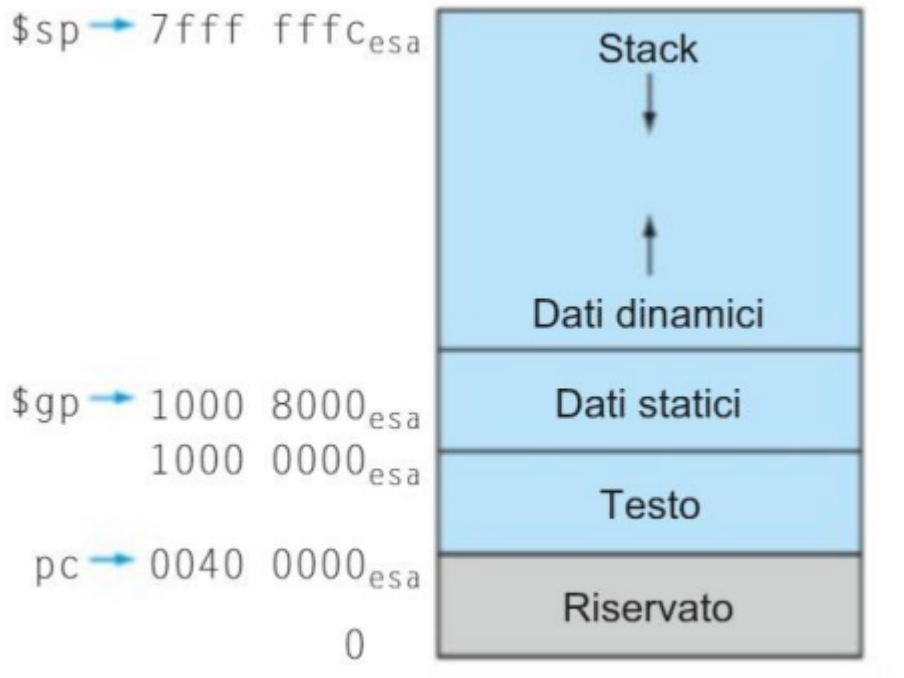
Architettura 2

Ogni processore ha il suo linguaggio macchina, ogni istruzione è definita in binario. Per arrivare dai bit ai linguaggi che usiamo come Java, C o Python serve colmare il "**semantic gap**".

Usiamo la così detta "macchina a strati" proprio perché ogni livello nasconde i dettagli del livello sottostante. Quindi ogni macchina M ha un linguaggio L. Un linguaggio di livello L_i **non deve differire troppo** da L_{i-1}.

Il programma in linguaggio L_i dovrà essere trasformato in istruzioni di linguaggio L_{i-1}.

Ogni istruzione di livello L_i è una sequenza di istruzioni L_{i-1} proprio perché astrarre un livello serve per rendere più facile la programmazione.



Ci sono due modi di realizzare una macchina virtuale: scrivere **un interprete** o scrivere **un traduttore**.

Nel primo caso di preleva una istruzione alla volta e si esegue, nel secondo caso di prende l'intero programma e si esegue solo dopo.

Livelli convenzionali

1. **Microarchitettura:** funzioni svolte dai circuiti, le componenti hardware.
2. **Linguaggio macchina:** istruzioni di base del sistema operativo (**ISA**).
3. **Nucleo sistema operativo:** utilizzo delle system calls.
4. **Assembler e librerie:** codifica simbolica per sviluppare programmi come drivers o librerie per linguaggi ad alto livello.
5. **Linguaggio alto livello:** macchina virtuale usata dai programmatori per eseguire e creare programmi.

I principali vantaggi di questo metodo sono: **astrazione** e **indipendenza**.

IJVM

Adottiamo questo linguaggio come livello 2 della nostra architettura, le istruzioni quindi non si possono eseguire direttamente sul livello 1 con istruzioni MAL.

La macchina virtuale che esegue IJVM è una macchina basata sullo stack. Questa struttura usa due operazioni: **pop** e **push**, il primo prende l'ultimo elemento inserito, ovvero il primo, il push invece inserisce sulla cima un nuovo elemento. Lo stack è una coda **LIFO** (Last In First Out).

La grande comodità dei linguaggi di programmazione ad alto livello è di poter **creare dei metodi** che possono essere chiamati in punti diversi del programma.

Le loro variabili interne si mettono in memoria e scompaiono una volta terminata la procedura.

es.

```
.main
CALL A;
```

In questo caso si chiama la funzione A che è definita in questo modo:

```
.A
VAR a1, a2, a3;
CALL B; CALL D;
RETURN;
```

Stack
a3
a2
a1

A questo punto si stanziano le tre variabili e si chiamano le funzioni B e D, ma la prima è B.

```
.B
VAR b1, b2, b3, b4;
CALL C;
RETURN;
```

Le variabili di B sono allocate in memoria, come possiamo osservare le variabili sono posizionate in modo che vengano risolte in modo inverso.

Stack
b4
b3
b2
b1
a3
a2
a1

Operazioni aritmetiche

L'operazione a1 = a2 + a3 è vista in questo modo: si caricano i valori nella memoria e poi si inserisce l'operazione.

Esempio pratico di una funzione ricorsiva con valori 1 e 3 in input:

```
int moltiplica(int n, int m) {
    int result;
    if(n == 0) {
        result = 0;
    } else {
        reult += moltiplica(n - 1, m);
    }
```

```
    return result;
}
```

Per prima cosa sono inseriti nello stack la variabili m, n e result con i loro valori rispettivi 1 3 e NULL.

Il primo record di attivazione moltiplica(0, 3) mette in memoria 0 e 3 e avremo una situazione del genere:

Address	Name	Value
SP →		0
	result	0
	m	3
LV →	n	0
	result	3
	m	3
	n	1

Il record di attivazione è l'insieme delle variabili della funzione chiamanti.

Quando vi è il record di attivazione **si salva anche la funzione da svolgere alla base del record (LV)** proprio perché le operazioni necessitano di operandi per essere svolte.

Memorie di IJVM

La memoria è suddivisa in quattro aree:

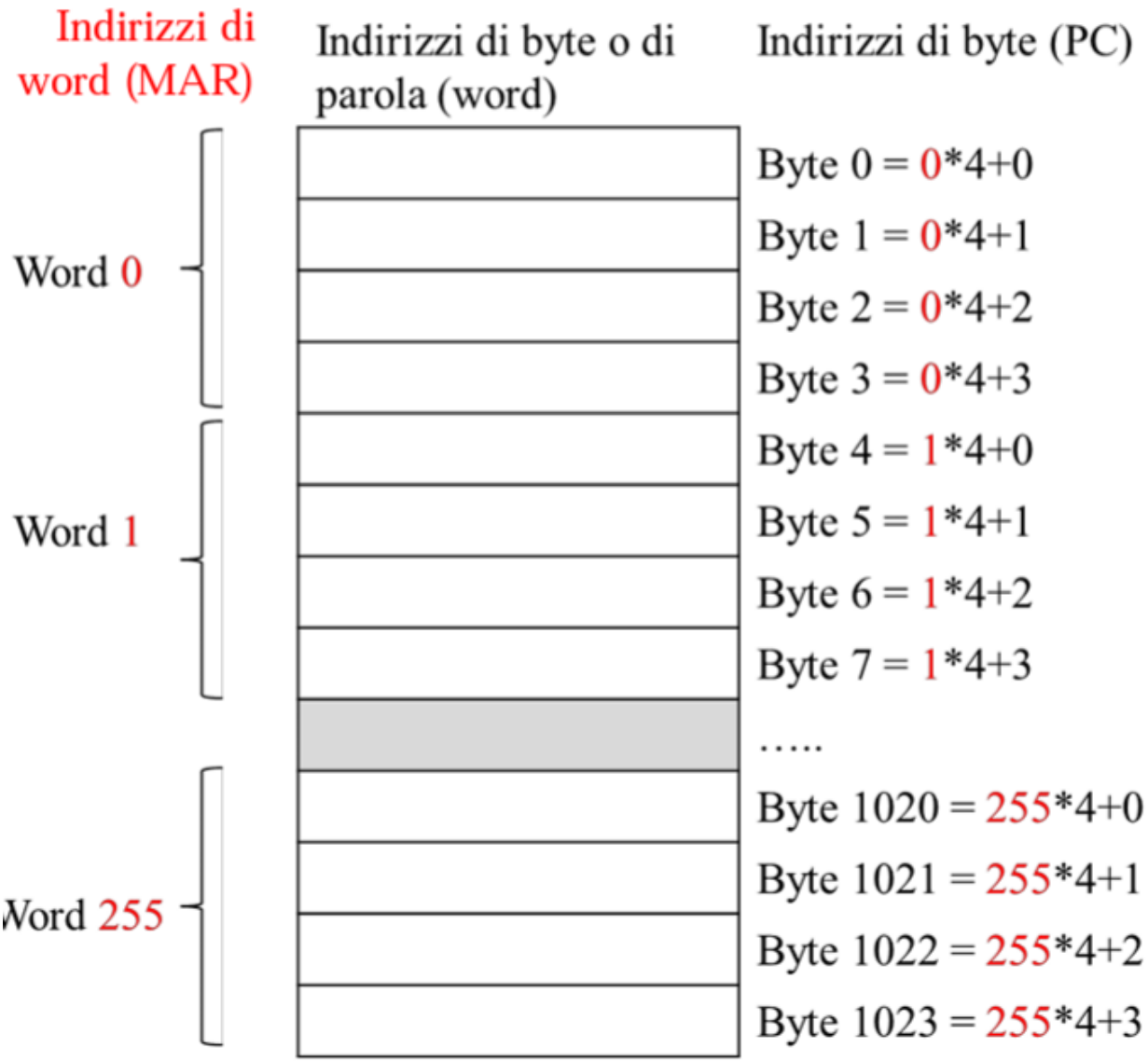
1. L'area contenente il **codice eseguibile** detta area dei **metodi**.
2. L'area delle **costanti** (constant pool).
3. L'area delle **variabili locali**.
4. L'area dello stack degli **operandi**.

