Apunts del Tema 6
Memòria Cache

Joan Manuel Parcerisa
Rubèn Tous

Departament d'Arquitectura de Computadors
Facultat d'Informàtica de Barcelona
Febrer 2023

Licencia Creative Commons

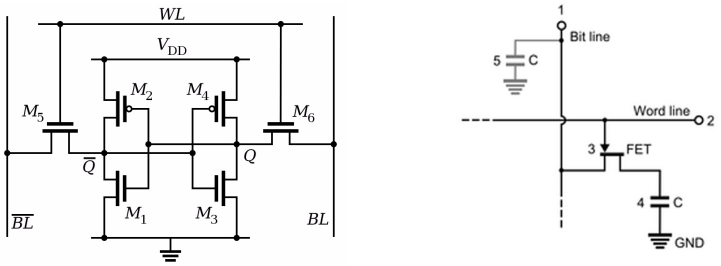
Tema 6. Memòria Cache

5.4

1. Introducció

Tot computador està provist d'una memòria persistent, també anomenada emmagatzemament secundari, on es guarden tots els programes quan aquest està apagat, i està normalment constituïda per discos durs (magnètics) o xips de memòria SSD (d'estat sòlid). La memòria secundària té un cost per bit molt baix i acostuma a tenir una gran capacitat d'emmagatzematge, però té l'inconvenient de tenir un *temps d'accés* extremadament lent, típicament més de 5 ms en el disc dur, i més de 0,02 ms en els SSD. Per a un període de rellotge de CPU típic de 0,25 ns, un accés tardaria desenes o centenars de milers de cicles.

Per aquesta raó, els computadors disposen també d'una memòria principal d'accés aleatori o RAM (random access memory), de tipus volàtil i normalment d'accés molt més ràpid que el disc (20 ns), que és on es copien les instruccions i les dades d'un programa quan el volem executar, i hi resideix mentre s'executa. La RAM es pot implementar com memòria estàtica (SRAM) o dinàmica (DRAM). La primera és més costosa ja que cada cel·la requereix almenys 6 transistors per bit, mentre que la segona pot implementar-se amb 1 transistor i un element capacitiu per bit (Figura Figura 6.1.). Els computadors solen usar DRAM per a la memòria principal ja que tenen molta més capacitat per a una mateixa àrea de xip i menor cost per bit.



(a) 1 bit en SRAM (6 T) (b) 1 bit en DRAM (1 T + 1C)

Figura 6.1. Esquemes elèctrics d'una cel·la de memòria de 1 bit en SRAM i en DRAM

1.1 El problema del *gap* entre memòria i processador

Al llarg dels anys, els processadors han augmentat exponencialment el rendiment gràcies a la reducció dels temps de conmutació dels transistors i al creixent processament paral·lel d'instruccions. Al seu torn, els progressos tecnològics han permès també reduir els temps d'accés a la memòria principal (DRAM). Però la millora de rendiment de les

memòries no ha evolucionat al mateix ritme que la dels processadors (vegeu Figura 6.2), en part degut que la capacitat de memòria instal·lada ha crescut també exponencialment.

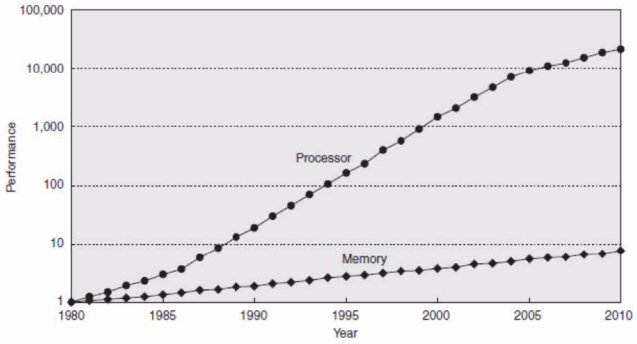


Figura 6.2. Evolució del rendiment relatiu de processador i memòria

El resultat és que actualment el temps d'un accés a *memòria* RAM[1] ($t_{accés}$) pot arribar a ser de 100 o 200 cicles de CPU, i és més de 100 cops major que el temps (t_{proc}) emprat en la resta d'operacions del *processament* d'una instrucció (decodificació, lectura de registres, ALU, etc.), diferència que anomenem *gap* entre processador i memòria.

