



ARCHITETTURA DEGLI ELABORATORI

Architettura degli Elaboratori Prof. Tramontana

Architettura degli Elaboratori

<https://www.dmi.unict.it/tramonta/ae/>

Appunti Architettura

1. FAMIGLIE DI CALCOLATORI

I/O

UNITA' DI MEMORIA

RETE DI INTERCONNESSIONE

PROCESSORE

Concetti operativi di base

Registri particolari

Interruzioni

Prestazioni

CENNI STORICI

Generazioni tecnologiche

2. INSTRUCTION SET ARCHITECTURE (ISA)

MEMORIA DEL CALCOLATORE

Organizzazione della memoria

Indirizzamento

Indirizzamento per byte

Ordinamento di byte

Operazioni di memoria

NOTAZIONE RTN

Notazione simbolica(assembly)

ARCHITETTURA RISC E CISC

Istruzioni RISC

Operazione di addizione

ESECUZIONE DI ISTRUZIONI

Esecuzione di n somme (senza salto)

Esecuzione con salto

Istruzione di salto

MODI DI INDIRIZZAMENTO RISC

Programma per la somma di una lista di numeri

LINGUAGGIO ASSEMBLATIVO

Direttive per l'assemblatore

Direttive per il caricatore

ORIGIN

RESERVE

DATAWORD

BGT

[Assemblaggio ed esecuzione](#)

NOTAZIONE PER I NUMERI (ASSEMBLY)

GESTIONE PILA

[Operazioni fondamentali sulla pila](#)

GESTIONE DEI SOTOPROGRAMMI

[Metodo di collegamento](#)

ANNIDAMENTO SOTOPROGRAMMI

[Passaggio di parametri in sottoprogrammi](#)

[Note su passaggio di parametri](#)

AREA DI ATTIVAZIONE

ULTERIORI ISTRUZIONI MACCHINA

[Scorrimento](#)

[Rotazione](#)

[Moltiplicazione e divisione](#)

VALORI IMMEDIATI A 32 BIT

INSIEMI DI ISTRUZIONI CISC

INDIRIZZAMENTI ULTERIORI

[Indirizzamento con modo relativo a PC](#)

BIT DI ESITO O CONDIZIONE

STILI RISC E CISC

CODIFICA DI ISTRUZIONI

[Formato con operando in registri](#)

[Formato con operando immediato](#)

[Formato per chiamata](#)

3.ACCESSO A DISPOSITIVI I/O

INTERFACCE DEI DISPOSITIVI I/O

LETTURA DI DATI DALLA TASTIERA

SCRITTURA CARATTERI SUL VIDEO

PROGRAMMA LETTURA DA TASTIERA E SCRITTURA A VIDEO (RISC)

[Esempio in stile CISC](#)

INTERRUZIONI

SERVIZIO DELLE INTERRUZIONI (ISR)

CONTROLLO DELLE INTERRUZIONI

[Dispositivi multipli](#)

[Annidamento interruzioni](#)

[Richieste di interruzione simultanee](#)

[Controllo della richiesta](#)

[Registro di controllo del processore](#)

[Gestore delle interruzioni](#)

CONCETTO DI ECCEZIONE

5.STRUTTURA DI BASE DEL PROCESSORE

[Componenti hardware di un processore](#)

[Hardware per l'elaborazione di dati](#)

[Struttura hardware a più stadi](#)

[Istruzione Load](#)

[Istruzioni Add](#)

[Istruzione Store](#)

HARDWARE: BANCO DI REGISTRI

UNITA' ARITMETICA LOGICA → ALU

STRUTTURA A CINQUE STADI

PERCORSO DATI (datapath)

PRELIEVO DELLE ISTRUZIONI (stadio 1)

GENERATORE DI INDIRIZZI DELLE ISTRUZIONI (PC)

[Istruzione add](#)

[Istruzione Load](#)

[Istruzione Store](#)

[Istruzione di salto incondizionato](#)

[Istruzione di salto condizionato](#)

ACCESSI ALLA MEMORIA E VELOCITA' DELLA MEMORIA

SEGNALI DI CONTROLLO

[Segnali al generatore di indirizzi](#)

TIPI DI CONTROLLO (RISC)

[Cablato](#)

RITARDO DELLA MEMORIA

PROCESSORI CISC

ORGANIZZAZIONE PROCESSORE CISC

[Add R5, R6](#)

[And X\(R7\), R9](#)

CONTROLLO MICROPROGRAMMATO

[Controllo cablato-microprogrammato](#)

6. PIPELINE IDEALE

ORGANIZZAZIONE PIPELINE

PROBLEMATICA PIPELINING

[Dipendenze di dato](#)

[Inoltro degli operandi \(HARDWARE\)](#)

[Gestione software di dipendenze di dati](#)

REMARK dipendenza di dato

[Ritardo della memoria e cache miss](#)

RITARDO NEI SALTI

[Salti incondizionati](#)

[Salti condizionati](#)

[Posto del ritardo del salto](#)

[Predizione di salti condizionati](#)

[Predizione statica di salto](#)

[Predizione dinamica di salto a 2 stati](#)

[Predizione dinamica di salto a 4 stati](#)

[Buffer di destinazione di salto](#)

LIMITI DI RISORSE

VALUTAZIONE DELLE PRESTAZIONI

[Effetti di stalli](#)

[Effetti di penalità di salto](#)

[Effetti di cache miss](#)

NUMERO DI STADI DELLA PIPELINE

FUNZIONAMENTO SUPERSCALARE

[Salvi e dipendenze di dato \(SUPERSCALARE\)](#)

[Esecuzione fuori ordine](#)

[Unità di smistamento](#)

[Pipeline processori CISC](#)

STRUTTURA A BUS

[Funzionamento del bus](#)

BUS SINCRONO

[Ritardi di propagazione](#)

[Trasferimento dati in più cicli](#)

BUS ASINCRONO

[Confronto fra sincrono e asincrono](#)

PILOTAGGIO BUS

ARBITRAGGIO DEL BUS

SISTEMI DI MEMORIA

[Introduzione](#)

[Concetti di base](#)

MEMORIA RAM A SEMICONDUTTORI

[Organizzazione interna di chip di memoria](#)

[Memoria statica cache](#)

[Memoria dinamica](#)

[Organizzazione RAM dinamiche](#)

[RAM dinamiche asincrone](#)

[RAM dinamiche sincrone](#)

MEMORIE ROM

MODULI DI MEMORIA

GERARCHIA DI MEMORIA

[Memoria cache e località](#)

USO DELLA CACHE

CACHE HIT

CACHE MISS

INDIRIZZAMENTO DIRETTO

INDIRIZZAMENTO ASSOCiativo

INDIRIZZAMENTO ASSOCiativo A GRUPPI

DATI SCADUTI IN CACHE (DMA)

ALGORITMO DI SOSTITUZIONE

CONSIDERAZIONI DI PRESTAZIONE

ESEMPIO STIMA DEL GUADAGNO

MIGLIORAMENTO DELLE PRESTAZIONI

PRESTAZIONI PER DUE LIVELLI DI CACHE

1. FAMIGLIE DI CALCOLATORI

Il linguaggio **assembly** è quel linguaggio comprensibile dall'uomo che più si avvicina al linguaggio macchina.

Il linguaggio macchina è quel linguaggio codificato e formato da tanti 0 e 1.

Esistono diversi **TIPI DI CALCOLATORI**:

- **Embedded**: programmati in modo definitivo con uno scopo ben definito. Essi **non sono riprogrammabili** e sono incorporati in un sistema più grande (ATM, lavatrici ecc..);
- **Personali**: ad uso personale (desktop), portatili o workstation;
- **Server**: usati per sistemi aziendali per gestire più utenti allo stesso tempo;
- **Supercalcolatori**: sorta di "armadi" con all'interno tanti processori montati su delle LAME. Sono costruiti da tanti processori per bilanciare il carico di lavoro tra le lame.

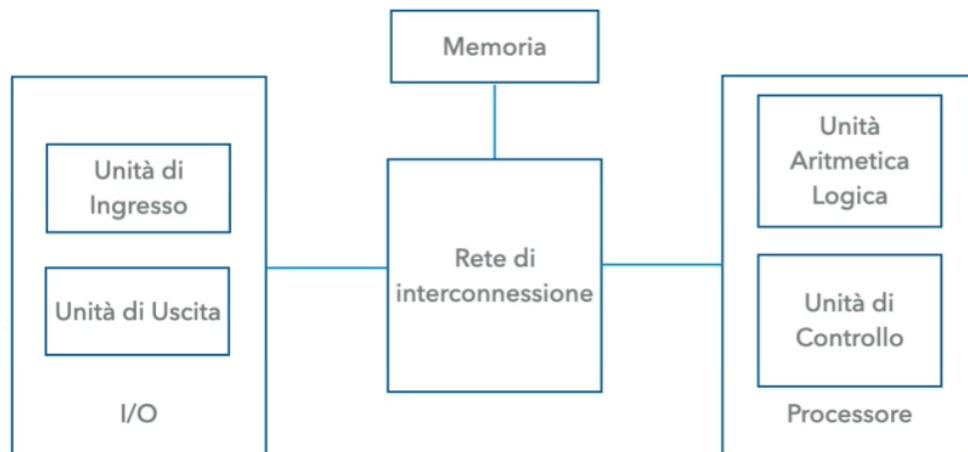
Il supercalcolatore più potente al mondo si trova a Kobe e ha 7.8 milioni di core e ogni processore ha 48 core.

I/O

- Le **Unità di Ingresso (input)** possono essere: tastiera, mouse, penna, touchscreen ecc..
- Le **Unità di Uscita (output)** possono essere: schermo, casse, proiettori.

Un dispositivo che unisce le unità Input/output è lo smartphone o qualsiasi altro dispositivo touchscreen.

*L'insieme di I/O è detto anche periferia → da questo proviene la parola **PERIFERICHE**.*



UNITA' DI MEMORIA

Memoria Principale (RAM dinamica): memoria volatile, ovvero i dati vengono persi se la memoria non è più alimentata dalla corrente elettrica. E' costosa e veloce. Il dato viene perso se non viene aggiornato costantemente. Ciò accade perchè, nelle celle, la corrente viene dissipata man mano.

Memoria Cache (RAM statica): memoria statica, ovvero i dati vengono salvati anche se la memoria non viene più alimentata dalla corrente. Più veloce della RAM dinamica. In questo caso non c'è bisogno dell'aggiornamento costante dei dati visto che vengono memorizzati e sono statici.

Si trova tra `memoria RAM dinamica ↔ processore`

Memoria di massa : memoria non volatile e i dati non vengono persi finché non eliminati dopo delle procedure. Il costo è basso ed è lento.

In un'unica operazione possono essere contenuti tanti bit (8, 16, 32, 64...) di dati. Ciò dipende dai vari componenti della macchina. Questa *unica informazione* è chiamata parola di memoria.

RETE DI INTERCONNESSIONE

E' composta dal **BUS** che trasporta i dati tra tutte le componenti funzionali.

PROCESSORE

E' formato dall'**Unità Aritmetica Logica ALU** e dall'**Unità di Controllo**.

→ **Unità Aritmetica Logica**: Si occupa delle operazioni aritmetiche. Prende i dati in considerazione e dà il giusto risultato;

Il processore è formato da dei *registri* che corrisponde alla memoria interna del processore. Essi sono organizzati in *banchi di registri* e proprio su questi lavora l'**ALU**. La dimensione dei registri va di pari passo con la parola di memoria. Per esempio: se la parola di memoria è 64bit, i dati saranno grandi 64bit e si sposteranno tutti in contemporanea.

→ **Unità di controllo**: Si occupa della coordinazione delle operazioni e della gestione del tempo dei flussi di informazioni.

Il processore, riassumendo, esegue le seguenti operazioni: lettura dell'istruzione in memoria → decodifica, comprensione del dato → governo dei dati e controllo di essi → mandata in output.

Il processore, inoltre, può venire interrotto da dei segnali esterni provenienti dalle periferiche o dal sistema operativo che fanno una "richiesta di servizio" durante il periodo di lavoro. Questa fase di interruzione momentanea è detta **INTERRUPT**.

Concetti operativi di base

Ci sono vari operandi di base in un codice:

- `Trasferimento dati Memoria-CPU:`

→ Load R2, LOC

| R2 → Destinazione; LOC → Indirizzo di memoria sorgente

"Prende il dato dall'indirizzo di memoria LOC e lo trasferisce nel registro R2"

- Trasferimento dati CPU→Memoria:

→ Store R4, LOC

| R4 → Sorgente; LOC → Indirizzo di memoria destinazione

"Prende il dato dal registro R4 e lo trasferisce all'indirizzo di memoria LOC"

- Operazione aritmetica della somma:

→ Add R2, R3, R4

| R2 → Destinazione; R3, R4 → Addendi

"Prende i registri addendi R3 ed R4, li somma e scrive il risultato nel registro R2"

Registri particolari

All'interno del processore ci sono vari registri. I più importanti sono:

- PC(program counter): contiene l'indirizzo dell'istruzione **successiva** e si aggiorna ogni volta che IR riceve la nuova istruzione;
- IR(instruction register): contiene l'istruzione **attuale** e la mantiene per tutta la durata dell'esecuzione.

Ricapitolando, i **passi** dell'istruzione macchina sono i seguenti:

1. **Prelievo** dalla memoria dell'istruzione puntata da PC e scrittura di essa in IR
2. **Incremento** di PC verso la successiva istruzione
3. **Decodifica**: comprensione da parte della macchina del tipo di istruzione ed emissione dei segnali necessari all'esecuzione di tale istruzione
4. **Esecuzione**: Load R2, LOC

Interruzioni

Durante la normale fase di lavoro del processore possono accadere imprevedibilmente degli **INTERRUPT**. Essi interrompono **momentaneamente** l'attività del processore per concentrarsi su dei segnali esterni a quel determinato processo proveniente dalle **periferiche I/O**.

Le interruzioni avvengono sempre con tempi non costanti. Ciò rende il processo di lavoro del processore **NON DETERMINISTICO**.

Prestazioni

Per prestazione si intende la velocità di lavoro del processore.

Essa dipende da:

1. **TECNOLOGIA DEL COMPONENTE** : i transistor presenti all'interno del processore devono poter emettere due tipi di segnali, 0 o 1. La **velocità di commutazione** del transistor determina le prestazioni del processore. Questa velocità dipende dalla **grandezza del transistor** stesso e più è piccolo, più còmmuta velocemente il suo stato.
2. **ORGANIZZAZIONE HARDWARE → Parallelismo a vari livelli:**
3. a *livello di istruzioni* esse vengono eseguite contemporaneamente in parallelo (**pipelining**)
4. a *livello di chip*: **multicore**, cioè ogni chip contiene più unità di elaborazione, detti **core**. Ogni core ha la sua **propria ALU e CACHE**.
5. a *livello di sistema*: **multiprocessori**. Fra i processori si ha la memoria condivisa(**shared memory**), mentre calcolatori collegati fra loro condividono dati scambiando messaggi(**message-passing**).

CENNI STORICI

- **Pascal** inventò la calcolatrice meccanica, denominata **Pascalina**.
- Nel XIX secolo le macchine diventano **programmabili**. Per esempio il programma di Lady Ada Lovelace.
- Nella prima metà del XX secolo compaiono i **modelli di calcolo**, come la macchina di Turing.
- Nei primi anni '40 compaiono i primi **calcolatori ABC**.

Generazioni tecnologiche

- I) Modello di von Neumann: programma scritto in **assembly** e velocità di operazioni di circa **1ms**
- II) Tecnologia del transistor e giunzione bipolare a semiconduttori(**BJT**), **linguaggi di alto livello**. Maggiori produttori di calcolatori come **IBM**
- III) Tecnologia dei circuiti integrati (**chip**). **Tanti transistor** che lavorano insieme. Il Sistema Operativo è **multiprogrammato** e c'è la memoria **cache** e memoria **virtuale**. Maggiori produttori **IBM e HP**
- IV) **Miliardi di transistor** in un unico chip(alta integrazione). Legge di Moore secondo il quale la densità di integrazione di un transistor **raddoppia** ogni **18 mesi**. Quindi ogni 18 mesi si potrebbe raddoppiare il numero di transistor. Maggiori produttori sono **AMD,STM,Intel**.

2. INSTRUCTION SET ARCHITECTURE (ISA)

Un processore sa eseguire **un insieme di istruzioni in formato eseguibile** ed esse sono dette **ISA**.

L'insieme ISA specifica:

- il **nome delle istruzioni** e le operazioni svolte;
- il **modo** con cui si possono **manipolare i dati**;
- le regole di **combinazione** delle varie **istruzioni**.

ISA è **l'interfaccia** fra hardware(che esegue le operazioni) e il software(che riceve le istruzioni dall'uomo)

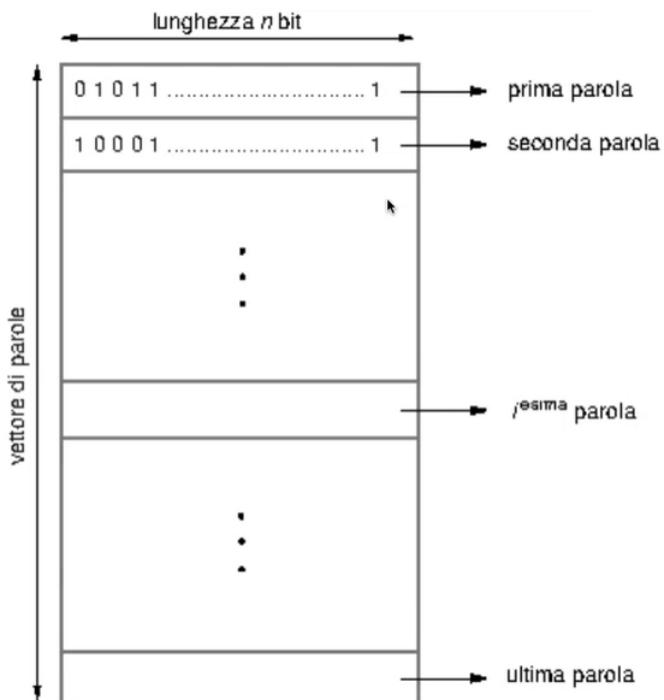
- *Il linguaggio assembly è una rappresentazione leggibile del linguaggio macchina.*
- *Il linguaggio di alto livello viene tradotto dal compilatore in linguaggio macchina affinchè questo venga eseguito dal processore.*
- *Nel linguaggio ad alto livello si fanno molte più cose rispetto al linguaggio assembly perché vi sono istruzioni sono più potenti.*

MEMORIA DEL CALCOLATORE

E' organizzata in tante **celle**
e ogni cella memorizza **un solo bit**.

Queste singole celle si
uniscono per formare
PAROLE DI MEMORIA
formate da 8,16,32,64 ... bit

Le parole di memoria sono
memorizzate una dopo
l'altra, **riga per riga**.
L'indirizzo più in alto è
l'indirizzo 0.



Organizzazione della memoria

Il bit più a destra della *parola di memoria* verrà chiamato *bit b₀*. Il successivo a sinistra bit b₁ fino al bit b_{n-1} dove n è la lunghezza della parola di memoria.

La parola di memoria può contenere numeri o caratteri ASCII (lettere, simboli, caratteri speciali).

Nel caso di una parola di memoria contenente uno o più caratteri ASCII, ogni carattere contiene 8bit, cioè 1 byte. Quindi se i caratteri ASCII sono 4, avremo 4 gruppi da 8bit per carattere per un

totale di 32bit.

Indirizzamento

Per **leggere** un dato dalla memoria (o **scriverlo**) è necessario conoscere l'indirizzo della parola di memoria. L'indirizzo è la locazione del dato nella memoria.

Per rappresentare un indirizzo servono m bit ed è un numero che va da 0 a $2^m - 1$

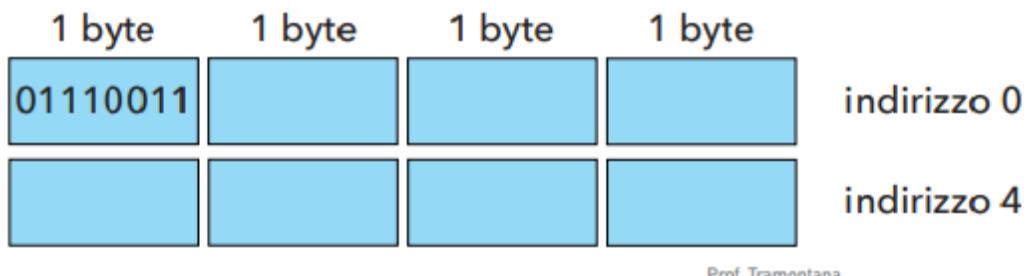
La **quantità totale di indirizzi** viene detta **spazio di indirizzamento**.

Se avessi $m=24$, si usano 24 bit per rappresentare un indirizzo. Si possono generare 2^{24} parole di memoria, ovvero circa 16 M parole (16 Mega parola).

Indirizzamento per byte

Ogni indirizzo individua 1 byte e non l'intera parola di memoria (32bit=4byte ← esempio).

Data una serie di caratteri *ABCDE*, ogni lettera ha 1 byte di dati e ogni singola lettera può essere letta/scritta dal processore.



Se si scrive un comando riguardo l'indirizzo 0, il processore preleva tutta la parola sulla prima la riga.

Se si scrive un comando riguardo l'indirizzo 4, il processore preleva tutta la parola sulla seconda riga.

Quindi tra 2 righe di parole c'è un salto di 4 indirizzi.

Se si scrive un comando riguardo il byte 1, il processore prende in considerazione il dato in posizione 1 sulla prima la riga.

Eventualmente si può anche prelevare il singolo byte presente all'indirizzo 0 o 4 o 8 ecc..

Ordinamento di byte

	indirizzo di byte			
indirizzo di parola	0	1	2	3
0	0	1	2	3
4	4	5	6	7
	•	•	•	•
$2^m - 4$	$2^m - 4$	$2^m - 3$	$2^m - 2$	$2^m - 1$

	indirizzo di byte			
indirizzo di parola	0	1	2	3
0	3	2	1	0
4	7	6	5	4
	•	•	•	•
$2^m - 4$	$2^m - 1$	$2^m - 2$	$2^m - 3$	$2^m - 4$

(a) schema crescente o big-endian (b) schema decrescente o little-endian

Ci sono due possibili ordinamenti dei byte nella memoria indirizzabile per byte:

- a) i byte sono disposti in ordine **crescente** da **sinistra verso destra**;
- b) i byte sono disposti in ordine **crescente** da **destra verso sinistra**.

Soltamente il bit più significativo sta a destra, cioè per un byte si avrà b7,b6..b0

Se dovessimo far comunicare 2 computer sicuramente bisognerà fare le conversioni dell'ordinamento dei byte se il secondo pc utilizza uno schema diverso da quello di provenienza. Per esempio potrebbe esserci anche un modo crescente di ordinamento dei byte rispetto a uno decrescente.

Quando una parola di memoria finisce con un multiplo della lunghezza della parola in byte si dice che lo **schema di indirizzamento è allineato**.

Il processore può prelevare una parola di memoria emettendo l'indirizzo 0,4,8 ecc, quindi da uno schema allineato, oppure potrebbe prelevare la stessa parola di memoria emettendo l'indirizzo 1 (prelevando 3 byte dalla prima riga e 1 byte nella riga successiva) quindi l'indirizzo non è allineato in questo caso. Si preleva una parte di una riga e una seconda parte di un'altra riga.

Se si preleva l'intera riga l'indirizzo è allineato. Se si preleva un pezzo di riga e poi un altro pezzo nella seconda riga allora l'indirizzo non è allineato.

Operazioni di memoria

Le due operazioni di base sono:

- **LOAD**, prelievo o fetch (istruzione di lettura): il dato viene **letto** dalla memoria ad un certo indirizzo della parola di memoria e spostato dentro il processore, cioè scritto dentro un registro del processore.
- **STORE**, o scrittura. si vuole **scrivere** un dato nella memoria prelevato da un certo registro del processore.

Inoltre ci sono 4 categorie di istruzioni:

- trasferimento, dati **memoria → processore** o viceversa
- **operazioni aritmetiche e logiche** su dati conosciuti all'interno del processore: somma, moltiplicazioni e inverse a loro. Operazione **AND, OR** tra 2 bit ecc
- controllo della sequenza di esecuzione delle istruzioni,
- trasferimento di dati tra unità I/O e processore.

*A volte è utile non seguire la sequenza di istruzioni presente in un programma e quindi si introduce un'istruzione di **SALTO** che permette di saltare le successive istruzioni o le precedenti se si decide di ritornare su qualche riga precedente del codice.*

NOTAZIONE RTN

La **Notazione di Trasferimento dei Registri** indica la copia di dati fra locazioni di memoria e registri. Si usano:

- ▶ Costanti numeriche o simboliche per indirizzi di parole di memoria, es. IND
- ▶ Nomi predefiniti per i registri del processore, es. R1, R2, etc.
- ▶ Parentesi quadre per indicare il contenuto della memoria indirizzata, es. [IND] è il contenuto della memoria alla locazione di indirizzo IND

R2 ← [LOC]

▶ Indica il trasferimento nel registro R2 del contenuto della memoria di indirizzo LOC

R4 ← [R2] + [R3]

▶ Indica il trasferimento in R4 della somma dei contenuti dei registri R2 e R3

▶ A sinistra della freccia deve esserci un solo elemento, capace di contenere un valore, quindi un registro o una locazione di memoria. Solo la locazione di destinazione viene cambiata.

La freccia verso sinistra ← indica il trasferimento, ovvero una copia.

Notazione simbolica(assembly)

Ogni istruzione è **un'istruzione operativa** LOAD, STORE, ADD e questa ha degli **operandi** cioè i dati su cui compiere **quell'operazione**.

Il linguaggio Assemblativo è adatto a rappresentare il linguaggio macchina.

I dati possono essere registri, locazioni di memoria ecc e si ha, per esempio:

Load R2, LOC

► col significato di $R2 \leftarrow [LOC]$

Add R3, R2, R1

► col significato di $R3 \leftarrow [R2] + [R1]$

ARCHITETTURA RISC E CISC

Ci sono varie tipologie di realizzazione di processori:

RISC comprende un set di istruzioni per computer **ridotte**:

- ogni istruzione occupa esattamente **una parola di memoria**, ha un hardware meno complesso ed è più veloce
- visto che lo spazio per un'istruzione è fissato possiamo codificare **meno istruzioni**, quindi bisognerebbe ridurre il numero di istruzioni;
- se dovessimo programmare in assembly dovremmo mettere più istruzioni per eseguire una determinata operazione perché **non c'è una 'macro-istruzione'** per eseguire operazioni più complesse.

CISC comprende un set di istruzioni per computer **complesse**:

- un'istruzione occupa **più di una parola di memoria** in base alla necessità di codifica, quindi la lunghezza dell'istruzione è variabile;
- maggiore complessità di esecuzione perchè la cpu deve anche calcolare la lunghezza della parola di memoria come istruzione aggiuntiva;
- si ha la possibilità di codificare molte più cose e possiamo avere una '**macro-istruzione**' per eseguire un'operazione più complessa.
- Realizzazione della CPU è più complicata

| ESEMPIO: 2 istruzioni per una cpu CISC equivale a 4 istruzioni RISC.

Istruzioni RISC

L'istruzione occupa **una sola parola di memoria** ed è un'architettura **load/store**. Si chiama così perchè:

1. l'accesso a operandi in memoria avviene **solo** tramite le istruzioni **load** e **store**;
2. gli operandi stanno nei registri oppure, in modo immediato, dentro la parola dell'istruzione stessa.

L'istruzione specifica quale registri utilizzare per l'operazione da eseguire.

FORMATO DI ISTRUZIONI (**d**=destinazione; **s**=sorgente)

```
Load d, s
Store s, d
Add d, s1, s2
```

1. Nella **load** la destinazione è sempre un registro e la sorgente è la memoria
2. Nella **store** la sorgente è sempre un registro e la destinazione è la memoria
3. Nella **add** la destinazione è un registro e gli altri due operandi solo delle sorgenti e sono **SEMPRE REGISTRI**.

Operazione di addizione

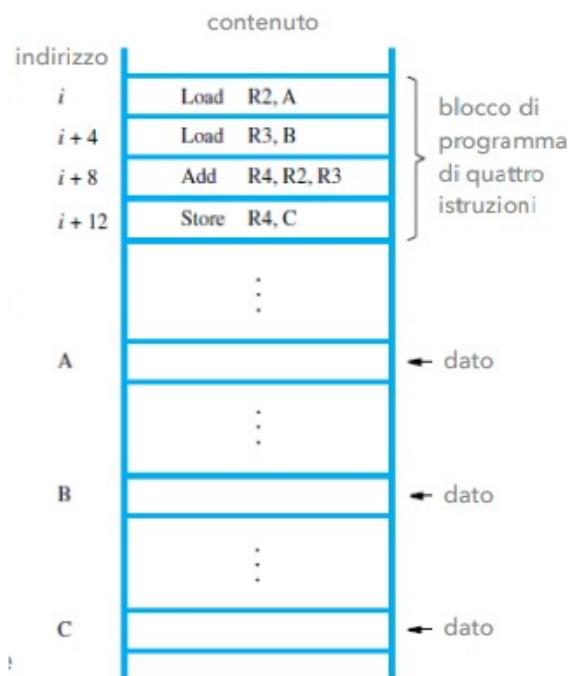
Sia dato $C=A+B$ allora la notazione **RTN** sarà:

$C-[A]+[B]$ e queste 3 variabili stanno in memoria.

In linguaggio **ASSEMBLY RISC** viene:

```
Load R2, A
Load R3, B
Add R4, R2, R3
Store R4, C
```

ESECUZIONE DI ISTRUZIONI



Il programma da eseguire è in memoria e deve essere codificato opportunamente.

Le **istruzioni** sono contenute in parole di memoria **consecutive**, iniziando dall'indirizzo i per la prima riga, $i+4$ per la seconda riga ecc..

Nel registro **PC** ci sarà la *prossima istruzione da eseguire*, in questo caso esso deve puntare l'istruzione i , cioè la prima.

Viene **prelevata(fetch)** l'istruzione da **PC** ed esso viene incrementato di 4 (quindi aggiornata) e viene inserita nel registro **IR** che contiene l'istruzione '**attuale**'. Questo procedimento si puo scrivere:

IR \leftarrow **[[PC]]**

PC ha due parentesi quadre perchè si deve prendere il valore dell'indirizzo di [PC] e utilizzarlo come puntatore per la prossima istruzione [[PC]].

Dopo il prelievo fetch avviene **l'esecuzione** dell'istruzione. **IR** viene decodificato e l'istruzione viene eseguita. (*l'esecuzione può comportare dei calcoli e attivita ausiliarie*).

Esecuzione di *n* somme (senza salto)

```
/*Dati n>1 numeri da sommare. Gli indirizzi saranno indicati con NUM1,NUM2 ecc e  
la locazione di memoria  
della somma sarà SOMMA*/  
  
Load R2, NUM1 //carica dato presente all'indirizzo NUM1  
Load R3, NUM2 //carica dato presente all'indirizzo NUM2  
Add R2, R2, R3 //Somma i dati in R2 e R3, scrive in R2  
Load R3, NUM3  
Add R2, R2, R3  
...  
Load R3, NUMn  
Add R2,R2,R3  
Store R2, SOMMMA
```

se avessi n=100 ci sarebbero fin troppe righe di codice, quindi bisogna trovare una soluzione a questo. Proprio per questo motivo si introduce **l'esecuzione con salto**.

Esecuzione con salto

Anzichè scrivere n volte la somma, si usa il ciclo che va a ripetere determinate operazioni. In un linguaggio di basso livello si ha:

N, **CICLO** e **SOMMA** sono gli indirizzi di memoria costanti;

R2 sarà il contatore del numero di passate del ciclo. **R3** è usato come accumulatore (somma parziale). **R5** contiene il numero da sommare.

Il # indica che il valore subito dopo è un **valore immediato** da utilizzare, #1 prende in considerazione 1 per un'operazione.

```
Load R2, N  
Clear R3 //mi assicuro che R3 sia a 0 "ripulendolo"  
CICLO: Determina l'inidirizzo del prossimo numero  
Carica il prossimo numero in R5 //CICLO è un'etichetta per marcare quel punto  
Add R3, R3, R5  
Subtract R2, R2, #1 //sottrae R2-1  
Branch_if_[R2]>0 CICLO //SALTO CONDIZIONALE. verifica la condizione e se è vera esegue il salto  
Store R3, SOMMA
```

L'istruzione **Branch** è l'unica istruzione capace di **cambiare il valore del PC** (program counter).

I salti possono essere sia verso un'istruzione precedente e sia verso un'altra istruzione successiva mettendo altre etichette nel corso del programma

Istruzione di salto

`Branch_if_[R2]>0 CICLO` ← se "[R2]>0" risulta **vera** si salta all'etichetta **CICLO**. Se risulta falsa non si prende affatto in considerazione. Essa sovrascrive **PC** col nuovo valore e questo indirizzo è detto **destinazione di salto**. L'istruzione di salto, chiaramente, interrompe il processo sequenziale del codice.

L'istruzione di salto si serve dall'indirizzo simbolico **CICLO**.

`Branch_if_[R2]>[R3] CICLO` ← è un confronto fra registri

Si usa l'istruzione di salto per tornare indietro nel programma verso l'etichetta scritta in precedenza (si può saltare anche in avanti).

MODI DI INDIRIZZAMENTO RISC

La Load, ad esempio, specifica gli operandi `Load R2, N`. Dal modo con il quale si specificano gli operandi possiamo distinguere vari modi di indirizzamento.

Modo di registro: indichiamo il dato sorgente(**operando**) o la **destinazione del risultato** attraverso i **registri**: `Add R3, R4, R5`

Modo diretto(o assoluto): specifico come **operando il contenuto di una parola di memoria** e questa viene individuata attraverso il nome di una variabile che verrà trasformata dall'assemblatore in un indirizzo. `Add R2, LOC`.

Modo indiretto: l'operando della sorgente **[R5]** viene usato come puntatore alla memoria. Quindi, all'indirizzo presente in R5 ci sarà il **contenuto** del dato che bisogna essere prelevato.

`Load R3, [R5]` → R3=24B9

`Load R3, R5` → R3=04547

indirizzo valore in LOC

R5 04547 24B9

In notazione RTN si usano le parentesi quadre per indicare che come operando si vuole il valore contenuto in quella variabile. `[LOC]`

Modo immediato(o costante): nell'istruzione scriviamo il **valore diretto dell'operando**. `Add R4, R6, #200`

Modo con indice e spiazzamento

Si usa l'**indice (registro)** e uno **spiazzamento numerico** che deve essere sommato alla parola di memoria puntata dall'indirizzo del registro indicato. `Load R2, 20(R5)`

"20" si somma al valore puntato da R5 e il risultato viene trasferito in R2. Quindi la sorgente è proprio la parola di memoria contenuta all'indirizzo R5.

Generalmente si ha in RTN:

`EA = X + [Ri]` dove EA è l'**indirizzo effettivo** e X è una costante **numerica** decisa da noi da sommare e [Ri] è il valore del registro selezionato.

► L'istruzione sopra indica **spiazzamento 20** e il registro R5 usato come **registro indice**, se in R5 vi è il valore 1000, si preleva il contenuto della locazione 1020. Ri è, chiaramente, variabile.

Modo con base e indice

In questo caso si ha `(Ri, Rj)` e i registri vengono sommati fra loro. Il risultato sarà l'indirizzo effettivo. E' utile per variare i valori in uno dei vari registri se vogliamo risultati diversi, quindi abbiamo più possibilità di aggiornare i puntatori alla memoria piuttosto che avere valori costanti.

`EA=[Ri]+[Rj]` (EA=Indirizzo effettivo)

In questo caso lo spiazzamento non è più cosante ma diventa variabile.

Programma per la somma di una lista di numeri

Load	R2, N	Carica dimensione lista
Clear	R3	Inizializza la somma a 0
Move	R4, #NUM1	Carica indirizzo primo numero
CICLO: Load	R5, (R4)	Preleva prossimo numero
Add	R3, R3, R5	Aggiungi numero alla somma
Add	R4, R4, #4	Incrementa puntatore a lista
Subtract	R2, R2, #1	Decrementa contatore
Branch_if_[R2]>0	CICLO	Salta indietro se non ha finito
Store	R3, SOMMA	Immagazzina somma finale

MOV	R2, #N
MOV	R3, #0
MOV	R4, #NUM1
CICLO	LDR R5, [R4]
ADD	R3, R3, R5
ADD	R4, R4, #4
SUB	R2, R2, #1
CMP	R2, #0
BNE	CICLO
STR	R3, [R4]

#NUM1 indica il primo numero nella lista dei numeri da sommare.

Move vs Load.

- **Load** sposta un qualcosa dalla memoria al registro. Accede alla memoria direttamente.
- **Move copia nel registro** il valore scritto nell'istruzione e non accede alla memoria.

► **Move** potrebbe essere una pseudo istruzione, effettuata con **Add R4, R0, #NUM1** dove R0 vale 0 (per convenzione)

► **Clear** potrebbe essere sostituito con **Add R3, R0, R0** per azzerare il valore di R3.