Vi sono anche altrettanti registri (32 bit) che sono:

1. **PC** (Program Counter): contiene la prossima istruzione nell'area dei metodi.
2. **CPP** (Constant Pool Pointer): registro che punta alla base delle Constant Pool.
3. **LV** (Local Variable): registro che punta alla base delle variabili locali.

4. **SP** (Stack Pointer): registro che punta alla cima degli operandi.

Ogni word sono 4 byte ovvero 32 bit. Come vengono assegnati gli indirizzi in memoria?



Istruzioni di IJVM

1. Istruzioni di pop e push nelle variabili locali: **ILOAD** e **ISTORE**.
2. Operazioni sullo stack: **BIPUSH**, **LDC_W**, **DUP**, **POP**, **SWAP**.
3. Operazioni aritmetico logiche: **IADD**, **IOR**, **ISUB**.

4. Controllo di flusso: **GOTO**, **ID_EQ**, **IF_LT**, **IF_CMPEQ**.
5. Richiamo dei metodi: **INVOKEVIRTUAL** e **IRETURN**.

Ogni **istruzione è une sequenza di byte**, ma viene scritta tramite **rappresentazione mnemonica** o tramite semantica relazionale. Ogni istruzione ha un codice operativo: 1 byte e da 0,1 o 2 operandi: offset o costante, lunghi 1 o 2 byte, con segno o senza segno.

SP sarà poi incrementato dalla lunghezza dell'istruzione in byte se on specificato diversamente.

La memoria è vista come un array indirizzabile grazie a SP tramite la forma `mem[SP]` il simbolo `:=` significa "memorizza quello a destra in quello che c'è a sinistra". Mentre `==` serve per fare il confronto tra due valori.

Istruzione	Funzione
BIPUSH *argomento1*	Inserisce in cima allo stack il valore *argomento1*.
LDC_W *argomento1*	Inserisce in cima allo stack il valore contenuto in CPP che ha come indirizzo *argomento1*.
ILOAD *argomento1* (con WIDE ILOAD *argomento1* diventa di 16 bit invece che 8 bit)	Inserisce in cima allo stack il valore contenuto nell'area delle variabili in base a LV come indirizzo *argomento1*.
ISTORE *argomento1*	Salva nella zona delle variabili con indirizzo *argomento1* il valore in cima allo stack.
IADD	Somma due valori presenti in memoria e salva il risultato nella cella in cui punta SP e si "cancella" quello più in alto.
ISUB	Sottrae due valori presenti in memoria e salva il risultato nella cella in cui punta SP e si "cancella" quello più in alto.
DUP	Duplica il valore di SP e lo immette in SP + 1.
SWAP	Scambia due valori in memoria con indirizzo SP con SP - 1.

Istruzione	Funzione
GOTO *argomento1*	PC prende il valore di *argomento1* che come indirizzamento ha 16 bit.
IFEQ *argomento1*	Se il valore di offset di *argomento1* nello stack è 0 allora si incrementa il SP di quell'offset.
IF_CMPEQ *argomento1*	Se il valore di offset di *argomento1* nello stack è uguale alla cima dello stack allora porta il valore di offset in cima allo stack.
GOTO *argomento1*	Sposta il puntatore PC di *argomento1* posizioni positive o negative.

Le istruzioni devono essere **tradotte** in **stringhe binarie** per essere svolte, noi scriveremo istruzioni in forma simbolica con il linguaggio **JAS** con il quale l'ambiente di sviluppo potrà trasformarle in binario.

es. ILOAD *j* in questo caso j ha come valore in memoria 800 la nostra istruzione avrà i seguenti stati:

ILOAD *j* → ILOAD *0x02* → 0001 0101 *0000 0010*.

Puntatore	Descrizione
SP (Stack Pointer)	Calcolo delle espressioni
LV (Local Variables)	Record di attivazioni
CPP (Constant Pool Pointer)	Area delle costanti
PC (Program Counter)	Codice eseguibile del programma

Realizzazione di IJVM

Usiamo IJVM come linguaggio ISA di livello 2 e lo realizziamo sulla microarchitettura di livello 1.

Per eseguire istruzioni di livello 2 da livello 1 adottiamo un interprete **M1** il quale ha il compito di eseguire un ciclo di **fetch decode** ed **execute** di linguaggio **M2**.

Per prima cosa si **prende il dato** nella RAM scritto in linguaggio L2 (IJVM) dove attualmente sta puntando PC, l'inteprete scritto in linguaggio L1 **capisce il tipo di**

istruzione, controlla eventuali operandi ed **esegue l'istruzione**.

Ciclo di Fetch Decode ed Execute

1. Viene richiesto il valore di PC al quale sta puntando e si mette in MBR.
    a. MAR va a ricercare il valore nello stack e lo salva su MDR. (o su area delle costanti o nelle espressioni).
    b. MAR va ricerca il valore nello stack e lo sovrascrive con il valore di MDR.

A ogni istruzione IJVM corrisponde una sequenza di istruzioni MAL.

es. IADD in realtà è:

Label	Istruzione
iadd1 (0x60)	MAR = SP = SP - 1; rd; goto iadd2
iadd2	H = TOS; goto iadd3
iadd3	MDR = TOS = MDR + H; wr; goto Main1

Prendiamo l'istruzione main1 formata da PC = PC + 1; fetch; goto (MBR).

Prima della sua esecuzione PC punto dove è memorizzato il codice della prossima istruzione:

Puntatori	Area Metodi
	xyz
PC →	opcode1

Dopo l'esecuzione di main1 salto alla microistruzione con indirizzo opcode1 (in questo caso MBR aveva l'indirizzo di opcode1) e cambio MBR.

Puntatori	Area Metodi
PC →	xyz
	opcode1

Funzione GOTO

L’istruzione GOTO si basa su un offset negativo o positivo che serve per muoversi in memoria

Label	Operazione
Main1	PC = PC + 1; fetch; goto(MBR)
goto1	OPC = PC - 1
goto2	PC = PC + 1; fetch
goto3	H = MBR << 8
goto4	H = MBRU OR H
goto5	PC = OPC + H; fetch
goto6	goto Main1

Istruzione IFLT

Condizione su un operando di 2 byte e viene svolta nel seguente metodo.