Quin és el problema? Per a una freqüència de rellotge donada, el rendiment és inversament proporcional al CPI (essent CPI = $t_{proc} + t_{accés}$). Suposant que $t_{accés} = 100 \cdot t_{proc}$, la fracció del temps d'execució invertida en el processament de la instrucció (sense comptar l'accés a memòria) és $\alpha = t_{proc}/(t_{proc} + t_{accés})$, és a dir $\alpha \approx 0{,}01$. Si solament som capaços de millorar t_{proc} (sense canviar $t_{accés}$), per la llei d'Amdahl, el màxim guany de rendiment (speedup) que podem obtenir és $1/(1-\alpha) \approx 1{,}01$ (o sigui, un 1%). Veiem doncs que millorar el temps d'accés a la memòria resulta imprescindible per a qualsevol intent de millorar el rendiment. Com veurem a continuació, la memòria cache serà la solució.

1.2 Localitat dels programes

La clau per a la solució del problema sorgeix de l'anàlisi del comportament dels programes. Els estudis de simulació mostren que la majoria de programes presenten (encara que uns en major mesura que altres) dues importants propietats anomenades "de localitat":

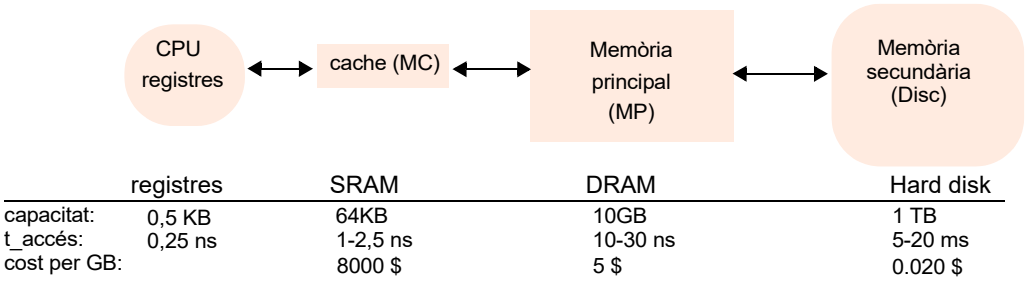
Localitat temporal: *Si accedim a una adreça de memòria, és probable que hi tornem a accedir en un futur proper.* Aquesta propietat és deguda, entre altres raons, a l'existència de bucles en els programes, que fan que s'accedeixi repetidament a les mateixes instruccions i dades.

Localitat espacial: *Si accedim a una adreça de memòria, és probable que s'accedeixi a adreces "properes" en un futur proper.* Pel que fa a la búsqueda d'instruccions en memòria, aquesta propietat és deguda al *seqüenciament implícit*: el processador sempre incrementa el PC (comptador de programa) i va a buscar la instrucció següent en memòria (excepte en els salts). Pel que fa a les dades, la localitat espacial és deguda als recorreguts de vectors i matrius emmagatzemats en posicions contigües de memòria.

1. Nota: L'execució d'una instrucció de llenguatge màquina implica com a mínim un accés a la memòria, el necessari per fer el *fetch* (cerca) de la instrucció. A més a més, les instruccions de tipus *load* o *store* realitzen un segon accés per llegir o escriure una dada. Si no es diu el contrari, en aquest tema s'usarà el terme "dada" per referir-se indistintament a les instruccions i les dades, i s'usarà el terme "referència" o "accés" a memòria per referir-se indistintament a una lectura o una escriptura.

1.3 La memòria cache

La memòria cache (MC) és una memòria auxiliar petita i ràpida que s'interposa entre el processador i la memòria principal[2] (MP), més gran i més lenta (vegeu Figura 6.3.). Les memòries construïdes amb tecnologies més ràpides (registres, SRAM) tenen un cost econòmic més gran per bit que les més lentes (DRAM, SSD, disc), i per tant una menor capacitat:



	registres	SRAM	DRAM	Hard disk
capacitat:	0,5 KB	64KB	10GB	1 TB
t_accés:	0,25 ns	1-2,5 ns	10-30 ns	5-20 ms
cost per GB:		8000 $	5 $	0.020 $

Figura 6.3. Capacitat, temps d'accés i cost per GB típics

La MC emmagatzema un subconjunt del contingut de la MP. L'objectiu de la MC és intentar retenir aquelles dades que tinguin més probabilitat de ser accedides en el futur, aprofitant la localitat per accedir-hi més ràpidament i millorar el rendiment:

- La MC explota la *localitat temporal*: cada cop que la CPU accedeix a una nova dada de la MP, també en guarda simultàniament una còpia a la MC. Si en endavant la mateixa dada es torna a referenciar i encara està a la MC, ja no caldrà accedir a la MP, s'accedirà a la MC en un temps molt més curt.
- La MC explota la *localitat espacial*: cada cop que la CPU accedeix a una nova dada de la MP, no sols copiem a la MC la dada individual sinó tot el bloc al qual pertany. Si en endavant accedim a una altra dada que està pròxima a aquesta (en el seu mateix bloc), s'hi accedirà a la MC en un temps molt més curt.