- R4 viene utilizzato come puntatore alla memoria. Il dato contenente all'indirizzo di R4 verrà puntato e trasferito in R5
- Add R4, R4, #4 si incrementa di 4 perchè l'indirizzamento è a byte e una parola di memoria ha 4 byte quindi per passare alla prossima si salta di 4. Si aggiorna affinchè il puntatore si aggiorni e punti al prossimo valore che si trova nella prossima parola di memoria.

Nel riquadro a destra:

- **CMP** è una Comparazione fra un registro e un valore (0).
- **BNE:** se i valori del CMP non sono uguali, si **salta** l'istruzione verso l'etichetta selezionata. Se sono uguali si prosegue verso la STR.

Nell'operazione di Add **non si possono inserire due valori immediati** allo stesso tempo visto che, per sintassi, un **operando è un registro** e l'altro operando può essere un può essere indicato con qualsiasi altro modo di indirizzamento.

*La **dereferenziazione** permette di estrarre il valore puntato dal puntatore che punta su una locazione di memoria senza accedere direttamente ad essa.*

*In linguaggio 'C' si indica con $A = *B$ dove $*B$ è il puntatore. In assembly si traduce:*

Load **R2, B**

Load R3, **(R2)**

Store R3, A

LINGUAGGIO ASSEMBLATIVO

E' più o meno uguale a tutti gli assemblatori.

L'assemblatore analizza la **correttezza** del programma e **traduce** le istruzioni in **linguaggio macchina**. L'istruzione contiene sempre un **codice operativo** (codice mnemonico) e a questo seguono degli **operandi** (registri, valori immediate, puntatori).

In questo linguaggio non esistono **TIPI** di variabili.

Direttive per l'assemblatore

Le direttive sono istruzioni composte da parole o parametri e non sono istruzioni vere e proprie. Possiamo definirli come dei comandi che noi mettiamo nel programma e sono utili per dire qualcosa all'assemblatore o al caricatore che legge il programma dal disco e lo trasferisce in memoria.

VENTI EQU 30 30 è una variabile e il valore corrispondente alla variabile è 30 in questo caso.

Sostituisce il valore 30 ogni volta che compare l'etichetta VENTI nel programma. (**EQU⇒EQUATE**)

	Etichetta di indirizzo di memoria	Operazione	Indirizzi o dati	
Direttiva di assemblatore		ORIGIN	100	
Dichiarazioni che generano istruzioni macchina		LD	R2, N	100 Load R2, N
		CLR	R3	104 Clear R3
		MOV	R4, #NUM1	108 Move R4, #NUM1
	CICLO:	LD	R5, (R4)	112 Load R5, (R4)
		ADD	R3, R3, R5	116 Add R3, R3, R5
		ADD	R4, R4, #4	120 Add R4, R4, #4
		SUB	R2, R2, #1	124 Subtract R2, R2, #1
		BGT	R2, R0, CICLO	128 Branch_if_[R2]>0 CICLO
		ST	R3, SOMMA	132 Store R3, SOMMA
			prossima istruzione	:
Direttive di assemblatore		ORIGIN	200	
	SOMMA:	RESERVE	4	
	N:	DATAWORD	150	200 → 150
	NUM1:	RESERVE	600	204 → 600
		END		208
				212
RESERVE dichiara uno spazio dati in byte				
DATAWORD indica il valore da inserire in memoria				
alla locazione con l'etichetta N				
Prof. Tramontana				
NUM _n 804				

Direttive per il caricatore

ORIGIN

Indica l'indirizzo dal quale bisogna iniziare a memorizzare i dati.

Il caricatore che legge il programma dal file sa da quale punto di memoria deve far memorizzare i dati. Indica anche a partire da dove saranno memorizzate le variabili che si useranno nel programma. Tutto quello che viene sotto si deve caricare dopo l'indirizzo specificato: ORIGIN 100 vuol dire che si inizierà dall'indirizzo 100.

ORIGIN si può usare più di una volta nello stesso programma.

RESERVE

Indico all'assemblatore che bisogna lasciare a disposizione per la variabile indicata il numero di byte specificato successivamente nell'operando.

NUM1: RESERVE 600 riserva 600byte alla variabile con etichetta NUM1.

SOMMA RESERVE 4. Questa direttiva dichiara uno spazio dati in byte per una variabile. Riserva 4 byte per la variabile SOMMA.

DATAWORD

Il dato deve essere esattamente lungo una parola di memoria e gli definiamo il valore con l'operando (150). Questa direttiva indica il valore da inserire in memoria alla locazione con etichetta 'N':

N: DATAWORD 150 → *N sarà una parola di memoria e avrà valore 150*

l'indirizzo iniziale che indico con num1 avrà indirizzo 208.

BGT

BGT salta se maggiore di 0 (quindi considera l'istruzione successiva verso il basso)

Assemblaggio ed esecuzione

Il caricatore, dopo aver tradotto il programma, scrive il programma in **versione eseguibile** (versione oggetto) su un file e quest'ultimo viene memorizzato in memoria. Il file che contiene il programma viene caricato grazie al SO in memoria subito dopo che vogliamo eseguirlo.

Direttiva **START**. Indica il punto di avvio del programma. E' posta **in testa al file** e specifica **l'indirizzo della prima istruzione** del programma. Sarà quindi la prima istruzione ad essere eseguita all'avvio del programma.

Modalità debug: L'esecuzione del programma **si ferma** dopo aver completato la **prima istruzione del programma** e **ogni istruzione successiva**. Permette di avere degli ambienti di esecuzione controllati del programma e vedere alla fine di ogni istruzione come variano i valori dei registri affinchè sia più semplice scovare errori.

NOTAZIONE PER I NUMERI (ASSEMBLY)

I numeri possono essere rappresentati varie forme: esadecimale, binario ecc.. Abbiamo due metodi:

1. **%n** per i valori binari;
2. **0x** per i valori esadecimali.

Questi due metodi sono delle direttive che permettono all'assemblatore di capire in che base è scritto un numero.

GESTIONE PILA

La **pila** è una struttura dati che si sviluppa **verticalmente** dove si ha una lista di elementi. Essi si possono:

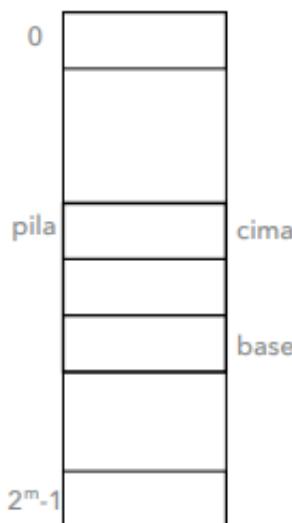
- inserire solo dall'alto;
- si possono prelevare solo dall'alto.

La pila si riempie, progressivamente, dal basso verso l'alto.

La **base** della pila indica il **fondo** di questo contenitore. Man mano che si inseriscono elementi la pila cresce verso l'alto e **l'ultimo elemento** sta in **cima** alla pila.

Ogni volta che si inserisce un elemento si aggiorna la cima della pila.

L'indirizzo di memoria della base è il più alto, mentre l'indirizzo di memoria della cima della pila è il più basso.



La pila adotta il sistema **LIFO**, cioè **Last-In-First-Out**. Sta a significare che se un dato è l'ultimo ad entrare allora sarà il primo ad uscire.

E' prelevabile solo l'ultimo elemento inserito nella pila. (*poi ovviamente, in successione, tutti gli altri*)

Operazioni fondamentali sulla pila

1. **push**: serve per **inserire** un nuovo elemento in cima alla pila;
2. **pop**: serve per **prelevare** l'elemento in cima alla pila.

L'uso della pila avviene con il registro speciale chiamato **Stack Pointer (SP)**. Esso è il registro che punta alla cima della pila.

Per utilizzare l'operazione di **push** bisogna:

1. alzare la cima di 4 per creare lo spazio per il dato;
2. inserire il dato.

```
Subtract SP, SP, #4  
Store Rj, (SP)
```

Per utilizzare l'operazione di **pop** bisogna:

1. caricare in un registro il dato che ci interessa;

2. abbassare la pila di 4.

```
Load Rj, (SP)
Add SP, SP, #4
```

GESTIONE DEI SOTTOPROGRAMMI

Un sottoprogramma è una **routine** che viene eseguita **più volte** nello stesso programma. La funzione chiamante verrà eseguita finchè la funzione chiamata non avrà terminazione.

CALL SUB permette di chiamare il sottoprogramma.

SUB : è il *nome* del sottoprogramma, ovvero **l'etichetta**.

CALL : è la **direttiva** di chiamata.

E' come un salto e la routine sarà presente ad un determinato indirizzo all'interno dello stesso programma.

Per indicare la fine di una routine è necessario inserire un'istruzione speciale che ne indichi la terminazione.

Questa istruzione di terminazione deve permettere di tornare al programma chiamante e proseguire con le istruzioni scritte che si trovano subito dopo CALL.

L'**istruzione return** indica la fine delle istruzioni routine e permette di saltare subito dopo l'istruzione CALL per riprendere il programma e uscire dalla routine.

L'**istruzione di rientro** indica l'indirizzo di ritorno del programma chiamante. L'indirizzo del **punto di rientro** è il valore del registro PC quando si esegue l'istruzione di chiamata a routine.

La routine può essere chiamata da vari punti del programma chiamante.

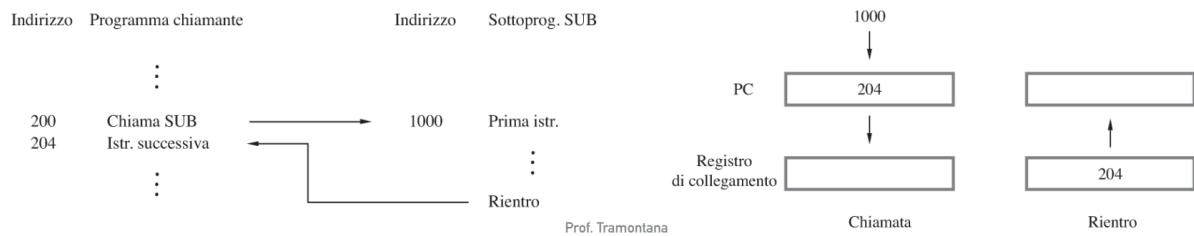
Metodo di collegamento

Il **Link Register (LR)**, o il registro di collegamento, è quel registro che prende l'indirizzo di PC prima che si effettui la sub routine. In questo modo si conserva l'indirizzo di ritorno. Dopodichè, quando la sub routine arriva all'**istruzione di ritorno**, PC viene aggiornato con il nuovo indirizzo della sub routine.

Alla fine della sub routine PC prende l'indirizzo di LR precedentemente conservato così che punti alla prossima istruzione dopo il CALL.

LR \leftarrow PC

PC \leftarrow LR quando la routine termina



ANNIDAMENTO SOTTOPROGRAMMI

Una routine puo' chiamare a sua volta un'altra routine.

Viene perso un valore in LR perchè servono 2 posti per l'indirizzo di rientro.

Per questo si dovrebbero usare tanti LR. E' infattibile perchè la CPU non ha migliaia di registri per prevedere questo caso.

In questo caso, quando ci sono le **chiamate annidate**, si ricorre alla **pila**. Essa permette di prelevare come primo dato l'ultimo dato che abbiamo inserito. Quando si trova **return**, si preleva il valore in cima alla pila. Più già esiste l'altro indirizzo di rientro che abbiamo salvato precedentemente.

I processori possono essere realizzati anche senza LR perchè possono lavorare solo con la pila, ma l'importante è che salvino nella pila l'indirizzo di rientro.

Questo passaggio di parametri avviene senza che il programmatore debba salvare a mano gli indirizzi di ritorno perchè avviene in modo automatico. In questo caso si parla di **uso della pila in modo implicito**.

*In caso di chiamate annidate ci pensa il processore a mettere gli indirizzi di ritorno e avviene in modo automatico. Per venire a conoscenza di come avviene questo passaggio di parametri bisogna leggere il **manuale del processore** e capire se bisogna scrivere un'istruzione apposita o no.*

Passaggio di parametri in sottoprogrammi

Ci sono due modi per passare parametri o valori di ritorno. (**return**)

- tramite **registri**;
- tramite la **pila**;
- I **registri sono limitati** quindi i parametri da passare possono essere limitati.
- La pila permette di passare un **qualsiasi numero di parametri**.

La pila, quindi, permette di liberare alcuni registri per lasciarli liberi ad altri usi.

Essa esegue accesso alla memoria quindi Load ci mette un po' di tempo per venire eseguita.

Il chiamante deve mettere nella pila, in condizione adeguate, le istruzioni mentre, il chiamato, deve prendere i valori dalla cima della pila.

Passaggio di parametro tramite valore: Load R2, N (es: N=dimensione di una lista) tutte le variazioni **NON** sono viste anche dal chiamante.

Passaggio di parametro tramite indirizzo: Move R4, #NUM1 (es: locazione di una lista), tutte le variazioni sono viste anche dal chiamante.

Note su passaggio di parametri

`StoreMultiple R2-R5, -(SP)` Memorizzare registri da R2 a R5 partendo dalla posizione SP decrementandola di 4 ogni volta, cioè decrementandola di una parola di memoria.

`LoadMultiple R2-R5, (SP)+` carica i valori da R2 a R5 a partire da SP. SP viene incrementato dopo che il primo caricamento è stato fatto. Viene incrementato affinchè punti alla prossima parola di memoria.

Ci pensa l'assemblatore a tradurre le istruzioni sopra citate in istruzioni Load o store.

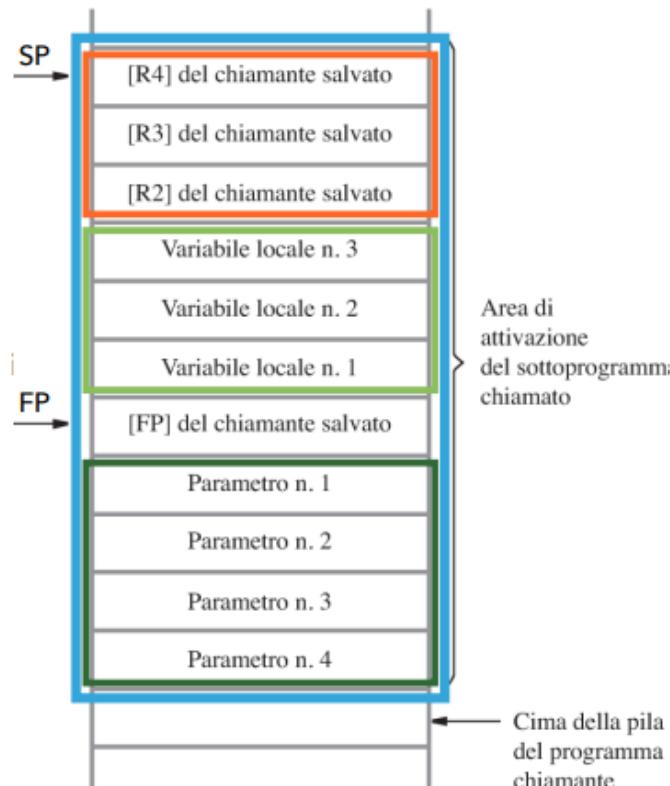
*Infatti quelle due istruzioni evidenziate possono essere considerate come **pseudo-istruzioni** visto che in assembly non può esistere un'istruzione del processore così complessa.*

AREA DI ATTIVAZIONE

L'area di attivazione è tutta l'area che comprende i dati di una sub routine. Essa si divide, partendo dal basso e andando verso l'alto, in varie parti.

E' una porzione di memoria che si espande al di sopra del programma chiamante. Sono presenti dei parametri da passare alla sub routine. Il valore subito sopra è **FP** e viene inserito quando la routine sta per cominciare. Sorrendo la memoria verso su, si ha ancor hanno le **variabili locali** e i valori dei **registri che sono stati salvati** che contenevano valori utili al programma chiamato.

Per definizione SP punterà sempre alla cima dello stack, quindi alla cima della pila.



44

E' molto pratico avere un **puntatore FP** all'area di attivazione che **punta al primo elemento dello stack**. Sotto FP ci sono i parametri. **Su FP** c'è il vecchio parametro di FP. Si può accedere ai parametri con FP al posto di dare un valore di spiazzamento a SP molto elevato. FP è in cima alla memoria prima di una sub routine, e per questo motivo si trova proprio sopra i parametri.

Quando la routine sta per iniziare, FP avrà un certo valore **relativo all'uso che ne sta facendo il programma chiamante**. FP viene salvato in cima allo stack. Il valore di FP viene salvato prima dell'inizio della routine. Prima di **return** bisogna ripristinare il valore precedente di FP. E' compito del programma chiamante ripristinare tale valore.

In caso di routine annidate, FP si aggiorna e serve per ripristinare una routine precedente alla fine della routine attuale. Quindi, in caso di più sub-routine si verificherà una concatenazione di FP.

Si nota che, in caso di più routine, le varie aree di attivazione si trovano **una sopra l'altra**.

Per salvare il valore di FP, bisogna prima decrementare SP di #4 e fare una **Store FP, (SP)**

ULTERIORI ISTRUZIONI MACCHINA

Le istruzioni viste sono state *Load, Store, Move, Clear, Add, Subtract, Branch, Call, Return*.

Altre istruzioni utili sono And, Or, Not

And R4, R2, R3 → calcola **L'AND** (prodotto) **bit a bit** degli operandi nei registri R2 e R3 e mette il risultato in R4

- ▶ Per azzerare i 3 byte a sinistra di R2, si usa And R2, R2, #0xFF
- ▶ FF è l'esadecimale per 11111111, che viene esteso a sinistra con 0 per 3 byte (per riempire una parola di memoria). Con l'AND bit a bit la parte di R2 che rimane invariata è solo il byte più a destra

`And R2, R2, #0xFF`

`Move R3, #0x5A` : carica "5A" in R3, 5A è il codice ASCII del carattere Z

`Branch_If_[R2]=[R3] TROVATOZ`

Scorrimento

LShiftL Ri, Ri, contatore → Logic Shift Left, **scorrimento logico a sinistra**.

LShiftR Ri, Ri, contatore → Logic Shift Right, **scorrimento logico a destra**.

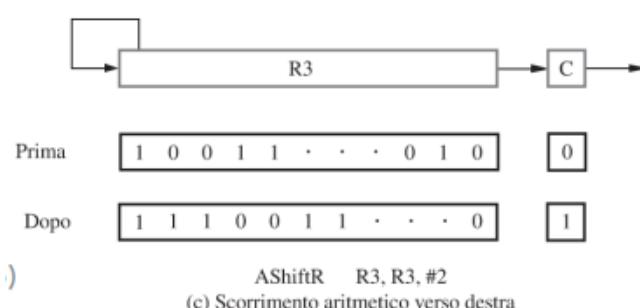
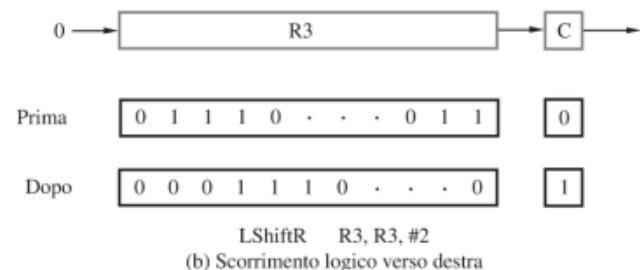
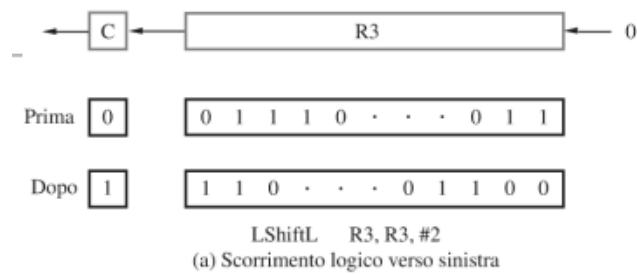
Lo scorrimento logico serve

per far scorrere i bit di un determinato registro verso sinistra o verso destra.

Scorrendo verso sinistra, si perdono i bit a sinistra e a sinistra si aggiungono bit 0.

Il bit che si fa uscire PER ULTIMO dopo un'operazione di scorrimento viene ricopiato in un **registro carry** e rappresenta un **bit di stato**.

Lo scorrimento verso destra è uguale, i bit meno significativi escono verso destra e l'ultimo viene ricopiato nel registro di stato carry. Man mano che si svuotano le posizioni di bit a sinistra, esse vengono riempite con degli zeri.



- Quando si scorre **verso destra** si dice che il numero viene **dimezzato**.
- Quando si scorre **verso sinistra** si dice che il numero viene **raddoppiato**.

Un utilizzo del carry può essere quello di visualizzare se un numero è pari o dispari.

Si possono utilizzare più parti di uno stesso byte per codificare elementi diversi, ognuno da 4 bit.

Quindi si possono impaccare cifre decimali.

Rotazione

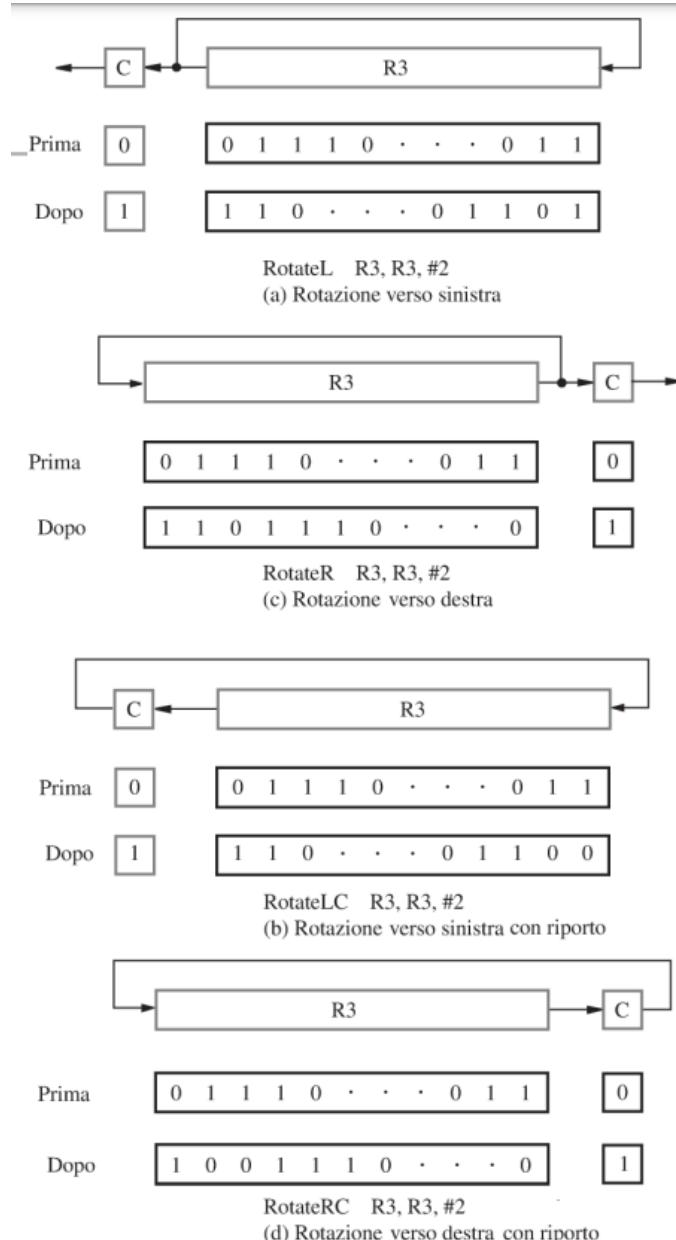
`RotateL R3, R3, #2` bit escono da sinistra e rientrano a destra;

`RotateR R3, R3, #2` bit escono da destra e rientrano a sinistra;

In entrambi i casi, l'ultimo bit a uscire dalla sequenza di bit viene salvato nel registro carry, o **registro di riporto**.

`RotateLC R3, R3, #2` sposta il bit sul carry e DOPO fa ruotare i bit verso sinistra.

`RotateRC R3, R3, #2` sposta il bit sul carry e DOPO fa ruotare i bit verso destra.



Moltiplicazione e divisione

Per moltiplicare due fattori con segno, quindi rappresentati in complemento a due, si userà:

```
Multiply Rk, Ri, Rj  
con Rk → destinazione e gli altri due registri sono i registri sorgente.
```

In una moltiplicazione è molto probabile che il numero di bit n necessario per il risultato possa essere poco rispetto al n di bit dei fattori. Quindi, per questo motivo, il risultato avrà un numero di bit **raddoppiato** rispetto ai bit di partenza.

Esempio: $15 \times 15 = 225 \rightarrow 1111 \times 1111 = 1110\ 0001$ (*il risultato ha un numero di bit raddoppiato*)

Può succedere che il risultato del prodotto non stia tutto nel registro risultato Rk, quindi a volte si mettono nel registro solo i bit meno significativi del prodotto.

In alcuni casi è possibile **dividere il risultato del prodotto in una coppia di registri**, in Rk+1 andrà scritta la metà più significativa.

Nel caso della divisione si utilizza:

```
Divide Rk, Ri, Rj
```

Il resto della divisione va nel registro Rk+1 oppure non viene messo da nessuna parte e viene perso.

VALORI IMMEDIATI A 32 BIT

Il modo di indirizzamento **immediato e assoluto** di un processore in stile RISC **limitano a 16 bit** lo spazio disponibile per l'operando o il suo indirizzo, nel formato dell'istruzione che occupa complessivamente 32 bit.

Il processore RISC sta attenta a codificare le istruzioni in 4 byte, cioè in **una parola di memoria**, e questa rappresenta anche la dimensione dei registri del processore RISC.

Con un OR tra un registro e un valore immediato si aggiunge il #valore nel registro operando Rsrc se quest'ultimo ha tutti i suoi 32 bit a 0. I 16 bit del #valore vanno sostituiti agli zeri **meno significativi** del Rsrc

```
Or Rdst, Rsrc, #Valore
```

OrHigh fa esattamente il contrario di Or, quindi mette il #Valore immediato nei bit **più significativi** piuttosto che nei meno significativi.

```
OrHigh Rdst, Rsrc, #Valore
```

Quindi il valore di 32 bit di 0x20004FF0 si può inserire in un registro mediante due istruzioni da 16 bit a loro volta separatamente.

Basandoci sugli Or precedenti si ha:

```
OrHigh R2, R0, #0x2000 //mette il #valore nella parte più significativa del registro R0  
//che contiene 0 e il risultato va nel registro R2.  
Or R2, R2, #0xFF0 // Mette il #valore nella parte meno significativa del registro R2  
//che contiene 0
```

INSIEMI DI ISTRUZIONI CISC

Le architetture **RISC** prendono dei valori, fanno delle operazioni con i registri e allocano in memoria i risultati ottenuti per utilizzare i registri in altri modi successivamente. Questa tecnologia si chiama, appunto, **Load/Store**.

Le architetture **CISC** possono avere istruzioni che occupano **PIU' DI UNA parola di memoria**. Quindi tale informazione dipende dall'istruzione stessa.

In CISC scriviamo **meno istruzioni** rispetto a RISC.

Per le istruzioni CISC si usano operandi che si **trovano direttamente in memoria**.

Add B, A → Legge valore di A. → Legge il valore di B → Mette il risultato in $B = B - [A] + [B]$

Nell'istruzione Add si possono specificare solo 2 operandi, di conseguenza se volessi fare $C=A+B$:

```
Move C, B  
Add C, A
```

Nell'istruzione Move si possono specificare sia locazioni di memoria e sia registri. Se dovessi usare registri per il caso $C=A+B$, si potrebbe scrivere:

```
Move Ri, A  
Add Ri, B  
Move C, Ri
```

INDIRIZZAMENTI ULTERIORI

Oltre ai modi di indirizzamento dei processori RISC si aggiungono in CISC ulteriori modi: **autoincremento** e **autodecremento**.

Per **impilare un nuovo elemento** si può usare un'istruzione di **push**:

```
Move -(SP), ELEMENTO  
/* decrementa prima SP di 4byte, poi inserisce la variabile  
ELEMENTO nella nuova posizione di SP */
```

Per **togliere dalla pila** si usa un'istruzione di **pop**:

```
Move ELEMENTO, (SP)+ /* trasferisce nella variabile ELEMENTO  
il valore puntato da SP, poi incrementa di 4byte SP e quindi  
abbassa la pila di una parola di memoria
```

Indirizzamento con modo relativo a PC

Il modo con indice e spiazzamento puo' essere applicato a **PC** ottenendo il **modo relativo** al contatore di programma PC.

X (PC) → indica una parola di memoria traslata di X byte rispetto all'intirizzo contenuto in PC.
Allora si ha che:

$$EA = X + [PC] \rightarrow EA \text{ è l'indirizzo effettivo} \rightarrow X \text{ è lo spiazzamento}$$

BIT DI ESITO O CONDIZIONE

Il registro di stato (**status register**) contiene bit di esito o di condizione.

I **bit di esito** più comuni sono:

- N(**negativo**), vale 1 se il risultato è negativo, altrimenti vale 0;
- Z(**zero**), vale 1 se il risultato è nullo, altrimenti vale 0;
- V(**trabocco**), vale 1 se vi è trabocco in complemento a due, altrimenti vale 0; (**overflow**)
- C(**riporto**), vale 1 se vi è trabocco in binario naturale, altrimenti vale 0; (**carry**)

```
Subtract R2, #1
Branch R2 > 0 CICLO //salta a CICLO se la precedente operazione è maggiore di zero
```

STILI RISC E CISC

Caratteristiche dello stile RISC:

- più lunghi di scrittura, quindi programmi più lunghi;
- istruzioni in una parola di memoria;
- meno istruzioni;
- esecuzione più rapida di una singola istruzione;
- le operazioni aritmetiche e logiche si applicano solo tramite registri

Caratteristiche dello stile CISC:

- corti di scrittura, quindi programmi più corti;
- istruzioni in più parole di memoria;
- più complicati da realizzare;
- modi di indirizzamento complessi;

CODIFICA DI ISTRUZIONI

In RISC le istruzioni occupano una parola di memoria.

In tutti i tipi di formati che sono elencati sotto, tutti i registri sono codificati sempre nello stesso range di bit.

Formato con operando in registri

Per codificare ciascuno dei registri in un processore a 32 bit, si hanno bisogno di 5 bit per registro perché $2^5 = 32\text{bit}$. Con un'istruzione codificata in 32 bit, si hanno 17 bit per codificare il codice operativo che indica l'operazione.

Per esempio: `Add Rdst, Rsrc1, Rsrc2`

- Rsrc1 è codificato SEMPRE nei bit [31,27];
- Rsrc2 è codificato SEMPRE nei bit [26,22];
- Rdst è codificato sempre nei bit [21,17];
- Il codice operativo è codificato sempre nei bit [16,0].

Quindi sono usati 4 bit per un registro!

Formato con operando immediato

`Add Rdst, Rsrc1, #valore` si hanno 10 bit per specificare i due registri, dei rimanenti 22 bit, si usano 16 bit per l'operando immediato, e i 6 bit per il codice operativo, e 5 bit per ciascuno dei registri.

Il salto condizionale `BGT R2, R0, CICLO` viene codificato secondo questo stesso formato.

- Rsrc1 è codificato SEMPRE nei bit [31,27];
- Rdst è codificato sempre nei bit [26,22];
- L'operando immediato è codificato SEMPRE nei bit [21,6]
- Il codice operativo è codificato sempre nei bit [5,0].

Formato per chiamata

Salvi a sottoprogrammi. Con 6 bit viene codificato il codice operativo e tutti i restanti bit vengono usati per codificare l'indirizzo del sottoprogramma.

- Il valore immmediato è codificato SEMPRE nei bit [31,6]
- Il codice operativo è codificato sempre nei bit [5,0].

31	27	26	22	21	17	16	0
----	----	----	----	----	----	----	---

Rsrc1	Rsrc2	Rdst	Codice operativo
-------	-------	------	------------------

(a) Formato con operandi in registri

31	27	26	22	21	6	5	0
----	----	----	----	----	---	---	---

Rsrc	Rdst	Operando immediato	Codice operativo
------	------	--------------------	------------------

(b) Formato con operando immediato

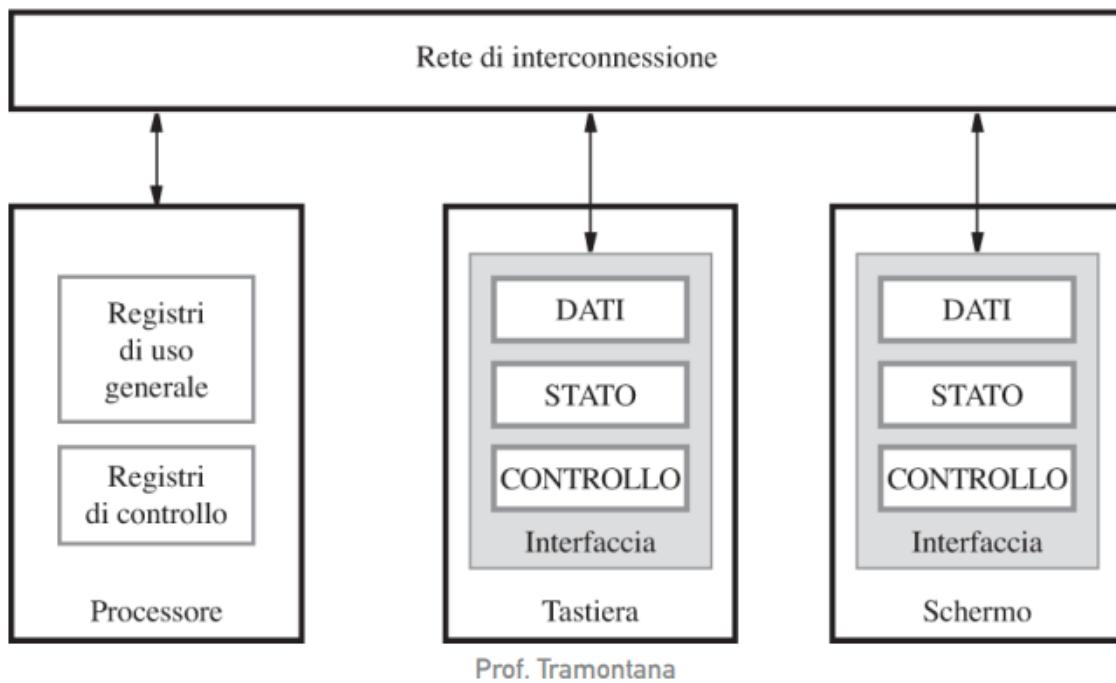
31	6	5	0
----	---	---	---

Valore immediato	Codice operativo
------------------	------------------

(c) Formato per chiamata

3. ACCESSO A DISPOSITIVI I/O

Il processore è collegato ad un **sistema di comunicazione** e a questo sistema sono connesse le periferiche, la memoria, il processore ecc..



Per l'operazione di LOAD il processore manda un indirizzo che viaggia nella rete di interconnessione e questo indirizzo riesce a individuare una determinata locazione di memoria. La memoria attiva delle celle di memoria e fa viaggiare il dato richiesto dalla memoria verso il processore sulla rete di interconnessione.

Alcuni indirizzi sono assegnati a delle periferiche. Questo allargamento degli indirizzi è detto **unificazione degli spazi di indirizzamento di memoria e di I/O. (MEMORY MAPPED I/O)**. Quindi determinati indirizzi del processore sono destinati per pilotare le periferiche. Load e Store possono essere utilizzati per dialogare con le periferiche.

Si usa questo metodo per scrivere a schermo determinate cose o prendere in input i caratteri da tastiera.

INTERFACCE DEI DISPOSITIVI I/O

Nelle periferiche esistono le interfacce del dispositivo. Dentro ogni periferica vi è una propria **interfaccia**. Questa interfaccia contiene 3 **registri**: **Dati**, **Stato** e **Controllo**.

Quando il processore comunica con le periferiche mediante Load, esso potrà leggere i registri presenti nelle periferiche.

Le periferiche forniscono i dati in maniera molto più lenta rispetto al processore.

- La tastiera è utilizzata dall'uomo quindi è abbastanza lenta perché **dipende dalla velocità di digitazione**.

Quando si preme un tasto, esso corrisponde ad un certo codice che rappresenta il tasto.

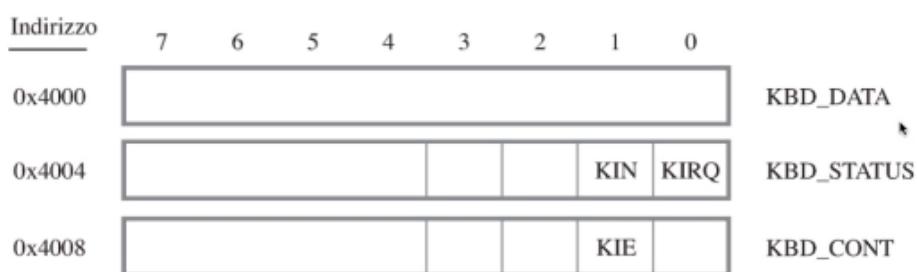
All'interno della tastiera vi è un piccolissimo circuito che fa corrispondere le lettere al proprio codice. Questo codice viene scritto nel registro dati all'interno dell'interfaccia della tastiera. Questo registro che contiene il dato fa da **buffer**, cioè da **registro temporaneo**. Inoltre, dopo la pressione del tasto, un bit di stato viene cambiato per indicare che il dato è pronto e il processore può leggerlo.

