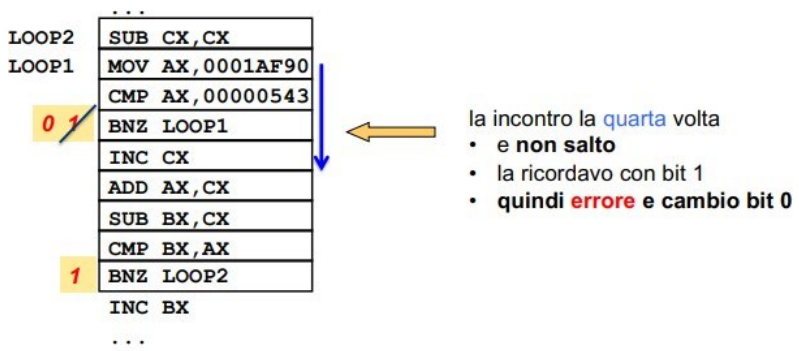
la incontro la *prima* volta e **salto**
ricordo questa istruzione con bit **1**

errori totali = **1**



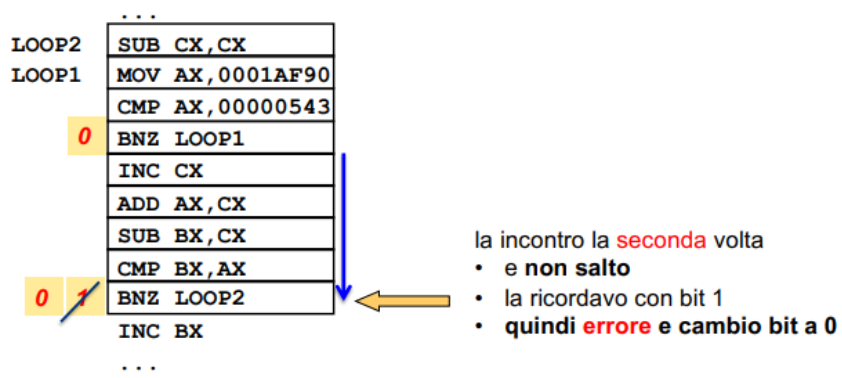
la incontro la terza volta
- e **salto**
- la ricordavo con bit **0**
- **quindi errore e cambio bit a 1**

errori totali = **2**



la incontro la quarta volta
- e **non salto**
- la ricordavo con bit 1
- **quindi errore e cambio bit 0**

errori totali = **3**

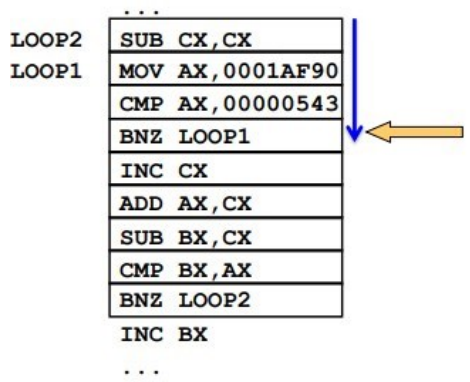


la incontro la seconda volta
- e **non salto**
- la ricordavo con bit 1
- **quindi errore e cambio bit a 0**

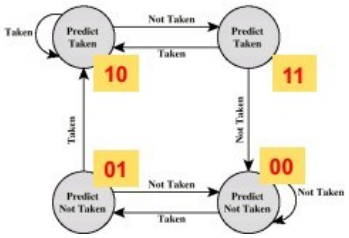
errori totali = **4**

Tornando alla predizione dinamica 2 bit:

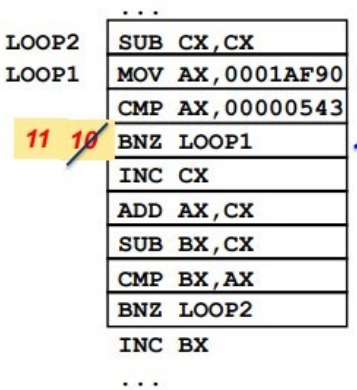
errori totali = 0



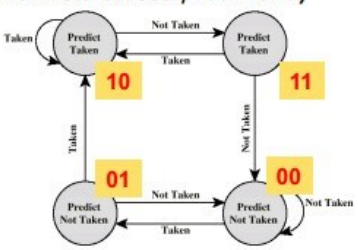
la incontro la prima volta e **salto**
la ricordo con bit **10**



errori totali = **1**



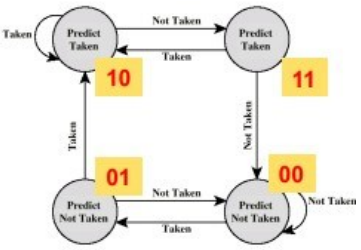
la incontro la <u>seconda</u> volta
- predice salto e **non salto**
- la ricordavo con bit **10**
- **quindi errore e cambio in 11**
  (*ma resta stessa predizione*)



errori totali = **1**

```
...
LOOP2   SUB CX,CX
LOOP1   MOV AX,0001AF90
        CMP AX,00000543
11      BNZ LOOP1
        INC CX
        ADD AX,CX
        SUB BX,CX
        CMP BX,AX
10      BNZ LOOP2
        INC BX
...
```

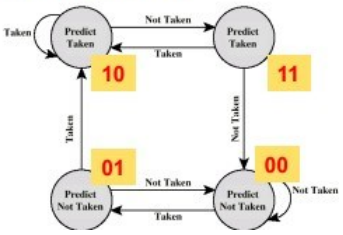
la incontro la prima volta e **salto**
la ricordo con bit **10**



errori totali = 1 non è errore in più

```
...
LOOP2   SUB CX,CX
LOOP1   MOV AX,0001AF90
        CMP AX,00000543
10 11   BNZ LOOP1
        INC CX
        ADD AX,CX
        SUB BX,CX
        CMP BX,AX
10      BNZ LOOP2
        INC BX
...
```

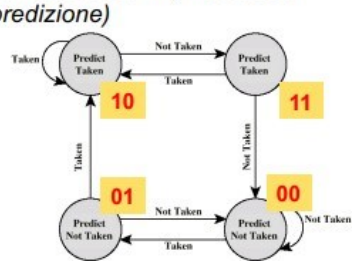
la incontro la terza volta
- predice salto e **salto**
- la ricordavo con bit **11**
- **quindi non errore di predizione**
- **ma cambio bit a 10**

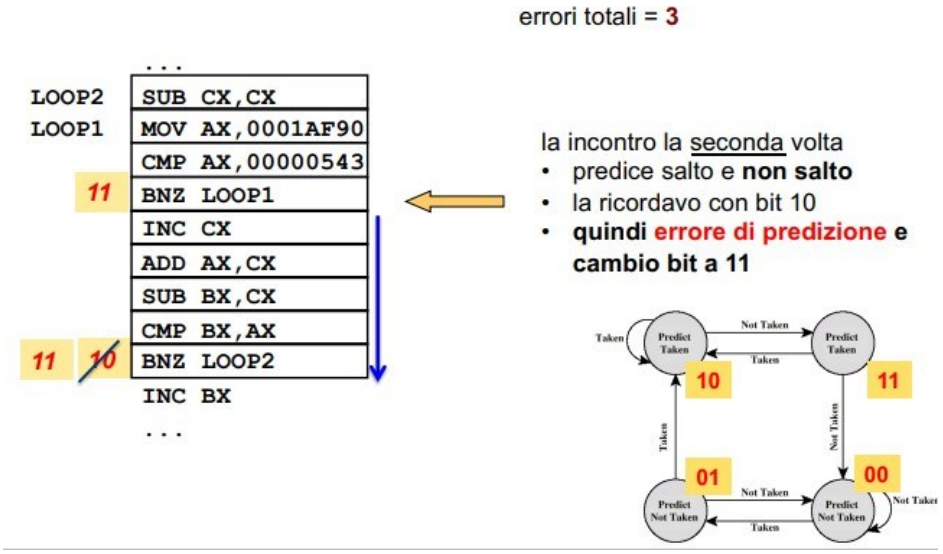


errori totali = **2**

```
...
LOOP2   SUB CX,CX
LOOP1   MOV AX,0001AF90
        CMP AX,00000543
11 10   BNZ LOOP1
        INC CX
        ADD AX,CX
        SUB BX,CX
        CMP BX,AX
10      BNZ LOOP2
        INC BX
...
```

la incontro la quarta volta
- predice salto e **non salto**
- la ricordavo con bit 10
- **quindi errore di predizione e cambio bit a 11** *(ma stessa predizione)*





La predizione dei dinamica usa il buffer di predizione dei salti/branch prediction buffer/branch history table) è una piccola memoria associata allo stadio fetch della pipeline
Ogni riga della tabella è costituita da 3 elementi:
1. indirizzo istruzione salto,
2. i bit di predizione
3. l'indirizzo destinazione del salto (o l'istruzione destinazione stessa), così quando la predizione è di saltare non devo attendere che si ri-decodifichi il target del salto (se la previsione è errata dovrò eliminare le istruzioni errate e caricare quelle corrette

Da questa predizione dei salti, dipendono due note vulnerabilità dei processori:
Spectre e *Meltdown* sono vulnerabilità di sicurezza estremamente pericolose che consentono a soggetti malintenzionati di aggirare le protezioni di sicurezza del sistema presenti in quasi tutti i dispositivi recenti dotati di CPU, non solo PC, server e smartphone, ma anche dispositivi Internet of Things (IoT) come router e smart TV. Sfruttando il duo, è possibile leggere la memoria di sistema protetta, ottenendo l'accesso a password, chiavi di crittografia e altre informazioni sensibili.

Spectre e Meltdown sono esempi rappresentativi di attacchi di "esecuzione transitoria", che si basano su difetti di progettazione hardware nell'implementazione dell'esecuzione speculativa, del pipelining delle istruzioni e dell'esecuzione fuori ordine nelle CPU moderne. Sebbene questi tre elementi siano essenziali per le ottimizzazioni delle prestazioni insite nei processori moderni, le loro implementazioni variano tra i produttori di CPU e le microarchitetture; di conseguenza, non tutte le varianti di Spectre e Meltdown sono sfruttabili su tutte le microarchitetture.

Come funziona Spectre

Secondo gli autori originali del documento Spectre, "[induce] una vittima a eseguire speculativamente operazioni che non si verificherebbero durante l'elaborazione in ordine strettamente seriale delle istruzioni del programma e che fanno trapelare informazioni riservate della vittima attraverso un canale nascosto all'avversario".

Gli attacchi Spectre si svolgono in tre fasi:

1) La fase di setup, in cui il processore viene distratto per fare "una previsione speculativa errata".
2) Il processore esegue speculativamente le istruzioni dal contesto di destinazione in un canale nascosto della microarchitettura.
3) I dati sensibili vengono recuperati. Questo può essere fatto temporizzando l'accesso agli indirizzi di memoria nella cache della CPU.

Scritto da Gabriel

Come funziona Meltdown

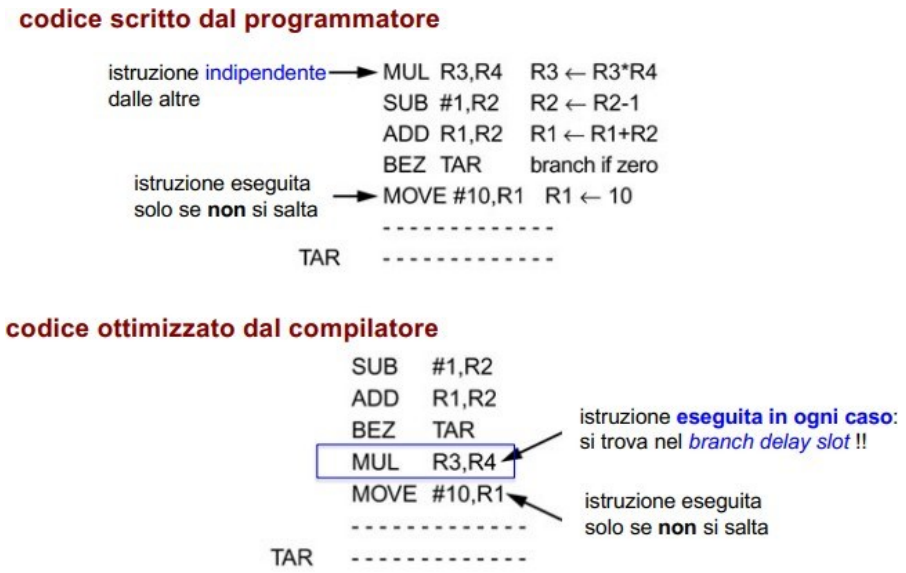
Meltdown sfrutta una race condition tra l'accesso alla memoria e il controllo del livello di privilegio durante l'elaborazione di un'istruzione. In combinazione con un attacco parallelo alla cache della CPU, i controlli dei livelli di privilegio possono essere aggirati, consentendo l'accesso alla memoria utilizzata dal sistema operativo o da altri processi in esecuzione. In determinate circostanze, questo può essere utilizzato per leggere la memoria nei container software.

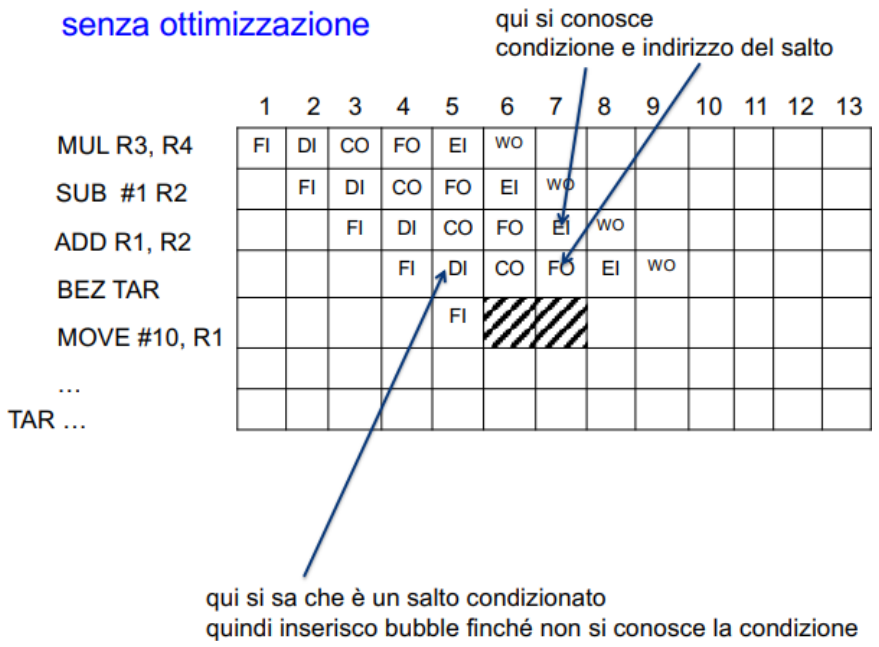
Gli attacchi Meltdown, secondo gli autori del documento originale, si svolgono in tre fasi:

1) Il contenuto di una posizione di memoria scelta dall'attaccante, inaccessibile a quest'ultimo, viene caricato in un registro.
2) Un'istruzione transitoria accede a una linea della cache in base al contenuto segreto del registro.
3) L'aggressore utilizza Flush+Reload per determinare la linea di cache a cui si accede e quindi il segreto memorizzato nella posizione di memoria scelta.

Nonostante la pubblicazione simultanea di Spectre e Meltdown, i due sfruttano proprietà diverse delle CPU; l'unico punto in comune tra Spectre e Meltdown è l'utilizzo dell'esecuzione transitoria.
Spectre si basa su eventi di errori sulla predizione dei branch per sollecitare istruzioni transitorie. Spectre funziona solo con i dati accessibili architettonicamente a un'applicazione. Meltdown, invece, si basa su istruzioni transitorie non ordinate in seguito a un'eccezione. Meltdown si basa su istruzioni transitorie inaccessibili architettonicamente a un'applicazione.

In ultimo, si ha il salto ritardato (delayed branch). Finché non si sa se ci sarà o no il salto (l'istruzione è in pipeline), invece di restare in stallo si può eseguire un'istruzione che non dipende dal salto.
L'istruzione successiva al salto è definita come branch delay slot. Il compilatore cerca di allocare nel branch delay slot una istruzione "opportuna" (magari inutile ma non dannosa) e la CPU esegue sempre l'istruzione del branch delay slot e, solo dopo, altera, se necessario, la sequenza di esecuzione delle istruzioni.





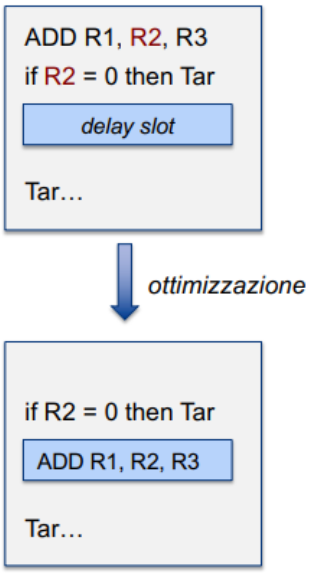
- se **non si salta**
  - continuo con MOVE
  - termino in 12 con **2** cicli di stallo
- se **si salta**
  - scarto MOVE e inizio con TAR
  - termino in 13 con **3** cicli persi (uno inutile + 2 stalli)

con delayed branch

qui si conosce condizione e indirizzo del salto

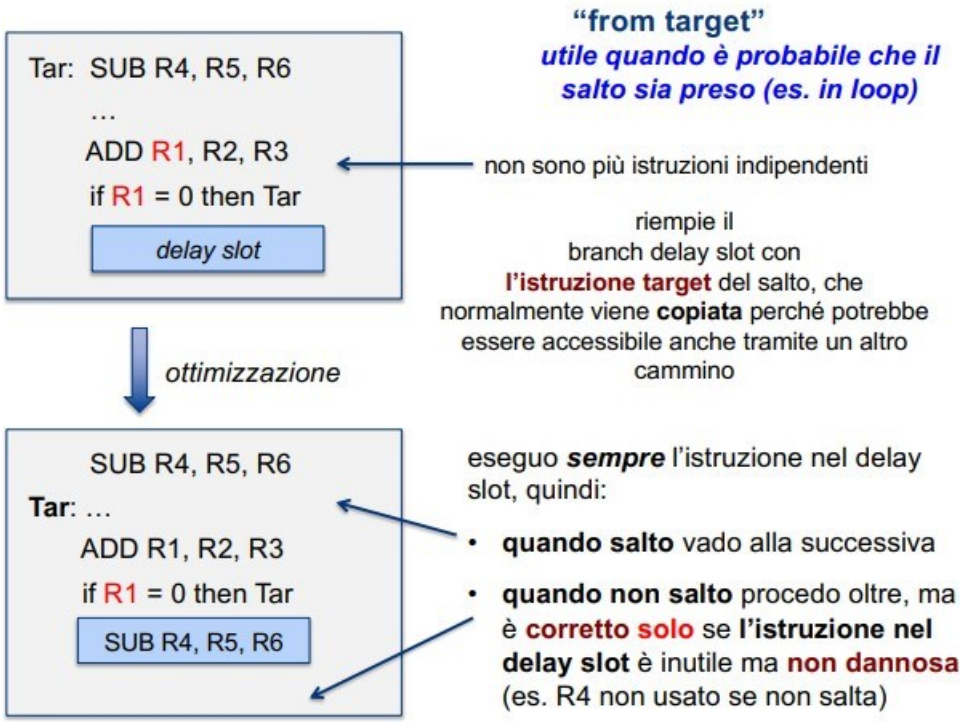
	1	2	3	4	5	6	7	8	9	10	11	12
SUB #1, R2	FI	DI	CO	FO	EI	WO						
ADD R1, R2		FI	DI	CO	FO	EI	WO					
BEZ TAR			FI	DI	CO	FO	EI	WO				
MUL R3,R4				**FI**	**DI**	**CO**	**FO**	**EI**	**WO**			
MOVE #10, R1					~~FI~~		DI	CO	FO	EI	WO	
…												
TAR …							**FI**	**DI**	**CO**	**FO**	**EI**	**WO**

- se **non si salta**
  - continuo con MOVE
  - termino in 11 con **1** ciclo di stallo
- se **si salta**
  - scarto MOVE e inizio con TAR
  - termino in 12 con **2** cicli persi (uno inutile e 1 stallo)



"from before"

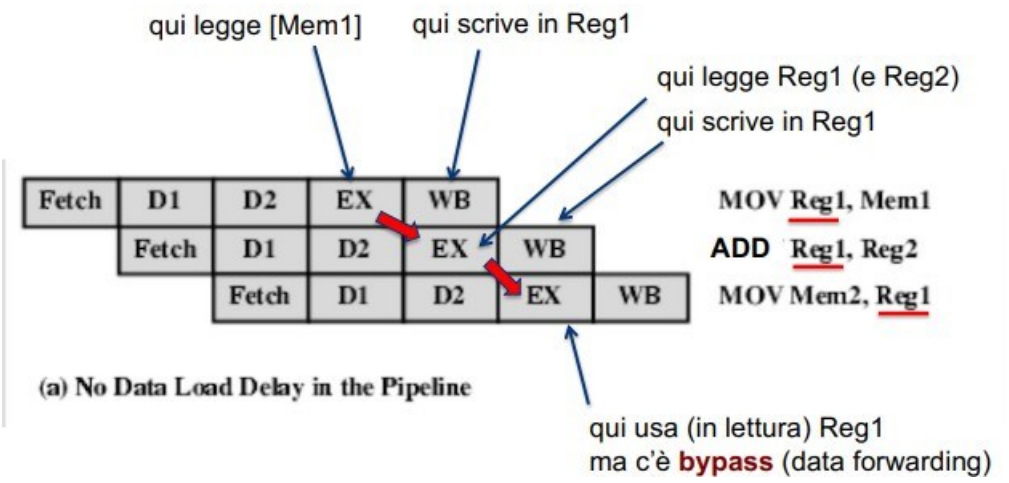
quando è possibile riempie il branch delay slot con un'istruzione **indipendente** proveniente dalla parte di codice che *precede* il salto



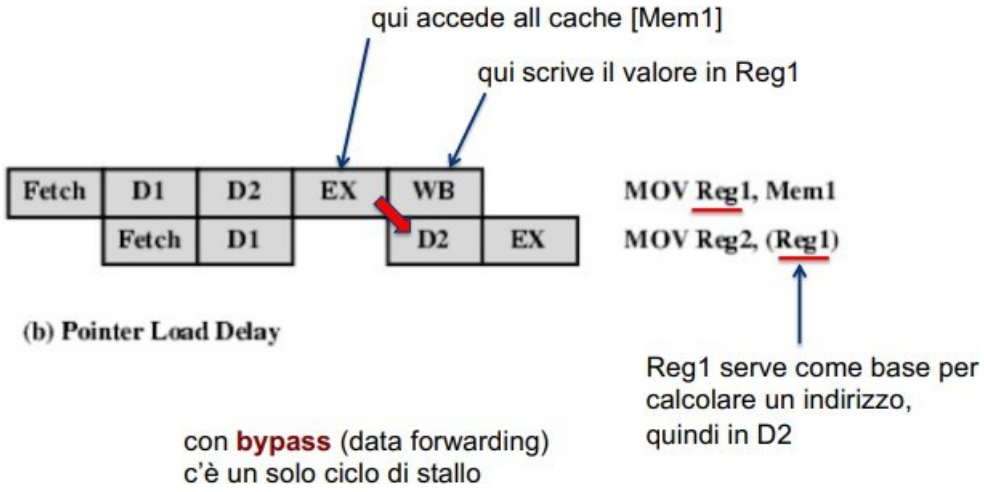
<u>Un esempio vero di pipelining: Intel 80846 Pipelining</u>

- Fetch
  - Istruzioni prelevate dalla cache o memoria esterna
  - Poste in uno dei due buffer di prefetch da 16 byte
  - Carica dati nuovi appena quelli vecchi sono "consumati"
  - Poiché le istruzioni sono a lunghezza variabile (1-11 byte), in media carica 5 istruzioni per ogni caricamento da 16 byte
  - Indipendente dagli altri stadi per mantenere i buffer pieni
- Decodifica 1 (D1)
  - Decodifica codice operativo e modi di indirizzamento
  - Le informazioni di sopra sono codificate (al più) nei primi 3 byte di ogni istruzione
  - Se necessario, indica allo stadio D2 di trattare i byte restanti (dati immediati e spiazzamento)
- Decodifica 2 (D2)
  - Espande i codici operativi in segnali di controllo per l'ALU
  - Calcola gli indirizzi in memoria per i modi di indirizzamento più complessi
- Esecuzione (EX)
  - Operazioni ALU, accesso alla cache (memoria).
- Retroscrittura (WB)
  - Se richiesto, aggiorna i registri e i flag di stato modificati in EX
  - Se l'istruzione corrente aggiorna la memoria, pone il valore calcolato in cache e nei buffer di scrittura del bus

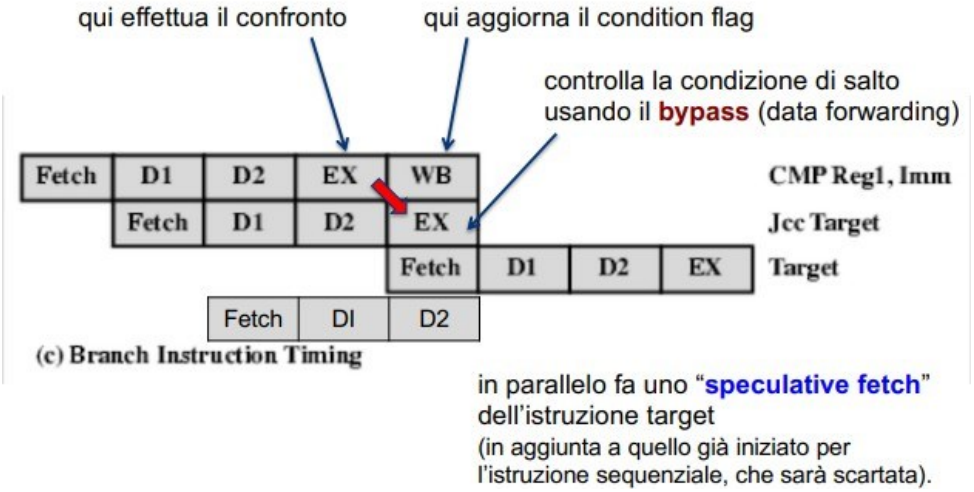
accessi consecutivi allo stesso dato non introducono ritardi



ritardo per valori usati per calcolare un indirizzo



salto condizionato. Assumiano venga eseguito



Esercizi pipeline

Esercizio: Dipendenze

- Che tipo di dipendenze si possono prevedere guardando questo codice sorgente?

```
if (a > c) {
  d = d + 5;
  a = b + d + e;
}
else {
  e = e + 2;
  f = f + 2;
  c = c + f;
}
b = a + f;
```

Soluzione: Dipendenze

- Che tipo di dipendenze si possono prevedere guardando questo codice sorgente?

```
if (a > c) {
  d = d + 5;
  a = b + d + e;
}
else {
  e = e + 2;
  f = f + 2;
  c = c + f;
}
b = a + f;
```

creeranno **dipendenze dal controllo**

- Che tipo di dipendenze si possono prevedere guardando questo codice sorgente?

```
if (a > c) {
  d = d + 5;
  a = b + d + e;
}
else {
  e = e + 2;
  f = f + 2;
  c = c + f;
}
b = a + f;
```

creeranno **dipendenze dai dati**

Esercizio Pipeline : Dipendenze

Si consideri il seguente frammento di codice:

```
LOOP: LW    $1  0 ($2)      R1← mem[0+[R2]]
      ADDI  $1  $1  1       R1← [R1] + 1
      SW    $1  0 ($2)      mem[0+[R2]] ← [R1]
      ADD   $2  $1  $2      R2← [R1] + [R2]
      SUB   $4  $3  $2      R4← [R3] - [R2]
      BENZ  $4  LOOP        if ([R4] != 0) PC ← indirizzo(loop)
```

si individuino le dipendenze **R**ead**A**fter**W**rite (RAW) e **W**rite**A**fter**W**rite (WAW).

Soluzione

#	codice	R1	R2	R3	R4	commento
1	LOOP: LW $1, 0 ($2)	**W**	**R**			legge R2, scrive R1
2	ADDI $1,$1, 1	**RW**				legge e scrive R1
3	SW $1, 0($2)	**R**	**R**			legge R1 e R2
4	ADD $2, $1, $2	**R**	**RW**			legge R1, legge e scrive R2
5	SUB $4, $3, $2		**R**	**R**	**W**	legge R2 e R3, scrive R4
6	BENZ $4, LOOP				**R**	legge R4

Linee codice	Spiegazione dipendenza	Tipo
2←1	ADDI legge R1 che è scritto da LW	RAW
2←1	ADDI scrive R1 che è scritto da LW	WAW
3←**2**, 3←**1**	SW legge R1 che è scritto da ADDI, e prima da LW	RAW
4←2, 4←1	ADD legge R1 che è scritto da ADDI, e prima da LW	RAW
5←4	SUB legge R2 che è scritto da ADD	RAW
6←5	BENZ legge R4 che è scritto da SUB	RAW

Si consideri una pipeline a 4 stadi (IF, ID, EI, WO) per cui:

- i salti incondizionati sono risolti (identificazione salto e calcolo indirizzo target) alla fine del secondo stadio (ID)
- i salti condizionati sono risolti (identificazione salto, calcolo indirizzo target e calcolo condizione) alla fine del terzo stadio (EI)
- il primo stadio (IF) è indipendente dagli altri
- ogni stadio impiega 1 ciclo di clock

- Si considerino le seguenti statistiche:
  - 15% delle istruzioni sono di salto condizionale
  - 1% delle istruzioni sono di salto incondizionale
  - Il 60% delle istruzioni di salto condizionale hanno la condizione soddisfatta (prese)

valutare i ritardi nella pipeline introdotti dai salti e calcolare il valore del fattore di velocizzazione della pipeline

fattore di velocizzazione di una pipeline a k stadi, a regime, in funzione del numero di stalli:

$$S_k = \frac{1}{1 + frazione\_cicli\_stallo}\, k$$

frazione_cicli_stallo = numero_cicli_di_stallo / numero_cicli_esecuzione_codice

Filosofia RISC

Tra le principali innovazioni dei computer, esiste proprio l'architettura per processori RISC (Reduced Instruction Set Computer), che è un tipo di architettura di microprocessore che utilizza un piccolo insieme di istruzioni altamente ottimizzate, anziché un insieme di istruzioni altamente specializzate come quelle tipiche di altre architetture. Il RISC è un'alternativa all'architettura CISC (Complex Instruction Set Computing) ed è spesso considerato la tecnologia di architettura CPU più efficiente oggi disponibile.

Con il RISC, un'unità di elaborazione centrale (CPU) implementa il principio di progettazione del processore che prevede istruzioni semplificate che possono fare meno cose ma possono essere eseguite più rapidamente. Il risultato è un miglioramento delle prestazioni. Una caratteristica fondamentale del RISC è che consente agli sviluppatori di aumentare il set di registri e di incrementare il parallelismo interno, aumentando il numero di thread paralleli eseguiti dalla CPU e aumentando la velocità di esecuzione delle istruzioni della CPU. ARM, o "Advanced RISC Machine", è una famiglia specifica di architetture a set di istruzioni basate su un'architettura a set di istruzioni ridotto sviluppata da Arm Ltd.

Essa è frutto della co-evoluzione tra hardware e linguaggi di programmazione, arrivando a linguaggi ad alto livello, che permettono di esprimere l'algoritmo risolutivo in modo più conciso, lasciano al compilatore il compito di gestire i dettagli e supportano costrutti di programmazione strutturata. Sussiste un gap semantico tra i linguaggi HLL/High Level Languages descritti e il linguaggio macchina, infatti si hanno:

- esecuzione inefficiente
- taglia eccessiva del programma in linguaggio macchina
- complessità del compilatore

In tutta risposta, i progettisti hardware hanno ampliato il set di istruzioni, hanno creato svariati modi di indirizzamento e si cerca un'implementazione hardware di costrutti di linguaggi ad alto livello (es. CASE (switch) su architettura VAX).
In questo modo:

- si semplifica il lavoro del compilatore
- migliora l'efficienza dell'esecuzione (sequenze di operazioni complesse implementate tramite microcodice)
- si supportano linguaggi HL più complessi

Alternativamente, si cerca di individuare le caratteristiche e i pattern di esecuzione delle istruzioni macchina generate dai programmi dei linguaggi ad alto livello e per *semplificare* l'architettura.
Con questo si intende che si vuole:

- semplificare le funzionalità del processore e la sua interazione con la memoria
- tipo e frequenza d'uso degli operandi determinano l'organizzazione della memoria e i modi di indirizzamento
- organizzazione del flusso dell'esecuzione (pipeline) e suo controllo.