Label	Istruzione
Main1	PC = PC + 1; fetch; goto(MBR)
iflt1	MAR = SP = SP - 1; rd
iflt2	OPC = TOS
iflt3	TOS = MDR
iflt4	N = OPC; if(N) goto T; else goto F
T	OPC = PC - 1; goto goto2
F	PC = PC + 1
F2	PC = PC + 1; fetch
F3	goto Main1

> Nella prossima parte:
>
> Programmazione IJVM

> Documentazione

> Funzione potenza

> MIC-2
>
> MIC-2

> MIC-3
>
> MIC-3

> MIC-4
>
> MIC-4

> Esercizi
>
> Esercizi

La Cache

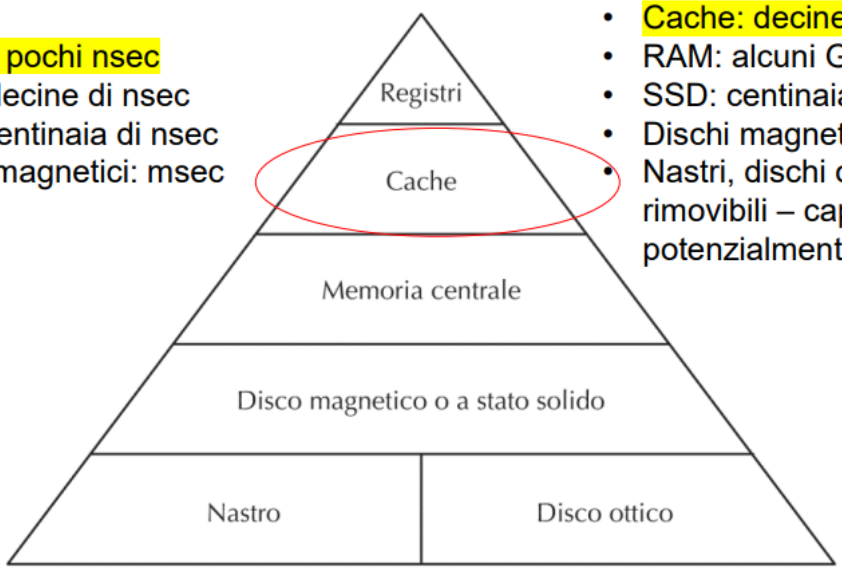
Il divario tra la velocità della RAM rispetto a quella della CPU è diventato un problema, si è così creata una memoria molto veloce per mitigare questo problema la quale permette di non perdere cicli di clock.

I tempi di accesso crescono verso il basso:

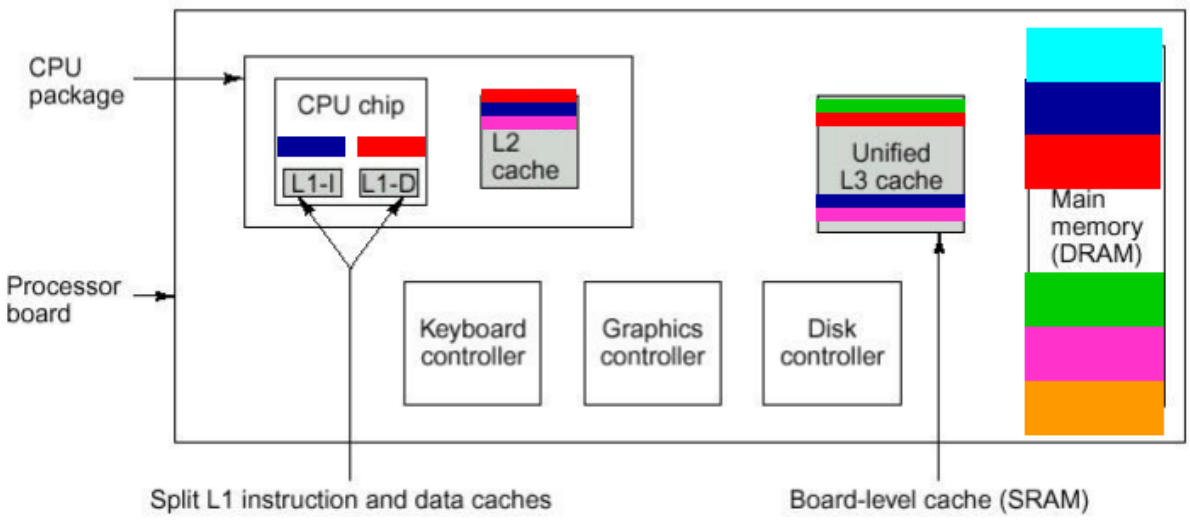
- Registri: <= 1 nsec (periodo clock)
- Cache: pochi nsec
- RAM: decine di nsec
- SSD: centinaia di nsec
- Dischi magnetici: msec

La capacità cresce verso il basso:

- Registri: decine di byte
- Cache: decine KB -qualche MB
- RAM: alcuni GB
- SSD: centinaia di GB
- Dischi magnetici: TB
- Nastri, dischi ottici: supporti rimovibili – capacità potenzialmente infinita



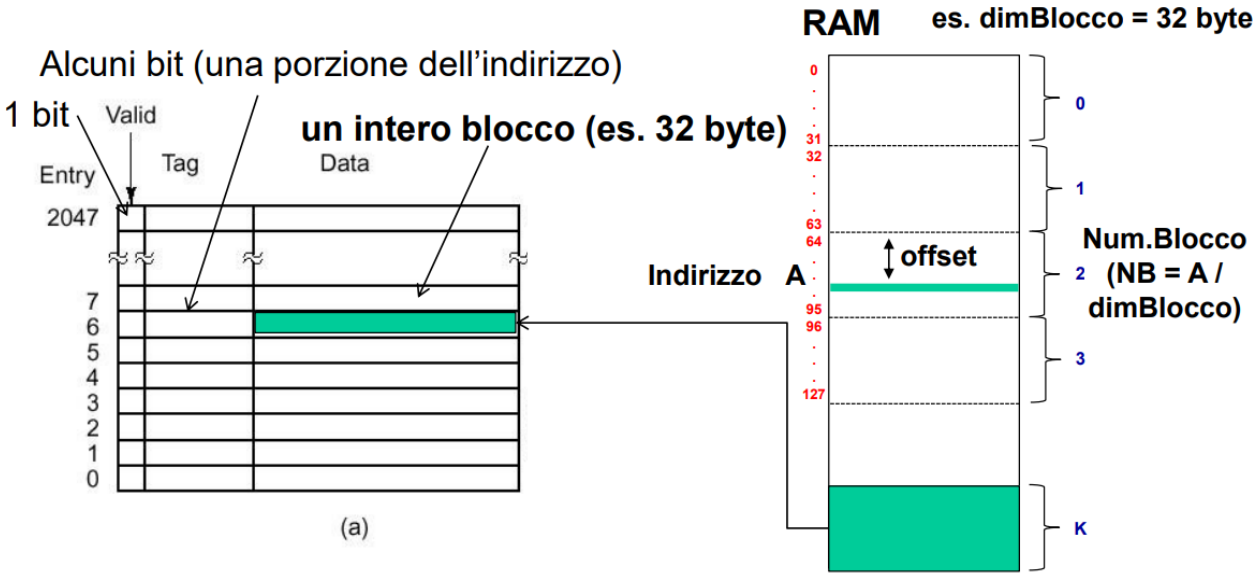
L'obiettivo è quello di ridurre i tempi di accesso tra RAM e CPU. RAM e Cache sono divsi a blocchi:



La cache non deve modificare le istruzioni, noi consideriamo i blocchi perchè quando spostiamo un blocco, stiamo spostando anche delle parole vicine al blocco. Per questo si mettono delle regole per l'accesso alla RAM:

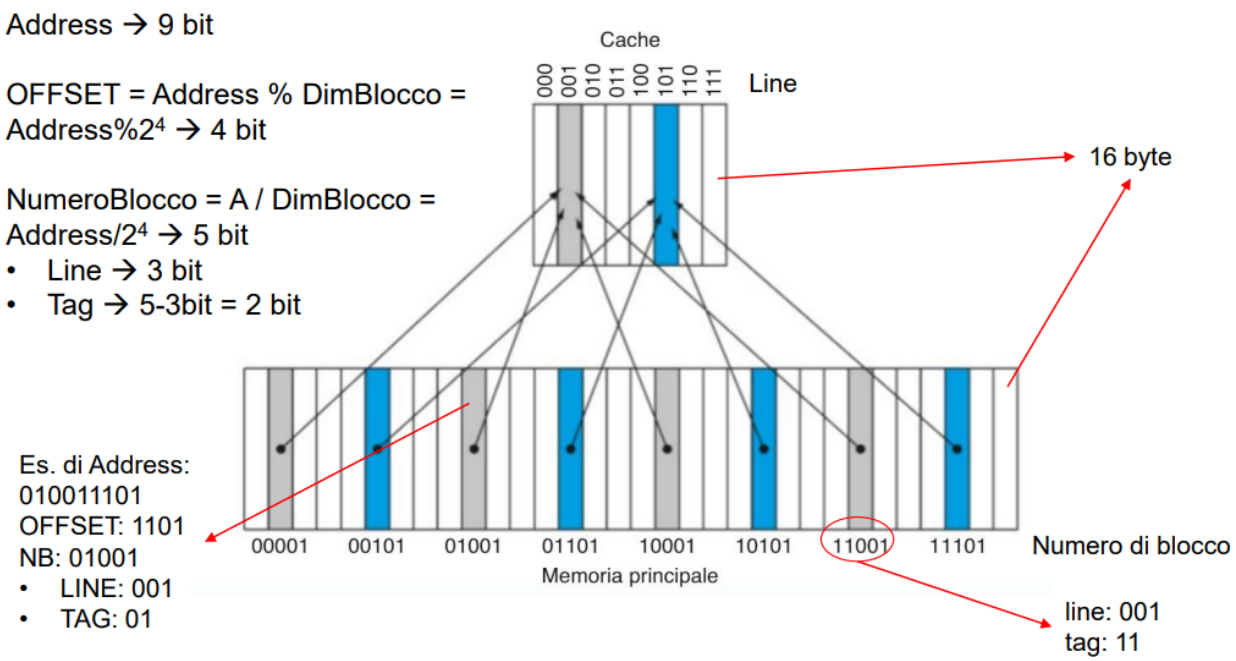
- **Località spaziale:** se acceddiamo a un dato probabilmente accederemo allo località vicine.
- **Località temporale:** quando si fa accesso alla stessa posizione più volte come in un ciclo, spostiamo allora il blocco in cache.

I blocchi della cache sono anche chiamate linee da non confondere con una riga della cache. L'organizzazione avviene:



Ogni riga è composta da parti diverse:

- **Line**: corrisponde all'indice della riga della cache.
- **Tag**: corrisponde al tipo di dati.
- **Word**: è l'offset in termini di word.
- **Byte**: il byte nella parola.

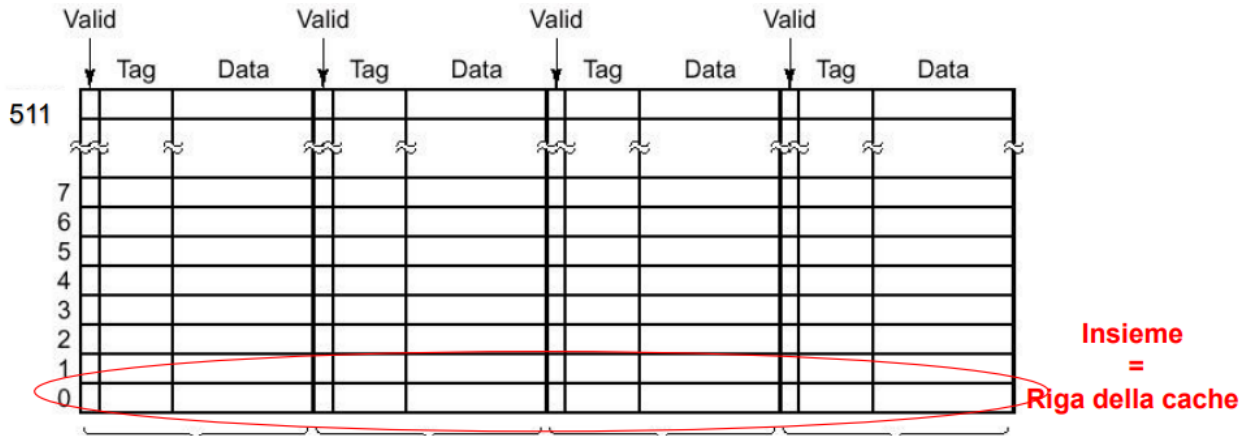


Per scomporre la parola da 9 bit: 101100010 dobbiamo:

10(Tag) 110(Line) 0010:

Line	Valid	Tag	Blocco
110	1	10	1101

Possiamo associare più blocchi di cache assieme attraverso uno schema:

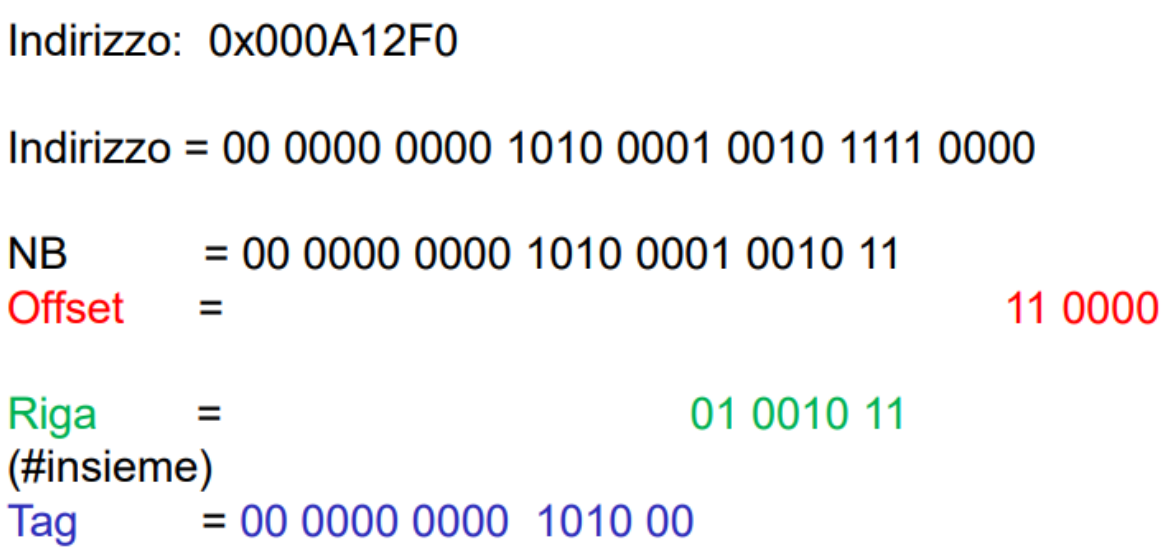


Così facendo possiamo avere in taotale 2048 linee di cache. La ricerca del blocco avviene su n righe della cahce.

- Per capire dove inserire i blocchi: riga = numBlocchi % numRighe.

- Identificare quali blocchi sono nella cahce = numBlocchi / numRighe.
- Quali blocchi rimpiazzare: su base statistica

Per calcolare il numero di righe: numRighe / n.



Gestione degli accessi

Ci sono due metodi: **Write Through**, la scrittura è sia in cache che in memeoria, se al momento della scrittura il blocco non è presente bisogna decidere se allocarlo o meno. **Write Back** si aggiora solo la cache e le modifiche si riportano solo dopo aver modificato il blocco.

Se dovesse fallire la scrittura abbiamo un **Write Allocation** adottata con il Write Back e non adottata con il Write Through, serve per portare il blocco in memoria al momento della scrittura in memoria.

Dobbiamo calcolare **l'hoverhead** ovvero i dati necessari per far funxionar la cache.

Altre funzioni della cache

Si possono usare per immagazinare l'indirizzo dell'istruzione di salto, oltre a questo possiamo avere informazioni per una predizione sul salto.

Possiamo avere tabelle per la previsione (**previsione dinamica**): bracn / no brach, prediction bits e target address

Esercizio

Una cache di primo livello per le istruzioni ha un'organizzazione di tipo "associativa a due vie". Tale cache può contenere fino a 64Kbyte di codice, suddivisi in blocchi da 32 byte. Supponendo che gli indirizzi generati dalla CPU siano di 32 bit, e che tali indirizzi si riferiscano ai byte, e partendo da una situazione iniziale di *cache completamente vuota*:

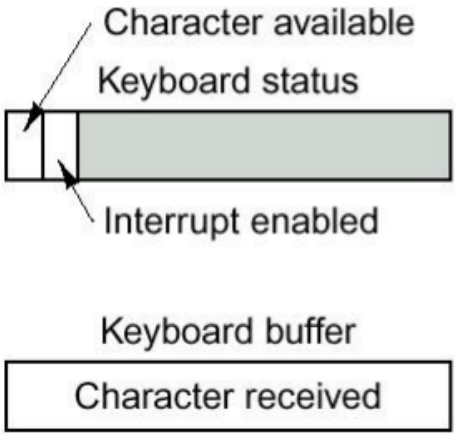
- Prendiamo 64Kbyte (dimensione massima) / 32byte → $2^{16}/2^5$ = $2^{16-5} = 2^{11}$ quindi in totale potremo memorizzare 2Kbyte blocchi **risultato totale blocchi**.
- **Numero totali di righe** = totale blocchi / numero vie → $2^{11}/2^1 = 2^{10}$.