- Il video può segnalare che è pronto a ricevere un altro dato impostando il suo **bit di stato** e il processore può scrivere un altro sul display.

LETTURA DI DATI DALLA TASTIERA

La tastiera ha 3 registri precedentemente nominati, che sono etichettati con:

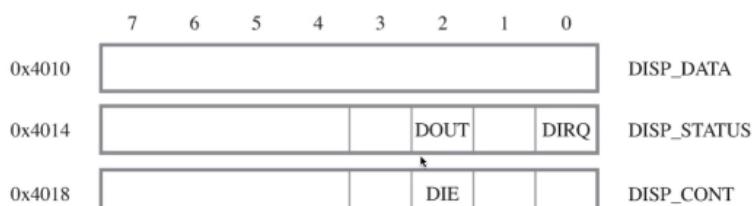
`KBD_DATA`, `KBD_STATUS`, `KBD_CONT`.



KIN Keyboard INput. (sta nella posizione bit 1, che vale 2 in decimale) Quando esso è a 1, vuol dire che c'è un nuovo dato da leggere in `KBD_DATA`. Dopo il processo di Load del processore da tastiera, KIN viene messo a 0.

SCRITTURA CARATTERI SUL VIDEO

Il processore manda 1 byte alla volta scrivendo sul registro `DISP_DATA`. `DOUT` (sta nella posizione di bit 2 e in decimale vale 4) indica se il video è pronto a ricevere un nuovo dato oppure no, quindi esso vale 1 o 0 rispettivamente.



(b) Interfaccia dello schermo

PROGRAMMA LETTURA DA TASTIERA E SCRITTURA A VIDEO (RISC)

`LoadByte` (leggi un singolo byte)

`MoveByte R3, #CR` Carica in R3 il codice ASCII per il ritorno carrello.

Ritorno Carrello è il tasto INVIO.

Move	R2, #LOC	Inizializza il registro puntatore R2 per puntare all'indirizzo della prima locazione nella memoria principale dove immagazzinare i caratteri	
MoveByte	R3, #CR	Carica in R3 il codice ASCII per il Ritorno Carrello	
LEGGI:	LoadByte And Branch_if_[R4]=0	R4, KBD_STATUS R4, R4, #2 LEGGI	Attendi l'immissione di un carattere Controlla la condizione di stato KIN
	LoadByte	R5, KBD_DATA	Leggi il carattere da KBD_DATA (ciò azzerà KIN)
	StoreByte	R5, (R2)	Scrivi il carattere nella memoria principale e incrementa il puntatore alla memoria principale
	Add	R2, R2, #1	
ECO:	LoadByte And Branch_if_[R4]=0	R4, DISP_STATUS R4, R4, #4 ECO	Attendi che lo schermo sia pronto Controlla la condizione di stato DOUT
	StoreByte	R5, DISP_DATA	Trasferisci il carattere appena letto al registro buffer dello schermo (ciò azzerà DOUT)
	Branch_if_[R5]≠[R3]	LEGGI	Controlla se il carattere appena letto sia il Ritorno carrello. Se non lo è, reitera la lettura di caratteri

Lettura di un carattere dalla tastiera

LoadByte indica che l'operando è un byte
And controlla il bit di stato

Visualizzazione di un carattere sullo schermo

8

Esempio in stile CISC

Move	R2, #BLOCCO	Inizializza il registro R2 per puntare all'indirizzo della prima locazione nella memoria principale dove immagazzinare i caratteri	
LEGGI	TestBit Branch=0	KBD_STATUS, #1 LEGGI	Monitorando la condizione di stato KIN, attendi l'immissione di un carattere nel registro di I/O KBD_DATA
	MoveByte	(R2), KBD_DATA	Scrivi nel byte di memoria puntato da R2 il carattere contenuto nel registro di I/O KBD_DATA (ciò azzerà KIN)
ECO	TestBit Branch=0	DISP_STATUS, #2 ECO	Attendi che lo schermo sia pronto monitorandone la condizione di stato DOUT
	MoveByte	DISP_DATA, (R2)	Scrivi il carattere puntato da R2 nel registro di I/O DISP_DATA (ciò azzerà DOUT)
	CompareByte Branch≠0	(R2)+, #CR LEGGI	Verifica se il carattere appena letto da tastiera sia il Ritorno Carrello: se non lo è, reitera la lettura di caratteri; in ogni caso incrementa il registro puntatore R2

INTERRUZIONI

Le interruzioni servono per comunicare con le periferiche per evitare di lasciare il processore in fase di attesa di lettura/scrittura tra periferiche.

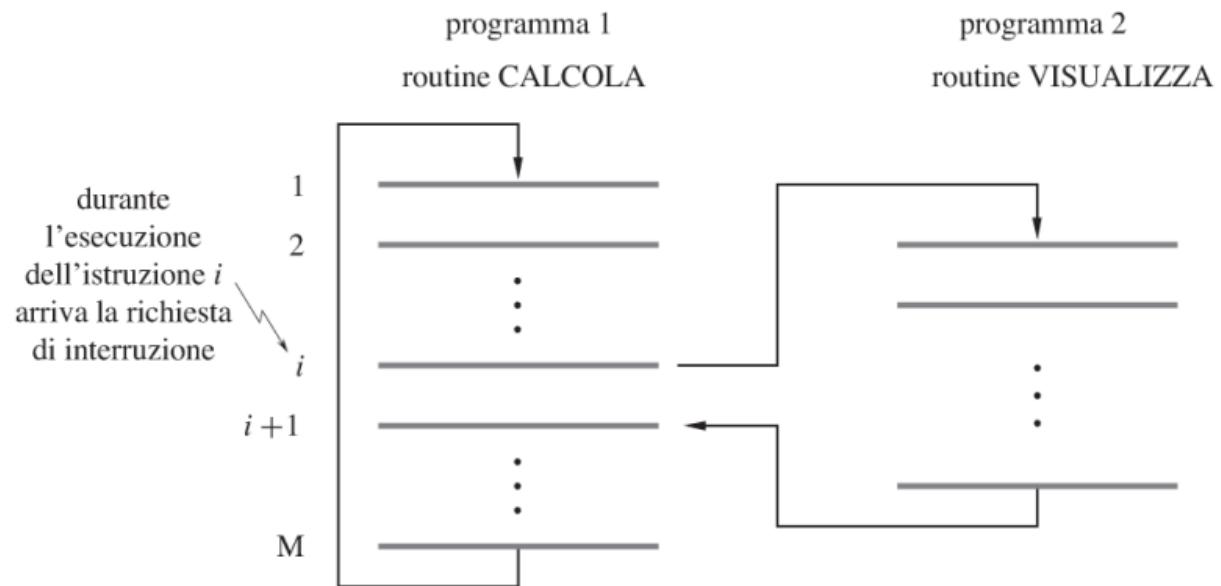
Un'interruzione è un segnale di controllo che permette alle periferiche di comunicare al processore che sono pronte a fare qualcosa. Questa comunicazione avviene tramite la linea **INT_REQ** e quando questa linea dà il segnale 1, il processore se n'è accorto e legge/scrive i dati

tra periferiche. Si chiama **Interruzione** perchè il processore interrompe le operazioni che sta facendo per dedicarsi alle periferiche.

Un sottoprogramma attivato alla richiesta di interruzione è detto **routine di servizio di interruzione**.

Return-from-interrupt è l'istruzione che permette di tornare al programma principale del processore dopo aver eseguito la routine di servizio di interruzione.

Il processore informa la periferica dell'avvenuta ricezione dell'interrupt tramite la linea **INT_ACK** nel bus di controllo.



SERVIZIO DELLE INTERRUZIONI (ISR)

Tempo di latenza dell'interruzione: tempo che intercorre tra quando il processore si accorge che esiste un interrupt e tra l'effettivo lasso di tempo per l'esecuzione della prima istruzione dell'interrupt. Serve duplicare i registri affinchè il salvataggio dei registri prima di un'interruzione sia breve e ridurre il tempo di latenza dell'interruzione.

Il registro **PS (Program Status)**, che contiene le informazioni sullo **stato corrente** delle operazioni del processore, **dove essere salvato**.

Il registro **PC** **dove essere anche salvato** per poter **tornare al programma principale**.

Principalmente si cerca di salvare tutti i registri prima della routine di servizio. Per salvare tutti i registri si potrebbe perdere molto tempo, quindi si cerca di salvare il minimo, cioè **PS** e **PC**. In automatico la routine salverà i registri che essa stessa andrà a modificare e quindi non salva i registri non utilizzati!

Per salvare PS si usano dei registri ausiliari che serve come copia del registro PS per evitare di salvarlo nella pila in memoria.

E' utile avere **un banco di registri ausiliario** per salvare più registri.

Serve un **servizio** di interruzione **più rapido possibile** per evitare rallentamenti. In sistemi real-time, cioè che hanno un tempo di risposta minimo, serve che la copia dei registri impieghi un tempo minimo.

CONTROLLO DELLE INTERRUZIONI

Alcune sequenze di esecuzione **non devono essere interrotte**. Per questo si parla di **abilitazione di interruzioni**.

Nel processore vi è un bit **IE** (**Interrupt Enable**) nel registro PS(registro di stato). Quando il bit **IE** è 1, le interruzioni sono abilitate.

All'interno del registro di controllo della periferica vi è un bit IE (nella tastiera= **KIE** \ nel display= **DIE**) che permette di segnalare richieste di interruzioni se è 1, altrimenti no.

Sequenza di eventi relativi a una richiesta di interruzione:

1. Sulla tastiera viene premuto un tasto. **KIN** viene messo a 1. Se **KIE** è 1 mette **INT_REQ** va alto e **IRQ** a 1. IRQ è un bit di stato che indica se è stata quella determinata periferica a richiedere un'interrupt o no.
2. Il processore sospende il programma, **salva** **PC** e **PS**
3. Il processore modifica il suo bit **IE** e lo mette a **0** perchè ha capito che c'è un'interruzione e deve servire una periferica.
4. La routine di servizio di interruzione viene eseguita, il processore manda un segnale sulla linea **INT_ACK** che è letto della periferica e capisce che sta per essere servita. Reagisce **abbassando** la linea **INT_REQ**. Il processore può anche evitare di segnalare tramite **INT_ACK** perchè i registri della periferica riescono a capire da soli se il dato è stato letto dal processore.
5. **return-from-interrupt** ripristina PC e PS e riabilita le interruzioni, quindi mette il suo bit **IE** a **1** e rientra nel programma interrotto.

Dispositivi multipli

Per capire quale periferica che manda ISR il **processore controlla il bit IRQ di ogni periferica** e questo processo è detto **PROCESSO DI POLLING**. In base all'ordine di scansione dei bit IRQ delle varie periferiche, si decide anche un ordine di **priorità** di esecuzione di routine di servizio. Leggere la memoria sull'interfaccia della periferica di 100 periferiche può richiedere molto tempo.

Si parla allora di **INTERRUZIONI VETTORIZZATE**. Con questa tecnica, la periferica che mette la linea **INT_REQ** alto manda un **codice di pochi bit** che permette al processore di capire **quale periferica ha richiesto interrupt**. Questi bit rappresentano un indice che servono per accedere al vettore delle interruzioni. In memoria, sono state memorizzate **nella parte con indirizzi più piccoli** una serie di dati, chiamati **vettori di interruzioni** e il dato che fornisce la periferica farà da indice nella parte più bassa della memoria. L'indirizzo della routine di servizio avrà un suo indirizzo e starà nella parte più bassa della memoria.

Annidamento interruzioni

Piuttosto che abilitare/disabilitare le interruzioni nel processore col suo bit IE, si usa un **livello di priorità**. Si attribuisce ad ogni periferica un livello di priorità. Se durante un'interruzione, arriva una richiesta di interruzione, si valuta il suo livello di priorità: se ha una priorità più alta, essa viene servita, altrimenti no.

Il **livello di priorità** è codificato in bit appositi nel registro di stato PS.

Esiste anche un registro **IPENDING** che salva le **interruzioni non ancora eseguite**.

Richieste di interruzione simultanee

In caso di richieste simultanee di interruzione da parte delle periferiche, verrà valutato il livello di priorità secondo l'ordine di **scansione del bit IRQ** oppure usando le interruzioni vettorizzate e selezionando una sola periferica per permettere di mandare il **vettore di interruzione**.

Esistono dei **circuiti di arbitraggio** che sono dei circuiti appositi che pensano a **decidere le priorità**. Solo la periferica con priorità più alta avrà priorità su richieste di interruzione.

Controllo della richiesta

La disabilitazione della periferica può essere fatta dal processore. Il processore agisce sul registro di controllo della periferica modificando il suo bit **IE** a 0.

Quando KIN è 1 vuol dire che il dato della tastiera è pronto per essere letto. Quando KIN e KIE sono 1, la richiesta viene segnalata ponendo **KIRQ** a 1 e viene messa alta la linea **INT_REQ**.

Registro di controllo del processore

IPS è quel **registro ausiliario** che ricopia il registro PS quando vi è un'interruzione. All'interno di PS vi è il bit **IE** (Interrupt Enable).

Se sono permesse interruzioni annidate, bisogna **salvare in pila** il contenuto di **IPS** prima di eseguire ISR.

IENABLE indica quali sono le **periferiche abilitate a trasmettere interruzioni**. Ogni periferica ha un bit **IENABLE** assegnato. Se il bit è 1, la periferica corrispondente è abilitata a mandare interruzioni. Se è 0, non è abilitata a farlo.

Con questo registro si ha la possibilità di **abilitare/disabilitare** le periferiche che hanno il permesso di inviare interruzioni o meno.

IPENDING indica le richieste di interruzioni che sono arrivate ma che ancora **non sono state eseguite**. Ogni bit di questo registro corrisponde ad una periferica specifica assegnata.

E' possibile controllare da programma questi registri se si volesse disabilitare le interruzioni di qualche periferica. Si può accedere sia in lettura che in scrittura a PS,IPS,IPENDING e IENABLE.

Si usa **MoveControl R2, PS** → copia i bit contenuti in PS su R2 e si possono azzerare o mettere a 1 i bit per i singoli registri.

Programma principale (Main)

INIZIO:	Move R2, #LINEA	
	Store R2, PNTR	Inizializza il puntatore del buffer
	Clear R2	
	Store R2, EOL	Azzera l'indicatore di fine linea
	Move R2, #2	Abilita le interruzioni nell'interfaccia della tastiera
	StoreByte R2, KBD_CONT	
	MoveControl R2, IENABLE	
	Or R2, R2, #2	Abilita le interruzioni da tastiera nel registro di controllo del processore
	MoveControl IENABLE, R2	
	MoveControl R2, PS	
	Or R2, R2, #1	
	MoveControl PS, R2	Poni a 1 il bit di abilitazione delle interruzioni in PS
Prossima istruzione		

Gestore delle interruzioni

All'interno del calcolatore vi è pre installato un programma default che permette di verificare i segnali di interruzioni e saltare direttamente alla routine di servizio. Questo procedimento utilizza il registro **IPENDING**.

ILOC:	Subtract SP, SP, #12	Salva i registri
	Store LINK_reg, 8(SP)	
	Store R2, 4(SP)	
	Store R3, (SP)	
	MoveControl R2, IPENDING	Controlla il contenuto di IPENDING
	And R3, R2, #4	Controlla se lo schermo ha segnalato la richiesta
	Branch_if_[R3]=0 TESTKBD	Se no, controlla se lo ha fatto la tastiera
	Call DISR	Chiama la routine di servizio delle interruzioni da schermo (display ISR)
TESTKBD:	And R3, R2, #2	Controlla se la tastiera ha segnalato la richiesta
	Branch_if_[R3]=0 PROSSIMO	Se no, controlla il prossimo dispositivo
	Call KISR	Chiama la routine di servizio delle interruzioni da tastiera (keyboard ISR)
PROSSIMO:	...	Controlla le interruzioni da altri dispositivi di I/O
	Load R3, (SP)	Ripristina i registri
	Load R2, 4(SP)	
	Load LINK_reg, 8(SP)	
	Add SP, SP, #12	
	Return-from-interrupt	

CONCETTO DI ECCEZIONE

La **causa di un'interruzione** è detta **eccezione**. Ce ne sono di vari **tipi**:

- Eccezione di memoria per **ripristino da errore**: se nella memoria ci sono dei dati alterati, essa manda al processore una richiesta di interruzione;

- **Divisione per zero produce un risultato inesistente**, quindi si ha un tentativo a parti protette della memoria, e ciò comporta interruzioni;
- **Esecuzione passo passo(debug)** sono delle interruzioni del sistema operativo che servono per interagire con i programmi utente.

A seguito di un'eccezione il processore sospende il lavoro in corso ed esegue la routine di servizio. Se si tratta di errore, quindi non proviene da una periferica, il processore rinuncia ad eseguire il lavoro in corso.

5.STRUTTURA DI BASE DEL PROCESSORE

Il processore:

1. **Preleva** la parola di memoria puntata dal registro **PC** e la inserisce in **IR**. *In RTN $IR \leftarrow [[PC]]$*
[PC] è un indirizzo, [[PC]] è una cella di memoria
2. **Incrementa il valore di PC** di 4 byte (prossima parola di memoria, o prossima istruzione);
3. Esecuzione istruzione prelevata.

I primi due passi sono detti di **prelievo (fetch)**, e il terzo passo è di **esecuzione**.

L'esecuzione è effettuata tramite → LOAD,ADD,STORE ecc.. :

1. Lettura del contenuto di memoria e caricamento in un registro
2. Lettura di dati da un registro del processore
3. Esecuzione di un'istruzione aritmetica o logica e scrittura in un registro
4. Scrittura dei dati di un registro in una locazione di memoria

Componenti hardware di un processore

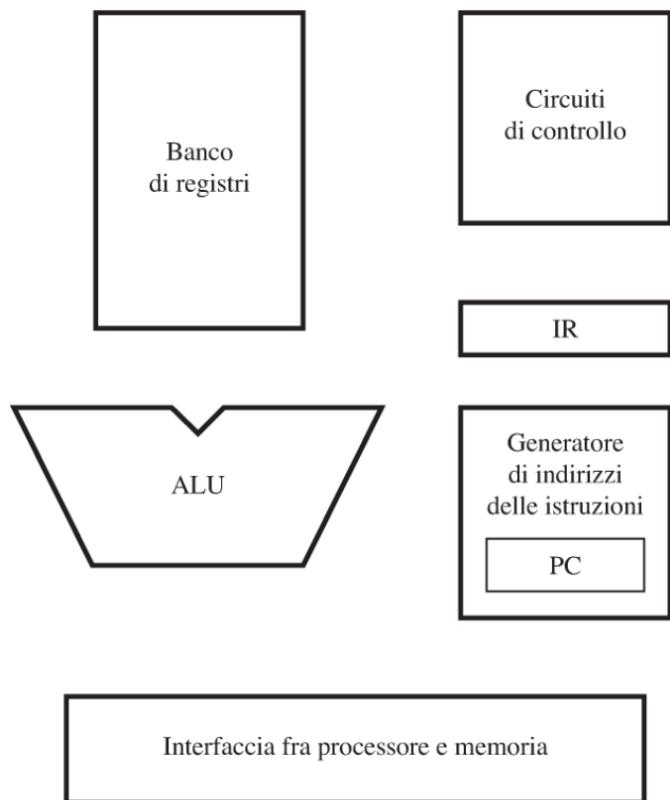
Il **BANCO DEI REGISTRI** contiene tutti i registri.

La **ALU** esegue tutte le operazioni aritmetiche e logiche. Ci deve essere un altro dispositivo che deve dare il segnale alla alu per dire quale operazione eseguire. E tale segnale viene generato dai **CIRCUITI DI CONTROLLO**.

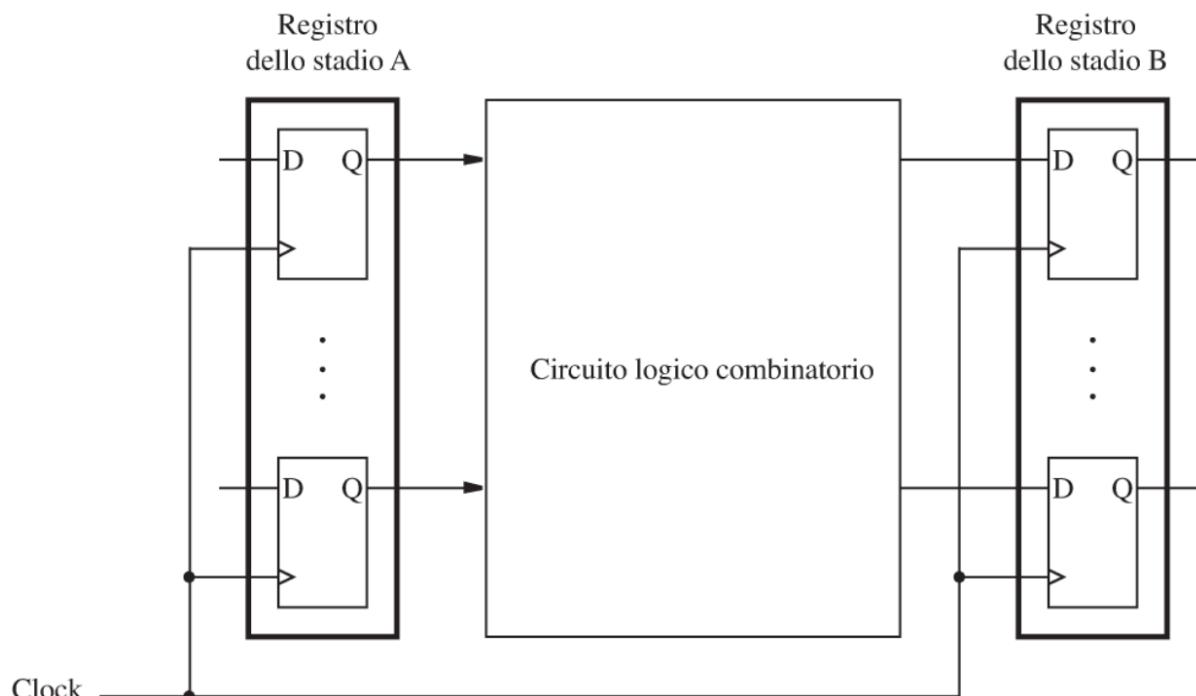
IR contiene l'istruzione da eseguire e sta lì per tutta la durata dell'esecuzione. Codice operativo e codifica degli operandi sono prelevati da IR per poter capire quale registro del banco dei registri leggere o scrivere. Il codice operativo deve essere mandato ai circuiti di controllo.

Il **GENERATORE DI INDIRIZZI DELLE ISTRUZIONI** incrementa PC opportunamente per puntare alla prossima istruzione o cambia il valore per fare una routine di servizio.

INTERFACCIA FRA PROCESSORE E MEMORIA permette di trasferire dati durante le operazioni di lettura e scrittura.



Hardware per l'elaborazione di dati



Ogni funzione logica che si vuole eseguire è possibile realizzarla come un **circuito logico combinatorio**.

Il periodo di **clock** deve essere **sufficientemente grande** affinchè il circuito combinatorio abbia il tempo necessario per produrre un risultato letto in input e scritto in output.

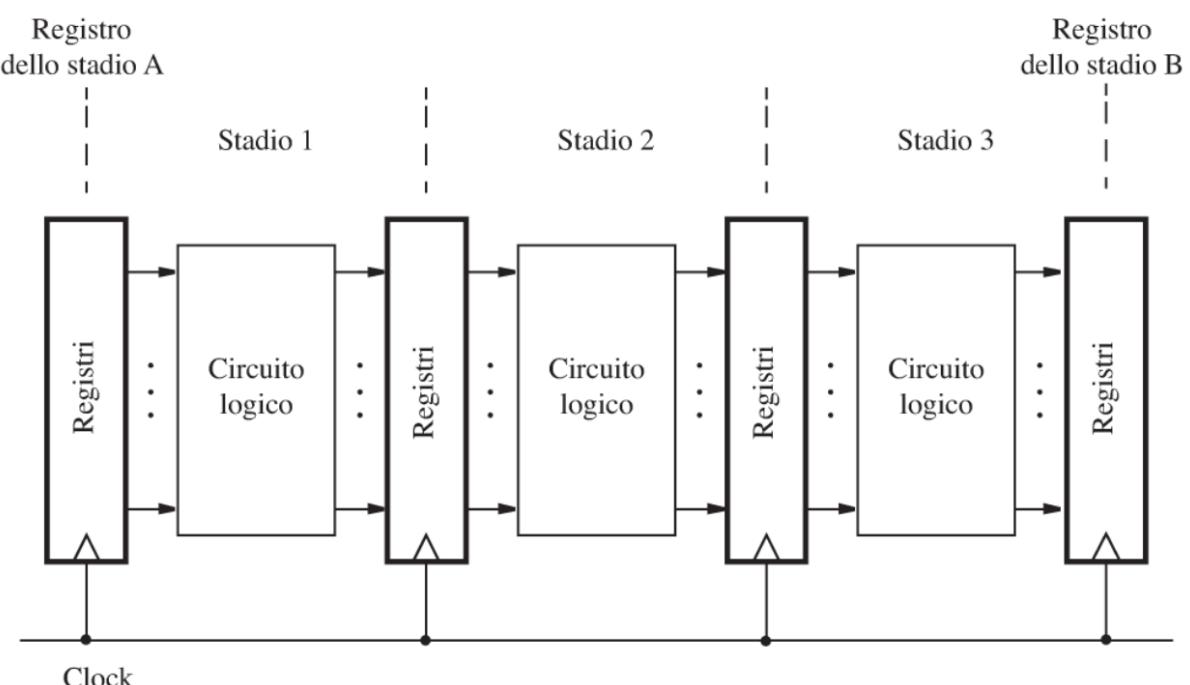
Con il primo **fronte di salita** del clock viene abilitata l'uscita input dei **registri a sinistra** e con il **secondo fronte di salita** di clock si memorizzano i dati nei **registri di destinazione a destra**.

Un periodo di clock elevato non permette un'esecuzione del circuito in modo rapido.

Si suddivide in più pezzi il circuito combinatorio perchè il tempo di propagazione di un circuito che ha meno porte in sequenza è più piccolo rispetto ad uno che di porte ne ha di più.

Si prevede allora la **suddivisione in più stadi** del circuito combinatorio.

Struttura hardware a più stadi



I registri vengono dati in input a uno stadio di elaborazione realizzato mediante un circuito combinatorio. Esso da risultati che verranno messi in un altro registro. Questo risultato è **output del primo stadio e input del secondo stadio**.

Questi registri intermedi sono detti registri **INTERSTADI**.

Il segnale di clock viene passato a tutti i registri interstadi per poter prelevare dati da registro(o scrivere). Il **periodo di clock è più basso** e quindi **l'esecuzione è più veloce**. Tutti gli stadi sono capaci di produrre un risultato in un periodo di clock.

Istruzione Load

L'hardware che vedremo ha 5 stadi. Per esempio:

Load R5, X(R7) (indirizzamento con indice e spiazzamento) può essere eseguita in 5 stadi diversi:

1. **Prelievo istruzione e incremento di PC;**

2. **Decodifica istruzione**(fatta dal circuito di controllo) e **lettura del registro sorgente** di R7
3. Calcola l'**indirizzo effettivo** X+[R7] (**elaborazione**)
4. **Lettura dell'operando** dalla memoria all'indirizzo effettivo (**memoria**)
5. **Scrittura** dell'operando nel registro R5 (**scrittura**)

Per ogni stadio c'è un circuito combinatorio apposito. Quando non si ha un registro con spiazzamento, nel passo 3 si fa sommare semplicemente zero.

Se si dà un valore immediato si ha una `Load, R3, #4` e nel passo 3 si ha X=4 e gli si somma 0.

Istruzioni Add

`Add R3, R4, R5` in 5 passi:

1. Prelievo istruzione e incremento PC
2. Decodifica istruzione e lettura R4 e R5
3. Calcolo della somma [R4]+[R5]
4. Nessuna azione (non si può interagire con la memoria)
5. Scrittura del risultato nel registro R3

In caso si specificasse un valore immediato nella ADD, nel passo 2 si leggerà un solo registro e il valore immediato si leggerà nell'istruzione stessa nel registro IR.

Istruzione Store

`Store R6, X(R8)` in 5 passi

1. Prelievo istruzione dalla memoria e incremento PC
2. Decodifica istruzione e lettura di **R6 E R8**
3. Calcola l'indirizzo effettivo X+[R8]
4. Immagazzinamento di R6 in memoria all'indirizzo effettivo
5. Nessuna azione

<code>Load R5, X(R7)</code>	<code>Add R3, R4, R5</code>	<code>Store R6, X(R8)</code>
<ol style="list-style-type: none"> 1. Prelievo istruz e increm PC 2. Decod istruz e lettura R7 3. Calcolo indirizzo X+[R7] 4. Lettura da memoria 5. Scrittura nel registro R5 	<ol style="list-style-type: none"> 1. Prelievo istruz e increm PC 2. Decod istruz e lettura R4 e R5 3. Calcolo somma [R4]+[R5] 4. Nessuna azione 5. Scrittura risultato in R3 	<ol style="list-style-type: none"> 1. Prelievo istruz e increm PC 2. Decod istruz e lettura R6 e R8 3. Calcolo indirizzo X+[R8] 4. Scrittura in memoria 5. Nessuna azione

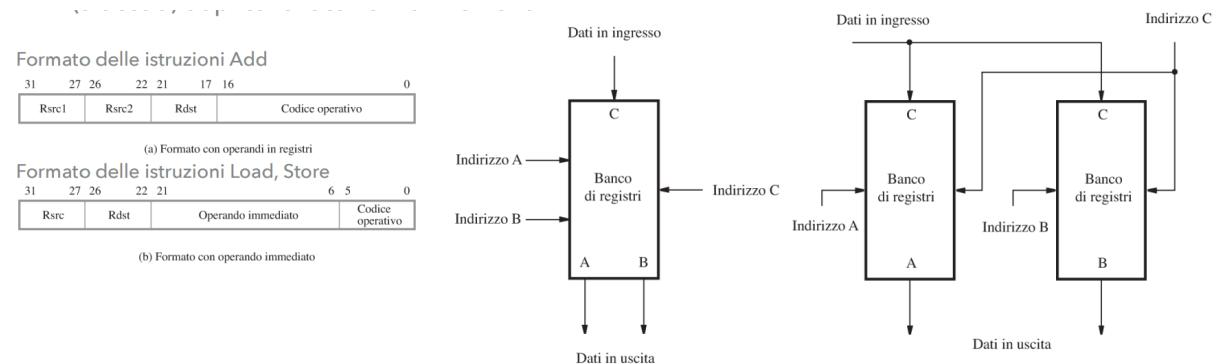
GENERALMENTE:

1. Preleva un'istruzione e incrementa il contatore del programma
2. Decodifica l'istruzione e leggi registri dal banco di registri
3. Esegui un'operazione dell'ALU
4. Legge o scrive i dati in memoria se l'istruzione coinvolge un operando in memoria
5. Scrivi il risultato nel registro di destinazione, se necessario

HARDWARE: BANCO DI REGISTRI

Il banco dei registri prevede che ci siano **2 ingressi** (*indirizzo A* e *indirizzo B*) che specificano indirizzo del registro che si vuole leggere perché nel passo 2, nei 5 stadi, si ha necessità di leggere, a volte, 2 registri. Il banco dei registri fornisce 2 dati (**dati in uscita**) che corrispondono ai contenuti dei registri indicati in ingresso.

Ci sono altre 2 porte in ingresso (**Dati in ingresso** e **Indirizzo C**) e servono per specificare la scrittura di dati quando si è al passo 5 e vengono usate come registro di destinazione.



Si vuole che, alla fine del passo 2, i valori di entrambi i registri uscenti da A e B siano **entrambi disponibili allo stesso momento**. Per realizzare ciò si prendono delle memorie di tipo **consueto** che hanno **un solo ingresso per l'indirizzo e una solo uscita**. Quindi avviene una **duplicazione della quantità di memoria** per memorizzare i registri.

In questo modo, quando si vuole prelevare un dato, si manda l'indirizzo A ad un banco di registri che ha un solo ingresso e l'indirizzo B all'altro banco di registri.

Ognuno di questi sono in grado di **emettere il dato** in uscita in **un passo di esecuzione**.

Per quanto riguarda la **SCRITTURA NEL BANCO DEI REGISTRI**, l'indirizzo di store viene mandato a tutti i banchi registri che hanno questa caratteristica di singola porta di ingresso per la lettura. Quindi esiste una **duplicazione del dato** su questi due banchi.

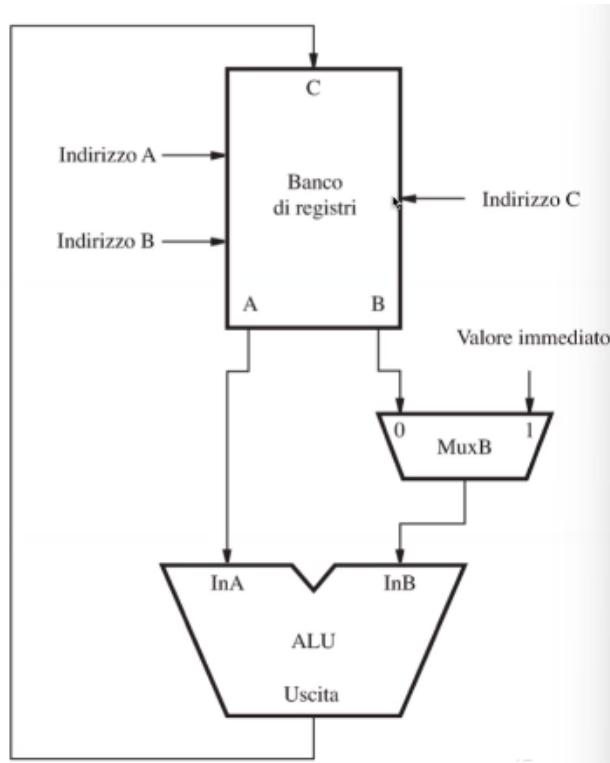
La duplicazione del dato serve perché **non si sa** se un dato è stato memorizzato nel banco A o nel banco B, quindi è necessario memorizzare il dato in entrambi i banchi così da renderlo sempre disponibile.

Si duplica la quantità di memoria perché i registri hanno solitamente una sola porta per

specificare il dato da leggere e una sola porta in uscita proveniente da quel dato. (cioè ad un ingresso corrisponde una sola uscita, figura a sinistra)

Quindi ci sono queste due possibilità per ottenere il dato in uscita in un unico passaggio:
duplicazione dei banchi dei registri oppure **duplicazione delle linee in ingresso** in un solo banco di registri.

UNITA' ARITMETICA LOGICA → ALU



Il dato uscente dal banco dei registri di A, va in **InA**, ovvero nel primo ingresso dell'ALU.

La ALU ha **sempre 2 ingressi**.

L'uscita B del banco di registri si collega a un **multiplexer(multiplatore)** che è un componente che ha più ingressi e una sola uscita. Ha un ingresso di controllo che permette di selezionare **uno dei vari input** e farne uscire solo uno.

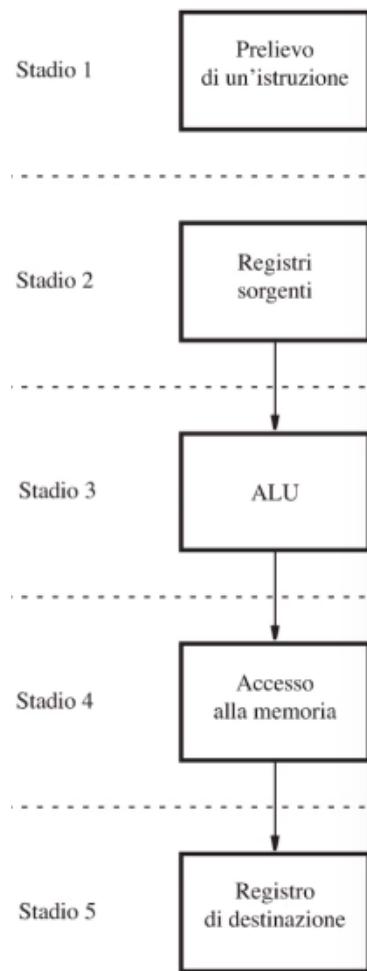
L'uscita del multiplexer è collegata all'input numero 2 dell'ALU **InB**.

Se c'è **un valore immediato**, nel multiplatore viene selezionato "1" e in output ci sarà un valore immediato.

Dopo che ALU fa l'operazione richiesta, manda l'output nella scrittura in un registro. La codifica del registro di destinazione va all'indirizzo C e il dato viene memorizzato.

Tra ingresso 1 di MuxB e l'operando immediato c'è un circuito che mette a 0 tutti i 16 bit restanti nella codifica del valore immediato altrimenti alla ALU arriverebbero solo 16 bit che sono troppo pochi e quindi ne servono 32.