Occorre in generale:

- fare un' analisi delle istruzioni macchina generate dai programmi scritti in HLL
- misure dinamiche: raccolte eseguendo il programma e contando il numero di occorrenze di una certa proprietà o di una certa caratteristica. (le misure statiche si basano solo sul programma sorgente, che non dice quante volte è eseguita un'istruzione)

Nei vari linguaggi sussiste una predominanza di istruzioni di assegnamento (quindi si deve rendere efficiente il trasferimento dei dati) e molte istruzioni condizionali (controllando le dipendenze dai salti).
Si deve anche capire quali sono le istruzioni macchina che rubano più tempo: normalmente sono i cicli, le chiamate e le condizioni.

	Occorrenza Dinamica		Occorrenza ponderata sulle istruzioni		Occorrenza ponderata sugli accessi a memoria	
	Pascal	C	Pascal	C	Pascal	C
ASSIGN	45%	38%	13%	13%	14%	15%
LOOP	5%	3%	42%	32%	33%	26%
CALL	15%	12%	31%	33%	**44%**	**45%**
IF	29%	43%	11%	21%	7%	13%
GOTO	—	3%	—	—	—	—
OTHER	6%	1%	3%	1%	2%	1%

In generale, dipende da:
• quale linguaggio HL
• quale tipo di applicazione
• quale architettura sottostante
• resta rappresentativa delle contemporanee architetture CISC (Il termine CISC sta per ''Complex Instruction Set Computer''. Si tratta di un progetto di CPU basato su singoli comandi, che sono in grado di eseguire operazioni in più fasi).

Per gli *operandi*, costituiti principalmente variabili scalari locali- L'ottimizzazione si deve concentrare sull'accesso alle variabili locali scalari.
In generale, le *chiamate di procedura* sono le istruzioni che consumano più tempo, va quindi trovata un'implementazione efficiente. Due aspetti significativi:
– il numero di parametri e variabili gestite
– il livello di annidamento (nesting)
Misurazioni:
– meno di 6 parametri, meno di 6 variabili locali
– la maggior parte degli operandi sono variabili locali
– poco annidamento di chiamate di procedure

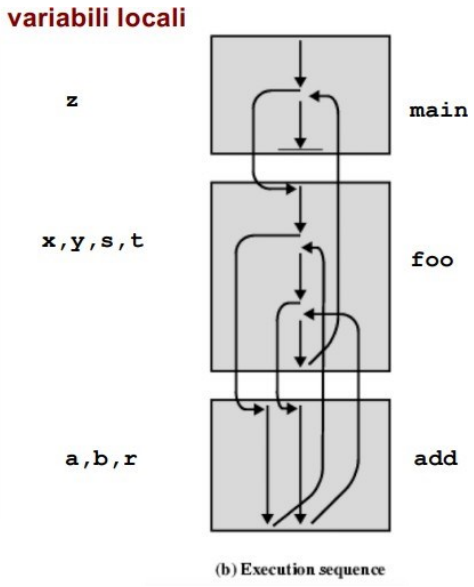
Strategia migliore per supportare i linguaggi di alto livello:
• non rendere le istruzioni macchina più simili alle istruzioni di HLL
• ottimizzare le performance dei pattern più usati e più time-consuming
1. ampio numero di registri o loro uso ottimizzato dal compilatore
– per ottimizzare gli accessi agli operandi (abbiamo visto che sono istruzioni molto frequenti, con operandi perlopiù scalari e locali, quindi è utile ridurre gli accessi alla memoria aumentando gli accessi ai registri)
2. progettazione accurata della pipeline
– gestione delle dipendenze dal controllo dovute a (frequenti) salti e chiamate di procedure evitando i prefetch errati
3. set di istruzioni semplificato (ridotto) e implementato in maniera efficiente.

Per quanto riguarda i *registri*, hanno indirizzi più brevi di quelli per l'uso di cache e memoria principale e bisogna assicurare che gli operandi usati siano il più possibile mantenuti nei registri, minimizzando i trasferimenti memoria-registro.

• Soluzione hardware:
– aumentare il numero di registri,
– così si mantengono più variabili per più tempo
• Soluzione software:
– il compilatore massimizza l'uso dei registri
– le variabili più usate per ogni intervallo di tempo sono allocate nei registri
– richiede sofisticate tecniche di analisi dei programmi

In generale, si cerca di memorizzare nei registri le variabili scalari locali (le più frequenti) e si hanno pochi registri per le variabili locali (quindi, i valori utili per ogni procedura attualmente in uso, con lo scope).

```
int main(){
  int z;
  z = foo(2,5);
  cout << "The result is" << z;
}

int foo(int x, int y){
  int s = add(x,8);
  int t = add(y,3);
  return s+t;
}

int add(int a, int b){
  int r;
  r = a+b;
  return r;
}
```

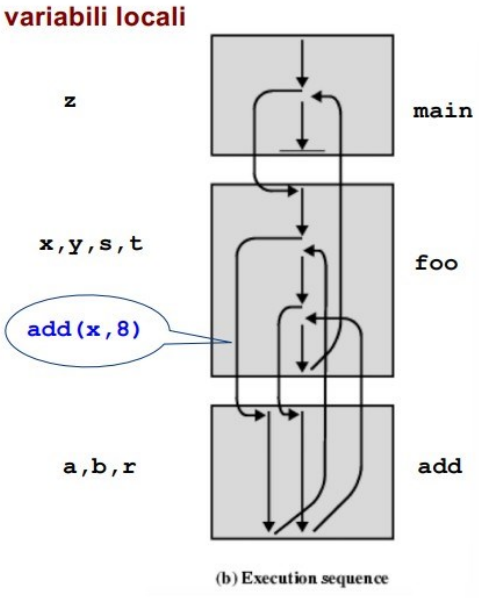


(b) Execution sequence

```
int foo(int x, int y){
  int s = add(x,8);
  int t = add(y,3);
  return s+t;
}

int add(int a, int b){
  int r;
  r = a+b;
  return r;
}
```

ogni **chiamata** di procedura:
- salva le variabili locali dai registri in memoria
- può riusare i registri per le nuove variabili locali
- passa i parametri

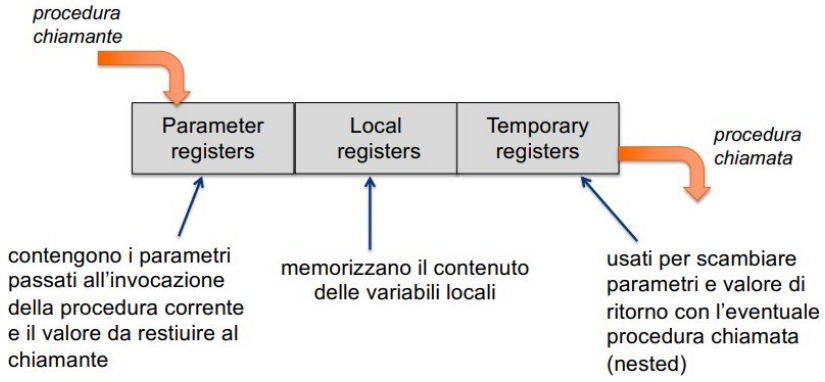
al **termine** della precedura:
- ripristina nei registri (i valori del) le variabili locali del chiamante
- restituisce il risultato

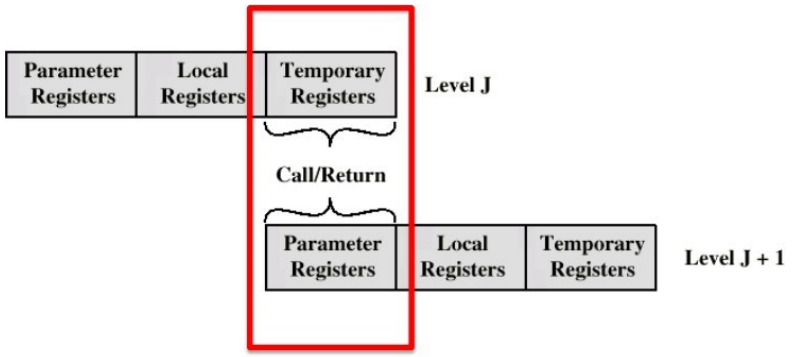


(b) Execution sequence

Idea per usare al meglio i (tanti) registri general-purpose:
• suddividere i registri in molti piccoli gruppi (di taglia fissa)
• ogni procedura ha il proprio gruppo/finestra di registri
• è sempre visibile (indirizzabile) un solo gruppo/finestra: quello corrispondente alla procedura attiva in quel momento.
Una chiamata di procedura:
– cambia automaticamente il gruppo di registri da usare
– invece di provocare il salvataggio dei dati in memoria
– al ritorno viene riselezionato il gruppo di registri assegnato in precedenza alla procedura chiamante
– le finestre relative a procedure adiacenti sono parzialmente sovrapposte, in modo da facilitare il passaggio dei parametri.

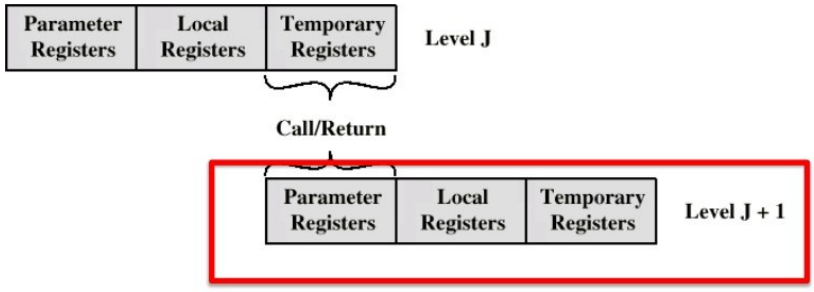
Usiamo quindi le cosiddette *finestre di registri*, in cui ogni gruppo/finestra è diviso in tre sottogruppi.





sono fisicamente gli stessi registri

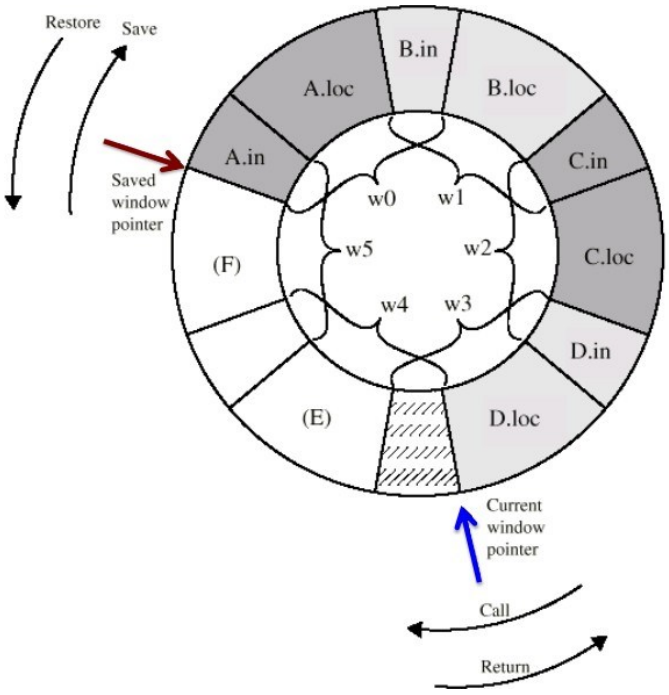
si possono passare i parametri **senza trasferire dati**



in ogni momento è visibile, e **indirizzabile**, **un solo gruppo/finestra**, usando indirizzi da 0 a N-1

Quante finestre di registri?
– una per chiamata di procedura attivata (nesting)
– c'è spazio per un numero limitato: solo le più recenti
– le attivazioni precedenti vanno salvate in memoria e poi recuperate quando diminuisce il nesting

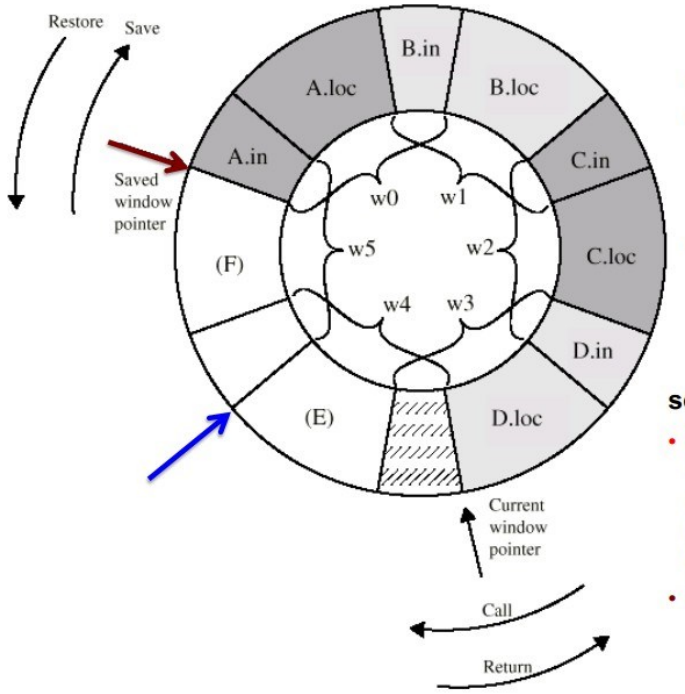
I registri sono organizzati a buffer circolare, come segue:



4 procedure annidate:
- A -> B -> C -> D
- D procedura attiva

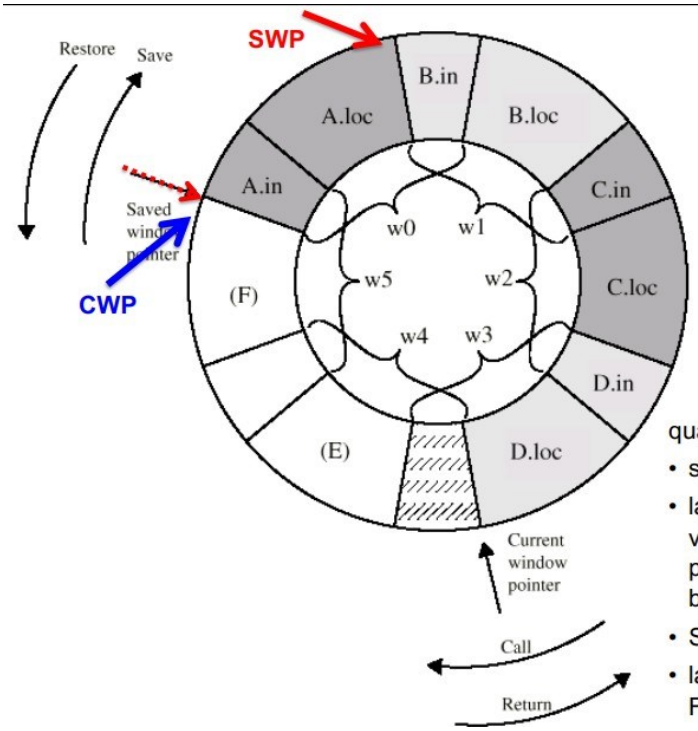
Current Window Pointer
- punta alla finestra della procedura correntemente attiva
- i riferimenti ai registri usati nelle istruzioni macchina sono **offset a partire dal CWP**

Saved Window Pointer
- indica dove si deve ripristinare l'ultima finestra salvata in memoria



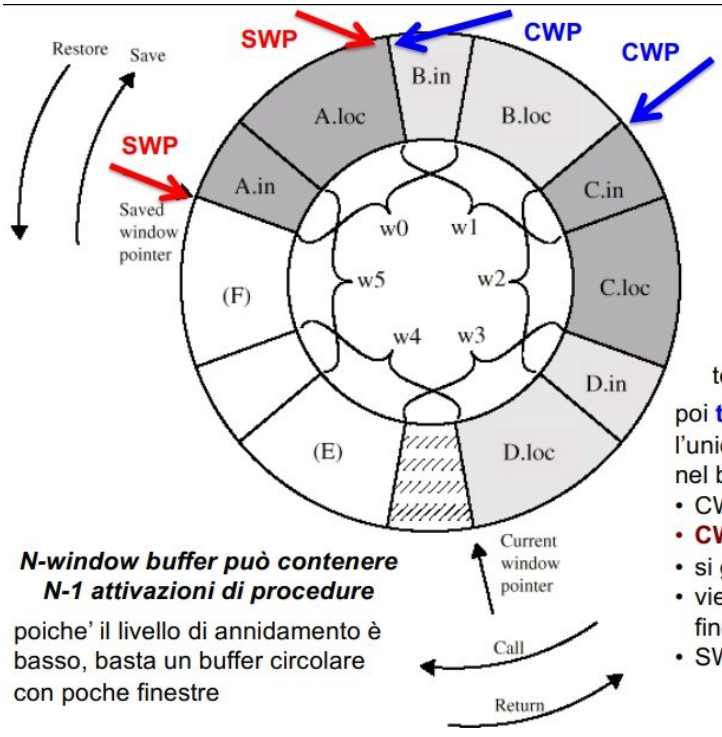
se D chiama E
- i parametri per E sono messi nei temporary registrers di D (= parameter reg. di E)
- il CWP avanza di una finestra

se E chiama F
- **non è possibile**: la finestra di F si sovrappone a quella di A rischia di sovrascrivere i parametri di A
- CWP = SWP (mod 6)



quando **CWP = SWP** (mod 6)

- si genera un interrupt
- la finestra più vecchia (A) viene salvata in memoria principale (bastano i primi 2 blocchi)
- SWP viene incrementato
- la chiamata della procedura F può procedere



N-window buffer può contenere N-1 attivazioni di procedure

poiche' il livello di annidamento è basso, basta un buffer circolare con poche finestre

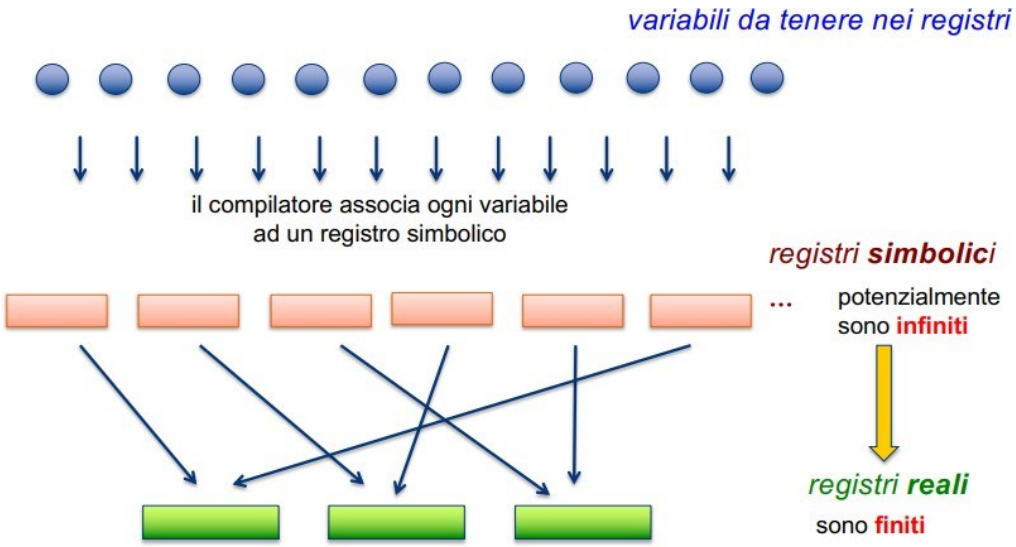
terminano F, E, D e C

poi **termina anche B**, che è l'unica procedura rimasta nel buffer

- CWP viene decrementato
- **CWP = SWP**
- si genera un interrupt
- viene ripristinata la finestra di A
- SWP viene decrementato

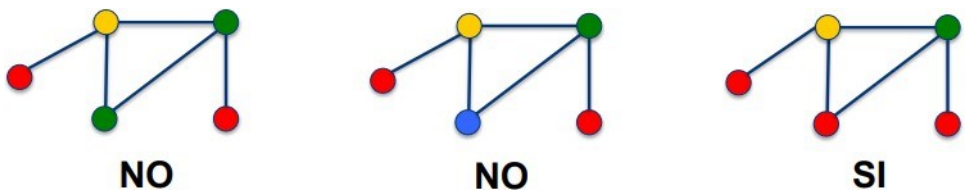
Parliamo anche delle *variabili globali*, cioè variabili accessibili da qualunque procedura, e più di una.
Dove memorizzarle?

- il compilatore le alloca in memoria, ma è poco efficiente se sono usate spesso
- Soluzione: usare un gruppo di registri ad hoc, disponibili a tutte le procedure
    - Scopo: trovare gli operandi il più possibile nei registri e minimizzare le operazioni di load/store
    - Soluzione software: l'architettura RISC può avere pochi registri (16-32) il cui uso viene ottimizzato dal compilatore
        - Linguaggi ad alto livello non fanno riferimento esplicito ai registri, eccezione in C: register int

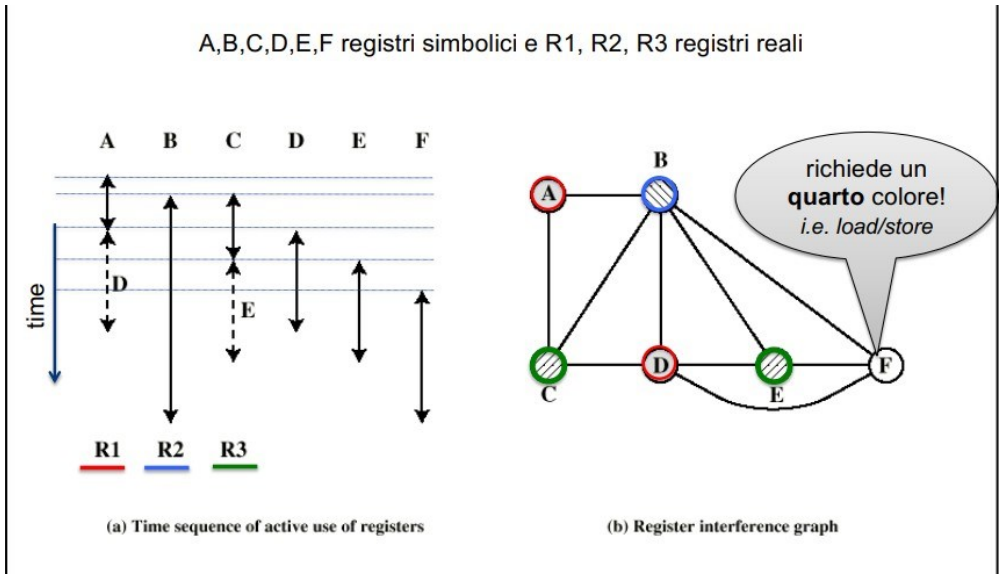


- il compilatore mappa ogni registro simbolico ad un registro reale
- se due registri simbolici **si usano in momenti diversi**, possono essere **mappati sullo stesso registro reale**
- se in un certo intervallo di tempo i **registri reali non sono in numero sufficiente** per contenere tutte le variabili riferite in quell'intervallo, alcune variabili vengono **mantenute nella memoria principale**

Continuando con l'ottimizzazione dei registri, occorre decidere quale registro simbolico (quale variabile) assegnare a quale registro reale in ogni momento.
Si hanno m compiti da eseguire, n risorse, con $m >> n$. Decidere quale compito assegnare a quale risorsa in ogni momento (es. m voli da effettuare, n aerei).
Equivale a risolvere un problema di colorazione di un grafo:
– assegnare un colore ad ogni nodo in modo che nodi adiacenti abbiano colori diversi usando il minimo numero di colori



Qui i nodi (tanti) corrispondono ai registri simbolici e due nodi sono collegati da un arco se i due registri simbolici (variabili) sono "in vita" nello stesso intervallo di tempo/porzione di codice. I colori (pochi) corrispondono ai registri reali ed i nodi dello stesso colore possono essere assegnati allo stesso registro reale. Se servono più colori di quanti sono i registri reali, allora i nodi che non riescono ad essere colorati vanno memorizzati in memoria principale.



Decidere se un grafo è colorabile con k colori è un problema non risolvibile polinomialmente → NP-Completo e si usano algoritmi efficienti per casi specifici, usando 32/64 registri fisici che spesso si dimostrano sufficienti.

CISC

Ora parliamo di CISC, appunto, quindi un ampio insieme di istruzioni, istruzioni più complesse per semplificare compilatore e migliorare performance. Il compilatore deve generare "buone" sequenze di istruzioni macchina, cioè brevi e veloci da eseguire. Un' istruzione complessa può essere eseguita più velocemente di una serie di istruzioni più semplici, ma:

- l'unità di controllo diventa più complessa
- il controllo microprogrammato necessita di più spazio
- quindi si rallenta l'esecuzione delle istruzioni più semplici, che restano le più frequenti

Queste le principali caratteristiche:
• un' istruzione per ciclo di clock
– (instruction cycle): tempo impiegato per fare fetch-decode-execute-write di un'istruzione elementare.
– RISC: hanno un ciclo esecutivo che dura un solo machine cycle, quindi se la pipeline è piena, ad ogni ciclo di clock termina un'istruzione
– istruzioni CISC richiedono più di un ciclo;
• operazioni da registro a registro, tranne LOAD e STORE
– CISC attuali hanno anche operazioni memory-to-memory e register/memory
– poichè si usano di frequente scalari locali, aumentando o ottimizzando i registri la maggior parte degli operandi stanno a lungo nei registri.
• pochi e semplici modi di indirizzamento
– indirizzo di registro, spiazzamento (relativo a PC)
– si semplifica l'istruzione e l'unità di controllo
• pochi e semplici formati fissi per le istruzioni
– campi e opcode a dimensione fissa, così la decodifica dell'opcode e l'accesso ai registri per gli operandi possono essere simultanei

Scritto da Gabriel

– istruzioni a lunghezza fissa sono allineate con la lunghezza delle parole, quindi il fetch è ottimizzato per prelevare (multipli di) una parola
– la regolarità facilita le ottimizzazioni del compilatore
– più responsivo agli interrupt, controllati tra due istruzioni più semplici

• unità di controllo cablata:
– se cablata (cioè hardware) è meno flessibile ma più veloce
– se microprogrammata più flessibile ma meno veloce

Non è evidente quale sia l'architettura nettamente migliore. Problemi per fare un confronto:
– Non esistono architetture RISC e CISC che siano direttamente confrontabili
– Non esiste un set completo di programmi di test
– Difficoltà nel separare gli effetti dovuti all'hardware rispetto a quelli dovuti al compilatore
– Molti confronti sono stati svolti su macchine prototipali e semplificate e non su macchine commerciali
– Molte CPU commerciali utilizzano idee provenienti da entrambe le filosofie:
  • PowerPC architettura RISC con elementi CISC
  • Pentium II architettura CISC con elementi RISC

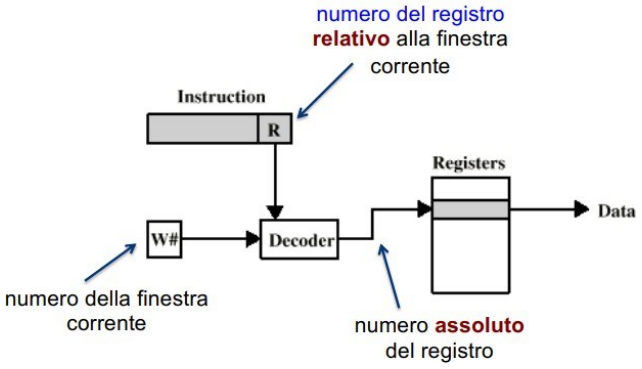
Le architetture specifiche per il dominio (domain-specific architectures) sono l'unica opzione per migliorare le prestazioni, dato che i benefici della legge di Moore iniziano a svanire:
Un'architettura software specifica per il dominio (DSSA) è un insieme di componenti software:
- specializzati per un particolare dominio
- generalizzati per un uso efficace in tutto il dominio e composti in una struttura standardizzata (topologia) efficace per costruire applicazioni di successo.

Similmente, confrontiamo i registri con la cache:

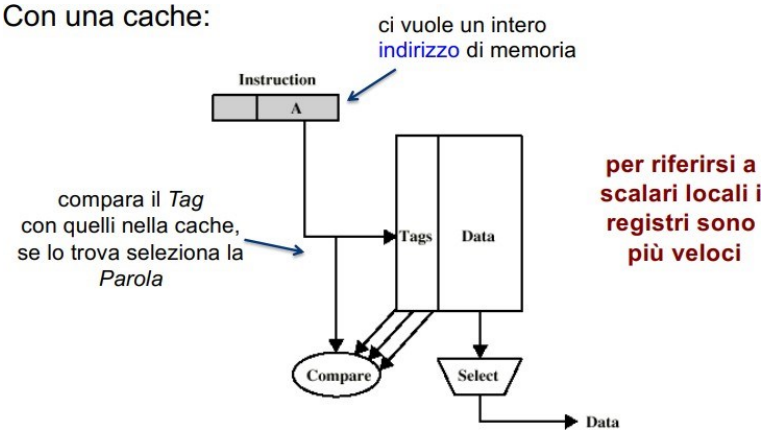
Ampio Banco di Registri (*Register file*) a finestre	**Cache**
Tutti gli scalari locali	Scalari locali **usati di recente**
Variabili individuali	Blocchi di memoria
Variabili globali assegnate dal compilatore ai registri ad hoc	Variabili globali usate di recente
Save/Restore basato sulla profondità di annidamento delle procedure	Save/Restore basato sull'algoritmo di sostituzione adottato dalla cache
Indirizzamento a registro	Indirizzamento a memoria

Il riferimento ad uno scalare cambia a seconda che si usi un banco a registri oppure una cache:

Con un banco di registri organizzato a finestre:



Con una cache:



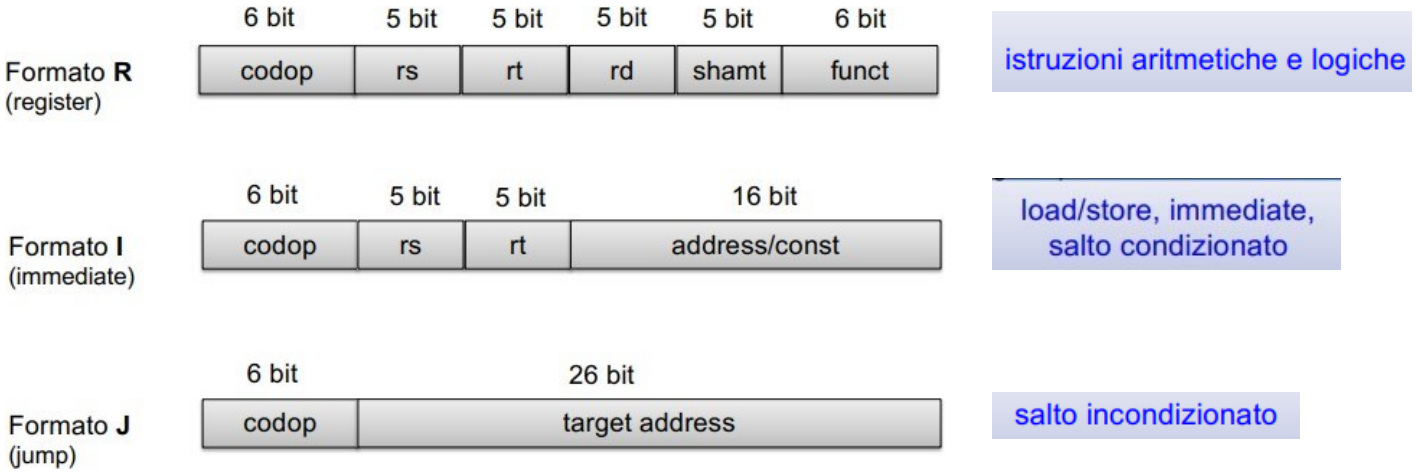
Architettura MIPS-32

Studiamo i processori MIPS (Microprocessor without Interlocked Pipeline Stages) come esempio di architettura RISC, un'architettura sperimentale sviluppata a Stanford negli anni '80 e poi sviluppata commercialmente.
Istruzioni:
• Tutte le istruzioni di dimensione 32 bit
• Tutte le operazioni sui dati sono da registro a registro
– le istruzioni che manipolano i dati usano i valori dei registri
• le operazioni sulla memoria:
– solo load e store, per trasferire dati tra memoria e registri
– nessuna operazione memoria-memoria
• quindi tutte le istruzioni operano su registri, es add $1, $2, $3

Registri:
– 32 registri di 32 bit
– si indicano con $1, $2, $3…. $0 contiene sempre 0

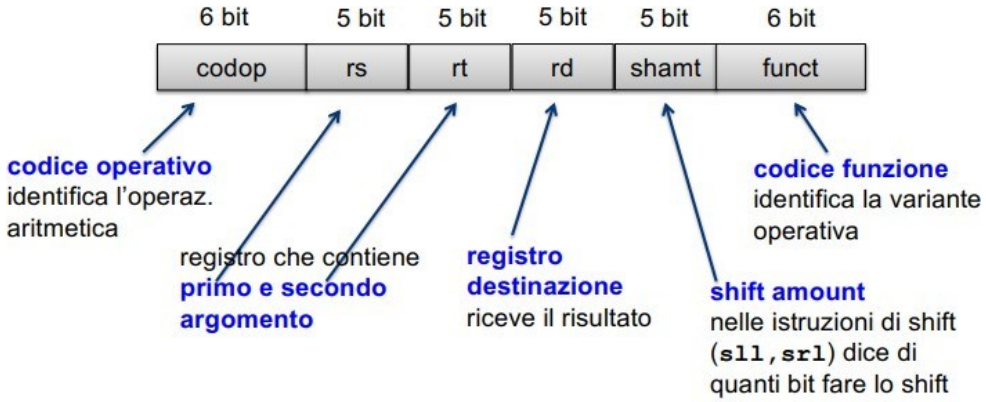
Dati:
• Registri possono essere caricati con byte, mezze parole, e parole
• i registri sono a 32 bit, quindi un dato "più corto" può essere "allungato" riempiendo i bit rimanenti con 0 o estendendo il segno (cioè replicandolo)

I modi di indirizzamento sono diversi: immediato, displacement/spiazzamento ma anche indiretta registro (displacement a 0) e assoluta (registro 0 come registro base).
Fornisce 32 bit per tutte le istruzioni con 3 formati diversi:

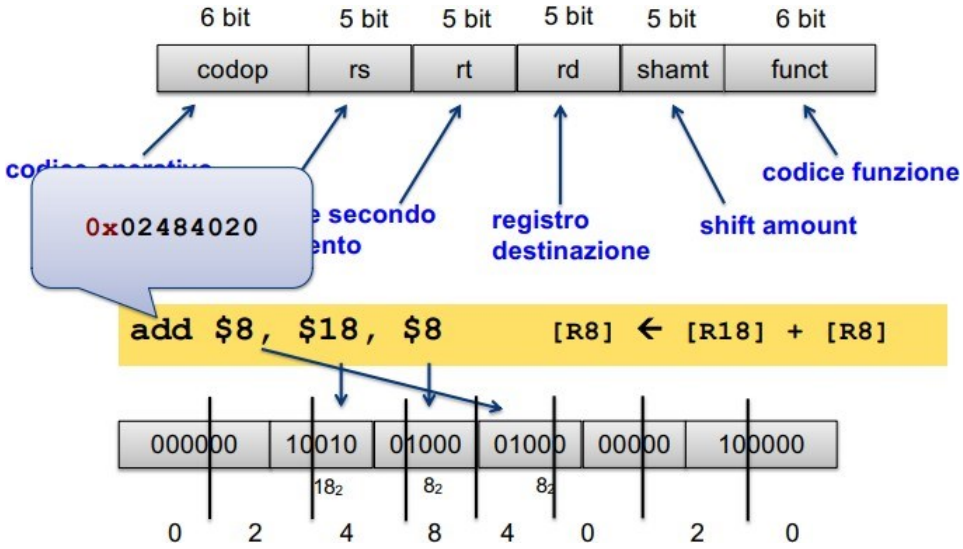


La dimensione fissa di opcode e di riferimenti a registri semplifica Instruction Decode e Fetch di Operandi.
Esaminiamo i formati singolarmente.