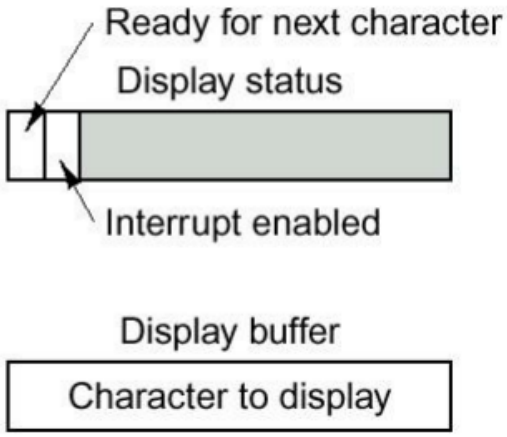
2. l'indirizzo 0x0000D8B1 si trova nello stesso blocco dell'indirizzo 0x0000D8A3? Perché?

- Trasformiamo il numero in binario 0x0000D8A3 → 0000 0000 0000 0000 1101 1000 1010 0011. Da destra sinistra sappiamo che: **l'offset** è dato da **l'esponente della dimensione del blocco** (32bit → 2^5) quindi 5 bit mentre la dimensione della **line dall'esponente del totale delle righe** quindi 10 bit. I **restanti per la tag**.
- Trasformando entrambi i numeri notiamo che appartengono alla stessa tag e alla stessa line quindi appartengono allo stesso blocco.

Linguaggio ISA

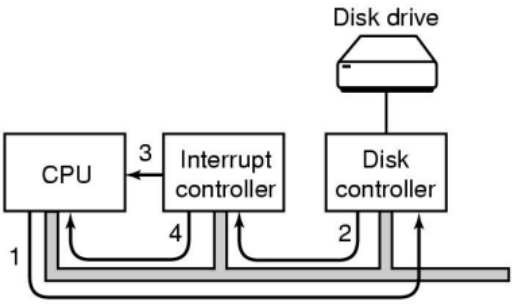
Dato che I/O e CPU hanno velocità diverse, non ha senso tenere la CPU bloccata (Busy Waiting), quindi avrebbe molto più senso ferle fare altro.





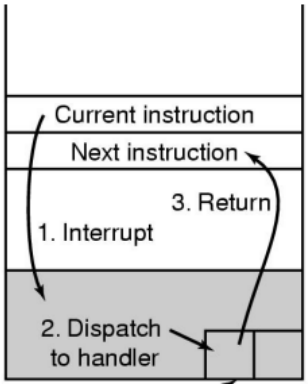
Ci sono dei bit detti di “interrupt enable” servono per attivare i **bit di interrupt**.

AZIONI HARDWARE

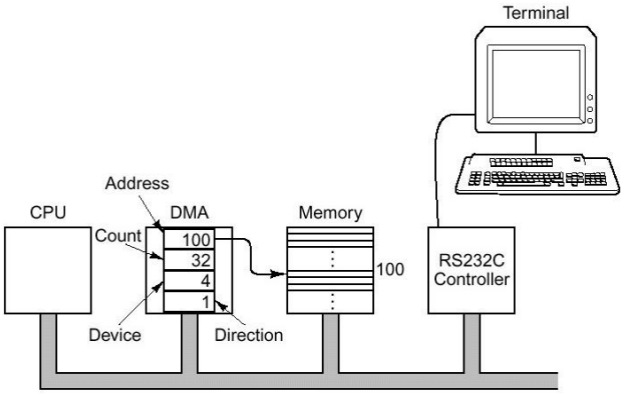


1. la CPU invia un comando di I/O al controller.
2. il controller del disco al termina invia un interrupt.
3. l’interrupt del controller invia un segnale alla CPU.
4. l’interrupt del controller indica alla CPU l’ID di chi ha generato il segnale

AZIONI SOFTWARE



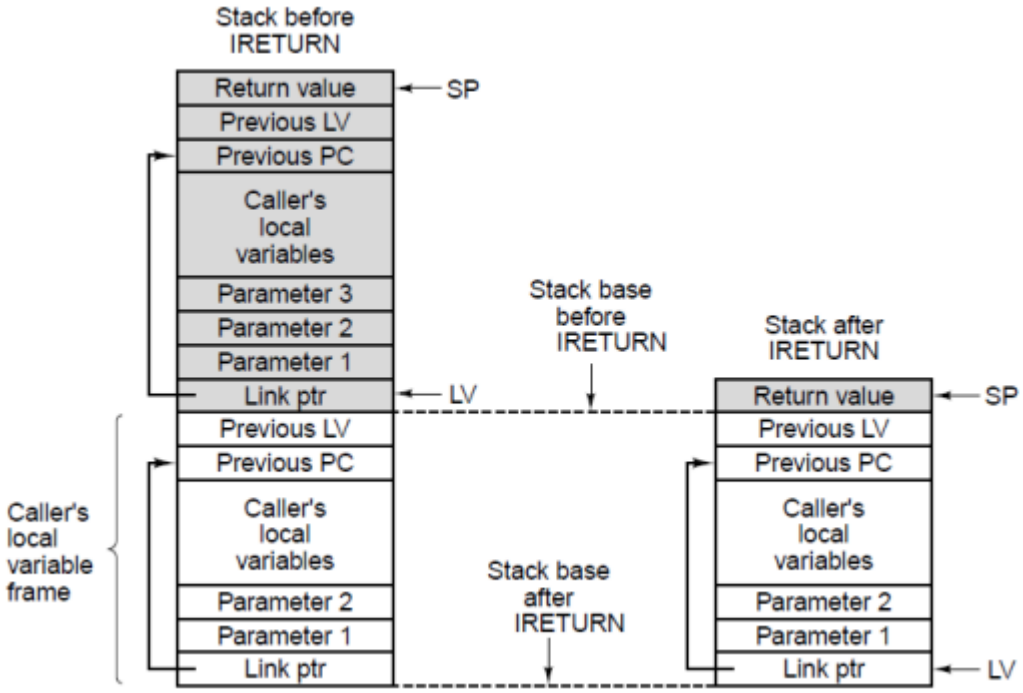
1. la CPU quando riceve una segnalazione termina la istruzione corrente.
2. passa il controllo a una routine per gestire l’interrupt
3. termina la gestione e riprende l’esecuzione



Il controller DMA è in gradi di gestire autonomamente lo spostamento dei dati. Ci sono però dei conflitti come il Cycle Stealing ma la CPU deve gestire meno interrupt.

Variazione del flusso di controllo

La procedura chiamante impila i parametri attuali ed esegue una **istruzione CALL**. Questa istruzione **impila l'indirizzo di ritorno**. La procedura chiamata impila FP (puntatore record attivazione) incrementa SP e aggiorna FB.



Nell'architettura MIPS vediamo come si realizza la chiamata a una procedura, ci serviamo delle seguenti risorse:

Nome	Numero del registro	Utilizzo
$zero	0	Costante 0
$v0-$v1	2–3	Risultati e valutazione di espressioni
$a0-$a3	4–7	Argomenti
$t0-$t7	8–15	Variabili temporanee
$s0-$s7	16–23	Variabili da preservare
$t8-$t9	24–5	Altri registri temporanei
$gp	28	Global pointer
$sp	29	Stack pointer
$fp	30	Frame pointer
$ra	31	Indirizzo di ritorno

Sfruttiamo i registri per il passaggio dei dati sui registri, abbiamo anche la necessità di salvare lo stato delle chiamate. Il software MIPS, per le chiamate a procedura, utilizza i registri secondo queste convenzioni:

- $a0-$a3: quattro registri per il passaggio dei parametri attuali.
- $v0-$v1: due registri per la restituzione dei risultati.
- $ra: un registro contenente l'indirizzo di ritorno.