STRUTTURA A CINQUE STADI

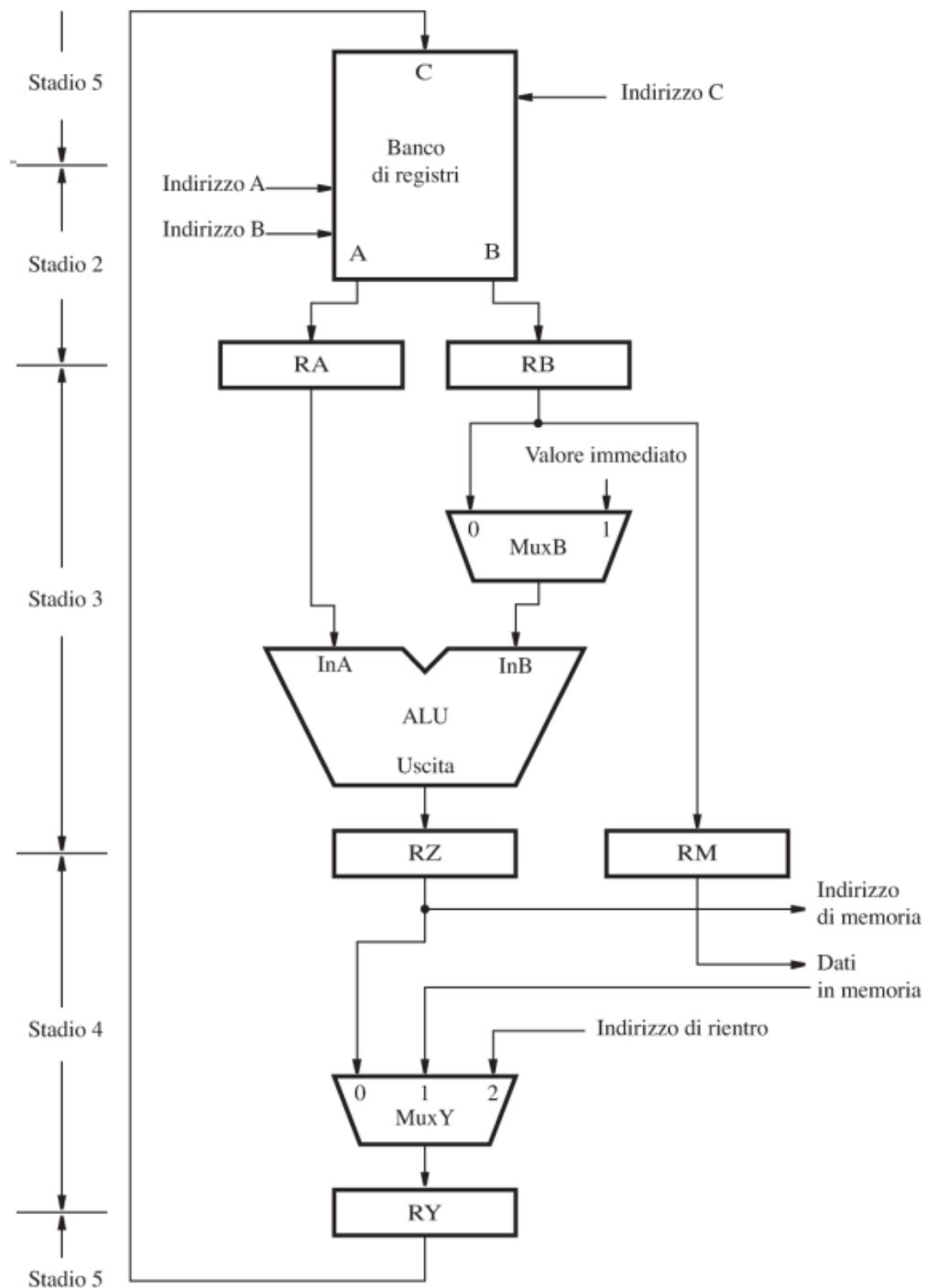


Ogni stadio viene eseguito in un **tempo pari ad un periodo di clock**. Lo stadio 1 non è collegato agli altri stadi perché esso è uno stadio eseguito sempre e quindi **non è sequenziale**. Tutto il resto è sequenziale e a cascata.

Tutto ciò viene eseguito mediante il **PERCORSO DATI** (datapath).

PERCORSO DATI (datapath)

Lo stadio 1 è omesso perché è spiegato a parte.



Ci sono dei **registri interstadio RA,RB,RZ,RM,RY** che memorizzano un risultato di uno stadio precedente. Essi sono anche ingressi per dei prossimi stadi.

Nello stadio 4, **RZ** va **mandato in memoria** e sarà un indirizzo. Sarà collegato alla parte della memoria che prende in input gli indirizzi.

RZ è collegato ad un altro multiplexer che **si attiva se il dato di RZ non ha necessità di interracciarsi con la memoria**, come nel caso di una **ADD**.

RM è un dato che viene passato alla memoria, verso gli ingressi della memoria che sono fatte apposta per far scrivere in memoria dei **DATI**, quindi esso entra in gioco quando si tratta di una

store per immagazzinare dati in memoria.

Nel caso della STORE, l'indirizzo di memoria che sarà usato per salvare un dato in memoria, viene passato a **RA** e, con la ALU, si avrà l'indirizzo effettivo. Quest'ultimo verrà scritto su **RZ** per poi essere mandato in memoria.

Alla fine del 4 stadio si memorizza qualcosa su **RY**. Tale dato viene mandato nel banco dei registri per memorizzare il contenuto ed entra nell'**Ingresso C**.

Da notare che il **banco dei registri** avrà 2 cicli di clock provenienti dallo **stadio 2** e dallo **stadio 5**.

In caso di una **load**, il dato proveniente dalla memoria viaggia sulla linea collegata ad 1 su **MuxY**. Nello stadio 4 l'ingresso 1 del muxY sarà selezionato (mediante un segnale di controllo) per scrivere il valore letto dalla memoria su **RY**.

Questo procedimento avviene per la maggior parte delle istruzioni. **LDR, STR, ADD, SUB, CMP**.

PRELIEVO DELLE ISTRUZIONI (stadio 1)

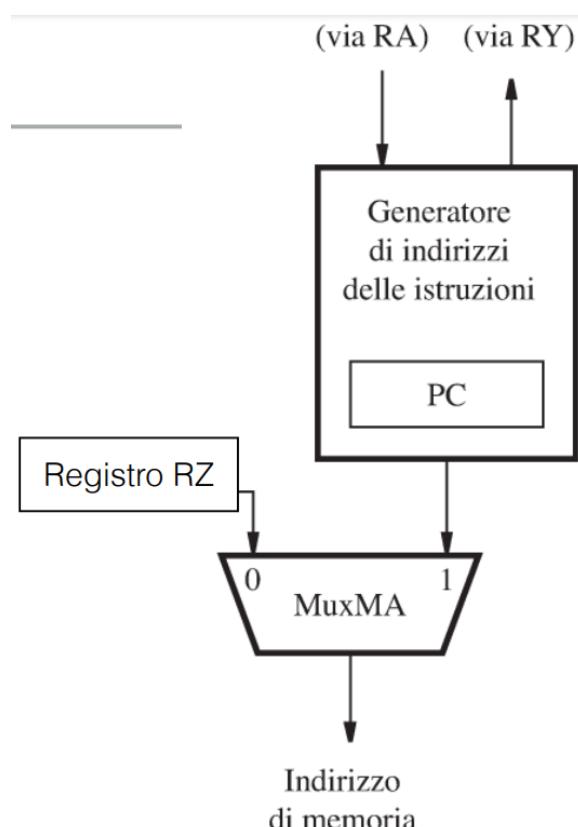
Lo **stadio 1 preleva le istruzioni**. In questo stadio si deve leggere la **successiva istruzione** dello stadio successivo. Questo valore si legge in **PC**.

Il **generatore degli indirizzi delle istruzioni** riesce ad **incrementare** il registro PC di **4 byte** (la grandezza di una parola di memoria) se si tratta della prossima parola di memoria da puntare oppure di **TOT** se si tratta di una **CALL** oppure qualsiasi altra istruzione di salto o interruzioni.

Il valore di PC viene mandato a un multiplexer **MuxMA** che sarà collegato all'ingresso degli indirizzi della memoria (accompagnato dall'eventuale segnale di controllo di lettura dalla memoria). La memoria darà l'istruzione che verrà memorizzata in **IR**.

MuxMA esiste perchè gli **indirizzi** da mandare alla memoria relativi alle **istruzioni** da prelevare sono diversi dagli **indirizzi** contenuti in una **load** o una **store** (entrambi calcolati in RZ). Per questo motivo il multiplexer decide quale tipo di indirizzo bisogna selezionare tra l'indirizzo presente in RZ e l'indirizzo presente in PC.

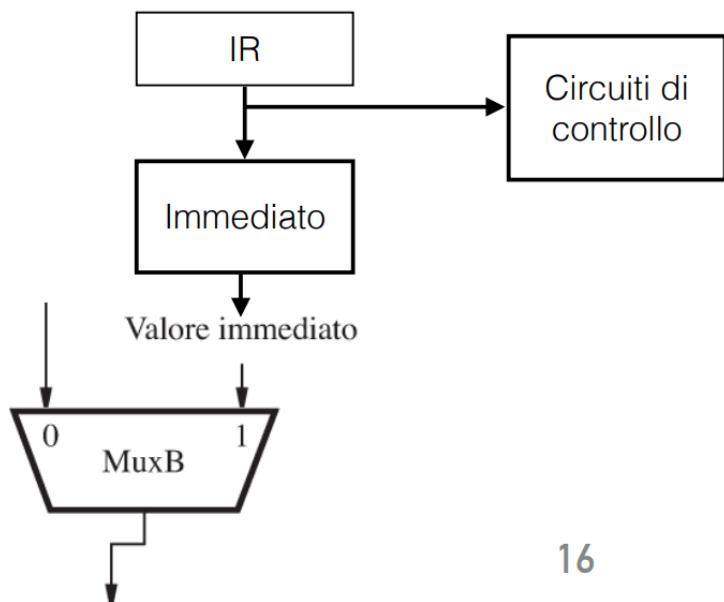
- Allo **stadio 1** viene passato sempre l'indirizzo presente in PC attraverso **l'ingresso 1 del multiplexer MuxMA**.
- Allo **stadio 4**, se si tratta di una load o una store, si seleziona **l'ingresso 0 nel multiplexer MuxMA** e quindi si prende in



considerazione l'indirizzo effettivo calcolato in RZ.

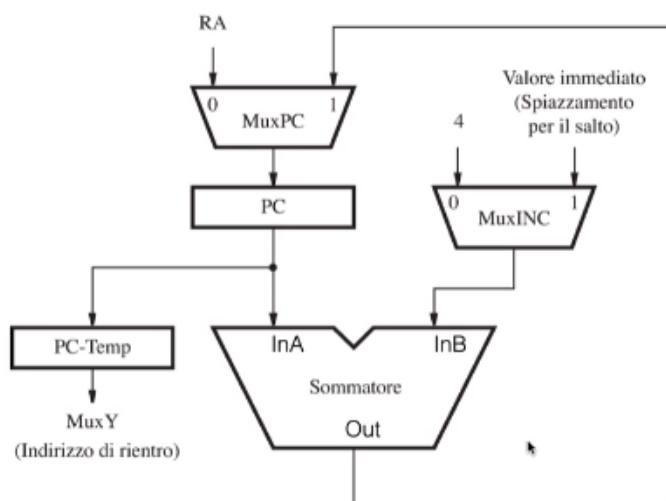
Il blocco **IMMEDIATO** serve per **estendere** i 16 bit prelevati dall'istruzione in IR affinché siano di **32 bit**. Esso mette a 0 tutti i bit più significativi perché nei bit meno significativi è codificato il valore immediato.

All'interno di **MuxB** arriverà un dato a **32 bit** perché esso **non può operare con una quantità di bit diversi fra loro**. Ciò serve per **evitare** di avere delle **reti spurie**.



I bit di IR sono forniti dall'operazione di prelievo che si è conclusa con il **primo ciclo di clock**. Si passa ad un **circuito di controllo** un'istruzione che decodifica il codice operativo dell'istruzione e vengono mandati tutti i segnali necessari nel percorso dati che servono per pilotare la memoria (in caso di LOAD). Il circuito di controllo comunica il codice operativo per far capire alla memoria l'operazione che è richiesta.

GENERATORE DI INDIRIZZI DELLE ISTRUZIONI (PC)



All'inizio del programma, con la direttiva `ORIGIN`, si ha un certo indirizzo su PC, in questo caso **l'inizio del programma**.

Questo indirizzo viene passato a `PC-Temp` e all'ALU.

`MuxINC` è sempre collegato al valore 4. Si seleziona l'ingresso 0 **se il circuito di controllo capisce che non si tratta di una CALL**. Il valore di PC viene sommato a 4 e il risultato viene scritto di nuovo in PC.

In caso di **BRANCH** sarà selezionato l'ingresso 1 di MuxINC e lo spiazzamento verrà scritto poi nel sommatore. Tale valore verrà aggiornato mediante la somma con PC. PC verrà aggiornato con l'indirizzo di salto in caso di branch. In questo caso non si somma 4 ma si somma lo spiazzamento che può essere positivo o negativo.

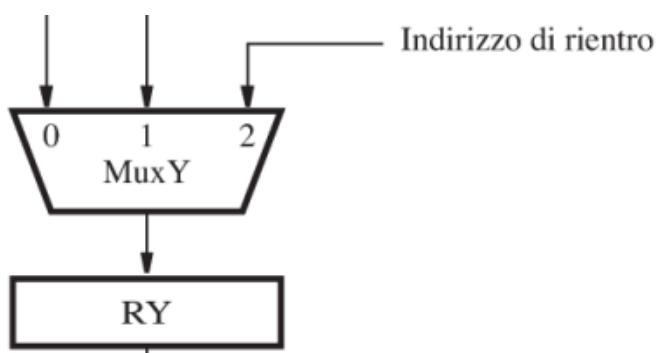
In caso di `CALL "ETICHETTA"`, **l'indirizzo della sub routine può essere scritta direttamente nell'istruzione**. In maniera diversa si può usare l'operando `Call_Register R9` e **R9 conterrà l'indirizzo della prima istruzione della sub routine**. L'indirizzo è stato precedentemente inserito nel registro R9, quindi si troverà nel banco dei registri.

`Call Register R9` è usato perché è più flessibile rispetto all'operando `CALL ETICHETTA` perchè l'indirizzo, con call register può essere codificato con 32 bit e quindi la routine **si può trovare in qualsiasi punto del programma**. In caso di CALL ETICHETTA, l'indirizzo è codificato con 7 bit e quindi ci sono 24 bit rimanenti per codificare la routine. In questo secondo caso la routine si trova semplicemente in **un punto SPECIFICO del programma**.

I 5 bit che identificano il registro viene letto nello stadio 2, cioè si troverà in RA.

Nell'immagine sopra, l'uscita di RA è collegato al multiplexer `MuxPC`. L'indirizzo della sub routine in R9 che prima si trovava nel banco dei registri, passa attraverso il MuxPC e poi viene messo su PC.

`PC-temp` è collegato a MuxY nel suo ingresso 2 e sarà utilizzato come istruzione di rientro dalla sub routine durante la chiamata di servizio.



Istruzione add

Nella linea Indirizzo A, B e C passano 5 bit perchè i registri sono codificati con 5 bit.

Passo Azione Add R3, R4, R5

- 1 Indirizzo di memoria \leftarrow [PC], Leggi memoria, IR \leftarrow Dati da memoria, PC \leftarrow [PC] + 4
- 2 Decodifica istruzione, RA \leftarrow [R4], RB \leftarrow [R5]
- 3 RZ \leftarrow [RA] + [RB]
- 4 RY \leftarrow [RZ]
- 5 R3 \leftarrow [RY]

Istruzione Load

RB viene comunque messo R5 perché ancora **non è stata decodificata l'istruzione** anche se non verrà utilizzato in questi primi stadi. Infatti in MuxB si viene selezionato il valore immediato. A valore immediato di MuxB c'è collegato il blocco **IMMEDIATO**, dove avviene l'estensione di bit.

Quando il dato viene letto dalla memoria, viene inserito nell'ingresso 1 di MuxY dal quale uscirà il dato letto e verrà messo in RY.

Passo Azione Load R5, X(R7)

- 1 Indirizzo di memoria \leftarrow [PC], Leggi memoria, IR \leftarrow Dati da memoria, PC \leftarrow [PC] + 4
- 2 Decodifica istruzione, RA \leftarrow [R7]
- 3 RZ \leftarrow [RA] + Valore immediato X
- 4 Indirizzo di memoria \leftarrow [RZ], Leggi memoria, RY \leftarrow Dati da memoria
- 5 R5 \leftarrow [RY]

Istruzione Store

In questa istruzione, il cambiamento rispetto alla Load avviene quando vi è il segnale di controllo che dice alla memoria che ci deve essere una "scrittura".

Passo Azione Store R6, X(R8)

- 1 Indirizzo di memoria \leftarrow [PC], Leggi memoria, IR \leftarrow Dati da memoria, PC \leftarrow [PC] + 4
- 2 Decodifica istruzione, RA \leftarrow [R8], RB \leftarrow R6
- 3 RZ \leftarrow [RA] + Valore immediato X, RM \leftarrow [RB]
- 4 Indirizzo di memoria \leftarrow [RZ], Dati per memoria \leftarrow [RM], Scrivi in memoria
- 5 Nessuna azione

Istruzione di salto incondizionato

Bisogna semplicemente aggiornare PC.

Passo	Azione
1	Indirizzo di memoria \leftarrow [PC], Leggi memoria, IR \leftarrow Dati da memoria, PC \leftarrow [PC] + 4
2	Decodifica istruzione
3	PC \leftarrow [PC] + Spiazzamento per il salto
4	Nessuna azione
5	Nessuna azione

Istruzione di salto condizionato

Passo	Azione
1	Indirizzo di memoria \leftarrow [PC], Leggi memoria, IR \leftarrow Dati da memoria, PC \leftarrow [PC] + 4
2	Decodifica istruzione, RA \leftarrow [R5], RB \leftarrow [R6]
3	Confronta [RA] con [RB], Se [RA] = [RB], allora PC \leftarrow [PC] + Spiazzamento per il salto
4	Nessuna azione
5	Nessuna azione

ACCESSI ALLA MEMORIA E VELOCITA' DELLA MEMORIA

Il periodo di **clock** è progettato per **durare quanto lo stadio 4** che è il **SOLITAMENTE più lungo**. Tutti gli stadi durano un ciclo di clock ma, durante un clock, alcuni stadi possono finire molto prima degli altri e quindi si mettono **in attesa**.

La **cache** permette di eseguire in **tempi più brevi** le operazioni durante un ciclo di clock.

Il segnale di controllo **MFC** (*Memory Function Completed*) indica che l'**operazione** in memoria è **stata eseguita** e in tal caso avrà valore 1.

E' un segnale generato dalla memoria perchè è l'unico componente a sapere se il dato è stato mandato in output, quindi **l'unico segnale di controllo NON generato dal processore**. Finchè MFC non è 1, bisogna attentare nello stadio 1 o stadio 4, dipende dallo stadio corrente.

SEGNALI DI CONTROLLO

I segnali di controllo **iniziano a muoversi nel circuito a partire dalla fine del secondo stadio** del percorso dati. Ciò avviene perchè **l'istruzione è decodificata** in quest'ultimo passo quindi solo da quel momento in poi è possibile "gestire" i vari segnali.

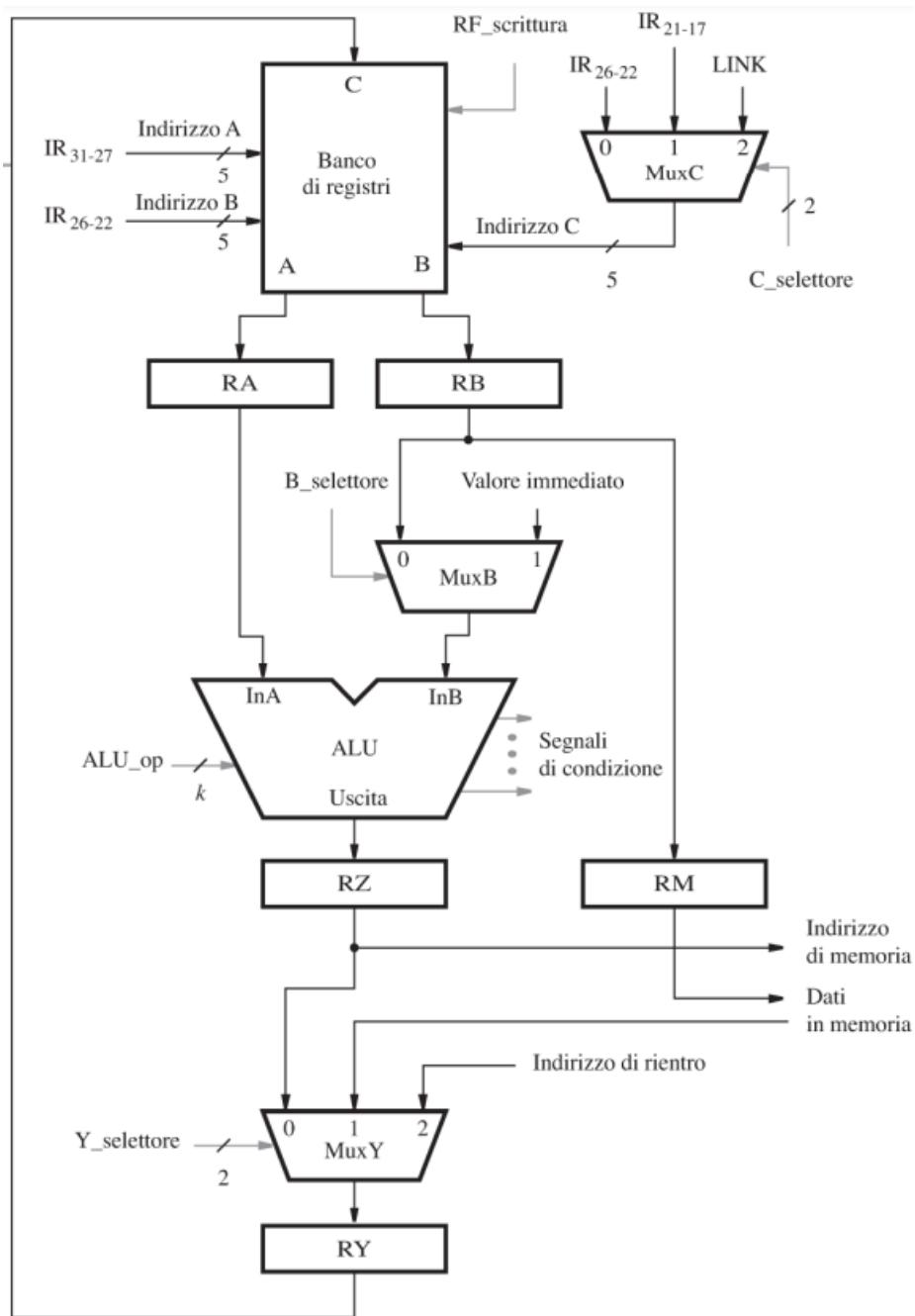
I **registri interstadi non hanno un segnale di controllo** perchè comunque dopo ogni stadio essi possono ricevere i dati eseguiti dalle precedenti operazioni. Quindi essi sono **sempre abilitati**.

Tra i segnali di controllo del banco dei registri si vedono i due degli indirizzi in input A e B, entrambi contenenti **5 bit**. Essi verranno prelevati dalle istruzioni.

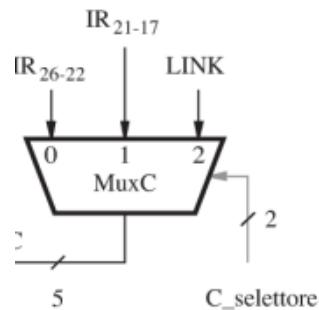
Esiste un **MuxC** che è collegato all'**Indirizzo C** e specifica l'indirizzo del dato C da scrivere.

LINK viene selezionato quando si ha una **CALL SUBROUTINE**.

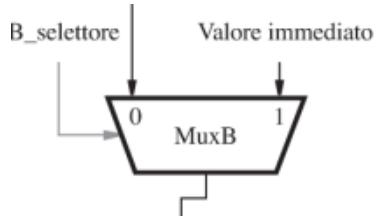
RF_scrittura abilita la **scrittura sul registro** nel banco dei registri.



I segnali arrivano agli altri componenti del percorso dati. E' possibile prendere come esempio il **multiplexer MuxC** che ha bisogno di un segnale di controllo, chiamato **C_selettore**, affinchè selezioni l'ingresso necessario. Ogni segnale di controllo fa viaggiare un **tot di bit** ed essi servono per far decidere ai

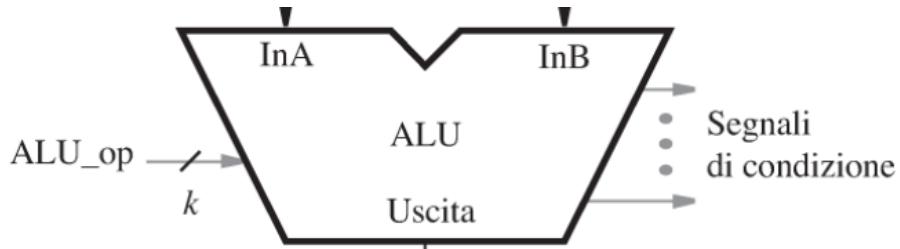


multiplexer quali input selezionare.



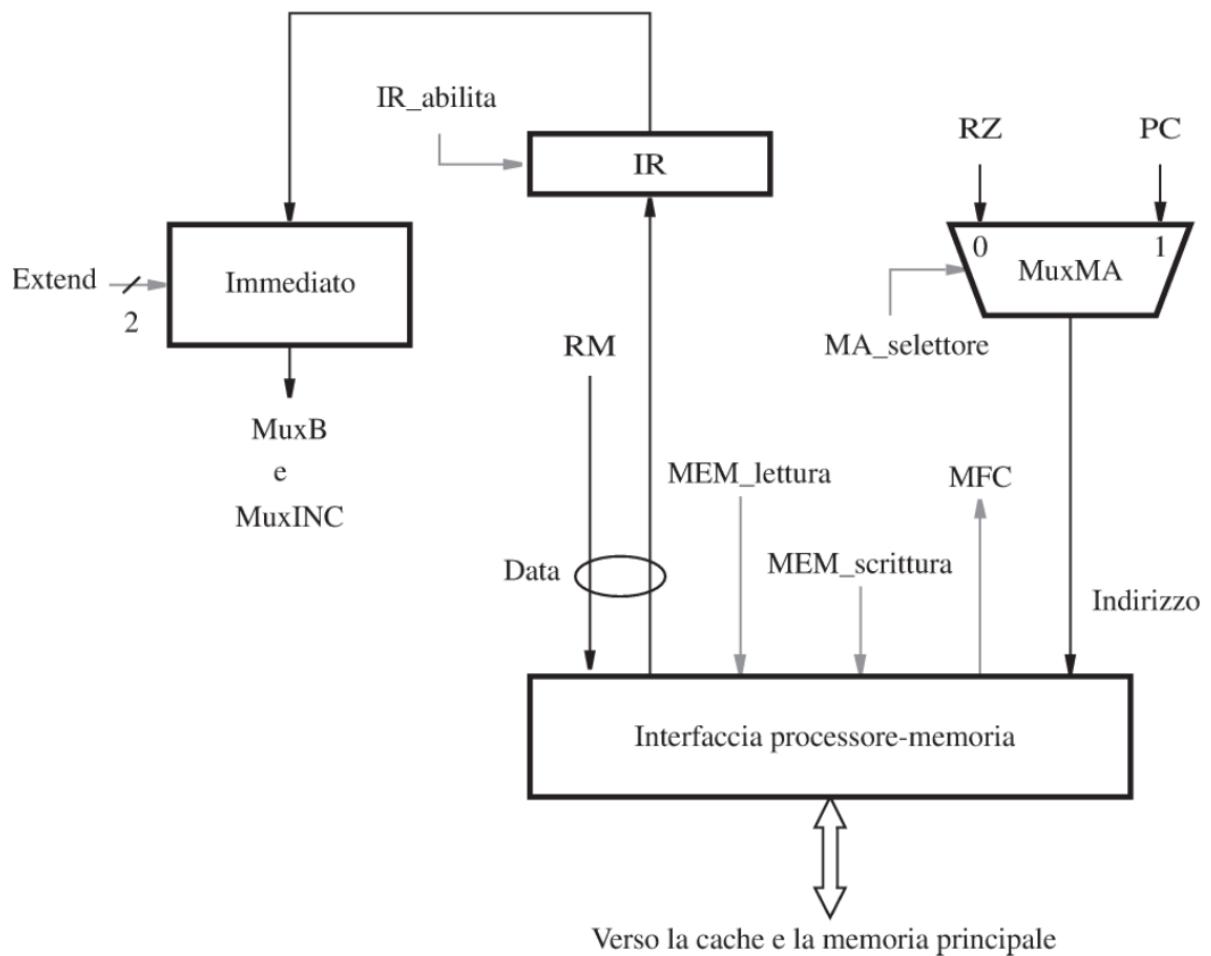
Per esempio: per selezionare un ingresso fra 3, sono necessari solo 2 bit perchè il massimo numero rappresentabile con 2 bit è proprio 3. (1 1)

Il segnale di controllo nell'ALU, denominato con `ALU_op`. Ci sono k bit sul segnale di controllo per selezionare 2^k operazioni. L'ALU genera anche dei segnali di condizione che permettono di scrivere i bit di condizione delle operazioni di compare.



Per l'interfaccia con la memoria ci sono anche i segnali di controllo appositi per far capire alla memoria se si deve eseguire una lettura `MEM_LETTURA` oppure di scrittura `MEM_SCRITTURA`.

In caso di **Load** si manda un segnale di controllo `MEM_LETTURA` alla memoria per indicare che si vuole prelevare un dato da essa.

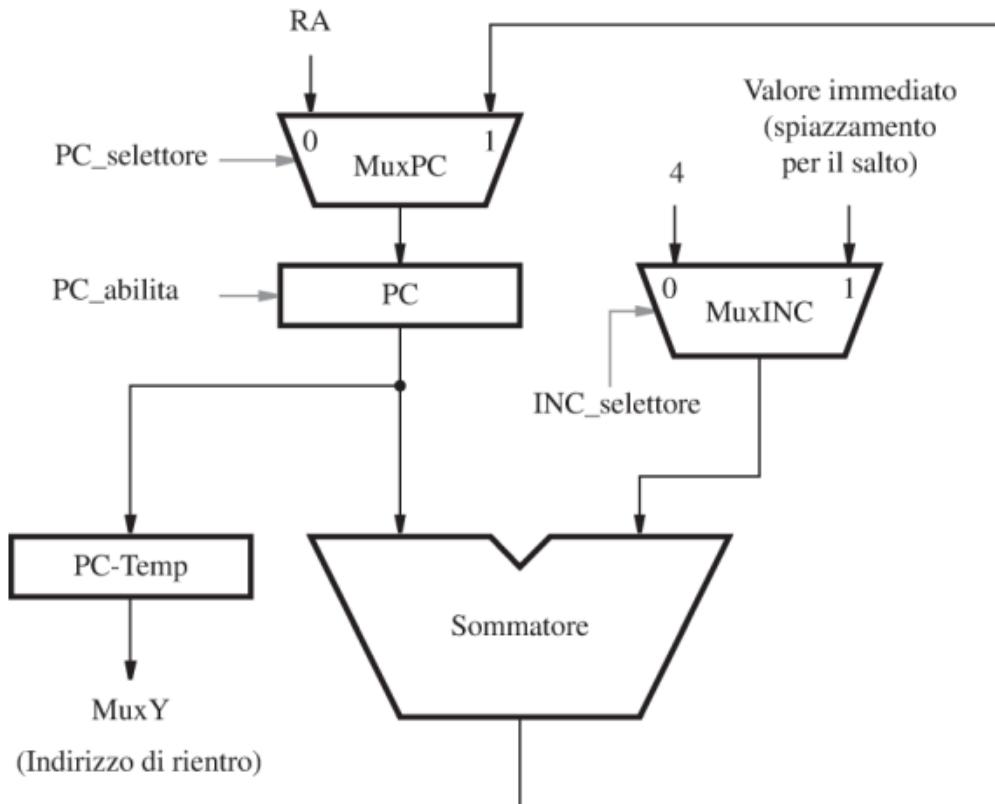


MFC va verso il processore. *Memory Function Completed* indica che la memoria ha completato l'operazione richiesta (lettura o scrittura) ed è l'**UNICO segnale uscente dalla memoria**. Serve per far capire al processore se può proseguire con lo stadio 5 oppure se si deve ancora attendere. Quindi lo **stadio 4 deve durare un po' di più** perché l'operazione non è stata ancora completata se MFC è a 0.

MuxMA serve per dare l'indirizzo della prossima istruzione / indirizzo del dato da leggere o scrivere. RZ dà l'indirizzo effettivo dopo le operazioni in ALU.

Se il dato prelevato dalla memoria è un valore immediato, esso verrà esteso estendendo i suoi bit mediante il blocco IMMEDIATO. Quest'ultimo pilotato dal segnale **Extend**. Esso serve per sapere se trattare il valore immediato con segno o senza segno.

Segnali al generatore di indirizzi



PC_abilita serve per sovrascrivere PC alla fine dello stadio 1 e alla fine dello stadio 3 per le istruzioni di salto.

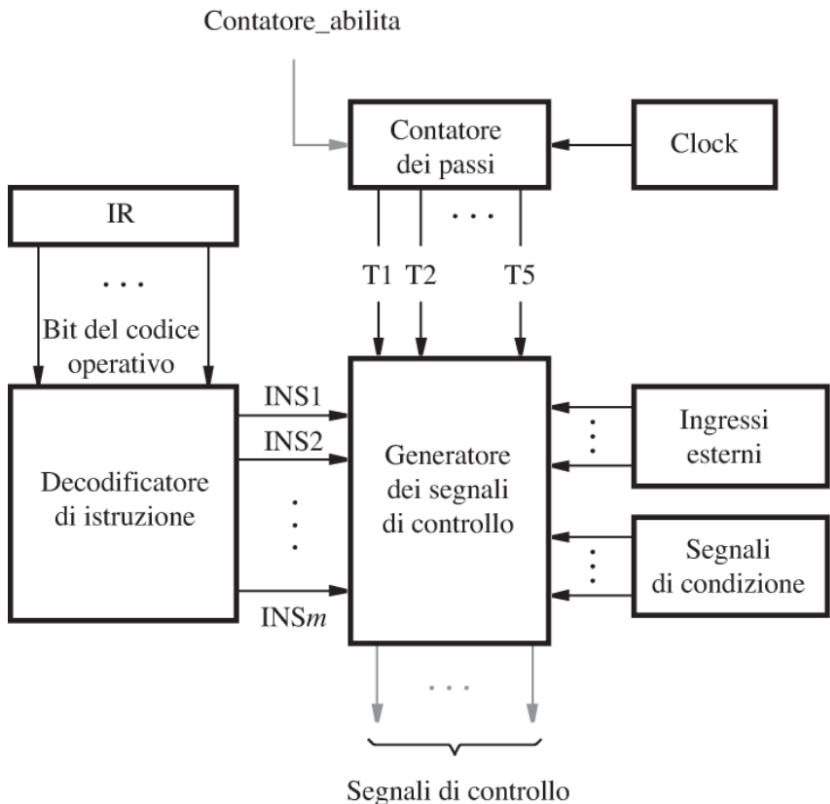
TIPI DI CONTROLLO (RISC)

Il **controllo cablato** e **controllo microprogrammato** sono 2 tecnologie che operano sui segnali di controllo. Chi progetta il processore sceglie se usare una tecnologia oppure un'altra.

Cablato

Questo tipo di controllo viene realizzato mediante una **rete combinatoria apposita** (tramite una funzione e una serie di espressioni logiche) che genera i segnali di controllo in base a quello che si ha in input. Per generare l'indirizzo servono alcune informazioni:

1. **passo** della sequenza, da 1 a 5
2. **istruzione corrente** da eseguire, cioè il contenuto di IR. L'op code nell'IR dell'istruzione;
3. **bit di esito correnti** (provenienti dall'ALU) (esclusi se non si tratta di una branch)
4. **segnali provenienti dall'esterno**, come le interruzioni e MFC.



Sono **FUORI DAL CIRCUITO DI CONTROLLO**: IR, CLOCK, INGRESSI ESTERNI, SEGNALI DI CONDIZIONE.

Il **decodificatore di istruzione** riceve l'istruzione da IR e manda un SOLO SEGNALE per capire il tipo di istruzione da eseguire mediante le linee INS1 o IN2 ecc.

Il **contatore di passi** serve per capire a quale passo/stadio ci si trova in quel momento. Questo contatore di passi si sposta dopo ogni ciclo di clock e lo segnala al generatore dei segnali di controllo mediante le linee T1, T2 rispettivamente in base al passo corrente.

Normalmente un fronte di salita di clock fa spostare il contatore di passi a quello successivo ma non è sempre così. Esistono alcune eccezioni, come quando **la memoria ci mette più tempo** a fornire il dato richiesto, quindi quando ancora il segnale **MFC** non è alto. In tal caso, il clock va avanti e rimane attivo il passo **T4**(passo corrente). Questo stesso caso si può verificare anche al passo 1 quando si comunica con la memoria.

Si passa al prossimo passo solo quando il segnale **contatore_abilita** è alto, quindi vale 1.