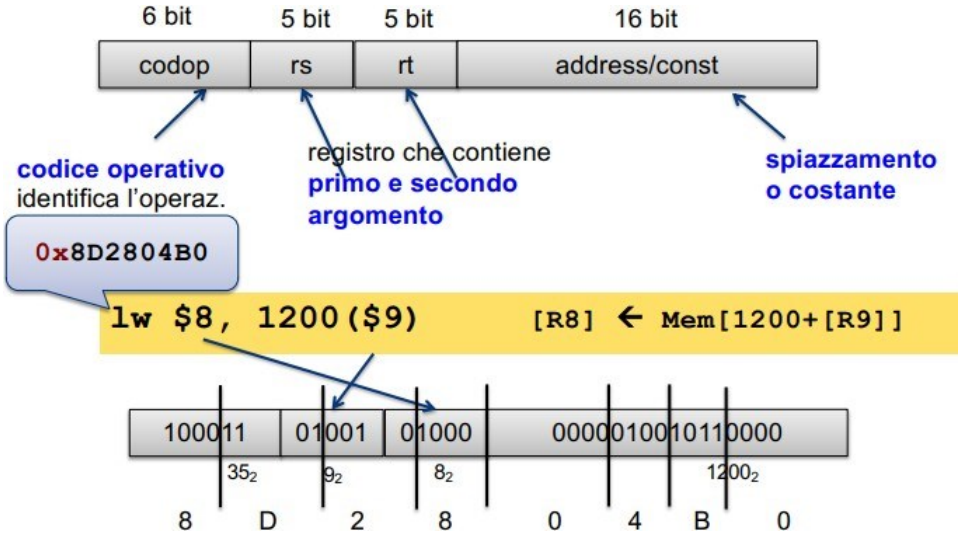
Per il formato R, usato per istruzioni aritmetiche e logiche. 6 campi a dimensione fissa:



5 bit bastano per indicare quale dei 32 registri

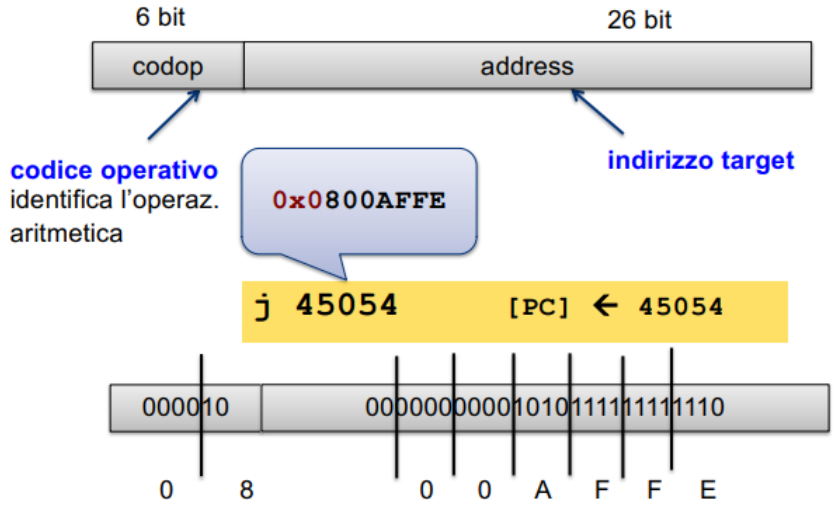


Per il formato I, istruzioni load/store, immediate e salto condizionato:



```
lw $8, 1200($9)      [R8] ← Mem[1200+[R9]]
sw $8, 1200($9)      Mem[1200+[R9]] ← [R8]
beq $1, $2, 16       if ($1==$2) PC=PC+16
addi $1, $8, 4       [R1] ← [R8]+4
```

Poi il formato J, per il salto incondizionato:



Le istruzioni MIPS hanno un ciclo esecutivo preciso:
1. IF Instruction Fetch

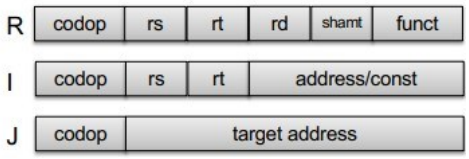
- IR ← Mem[PC] *(IR = instruction register, PC = program counter)*
- PC, NPC ← PC+4 *(NPC è un registro temporaneo) 1 word = 4 byte*

2. ID Instruction Decode / register fetch

- **Formato R** A ← Regs[rs]; B ← Regs[rt]
- **Formato I** A ← Regs[rs]; B ← Regs[rt] ; Imm ← campo immediato di IR
- **Formato J** Imm ← campo immediato di IR
- il campo immediato di IR va esteso a 32 bit, estendendo il segno
- A, B, Imm registri temporanei

3. EX Execution / address calculation
   – tutte le istruzioni usano la ALU (tranne salto incondizionato):
   – operazioni logico-aritmetiche
   – load/store e jump per calcolare l'indirizzo
   – salti condizionati per calcolare la condizione

- **Riferimento a memoria**
  - `lw $8, 1200($9)` Formato I A=R[rs]=[$9] Imm=1200*(esteso)*
  - ALUOutput ← A + Imm //address calculation
- **Istruzione ALU registro-registro**
  - `add $8, $18, $8` Formato R A=R[rs]=[$18] B=R[rt]=[$8]
  - ALUOutput ← A ***funct*** B
- **Istruzione ALU registro-immediato**
  - `addi $8, $18,4` Formato I A=R[rs]=[$18] Imm=4*(esteso)*
  - ALUOputpu ← A ***op*** Imm
- **Salto**
  - `j 45054` Formato J
  - `beq $1, $2, 16` Formato I



- **Salto incondizionato**
  - `j 45054` Formato J Imm = 45054
  - l'indirizzo nell'istruzione si riferisce al numero di **parole** di cui far avanzare il PC (NPC), ma gli indirizzi MIPS sono al **byte**
  - quindi il numero di parole va trasformato in numero di byte: moltilplicando per 4 cioè **due shift a sinistra <<2**
  - Target ← NPC + (Imm << 2)
- **Salto condizionato**
  - `beq $1, $2, 16` Formato I A=R[rs]=[$1] B=R[rt]=[$2] Imm=16
  - deve valutare la condizione e calcolare l'indirizzo di salto
  - Cond ← (A-B)==0 la ALU fornisce in uscita un segnale che indica se il risultato è 0
  - Target ← NPC + (Imm <<2)

4. MEM Memory access / branch completion

- **Riferimento a memoria**
  - `lw $8, 1200($9)` A=[$9] B=[$8] Imm=1200 ALUOutput=A+Imm
  - LMD ← Mem[ALUOutput] (Load Memory Data register)
  - `sw $8, 1200($9)` A=[$9] B=[$8] Imm=1200 ALUOutput=A+Imm
  - Mem[ALUOutput] ← B
- **Salto**
  - **incondizionato** PC ← Target
  - **condizionato** if (Cond) PC ← Target

5. WB Write Back: scrittura del risultato nei registri

- **Riferimento a memoria (solo load)**
  - `lw $8, 1200($9)` A=$9 B=Reg[rt]=$8 LMD ← Mem[ALUOutput]
  - Regs[rt] ← LMD
- **Istruzione ALU registro-registro**
  - `add $8, $18, $8` Formato R A=R[rs]=$18 B=R[rt]=$8
  - Regs[rd] ← ALUOutput
- **Istruzione ALU registro-immediato**
  - `addi $8, $18,4` Formato I A=R[rs]=$18 B=R[rt] Imm=4
  - Regs[rt] ← ALUOutput

Nel dettaglio, sulle singole istruzioni MIPS:

`add $8, $18, $8` R

codop	rs	rt	rd	shamt	funct

IF
- IR ← Mem[PC]
- PC, NPC ← PC+4

ID
- A ← Regs[rs] = [$18]
- B ← Regs[rt] = [$8]

EX
- ALUOutput ← A funct B

MEM
- nessuna operazione

WB
- Regs[rd]=$8 ← ALUOutput

`lw $8, 1200($9)` I

codop	rs	rt	address/const

IF
- IR ← Mem[PC]
- PC, NPC ← PC+4

ID
- A ← Regs[rs] = [$9] B ← Regs[rt]=[$8]
- Imm ← campo immediato di IR = 1200 *(esteso)*

EX
- ALUOutput ← A + Imm

MEM
- LMD ← Mem[ALUOutput]

WB
- Regs[rt]=$8 ← LMD

`sw $8, 1200($9)` I

- IF
  - IR ← Mem[PC]
  - PC, NPC ← PC+4
- ID
  - A ← Regs[rs] = [$9] B ← Regs[rt]=[$8]
  - Imm ← campo immediato di IR = 1200 *(esteso)*
- EX
  - ALUOutput ← A + Imm
- MEM
  - Mem[ALUOutput] ← B
- WB
  - nessuna operazione

`beq $1,$2,16` I

- IF
  - IR ← Mem[PC]
  - PC, NPC ← PC+4
- ID
  - A ← Regs[rs] = [$1] B ← Regs[rt]=[$2]
  - Imm ← campo immediato di IR = 16 *(esteso)*
- EX
  - Cond ← (A-B)==0
  - Target ← NPC + (Imm <<2)
- MEM
  - if (Cond) PC ← Target
- WB
  - nessuna operazione

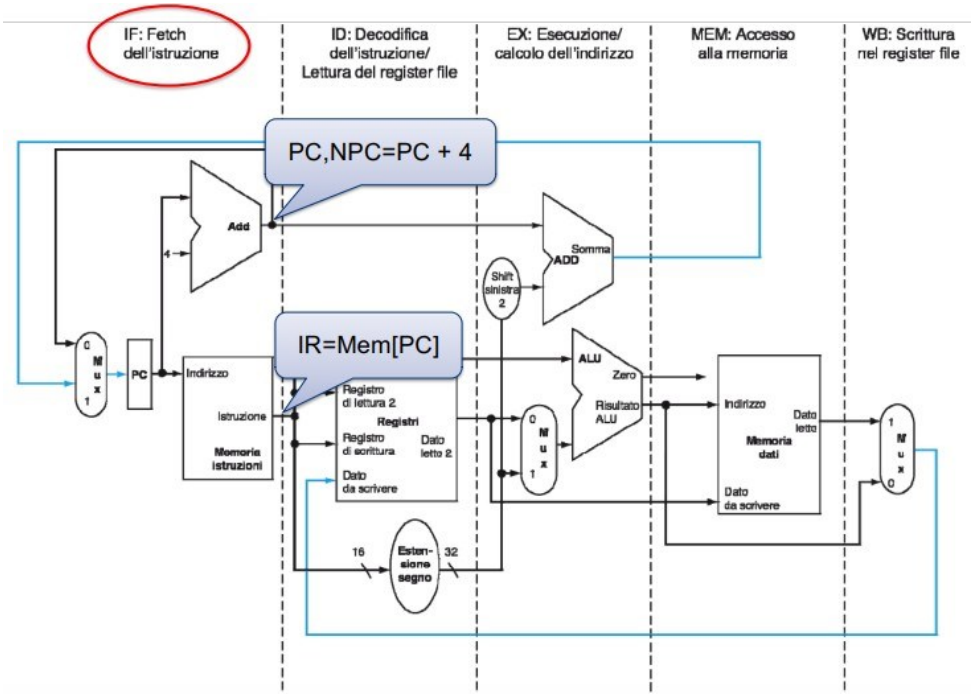
…non è nella fase MEM di beq, ma nella fase IF che avviene al ciclo successivo a quello di EX di beq

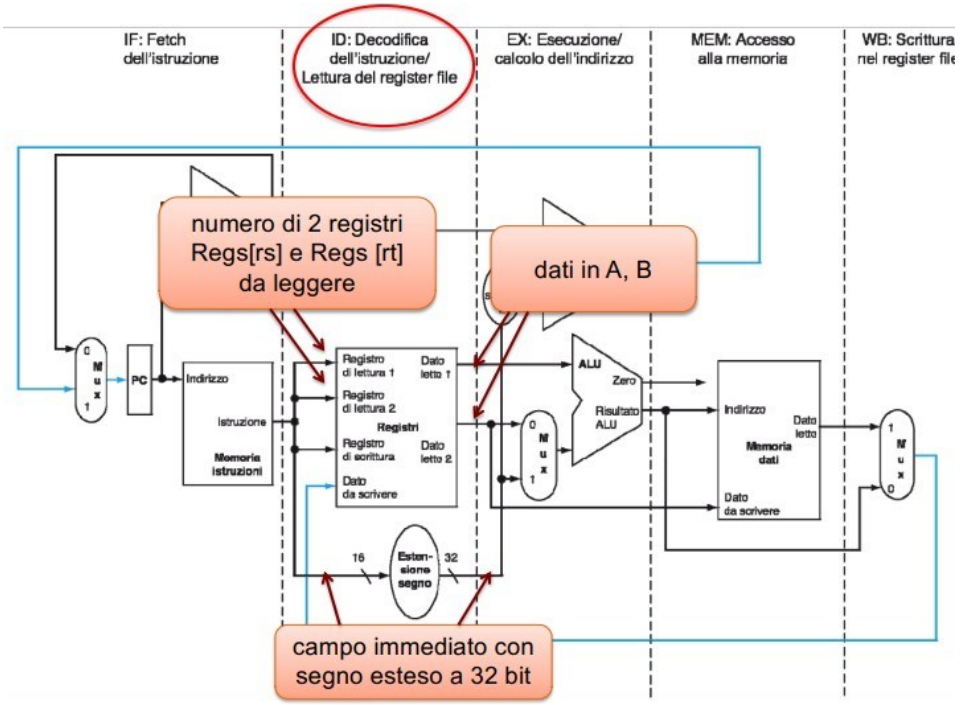
`j 45054` J

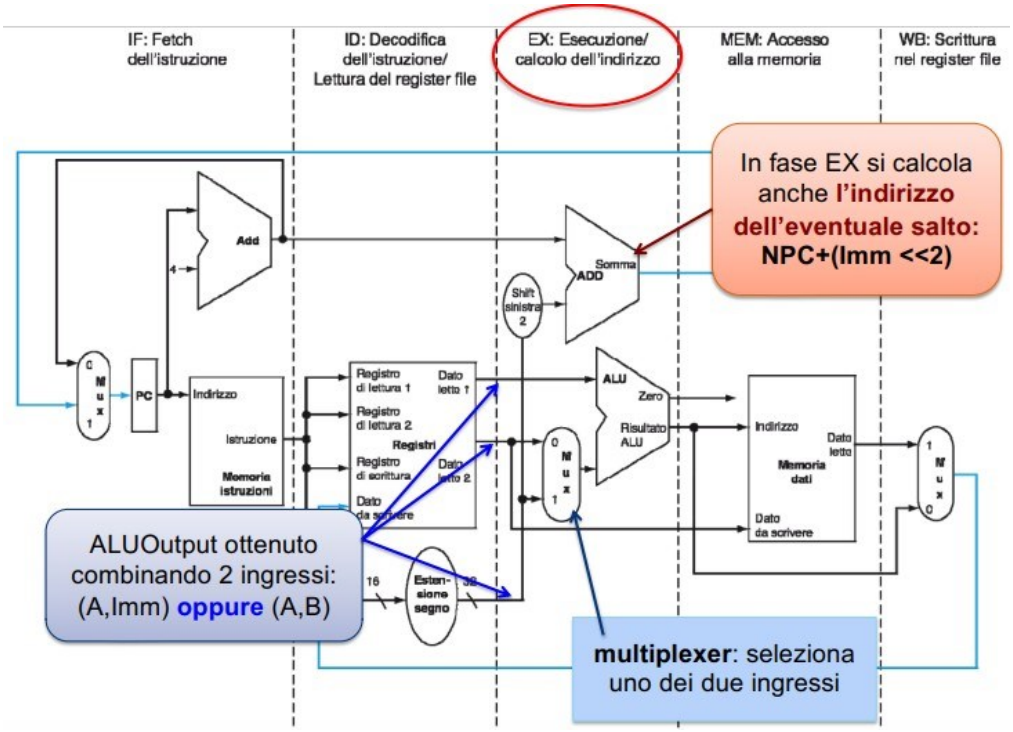
- IF
  - IR ← Mem[PC]
  - PC, NPC ← PC+4
- ID
  - Imm ← campo immediato di IR = 45054 *(esteso)*
- EX
  - Target ← NPC + (Imm <<2)
- MEM
  - PC ← Target
- WB
  - nessuna operazione

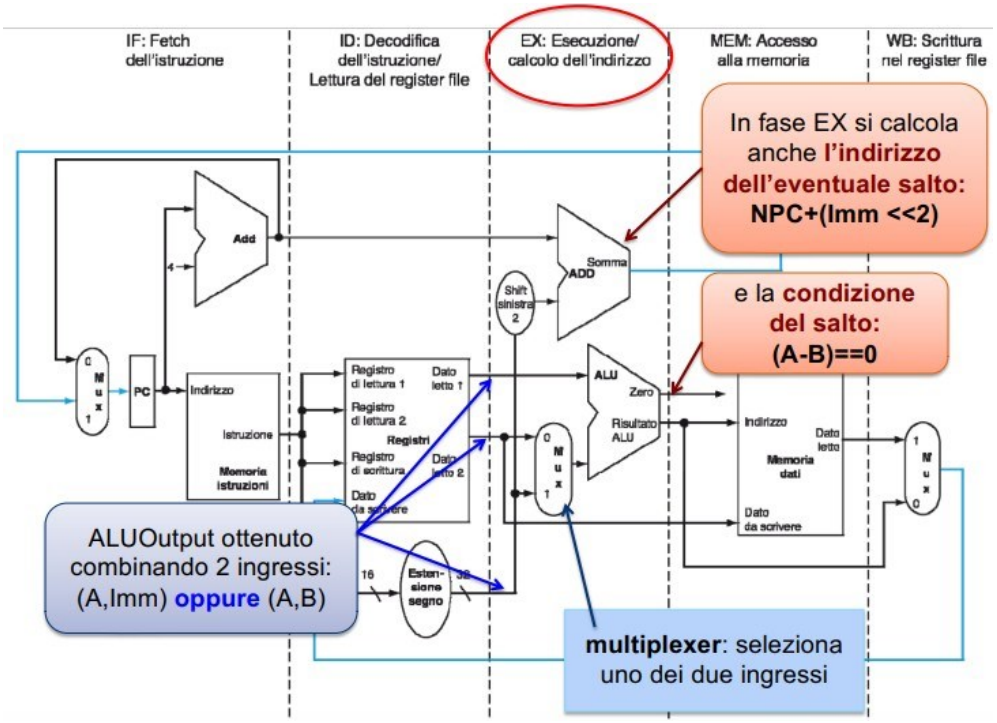
…non è nella fase MEM di j, ma nella fase IF che avviene al ciclo successivo a quello di EX di j

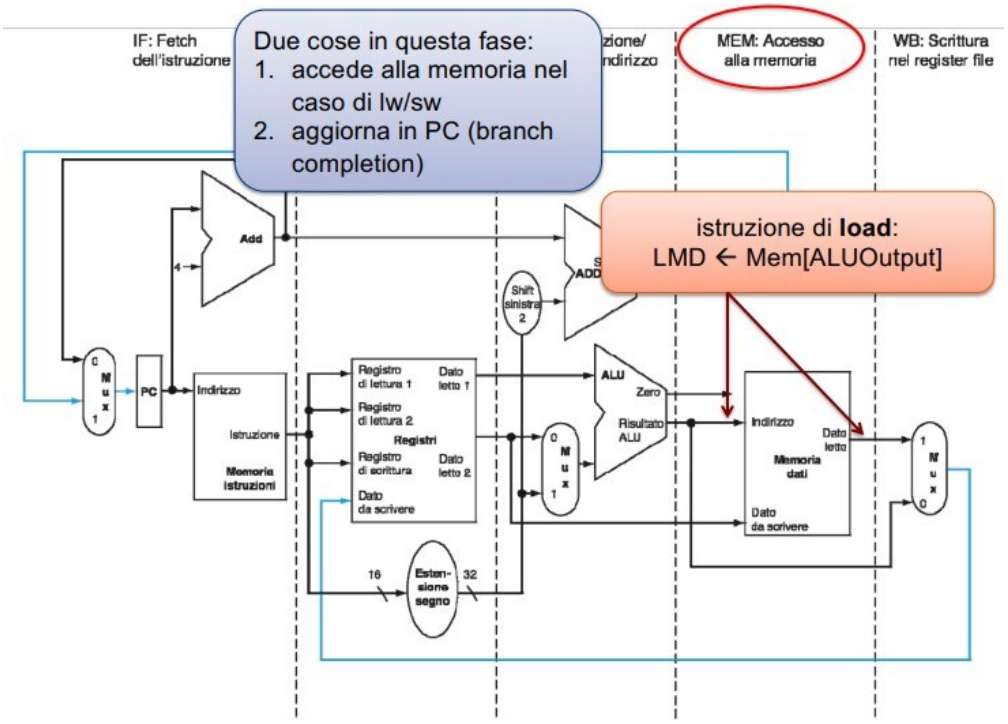
Schema di implementazione di MIPS:
- diverse unità funzionali (es. banco di registri, ALU, memoria…)
- loro connessioni
- manca unità di contro llo e linee di controllo

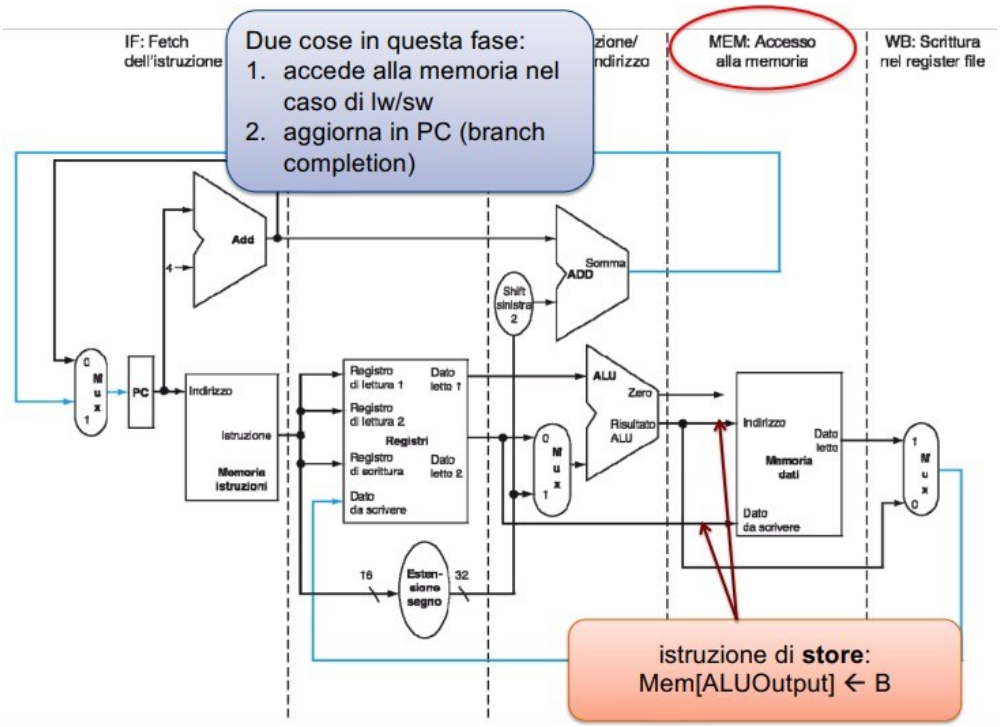


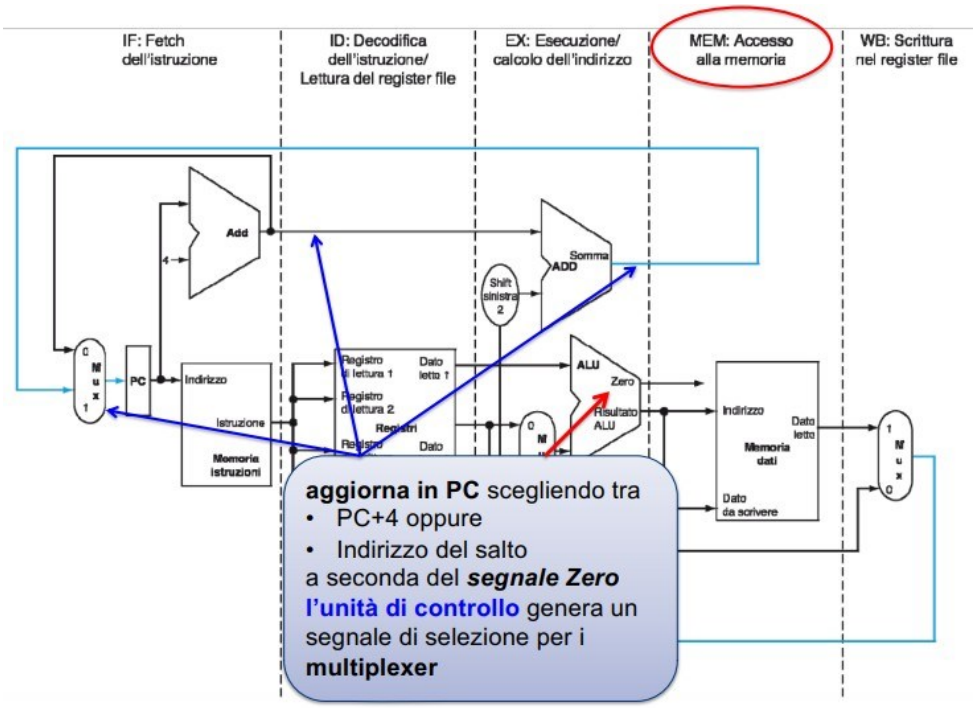


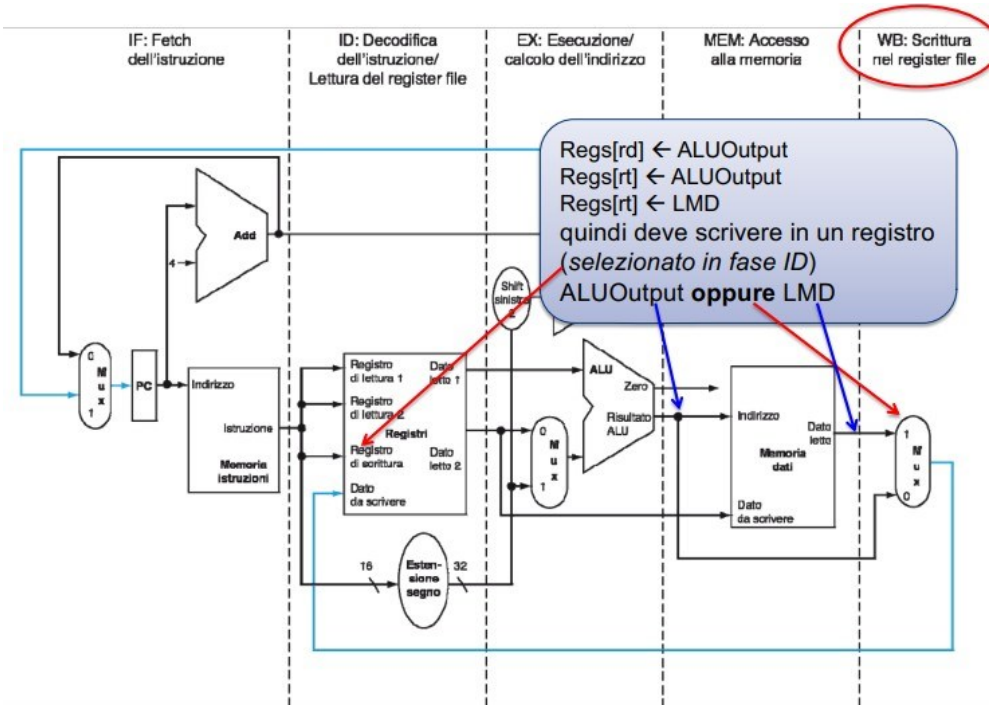

IF: Fetch dell'istruzione
ID: Decodifica dell'istruzione/ Lettura del register file
EX: Esecuzione/ calcolo dell'indirizzo
MEM: Accesso alla memoria
WB: Scrittura nel register file
In fase EX si calcola anche l'indirizzo dell'eventuale salto: NPC+(Imm <<2)
ALUOutput ottenuto combinando 2 ingressi: (A,Imm) oppure (A,B)
multiplexer: seleziona uno dei due ingressi