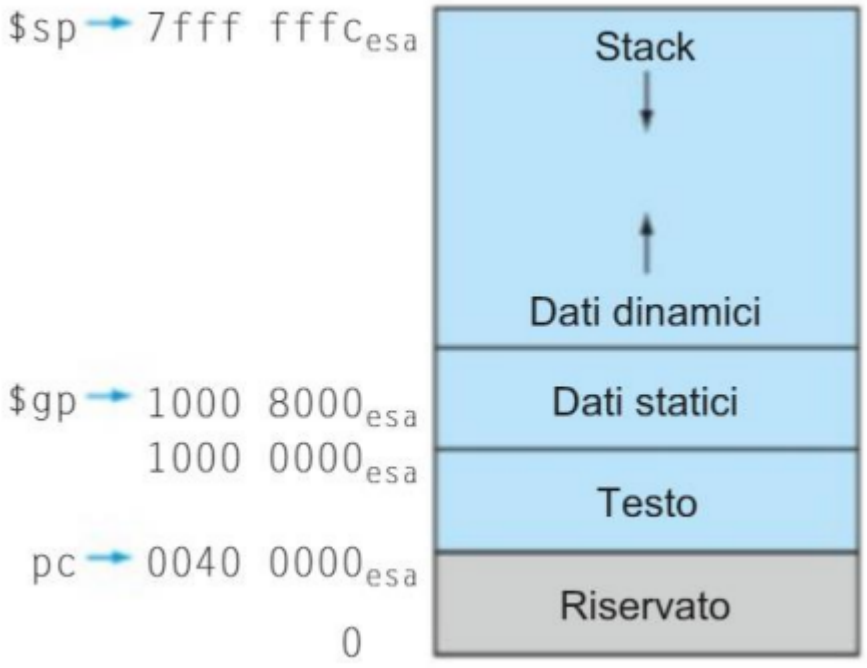
Il chiamante:

- inserisce i parametri attuali nei registri $a0-$a3.
- utilizza jal X per saltare alla procedura X.
- recupera i risultati dai registri $v0-$v1.

La procedura chiamata:

- esegue le operazioni per elaborare i valori nei registri $a0-$a3.
- inserisce i risultati nei registri $v0-$v1.

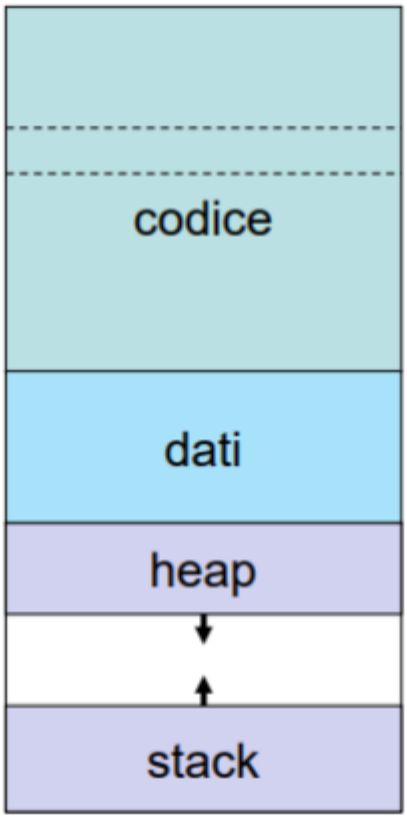
- utilizza jr $ra per restituire il controllo al chiamante.



Memoria virtuale

Cache	Memoria virtuale
• La cache (gestita dall'hardware) permette di fornire l'illusione all'utente di disporre di una memoria grande quanto la RAM, con una velocità intermedia fra quella della RAM e quella (molto più veloce) della cache • Si mantiene in cache solo una parte dell'immagine in memoria del processo in esecuzione a cui il processo accede ripetutamente: – codice delle procedure in esecuzione – dati su cui lavorano tali procedure.	• Il disco è usato per contenere l'intera immagine del processo (e di tutti i processi in vita nel sistema) • Si carica in RAM solo la parte dell'immagine del processo (codice e dati) acceduta da un processo in una data fase dell'esecuzione. – In RAM si può mantenere una porzione più grande del processo di quanta se ne tiene in cache. – Lo spostamento da disco a RAM e viceversa avviene a blocchi (pagine o segmenti).

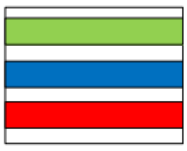
Agli inizi la memoria era divisa in parti detta “overlay” contente parti di codice e programma, tutto a carico del programmatore.



In un sistema mono-programma

- permettere di eseguire programmi che necessitano di più spazio di quello totale disponibile in memoria centrale.

Spazio degli indirizzi



Memoria

In un sistema multi-programmato

- permettere di mantenere attivi un insieme di processi che complessivamente occuperebbero più spazio della memoria centrale disponibile.
- proteggere lo spazio di indirizzamento di un processo, rendendolo accessibile solo a quel processo

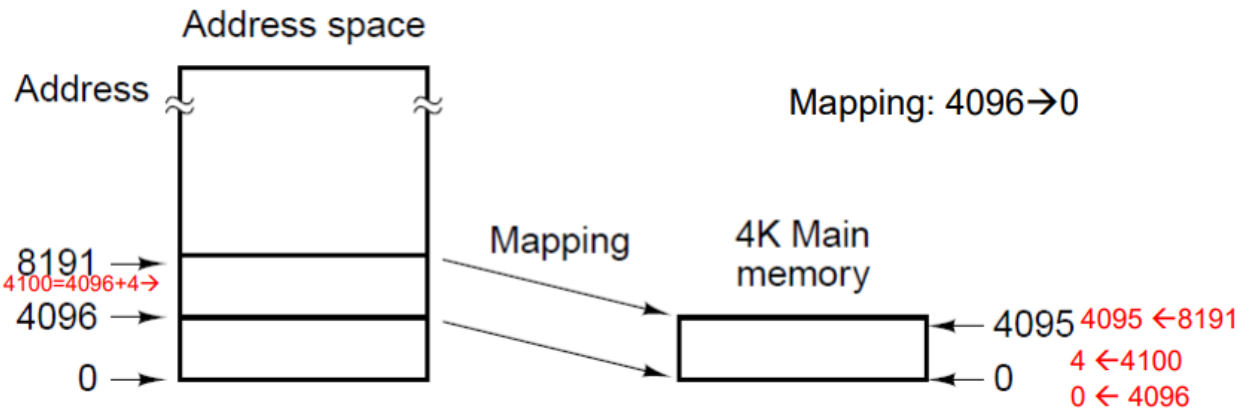
Alcune parti dell'immagine di un processo possono, in un'alta percentuale delle esecuzioni del programma, non essere mai necessarie:

- routine che sono chiamate raramente: funzionalità di un programma utilizzate poco di frequente dall'utente, routine di gestione di errori
- strutture dati (es. array) allocate di grandi dimensioni ma, in molte esecuzioni, utilizzate soltanto per una piccola parte
- La memoria virtuale su richiesta carica solo le parti effettivamente utilizzate (solo quando servono) risparmiando spazio

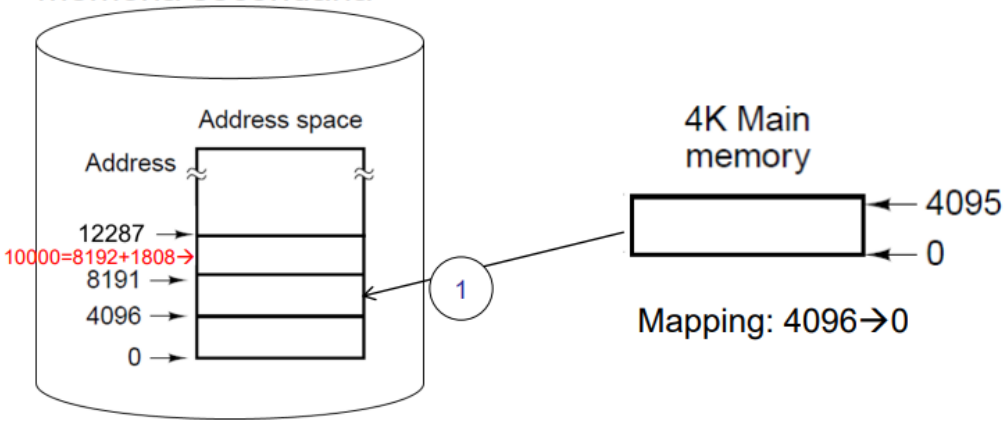
- La memoria virtuale si basa sulla separazione dei concetti di:
  - locazioni di memoria: l'insieme degli indirizzi delle celle di memoria (dipende dalla dimensione della RAM)
  - spazio degli indirizzi (immagine): le posizioni indirizzabili da un programma tramite le modalità di indirizzamento offerte da ISA