RF_scrittura è un segnale di controllo e va alto quando si deve scrivere qualcosa sul banco dei registri, come nel passo 5, e vale per le istruzioni LOAD,ADD,CALL ecc..

L'espressione di **RF_scrittura** può essere scritto con l'espressione logica:

RF_scrittura = T5 · (ALU + Load + Call) → "·" è l'**AND LOGICO** "+" è l'**OR LOGICO**

ALU indica **QUALSIASI** operazione esegue l'alu e non indica il componente in se.

RITARDO DELLA MEMORIA

Il `contatore_abilita` che abilita l'avanzamento del contatore di passi lavora con la seguente espressione:

Contatore_abilita = not(WMFC) + MFC → e `WMFC` è un segnale generato dal processore e sta per `Wait for MFC`, cioè vuol dire che si deve attendere la fine delle operazioni della memoria.

Invece `pc_abilita` ha la seguente espressione

PC_abilita = T1 · MFC + T3 · BR dove `BR` è una qualsiasi istruzione di salto.

Ogni espressione poi si rappresenta un circuito combinatorio con porte logiche.

PROCESSORI CISC

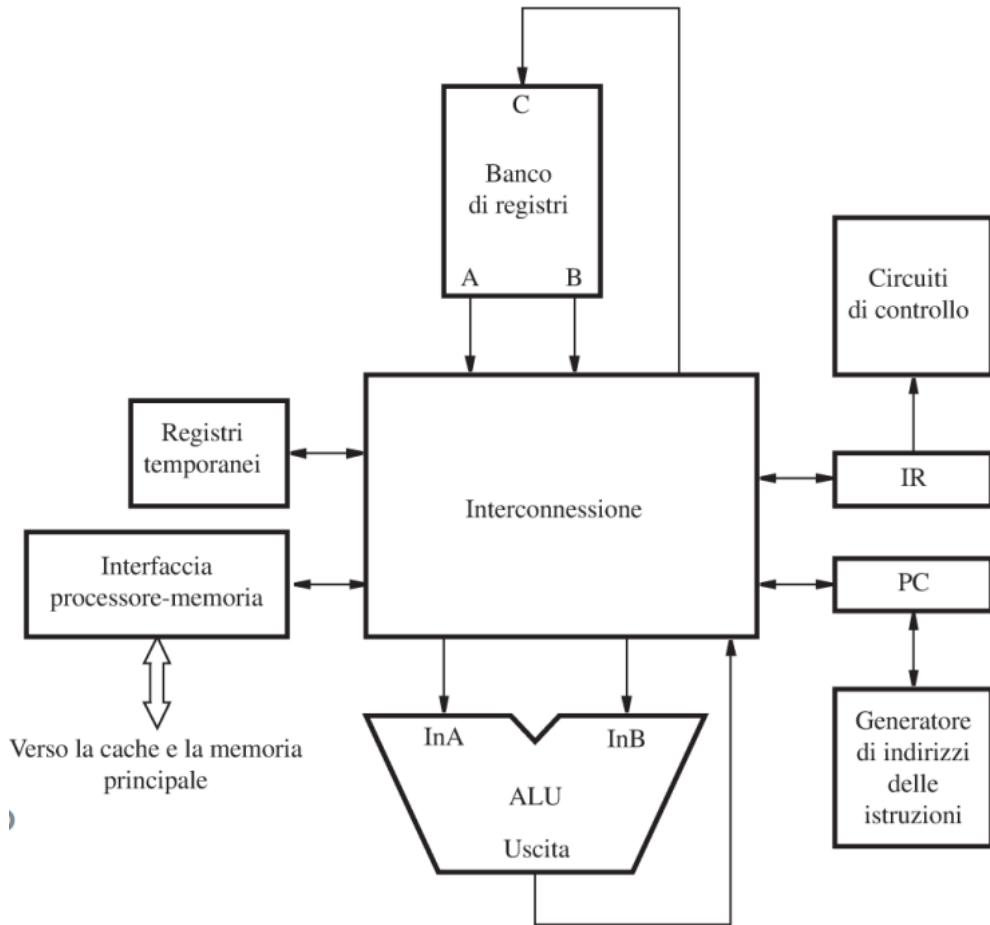
I CISC hanno un'organizzazione hardware più complessa per vari motivi:

- grande varietà di codici operativi e modi di indirizzamento
- operazioni con operandi in memoria, non solo Load/Store
- lunghezza variabile delle istruzioni (non sempre una parola di memoria)

Un processore CISC ha molte componenti simili al processore RISC:

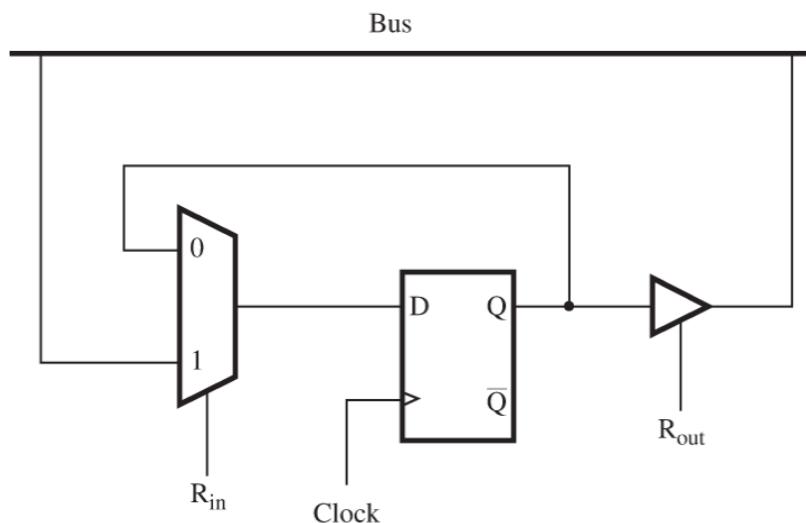
1. c'è un blocco di **REGISTRI TEMPORANEI**;
2. il blocco di **INTERCONNESSIONE** ha la maggior parte delle frecce in **doppio senso**.
I dati possono viaggiare **da qualunque punto a qualunque punto**.

Esso è realizzato mediante i BUS che sono un insieme di linee che collegano più dispositivi ai bus.



Se si collegano più dispositivi ai bus non si capisce chi è che sta imponendo un segnale al bus.
Bisogna evitare di imporre valori di altri collegamenti sullo stesso bus.

Per tale motivo è stata inventata **una porta a 3 stati** pilotato dal segnale di controllo "Rout".



Il segnale di controllo alto sulla porta a 3 stati impone il valore sul bus, altrimenti, se fosse 0, questa porta si scollega dal bus e quindi non c'è più il collegamento elettrico.

Quindi **soltanto un dispositivo sarà capace di imporre il proprio dato sul bus** mediante la porta 3 stati.

ORGANIZZAZIONE PROCESSORE CISC

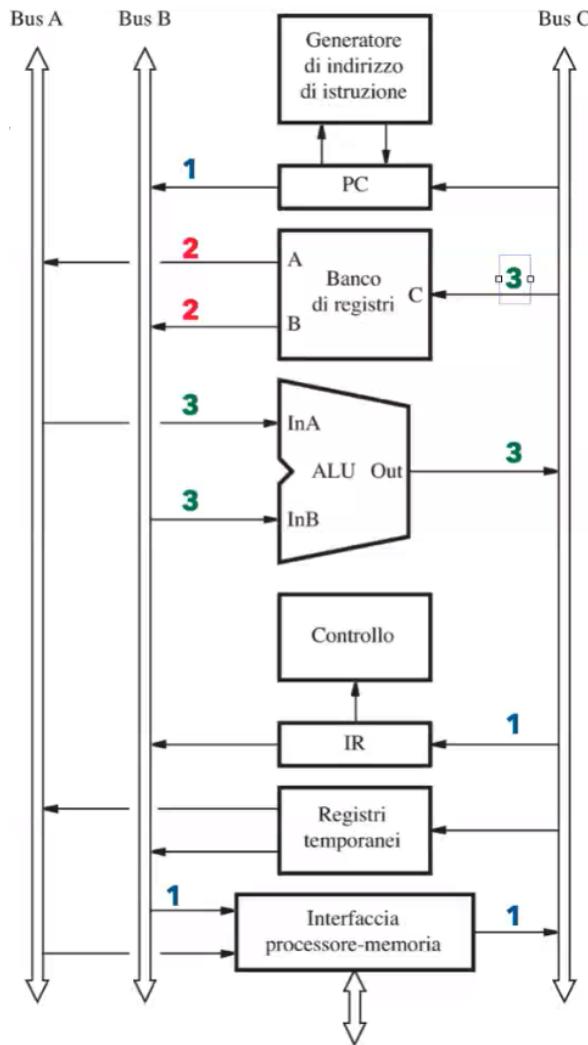
Nei meccanismi interni al processore CISC vi sono 3 **BUS** che fanno viaggiare i dati quindi tutto l'hardware è collegato tramite bus. Visto che alcuni di essi sono collegati allo stesso componente è necessario usare una porta a 3 stati per decidere se far passare o meno un dato.

I componenti funzionali sono:

- PC che viene collegato a 2 bus: **bus C** manda dati in input ed esso stesso viene inviato ad altri dispositivi mediante il **bus B**
- Banco dei registri
- ALU
- IR
- **Registri temporanei** hanno la stessa funzione dei registri interstadi nei processori RISC;
- Interfaccia processore-memoria

Add R5, R6

Esempio istruzione → **Add R5, R6** e vuol dire $[R5] = [R5] + [R6]$



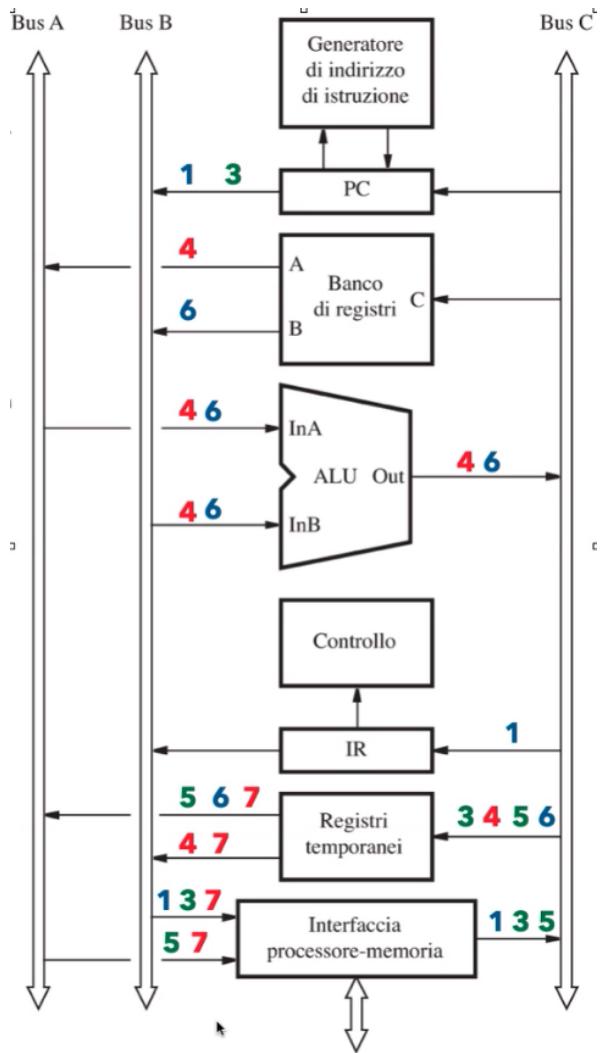
Tale istruzione viene effettuata in 3 passi:

- **Passo 1:** si invia il contenuto del PC all'interfaccia processore-memoria sul Bus B, e si inviano i dati dalla memoria a IR sul Bus C
- **Passo 2:** l'istruzione è decodificata e si leggono i registri R5 e R6
- **Passo 3:** le uscite A e B del banco di registri sono disponibili e inviate all'ALU usando i Bus A e B. L'ALU effettua la somma e invia il risultato sul Bus C, quindi alla fine del ciclo di clock del passo 3, si scrive nel registro R5

In RTN:

Indirizzo di memoria \leftarrow [PC], Leggi memoria, Attesa MFC, IR \leftarrow Dati da memoria, PC \leftarrow PC+4 Decodifica istruzione
 $R5 \leftarrow R5 + R6$

And X(R7), R9



Quest'istruzione avviene con passi diversi e un numero diversi di passi. Quindi nei **processori CISC i passi di esecuzione sono variabili**, a differenza dei processori RISC dove erano presenti solo da 5 passi.

`And X(R7), R9` esegue l'**AND** bit a bit fra i 2 registri e i passi sono i seguenti

1. Lettura istruzione da memoria e incremento PC, ovvero: Indirizzo memoria $\leftarrow [PC]$, Leggi memoria, Attesa MFC, IR \leftarrow Dati da memoria, $PC \leftarrow [PC]+4$
2. Decodifica istruzione
3. Lettura seconda parola dell'istruzione, ovvero: Indirizzo memoria $\leftarrow [PC]$, Leggi memoria, Attesa MFC, Temp1 \leftarrow Dato da memoria, $PC \leftarrow [PC]+4$
4. Calcolo indirizzo X(R7), ovvero: Temp2 $\leftarrow [Temp1]+[R7]$
5. Lettura dato in locazione X(R7), ovvero: Indirizzo memoria $\leftarrow [Temp2]$, Leggi memoria, Attesa MFC, Temp1 \leftarrow Dati da memoria
6. Calcolo AND, ovvero: Temp1 $\leftarrow [Temp1] \text{ AND } [R9]$

7. Scrittura risultato in memoria, ovvero: Indirizzo memoria \leftarrow [Temp2], Dati memoria \leftarrow Temp1, Scrivi in memoria, Attesa MFC

CONTROLLO MICROPROGRAMMATO

Si possono fare cose più flessibili rispetto al cablato dove si scrivono delle funzioni logiche e poi si realizza l'hardware corrispondente a tale funzione mediante porte logiche.

Il **controllo microprogrammato** è più generale e più **flessibile** perché occorrono **meno circuiti** da realizzare.

Ogni segnale di controllo deve essere generato durante l'esecuzione di un'istruzione e per ogni passo serve ciascuno dei segnali di controllo.

Ogni passo contiene la lista di tutti i segnali di controllo e il valore che deve assumere tale segnale di controllo per quel determinato passo.

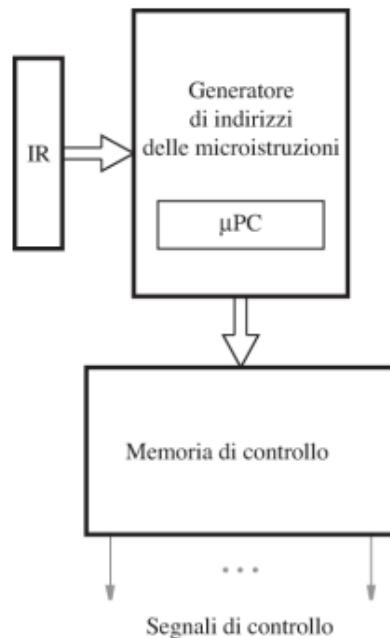
parola di controllo: insieme segnali di controllo necessari in un certo istante di tempo **per un singolo passo di esecuzione;**

microistruzione: parola di controllo impartita all'inizio di ogni passo;

microroutine: una sequenza temporale di microistruzioni di esecuzione di un'istruzione singola.

I primi passi di un processore CISC sono simili: prelievi istruzione e decodifica. Le prime due microistruzioni sono identiche.

Il controllo microprogrammato è più flessibile perché i segnali di controllo sono **già codificati** all'interno della memoria di controllo ed essi vengono mandati a tutti i dispositivi nei passi appositi in base alla posizione della microistruzione nella microroutine. Quindi serve un contatore di programma chiamato **microPC** per le microroutine affinché prosegua sequenzialmente con le microroutine nei vari passi.



Controllo cablato-microprogrammato

Per ogni istruzione presente nel controllo cablato devo avere un circuito che mi genera il segnale di controllo. E' più complesso da realizzare perché c'è una realizzazione hardware più complessa.

Il controllo cablato, nonostante tutto, è più **veloce** rispetto al controllo microprogrammato

Il costo dei circuiti logici non è più un fattore significativo per i processori RISC, quindi il **controllo cablato** è diventata nuovamente (rispetto al passato) la scelta **preferita** negli ultimi tempi.

6. PIPELINE IDEALE

PIPELINE: questo termine indica la tecnica che permette di eseguire istruzioni diverse in parallelo



Ogni istruzione è **terminata** ogni **5 cicli di clock**.

La successiva istruzione inizia dopo la fine del primo clock dell'istruzione precedente, quindi è come se le istruzioni fossero "**shiftate**" di un ciclo di clock.

Vi è il **vantaggio** che ad ogni ciclo di clock finisce l'esecuzione di un'istruzione e quindi vi è un guadagno di tempo di esecuzione.

Il **throughput** indica la **quantità di cose** che si fanno in un **intervallo di tempo** e in questo caso è **maggiori**.

I vari stadi vengono chiamati:

PRELIEVO → DECODIFICA → ELABORAZIONE → MEMORIA → SCRITTURA.

ORGANIZZAZIONE PIPELINE

Date le istruzioni:

100 Add R2, R3, #100

101 Or R4, R5, R6

102 Subtract R9, R2, #30

- Ciclo di **clock 1**, prelievo istruzione 1, In **B1**:

PC ha 101, IR ha **Add**

- Ciclo di **clock 2**, prelievo istruzione 2, decodifica istruzione 1

In **B1**: PC ha 102, IR ha **or**

In **B2**: PC ha 101, IR ha **Add**, RA ha [R3]

- Ciclo di **clock 3**, prelievo istruz 3, decodifica istruz 2, elabora istruz 1

In **B1**: PC ha 103, IR ha **Subtract**

In **B2**: PC ha 102, IR ha **or**, RA ha [R5], RB ha [R6]

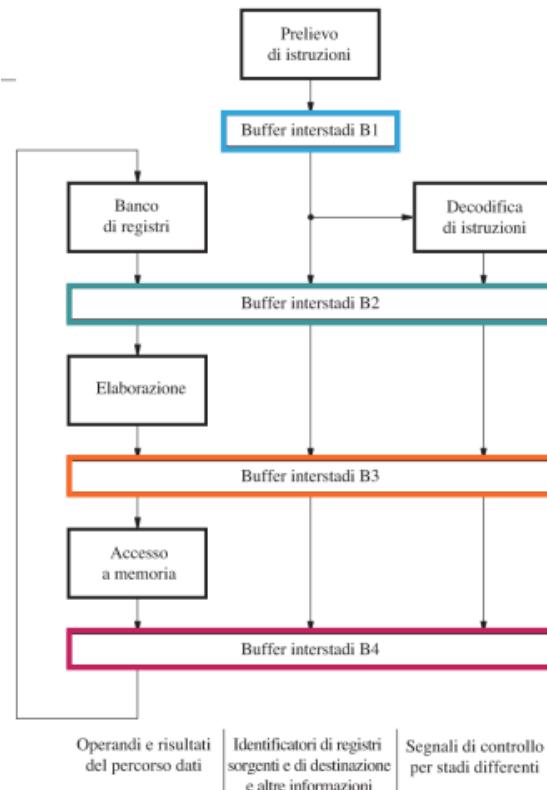
In **B3**: PC ha 101, IR ha **Add**, RZ ha [R3]+100

- Ciclo di **clock 4**, preliev istruz 4, decod istruz 3, elab istruz 2, mem istruz 1

In **B1**: PC ha 104, ... In **B2**: PC ha 103, IR ha **Subtract**, RA ha [R2]

In **B3**: PC ha 102, IR ha **or**, RZ ha [R5] OR [R6]

In **B4**: PC ha 101, IR ha **Add**, RY ha [R3]+100



- Ciclo di **clock 5**, preliev istruz 5, decod istruz 4, elab istruz 3, mem istruz 2, scritt istr 1

In **B1**: PC ha 105, ... In **B2**: PC ha 104, ...

In **B3**: PC ha 103, IR ha **Subtract**, RZ ha [R2]-30

In **B4**: PC ha 102, IR ha **or**, RY ha [R5] OR [R6]

In R2 si scrive [R3]+100

Al posto dei registri interstadi ci sono i **BUFFER INTERSTADI**. Ognuno di essi contiene i registri interstadi insieme ai registri **IR**, **PC** e altri **segnali di controllo** che servono a pilotare lo stadio successivo.

Questi buffer permettono di dare le informazioni giuste per lo **stadio successivo** e di alimentare lo stadio appena successivo. Il buffer servirà a fornire tutte le informazioni di decodifica delle istruzioni. Ogni informazione verrà portata, quindi, avanti nelle prossime istruzioni.

*Quindi il registro **IR**, dal buffer B1 verrà poi copiato nel registro IR presente nel buffer B2 successivo e così via. Il registro IR, che è stato copiato nel buffer successivo, verrà sostituito con la nuova istruzione appena prelevata.*

- Ogni buffer ha il registro PC del **buffer precedente** più lo stadio successivo di elaborazione del buffer precedente.

- B1 avrà un prelievo di un'istruzione al primo ciclo di clock.
- B2 avrà un prelievo della seconda istruzione + la decodifica(stadio successivo) dell'istruzione precedente al secondo ciclo di clock e così via.

PROBLEMATICHE PIPELINING

Realmente ci possono essere delle problematiche che possono rallentare l'operazione di pipeline **IDEALE** che prevede la sovrapposizione degli stati di esecuzione. Questo metodo **non è sempre realizzabile**.

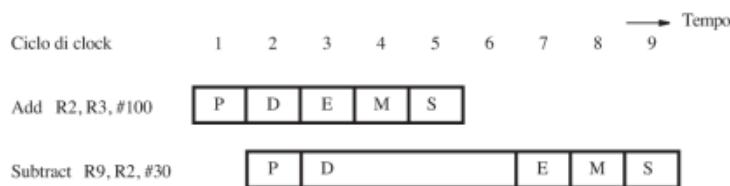
Vi è un **conflitto di dato** (detto **HAZARD**) che non permette alla seconda istruzione di prelevare un valore realmente desiderato nel secondo ciclo di clock perché esso non è ancora arrivato alla fase (ciclo 5) di scrittura. Un esempio è:

ADD R2, R3 #100 ; R2 é il risultato della prima istruzione
SUB R9, R2, #30 ; R2 é un operando della seconda istruzione

L'esecuzione di **Subtract** va in **stallo** perché deve attendere il completamento dell'istruzione precedente.

Esistono anche altri tipi di conflitti, quali: **ritardo della memoria, istruzioni di salto o limiti di risorse.**

Dipendenze di dato



Se si legge il valore di **R2 prematuramente**, si leggerebbe un valore **errato** visto che ancora la prima istruzione non ha scritto il valore finale in R2.

Quindi c'è un **allungamento delle operazioni** e quindi l'operazione di *Decodifica* della **Subtract** rimane in **stallo** per 3 cicli. Questo ritardo si ripercuote su tutte le istruzioni che hanno una dipendenza di dato.

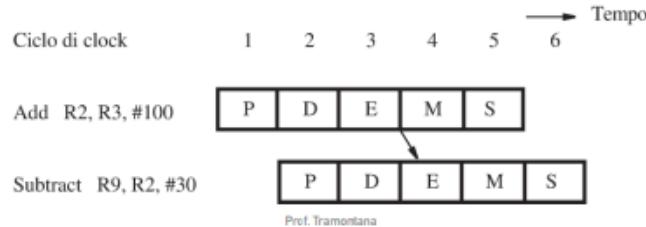
Una **possibile strategia** per migliorare l'operazione è quella che consiste nel **generare 3 cicli di clock di istruzioni NOP** (*no operation*) virtuali che provocano uno stadio di **decodifica** che si allunga. Tale allungamento viene detto anche inserimento di **bolle** e permette la giusta esecuzione della pipeline.

Inoltro degli operandi (HARDWARE)

Vi è anche **un'altra strategia** che è quella che permette di **inoltrare gli operandi** per la quale è necessario un hardware che permette di prendere il valore che occorre seppur non presente nel

registro corrispondente ma da qualche parte nel percorso dati, in particolare in **RZ**.

Questo **inoltro** si fa mediante il registro RZ che si trova alla fine dell'elaborazione dell'istruzione **Add**(quindi output ALU). Quindi il **dato corrispondente viene prelevato da RZ** piuttosto che dal banco dei registri.



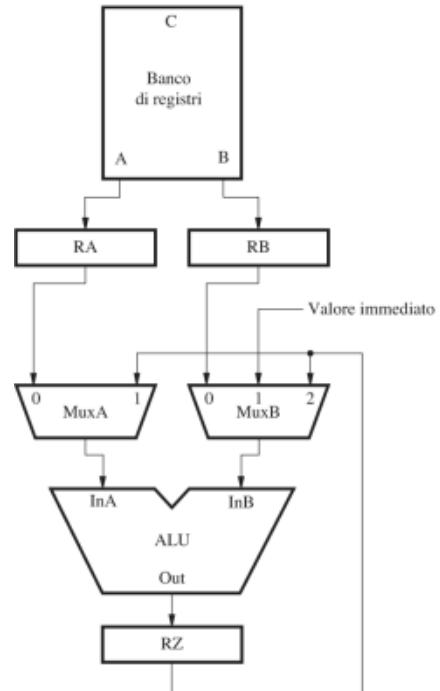
Questa procedura va effettuata per evitare rallentamenti, quindi **evitare stalli**.

Nel percorso dati, allora, viene aggiunto un multiplexer **MuxA** che viene collegato subito prima dell'ingresso A dell'ALU.

*Tale **MuxA** decide quale dato dare in pasto all'ALU tra RA o RZ.*

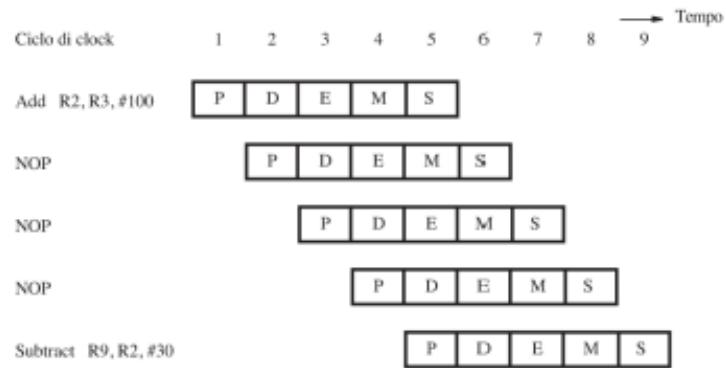
Visto che può essere un **primo operando** o un **secondo operando**, tale valore di RZ passa anche da **MuxB** nell'ingresso 2.

In caso di **dipendenza di dato** che non si trova all'istruzione appena successiva ma in quella ancora dopo,**quindi con un'istruzione di scarto**, il valore che ci interessa verrà **prelevato** dallo stadio successivo,**quindi da RY** visto che nel percorso dati il dato risultante viaggia verso il basso.



Gestione software di dipendenze di dati

L'inserimento di **NOP** consiste nel non far partire altre istruzioni durante il periodo di stall e vengono inseriti dall'assemblatore perché è a conoscenza del fatto che l'hardware non riesce a gestire questi rallentamenti. Sintatticamente assomiglia al seguente codice:



ADD R2,R3,#100

NOP

NOP

NOP

SUB R9,R2,#30

Inoltre l'assemblatore, se possibile, potrebbe fare un **riordine delle istruzioni** al posto delle **NOP**, senza cambiare la correttezza dell'esecuzione.

REMARK dipendenza di dato

Ricapitolando, si puo' rimediare a un conflitto dato da una dipendenza di dato mediante:

- **hardware:** se la dipendenza di dato è distante **UNA ISTRUZIONE**, si applica il metodo dell'**inoltro degli operandi** che permette di **trasferire il valore del registro RZ** (che si trova dopo l'ALU) all'ingresso della stessa ALU (nel ciclo di clock immediatamente successivo) mediante MuxA o MuxB; se la dipendenza di dato dista **DUE ISTRUZIONI**, allora si parla di quel metodo che permette di **trasferire il dato da RY** piuttosto che da RZ;
- **software:** quel metodo che permette di creare delle istruzioni **NOP** che consistono nel creare delle cosiddette **bolle** che permettono di rallentare l'esecuzione delle istruzioni. Le bolle sono delle istruzioni nulle.

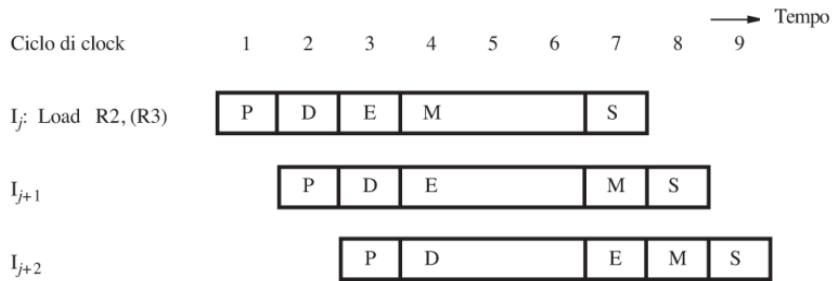
Ritardo della memoria e cache miss

Solitamente la memoria é molto più lenta rispetto al processore quindi, a volte, quest'ultimo deve **attendere il completamento delle operazioni in memoria**. Il segnale di **MFC** (Memory Function Completed) permette al processore di andare avanti con la sua esecuzione quando questo segnale è alto.

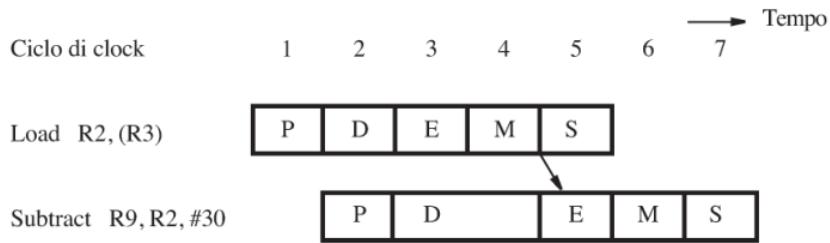
Se il dato che deve essere prelevato dalla memoria é presente nella **memoria cache**, allora tale dato viene prelevato da lì, dove l'esecuzione é decisamente **più veloce** della memoria in sé, quindi tale operazione potrebbe avvenire in **un ciclo di clock** soltanto. Altrimenti, **se il dato non é presente nella cache, si parla di CACHE MISS**. E questo provoca un rallentamento dell'esecuzione.

Il ritardo della memoria provoca lo stesso ritardo per le istruzioni successive ad essa. Se la memoria genera il dato in 3 cicli di clock, questo ritardo si ripercuote sulle istruzioni successive

ad essa.



Quando il dato viene prelevato dalla memoria e non dalla cache (cache miss) esso viene mandato al registro RY prima di essere mandato al banco dei registri. Tale dato è **necessario all'ELABORAZIONE** dell'istruzione successiva quindi viene preso dal registro RY e mandato in input all'ALU tramite i 2 multiplexer(MuxA e MuxB) mediante un **circuito apposito(hardware)**.

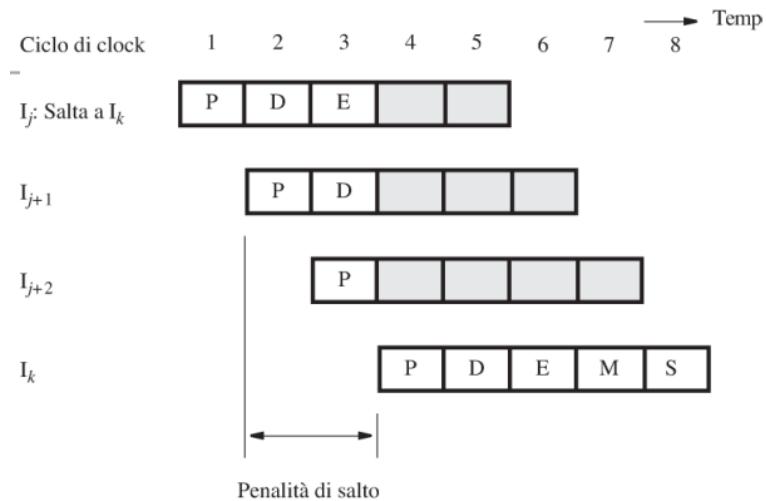


RITARDO NEI SALT

Salti incondizionati

Quando vi è un'istruzione di salto, il programma salta **avanti** o **indietro** nel programma dopo che l'indirizzo di destinazione è stato **calcolato(ELABORAZIONE)**, quindi, in Pipeline, le *istruzioni che si trovano subito dopo il salto NON VENGONO CONSIDERATE perché scartate prima della loro elaborazione.*

Durante lo scarto di queste istruzioni di intermezzo vi è una perdita di tempo perché i cicli di clock vanno avanti e comunque non eseguono istruzioni.



I cicli di clock che saltano e non portano a nulla vengono chiamati **penalità di salto**.

In questo caso si stanno perdendo 2 cicli di clock.

Il **20%** delle volte, in un programma si hanno istruzioni di **salto incondizionato**. Considerando che si saltano 2 cicli di clock, il programma **rallenta circa del 40%**.

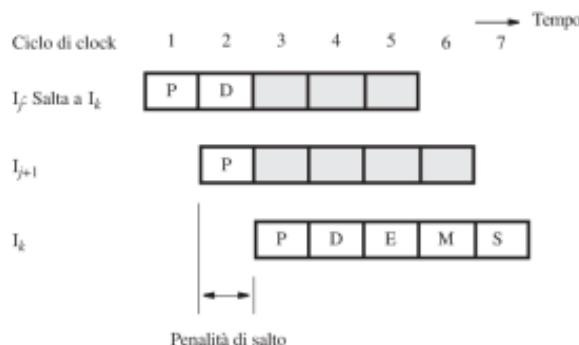
Ci sono varie strategie che permettono di diminuire la penalità di salto:

- con un hardware appositamente migliorato si anticipa il calcolo della destinazione di salto incondizionato al **secondo** ciclo anziché farlo al terzo, quindi si ha una penalità di un ciclo di clock piuttosto che due e si ha un rallentamento del programma solo del 20%.

Salti condizionati

In questo caso i valori espressi nella condizione si confrontano al **secondo passo** anziché al **terzo**, così, all'interno del *secondo passo*, si può determinare se si deve saltare e si può capire a quale istruzione si deve saltare. Tutto viene eseguito, ovviamente, con un **hardware appositamente migliorato**.

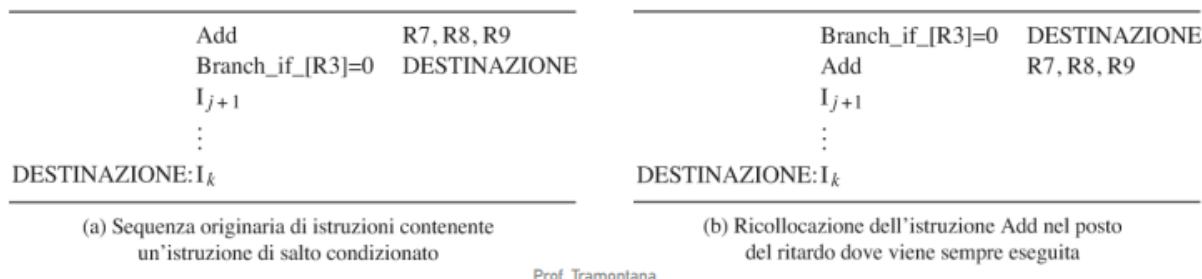
In questo caso **si ha un solo ciclo di clock di rallentamento**, quindi un solo **rallentamento del 20%**.



Posto del ritardo del salto

Quando si ha un'istruzione di salto, essa viene caricata e, al ciclo successivo, viene caricata anche l'istruzione successiva(*PRELIEVO*) visto che ancora non si sa a quale indirizzo di memoria bisogna saltare. Appena la prima istruzione arriva al secondo ciclo di clock, quindi alla decodifica, la macchina riesce a capire l'indirizzo della sub routine e quindi salta ad essa. Di conseguenza, **l'istruzione che veniva subito dopo** la prima istruzione viene **scartata**, facendo rimanere un **posto del ritardo di salto**.

Questo posto può essere **sostituito con un'istruzione utile che comunque dovrà venire eseguita** (per evitare di scartarla) e che il suo risultato non influenzi ciò che deve essere eseguito quindi **NON ha una dipendenza di dato**.



Tale riordino viene effettuato dall'assemblatore in modo automatico dopo aver controllato che effettivamente non ci sia dipendenza di dato. In caso **ci sia dipendenza di dato** verificata dal compilatore, tale **riordino** delle istruzioni **non viene eseguito**, si crea una **penalità di salto** e si crea una semplice **NOP** che indica che quella istruzione non deve essere eseguita.

Statisticamente si riesce a riempire il posto del ritardo di salto nel 70% dei casi.

Predizione di salti condizionati

Visto che si tratta di un salto condizionato è sempre un dubbio se si deve saltare ad una sub routine oppure no. Esiste una **tecnica** per capire se il salto verrà fatto oppure no. Esso si determina con una certa probabilità.

La decisione di salto viene presa al ciclo 2 e l'istruzione che segue potrebbe essere scartata dalla pipeline (se si salta).