Analogia de la cache amb l'escriptori: Considerem la següent analogia: un estudiant d'història se'n va a l'arxiu de la seva ciutat per elaborar un treball de recerca (anàleg al nostre *programa*) per al qual necessitarà consultar diversos informes o documents (les *dades*) que es troben agrupats en carpetes (*blocs de dades*) a les prestatgeries de l'arxiu (la *memòria principal*). Cada cop que necessita fer una consulta d'un document (*accés a memòria*), es desplaça a la corresponent prestatgeria de l'arxiu, fa la consulta, i retorna al seu lloc de treball. Però com que alguns documents s'han de consultar repetidament (*localitat temporal*), l'estudiant decideix emportar-se'ls al seu escriptori (analogia de *la cache*) per estalviar viatges i tenir-los més a mà (*temps d'accés* més curt). Per altra banda, com que molts dels documents que va a buscar a la prestatgeria no es troben dispersos sinó que estan agrupats dins les mateixes carpetes, a prop uns dels altres (*localitat espacial*), l'estudiant decideix que cada cop que faci una consulta a la prestatgeria s'endurà no tan sols el document que necessita (*dada*) sinó la carpeta sencera que el conté (*bloc de dades*).

2. Originalment, cache és el nom que es va donar a la memòria que es situa entre el processador i la memòria principal. Avui dia, aquest terme fa referència a qualsevol tipus d'emmagatzematge que aprofiti la localitat. Actualment, a diferència de la memòria principal, generalment implementada en DRAM, la cache s'implementa en SRAM i sovint s'integra dins el mateix xip del processador.

1.4 Terminologia

Definim a continuació alguns termes que usarem per descriure el funcionament d'una cache i per mesurar-ne el rendiment,

Encert, fallada i reemplaçament: Quan la CPU ha d'accedir a una dada de la memòria, en primer lloc comprova si aquesta està en algun dels blocs guardats a la cache. Si la dada ja es troba a la cache, es produeix un *encert* (*hit*) i l'accés té lloc en un temps molt curt. En cas contrari, si la dada no hi és direm que es produeix una *fallada* (*miss*). En aquest cas, caldrà accedir a la memòria principal en un temps considerablement més llarg i (suposant de moment que es tracta d'una lectura) caldrà copiar a la cache no sols la dada sinó també tot el bloc de memòria al qual pertany. Com que la cache té una capacitat limitada, si en intentar copiar-hi un nou bloc aquest ja no hi cap, llavors aquest bloc ha de *reemplaçar* algun dels blocs guardats.

Taxa d'encert, taxa de fallada: S'anomena *taxa d'encerts* (*hit ratio*) d'un programa, i es denota per *h*, al quocient entre el nombre d'encerts de cache i el nombre total de referències a memòria. Igualment, s'anomena la *taxa de fallades* (*miss ratio*), i es denota per *m*, al quocient entre el nombre de fallades i el de referències a memòria:

$$h = \frac{num\_encerts}{num\_referències}$$

$$m = \frac{num\_fallades}{num\_referències} = 1 - h$$

Temps d'accés (o de servei), temps en cas d'encert i temps de penalització: El *temps d'accés* (o *temps de servei*) a memòria, que es denota per $t_{accés}$ és el temps transcorregut entre que la CPU sol·licita accedir a una dada del subsistema de memòria i el moment que finalitza la transferència, de la memòria a la CPU o viceversa, i és variable. En cas d'encert, aquest temps és t_h, i és una característica constructiva de la cache. Normalment s'expressa en segons (però segons el context, el podem expressar també en cicles de rellotge de la CPU). Aquest temps inclou la comprovació de si la cache té guardat o no el bloc de memòria al qual pertany la dada, així com el temps que dura la transferència de la dada de la cache a la CPU (o en sentit invers). En cas de fallada, al temps que tarda la comprovació (i que suposarem igual a t_h) caldrà afegir-hi un *temps de penalització* extra t_p. El temps de penalització pot variar segons l'organització de la cache, però generalment inclou primer copiar el bloc de la memòria principal a la cache i després transferir la dada de la cache a la CPU (o en sentit invers):