- Esempio: in un elaboratore con una RAM di 2^{12} parole e istruzioni con campi di indirizzamento diretto di 16 bit
  - le locazioni di memoria sono 4096 (=2^{12}) con indirizzi fisici: 0-4095
  - lo spazio di indirizzi consiste di 65536 (=2^{16}) posizioni con indirizzi virtuali (o indirizzi logici): 0-65535

- La memoria virtuale permette di realizzare corrispondenze flessibili e dinamiche tra:
  - indirizzi virtuali: gli indirizzi dell'immagine di un processo
  - indirizzi fisici: le locazioni della memoria fisica



- Tutto lo spazio degli indirizzi (l'immagine del processo) è salvata su memoria secondaria



- Nel momento in cui il programma fa riferimento ad un indirizzo di un "blocco" non in memoria (es. 10000):
  1. il "blocco" in memoria è salvato nella corrispondente posizione nel disco

Con il nuovo metodo dello swapping, il programmatore non deve più pensare a gestire gli overlay, è tutto automatico.

L'immagine del processo si trova sull'hard disk, quello in memoria è una copia.

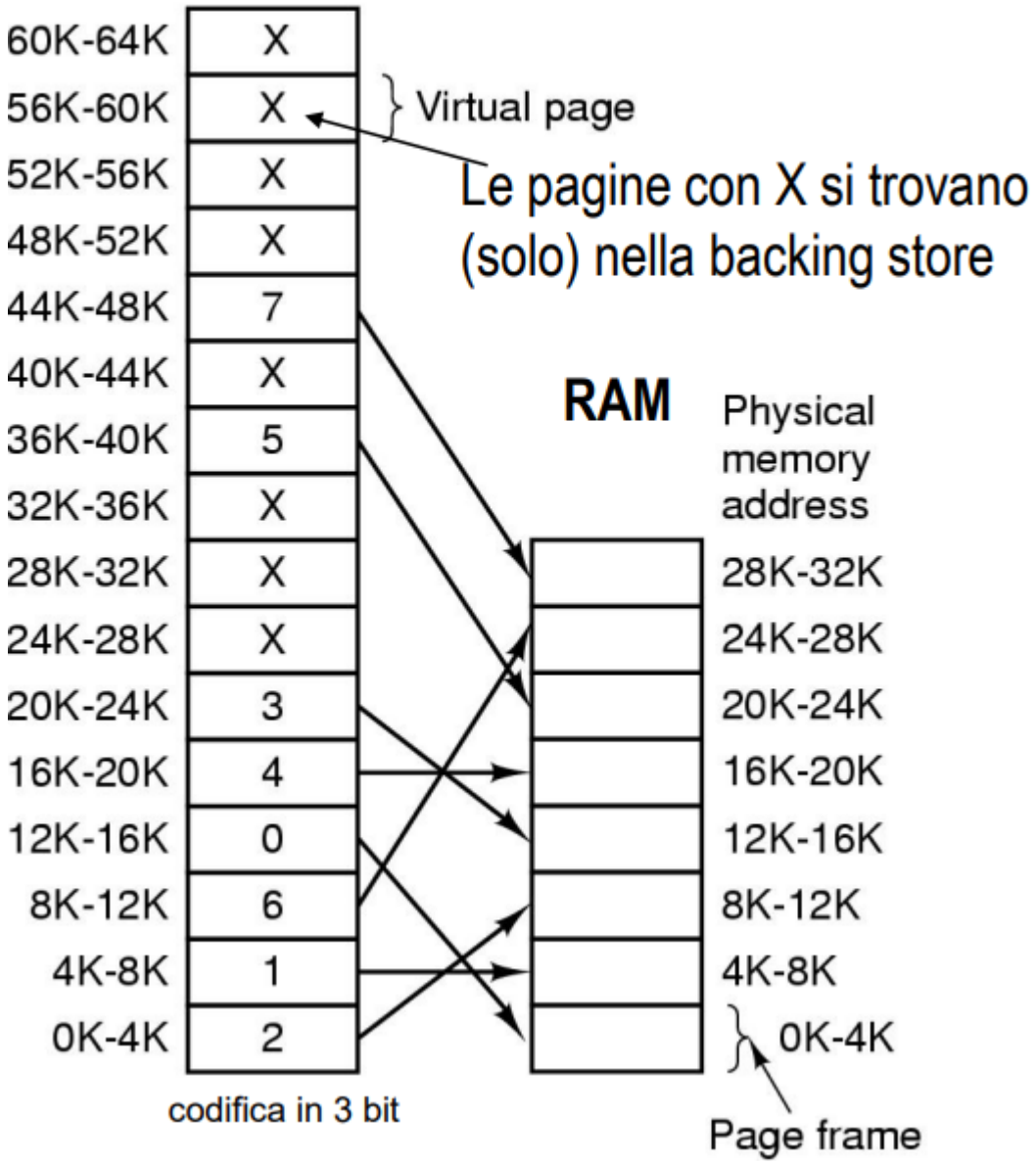
Il programma effettivamente non p sìpresente in memoria RAM ma viene salvata dentro la memoria centrare hard disk, e quando viene la necessita si mette in memoria RAM, vengono create della pagine dette frame le quali hanno delle mimensioni prestabilite.

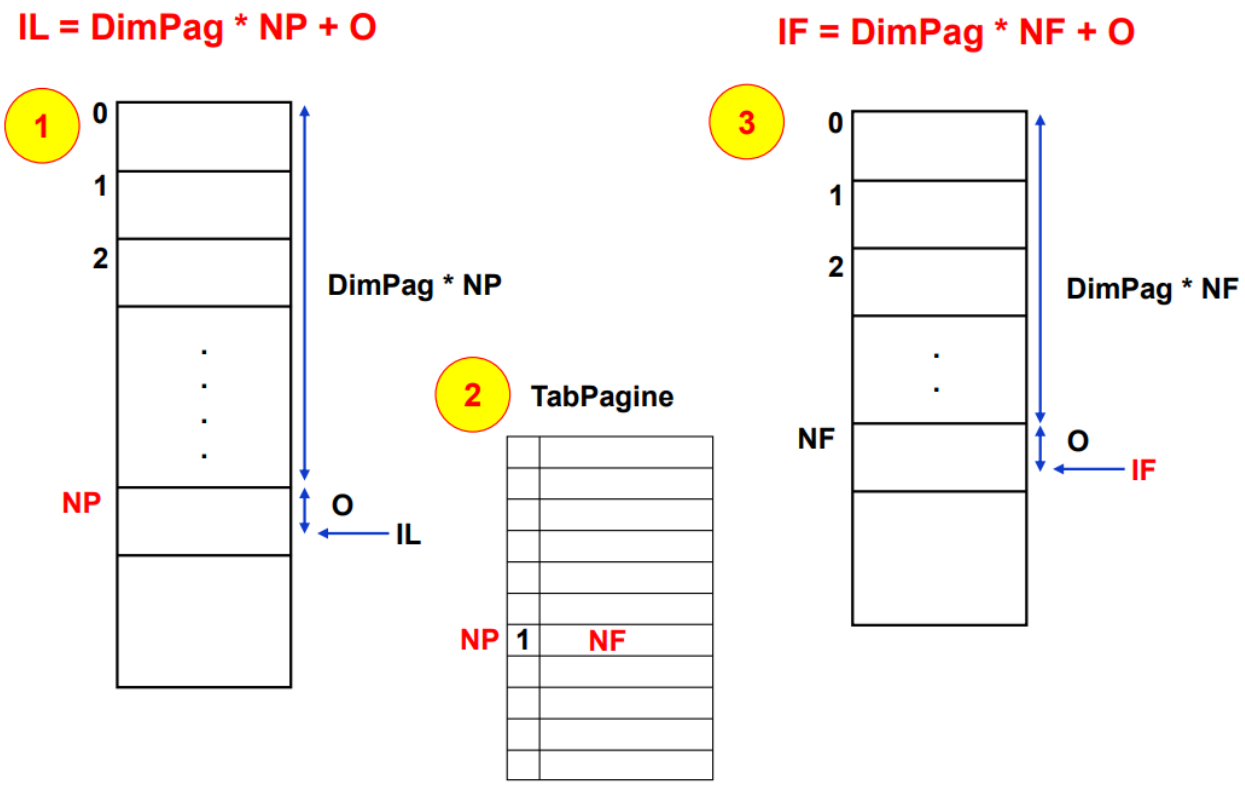
Page	Virtual addresses
≈	≈ ≈
15	61440 – 65535
14	57344 – 61439
13	53248 – 57343
12	49152 – 53247
11	45056 – 49151
10	40960 – 45055
9	36864 – 40959
8	32768 – 36863
7	28672 – 32767
6	24576 – 28671
5	20480 – 24575
4	16384 – 20479
3	12288 – 16383
2	8192 – 12287
1	4096 – 8191
0	0 – 4095