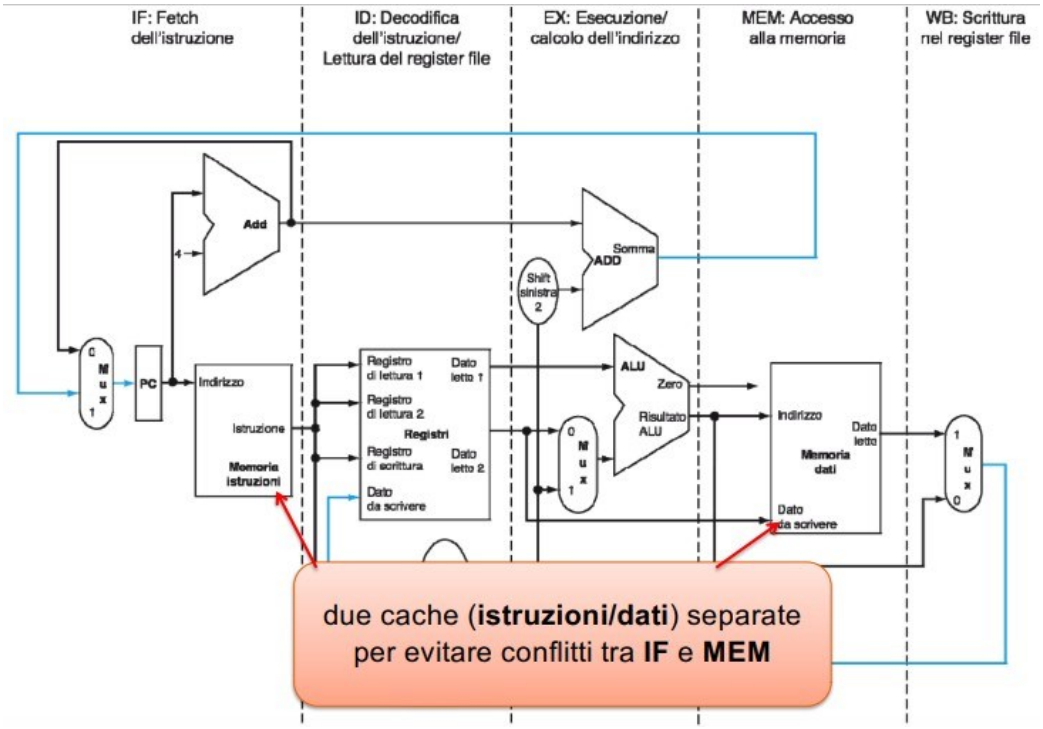

IF: Fetch dell'istruzione
ID: Decodifica dell'istruzione/ Lettura del register file
EX: Esecuzione/ calcolo dell'indirizzo
MEM: Accesso alla memoria
WB: Scrittura nel register file
In fase EX si calcola anche l'indirizzo dell'eventuale salto: NPC+(Imm <<2)
e la condizione del salto: (A-B)==0
ALUOutput ottenuto combinando 2 ingressi: (A,Imm) oppure (A,B)
multiplexer: seleziona uno dei due ingressi

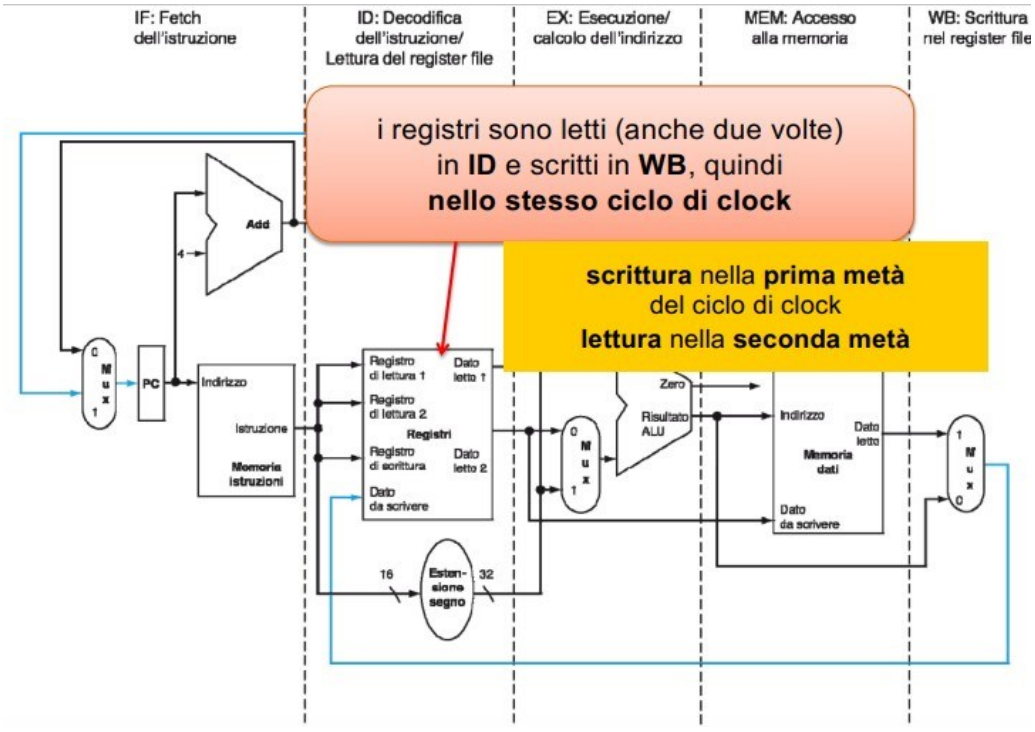

IF: Fetch dell'istruzione
MEM: Accesso alla memoria
WB: Scrittura nel register file
Due cose in questa fase:
1. accede alla memoria nel caso di lw/sw
2. aggiorna in PC (branch completion)
istruzione di **load**:
LMD ← Mem[ALUOutput]
Add
4
PC
Indirizzo
Istruzione
Memoria istruzioni
Registri
Registro di lettura 1
Registro di lettura 2
Registro di scrittura
Dato da scrivere
Dato letto 1
Dato letto 2
ALU
Zero
Risultato ALU
Shift sinistra 2
Somma ADD
Indirizzo
Dato letto
Memoria dati
Dato da scrivere
16
Estensione segno
32


IF: Fetch dell'istruzione
MEM: Accesso alla memoria
WB: Scrittura nel register file
Due cose in questa fase:
1. accede alla memoria nel caso di lw/sw
2. aggiorna in PC (branch completion)
istruzione di **store**:
Mem[ALUOutput] ← B
Add
4
PC
Indirizzo
Istruzione
Memoria istruzioni
Registri
Registro di lettura 1
Registro di lettura 2
Registro di scrittura
Dato da scrivere
Dato letto 1
Dato letto 2
ALU
Zero
Risultato ALU
Shift sinistra 2
Somma ADD
Indirizzo
Dato letto
Memoria dati
Dato da scrivere
16
Estensione segno
32


IF: Fetch dell'istruzione
ID: Decodifica dell'istruzione/ Lettura del register file
EX: Esecuzione/ calcolo dell'indirizzo
MEM: Accesso alla memoria
WB: Scrittura nel register file
Add
4
Somma ADD
Shift sinistra 2
M u x
PC
Indirizzo
Istruzione
Memoria istruzioni
Registro di lettura 1
Registro di lettura 2
Registri
Registro di scrittura
Dato da scrivere
Dato letto 1
Dato letto 2
ALU
Zero
Risultato ALU
Indirizzo
Dato letto
Memoria dati
Dato da scrivere
aggiorna in PC scegliendo tra
• PC+4 oppure
• Indirizzo del salto
a seconda del segnale Zero
l'unità di controllo genera un segnale di selezione per i multiplexer


IF: Fetch dell'istruzione
ID: Decodifica dell'istruzione/ Lettura del register file
EX: Esecuzione/ calcolo dell'indirizzo
MEM: Accesso alla memoria
WB: Scrittura nel register file
Regs[rd] ← ALUOutput
Regs[rt] ← ALUOutput
Regs[rt] ← LMD
quindi deve scrivere in un registro (selezionato in fase ID)
ALUOutput oppure LMD
Add
4
Shift sinistra 2
M u x
PC
Indirizzo
Istruzione
Memoria istruzioni
Registro di lettura 1
Registro di lettura 2
Registri
Registro di scrittura
Dato da scrivere
Dato letto 1
Dato letto 2
ALU
Zero
Risultato ALU
Indirizzo
Dato letto
Memoria dati
Dato da scrivere
16
Estensione segno
32

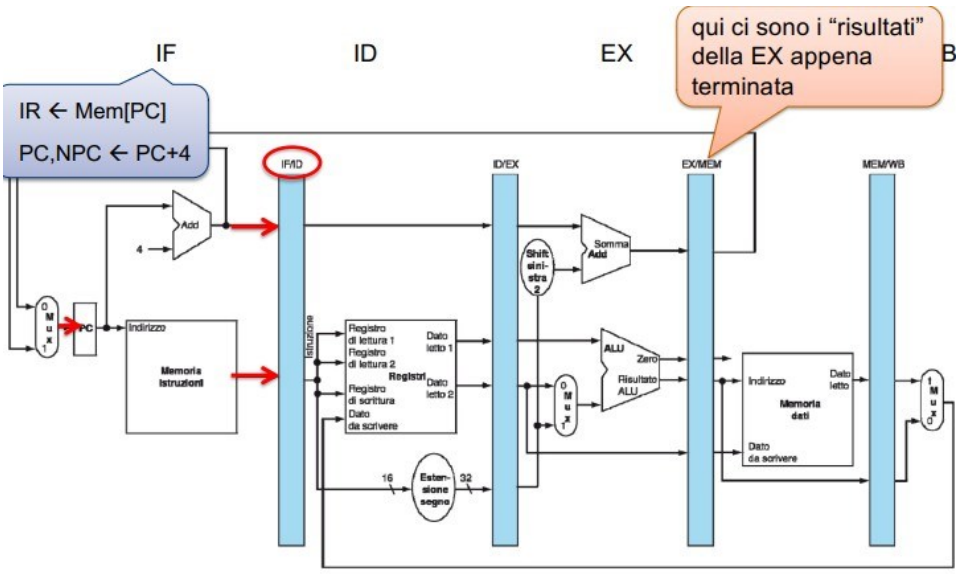

IF: Fetch dell'istruzione
ID: Decodifica dell'istruzione/ Lettura del register file
EX: Esecuzione/ calcolo dell'indirizzo
MEM: Accesso alla memoria
WB: Scrittura nel register file
Add
4
Somma ADD
Shift sinistra 2
M u x
PC
Indirizzo
Istruzione
Memoria istruzioni
Registro di lettura 1
Registro di lettura 2
Registri
Registro di scrittura
Dato da scrivere
Dato letto 1
Dato letto 2
ALU
Zero
Risultato ALU
Memoria dati
Dato letto
due cache (**istruzioni/dati**) separate per evitare conflitti tra **IF** e **MEM**


IF: Fetch dell'istruzione
ID: Decodifica dell'istruzione/ Lettura del register file
EX: Esecuzione/ calcolo dell'indirizzo
MEM: Accesso alla memoria
WB: Scrittura nel register file
i registri sono letti (anche due volte) in **ID** e scritti in **WB**, quindi **nello stesso ciclo di clock**
scrittura nella **prima metà** del ciclo di clock
lettura nella **seconda metà**
Add
4
M u x
PC
Indirizzo
Istruzione
Memoria istruzioni
Registro di lettura 1
Registro di lettura 2
Registri
Registro di scrittura
Dato da scrivere
Dato letto 1
Dato letto 2
Zero
Risultato ALU
Memoria dati
Dato letto
16
Estensione segno
32

Per la pipeline si ha:

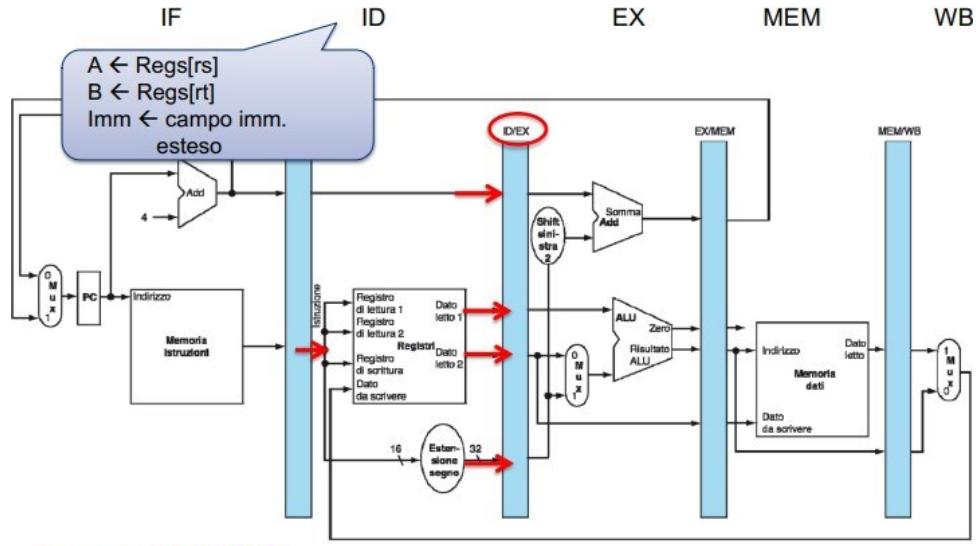
- 1 ciclo di clock per fase/stadio
- 1 ciclo di clock per IF, ID, EX, MEM, WB
- 1 istruzione in 5 cicli di clock
- 1 ciclo di clock completa 5 fasi di 5 istruzioni diverse (a regime)
- ogni ciclo di clock termina un'istruzione (a regime)

Similmente, il gruppo di registri delle pipeline, quindi i *pipeline registers (pipeline latches)*. I dati utili a fasi (anche non immediatamente) successive sono memorizzati nel registro successivo e i registri memorizzano sia dati che segnali di controllo (utili in seguito). In ogni istante registri diversi contengono dati relativi ad istruzioni diverse.



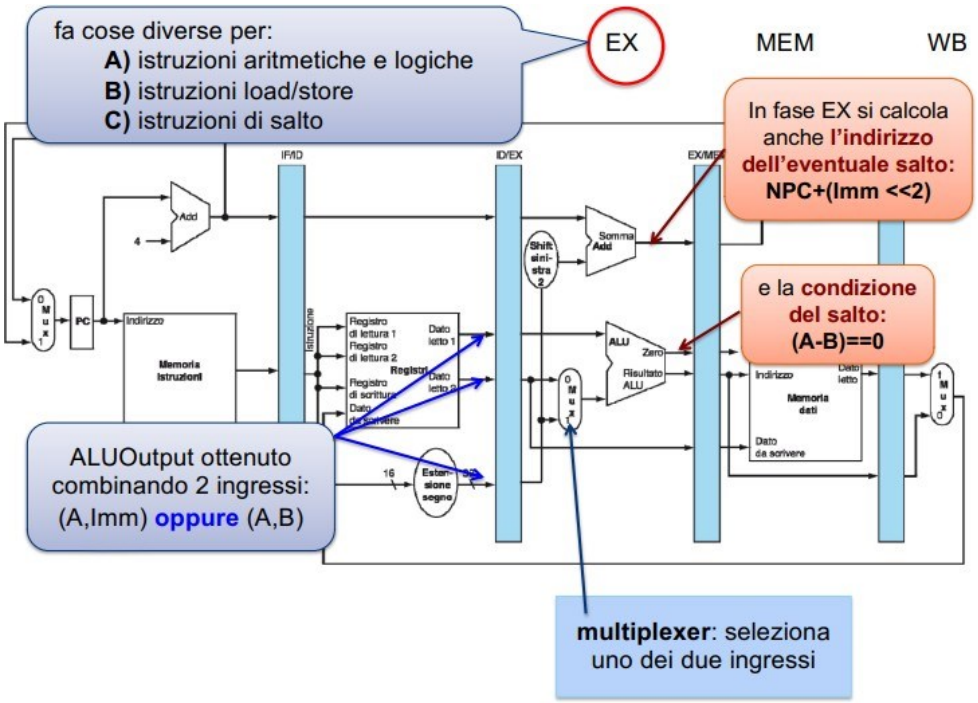
Banco registri IF/ID:

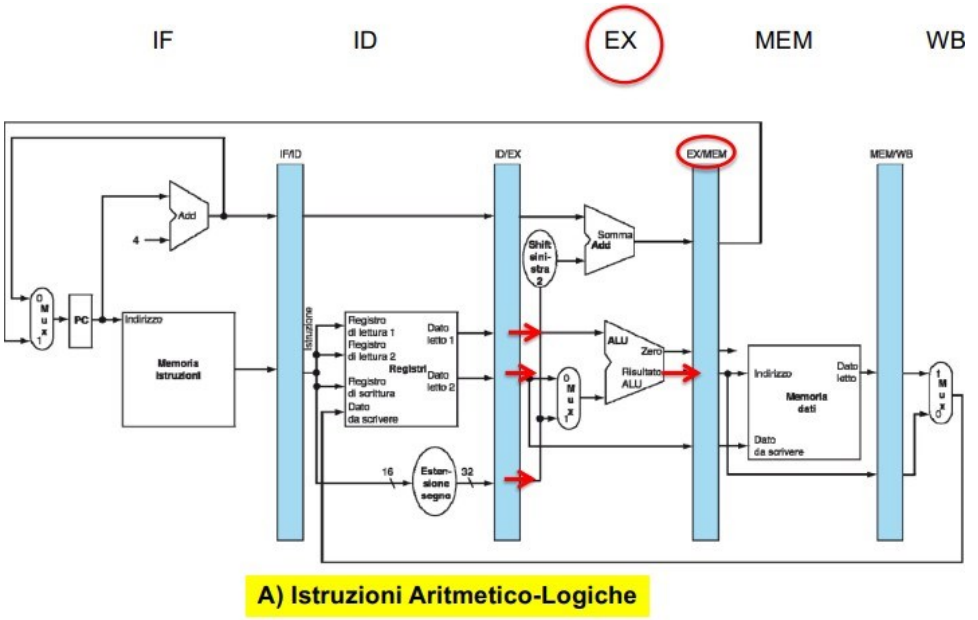
- `IF/ID.IR ← Mem[PC]`
- `IF/ID.NPC, PC ← if ((EX/MEM.IR==branch)&& EX/MEM.Cond) then {EX/MEM.Target} else {PC+4}`



Banco registri ID/EX:

- `ID/EX.IR ← IF/ID.IR;`
- `ID/EX.A ← Regs[IF/ID.IR[rs]]; ID/EX.B ← Regs[IF/ID.IR[rt]];`
- `ID/EX.NPC ←IF/ID.NPC;`
- `ID/EX.Imm ← sign-extend (IF/ID.IR[Immediate field]);`



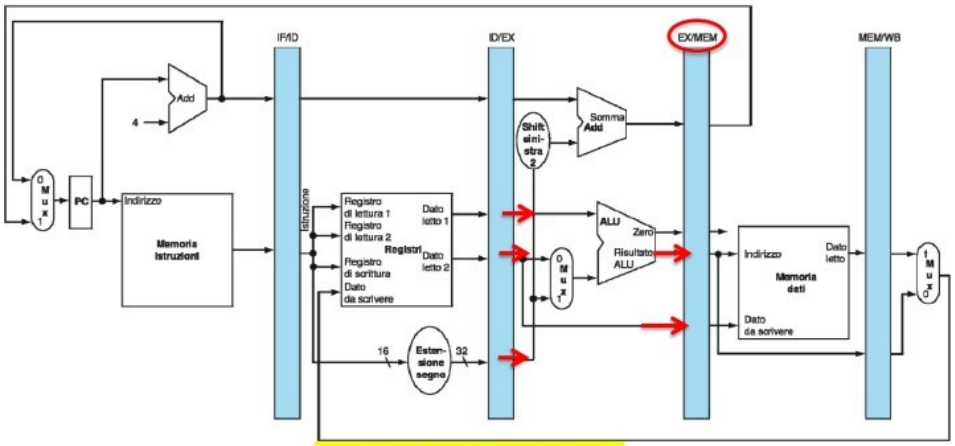


A) Istruzioni Aritmetico-Logiche

Banco registri EX/MEM:

- `EX/MEM.IR ← ID/EX.IR;`
- `EX/MEM.ALUOutput ← ID/EX.A op ID/EX.B;` (oppure `ID/EX.Imm`)

IF ID EX MEM WB

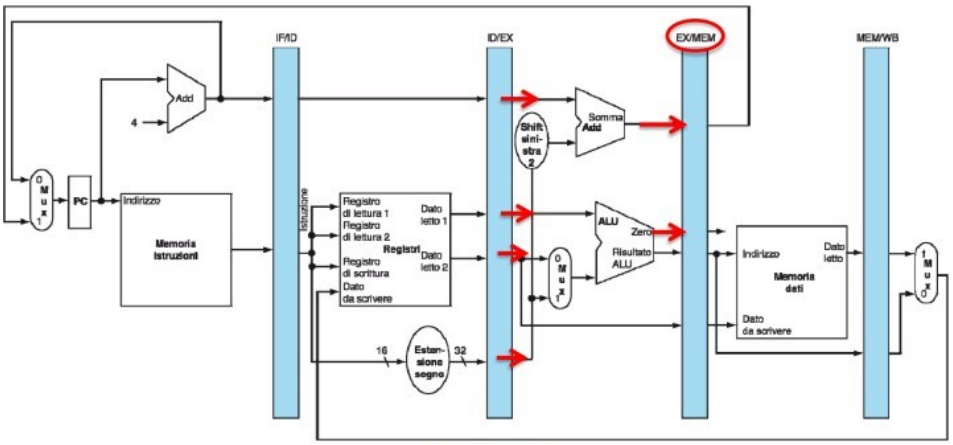


B) Istruzione Load/Store

Banco registri EX/MEM:

- `EX/MEM.IR ← ID/EX.IR;`
- `EX/MEM.ALUOutput ← ID/EX.A + ID/EX.Imm;`
- `EX/MEM.B ← ID.EX.B` (servirà per fare lo store dopo)

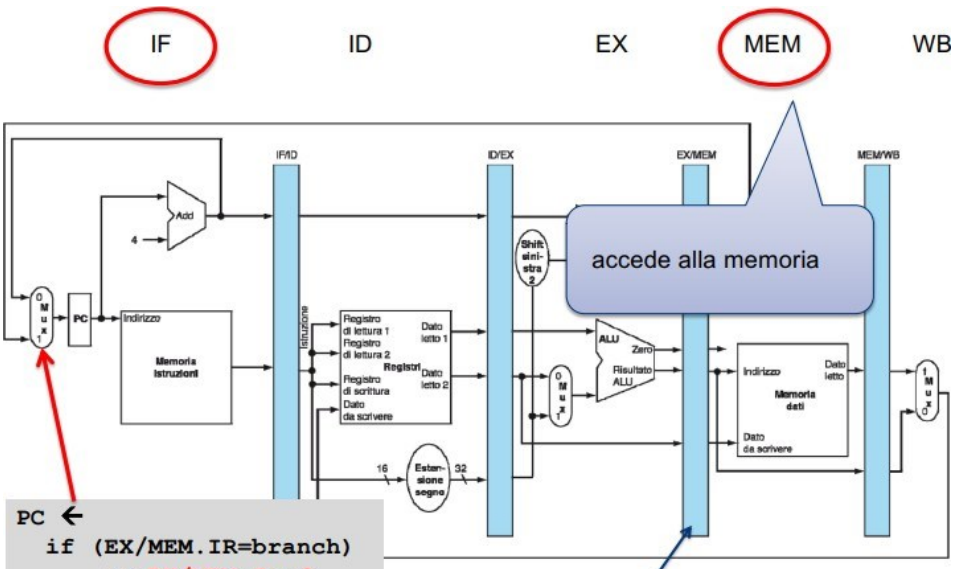
IF ID EX MEM WB



C) Salti

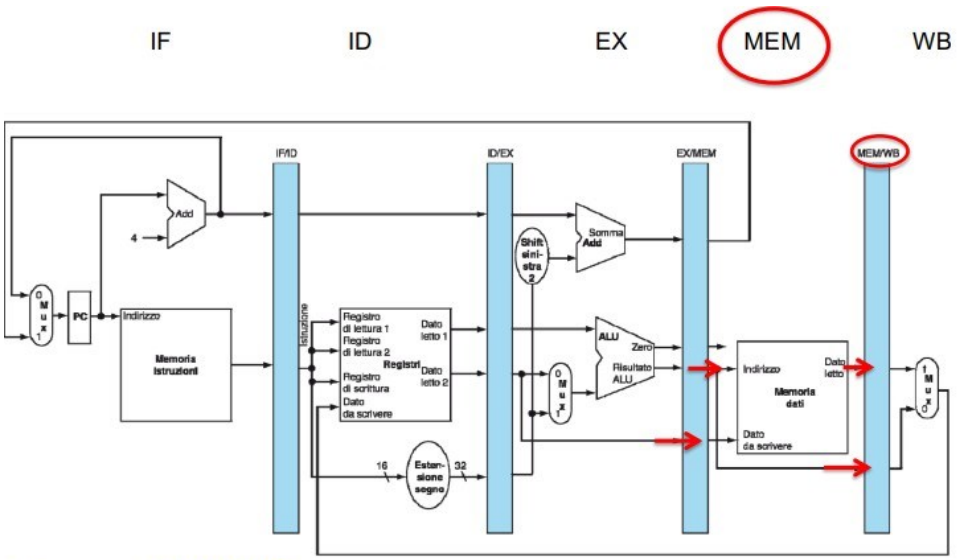
Banco registri EX/MEM:

- `EX/MEM.IR ← ID/EX.IR`
- `EX/MEM.Target ← ID/EX.NPC + (ID/EX.Imm<<2);`
- `EX/MEM.Cond ← Zero` (cioè `ID/EX.A - ID.EX.B`)



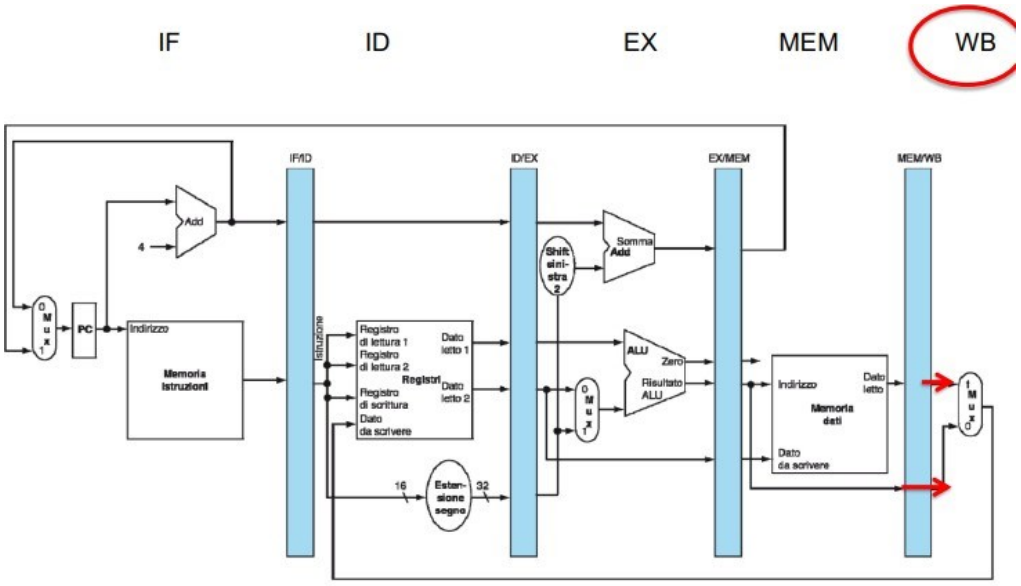
Banco registri EX/MEM:

- **EX/Mem.IR**
- **EX/MEM.Target**
- **EX/MEM.Cond**



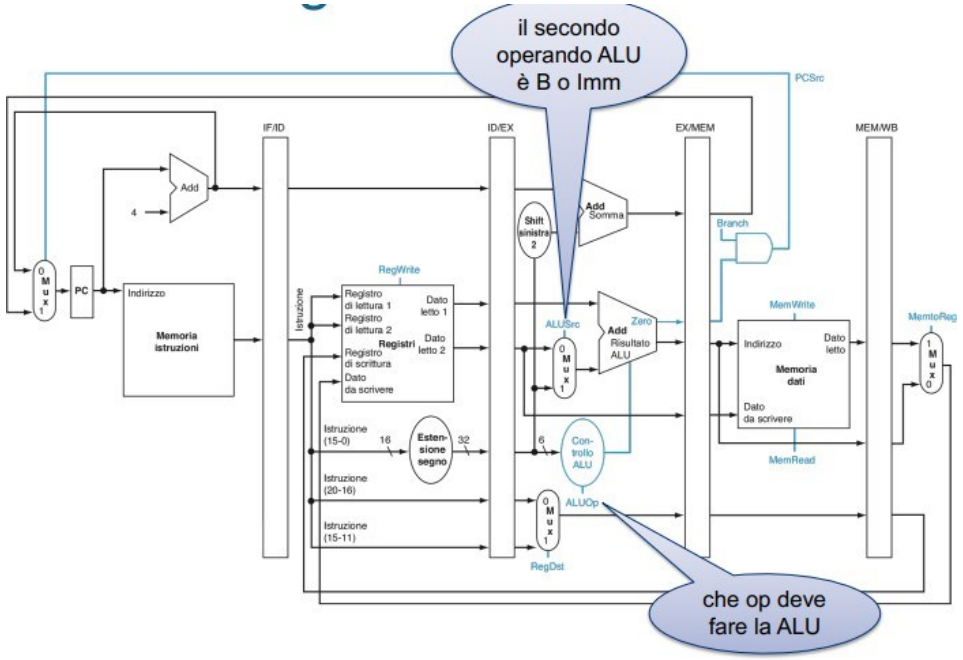
Banco registri MEM/WB:

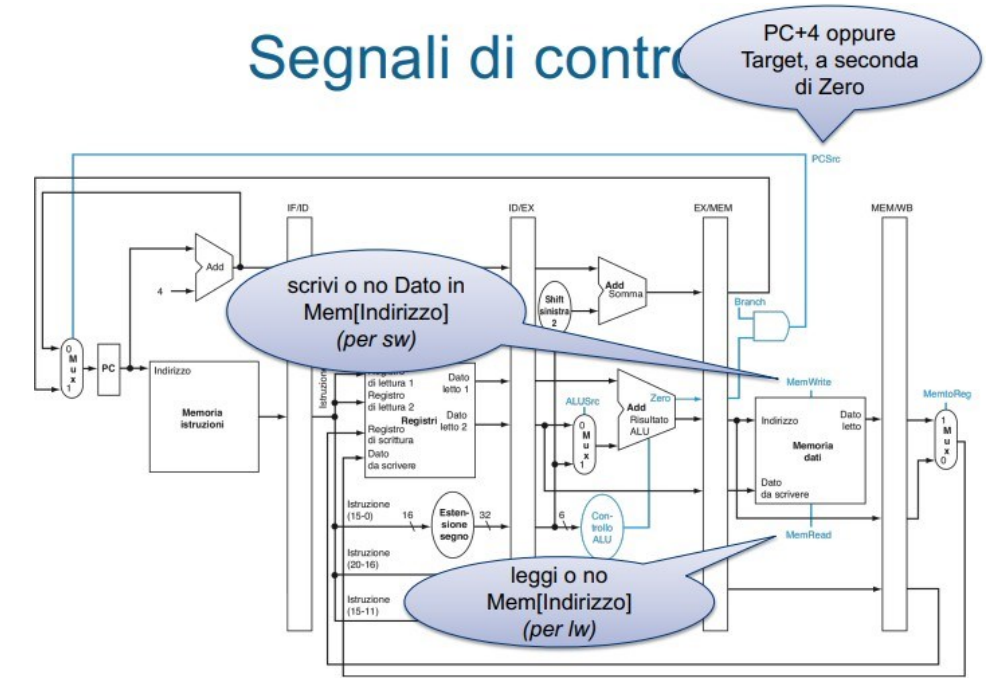
- **MEM/WB.IR ← EX/MEM.IR;**
- **MEM/WB.ALUOutput ← EX/MEM.ALUOutput** per operazioni aritmetico-logiche
- **MEM/WB.LMD ← Mem[EX/MEM.ALUOutput];** per load
- **Mem[EX/MEM.ALUOutput] ← EX/MEM.B;** per store