$$t_{acces} = t_h \qquad \text{(en cas d´encert)}$$

$$t_{acces} = t_h + t_p \qquad \text{(en cas de fallada)}$$

1.5 La jerarquia de memòria

Recapitulant una mica els conceptes vistos en aquesta secció, resulta convenient considerar el subsistema de memòria del computador com una jerarquia de nivells, tal i com es mostra a la Figura 6.4.[3] Els nivells superiors proporcionen velocitat d'accés mentre que els inferiors proporcionen capacitat d'emmagatzematge. Cada nivell conserva solament les dades del nivell inferior que tinguin més probabilitat de ser accedides en el futur.

L'objectiu és accedir a les dades amb el menor temps possible explotant un determinat grau de localitat en cada un dels nivells. En les darreres seccions d'aquest tema veurem que, a mesura que segueix creixent el *gap* entre processador i memòria, el nivell de cache es subdivideix en múltiples subnivells, a fi de mitigar els cada cop més llargs temps de resposta (en termes relatius) dels nivells inferiors de la jerarquia.

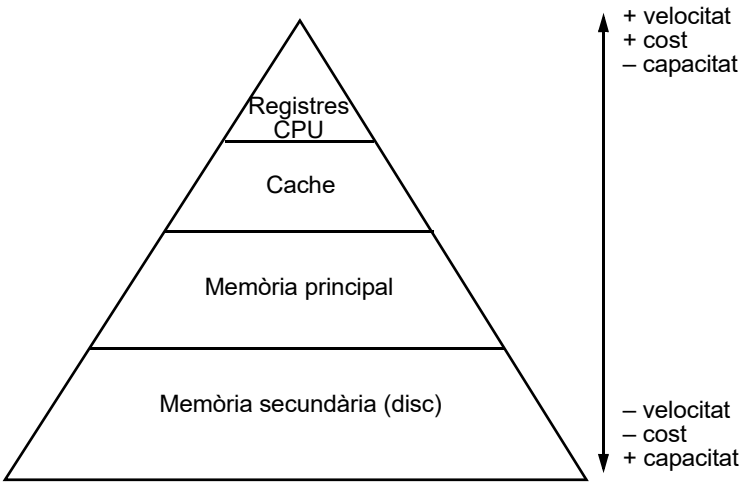


Figura 6.4. La jerarquia de memòria

3. Teòricament (obviant el cost), es podria construir tota la memòria en un sol nivell en tecnologia ràpida, però aquesta deixaria de ser tan ràpida degut que el temps d'accés augmenta amb la grandària.

2. Disseny bàsic d'una cache

2.1 Organització de la memòria en blocs

La cache no guarda dades soltes sinó blocs sencers de mida fixa *B* bytes, essent *B* una potència de 2 (mides típiques són entre 16 i 128 bytes). L'espai d'adreçament de memòria es subdivideix lògicament en blocs de *B* bytes, de manera que l'adreça inicial de cada bloc és un múltiple de *B*. Podem assignar a cada bloc de la memòria principal un identificador únic, simplement numerant correlativament tots els blocs de memòria des de zero. Aquest identificador s'anomena *número de bloc* (o també *adreça de bloc*), i li serveix a la cache per identificar els blocs que té guardats.

Vegem amb un simple exemple com funcionarien els encerts (hits) i fallades (misses) d'una cache quan la CPU llegeix successivament els 16 primers bytes d'una memòria. Suposem que els blocs de la MP (columna esquerra) són de mida *B*=4 bytes, i estan numerats correlativament: el bloc 0 conté els 4 primers bytes, el bloc 1 conté els 4 bytes següents, etc. Suposem que la cache (columna dreta) té capacitat per guardar sols 2 blocs, i està inicialment buida:

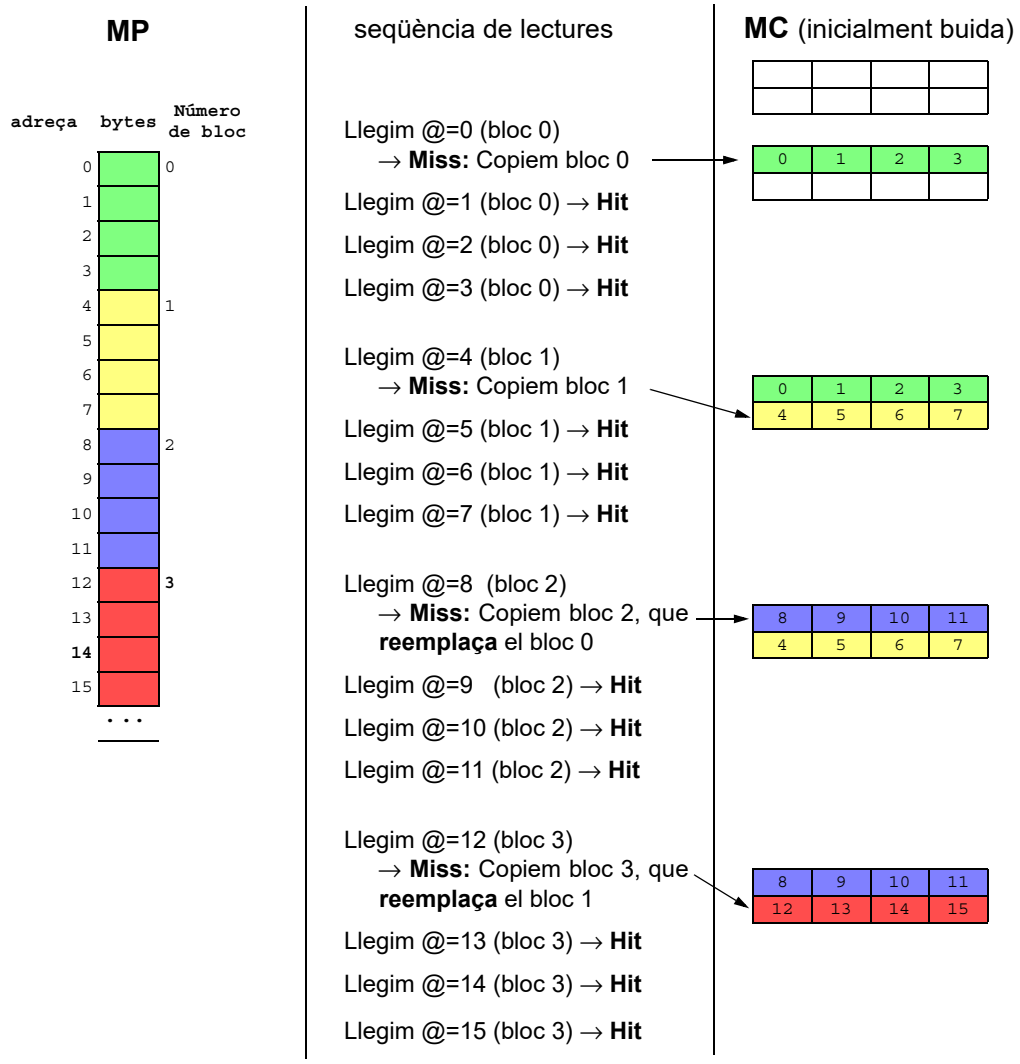


Figura 6.5. Exemple de funcionament de la cache amb una seqüència de 16 lectures

2.2 Càlcul del *num_bloc* i l'*offset* per a una adreça *A*

El *número de bloc* és el que resulta de dividir l'adreça inicial del bloc entre la mida *B*. Naturalment, en tractar-se de divisió entera, s'obté el mateix número per a qualsevol de les adreces *A* de dins el bloc:

$$num\_bloc = A \text{ div } B$$

La cache ha de cercar aquest bloc entre els que té emmagatzemats. Si el troba, ha d'ubicar la dada que busca dins del bloc. Si anomenem *offset* a la distància *en bytes* entre l'adreça inicial del bloc (que és un múltiple de *B*) i l'adreça *A*, l'*offset* és el residu (o mòdul) de dividir l'adreça *A* entre la mida B:

$$offset = A \text{ mod } B$$

Alguns cops resulta convenient definir *en words* la distància entre l'adreça inicial del bloc i l'adreça *A*, i l'anomenem *word_offset*. Per al cas de words de 4 bytes, serà

$$word\_offset = offset / 4$$

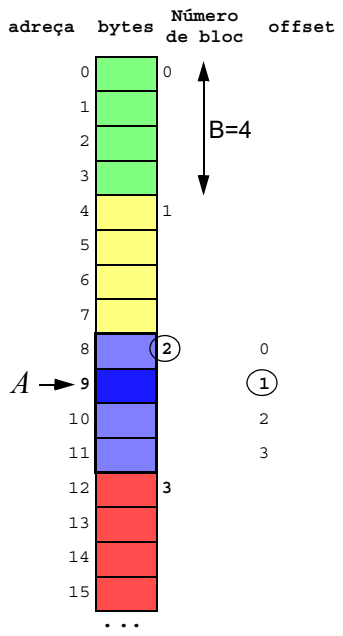


Figura 6.6. Número de bloc (2) i offset (1) del byte ubicat a l'adreça A=9 de MP.