È possibile **prevedere**, invece, **al primo ciclo** di clock, se l'istruzione prelevata è di salto in modo da ridurre la penalità di salto.

In questo caso bisogna **predire il risultato dell'istruzione di salto**.

Predizione statica di salto

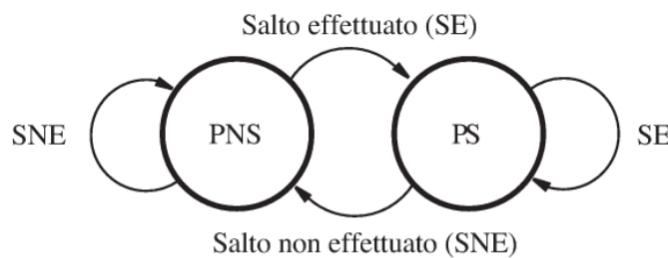
Si ipotizza, in modo **costate**, se l'istruzione di salto verrà eseguita oppure no.

Nel caso di un **do while()** è molto probabile che si salterà perché il ciclo è stato inserito per eseguire delle istruzioni per poi **tornare indietro nel programma** ed eseguirle di nuovo. In tal caso si parla di uno **spiazzamento negativo** e quindi la probabilità di indovinare è molto maggiore al 50%.

Si ipotizza, invece, che il salto non verrá mai eseguito se all'inizio del programma é stata inserita un'istruzione `while()`. Lo **spiazzamento**, in questo caso, sarà **positivo** (visto che dobbiamo saltare avanti nel programma) e si ha una maggiore probabilitá di indovinare.

Predizione dinamica di salto a 2 stati

Si ha la possibilità di fare una predizione dinamica di salto per cercare di ridurre gli errori di predizione. Questa predizione viene utilizzata dal **processore** mediante una **macchina a due stati**, **PS**(probabilmente salta) e **PNS** (probabilmente non salta).



(a) Un algoritmo a due stati

Per migliorare l'accuratezza della predizione, si usa l'attuale comportamento. Nella forma più semplice: il processore assume che la **prossima volta** che l'istruzione è eseguita la decisione sul salto sia la **stessa dell'ultima volta**.

Quindi, se si tratta di un `do while()` e il processore si trova nello stato PS, esso indovina tutte le volte finché il ciclo non finisce e quindi esso si **sbaglia solamente quando l'istruzione termina**.

Se si incontra di nuovo quell'istruzione, il processore valuta PNS all'inizio, perché l'ultima volta era in PNS visto che é finito il ciclo.

Questa informazione utilizza **un solo bit** per rappresentare la "storia dell'istruzione"(o "cronologia").

Predizione dinamica di salto a 4 stati

Questo schema é molto simile al precedente con l'aggiunta degli stati **MPNS** (Molto Probabilmente Non Salta) e **MPS**(Molto Probabilmente Salta).

Si assuma che lo stato sia inizialmente **PNS**, se

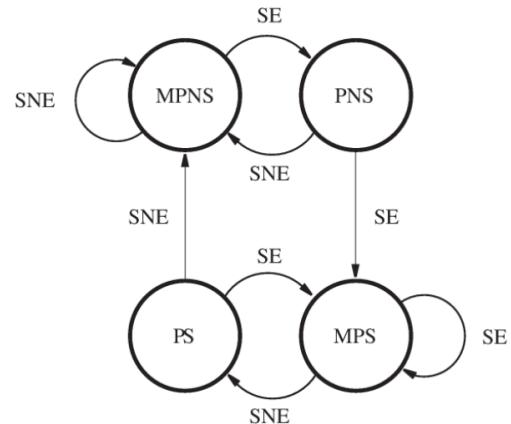
l'istruzione di salto viene eseguita lo stato si

aggiorna in **MPS**, altrimenti in **MPNS**

- Per un ciclo, che ha la condizione alla fine, si inizia con lo stato **PNS** e si cambia in **MPS** alla prima passata, la previsione sarà corretta per le altre passate;

- Per **l'ultima passata** la previsione è **sbagliata** e lo stato viene aggiornato a **PS**, visto che non si è effettuato il salto;

- Quando si incontra il **ciclo nuovamente**, si parte da **PS** e la predizione **sarà corretta se si fa almeno una passata**.



(b) Un algoritmo a quattro stati

Si prendono i dati della “storia dell’istruzione” per capire in che modo prevedere se si salta o no, **migliorando**, di conseguenza, le **probabilità di indovinare**.

Per tenere in memoria questa storia si utilizzano **2 bit**, visto che la combinazione di **2 bit** è di 4 , cioè gli stati della predizione.

Buffer di destinazione di salto

Il buffer di destinazione di salto è una **tabella** che contiene **tutte le istruzioni di salto incontrate nel programma**. All’interno di questa tabella vi sono: **indirizzo dell’istruzione di salto, indirizzo di destinazione di salto, bit di predizione (uno o più bit)**.

Man mano così la tabella si riempie (ogni secondo ciclo di clock delle istruzioni di salto). In questo modo, al prelievo dell’istruzione, quindi al **primo ciclo di clock**, è possibile confrontare l’istruzione con questo buffer e, se si tratta di un indirizzo di salto già registrato, è già possibile leggere l’indirizzo di destinazione di salto e il bit di predizione.

Se **non vi è un match al primo ciclo di clock** con questo buffer, vuol dire che **l’istruzione non è di salto oppure non è stata ancora presa in considerazione dal buffer**; in questo secondo caso si aspetta il ciclo 2, si decodifica l’istruzione e viene **conservata** nel buffer di destinazione di salto.
Solitamente il buffer contiene pochi elementi (1024) in modo che sia abbastanza veloce.

LIMITI DI RISORSE

Durante la stessa pipeline, ogni istruzione si trova nel proprio stadio di esecuzione. Può succedere, a volte, che un’istruzione deve **accedere ad una parte di hardware MENTRE**

un'altra istruzione vuole **accedere alla stessa risorsa hardware**. In questo caso vi é un conflitto nell'uso delle risorse.

Un caso tipico é il caso quando si vuole accedere alla memoria.

Se si immagina di allungare verticalmente la pipeline, si nota che, prima o poi, se vi é un'istruzione **Load** o **Store**, é probabile che essa venga scritta quando un'altra istruzione sta accedendo alla memoria mediante il proprio primo ciclo di clock dove **Preleva** l'istruzione. Quindi capita che le due istruzioni vogliono accedere alla memoria allo stesso tempo (P e M sono sovrapposti nello stesso ciclo) e quindi un'istruzione andrá in **stallo** affinché attenda la fine dell'istruzione in conflitto di Load o Store.

Statisticamente il **25%** dei casi contengono istruzioni di **Load** o **Store**, quindi il processo si **rallenta nel 25% dei casi**.

Per ovviare a tale problema vengono usate **MEMORIE CACHE SEPARATE**: una cache contiene i **dati** e una cache contiene **istruzioni da prelevare**.

In questo modo si possono eseguire istruzioni di Prelievo e Load e Store allo stesso momento, quindi un ciclo di clock puó eseguire gli stadi **P** ed **M** allo stesso tempo.

Alcuni calcolatori hanno scritto “**cache unica**” che permette di capire che tale procedura non é eseguibile, quindi risulta un minimo piú lento rispetto ad altre macchine.

VALUTAZIONE DELLE PRESTAZIONI

Per valutare le prestazioni di un processore si utilizza la formula:

$$T = \frac{N \cdot S}{R} \text{ dove:}$$

- **T** **tempo** di esecuzione
- **S** numeri di **cicli** di clock **per istruzione**
- **N** **conteggio** dinamico delle **istruzioni**
- **R** **frequenza** di clock

La frequenza di operazione throughput **P** (detto anche **flusso**) indica meglio la prestazione di un processore ed esso indica il **numero medio di istruzioni eseguite nell'unità di tempo**. Esso viene calcolato:

- Senza pipeline $P_{np} = \frac{R}{S}$ (dall'equazione di partenza)
- Con pipeline ideale $P_p = R$

P ideale, però, é affetto da stalli, penalità di salto, limiti di risorse ecc..

Il **numero di istruzioni N** é possibile calcolarlo anche con la seguente formula:

$$N = P \cdot T \text{ da cui si puó anche ricavare il flusso: } P = \frac{N}{T}$$

Effetti di stalli

Stima degli effetti quantitativi dei conflitti sul guadagno della pipeline, valutando δ .

δ è l'incremento del tempo di esecuzione, con $\delta=0$ si ha il caso ideale.

- Per il processore con inoltro degli operandi:
 - Stalli residui (pari a un ciclo) per dipendenze di dato da istruzioni Load

Esempio: istruzioni Load pari a 25% del conteggio dinamico, con 40% di queste seguite da istruzioni dipendenti:

$$P_p = \frac{R}{1 + \delta_{tot}} \text{ FORMULA GENERALE}$$

$$\delta_{stallo} = i_{strload} \cdot i_{strdipend} \cdot cicli_{extra} = 0.25 \cdot 0.40 \cdot 1 = 0.1$$

$$P_p = \frac{R}{1.1} = 0.91R$$

Quindi si perde circa il **10% delle prestazioni** perfette nel caso di dipendenza di dato di istruzioni Load.

Effetti di penalità di salto

Per un processore con:

- calcolo della destinazione di salto al secondo stadio;
- predizione dinamica di salto;
- buffer di destinazione del salto(predizione al primo stadio);

Si ha che le **penalità** di salto residue **dipendono solo dagli errori di predizione dei salti**.

In questo caso $\delta_{penalita_salto}$ è quell'incremento di esecuzione dovuto a tale penalità.

Es. istruzioni di salto: 20% del conteggio dinamico, tasso errore di predizione: $10\% \delta_{penalita_salto} = i_{strsalto} \cdot tasso_{errore} \cdot cicli_{extra} = 0,20 \cdot 0,10 \cdot 1 = 0,02$

Quindi, è evidente che **inserire dell'hardware che permette di prevedere i salti aumentano le prestazioni effettive**, quindi il flusso (throughput), anche se di poco.

Effetti di cache miss

Il δ aumenta considerevolmente quando trattiamo gli effetti dovuti a cache miss. Si applica la formula:

$$\delta_{miss} = (m_i + d \cdot m_d) \cdot p_m \text{ dove:}$$

- m_i istruzioni prelevate soggette a cache miss;
- d istruzioni load del conteggio dinamico;
- m_d accessi alla memoria soggetti a cache miss;

- p_m cicli di clock che servono alla RAM per la risposta.

Esempio: $m_i = 0.05, d = 0.3, m_d = 0.1, p_m = 10$

$$\delta_{miss} = (istr_{prelevate} + istr_{load} \cdot accessi_{memoria}) \cdot cicli_{clock}$$

$$\delta_{miss} = (0.05 + 0.3 \cdot 0.1) \cdot 10 = 0.8$$

$$\text{Quindi la formula finale sarebbe: } P_p = \frac{R}{1 + \delta_{stallo} + \delta_{penalitaSalto} + \delta_{miss}} = \frac{R}{1 + 0.92}$$

Si nota che l'effetto di **cache miss** è quello che incide di più sul throughput rispetto agli altri due effetti.

In casi come questi si possono seguire due soluzioni:

1. Aumentare la **capienza** della cache;
2. Inserire **più livelli** di cache affinché la possibilità di trovare il dato in cache aumenti.

NUMERO DI STADI DELLA PIPELINE

In base alle formule prima scritte, si nota che più si aumenta S (il numero di cicli di clock per istruzione) e più si ha un incremento del **throughput**.

Aumentare il numero degli stadi vuol dire che all'interno della pipeline si avranno più istruzioni contemporaneamente. Quindi le istruzioni aumentano proporzionalmente al numero di stadi, quindi si hanno **pipeline più profonde**.

In questo caso le dipendenze fra le istruzioni si devono calcolare per un frammento più alto di istruzioni e ciò vuol dire che è **più probabile avere delle dipendenze**; quindi sotto questo punto di vista è tutto equilibrato fra **stadi e dipendenze**.

In questo caso si parla di **parcellizzazione**, ovvero **la divisione di stadi in stadi più piccoli**.

Così facendo il **costo di realizzazione cresce**, ma il guadagno è relativamente piccolo man mano che si incrementano il numero degli stadi.

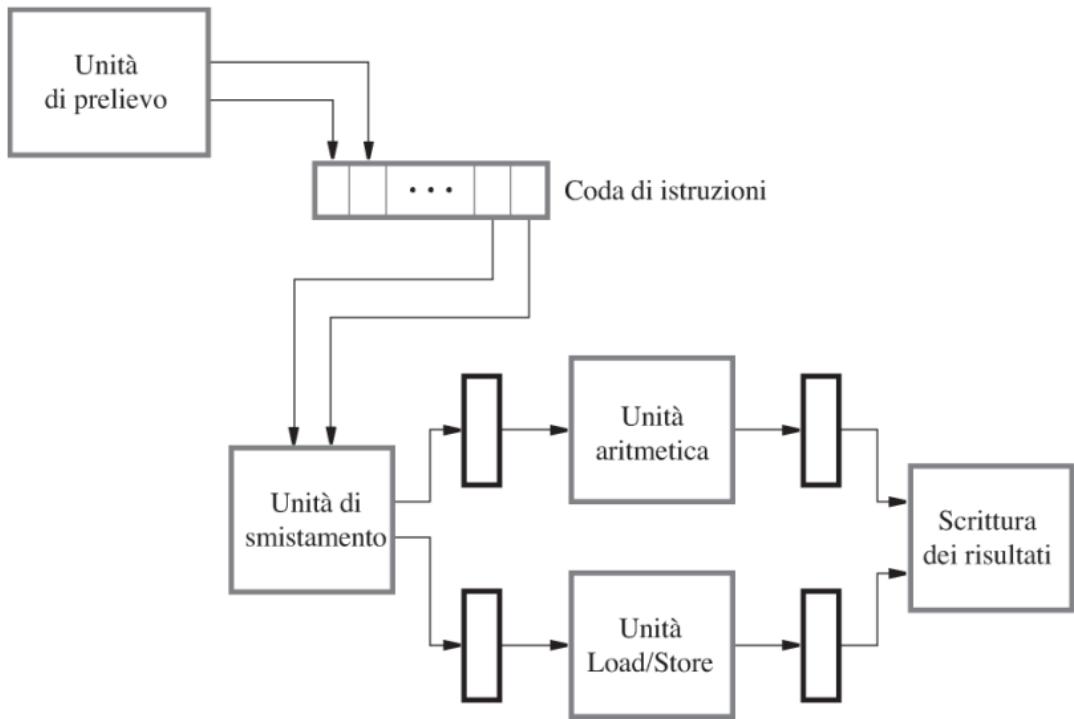
Ogni volta che si aggiunge uno stadio si ha una diminuzione del guadagno visto che comunque il tempo maggiore di esecuzione degli stadi è dato dal tempo necessario all'ALU per eseguire le istruzioni.

Il numero di stadi, pertanto, non può aumentare a dismisura per tali motivi.

FUNZIONAMENTO SUPERSCALARE

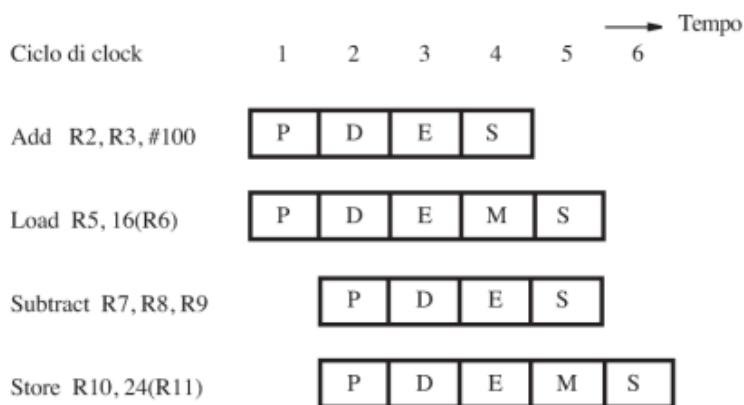
Piuttosto che aumentare il numero di stadi, **si può utilizzare anche un'altra strategia** per migliorare le prestazioni: **FUNZIONAMENTO SUPERSCALARE**.

Si tratta di inserire **più unità funzionali in parallelo** e ciascuna di esse con la **propria pipeline**. In questo modo, molte istruzioni finiranno allo stesso momento. Per ogni ciclo di clock, nella pipeline **si prelevano, quindi, più istruzioni allo stesso tempo**, per esempio 4, quindi il throughput maggiore di 1.



In questo schema si hanno delle componenti hardware aggiuntive che eseguono separatamente le istruzioni in base al loro codice operativo. In caso si abbiano 2 istruzioni, una **LOAD** e una **ADD**, è possibile eseguire queste due istruzioni **contemporaneamente** visto che **L'UNITÀ DI SMISTAMENTO**:

- gestisce la direzione verso le unità corrispondenti a tali istruzioni;
- **si assicura che il buffer sia libero per contenere il risultato dell'operazione.**



*Si nota che la ADD si completa tranquillamente in 4 stadi.
Questo procedimento è quello **ideale** visto che **non vengono considerate le dipendenze di dato.***

Salti e dipendenze di dato (SUPERSCALARE)

Nella realtà dei fatti bisogna considerare anche le dipendenze di dato.

Nel caso ideale le istruzioni sono date in pasto al processore in modo alternato:

“*Add,store,add,load,subtract,store* affinché si abbia un throughput migliore possibile.

Per fare questa **ALTERNANZA** ci pensa il compilatore stesso in base al tipo di processore che monta. Tale procedimento, però è possibile esclusivamente se non c'è dipendenza di dato perché non si possono spostare delle istruzioni in punti di programma dove tali dati non siano stati ancora calcolati.

Soluzioni:

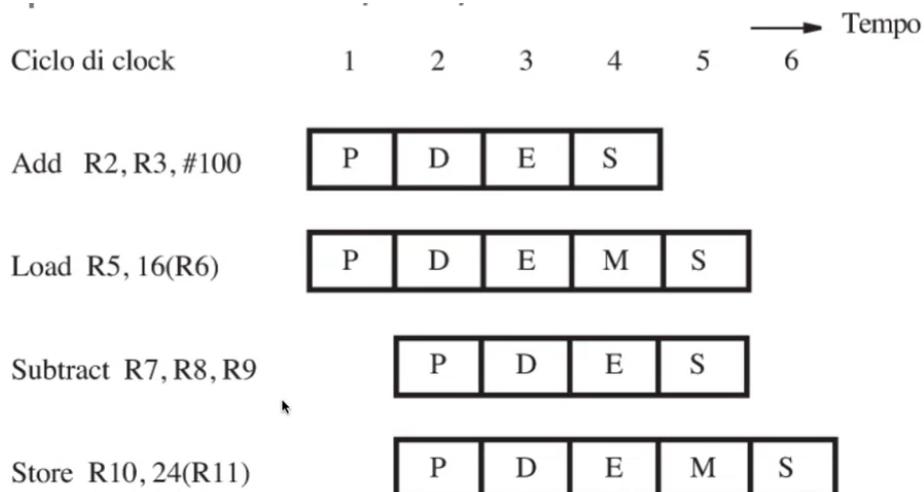
- **SALTI**: si usa una predizione dinamica con **buffer di destinazione di salto + esecuzione speculativa**: visto lo spazio hardware e le unità funzionali a disposizione, è possibile eseguire delle istruzioni tenendo conto delle predizioni dei salti che verranno fatti o no, quindi **il dato viene calcolato comunque** e, **se si verifica la condizione**, viene **salvato** altrimenti viene **scartato** (visto che è stato calcolato comunque).
- **DIPENDENZE DI DATO**: In questo caso, se in un'istruzione vi è una dipendenza di dato (quando vengono eseguite contemporaneamente mediante l'unità di smistamento) l'istruzione contenente la variabile **dipendente dovrà attendere nell'unità di smistamento** che essa venga calcolata, anche se ci fosse un inoltro degli operandi (mediante il registro RZ o RY).

Esecuzione fuori ordine

La dipendenza di dato può alterare l'ordine delle istruzioni visto che l'unità di smistamento potrebbe far passare le istruzioni completamente indipendenti dalle altre.

Questo comporterebbe un ordine di risultati **non desiderato** visto che viene effettuato un riordine di istruzioni che però non influisce più di tanto visto che le istruzioni indipendenti sarebbero state comunque eseguite.

In un caso del genere:



Si potrebbe presupporre che la **Load** abbia generato **un'eccezione** (ovvero è stato messo un indirizzo di memoria **non accessibile**). In questo caso il programma dovrebbe terminare però, per via del funzionamento superscalare, viene eseguita anche la **Subtract**.

Per ovviare a questo problema vi è un'unità chiamata **COMMITMENT** che:

- **gestisce** le operazioni finali di ogni istruzione;
- **riordina** le istruzioni;
- **annulla** la scrittura in caso di **ECCEZIONE**.

Gli effetti dell'esecuzione sono irreversibili solo dopo il commitment.

Unità di smistamento

Come accennato precedentemente questo componente smista le varie istruzioni fra unità aritmetiche e unità di load/store.

Inoltre, essa si “**prenota**” dei buffer sui quali scrivere i rispettivi risultati dopo l'esecuzione dell'operazione.

Nella **FASE DI PRENOTAZIONE**, se vi è un riordino delle istruzioni, si può verificare un **DEADLOCK**. Se si nota la figura precedente, si vede che la **Subtract** viene eseguita dopo la **Load**. Se si invertono queste istruzioni si avrà una prenotazione del **buffer** per la **Subtract**. Dopodichè si tenterà di prenotare il buffer per la **Load** ma sarà impossibile perché occupato dalla **Subtract** e quest'ultima non può completarsi perchè ha bisogno dell'esecuzione di **Load**. Il rimedio più semplice del **DEADLOCK** consiste nello seguire l'ordine ben preciso della scrittura delle istruzioni oppure nel calcolo ben preciso della prenotazione dei buffer.

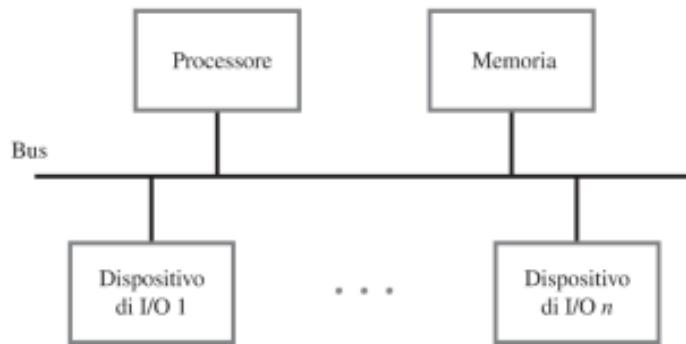
Pipeline processori CISC

Si ricorda che nei processori CISC si hanno:

- istruzioni più **complesse**;
- **più operandi** in memoria;

Per esempio: **Move R5, (R8)+** ha una dipendenza di dato su **R5** e **R8** che deve essere gestita con un hardware molto più complesso. Solitamente, nella pipeline di processori CISC si utilizzano **istruzioni CISC mappate come istruzioni RISC** quindi eseguibili con la normale pipeline RISC.

STRUTTURA A BUS



Il **BUS** é un insieme di linee in parallelo fra loro che connettono il **processore con la memoria** e le **periferiche**. Vengono messi in comunicazione **due dispositivi alla volta**, quindi vi é solo **un dato alla volta** che viaggia su queste linee.

Questo bus spesso é partizionato in più bus. Essi si distinguono in base al **tipo di dato** che viaggia su esso. Si hanno:

- bus degli **indirizzi** sul quale viaggiano gli indirizzi emessi dalla CPU destinati a indirizzare determinati dati. Il **numero di linee** del bus degli indirizzi **dipende dallo spazio di indirizzamento**. Se quest'ultimo é a 16 bit, allora serviranno 16 linee per questo tipo di bus;
- bus di **dati**;
- bus di **controllo** che servono per capire il tipo di operazione da eseguire (lettura o scrittura), interrupt, MFC(Memory Function Completed).

Funzionamento del bus

Ogni bus deve seguire un **protocollo** del bus che indica quali sono le **attività** che possono fare la CPU e le periferiche e quali sono le proprie **tempistiche**.

Un segnale molto importante all'interno del protocollo é il segnale chiamato R/\bar{W} . Questo segnale specifica se il processore vuole eseguire una **lettura** ($R/\bar{W}=1$) oppure una **scrittura** ($R/\bar{W}=0$) su una determinata periferica. (R/\bar{W} sta per *Read / Write*).

Il segnale R/\bar{W} va messo alto per un determinato tempo e messo basso per un altro determinato tempo perché bisogna dare il tempo necessario alle periferiche di immettere il dato sul bus o fare altre operazioni con il bus. Tale tempo é specificato nel protocollo del bus.

Scendendo piú nei dettagli si distingue:

- **bus sincroni**, ovvero la **temporizzazione** é fissata al processore e alla periferica. Il **segnale di clock** del bus é **conosciuto da tutte le componenti collegate al bus**;
- **bus asincroni**, la **temporizzazione** é dettata da **altri segnali che comunicano fra loro**.

Si definiscono anche:

- **MASTER**: quel componente che **inizia le varie richieste** di lettura/scrittura che solitamente è il processore;
- **SLAVE**: quel componente che **risponde a tale richiesta** mandando il dato richiesto sul bus di riferimento.

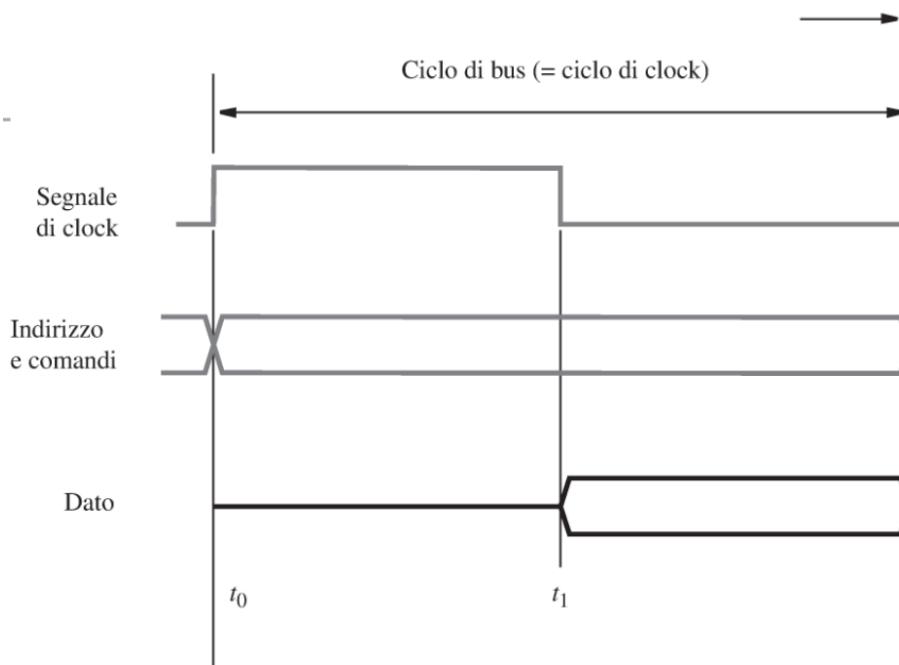
BUS SINCRONO

Il segnale di clock del bus è **UNICO**. Nell'istante t_1 si ha il transito tra il segnale di clock alto e quello basso.

Il grafico viene rappresentato con due linee parallele per indicare che vi è un dato importante su quella linea.

Quando le due linee si incrociano vuol dire che su quelle linee c'è stato un cambio di valore.

*Quando una linea è singola, vuol dire che non c'è un valore non significativo, quindi le periferiche sono scollegate da quel bus e possono avere un valore spurio. (si dice anche **Alta Impedenza**)*



In caso di **lettura** avvengono i seguenti passaggi:

1. **MASTER** mette l'**indirizzo** della periferica dal quale desidera leggere il dato e $R/\overline{W}=1$ sulla linea *Indirizzi e comandi*. Tale **dato** rimane **costante** fino alla fine dell'operazione
2. Il dato viene letto da **molteplici periferiche** e solamente **UNA** lo **riconoscerà** come **proprio indirizzo**;

3. La periferica deve **poder riconoscere** il proprio indirizzo ed **emettere** il dato **entro il tempo** t_1 ;
4. **MASTER** legge il dato emesso da **SLAVE** nel tempo che va da t_1 a t_2 .

In caso di **scrittura** $R/\overline{W}=0$ e **MASTER** scrive sulla periferica entro il tempo t_1 e t_2 .

Ritardi di propagazione

Il caso sopra spiegato é il caso ideale. Nella realtá dei fatti, le componenti sono **distanti** fra loro quindi i segnali non vengono visti allo stesso istante da tutte le componenti.

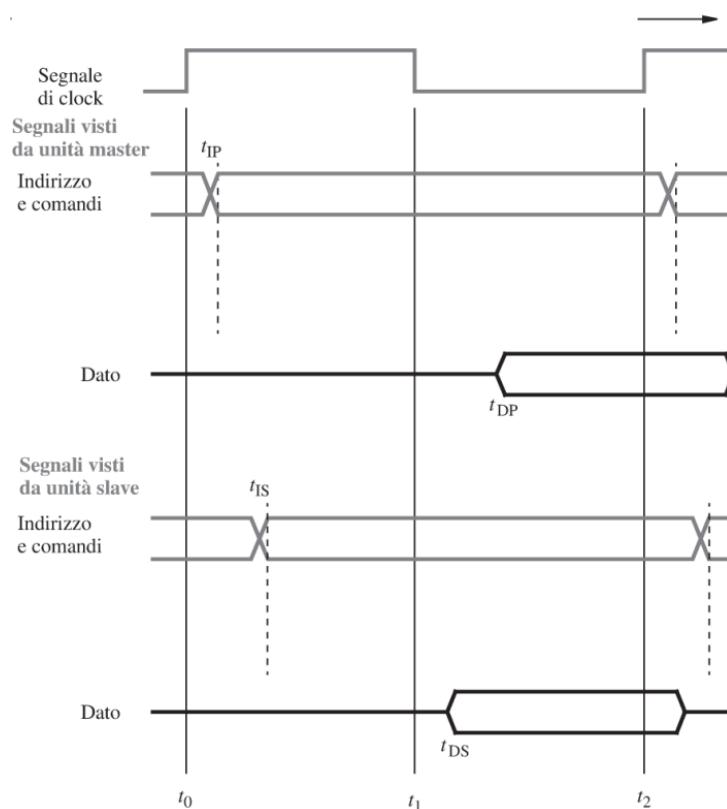
MASTER emette il segnale R/\overline{W}

e l'indirizzo un po' in ritardo rispetto al tempo di clock t_0 perché il clock non é visibile instantaneamente.

Visto che **SLAVE** é posta dopo **MASTER**, essa vedrá tale segnale ancora piú in ritardo del **MASTER**.

Il tempo che intercorre fra t_{IS} e t_{IP} é detto **ritardo di propagazione** e sta a indicare il tempo che serve a **SLAVE** per accorgersi che **MASTER** ha imposto un comando o un indirizzo.

SLAVE, quindi ha meno tempo per poter lavorare ed emettere i dati necessari.

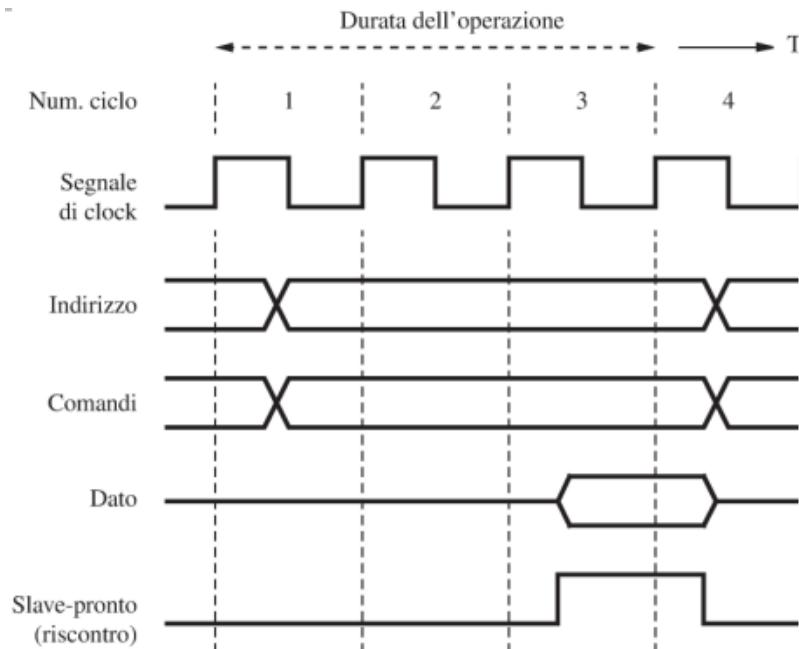


Per lo stesso motivo **MASTER** vedrá il dato un po' dopo rispetto a **SLAVE**.

Tale dato **rimarrá** sul bus di dati per un periodo **superiore a quello di clock** per permettere a **MASTER** di poterlo leggere correttamente ed evitare di prelevare dei dati **spuri** e tale periodo viene detto **tempo di hold**.

Trasferimento dati in piú cicli

Con il bus sincrono si puó avere anche una diversa temporizzazione fatta da piú **cicli di clock** con **periodo piú piccolo**. Il periodo di **clock** deve essere **sufficiente** affinché la periferica piú lenta tra le componenti abbia *il tempo di accorgersi che essa sarà SLAVE*. Di conseguenza il periodo di clock sarà sufficiente a tutte le componenti piú veloci.



Si introduce il segnale **Slave-Pronto** che viene emesso dall'unità **SLAVE**. Tale segnale sta a indicare che SLAVE é pronta e quindi scrive il dato sul bus dati. Quando **Slave-Pronto** é alto, **MASTER** capisce che **puó leggere** tale dato dal bus dati.

Così facendo é possibile gestire i ritardi delle varie periferiche visto che alcune sono piú veloci, quindi possono emettere il segnale Slave-Pronto nel ciclo di clock 2, e quelle che sono piú lente, quindi possono emettere il segnale Slave-Pronto nel ciclo di clock 3,4..ecc.

Si ha quindi un tempo di risposta di SLAVE **personalizzato** in base al tempo di risposta della periferica stessa.

Quindi non é necessario imporre un periodo di clock elevato proporzionale alla velocità della periferica piú lenta visto che quando si incontra una periferica veloce, é possibile diminuire i vari cicli di clock.

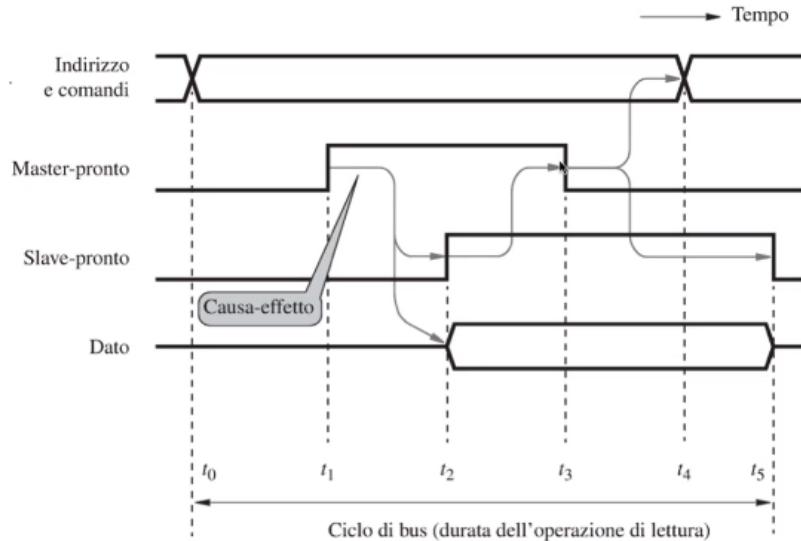
La periferica puó rispondere in un tempo maggiore rispetto ad un ciclo di clock ma **i vari cicli sono limitati**. MASTER, di fabbrica, ha un numero di cicli di clock dopo i quali dà per scontato che la periferica non risponde piú.

BUS ASINCRONO

Si introduce un nuovo segnale di controllo chiamato **Master-Pronto** che viene emesso dal **MASTER**.