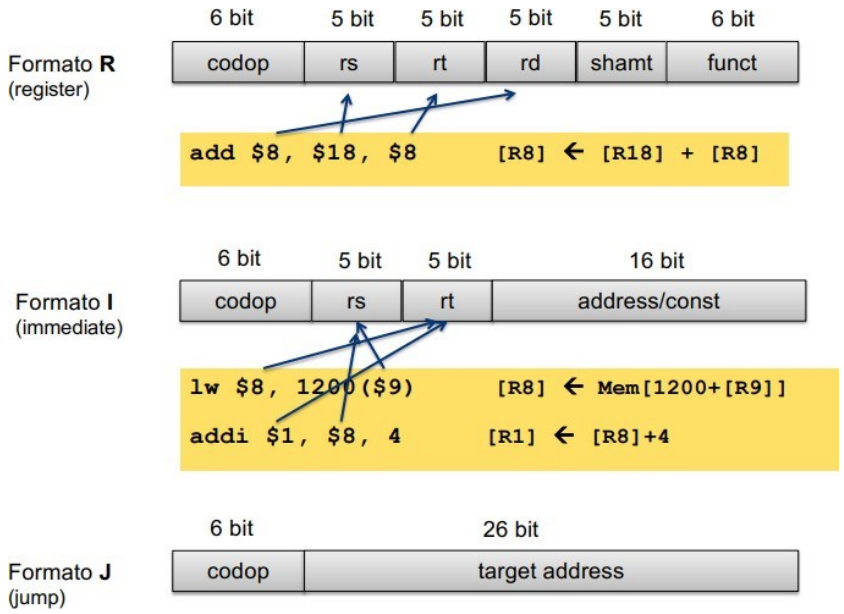


- `Regs[MEM/WB.IR[rd]]` (opp `IR[rt]`) ← `MEM/WB.ALUOutput`; per istr arit-log
- `Regs[MEM/WB.IR[rt]` ← `MEM/WB.LMD`; per load

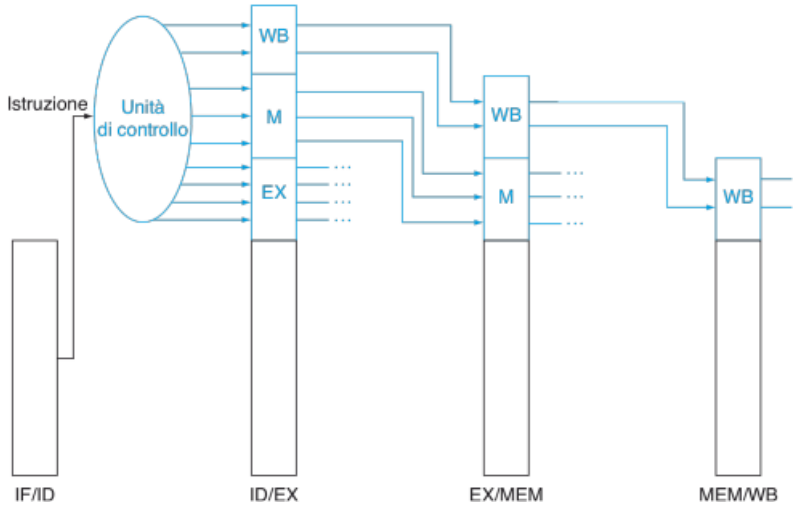
Analizziamo nel dettaglio i segnali di controllo:

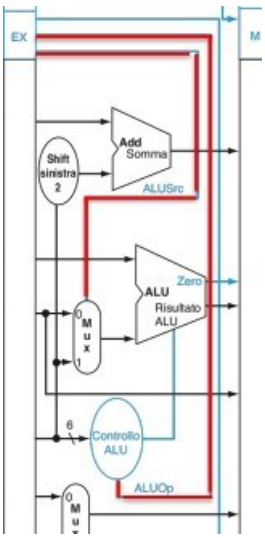


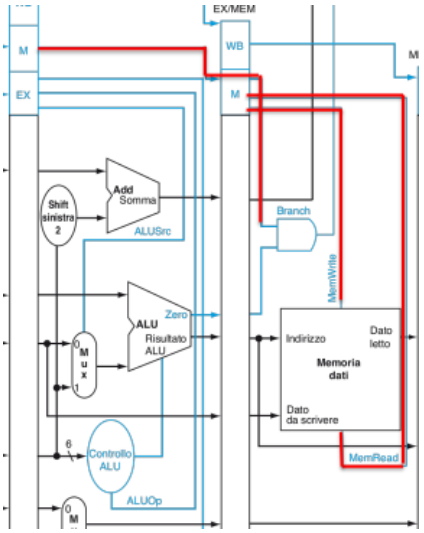

Segnali di contro
PC+4 oppure Target, a seconda di Zero
scrivi o no Dato in Mem[Indirizzo] *(per sw)*
leggi o no Mem[Indirizzo] *(per lw)*
PCSrc
IF/ID
ID/EX
EX/MEM
MEM/WB
Add
Somma
Shift sinistra 2
Branch
MemWrite
MemtoReg
MemRead
ALUSrc
ALUOp
RegWrite
RegDst
PC
Indirizzo
Memoria istruzioni
Registri
Registro di lettura 1
Registro di lettura 2
Registro di scrittura
Dato da scrivere
Dato letto 1
Dato letto 2
Istruzione
Istruzione (15-0)
Istruzione (20-16)
Istruzione (15-11)
Estensione segno
Controllo ALU
Zero
Risultato ALU
Memoria dati
Dato letto
va fatta una scrittura (finale) su registro o no
il **dato da scrivere** nel registro viene da ALU *(per add)* o da Memoria dati *(per lw)*
il **num del registro di scrittura** proviene dal campo **rt** (istr I bit 20-16) o dal campo **rd** (isrt R bit 15-11)

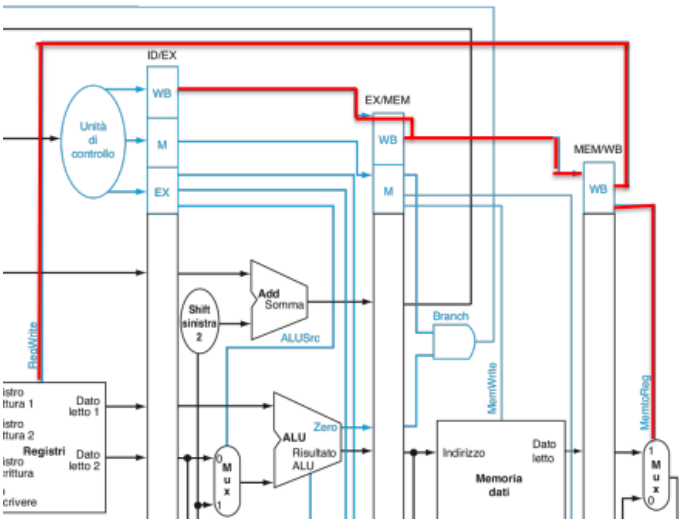


Per l'unità di controllo, le fasi IF e ID non dipendono dai valori dei segnali di controllo; in fase ID si possono calcolare i segnali corretti per le fasi successive e i segnali sono calcolati in ID per poi essere propagati attraverso i registri di pipeline.









Nella Pipeline MIPS è possibile individuare tutte le dipendenze dai dati nella fase ID. Se si rileva una dipendenza dai dati per una istruzione, questa va in stallo prima di essere rilasciata (issued, cioè quando una passa dalla fase ID a quella EX). Inoltre, sempre nella fase ID, è possibile determinare che tipo di data forwarding adottare per evitare lo stallo ed anche predisporre gli opportuni segnali di controllo.
Esempio: realizziamo un forwarding nella fase EX per una dipendenza di tipo RAW (Read After Write) con sorgente che proviene da una istruzione load (load interlock).
Vediamo l'esempio di una dipendenza RAW per le istruzioni Load:

Possibili casi

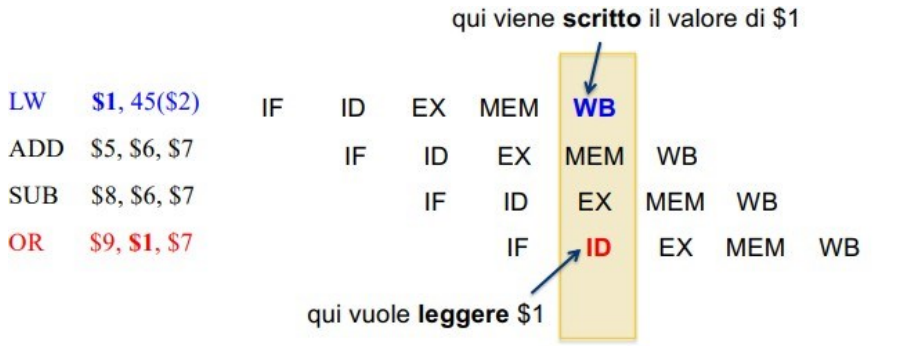
Situazione	Esempio di codice	Azione

Nessuna dipendenza	LW **$1**, 45($2)
	ADD $5, $6, $7
	SUB $8, $6, $7
	OR $9, $6, $7
		Non occorre fare nulla

	LW **$1**, 45($2)
	ADD $5, **$1**, $7
	SUB $8, $6, $7
	OR $9, $6, $7	

	LW **$1**, 45($2)
	ADD $5, $6, $7
	SUB $8, **$1**, $7
	OR $9, $6, $7	

	LW **$1**, 45($2)
	ADD $5, $6, $7
	SUB $8, $6, $7
	OR $9, **$1**, $7	



la scrittura avviene nella **prima metà** del ciclo di clock,
la lettura nella **seconda metà**

nessun problema

Possibili casi

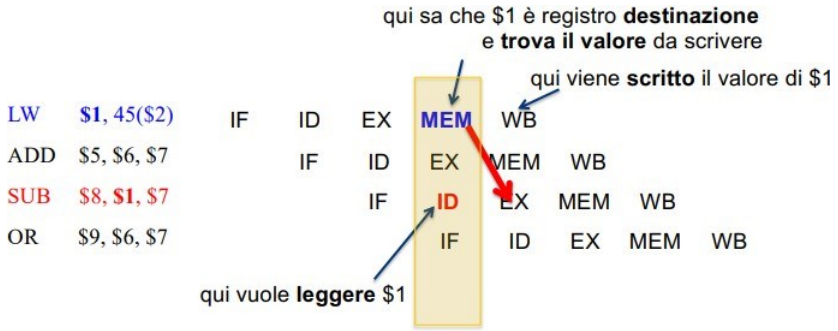
Situazione	Esempio di codice	Azione

Nessuna dipendenza	LW **$1**, 45($2)
	ADD $5, $6, $7
	SUB $8, $6, $7
	OR $9, $6, $7
		Non occorre fare nulla

	LW **$1**, 45($2)
	ADD $5, **$1**, $7
	SUB $8, $6, $7
	OR $9, $6, $7	

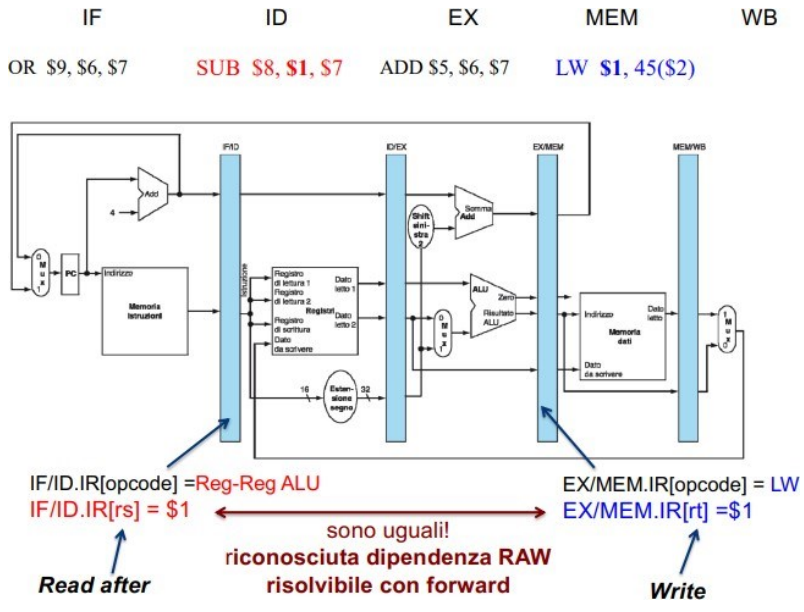
	LW **$1**, 45($2)
	ADD $5, $6, $7
	SUB $8, **$1**, $7
	OR $9, $6, $7	

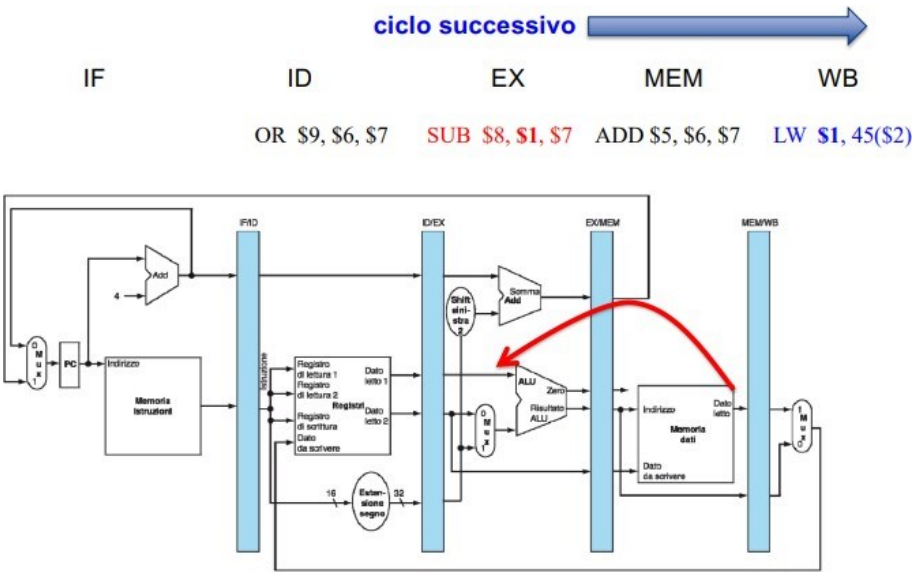
Dipendenza con accessi in ordine	LW **$1**, 45($2)
	ADD $5, $6, $7
	SUB $8, $6, $7
	OR $9, **$1**, $7
		Non occorre fare nulla perché la lettura di $1 in OR avviene dopo la scrittura del dato caricato



in questo ciclo si **identifica la dipendenza** RAW

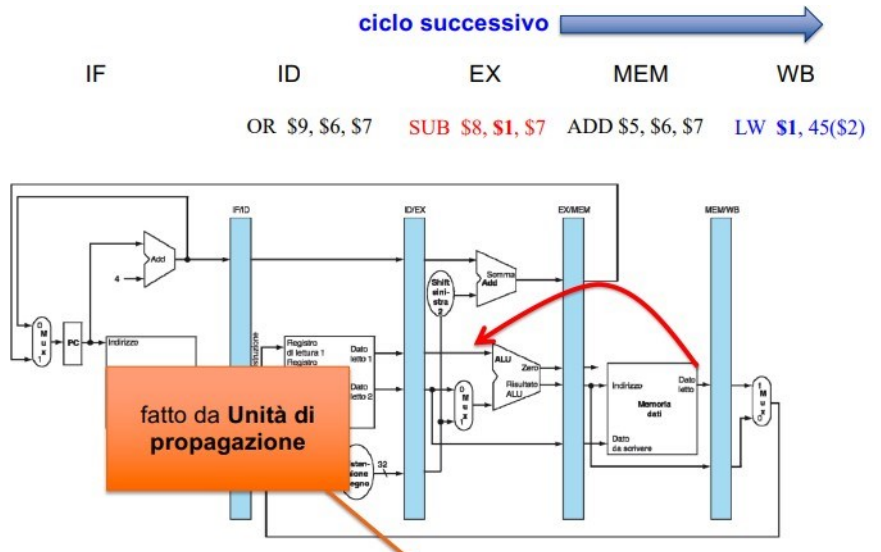
il data forwarding
(il contenuto della memoria è inviato alla ALU prima della scrittura nel registro $1)
evita lo stallo





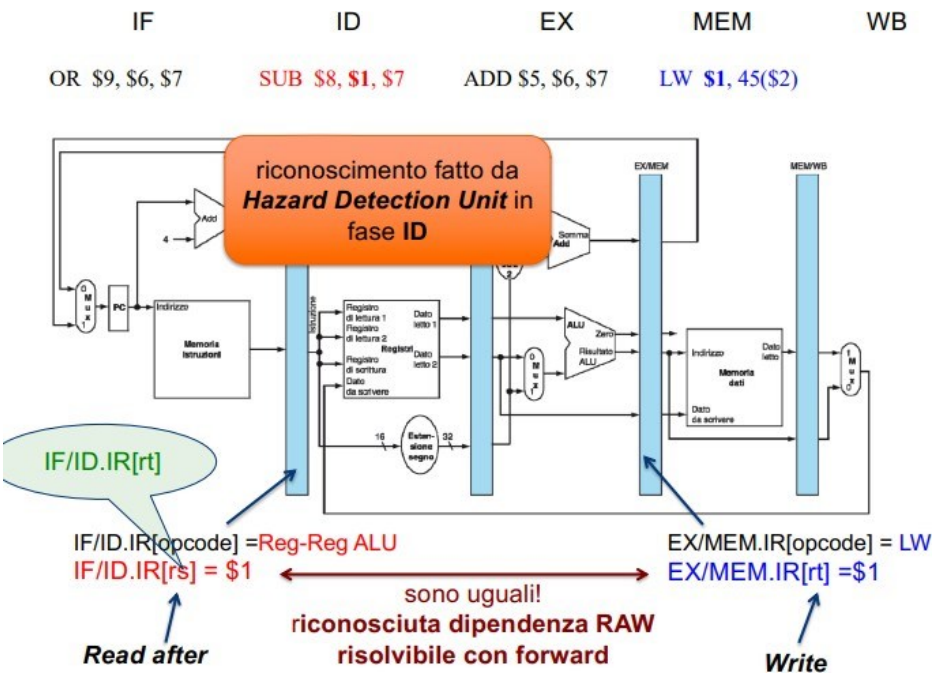
Effettua il forward da Load a Reg-Reg-ALU

- se MEM/WB.IR[rt] == ID/EX.IR[rs] allora **manda MEM/WB.LMD a Top ALU Input**



Effettua il forward da Load a Reg-Reg-ALU

- se MEM/WB.IR[rt] == ID/EX.IR[rs] allora **manda MEM/WB.LMD a Top ALU Input**
- se MEM/WB.IR[rt] == ID/EX.IR[**rt**] allora **manda MEM/WB.LMD a Bottom ALU Input**
- istruzioni diverse hanno condizioni diverse



Possibili casi

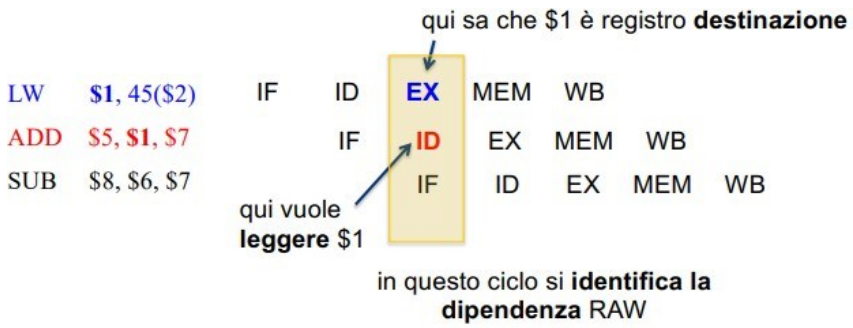
Situazione	Esempio di codice	Azione

Nessuna dipendenza	LW **$1**, 45($2)
	ADD $5, $6, $7
	SUB $8, $6, $7
	OR $9, $6, $7
		Non occorre fare nulla

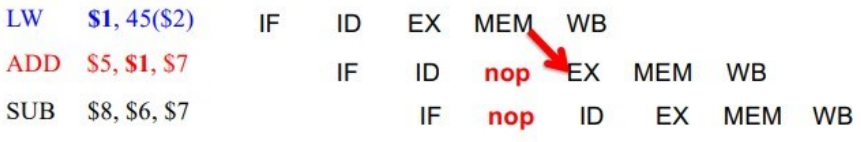
	LW **$1**, 45($2)
	ADD $5, **$1**, $7
	SUB $8, $6, $7
	OR $9, $6, $7	

Dipendenza **risolvibile con un forwarding**	LW **$1**, 45($2)
	ADD $5, $6, $7
	SUB $8, **$1**, $7
	OR $9, $6, $7
		Opportuni comparatori rilevano l'uso di $1 in SUB e inoltrano il risultato della load alla ALU in tempo per la fase EX di SUB

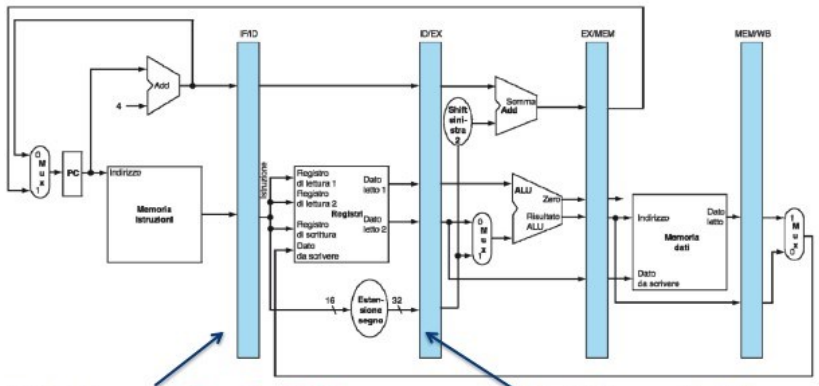
Dipendenza con accessi in ordine	LW **$1**, 45($2)
	ADD $5, $6, $7
	SUB $8, $6, $7
	OR $9, **$1**, $7
		Non occorre fare nulla perché la lettura di $1 in OR avviene dopo la scrittura del dato caricato

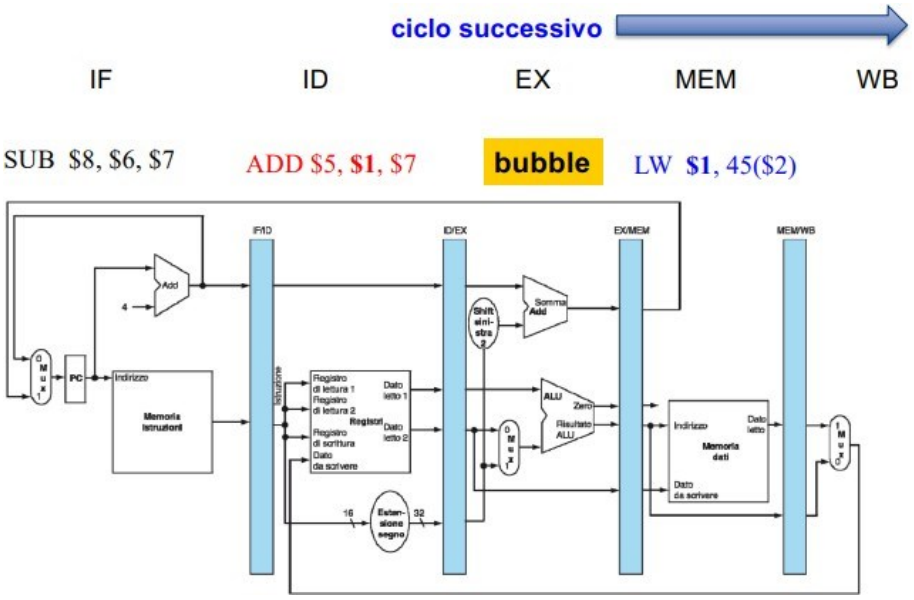


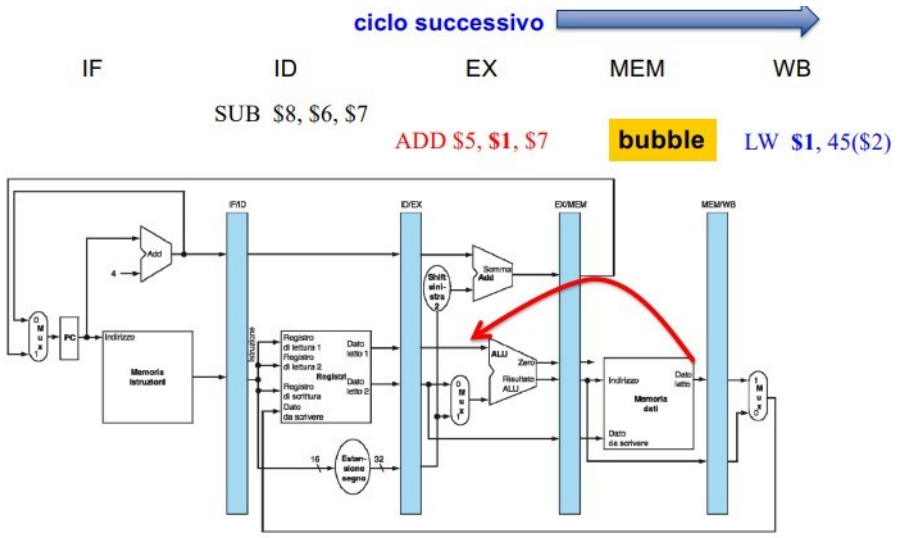
in questo ciclo si **identifica la dipendenza** RAW



1 ciclo di stallo e poi data forwarding







- se MEM/WB.IR[rt] == ID/EX.IR[rs] allora manda MEM/WB.LMD a Top ALU Input

Possibili casi

Situazione	Esempio di codice	Azione

Nessuna dipendenza	LW $1, 45($2)
	ADD $5, $6, $7
	SUB $8, $6, $7
	OR $9, $6, $7
		Non occorre fare nulla

Dipendenza che **richiede uno stallo**	LW $1, 45($2)
	ADD $5, $1, $7
	SUB $8, $6, $7
	OR $9, $6, $7
		Opportuni comparatori rilevano l'uso di $1 in ADD ed evitano il rilascio di ADD (quindi stallo)

Dipendenza risolvibile con un forwarding	LW $1, 45($2)
	ADD $5, $6, $7
	SUB $8, $1, $7
	OR $9, $6, $7
		Opportuni comparatori rilevano l'uso di $1 in SUB e inoltrano il risultato della load alla ALU in tempo per la fase EX di SUB

Dipendenza con accessi in ordine	LW $1, 45($2)
	ADD $5, $6, $7
	SUB $8, $6, $7
	OR $9, $1, $7
		Non occorre fare nulla perché la lettura di $1 in OR avviene dopo la scrittura del dato caricato

Condizioni per riconoscere le dipendenze da una LOAD che richiedono uno stallo

Opcode (ID/EX)	Opcode (IF/ID)	Matching operand fields
Load	R-R ALU	ID/EX.IR[rt] == IF/ID.IR[rs]
Load	R-R ALU	ID/EX.IR[rt] == IF/ID.IR[rt]
Load	Load, Store, Imm ALU, branch	ID/EX.IR[rt] == IF/ID.IR[rs]

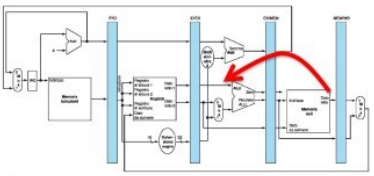
Load in fase EX — Istruzione successiva in fase ID — registro di **scrittura** di Load — registro di **lettura** di Istruzione successiva

condizioni verificate da **Hazard Detection Unit**

Condizioni per decidere come effettuare il forwarding

Tutti i **dati** su cui effettuare il forwarding

- **provengono**:
  - dalla memoria dati
  - dall'output della ALU
  - quindi stanno in registro MEM/WB opp EX/MEM
- e sono **diretti** verso:
  - l'input della ALU
  - (l'input della memoria dati)
  - quindi verso registri ID/EX (e EX/MEM)



il **campo destinazione di IR** (cioe' quale registro viene ***scritto*** dall'istruzione) in EX/MEM e MEM/WB

Quindi occorre confrontare:

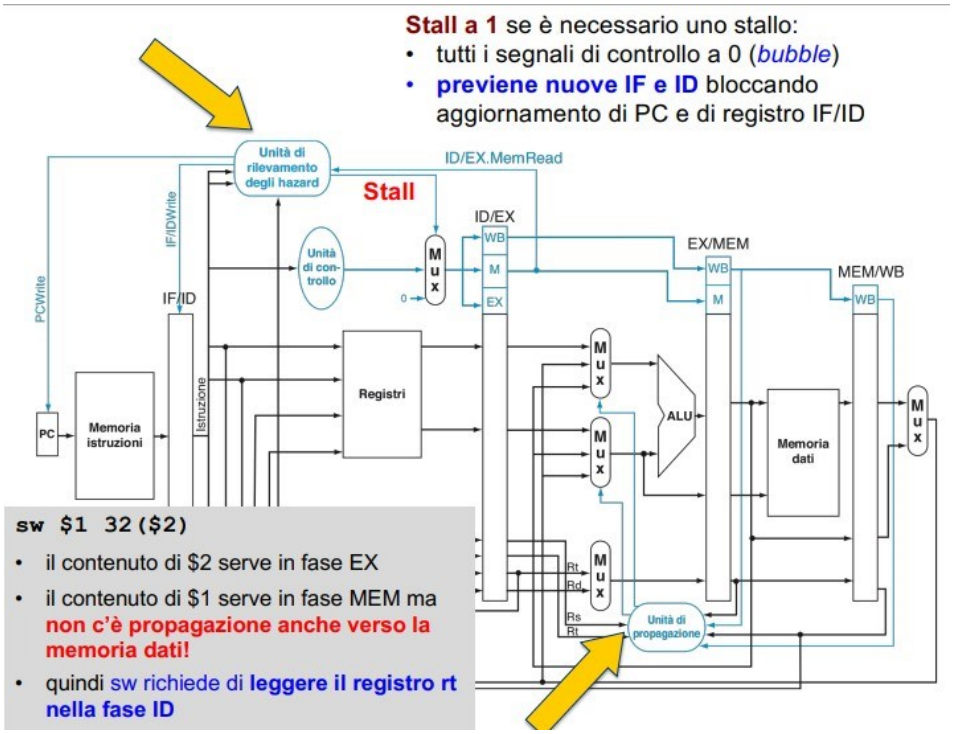
se sono uguali va fatta la propagazione (dall'Unità di Propagazione)

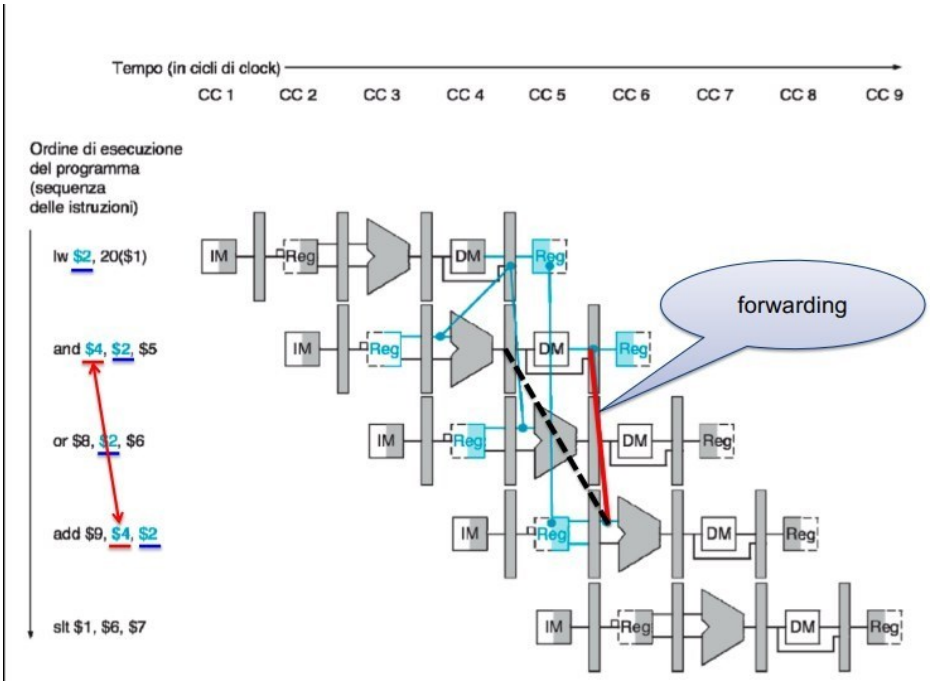
con i **campi sorgente di IR** (cioè quale registro viene ***letto***) in ID/EX e EX/MEM