Page frame	Bottom 32K of main memory Physical addresses
7	28672 – 32767
6	24576 – 28671
5	20480 – 24575
4	16384 – 20479
3	12288 – 16383
2	8192 – 12287
1	4096 – 8191
0	0 – 4095

Questo mapping è possibile grazie alla tabella delle pagine, la quale è formata nel seguente modo:

Associazione pagina logica-frame per il processo P1





Per calcolare la posizione del frame usiamo il metodo sovrastante.

Trattamento del page fault

Se la pagina non è presente il sistema oprativo deve:

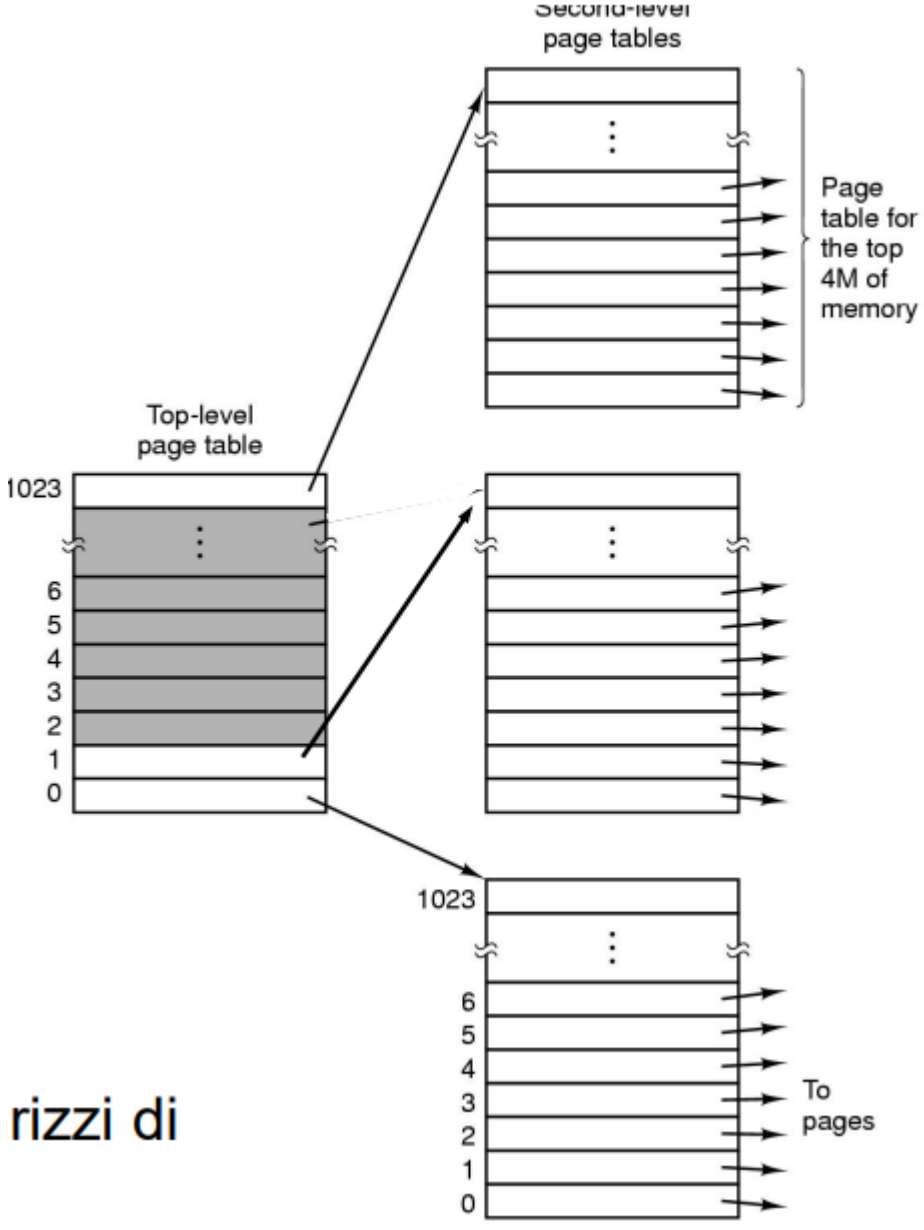
- Cercare un frame libero in RAM, se non c'è sceglie una pagina vittima
- Se necessario se la pagina vittima è odificata rispatto al disco bisogna salvasre la vittima su disco
- Caricare la pagina che causata il page fault da disco a RAM.

Ogni processso ha la sua tabella delle pagine, se l'inidirzzo fisico appartine e a un altro processo non possiamo accedervi, grazie alla logica della tabelle.

Si introduce una "cache" per un numero di pagine logiche corrispondente alle pagine fisiche, **Translation Lookaside Buffer** (TLB), si entra in parallelo logico (n - k bit), si matcha su differenti page number, se va a buon fine si restituisce il frame number.

Per le tabelle delle pagine vi è una directory per organizzare:

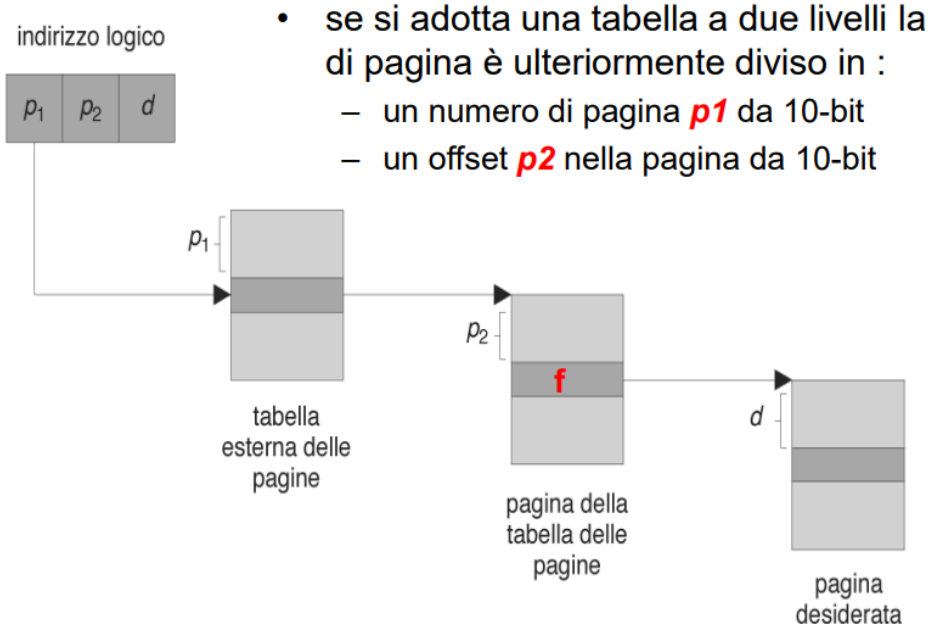
- Una root dove ogni elemento punta a ogni sottotabella.
    - Una sottotabella invece contiene i frame con il mapping.



rizzi di

Un indirizzo logico su una macchina a 32-bit con pagine di 4K è diviso in un numero di pagina di 20 bit e un offset ***d*** di 12 bit

- se si adotta una tabella a due livelli la tabella, il numero di pagina è ulteriormente diviso in :
  - un numero di pagina ***p1*** da 10-bit
  - un offset ***p2*** nella pagina da 10-bit



Così è come funziona la MMU (Memory Management Unit)