La **temporizzazione** é fatta
senza bus clock ed é
ottenuta tramite dei **segnali**

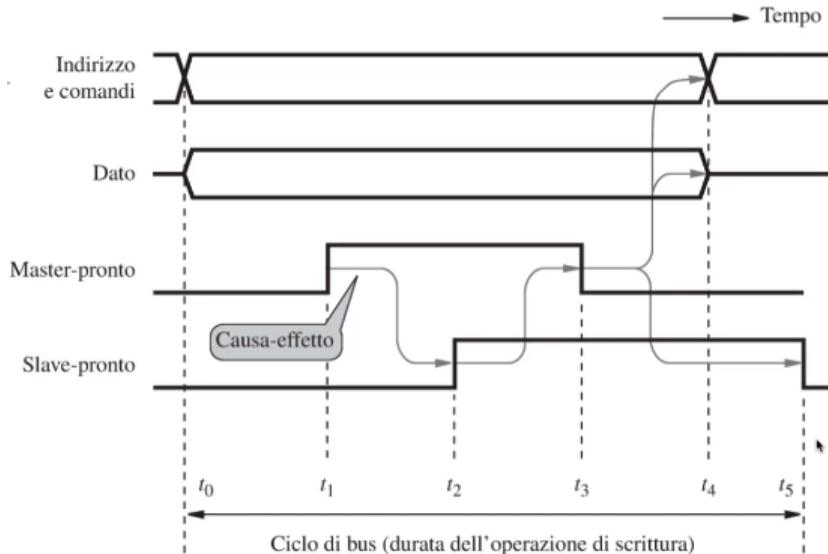
di conferme tra **Slave-Pronto** e **Master-Pronto**.
Il metodo sotto spiegato viene detto **HANDSHAKE** (stretta di mano) che é un metodo di tipo causa-effetto ed é valido per la fase di **LETTURA** del dato:



1. **MASTER** mette il segnale **Master-Pronto** alto per far capire alle periferiche che é stato immesso un indirizzo ed esse devono leggerlo.
2. La periferica interessata **SLAVE** si accorge che l'indirizzo é riferita a lei e mette alto **Slave-Pronto** e scrive sul bus dati.
3. Visto che **Slave-Pronto** é alto, **MASTER** si accorge e **Master-Pronto** va basso e quindi capisce che puó leggere il dato sul bus dei dati.
4. Visto che **Master-Pronto** é basso allora l'indirizzo sul bus degli indirizzi e il segnale **Slave-Pronto** va basso perché si é accorta che **Master-Pronto** é riuscito a leggere il dato dal bus di dati.
5. Ricomincia l'operazione

$t_1 - t_0$ é lo **sfasamento temporale** che vi é fra **MASTER** e **SLAVE** dovuto alle varie distanze fra le componenti.

Per quanto riguarda la **SCRITTURA** del dato sulla periferica, si ha un disegno del genere:



In questo caso il dato viene imposto dal **MASTER** quando immette l'indirizzo sul bus degli indirizzi insieme al comando R/\overline{W} che sarà uguale a 0. Il dato viene messo subito all'inizio perché chiaramente conosce già il dato che deve scrivere.

La periferica sa che deve leggere il dato dal bus di dati e appena si accorge che essa è **SLAVE** mette **Slave-Pronto** alto e quindi legge il dato entro il tempo t_4 , dove l'indirizzo e il dato vengonoolti dai bus rispettivi.

Confronto fra sincrono e asincrono

+Il bus **asincrono** è **più affidabile** perché lavora con i segnali **Slave-Pronto** e **Master-Pronto**. Se **Slave-Pronto** non va mai alto, *MASTER non leggerà mai un dato spurio dal bus dei dati.*

+Il bus **asincrono** è **più robusto** per quanto riguarda la **temporizzazione** perché se si incontra una periferica più lenta, il protocollo asincrono funziona automaticamente e *non si devono modificare i periodi di clock* visto che i segnali vengono aspettati.

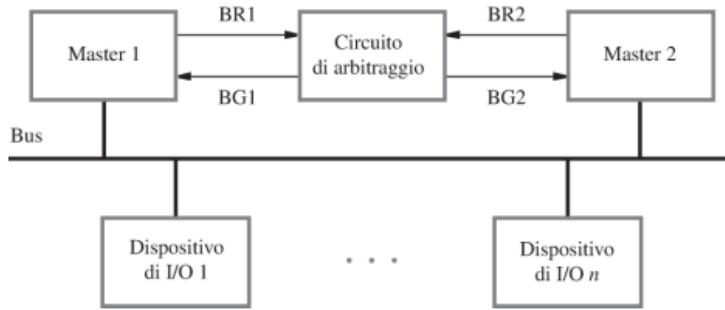
-Il protocollo **asincrono**, comunque, è **più lento** rispetto a quello sincrono perché vi sono i segnali **Master-Pronto** e **Slave-Pronto** che **devono viaggiare avanti e indietro 4 volte** tra **MASTER** e **SLAVE**. Quindi questo ulteriore tempo di propagazione va tenuto in considerazione.

Spesso, infatti, vengono utilizzati **bus sincroni** nei vari processori.

PILOTAGGIO BUS

In ogni momento *un solo dispositivo può essere abilitato all'invio del segnale* mentre tutti gli altri devono **attendere** il completamento di tale operazione. Per fare questa operazione vengono utilizzate le porte **THREE-STATE**.

ARBITRAGGIO DEL BUS



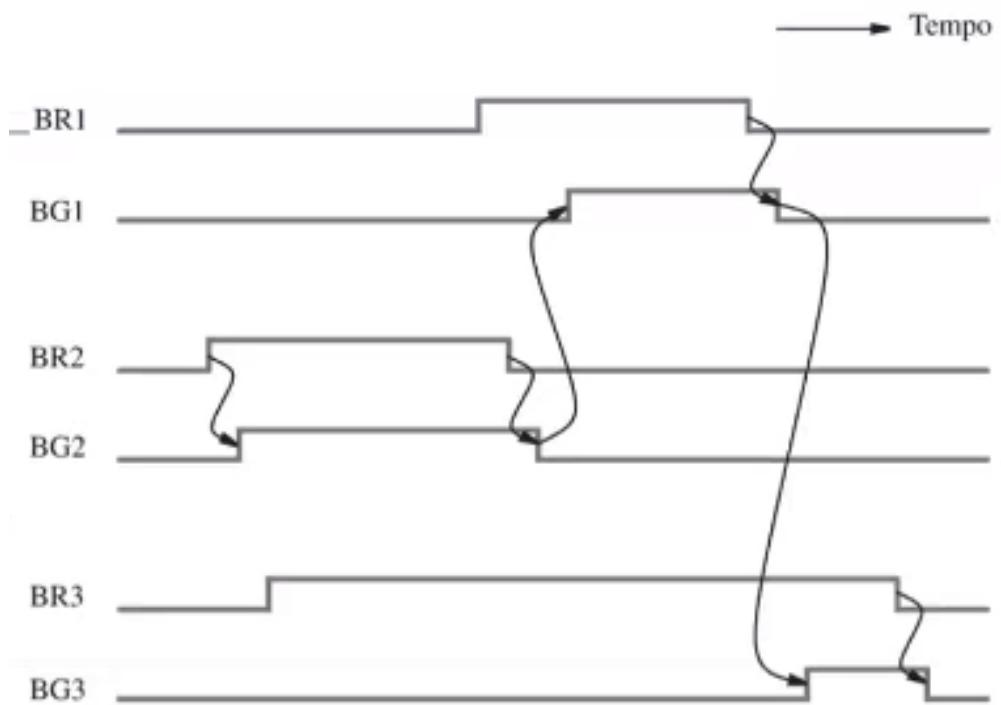
Più dispositivi collegati al bus possono avere il ruolo di **MASTER** (processore, controllori di I/O) quindi può essere necessario accedere allo stesso **SLAVE** allo stesso momento e ciò non è possibile.

Per decidere **quale MASTER deve prendere il controllo del bus si utilizza un CIRCUITO DI ARBITRAggIO**.

Il circuito di arbitraggio utilizza le linee in ingresso **BR1** e **BR2** che indicano la **richiesta** di utilizzo del bus da parte dei **MASTER** e le linee di uscita **BG1** e **BG2** che indicano le **concessioni** a quel **MASTER** di utilizzo del bus.

1. **BR1** e **BR2** vanno alte;
2. In base alla priorità, il circuito di arbitraggio permette a **un solo MASTER** di usare il bus e quindi manda, per esempio, **BG1** alto.
3. **MASTER 1** si accorge che ha il permesso di usare il bus e fa le dovute operazioni.
4. **MASTER 1** finisce le operazioni sul bus e mette **BR1** basso.
5. Circuito di controllo vede che **BR1** è andato basso e quindi concede l'utilizzo del bus a **MASTER 2** di usare il bus mettendo **BG2** alto.
6. Quando **MASTER 2** finisce le operazioni sul bus, mette **BR2** basso.

Ciò è valido anche per 3 **MASTER** e l'ordine di utilizzo del bus, se vi sono richieste di utilizzo del bus che avvengono allo stesso momento, è determinato dalle **priorità**.



SISTEMI DI MEMORIA

Introduzione

Il calcolatore ha al suo interno **diversi tipi di memoria**:

1. Memoria **RAM**;
2. Memoria **cache**;
3. **Dischi e/o periferiche di I/O**.

Ogni tipo di memoria ha il suo costo e le proprie caratteristiche. Tra le principali **caratteristiche** si ricordano la **latenza, capacità, costo, tempo di accesso** ecc..

Il **costo** dipende da **fattori di fabbricazione**, cioè più è complessa la fabbricazione di quella memoria, di conseguenza il costo aumenta.

La capacità di memoria sarà limitata dallo schema di indirizzamento.

Se si vuole mettere una RAM enorme, come per esempio 1 TB, essa sarebbe inutile perché il processore non riuscirebbe a gestirla completamente. Si devono fare conti fra quanti bit vengono gestiti dal processore e il numero massimo di indirizzi che si possono selezionare in memoria.

Il bus degli indirizzi solitamente ha un numero di bus minore al numero di bit che si impiegano per i registri e sarà uguale al numero di bit che la cpu utilizza per l'indirizzamento.

Concetti di base

La **velocità di una memoria** (principale solitamente) è **misurata tramite** l'inverso al tempo di accesso alla memoria. Si intende, cioè, il **tempo di risposta della memoria per emettere un dato richiesto dalla CPU**.

Il **tempo di ciclo di memoria** è il tempo minimo fra l'inizio di due operazioni di memoria consecutive.

RAM: Random Access Memory → qualunque sia la locazione della memoria che viene richiesta in lettura, tale dato viene fornito con tempo costante.

Memoria cache: piccola di dimensioni e più veloce della principale. Essa si inserisce in mezzo tra CPU e RAM.

Perché non si utilizza sempre la cache al posto della ram? Oppure perché non si utilizzano cache giganti? Perché la cache costa MOLTO di più della RAM.

All'interno del processore i registri sono pochi. In un programma si hanno tante variabili, per tale motivo il processore ha bisogno di riutilizzare gli stessi registri perché alcune variabili non entrano nei registri perché occupano molto spazio.

Lo scambio di dati fra processore e memoria è continuo.

Per migliorare le prestazioni vengono trasferiti **blocchi contigui di dati** che contengono decine, centinaia o migliaia di parole di memoria piuttosto che passare una singola parola di memoria perché, per esempio, un disco rigido è veramente lento e ha una latenza molto elevata.

Il tempo di accesso alla RAM è maggiore rispetto al tempo di accesso alla cache. Per lo stesso motivo vengono trasferiti grandi blocchi di dati alla volta dalla RAM alla cache per "**pagare la latenza**" meno volte possibile.

MEMORIA RAM A SEMICONDUTTORI

Il tempo di accesso alla RAM varia da 100 nanosecondi a 1 nanosecondo. I costi, chiaramente, saranno più elevati per le RAM con tempo di accesso minore.

All'interno della RAM sappiamo che **il dato viene mantenuto finché c'è l'alimentazione**, quindi la **RAM è volatile**.

Esistono due tipi di **RAM**: *statica* e *dinamica*.

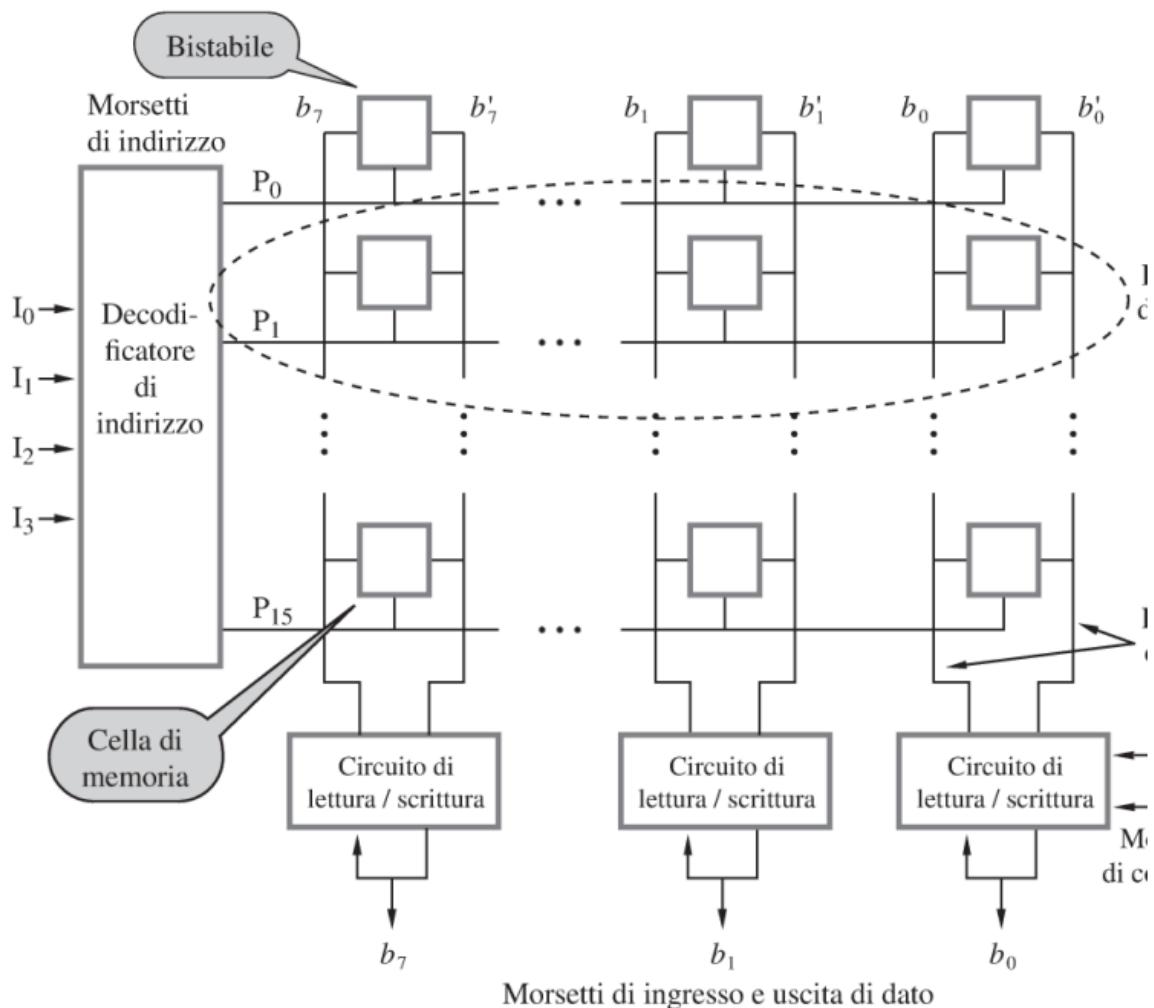
Nella **statica** il dato viene mantenuto **finché viene alimentata** da corrente. Non vi sono dissipazioni di corrente.

Nella **dinamica**, il dato contenuto al suo interno viene perso anche se vi è l'alimentazione se non viene **riscritto periodicamente (rinfrescato)**. Rinfrescare un dato vuol dire mettere dei circuiti aggiuntivi.

Rinfresco e **lettura** della memoria vengono fatti **separatamente** e quindi non possono venire eseguiti allo stesso tempo. Nella RAM dinamica (con rinfresco) il dato si perde perché ci sono delle dissipazioni di corrente dalla cella che memorizza il dato e tali perdite ci sono perché così è la propria tecnologia di realizzazione.

Perchè non si usa sempre memoria statica? Perché la statica costa di più rispetto alla memoria dinamica.

Organizzazione interna di chip di memoria



Il **chip di memoria** è fatto come una matrice di celle. Ogni cella è fatta da un **bistabile** di memoria che riesce a **memorizzare un singolo bit**, quindi o 1 o 0.

In questo esempio ci sono 8 celle in orizzontale, quindi 8 colonne(organizzazione a byte) che rappresentano 8 bit. La matrice contiene anche 16 righe e quindi ogni riga memorizza 8 bit.

Quindi si possono memorizzare, con un chip $16 \times 8 = 128$ bit.

Visto che 128 bit sono pochi, si avranno più chip o chip con matrici più grandi.

La lettura avviene **selezionando una delle righe** (quindi per leggere 8 bit). Per indirizzare una delle 16 righe, occorrono 4 linee per ottenere combinazioni di 16 righe a 2 a 2 perché $2^4 = 16$.

Se la linea R/\overline{W} é alta, vuol dire che si vuole eseguire una lettura, quindi dalla cella corrispondente esce il dato richiesto e facendolo arrivare, mediante il bus dati, al componente che ha richiesto tale dato.

Quante linee si devono collegare al chip?

- 4 **linee di indirizzo (4 righe)** se é una 16×8 per selezionare 16 byte;
- 8 **linee per i dati;(8 colonne/bit)**
- 1 **linee per R/\overline{W}** ;
- 1 linea **CS(Chip Selection)** che serve per capire **quale chip in questione deve essere selezionato**;
- 2 linee per la **corrente elettrica** (1 di massa e 1 per la fase).

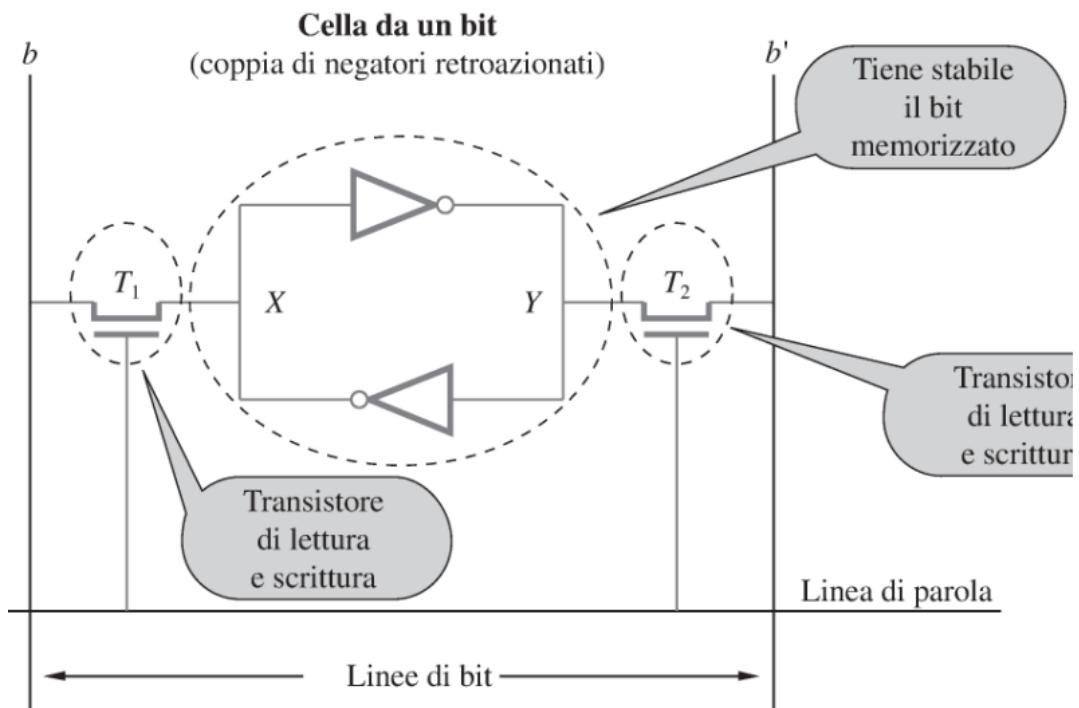
In totale servono 16 collegamenti per questo chip. Questi collegamenti sono detti **MORSETTI**.

Chi realizza il chip deve dire quante righe e colonne ci sono. Ci puó dire che é 1 kbit (1024 bit) e ci deve dire anche la matrice. per esempio ci sono parole di memoria di 8 bit. Per arrivare a 1024 servono 128 righe ($1024/8$). Il circuito deve poter selezionare una riga fra 128 possibili. Servono n linee dove 2^n é uguale 128 e questo n é 7 ($2^7 = 128$). Quindi servono 7 linee per avere 128 combinazioni.

Servono 3 linee in più di prima.

I **collegamenti** vengono contati perché devono essere **tenuti sotto controllo** perché si parla di **componenti molto piccole** e quindi é importante per avere una minima idea dei collegamenti che servono per un **chip minuscolo**. Di conseguenza le linee devono essere anche piccolissime.

Memoria statica cache



All'interno della singola della della matrice vi è il circuito sopra indicato.

Ogni cella ha diverse linee di indirizzo, indicate con linee con $P_0, P_1 \dots$

Sono dette **LINEE DI PAROLA** (linea orizzontale).

Ci sono le linee laterali verticali **b** e **b'** che sono le **LINEE DI DATI**.

All'interno della memoria statica ci sono **2 transistor** T_1 e T_2 collegati mediante **b** e **b'**. I transistor sono collegati ad un altro circuito formato da una **coppia di porte NOT**. T_1 e T_2 hanno un collegamento con la linea di parola al loro centro.

- Quando nella **linea di parola il valore è 0** (0 Volt) i transistor ricevono 0 nella loro linea di ingresso (fase o gate). Questo vuol dire che il transistor è in **interdizione** e si comporta come un **circuito aperto**. quindi non ce collegamento fra **b** e **x** e fra **b'** e **y**. Il valore della cella di memoria è memorizzato all'interno della coppia di negatori (**2 porte NOT**). Esempio: se **y** è 0, **x** è 1. Quindi vi è un **ciclo infinito** finché la coppia di negatori è **isolato** (T_1 e T_2 in interdizione).

Il valore memorizzato all'interno della coppia di negatori è dato dal valore di x, quindi 1 (in questo caso).

- Quando la **linea di parola ha valore alto** (quindi 1) si può leggere o scrivere all'interno della cella. in questo caso T_1 e T_2 sono in **saturazione**, ovvero si comportano come un **circuito chiuso**. **x** e **b** sono collegati e **b'** ed **y** sono collegati.
Il **valore** che era presente in **x** **esce in b** e questo è il valore del bit che era memorizzato all'interno della coppia di negatori.
b' sarà pari al valore negato rispetto a **b**. Se **x=1, b=1, b'=0**.

Il **circuito di lettura e scrittura** riceve la linea **b** e **b'** e fa uscire il valore **b** per la lettura. (R/\bar{W}) con valore **1** vuol dire che voglio leggere all interno della cella e viene preso il valore di **b'**. In caso di scrittura, con la linea di parola a 1 (valore alto), il circuito lettura e scrittura impone il

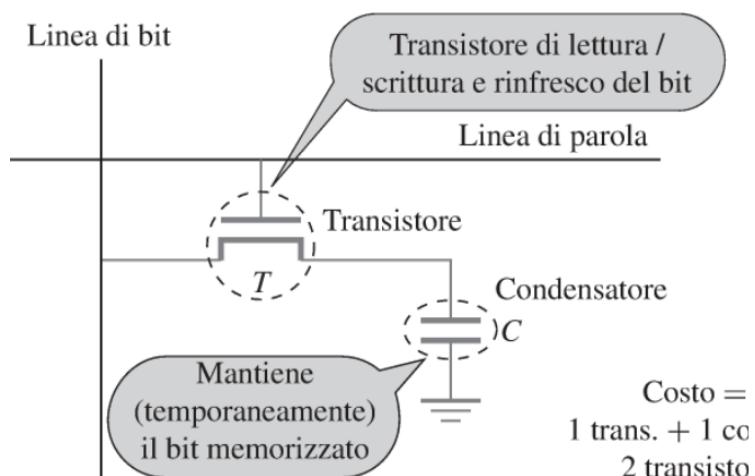
valore che si vuole scrivere, **proveniente dal basso**, sulla linea **b** che, mediante T_1 scrive tale valore sulla **x**. Dopo che si è scritto il valore sulla cella, si toglie la tensione alla linea di parola e il dato viene memorizzato all'interno della cella.

Come realizziamo la cella? Ciascuna **porta NOT** è realizzata con **2 transistor**. Visto che le porte NOT sono due, servono **6 transistor** in totale per realizzare la singola cella di memorizzazione nella memoria statica.

Il **costo di fabbricazione** con **6 transistor** è un pò **alto** rispetto alla memoria dinamica.

Questa memoria statica ha 1 **vantaggio**: quando si aprono circuiti con i transistor **non si ha una dissipazione** di corrente all'interno delle porte **NOT**. La carica delle porte **NOT** non viene persa. Il tempo di lettura o memorizzazione del dato è basso. *La memoria è sempre disponibile alla risposta e quindi non serve rinfrescare i dati come nelle memorie dinamiche.*

Memoria dinamica



Vi è un **transistor T** che fa la stessa azione del T_1 della statica. **T** è collegato alla linea di parola, linea di bit e ad un **condensatore**. Se la linea di parola è 0, il circuito che memorizza il bit è scollegato. Il valore del bit che si vuole memorizzare è tenuto dal condensatore (*temporaneamente*).

Quando si vuole memorizzare un valore, si seleziona la linea di parola, **T** è in **saturazione** e la linea di bit è collegata al condensatore. Per scrivere il valore sul condensatore, la linea di bit impone una certa tensione che il condensatore usa per caricarsi (se vi è una tensione alta) o per scaricarsi (se vi è una tensione bassa). Un condensatore, se caricato, riesce a mantenere un dato (per un **tempo limitato**).

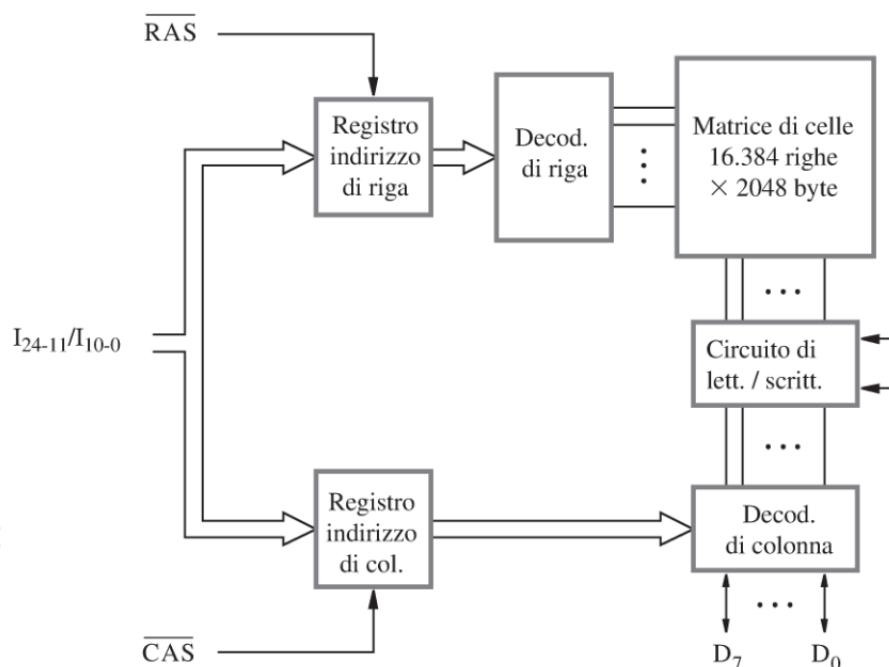
Quando la linea di parola vale 0, **T** scollega il condensatore dalla linea di bit e, per via di **dissipazione**, il condensatore **comincia a perdere la propria carica** (per un certo tempo limitato).

Si deve rileggere il valore del condensatore **prima che la sua carica diminuisca** e ripristinare al massimo il valore che quel condensatore può tenere.

In caso si voglia memorizzare il valore 0 in questa cella di memoria, bisogna imporre la linea di parola a 1. Di conseguenza **T** sarà collegato alla linea di bit e il valore 0 verrà memorizzato sul condensatore visto che **T** si sarà in **saturazione**. Anche in questo caso è **necessario il circuito di refresh** perché il condensatore potrebbe leggere qualche valore **spurio** diverso da 0 (tipo 0.1, 0.5) visto che la sua **carica è variabile**.

Per la realizzazione della memoria dinamica servono 1 transistor CMOS e un condensatore. Vi è una **diminuzione di componenti** rispetto alla statica. Il **costo è più basso** ma impone questo **"refresh"**.

Organizzazione RAM dinamiche



Il numero di collegamenti è stato conteggiato per cercare di diminuirlo il più possibile.

Per esempio si ha un chip che ha una capacità di 256 Mbit. Se la matrice è organizzata con una parola di 8 bit, allora ci sono 8 colonne. Quindi $256/8 = 32Mbit$, ovvero ci sono 32 Mega linee.

Non è agevole avere solo 8 colonne. Un'organizzazione potrebbe essere $16k \times 16k$ ($k=1024$). $16k = 2048 \cdot 8$ Posso avere una matrice $16k \times 16k$, ma se voglio selezionare il singolo byte, devo selezionare una riga fra le $16k$ righe e una colonna fra le 2048 righe. Quindi **serve un indirizzo per le righe e uno per le colonne**.

- Per selezionare le righe, servono **14 linee** di indirizzo, perché 2^{14} fa $16k$.

- Per le colonne servono **11 linee** per selezionare uno fra i possibili gruppi di 8 bit tra le 16k colonne perché 2^{11} fa 2048.

Servono, quindi $14+11=25$ linee per selezionare una parola di 8 bit tra 32M parole. infatti 2^{25} fa 32M.

Per ridurre il numero di linee per collegamento si possono ***inviare gli indirizzi in modo separato***.

Quando fornisco un indirizzo di riga, fornisco 14 linee, quindi vi è il **segnale di controllo** chiamato **RAS** (Row Address Strode, campionamento di indirizzo di riga) che serve per indirizzare la riga.

Quando si deve selezionare una colonna si mandano gli 11 bit rimanenti e si manda il **segnale di controllo CAS** per far capire al chip che si sta mandando un indirizzo di colonna. Il chip di memoria memorizza l'indirizzo di 11 bit e capisce che si seleziona la colonna.

Quindi gli **indirizzi vengono alternati sulle stesse linee** e questo permette di **ridurre il numero di linee** che servono per ogni chip.

Quando si riceve un indirizzo, questo ultimo va al *decodificatore* che decodifica il valore che è stato inviato come valore di indirizzo per riga o colonna, che seleziona la colonna o riga giusta.

Dopo di che, con CS e R/W si fanno le letture o scritture dal chip di memorizzazione.

RAM dinamiche asincrone

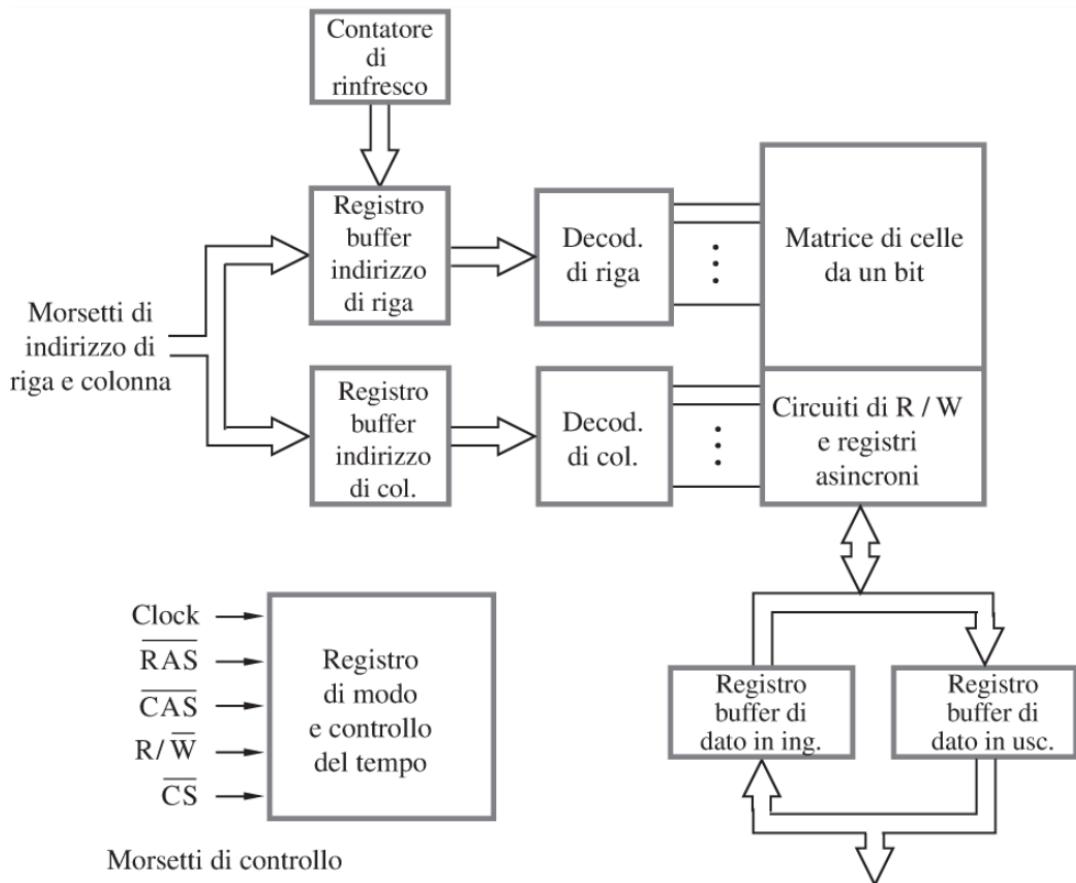
Se si vuole leggere un byte successivo, l'indirizzo di riga rimane uguale e cambia solo l'indirizzo di colonna, quindi viene aggiornata la linea **CAS**. In questo modo **la riga rimane costante** e si evita di inviare continuamente riga e colonna. Questo procedimento **velocizza i processi** e quindi si leggono varie parole di memoria nella stessa riga.

Si parla quindi di un **MODO VELOCE DI LETTURA DI PAGINA**.

Nella RAM dinamiche asincrone il **refresh** dei dati viene eseguito da un **chip esterno** a loro.

RAM dinamiche sincrone

Per indicare questo tipo di memoria RAM si utilizza la sigla **SDRAM**.



Il **refresh** dei dati (del condensatore) viene eseguito da **un circuito interno** e utilizzano **il segnale di clock della CPU**. Di fronte al processore, quindi, le **SDRAM appaiono come delle memorie statiche** e quindi non fa vedere dei tempi di attesa alla CPU ed è sempre disponibile visto che può ricevere al proprio interno più comandi come una sorta di PIPELINE.

Di conseguenza si ha bisogno di alcuni **registri buffer** per righe e colonne per eseguire tali operazioni.

Quindi vi è una sincronizzazione con il segnale di clock della CPU.

MEMORIE ROM

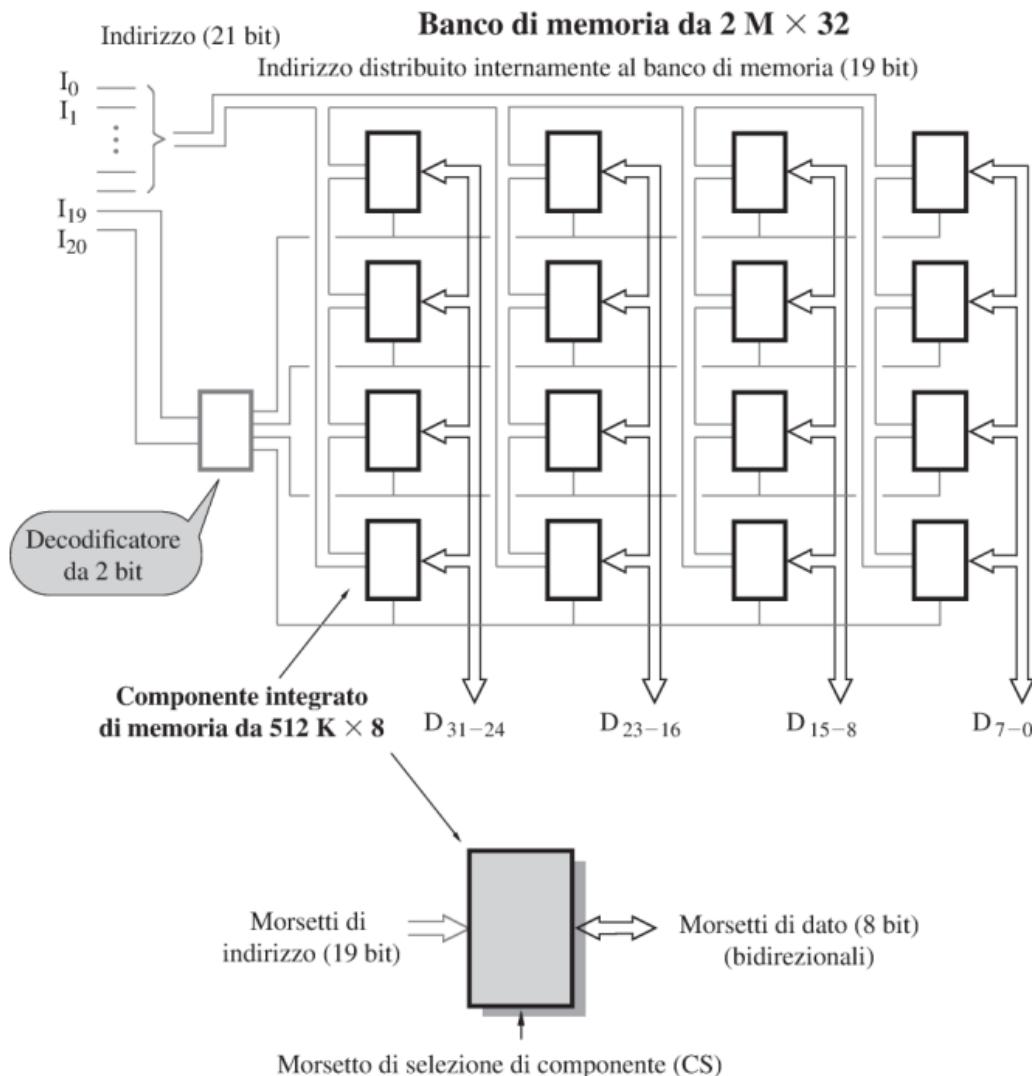
Sono delle memorie di **SOLA LETTURA** (*Read Only Memory*). Nella ROM solitamente si possono mettere delle istruzioni che servono per far leggere al calcolatore le **prime istruzioni di avvio** e comandi simili. Sono informazioni che serviranno per tutto il *ciclo vitale del calcolatore* e non dovranno essere **MAI** cambiate.