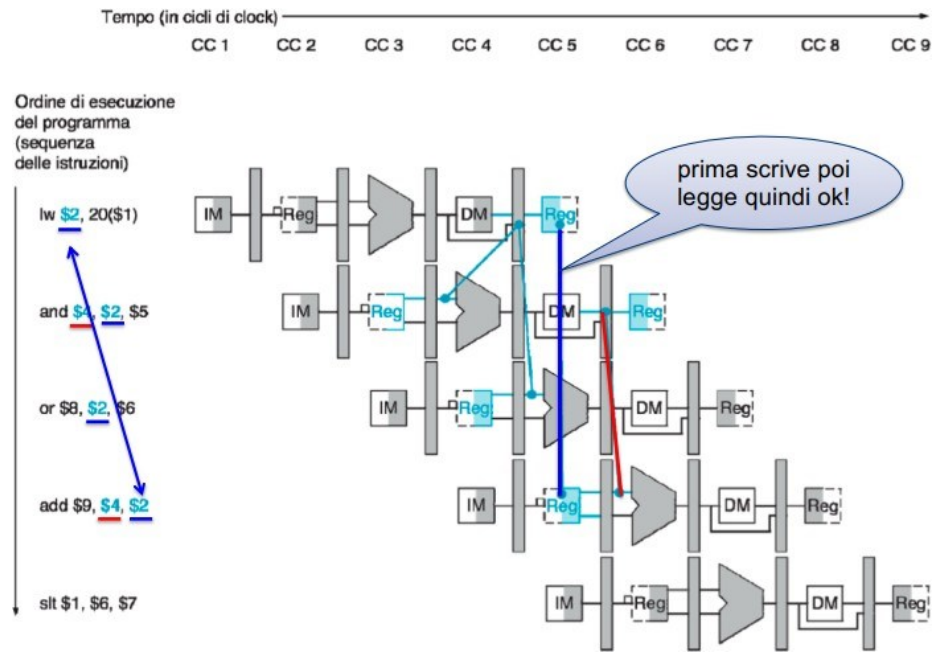
Pipeline (MIPS)

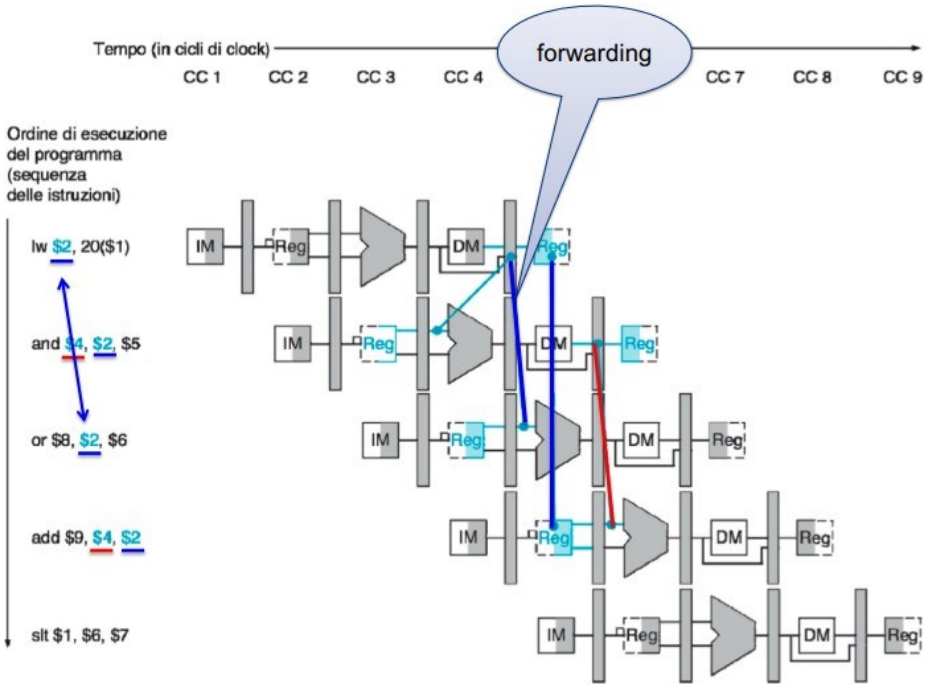
condizioni per la sola propagazione all'input di ALU

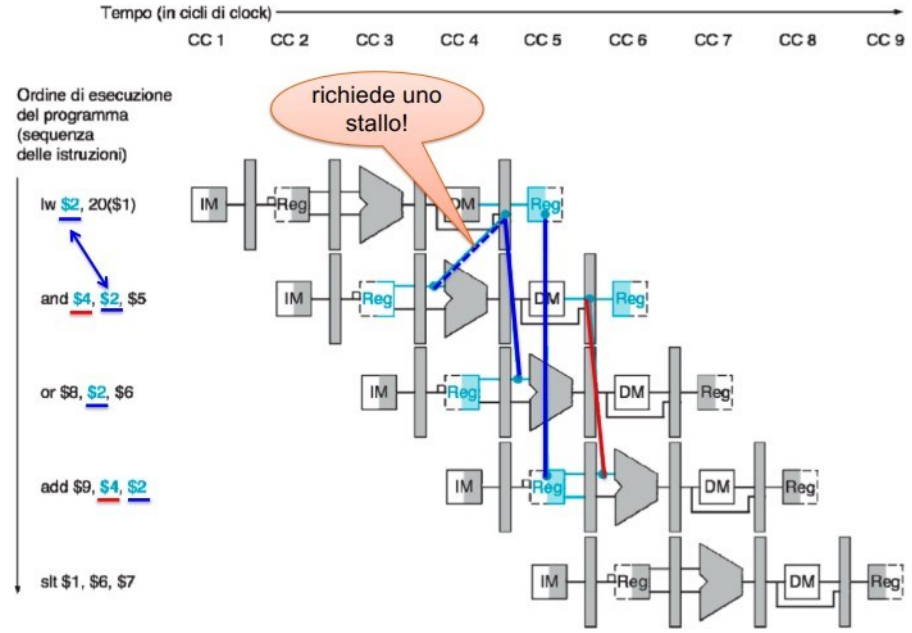
Pipeline register containing source instruction	Opcode of source instruction	Pipeline register containing destination instruction	Opcode of destination instruction	Destination of the forwarded result	Comparison (if equal then forward)
EX/MEM	Register-register ALU	ID/EX	Register-register ALU, ALU immediate, load, store, branch	Top ALU input	EX/MEM.IR[rd] == ID/EX.IR[rs]
EX/MEM	Register-register ALU	ID/EX	Register-register ALU	Bottom ALU input	EX/MEM.IR[rd] == ID/EX.IR[rt]
MEM/WB	Register-register ALU	ID/EX	Register-register ALU, ALU immediate, load, store, branch	Top ALU input	MEM/WB.IR[rd] == ID/EX.IR[rs]
MEM/WB	Register-register ALU	ID/EX	Register-register ALU	Bottom ALU input	MEM/WB.IR[rd] == ID/EX.IR[rt]
EX/MEM	ALU immediate	ID/EX	Register-register ALU, ALU immediate, load, store, branch	Top ALU input	EX/MEM.IR[rt] == ID/EX.IR[rs]
EX/MEM	ALU immediate	ID/EX	Register-register ALU	Bottom ALU input	EX/MEM.IR[rt] == ID/EX.IR[rt]
MEM/WB	ALU immediate	ID/EX	Register-register ALU, ALU immediate, load, store, branch	Top ALU input	MEM/WB.IR[rt] == ID/EX.IR[rs]
MEM/WB	ALU immediate	ID/EX	Register-register ALU	Bottom ALU input	MEM/WB.IR[rt] == ID/EX.IR[rt]
MEM/WB	Load	ID/EX	Register-register ALU, ALU immediate, load, store, branch	Top ALU input	MEM/WB.IR[rt] == ID/EX.IR[rs]
MEM/WB	Load	ID/EX	Register-register ALU	Bottom ALU input	MEM/WB.IR[rt] == ID/EX.IR[rt]

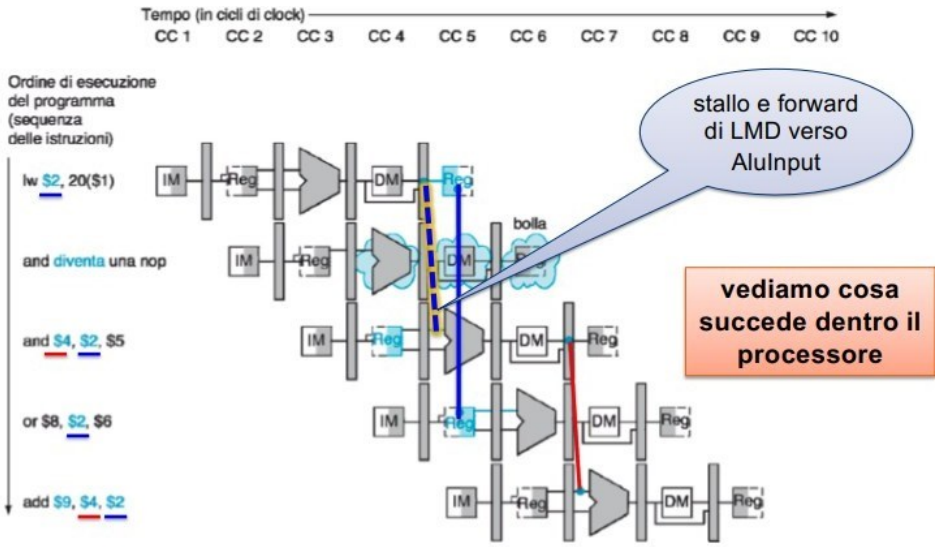

Stall a 1 se è necessario uno stallo:
• tutti i segnali di controllo a 0 (bubble)
• previene nuove IF e ID bloccando aggiornamento di PC e di registro IF/ID
Unità di rilevamento degli hazard
Stall
ID/EX.MemRead
IF/IDWrite
PCWrite
Unità di controllo
ID/EX
EX/MEM
MEM/WB
WB
M
EX
IF/ID
Istruzione
PC
Memoria istruzioni
Registri
ALU
Memoria dati
Rt
Rd
Rs
Rt
Unità di propagazione
sw $1 32($2)
• il contenuto di $2 serve in fase EX
• il contenuto di $1 serve in fase MEM ma non c'è propagazione anche verso la memoria dati!
• quindi sw richiede di leggere il registro rt nella fase ID

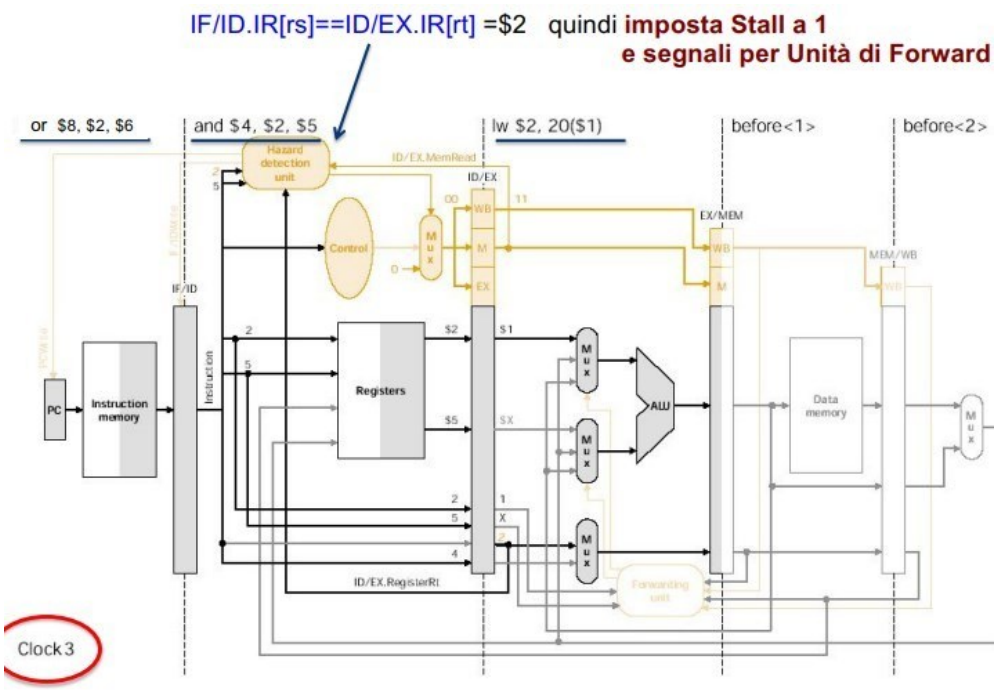

Tempo (in cicli di clock)
CC 1
CC 2
CC 3
CC 4
CC 5
CC 6
CC 7
CC 8
CC 9
Ordine di esecuzione del programma (sequenza delle istruzioni)
lw $2, 20($1)
and $4, $2, $5
or $8, $2, $6
add $9, $4, $2
slt $1, $6, $7
IM
Reg
DM
forwarding


Tempo (in cicli di clock)
CC 1
CC 2
CC 3
CC 4
CC 5
CC 6
CC 7
CC 8
CC 9
Ordine di esecuzione del programma (sequenza delle istruzioni)
lw $2, 20($1)
and $4, $2, $5
or $8, $2, $6
add $9, $4, $2
slt $1, $6, $7
IM
Reg
DM
prima scrive poi legge quindi ok!

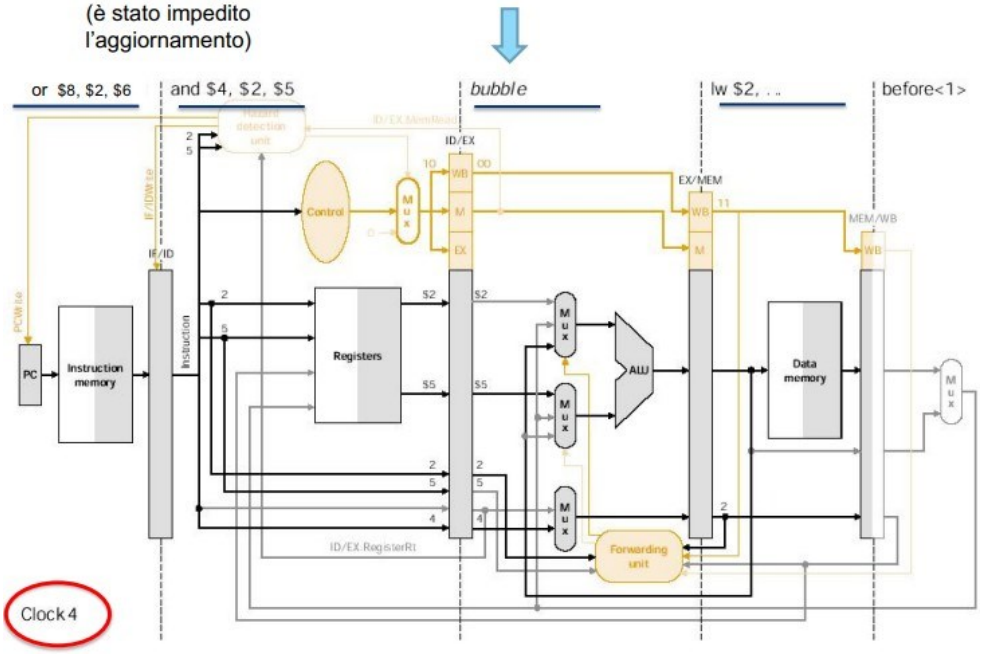

Tempo (in cicli di clock)
CC 1
CC 2
CC 3
CC 4
CC 7
CC 8
CC 9
forwarding
Ordine di esecuzione del programma (sequenza delle istruzioni)
lw $2, 20($1)
and $4, $2, $5
or $8, $2, $6
add $9, $4, $2
slt $1, $6, $7
IM
Reg
DM

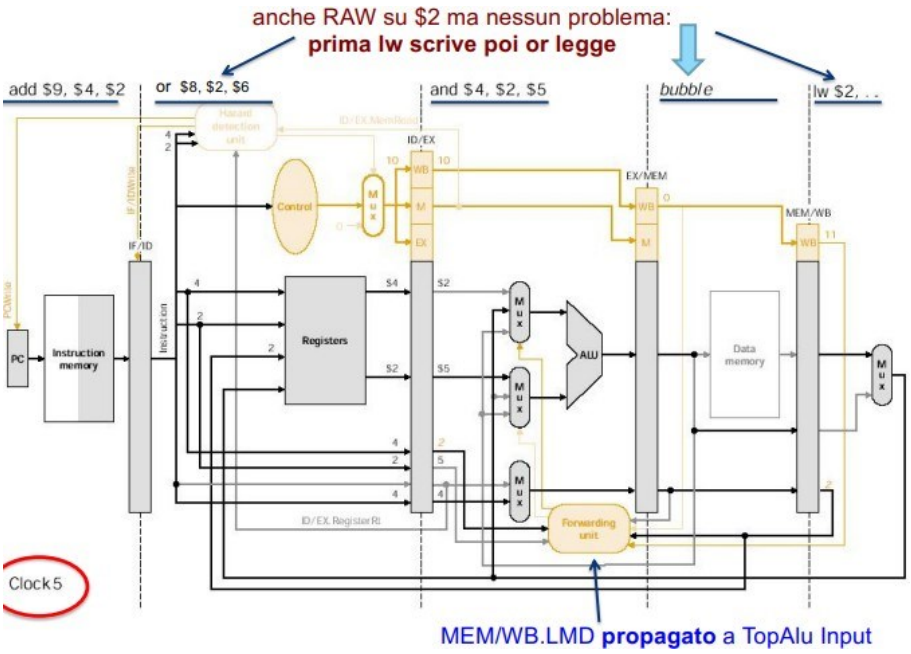

Tempo (in cicli di clock)
CC 1
CC 2
CC 3
CC 4
CC 5
CC 6
CC 7
CC 8
CC 9
Ordine di esecuzione del programma (sequenza delle istruzioni)
richiede uno stallo!
lw $2, 20($1)
and $4, $2, $5
or $8, $2, $6
add $9, $4, $2
slt $1, $6, $7
IM
Reg
DM

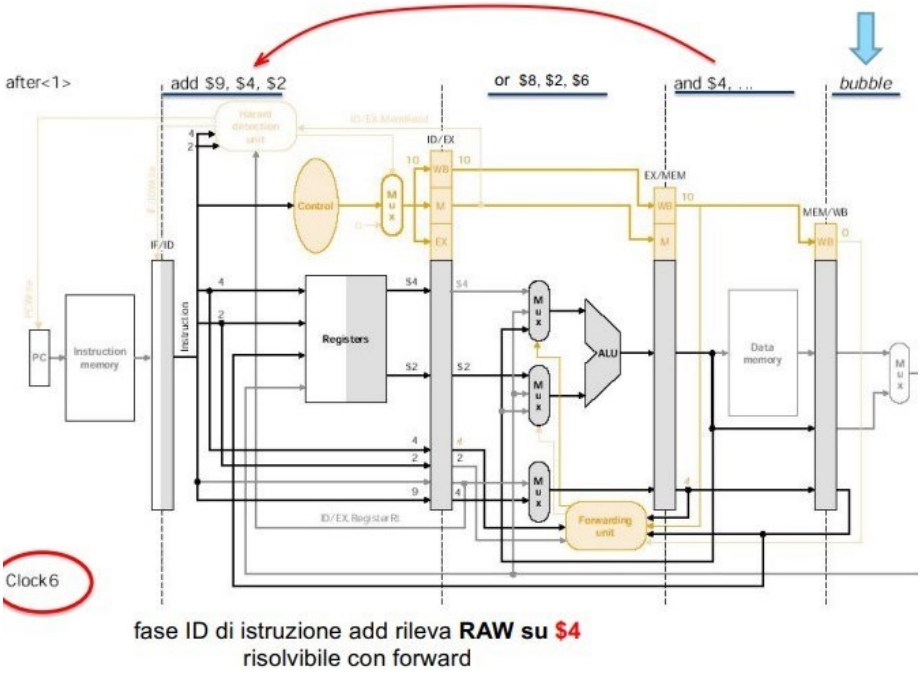

Tempo (in cicli di clock)
CC 1
CC 2
CC 3
CC 4
CC 5
CC 6
CC 7
CC 8
CC 9
CC 10
Ordine di esecuzione del programma (sequenza delle istruzioni)
stallo e forward di LMD verso AluInput
lw $2, 20($1)
and diventa una nop
and $4, $2, $5
or $8, $2, $6
add $9, $4, $2
bolla
vediamo cosa succede dentro il processore
IM
Reg
DM



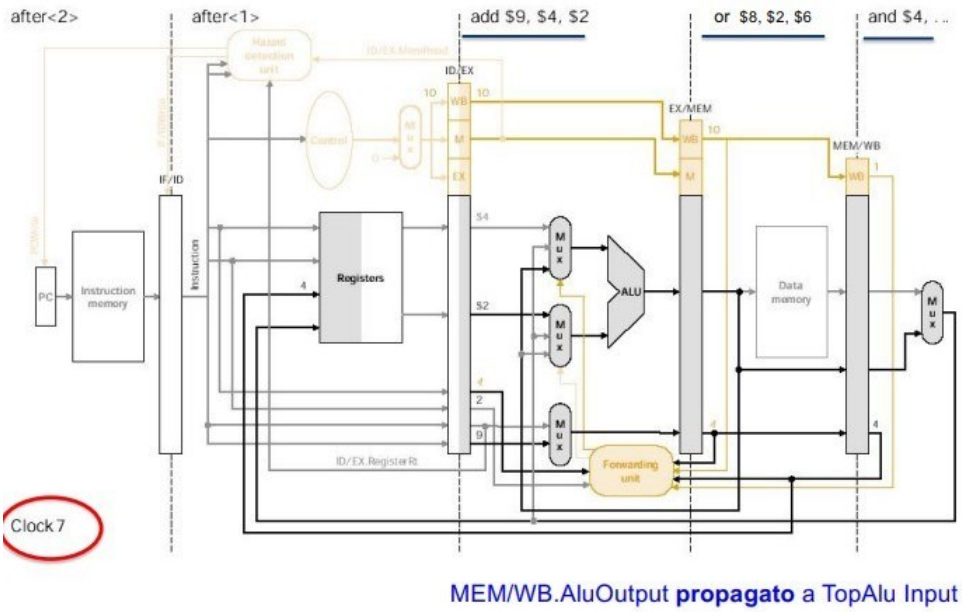
IF e ID sono ripetute
(è stato impedito
l'aggiornamento)







fase ID di istruzione add rileva **RAW su $4**
risolvibile con forward



MEM/WB.AluOutput **propagato** a TopAlu Input

Processori multicore

I microprocessori hanno visto una crescita esponenziale delle prestazioni, con grossi miglioramento della organizzazione e grande incremento della frequenza di clock
Similmente, una grande crescita del parallelismo, attraverso le pipeline o le pipeline parallele (superscalari)
Un approccio più aggressivo consiste nel dotare il processore di più unità di elaborazione per gestire diverse istruzioni in parallelo in ogni fase di elaborazione. In questo modo, diverse istruzioni iniziano l'esecuzione nello stesso ciclo di clock e si dice che il processo utilizza un'emissione multipla. Tali processori sono in grado di raggiungere un throughput di esecuzione delle istruzioni superiore a un'istruzione per ciclo.
Questo conduce ad usare le superscalari anche per il multithreading.
Problemi

- Maggiore complessità richiede logica più complessa
- Aumento dell'area del chip per supportare il parallelismo
- Più difficile da progettare, realizzare e verificare (debug)

La potenza cresce esponenzialmente con la densità del chip e la frequenza del clock
Rimedio: usare più spazio per la cache, meno densa e richiede molta meno potenza (ordini di magnitudine)
Nel 2015, si usano 100 miliardi di transistor in 300mm2 sul "die" (chip) e anche in cache di 100MB possiedono 1 miliardo di transistor per la logica.
Regola di Pollack: Le prestazioni sono all'incirca proporzionali alla radice quadrata dell'incremento in complessità
Il raddoppio in complessità restituisce il 40% in più di prestazione. Architetture multicore hanno il potenziale per ottenere un miglioramento quasi lineare- Improbabile che un core possa utilizzare efficacemente tutta la memoria cache.

I vantaggi prestazionali dipendono dallo sfruttamento efficace delle risorse parallele. Anche una piccola quantità di codice seriale ha un impatto significativo sulle prestazioni. Il 10% di codice intrinsecamente seriale eseguito su un sistema a 8 processori dà un incremento di prestazioni di solo 4,7 volte.
Overhead dovuto alla comunicazione, distribuzione del lavoro e mantenimento della coerenza della cache
Alcune applicazioni effettivamente sfruttano i processori multicore, soprattutto con un codice molto sequenziale, seguendo direttamente la crescita del numero dei processori.

Scritto da Gabriel

Il processore multicore dipende anche dalla struttura di lavoro utilizzata. Alcuni sistemi di lavoro non sono in grado di gestire processori multicore che richiedono una maggiore potenza. Supponiamo di avere un processore rapido, allora utilizzerà una maggiore quantità di energia, causando un elevato utilizzo della batteria del PC. Nel caso in cui si stia giocando ad un gioco ad alto contenuto di design, è necessaria un'ulteriore forza di gestione e una maggiore potenza di preparazione implica un maggiore utilizzo di energia, per cui la batteria del PC evapora rapidamente.

I processori multicore sono utilizzati nei campi seguenti:

- Disposizione di incredibili illustrazioni
- Progettazione supportata da PC (CAD)
- Applicazioni visive e sonore
- Giochi 3D
- Modifiche video
- Lavoratori basati sull'informazione
- Codifica

Nell'organizzazione multicore, si ha un certo numero di core per chip, diversi livelli di cache per chip e una certa quantità di cache condivisa.
Una tra le varie alternative, tra i vari tipi di cache multilivello (che utilizzano più di un livello di implementazione della cache per rendere la velocità di accesso alla cache quasi uguale alla velocità della CPU e per contenere un gran numero di oggetti della cache), vediamo che ce ne sono diverse, distinte tra:

- Cache L1 dedicata, con una serie di core e la main memory con una cache di livello 2
- Cache L2 dedicata, cioè una cache L2 per ogni core
- Cache L2 condivisa, quindi una cache L2 condivisa tra tutti i core
- Cache L3 condivisa. quindi una cache L3 condivisa tra tutti i core

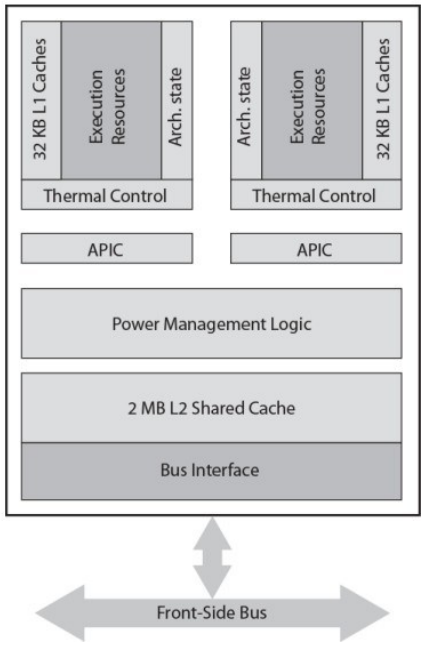
Vantaggi di una cache L2 condivisa:

- Riduzione (accidentale) del numero di miss totali
- Dati condivisi da più core non sono replicati a livello di cache (a livello 2, ma possibile replicazione a livello 1)
- Con appropriati algoritmi di sostituzione dei blocchi, la quantità di cache dedicata ad ogni core è dinamica
- Thread con minore località possono utilizzare più spazio di cache
- Comunicazione fra processi (anche in esecuzione su core diversi) facilitata dall'utilizzo della memoria condivisa
- Problema della coerenza della cache confinata al L1
- Cache L2 dedicate danno però un accesso alla memoria più rapido
- Migliori prestazioni per thread con forte località
- Anche una cache L3 condivisa può migliorare le prestazioni

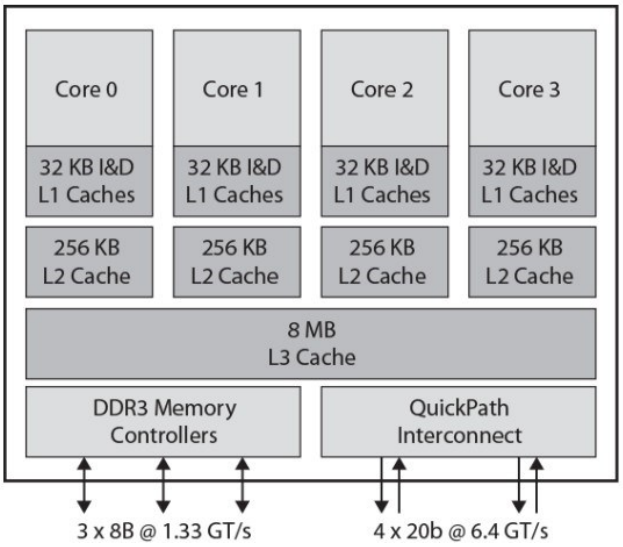
Alcuni esempi:

Intel Core Duo

- 2006
- core superscalari
- cache L2 condivisa
- cache L1 dedicata
- unità di controllo termico
- controllori di interruzioni programmabili (APIC)
- logica di controllo della potenza
- Interfaccia bus



Intel Core i7



- Novembre 2008
- core SMT
- cache L3 condivisa
- cache L1, L2 dedicate
- controllore memorie DDR3 sul chip
- Logica di connessione con controllo di coerenza della cache molto efficiente e veloce (banda totale di 25.6GB/s)

Vantaggi dei processori multicore:

- I processori multicore possono portare a termine più lavoro rispetto ai processori a singolo centro.
- Risultano incredibili per le applicazioni multi-stringa.
- Possono portare a termine un lavoro sincrono a bassa ricorrenza.
- Possono gestire più informazioni rispetto ai processori a centro singolo.
- Possono portare a termine più lavoro consumando poca energia rispetto ai processori a centro singolo.
- È possibile eseguire lavori complessi come il filtraggio dell'infezione contro l'infezione e la visione di un film contemporaneamente.
- Poiché i due centri dei processori sono su un singolo chip, le riserve del PC sono sfruttate e le informazioni non devono viaggiare più a lungo.
- Il PCB (circuito stampato) richiede meno spazio nel caso di utilizzo di processori multicore.
- I processori multicore sono ampiamente tolleranti ai guasti e sono molto affidabili.

Svantaggi dei processori multicore:

- Sono difficili da controllare rispetto ai processori a singolo centro.
- Sono più costosi di un processore a centro singolo.
- La loro velocità non è doppia rispetto a quella di un processore tipico.
- La presentazione del processore multicore dipende da come il cliente utilizza il PC.
- Consumano più energia.
- Questi processori si surriscaldano mentre svolgono più lavoro.
- Se un ciclo richiede una gestione diretta o consecutiva, il processore multicore deve rimanere in funzione più a lungo.

Esercizi pipeline parte 2

Pipeline e Riordino

Esercizio

Si consideri la pipeline MIPS a 5 stadi vista a lezione, con possibilità di data-forwarding e con possibilità di scrittura e successiva lettura dei registri in uno stesso ciclo di clock.
Per ognuna delle seguenti sequenze di istruzioni assembler, dove i dati immediati sono espressi in esadecimale, si chiede di:

1. indicare quali dipendenze dai dati sono presenti;
2. mostrare come evolve la pipeline durante l'esecuzione del codice, spiegando nel dettaglio i motivi di un eventuale stallo o dell'utilizzo di un particolare circuito di by-pass;
3. indicare se è possibile riordinare le istruzioni in modo da ridurre le dipendenze dai dati.