Una carta di credito, per esempio, ha un chip che entra in contatto con la macchina che riceve la carta. Il chip contiene un processore e una serie di memorie. Le memorie della carta sono ROM

MODULI DI MEMORIA

Un chip molto piccolo ($16 \times 8 = 128$ bit) può avere una capacità di memoria molto piccola. Se si mettono accanto 4 **chip uno accanto all'altro**, riesco ad avere una parola di memoria a 32 bit. Se ne metto altri sotto ottengo 16×4 righe ecc...

Una configurazione finale di un **banco di memoria** sarà tipo questa:



Ogni rettangolo è un chip di memoria con una certa capacità.

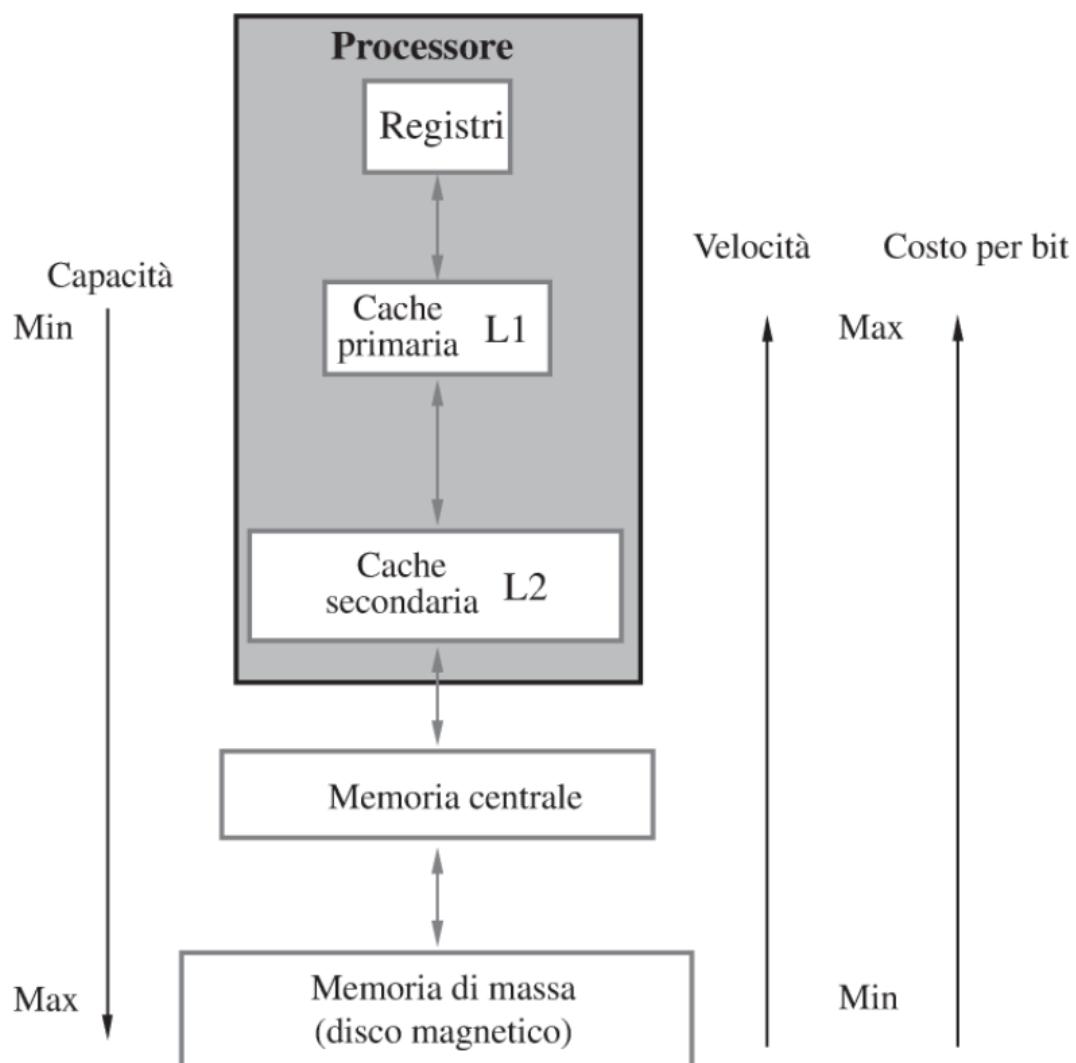
In questo esempio supponiamo che ogni chip è formato 512K x 8bit, quindi 512k righe e 8 colonne. Selezionando, quindi una riga, vengono selezionati i 4 chip e si otterrà in uscita un dato 32 bit. E' necessario, però, essere in grado di poter selezionare **una riga di ogni singolo chip**. Allora interviene il segnale **CS** (**Chip Select**) e serve per **selezionare un chip in tutto il banco di memoria**. Visto che ci sono 4 righe di chip, allora serviranno 4 bit per selezionare un chip

corrispondente, quindi 2 linee per fare 4 configurazioni. Si usano i bit **più significativi dell'indirizzo per determinare il segnale CS.**

Visto che si possono indirizzare 512K parole di memoria servono altri indirizzi per selezionare una parola all'interno del singolo chip da 512K. Uso 19 bit per selezionare una parola all'interno del chip e altri 2 bit per selezionare 1 riga su 4 (CS). In uscita avrò 32 bit di dati nelle linee D7-0, D15-8 ecc..

Se si vuole una capacità più ampia ma sempre a 32 bit, basta aggiungere delle righe e lasciare invariate il numero di colonne. Si potrebbero aggiungere 2,4,8,16 righe. E' buona norma aggiungere delle righe secondo le potenze di 2 così da sfruttare bene tutti i 32 bit in uscita.

GERARCHIA DI MEMORIA



All'interno di un calcolatore si utilizzano **diversi tipi di memoria** perchè dipende dalla capacità necessaria, budget ecc.

La memoria più lenta e meno costosa è la **memoria di massa**(disco). La memoria più veloce e più costosa è **quella dei registri**. Vi sono vari **livelli di cache L1 ed L2** perchè se il dato richiesto dal processore non viene trovato in **L1**, potrebbe essere cercato in **L2** piuttosto che andare subito alla memoria centrale. Quindi sotto questo punto di vista conviene avere più livelli di cache per aumentare la probabilità di cache hit.

I chip della cache sono installati sullo stesso silicio del processore.

Memoria cache e località

Il **principio di località temporale** determina che una **istruzione viene eseguita molte volte** all'interno dello stesso programma. Si può pensare all'apertura di un software che esegue le stesse medesime istruzioni più volte. Se si fa accesso ad una certa locazione di memoria è molto probabile che questa locazione verrà prelevata nuovamente.

Il **principio di località spaziale** determina l'alta probabilità di accedere a determinate **località di memoria VICINE** alle precedenti. Si stima, statisticamente, che se si accede ad una locazione di memoria (per esempio una posizione di un valore in un vettore) è **molto probabile che si accederà ANCHE** ad una locazione di memoria vicina alla precedente.

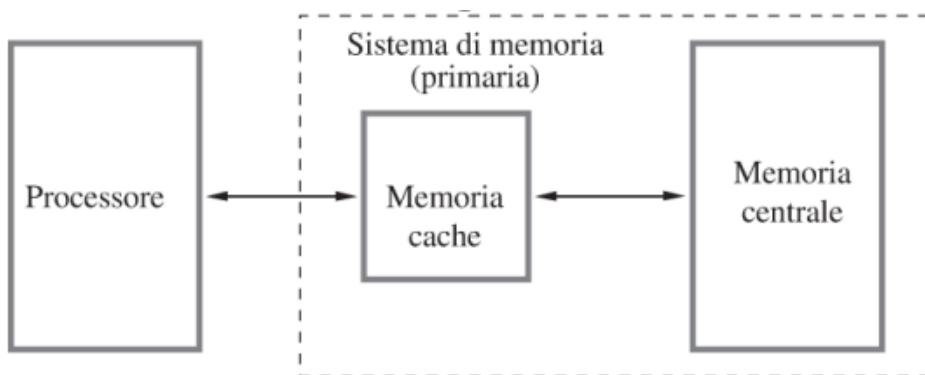
I due principi di località temporale e spaziale, seppur indipendenti, spesso vengono verificati contemporaneamente.

Un esempio è una scansione di un vettore mediante un ciclo for.

Quindi il principio di **LOCALITA' SPAZIO-TEMPORALE** afferma che:

- con alta probabilità, **entro breve tempo**, si accede alla stessa parola di memoria più volte o parole a breve distanza da essa saranno usate.
- **piccoli gruppi di parole di memoria sono usate più volte entro breve tempo.**

USO DELLA CACHE



Quando il processore deve leggere un dato dalla memoria emette un indirizzo della parola di memoria.

Anzichè trasferire un singolo indirizzo della parola di memoria, vengono **trasferiti dei blocchi di dati** che comprende quella parola di memoria interessata ma anche altre parole di memoria. Quando trasferisco tutto il blocco di dati, do la possibilità al processore di procedere con una velocità maggiore se mai dovesse chiedere **parole di memoria vicine alla precedente**. Il **tempo di trasferimento** di OGNI singola parola di memoria alla memoria centrale **viene pagato solo una volta** facendo viaggiare questo BLOCCO di parole di memoria.

La cache ci dà prestazioni maggiori perchè valgono i 2 principi di località spazio-temporale.

Il processore non conosce l'esistenza della cache quindi è indipendente dalla RAM.

La cache è organizzata a **posizioni di cache**. Ogni posizione ha un singolo blocco di parole di memoria che probabilmente verranno utilizzate nel breve futuro.

Capita, però, che tali **posizioni si riempiono** e quindi è necessario liberare delle posizioni di cache per permettere ai nuovi blocchi di essere memorizzati.

La liberazione della cache avviene mediante la tecnica **FIFO (First In First Out=il primo blocco che ho inserito sarà il primo a uscire)** oppure la tecnica **LRU(Least Recently Used=il blocco che ho usato meno recentemente verrà sostituito)**.

CACHE HIT

Cache HIT vuol dire che quello che è stato richiesto dal processore **si trova nella cache**.

In caso di **lettura** si parla di **CACHE READ HIT** se il dato si trova in cache. In questo caso la lettura viene eseguita senza coinvolgere direttamente la memoria centrale.

In caso di **scrittura** si parla di **CACHE WRITE HIT**. Il dato può essere scritto seguendo 2 modi:

- **SCRITTURA DIFFERITA (WRITE BACK)** o in tempi diversi. Il dato aggiornato viene scritto inizialmente sulla cache e si attiverà un **bit corrispettivo** che starà ad indicare che il **dato aggiornato si trova nella cache e non più nella RAM**. Quando la posizione di cache rispettiva al dato aggiornato verrà liberata, tale dato verrà poi aggiornato anche sulla memoria centrale.
- **SCRITTURA IMMEDIATA (WRITE THROUGH)** In questo caso il dato scritto sulla cache viene immediatamente scritto anche sulla memoria centrale.

Per evitare di incrementare lo stesso dato più volte sulla memoria centrale (tipo ciclo for) e quindi interagire con la memoria centrale perdendo tempo, si usa spesso la tecnica WRITE BACK affinchè si aggiorni il dato in cache.

CACHE MISS

Cache MISS vuol dire che il dato richiesto dal processore **non è all'interno della cache**.

In caso di **lettura (CACHE READ MISS)**, il dato non è presente in cache e quindi si deve leggere dalla RAM. Tale dato viene trasferito dalla RAM alla cache e dalla cache alla CPU.

La lettura può avvenire in due modi:

- Trasferimento parola di memoria RAM → CACHE → CPU. Allora si parla di **LETTURA IMMEDIATA(READ BACK)** e viene letto solo il dato che è richiesto.
- Trasferimento blocco di parole di memoria RAM → CACHE. Trasferimento parola di memoria interessata CACHE → CPU. Allora si parla di **LETTURA DIFFERITA(LOAD THROUGH)**. In questo secondo caso la CPU dovrà attendere un tempo leggermente maggiore affinchè venga **trasferito l'intero blocco di parole di memoria**.

In caso di **scrittura** emette l'indirizzo della parola di memoria e il dato corrispondente. Con cache miss si hanno 2 situazioni:

- Il dato da scrivere viene scritto **direttamente in RAM bypassando la cache**. Allora si ha la **SCRITTURA IMMEDIATA**;
- Il blocco di parole di memoria viene trasferito **RAM → CACHE**. Il dato viene scritto in cache. Tale dato viene trasferito **CACHE → RAM**. Allora si ha la **SCRITTURA DIFFERITA**. Tale blocco viene trasferito per sfruttare il principio di località spazio-temporale visto che altre parole di memoria vicine potrebbero servire.

INDIRIZZAMENTO DIRETTO

Ogni **posizione cache** ha la dimensione esatta o appena sufficiente per **contenere un blocco** proveniente dalla memoria centrale.

Quando questi blocchi si spostano dalla RAM alla CACHE è necessario conoscere dove devono essere posizionati tali blocchi.

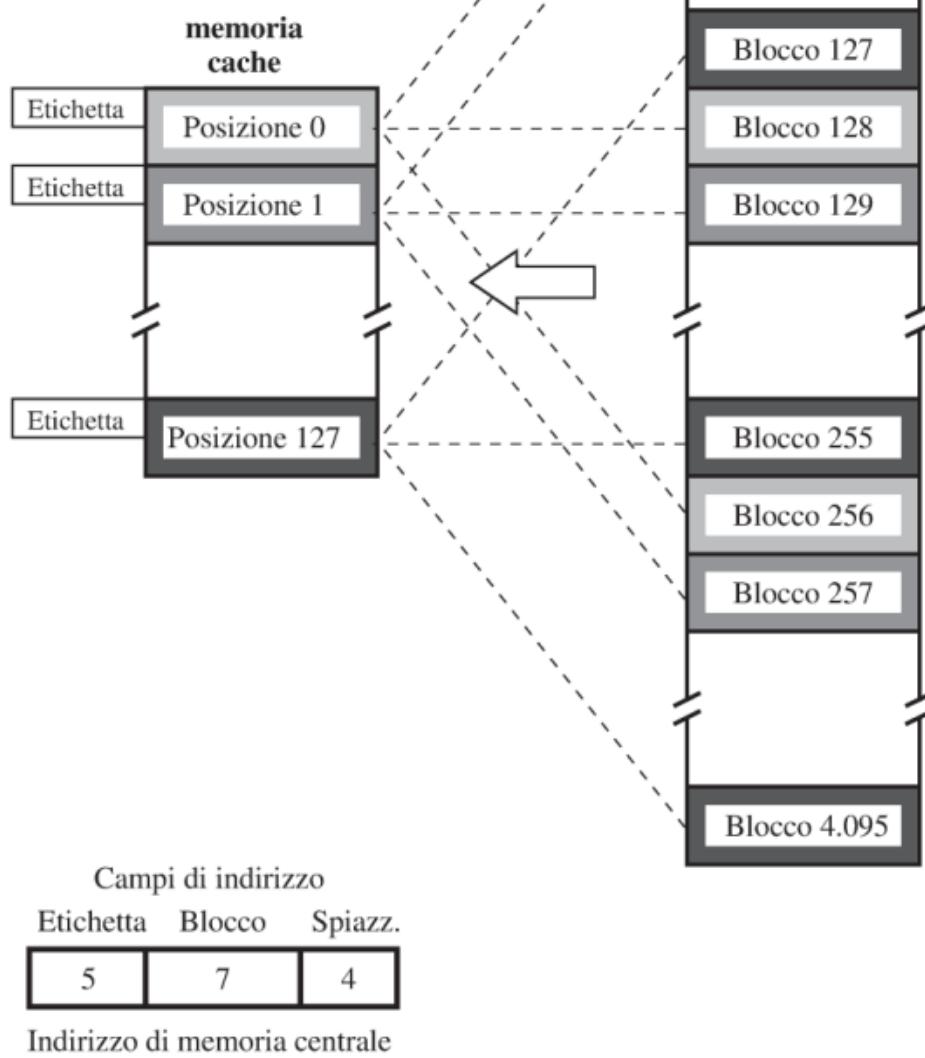
Si nota, dal grafico, che nella posizione 0 potrebbero venir inseriti uno dei blocchi 0, 128 o 256. Quindi vi è un salto di 128 blocchi.

E' quindi necessario sapere quale blocco specifico è presente nella posizione cache. Per vedere se il dato letto nella cache è effettivamente quello che ci interessa, bisogna vedere se quel determinato **blocco presente in cache è presente anche in RAM**.

Ogni **posizione cache** ha un **campo etichetta** che è associato ad ogni blocco.

Legenda:

$n = \#$ bit di ind. di mem. centrale
 $m = \#$ bit di ind. di mem. cache
 $b = \text{dim. in parole del blocco}$
 $\# \text{ blocchi di memoria} = \lceil 2^n / b \rceil$
 $\# \text{ posizioni di cache} = \lceil 2^m / b \rceil$
 Spiazzamento = $\lceil \log_2 b \rceil$
 Blocco = $\lceil \log_2 (\# \text{ posizioni}) \rceil$
 Etichetta = $n - \text{Spiazz.} - \text{Blocco}$



Quando la CPU fa un'operazione di lettura (per esempio) emette anche un indirizzo di **16 bit della parola di memoria alla quale vuole accedere** visto che non sa dell'esistenza della cache.

Questi 16 bit vengono confrontati con i dati nella cache per vedere se si ha una **HIT** o una **MISS**.

I 16 bit emessi dalla CPU si possono dividere in 3 parti, detti anche **campi dell'indirizzo**:

- i 4 bit più a destra sono i meno significativi e vengono detti campo **SPIAZZAMENTO**: questi bit permettono di **trovare la parola all'interno del blocco**;

- i 7 bit centrali formano il **BLOCCO**: essi permettono di **trovare un blocco** all'interno della memoria cache. ($2^7 = 128$ che rappresentano proprio il salto fra un blocco e un altro in cache). Per capire quale sarà la **posizione del blocco** si fa l'operazione
Blocco%nPosizioni

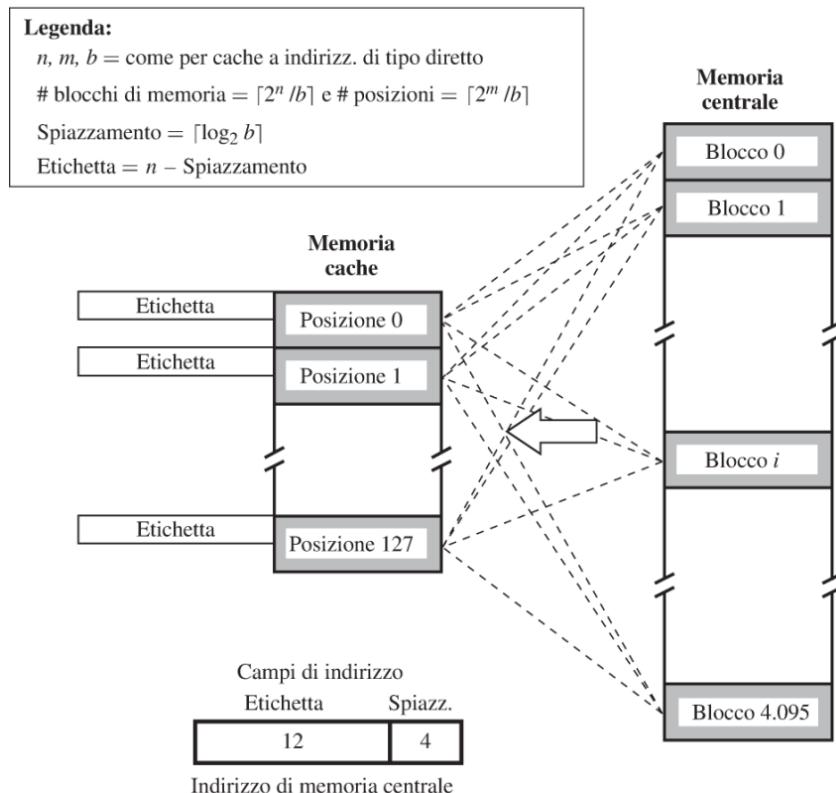
- i 5 bit finali formano il campo **ETICHETTA**: questo campo viene confrontato con l'etichetta presente nella memoria cache

Se vi è uguaglianza fra etichetta dell'indirizzo emesso dalla CPU e un'etichetta presente nella cache, allora tale dato corrisponde al dato richiesto. Si usa il valore spiazzamento per leggere all'interno del blocco la parola che la CPU voleva leggere. In caso di nessuna corrispondenza, vuol dire che la cache ha un **blocco differente** da quello che serve alla CPU e quindi bisogna trasferire il blocco di interesse dalla RAM alla CACHE.

Quindi **con il solo confronto delle due etichette** è possibile **verificare** se si è in presenza di un **cache hit** o una **cache miss**. E questo rappresenta **l'unico vantaggio** nell'utilizzare questo tipo di indirizzamento.

In questo tipo di indirizzamento si è costretti a mettere nelle stesse posizioni cache i vari blocchi della RAM.

INDIRIZZAMENTO ASSOCIATIVO



Nell'indirizzamento diretto è possibile che la CPU potrebbe emettere indirizzi di parole di memoria che corrispondono al blocco 0 e al blocco 128. Può sorgere il problema che questi due blocchi **collidono nella stessa posizione cache**. In questo caso la memoria cache non ci è

d'aiuto perchè, visto che si tratterà sempre di una cache miss, si dovrà sempre trasferire il blocco che ci interessa dalla RAM alla CACHE.

Le posizioni diverse dalla posizione dei blocchi di interesse potrebbero essere comunque inutilizzate e quindi non lette.

Con l'**indirizzamento associativo** il blocco della RAM può essere messo in **qualsiasi posizione** della CACHE.

Quindi se si hanno **posti liberi** nella memoria cache, questi possono essere **riempiti da un qualsiasi altro blocco** presente in RAM.

In caso di una lettura da parte della CPU, viene emesso un indirizzo della parola di memoria che vuole leggere e questo indirizzo ha il campo spiazzamento che serve per individuare una parola all'interno del blocco. Il campo etichetta dell'indirizzo prende 12 bit e viene confrontato con le varie posizioni cache. Visto che non si sa dove potrebbe trovarsi il blocco in cache che ci interessa, è dovuto **confrontare tutte le etichette della cache con l'etichetta dell'indirizzo emesso dalla CPU** visto che il blocco si può trovare in una qualunque posizione nella cache. E questa è la più grande **differenza** fra indirizzamento associativo o diretto.

Questo rappresenta un lungo procedimento, soprattutto se il blocco si trova all'ultima posizione cache disponibile. Per migliorare la situazione e quindi sfruttare le potenzialità della memoria cache, si dovrebbe eseguire tale **confronto fra etichette in parallelo** (di indirizzo e di cache) e non in maniera sequenziale. In tal caso si parla di **RICERCA ASSOCIATIVA**.

Tale metodo non è detto che è applicabile dalla memoria cache perchè le etichette da confrontare allo stesso momento in parallelo sarebbero veramente tante. E se tale procedimento è lento non si sfrutta le potenzialità della cache stessa.

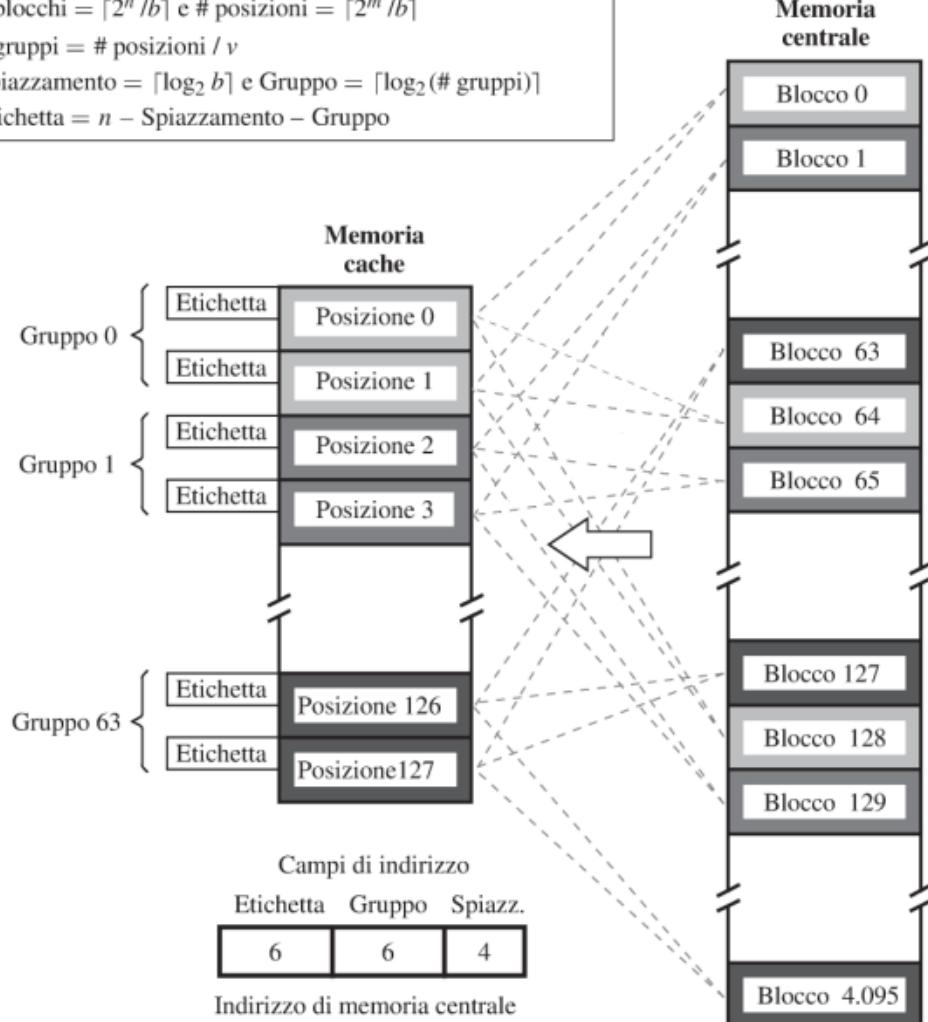
In tal caso si potrebbero diminuire i confronti e ciò è possibile mediante una sorta di 'misto' fra le due soluzioni proposte. Si parla di indirizzamento associativo a gruppi.

INDIRIZZAMENTO ASSOCIAТИVO A GRUPPI

Questo metodo ha i vantaggi dell'indirizzamento associativo ma richiede meno prestazioni del confronto alla memoria cache. Esso consente a un **blocco della RAM** di essere inserito **all'interno di un gruppo** della cache e ciascun gruppo della CACHE è dato da un **certo numero di posizioni**. Il blocco delle parole di memoria della RAM che devono essere trasferite all'interno della CACHE sono vincolate dal fatto che si devono sempre trovare in un gruppo della cache già definito ma ogni gruppo contiene più posizioni.

Legenda:

n, m, b = come per cache a indirizz. di tipo diretto
 v = dim. in posizioni del gruppo = # vie della cache
blocchi = $\lceil 2^n / b \rceil$ e # posizioni = $\lceil 2^m / b \rceil$
gruppi = # posizioni / v
Spiazzamento = $\lceil \log_2 b \rceil$ e Gruppo = $\lceil \log_2(\# \text{gruppi}) \rceil$
Etichetta = $n - \text{Spiazzamento} - \text{Gruppo}$



Quindi si hanno più 'slot' disponibili di posizionamento dei blocchi all'interno dello stesso gruppo.

In questo caso si riduce di molto il conflitto precedentemente accennato.

L'indirizzo della parola di memoria emesso dalla CPU è formato da 3 parti:

- 4 bit meno significativi per lo **spiazzamento**, danno la **posizione** della parola **all'interno del blocco**;
- 6 bit centrali che distinguono il **gruppo**;
- 6 bit più significativi che distinguono l'**etichetta**.

Per capire se il blocco di parole di memoria che ci interessa è presente in cache, si fa il solito **confronto** mediante l'**etichetta del blocco** presente nella cache e l'**etichetta** emessa dall'**indirizzo della parola di memoria** proveniente dal processore. Questo confronto, in questo

caso, viene eseguito, in parallelo, solo tra le etichette che sono presenti in un determinato gruppo e non fra tutte le etichette che contiene la cache.

Si dice che la cache è a n vie, dove n indica le **posizioni all'interno di un gruppo**. Nella figura sopra mostrata si parla di una cache a 2 vie.

VANTAGGI:

- **conflitti ridotti** e **confronti necessari** ridotti;
- metodo più utilizzato al momento.

DATI SCADUTI IN CACHE (DMA)

Per utilizzare correttamente la memoria cache bisogna valutare la coerenza.

Si introduce l'**unità funzionale DMA** (*Directly Memory Access*) ed essa è capace di leggere/scrivere in RAM. Trasferisce i dati dalla **periferica** (spesso dall'unità di massa) alla **RAM** o viceversa per **evitare del lavoro al processore** stesso. **DMA** è in grado di farlo perché comunque si tratta di indirizzi sequenziali e/o istruzioni ripetitive.

Il processore potrebbe aggiornare in cache un determinato numero di dati e la DMA potrebbe aggiornare gli **stessi** dati in memoria.

Quindi quali dati sono considerati ‘veritieri’?

Quando **DMA** esegue **trasferimenti fra disco e RAM**, le corrispondenti copie in cache devono essere contrassegnate come **invalidi** (o **scadute**) perché ci sono dati più recenti letti dal disco.

La ‘**validità**’ di un dato viene rappresentata da **un bit a sé**. Quindi se al processore serve un dato precedentemente usato dalla DMA, esso deve ri-trasferire tale dato dalla RAM alla CACHE.

In caso di **scrittura differita**, la DMA dovrebbe usare i dati più recenti non ancora presenti nella RAM. Per tale motivo i dati vengono scritti dalla CACHE alla RAM effettuando così lo svuotamento della cache (**cache flush**).

Il cache flush viene effettuato dal sistema operativo e non comporta grandi rallentamenti.

ALGORITMO DI SOSTITUZIONE

Quando la cache è piena bisogna **sostituire le vecchie posizioni** con nuovi blocchi provenienti dalla RAM.

In caso di **indirizzamento diretto** non si applica nulla di particolare visto che la posizione da trascrivere è sempre la stessa.

In caso di **indirizzamento associativo** e **indirizzamento associativo a gruppi**, il blocco di parole di memoria provenienti dalla RAM si può allocare in una o più posizioni.

Solitamente viene usato l'algoritmo **LRU** (*Least Recent Used*) che sostituisce il **blocco meno recentemente usato**. Per capire che il blocco è stato usato si usano degli opportuni **bit**.

Se si ha una cache a 4 vie (4 posizioni per gruppi di blocchi) si userà un contatore da due bit:

- in caso di **cache hit**, il contatore di tale posizione va messa a 0 e tutti i rimanenti vengono incrementati di 1;
- in caso di **cache miss** e posti ancora liberi, il nuovo blocco viene caricato in cache, tale bit va messo a 0 nella posizione corrispondente e i rimanenti vengono incrementati di 1;
- in caso di **cache miss e gruppo pieno**, la posizione con contatore massimo viene liberata e riempita col nuovo blocco. Il contatore corrispondente va messo a 0 e gli altri contatori vengono incrementati di 1.

| *Ciascun gruppo di 4 contatori sono sempre diversi fra loro*

CONSIDERAZIONI DI PRESTAZIONE

La gerarchia di memoria permette di minimizzare i tempi necessari al processore che usa per svolgere delle operazioni. La presenza di una cache permette di eseguire operazioni con un **tempo più vicino al periodo di clock**.

Un indicatore di efficacia per la cache è il **TASSO DI SUCCESSO**, o **Hit Rate**, h . Esso viene determinato sulla base degli accessi con *hit* e il totale numero di accessi alla cache.

Il **TASSO DI INSUCCESSO**, o **Miss Rate**, è dato da $1 - h$.

| *Ovviamente le prestazioni degradano nel caso di una cache miss.*

In caso di cache miss, si deve usare la RAM per reperire il dato o scriverlo. Il **tempo necessario per accedere al dato** che si trova in cache viene indicato con **C**. Il **tempo totale di accesso quando si ha una miss è detto penalità di miss**, M (cioè il tempo che ci vuole per caricare un blocco da RAM a CACHE).

Il tempo medio di accesso alla memoria è dato $t_{avg} = hC + (1 - h)M$

Schema riepilogato:

t_{avg} = tempo medio di accesso alla memoria;

h = tasso di cache hit $\rightarrow 1 - h$ = tasso di cache miss

C = tempo medio di accesso alla cache;

M = penalità di miss (tempo totale si impiega per accedere alla memoria);

ESEMPIO STIMA DEL GUADAGNO

- ▶ Si abbia: tempo di accesso τ alla cache, e 10τ alla memoria centrale
- ▶ Si abbiano 8 parole per blocco, allora il tempo totale di trasferimento del blocco da memoria a cache è $(10 + 7)\tau = 17\tau$, ovvero occorrono 10τ per la prima parola e 1τ per ogni parola successiva
- ▶ In una cache miss si hanno due accessi alla cache, quindi la penalità di miss è

$$M = \tau + 10\tau + 7\tau + \tau = 19\tau$$
- ▶ Si supponga che ci siano 100 istruzioni e che il 30% di esse acceda alla memoria
- ▶ Si abbia un **tasso di hit** per le **istruzioni** pari a 0,95, e un **tasso di hit** per i **dati** pari a 0,9
 - ▶ Il tempo di accesso senza cache è $(100 + 30) \cdot 10\tau = 1300\tau$
 - ▶ Il tempo di accesso con cache è

$$100 \cdot (0,95\tau + 0,05 \cdot 19\tau) + 30 \cdot (0,9\tau + 0,1 \cdot 19\tau) = 190\tau + 84\tau = 274\tau$$
- ▶ guadagno = tempo senza cache / tempo con cache = 4,7

MIGLIORAMENTO DELLE PRESTAZIONI

Per migliorare il tasso di successo, si può:

- **aumentare la dimensione della cache.** In questo caso bisogna tener conto del costo che va ad aumentare;
- **aumentare la dimensione dei blocchi** per aver migliori effetti dati dalla località spaziale (però diminuirebbe le hit di successo);
- diminuire la penalità di miss usando la **lettura diretta** (lettura diretta in RAM bypassando la cache) in caso di miss.

| La dimensione migliore della cache è fra 16 e 128 byte.

PRESTAZIONI PER DUE LIVELLI DI CACHE

In caso di **due livelli di cache L1 ed L2** bisogna contare il tempo di accesso anche alla seconda cache, quindi semplicemente, bisogna aggiungere il termine $h_2 C_2 + (1 - h_2)$ alla formula principale e poi moltiplicare tutto per M .

Complessivamente si avrà $t_{avg} = h_1 C_1 + (1 - h_1)(h_2 C_2 + (1 - h_2) \cdot M)$.

- Se $(1 - h_1)(1 - h_2)$ è molto piccolo allora è tollerabile un'alta penalità di miss, cioè M ;
- se $h_1 = h_2 = 0.9$ allora $(1 - h_1)(1 - h_2) = 0.1 \cdot 0.1 = 0.01$ quindi l'incidenza del tempo di accesso alla memoria centrale è ridotta all'1% dei casi.

*| Spesso si preferisce installare **due livelli di cache** (di cui la seconda maggiore della prima) a **parità di spese** visto che il tempo di accesso*

medio alla cache a due livelli è minore rispetto al tempo di accesso medio ad una singola cache a parità di spese.

Esempio:

Si abbia: $C_1 = t$ (il tempo di accesso per le due cache L1), inoltre per trasferire un blocco da L2 a L1 occorre un tempo $C_2 = 15t$ e per trasferire un blocco da Memoria a L2 occorre un tempo $M = 100t$

Si assuma che i tassi di hit siano uguali per istruzioni e dati e che per L1 e L2 siano $h_1 = 0,96$ e $h_2 = 0,80$ rispettivamente

Il tempo medio di accesso visto dal processore è

$$t_{avg} = h_1 C_1 + (1 - h_1)(h_2 C_2 + (1 - h_2)M)$$

$$\text{Quindi } t_{avg} = 0,96t + 0,04(0,80 \cdot 15t + 0,20 \cdot 100t) = 2,24t$$

Si supponga ora che L2 sia rimossa e che L1 sia più grande in modo da dimezzare il tasso di miss. Il tempo medio di accesso alla memoria è

$$t_{avg} = 0,98t + 0,02 \cdot 100t = 2,98t$$