Sequenza 1

```
SUB $2, $7, $5
LW  $1, 7 ($2)
ADD $2, $1, $8
SW  $3, 73 ($1)
SUBI $2, $3, 4
ADDI $7, $3, 8
ADD  $1, $7, $2
```

Sequenza 2

```
LW  $3, 80 ($0)
ADD $2, $3, $1
LW  $1, 800($2)
SUBI $1, $1, 3
ADDI $2, $2, 4
SW  $1, 108($2)
SUB $4, $3, $1
```

Sequenza 3

```
SW  $9, 0 ($1)
LW  $1, 7 ($9)
SUB $9, $1, $8
SW  $3, 73 ($9)
SUBI $9, $3, 9
SW  $7, 78 ($9)
LW  $9, A ($7)
```

Sequenza 1

```
SUB $2, $7, $5      R2 <- [R7]-[R5]
LW $1, 7 ($2)       R1 <- mem[7+[R2]]
ADD $2, $1, $8      R2 <- [R1]+[R8]
SW $3, 73 ($1)      mem[73+[R1]] <- [R3]
SUBI $2, $3, 4      R2 <- [R3]-4
ADDI $7, $3, 8      R7 <- [R3]+8
ADD $1, $7, $2      R1 <- [R7]+[R2]
```

Sequenza 1

```
SUB $2, $7, $5
LW $1, 7 ($2)
ADD $2, $1, $8
SW $3, 73 ($1)
SUBI $2, $3, 4
ADDI $7, $3, 8
ADD $1, $7, $2
```

	1	2	3	4	5	6	7	8	9	10
SUB	IF	ID	EX	MEM	WB					
LW		IF	ID	EX	MEM	WB				

si accorge che il registro di **lettura** IF/ID.IR[rs]
è uguale al registro di **scrittura** ID/EX.IR[rd] => **RAW $2**

risovibile con data forwarding:
EX/MEM.AluOutput inviato a TopAluInput

SUB	R	codop	rs	rt	rd	sham	funct
LW	I	codop	rs	rt	address/const		

Sequenza 1

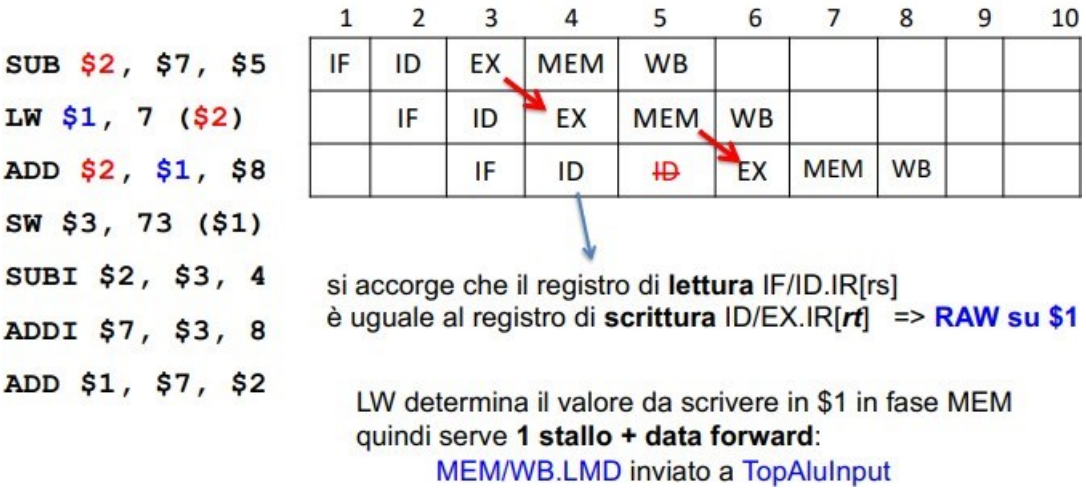
```
SUB $2, $7, $5
LW $1, 7 ($2)
ADD $2, $1, $8
SW $3, 73 ($1)
SUBI $2, $3, 4
ADDI $7, $3, 8
ADD $1, $7, $2
```

	1	2	3	4	5	6	7	8	9	10
SUB	IF	ID	EX	MEM	WB					
LW		IF	ID	EX	MEM	WB				
ADD			IF	ID						

si accorge che il registro di **lettura** IF/ID.IR[rs]
è uguale al registro di **scrittura** ID/EX.IR[***rt***] => **RAW su $1**

ADD	R	codop	rs	rt	rd	sham	funct
LW	I	codop	rs	rt	address/const		

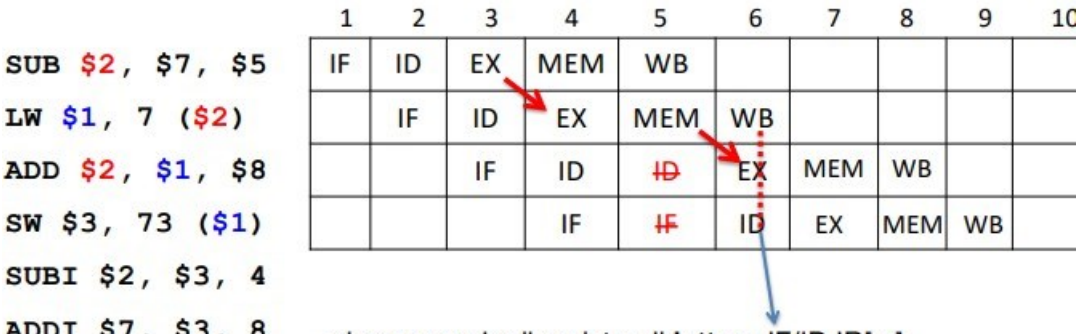
Sequenza 1



si accorge che il registro di **lettura** IF/ID.IR[rs]
è uguale al registro di **scrittura** ID/EX.IR[***rt***] => **RAW su $1**

LW determina il valore da scrivere in $1 in fase MEM
quindi serve **1 stallo + data forward**:
MEM/WB.LMD inviato a TopAluInput

Sequenza 1



si accorge che il registro di **lettura** IF/ID.IR[rs]
è uguale al registro di **scrittura** ***della LW*** MEM/WB.IR[***rt***]
=> **RAW su $1**

ma nella **prima metà** del ciclo 6 WB di LW scrive $1
e nella **seconda metà** ID legge $1, quindi **tutto ok**

Sequenza 1

```
SUB $2, $7, $5
LW $1, 7 ($2)
ADD $2, $1, $8
SW $3, 73 ($1)
SUBI $2, $3, 4
ADDI $7, $3, 8
ADD $1, $7, $2
```

	1	2	3	4	5	6	7	8	9	10
SUB $2, $7, $5	IF	ID	EX	MEM	WB					
LW $1, 7 ($2)		IF	ID	EX	MEM	WB				
ADD $2, $1, $8			IF	ID	~~ID~~	EX	MEM	WB		
SW $3, 73 ($1)				IF	~~IF~~	ID	EX	MEM	WB	
SUBI $2, $3, 4						IF	ID	EX	MEM	WB

tutto OK:

- $3 è usato in **lettura** sia da SUBI che da SW
- SUBI scrive su $2 in fase WB, senza conflitto con i precedenti

Sequenza 1

```
SUB $2, $7, $5
LW $1, 7 ($2)
ADD $2, $1, $8
SW $3, 73 ($1)
SUBI $2, $3, 4
ADDI $7, $3, 8
ADD $1, $7, $2
```

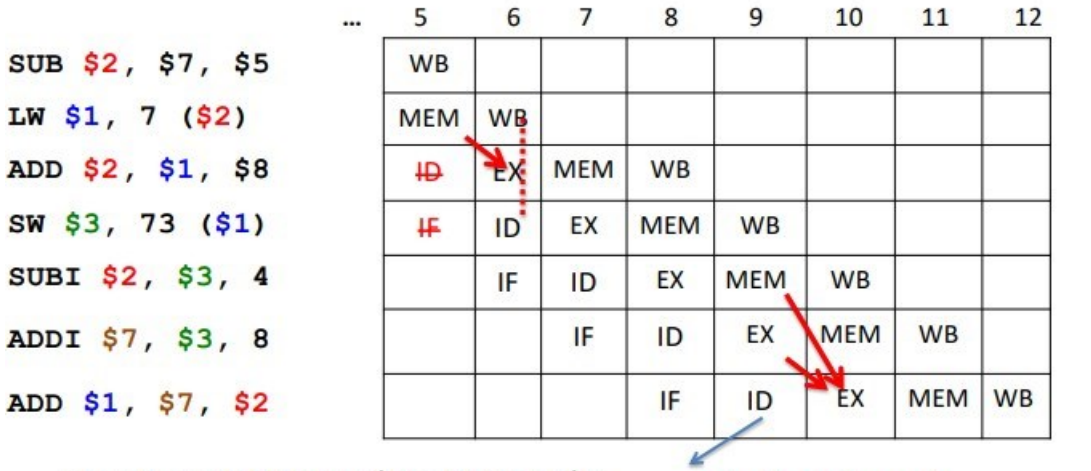
	...	5	6	7	8	9	10	11	12
SUB $2, $7, $5		WB							
LW $1, 7 ($2)		MEM	WB						
ADD $2, $1, $8		~~ID~~	EX	MEM	WB				
SW $3, 73 ($1)		~~IF~~	ID	EX	MEM	WB			
SUBI $2, $3, 4			IF	ID	EX	MEM	WB		
ADDI $7, $3, 8				IF	ID	EX	MEM	WB	
ADD $1, $7, $2					IF	ID			

tutto OK: $3 è usato in **lettura** sia da SUBI che da SW

si accorge di **RAW** su **$7** e **RAW** su **$2**

- $7 = IF/ID.IR[rs] = ID/EX.IR[rt]
- $2 = IF/ID.IR[rt] = EX/MEM.IR[rt]

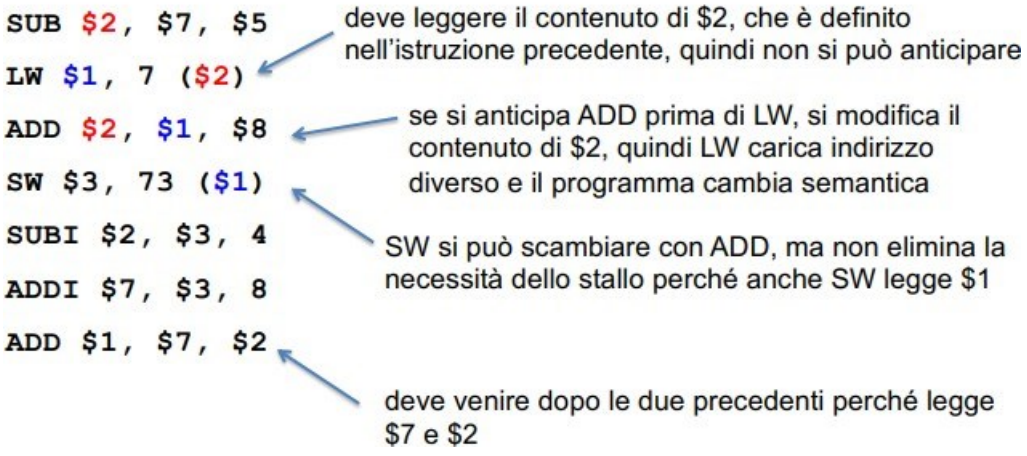
Sequenza 1



si accorge di **RAW** su **$7** e **RAW** su **$2** risolti con forward:

- $7 = IF/ID.IR[rs] = ID/EX.IR[rt] EX/MEM.AluOutput inviato a TopAluInput
- $2 = IF/ID.IR[rt] = EX/MEM.IR[rt] MEM/WB.AluOutput va a BottomAluInput

Sequenza 1: Riordino?



Sequenza 1: Riordino?

```
SUB $2, $7, $5
LW $1, 7 ($2)
ADD $2, $1, $8
SW $3, 73 ($1)
SUBI $2, $3, 4
ADDI $7, $3, 8
ADD $1, $7, $2
```

non avevano dipendenze, provo ad anticiparle per allontanare la dipendenza RAW su $1 che genera lo ***stallo*** (ricorda che SW **legge** $3)

```
SUB $2, $7, $5
LW $1, 7 ($2)
SUBI $2, $3, 4
ADDI $7, $3, 8
ADD $2, $1, $8
SW $3, 73 ($1)
ADD $1, $7, $2
```

legge $1 corretto

NO: legge $2 definito dalla ADD invece che da SUBI

e se anticipo anche questa subito dopo ADDI, allora sovrascrive $1

Sequenza 1: Riordino?

```
SUB $2, $7, $5
LW $1, 7 ($2)
ADD $2, $1, $8
SW $3, 73 ($1)
SUBI $2, $3, 4
ADDI $7, $3, 8
ADD $1, $7, $2
```

```
SUB $2, $7, $5
LW $1, 7 ($2)
ADD $2, $1, $8
SUBI $2, $3, 4
ADDI $7, $3, 8
SW $3, 73 ($1)
ADD $1, $7, $2
```

toglie il forward doppio

```
SUB $2, $7, $5
LW $1, 7 ($2)
ADDI $7, $3, 8
ADD $2, $1, $8
SW $3, 73 ($1)
SUBI $2, $3, 4
ADD $1, $7, $2
```

toglie lo stallo

si possono fare entrambi
altri riordini sono possibili

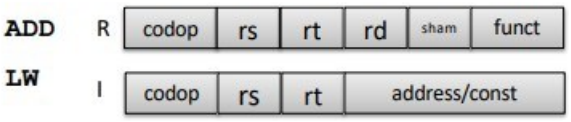
Sequenza 2

```
LW $3, 80 ($0)
ADD $2, $3, $1
LW $1, 800($2)
SUBI $1, $1, 3
ADDI $2, $2, 4
SW $1, 108($2)
SUB $4, $3, $1
```

	1	2	3	4	5	6	7	8	9	10
LW $3, 80 ($0)	IF	ID	EX	MEM	WB					
ADD $2, $3, $1		IF	ID	~~ID~~	EX	MEM	WB			

si accorge che il registro di **lettura** IF/ID.IR[rs]
è uguale al registro di **scrittura** ID/EX.IR[rt] => **RAW**

LW determina il valore da scrivere in $3 in fase MEM
quindi serve **1 stallo + data forward**:
MEM/WB.LMD inviato a TopAluInput



Sequenza 2

```
LW $3, 80 ($0)
ADD $2, $3, $1
LW $1, 800($2)
SUBI $1, $1, 3
ADDI $2, $2, 4
SW $1, 108($2)
SUB $4, $3, $1
```

	1	2	3	4	5	6	7	8	9	10
LW $3, 80 ($0)	IF	ID	EX	MEM	WB					
ADD $2, $3, $1		IF	ID	~~ID~~	EX	MEM	WB			
LW $1, 800($2)			IF	~~IF~~	ID	EX	MEM	WB		

si accorge che il registro di **lettura** IF/ID.IR[rs]
è uguale al registro di **scrittura** ID/EX.IR[***rd***] => **RAW** su $2

risovibile con data forwarding:
EX/MEM.AluOutput inviato a TopAluInput

Sequenza 2

```
LW $3, 80 ($0)
ADD $2, $3, $1
LW $1, 800($2)
SUBI $1, $1, 3
ADDI $2, $2, 4
SW $1, 108($2)
SUB $4, $3, $1
```

	1	2	3	4	5	6	7	8	9	10
LW	IF	ID	EX	MEM	WB					
ADD		IF	ID	~~ID~~	EX	MEM	WB			
LW			IF	~~IF~~	ID	EX	MEM	WB		
SUBI					IF	ID	~~ID~~	EX	MEM	WB

si accorge che il registro di **lettura** IF/ID.IR[rs]
è uguale al registro di **scrittura** ID/EX.IR[***rt***] => **RAW** su $1

LW determina il valore da scrivere in $1 in fase MEM
quindi serve **1 stallo + data forward**:
MEM/WB.LMD inviato a TopAluInput

SUBI/LW

Sequenza 2

```
LW $3, 80 ($0)
ADD $2, $3, $1
LW $1, 800($2)
SUBI $1, $1, 3
ADDI $2, $2, 4
SW $1, 108($2)
SUB $4, $3, $1
```

	... 5	6	7	8	9	10	11
LW	WB						
ADD	EX	MEM	WB				
LW	ID	EX	MEM	WB			
SUBI	IF	ID	~~ID~~	EX	MEM	WB	
ADDI		IF	~~IF~~	ID	EX	MEM	WB
SW				IF	ID		

tutto OK

`mem[108+[R2]] <- [R1]`

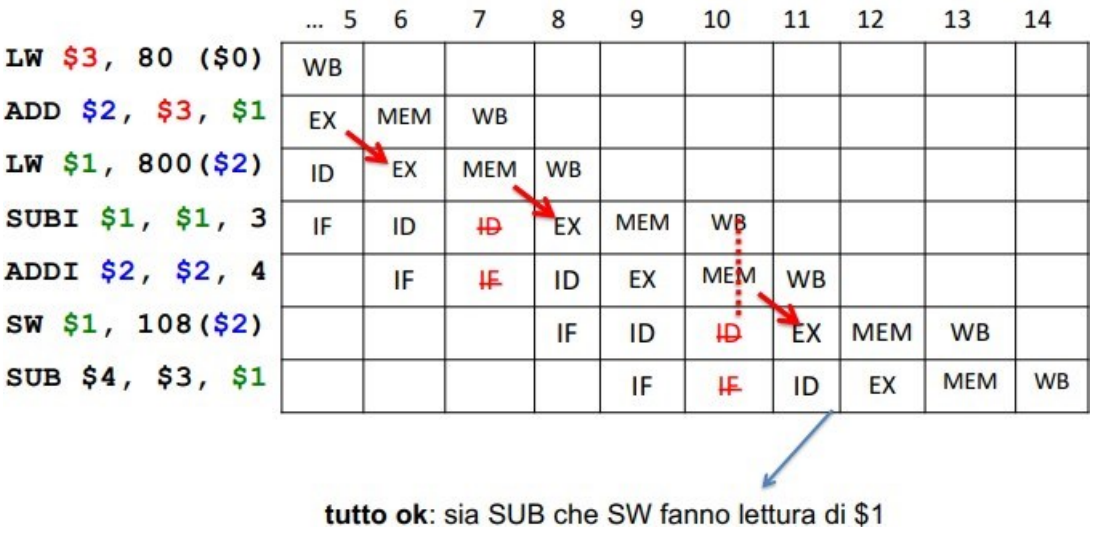
si accorge di **RAW su $2**, risolvibile con **forward**
di EX/MEM.AluOutput verso TopAluInput

ma c'è un'altra **RAW su $1**:

SW in fase ID deve anche **leggere** $1=IF/ID.IR[rt] e
propagarlo nei registri di pipeline **fino alla fase MEM** ma
il valore corretto di $1 sarà scritto in fase WB di SUBI

l'esecuzione di SW
non può procedere
-> **stallo**

Sequenza 2



tutto ok: sia SUB che SW fanno lettura di $1

Pipeline **SENZA** data forward

	1	2	3	4	5	6	7	8	9	10	11
lw $3 80($0)	IF	ID	EX	MEM	WB						
add $2 $3 $1		IF	ID	ID	ID	EX	MEM	WB			
lw $1 38($2)			IF	IF	IF	ID	ID	ID	EX	MEM	WB
subi $1 $1 3						IF	IF	IF	ID		
addi $2 $2 4											
sw $1 11($2)											
sub $4 $3 $1											

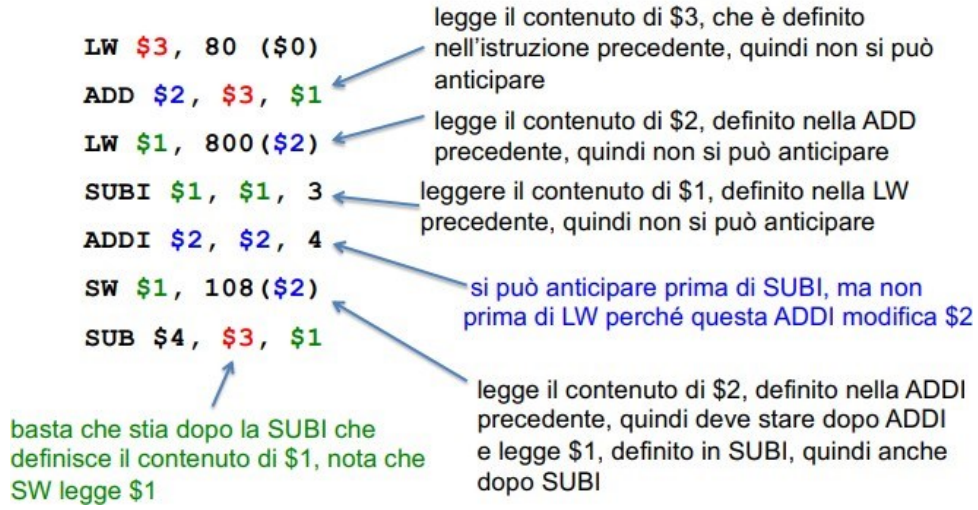
Pipeline **SENZA** data forward

	5	6	7	8	9	10	11	12	13	14	15
lw $3 80($0)	WB										
add $2 $3 $1	ID	EX	MEM	WB							
lw $1 38($2)	IF	ID	ID	ID	EX	MEM	WB				
subi $1 $1 3		IF	IF	IF	ID	ID	ID	EX	MEM	WB	
addi $2 $2 4					IF	IF	IF	ID	EX	MEM	WB
sw $1 11($2)											
sub $4 $3 $1											

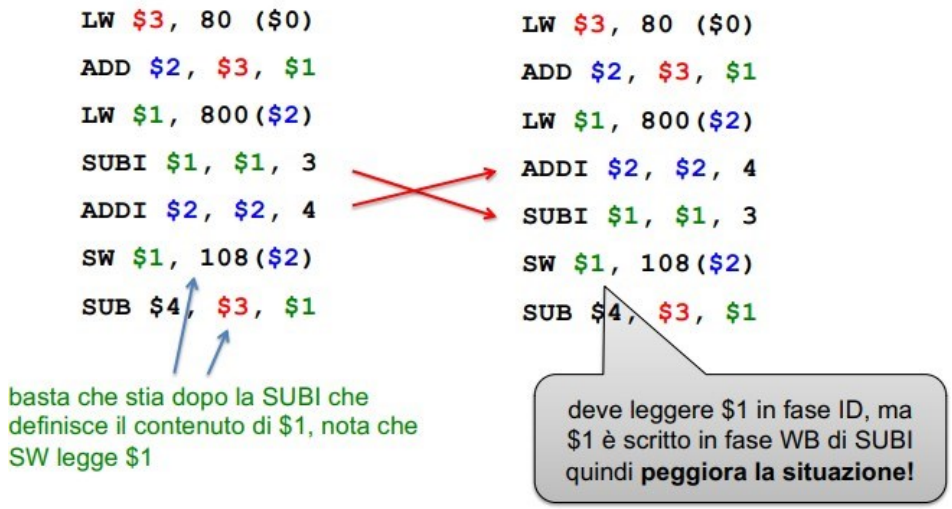
Pipeline **SENZA** data forward

	9	10	11	12	13	14	15	16	17	18	19
`lw $3 80($0)`											
`add $2 $3 $1`											
`lw $1 38($2)`	EX	MEM	WB								
`subi $1 $1 3`	ID	ID	ID	EX	MEM	WB					
`addi $2 $2 4`	IF	IF	IF	ID	EX	MEM	WB				
`sw $1 11($2)`				IF	ID	ID	ID	EX	MEM	WB	
`sub $4 $3 $1`					IF	IF	IF	ID	EX	MEM	WB

Sequenza 2: Riordino?



Sequenza 2: Riordino?



Sequenza 3

```
SW $9, 0 ($1)
LW $1, 7 ($9)
SUB $9, $1, $8
SW $3, 73 ($9)
SUBI $9, $3, 9
SW $7, 78 ($9)
LW $9, A ($7)
```

Pipeline con data forward

	1	2	3	4	5	6	7	8	9	10	11	12
SW $9, 0 ($1)	IF	ID	EX	MEM	WB							
LW $1, 7 ($9)		IF	ID	EX	MEM	WB						
SUB $9, $1, $8			IF	ID	ID	EX	MEM	WB				
SW $3, 73 ($9)				IF	IF	ID	EX	MEM	WB			
SUBI $9, $3, 9						IF	ID	EX	MEM	WB		
SW $7, 78 ($9)							IF	ID	EX	MEM	WB	
LW $9, A ($7)								IF	ID	EX	MEM	WB

(ID/EX.IR[opcode] == lw && IF/ID.IR[rs] == ID/EX.IR[rt]) =>
1 stallo + MEM/WB.LMD -> TopAluInput

(ID/EX.IR[opcode] = Reg-Reg && F/ID.IR[rs] == ID/EX.IR[rt]) =>
EX/MEM.AluOutput -> TopAluInput

Sequenza 3

```
SW $9, 0 ($1)
LW $1, 7 ($9)
SUB $9, $1, $8
SW $3, 73 ($9)
SUBI $9, $3, 9
SW $7, 78 ($9)
LW $9, A ($7)
```

legge il contenuto di $1, che è scritto nell'istruzione successiva, quindi non si può spostare dopo la LW

scrive $1, quindi non si può posizionare dopo SUB che legge $1

legge il contenuto di $1, scritto nella LW precedente, quindi non si può anticipare

legge il contenuto di $9, scritto nella SUB precedente e SUBI successiva, quindi non si può anticipare né posticipare

non si può anticipare prima di SW, perché scrive $9, letto da SW

legge il contenuto di $9, scritto nella SUBI precedente, quindi deve occorrere dopo SUBI

scrive $9 e quindi non si può spostare sopra SW che legge $9

NESSUN RIORDINO POSSIBILE

Sequenza 3 bis

```
lw    $1   0  ($2)
add   $2   $3   $1
sw    $2   21  ($1)
beq   $2   $1   11
add   $3   $2   $2
add   $1   $1   $3
sw    $3   0  ($1)
```

Pipeline CON data forward

	1	2	3	4	5	6	7	8	9	10	11
lw $1 0 ($2)	IF	ID	EX	MEM	WB						
add $2 $3 $1		IF	ID	ID	EX	MEM	WB				
sw $2 21 ($1)											
beq $2 $1 11											
add $3 $2 $2											
add $1 $1 $3											
sw $3 0 ($1)											

istruzione 2

IF/ID.IR[rt] = ID/EX.IR[rt] ➔ RAW $1

1 stallo e forward:
MEM/WB.**LMD** → BottomALUInput

Pipeline CON data forward

	1	2	3	4	5	6	7	8	9	10	11
lw $1 0 ($2)	IF	ID	EX	MEM	WB						
add $2 $3 $1		IF	ID	ID	EX	MEM	WB				
sw $2 21 ($1)			IF	IF	ID	ID	ID	EX	MEM	WB	
beq $2 $1 11											
add $3 $2 $2											
add $1 $1 $3											
sw $3 0 ($1)											

istruzione 3

IF/ID.IR[rs] = MEM/WB.IR[rt] ➔ RAW $1

IF/ID.IR[rt] = ID/EX.IR[rd] → RAW $2

deve leggere $2 in fase ID
quindi **2 stalli** e lettura in seconda metà di ciclo
così si risolve anche RAW $1

Pipeline CON data forward

	1	2	3	4	5	6	7	8	9	10	11
lw $1 0 ($2)	IF	ID	EX	MEM	WB						
add $2 $3 $1		IF	ID	ID	EX	MEM	WB				
sw $2 21 ($1)			IF	IF	ID	ID	ID	EX	MEM	WB	
beq $2 $1 11					IF	IF	IF	ID	EX	MEM	WB
add $3 $2 $2											
add $1 $1 $3											
sw $3 0 ($1)											

istruzione 4

beq legge sia $2 che $1, che erano scritte da istruzioni 2 e 1, ma al ciclo 8 sono già completate

in ciclo 10 usa la condizione di salto.
Supponiamo sia falsa

Pipeline CON data forward

	1	2	3	4	5	6	7	8	9	10	11
lw $1 0 ($2)	IF	ID	EX	MEM	WB						
add $2 $3 $1		IF	ID	ID	EX	MEM	WB				
sw $2 21 ($1)			IF	IF	ID	ID	ID	EX	MEM	WB	
beq $2 $1 11					IF	IF	IF	ID	EX	MEM	WB
add $3 $2 $2								IF	ID	EX	..
add $1 $1 $3											
sw $3 0 ($1)											

istruzione 5

tutto ok

Pipeline CON data forward

	5	6	7	8	9	10	11	12	13	14
lw $1 0 ($2)	WB									
add $2 $3 $1	EX	MEM	WB							
sw $2 21 ($1)	ID	ID	ID	EX	MEM	WB				
beq $2 $1 11	IF	IF	IF	ID	EX	MEM	WB			
add $3 $2 $2				IF	ID	EX	MEM	WB		
add $1 $1 $3					IF	ID	EX	MEM	WB	
sw $3 0 ($1)										

istruzione 6

IF/ID.IR[rt] = ID/EX.IR[rd] → RAW $3

si risolve con forward:
EX/MEM.**ALUOutput** → BottomALUInput

Pipeline CON data forward

	5	6	7	8	9	10	11	12	13	14	15
lw $1 0 ($2)	WB										
add $2 $3 $1	EX	MEM	WB								
sw $2 21 ($1)	ID	ID	ID	EX	MEM	WB					
beq $2 $1 11	IF	IF	IF	ID	EX	MEM	WB				
add $3 $2 $2				IF	ID	EX	MEM	WB			
add $1 $1 $3					IF	ID	EX	MEM	WB		
sw $3 0 ($1)						IF	ID	ID	EX	MEM	WB

istruzione 7

IF/ID.IR[rs] = ID/EX.IR[rd] → RAW $1

IF/ID.IR[rt] = EX/MEM.IR[rd] → RAW $3

deve leggere $3 in fase ID
quindi 1 stalli e r/w in ciclo 12
più forward:
MEM/WB.AluOutput →TopAluInput

Sequenza con branch

```
        LW   $2, 0 ($1)
Label1: BEQ  $2, $0, Label2
        LW   $3, 0 ($2)
        BEQ  $3, $0, Label1
        ADD  $1, $3, $1
Label2: SW   $1, 0 ($2)
```

Scenari:

← preso / non preso

← preso / non preso

per semplicità

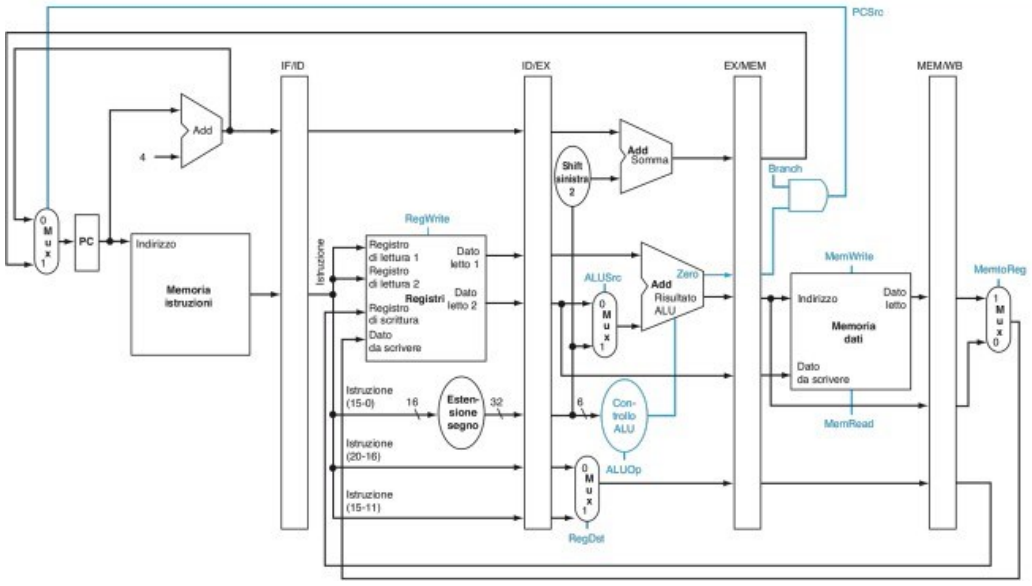
- usiamo Label al posto del campo Imm (esteso a 32 bit) da usare come offset per calcolare l'indirizzo del salto:

  `beq $rs, $rt, Imm  if([$rs]-[$rt]==0) then PC=NPC+(Imm<<2)`

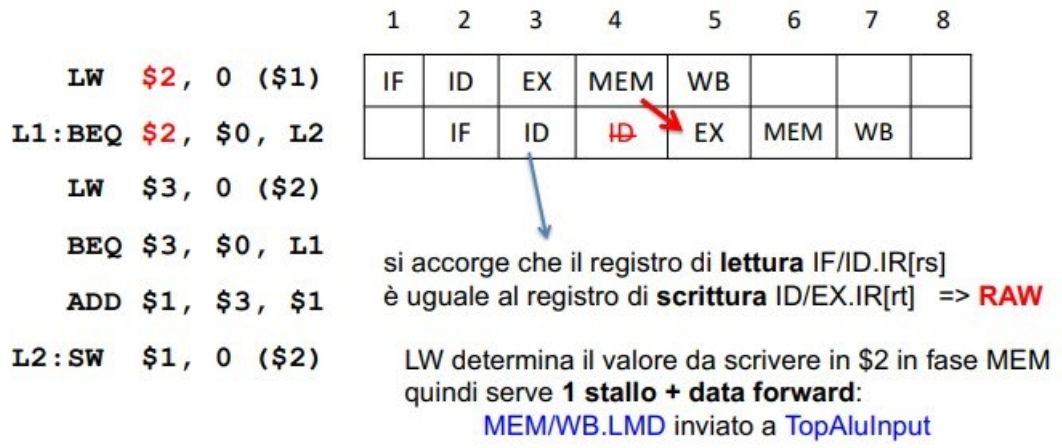
- **non assumiamo alcuna tecnica di predizione dei salti**
- in fase EX di istruzione beq calcola la condizione e il target, ma è in fase **MEM che decide se saltare,** cioè **usa** la condizione per decidere il valore di PC

Segnali di controllo

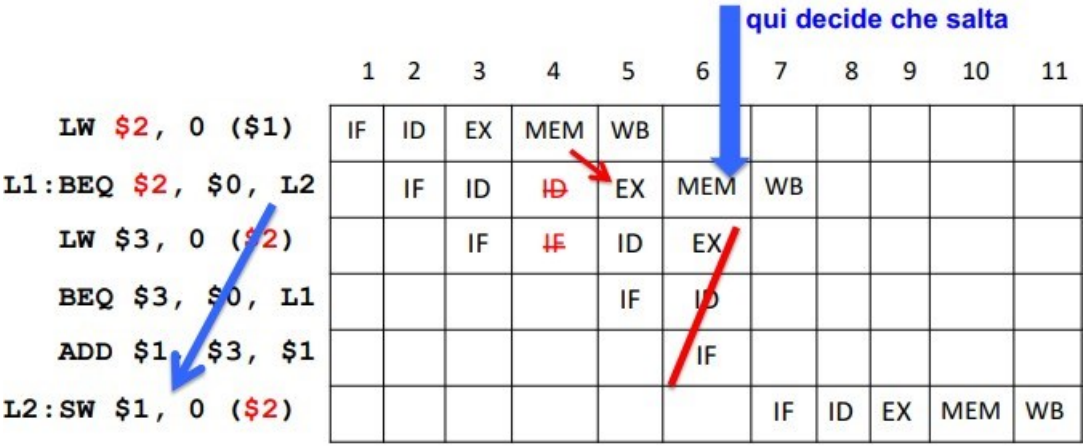
in fase MEM decide come aggiornare il PC a seconda di salto preso o no



Sequenza con branch



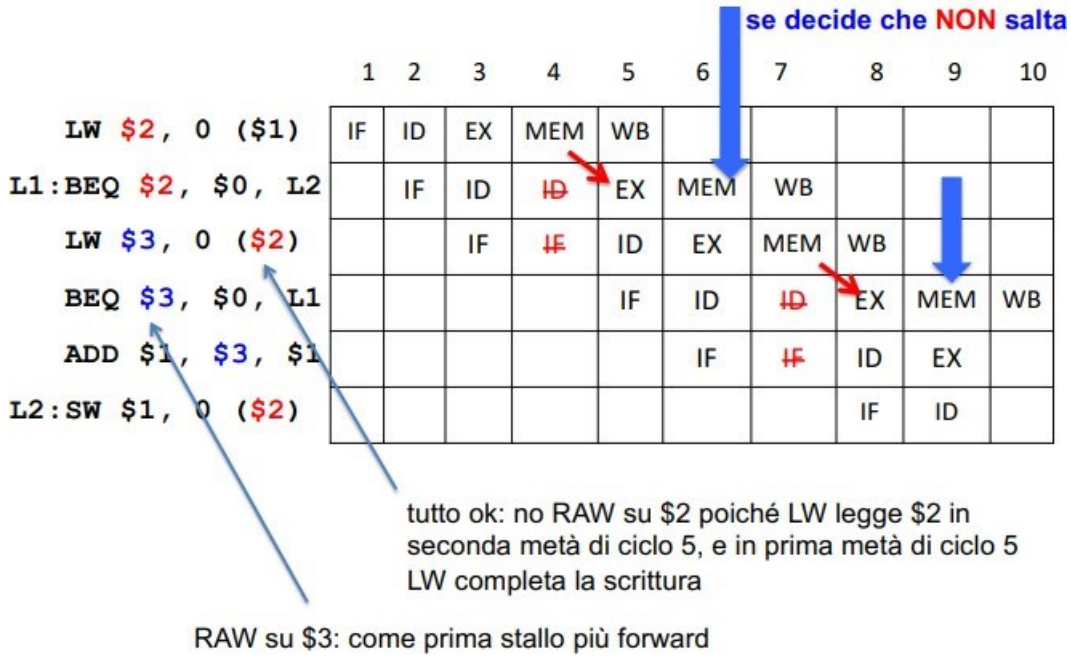
Sequenza con branch



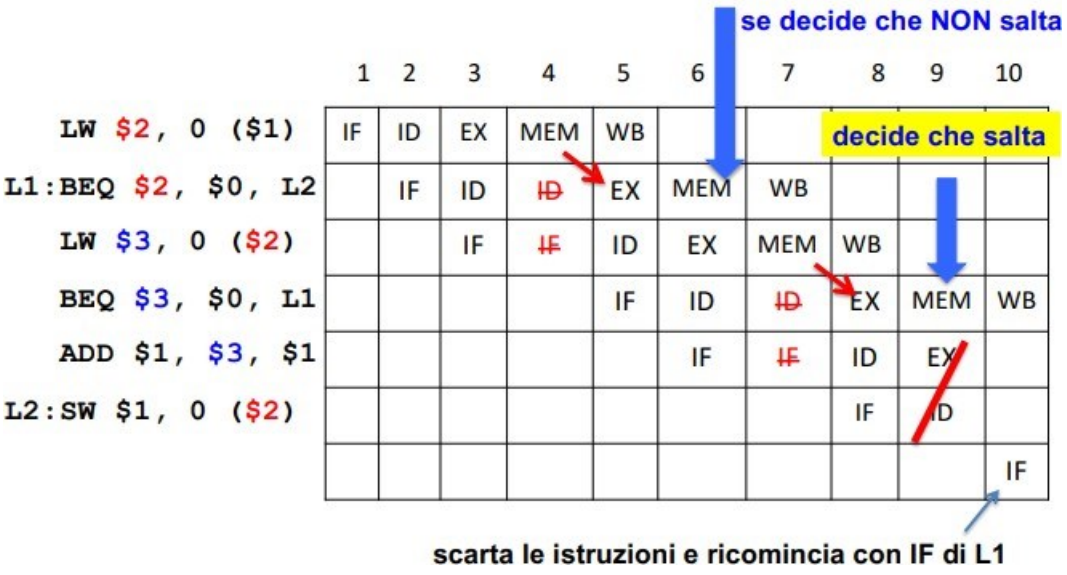
scarta il contenuto della pipeline e inizia IF dell'istruzione in L2
nessun problema di dipendenza

branch penalty = n. di cicli in cui non conclude nessuna istruzione = 3

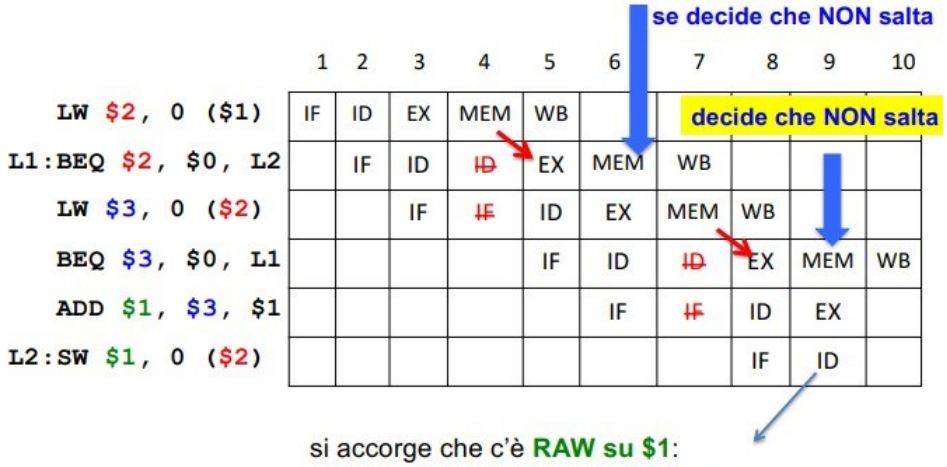
Sequenza con branch



Sequenza con branch



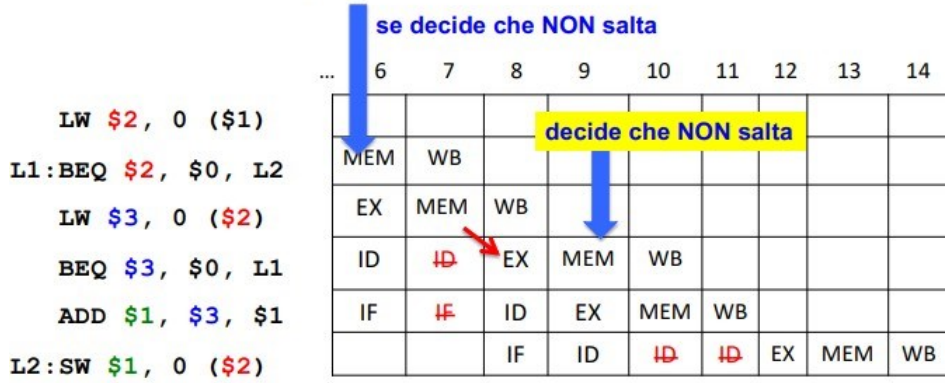
Sequenza con branch



si accorge che c'è **RAW su $1**:

SW in fase ID deve **leggere** $1 e **propagarlo** nei registri di pipeline **fino alla fase MEM** ma il valore corretto di $1 sarà scritto in fase WB di ADD
non c'è forward verso fase ID, quindi 2 stalli

Sequenza con branch



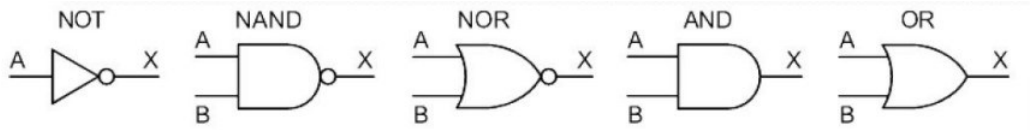
ripete la fase ID:

- nella prima metà del ciclo ADD scrive $1
- nella seconda metà del ciclo SW può leggere $1

CON DF	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	
SUB $5, $1, $4	IF	ID	EXE	MEM	WB	EXE/MEM.ALUOutput_sub -> EXE.Top_ALU_input_lw										
LW $1, 7($5)		IF	ID	EXE	MEM	WB	MEM/WB.LMD_lw -> EXE.Top_ALU_input_add									
ADD $5, $8, $1			IF	ID	ID	EXE	MEM	WB	1) EXE/MEM.ALUOutput_add -> EXE.Top_ALU_input_lw 2) MEM/WB.ALUOutput_add -> EXE.Top_ALU_input_add						
LW $3, 73($5)				IF	IF	ID	EXE	MEM	WB	MEM/WB.LMD_lw -> EXE.Top_ALU_input_sw						
ADDI $5, $5, 3						IF	ID	EXE	MEM	WB						
SW $7, 78($3)							IF	ID	EXE	MEM	WB					
LW $5, A($7)								IF	ID	EXE	MEM	WB				

Contenuti integrativi: Circuiti e Microprogrammazione

Dato l'insieme di porte logiche di base:



A	X
0	1
1	0

A	B	X
0	0	1
0	1	1
1	0	1
1	1	0

A	B	X
0	0	1
0	1	0
1	0	0
1	1	0

A	B	X
0	0	0
0	1	0
1	0	0
1	1	1

A	B	X
0	0	0
0	1	1
1	0	1
1	1	1

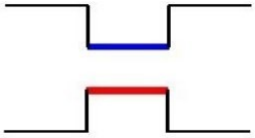
NAND o NOR sono complete → circuiti con solo porte NAND o solo porte NOR.
Rete combinatoria: insieme di porte logiche connesse il cui output in un certo istante è funzione solo dell'input in quell'istante. N input binari e m output binari e ad ogni combinazione di valori di ingresso corrisponde una ed una sola combinazione di valori di uscita.
Utili per implementare la ALU e la connessione tra parti della CPU, ma non sono in grado di memorizzare uno stato; quindi, non possono essere usate per implementare la memoria. Per questo servono le reti sequenziali dall'output dipende non solo dall'input corrente, ma anche dalla storia passata degli input

Vediamo alcuni esempi di circuiti:

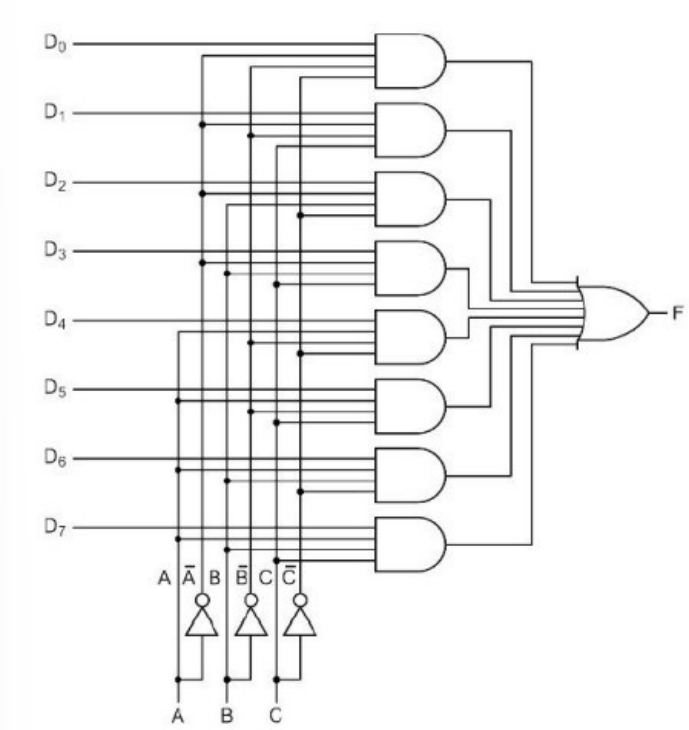
- I segnali sono discretizzati e di solito assumono solo due stati:

0 / FALSO / [0..1] Volt

1 / VERO / [2..5] Volt



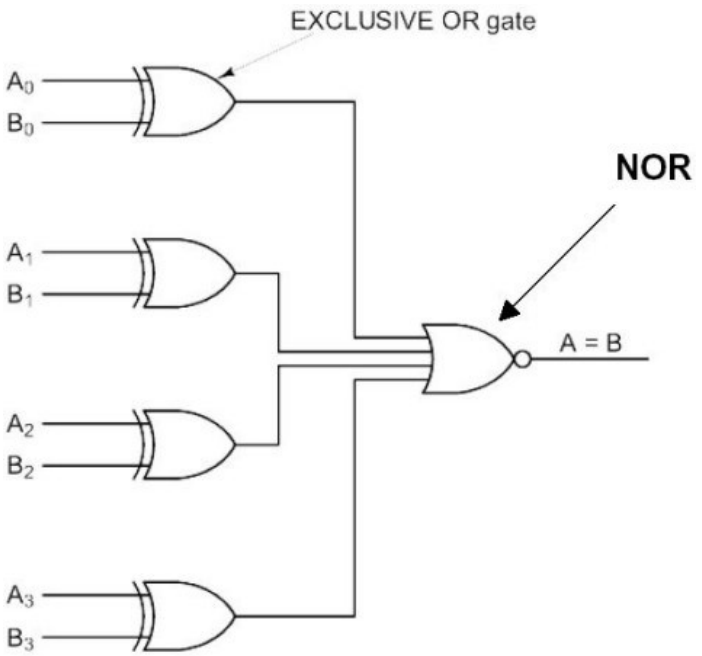
- I circuiti più complessi sono realizzati attraverso la combinazione di circuiti semplici (porte logiche)
- Le porte logiche sono realizzate tramite transistor (sono in pratica interruttori automatici)



Il multiplexer è un circuito capace di selezionare uno tra i vari ingressi possibili e di trasferire il dato in esso presente in uscita. È sempre dotato di uno o più ingressi di selezione: m ingressi di selezione servono per pilotare $n=2^m$ ingressi dati. Solo uno degli ingressi viene trasferito all'output. Ci sono "n" ingressi di controllo, che indicano l'ingresso da trasferire:

- 2^n linee di input (D0 - D7)
- n linee di controllo (A,B,C)
- 1 linea di output (F)
- Per ogni combinazione degli ingressi di controllo, 2n -1 delle porte AND hanno uscita 0, l'altra fa uscire l'ingresso.

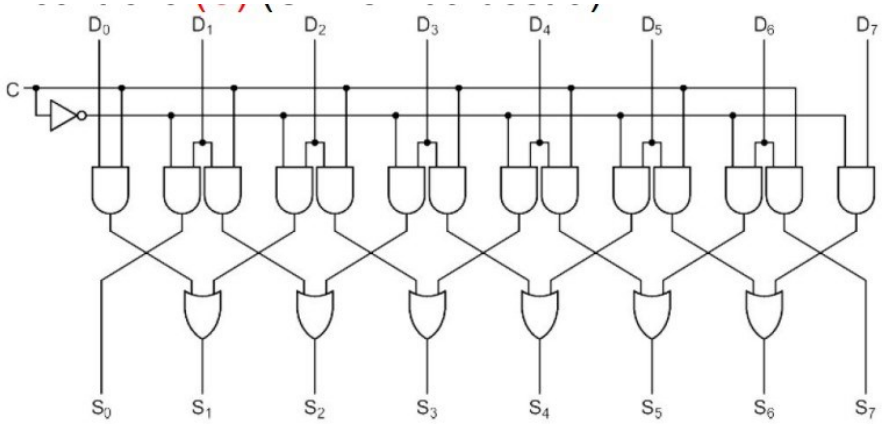
Un comparatore digitale o comparatore di magnitudine è un dispositivo elettronico hardware che riceve in input due segnali e in output determina se sono uguali, o quale dei due è il maggiore.



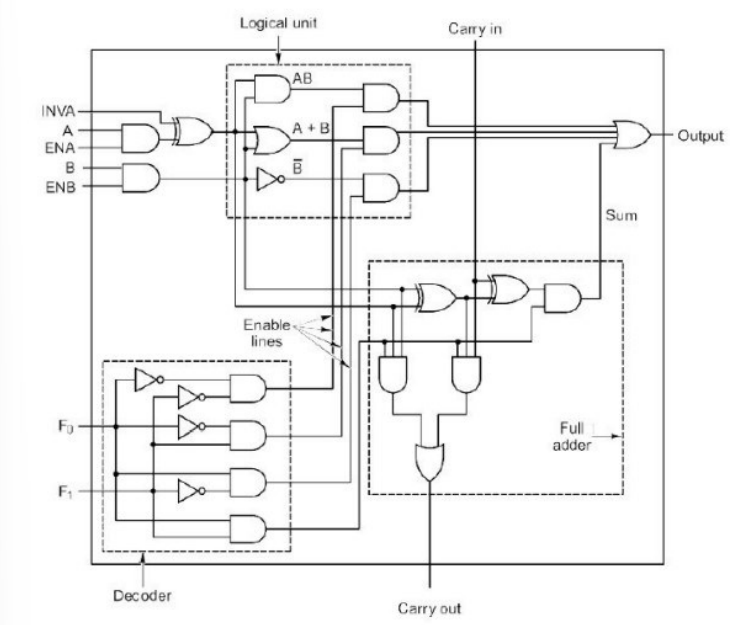
Per realizzare un comparatore per parole a più bit occorre confrontare i bit di uguale peso delle due parole, l'uscita segnalerà l'uguaglianza solo nel caso in cui tutti i bit corrispondenti risultano uguali.

- Comparatori ad 1 bit vengono collegati tramite una porta NOR
- L'output vale 1 solo se tutti gli output dei singoli comparatori ad 1 bit valgono 0 (Ai=Bi) per ogni i, cioè A=B

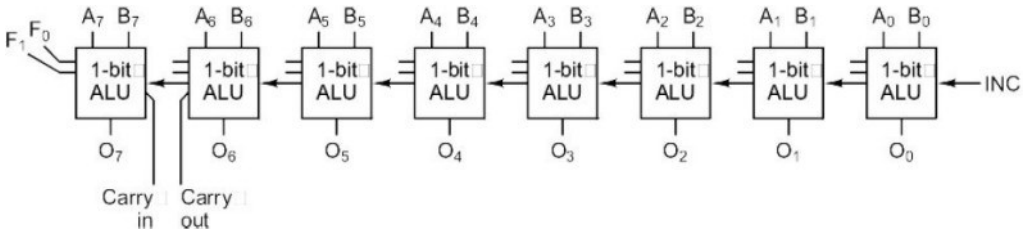
Traslatore (shifter: Trasla i bit in ingresso (D) di una posizione, a sinistra o a destra a seconda del valore del bit di controllo (C) (C=1 shift a destra)



Esiste una variante della ALU ad 1 bit, che è una semplice estensione a 32 bit di semplice realizzazione. Essa realizza 4 operazioni (selezionate da F_0 e F_1). AB, A or B, not(B), A+B. ENA, ENB: per forzare a 0 gli input A ed INVA, INVB: per invertire gli input.



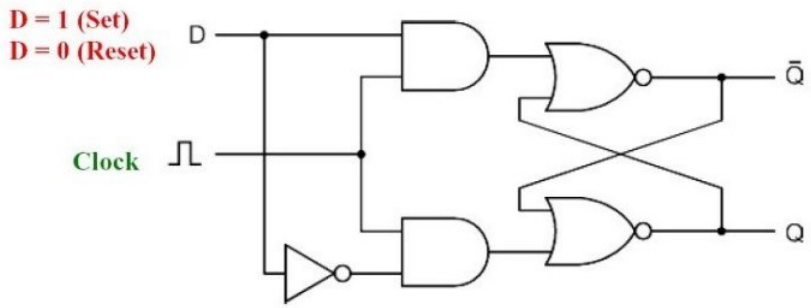
Concatenando *n* ALU ad 1 bit, si ottiene una ALU a n bit. F_0 e F_1 collegati a tutte le ALU, con riporto intermedio propagato da una ALU alla successiva ed INC (corrispondente al carry in della ALU "0") che permette di sommare 1 al risultato in caso di addizione.



Flip flop: Forma più semplice di una rete sequenziale. Tanti tipi, ma due proprietà per tutti:
Essi sono bistabili:

- possono trovarsi in uno di due stati diversi
- in assenza di input, rimangono nello stato in cui sono
- memoria per un bit e due output, in cui uno è sempre il complemento dell'altro

Esiste il flip-flop D, con solo input (D). Usa segnale di clock per stabilizzare l'output (sincronizzazione) e quando clock =0, gli output dei due AND sono 0 (stato stabile) e quando clock=1, gli input sono uno l'opposto dell'altro → Q=D



L'elemento fondamentale dei registri può essere considerato il flip-flop di tipo D che costituisce, a tutti gli effetti, una cella elementare di memoria. Esso è il circuito sincrono più semplice che realizza un registro.

- Memorizzazione (store): dati presentati in ingresso e clock da 0 a 1 (uscita riproduce ingresso)
- Mantenimento (hold): clock da 1 a 0 (poi costante); l'uscita rimane invariata indipendentemente dal valore degli ingressi

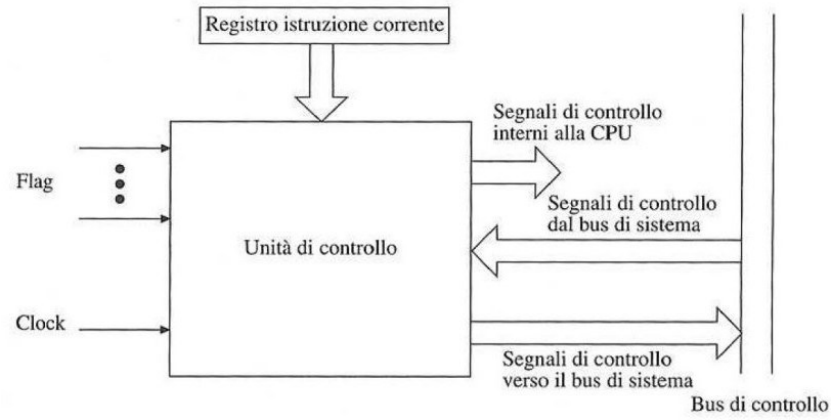
La CPU si fa carico di realizzare il flusso di controllo appropriato per ogni istruzione tramite l'invio di appositi segnali di controllo alla Parte Operativa attraverso l'unità di controllo (PC)
Vanno dunque:

- Definiti gli elementi di base del processore
- Definite le micro-operazioni eseguibili dal processore (trasferimento di dati tra i registri, oppure tra i registri e le interfacce e viceversa o esecuzioni di operazioni aritmetico-logiche)
- Determinare le funzioni che la PC deve effettuare per l'esecuzione delle micro-operazioni

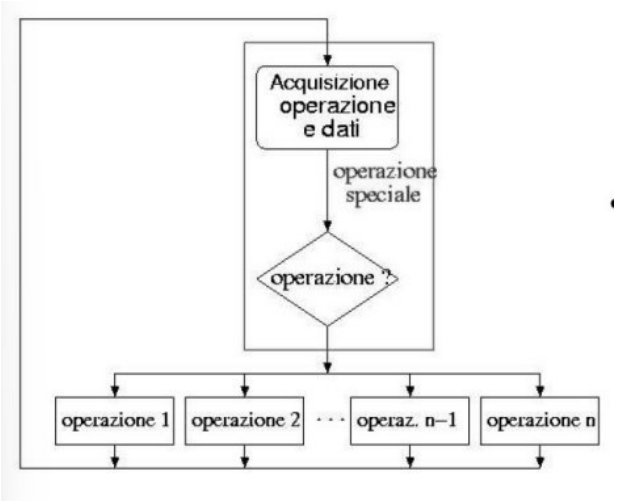
Gli elementi base sono:

- La ALU
- I registri
- Bus dati interno/esterno
- Unità di controllo

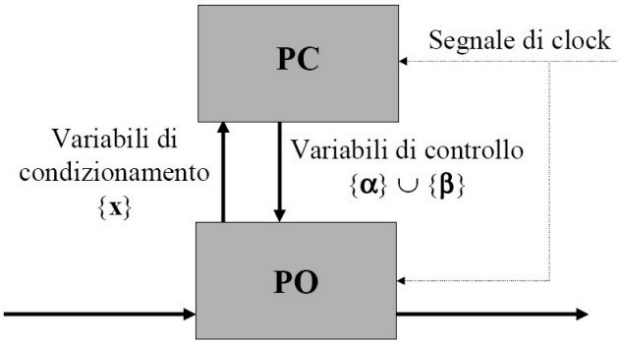
I compiti base della PC sono la serializzazione, che permette di determinare la giusta sequenza di micro-operazioni da eseguire in funzione del codice operativo dell'istruzione e per l'esecuzione delle micro-operazioni. Questo si realizza con appositi segnali di controllo.



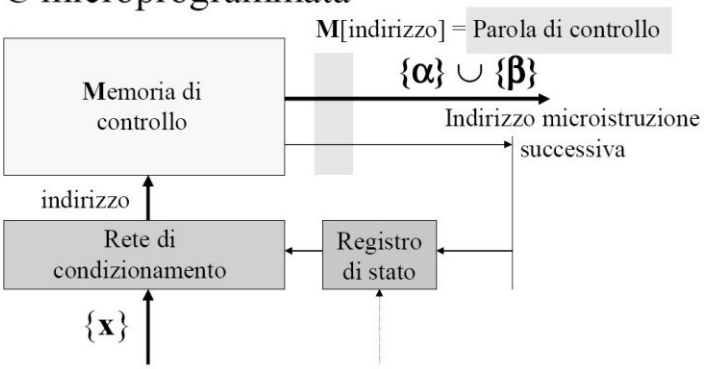
La PC può essere:
- Cablata, realizzata direttamente con circuiti digitali (architetture RISC)
- Microprogrammata, con maggiore flessibilità in fase di progettazione che rende facile modificare le sequenze di micro-operazioni (tramite microprogrammazione)



A livello firmware, il microprogramma riunisce i frammenti di programma delle diverse operazioni. Esso ha una struttura ciclica in cui si alterna l'esecuzione della operazione speciale con l'esecuzione della operazione esterna il cui codice e dati da elaborare sono stati acquisiti dalla operazione speciale.
Quindi, le variabili vengono controllate a cicli di clock, rendendo utile lo scambio tramite apposite variabili di condizionamento.



PC microprogrammata



{α} ∪ {β} designa la microoperazione richiesta

Altri esercizi pipeline

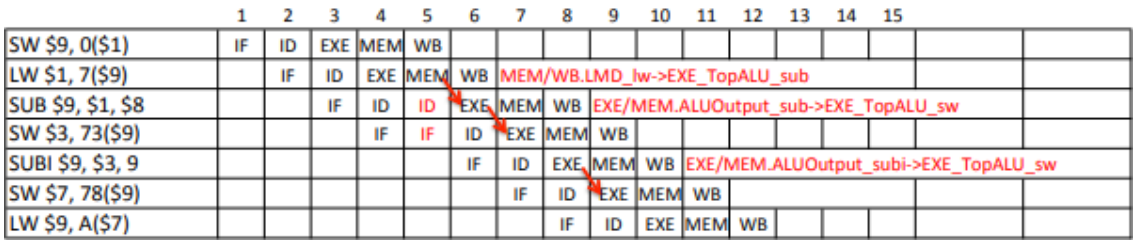
Sia data la seguente sequenza di istruzioni assembler, dove i dati immediati sono espressi in esadecimale

```
SW $9, 0($1)
LW $1, 7($9)
SUB $9, $1, $8
SW $3, 73($9)
SUBI $9, $3, 9
SW $7, 78($9)
LW $9, A($7)
```

Si consideri la pipeline MIPS a 5 stadi vista a lezione, con possibilità di data-forwarding e con possibilità di scrittura e successiva lettura dei registri in uno stesso ciclo di clock:

- mostrare come evolve la pipeline durante l'esecuzione del codice, spiegando nel dettaglio i motivi di un eventuale stallo o dell'utilizzo di un particolare circuito di by-pass.

Soluzione



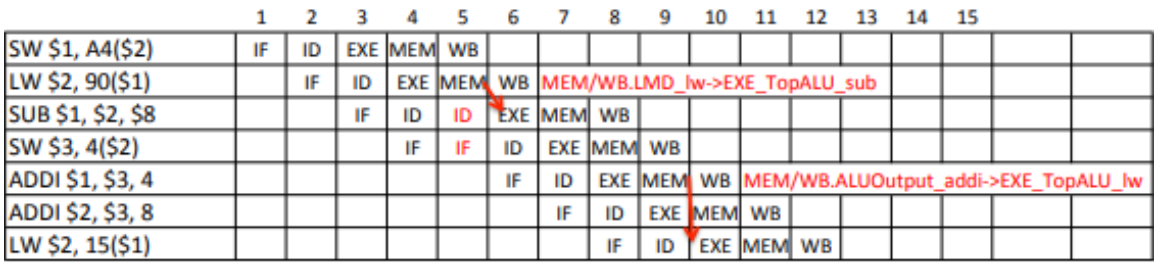
Sia data la seguente sequenza di istruzioni assembler, dove i dati immediati sono espressi in esadecimale

```
SW $1, A4($2)
LW $2, 90($1)
SUB $1, $2, $8
SW $3, 4($2)
ADDI $1, $3, 4
ADDI $2, $3, 8
LW $2, 15($1)
```

Si consideri la pipeline MIPS a 5 stadi vista a lezione, con possibilità di data-forwarding e con possibilità di scrittura e successiva lettura dei registri in uno stesso ciclo di clock:

- mostrare come evolve la pipeline durante l'esecuzione del codice, spiegando nel dettaglio i motivi di un eventuale stallo o dell'utilizzo di un particolare circuito di by-pass.

Soluzione



Sia data la seguente sequenza di istruzioni assembler, dove i dati immediati sono espressi in esadecimale

```
SUB $5, $1, $4
LW $1, 7($5)
ADD $5, $1, $8
LW $3, 73($5)
ADDI $5, $5, 3
SW $7, 78($3)
LW $5, A($7)
```

Si consideri la pipeline MIPS a 5 stadi vista a lezione, con possibilità di data-forwarding e con possibilità di scrittura e successiva lettura dei registri in uno stesso ciclo di clock:

- mostrare come evolve la pipeline durante l'esecuzione del codice, spiegando nel dettaglio i motivi di un eventuale stallo o dell'utilizzo di un particolare circuito di by-pass.

Soluzione

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15		
SUB $5, $1, $4	IF	ID	EXE	MEM	WB	EXE/MEM.ALUOutput_sub -> EXE.Top_ALU_input_lw											
LW $1, 7($5)		IF	ID	EXE	MEM	WB	MEM/WB.LMD_lw -> EXE.Top_ALU_input_add										
ADD $5, $8, $1			IF	ID	ID	EXE	MEM	WB	1) EXE/MEM.ALUOutput_add -> EXE.Top_ALU_input_lw 2) MEM/WB.ALUOutput_add -> EXE.Top_ALU_input_add							
LW $3, 73($5)				IF	IF	ID	EXE	MEM	WB	MEM/WB.LMD_lw -> EXE.Top_ALU_input_sw							
ADDI $5, $5, 3						IF	ID	EXE	MEM	WB							
SW $7, 78($3)							IF	ID	EXE	MEM	WB						
LW $5, A($7)								IF	ID	EXE	MEM	WB					