Càlcul del *num_bloc* i l'*offset* en binari

Essent $B=2^n$, les operacions div i mod impliquen una simple descomposició de bits de l'adreça *A*. Vegem-ho amb un exemple, amb adreces de 32 bits com les de MIPS. Suposem que els blocs són de mida *B*=16 bytes i els words de 4 bytes. Si accedim a l'adreça **A=0x100100F8**, el *número de bloc* és

$$num\_bloc = A \text{ div } 16 = 0x100100F$$

Si llegim un byte (amb `lb`), aquest està a 8 bytes de distància de l'inici del bloc

$$offset = A \text{ mod } 16 = 8 = 1000_2$$

Si llegim un word (amb `lw`), aquest està a 2 words de distància de l'inici del bloc

$$word\_offset = offset / 4 = 2 = 10_2$$

Vegem a la següent figura com tant el *num_bloc*, com l'*offset* o el *word_offset* s'obtenen d'una simple descomposició dels bits de l'adreça:

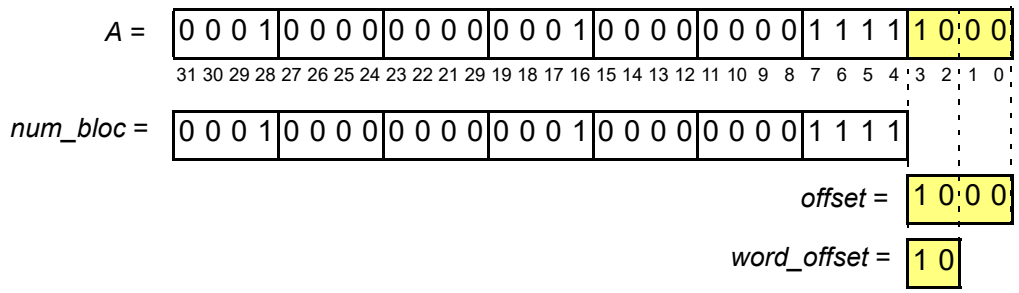


Figura 6.7. Descomposició dels bits de l'adreça en *numero de bloc* i *offset* (o *word_offset*)

2.3 Organització de la cache en línies, etiquetes i bits de validesa

La cache està organitzada com una taula on cada *entrada* (anomenada també *línia*) pot contenir un bloc de dades, o estar buida. Podem pensar en les línies de cache com calaixos, i que cada calaix pot guardar un sol bloc de dades. Com que diversos blocs de la memòria poden ocupar la mateixa línia al llarg de l'execució d'un programa, la cache ha de poder identificar-los quan li arriba un accés a l'adreça *A*, per saber si el bloc al qual pertany aquesta adreça és un dels que té emmagatzemats. Així doncs, per cada línia, la cache no sols guarda les dades del bloc sinó també un *bit de validesa* (V) i un identificador del bloc que anomenem *etiqueta* (*o tag*). El bit de validesa indica si la línia conté un bloc (V=1) o està buida (V=0). L'etiqueta identifica el bloc que està guardat, i està formada pels bits del *num_bloc* (o una part d'aquests, com veurem més endavant).

V	Etiqueta	Blocs de dades
1	0x100100F	bloc
1	0x7FFFF01	bloc
1	0x1001010	bloc
0		

Figura 6.8. Cache de 4 línies, amb les seves etiquetes i bits de validesa

2.4 Gestió d'una lectura a la cache

Per il·lustrar el funcionament bàsic d'una lectura a l'MC suposem novament que la mida de bloc és 16 bytes i que la CPU llegeix una dada a l'adreça A=0x100100F8. L'MC busca el seu número de bloc (0x100100F) entre les etiquetes de les línies vàlides.

En cas d'encert (hit): Suposem en primer lloc que el bloc està present a la cache (Figura 6.9.) i l'accés és un encert. Se succeeixen les següents accions:

1. L'MC consulta les etiquetes guardades i comprova que el número de bloc de l'adreça A coincideix amb l'etiqueta d'un dels blocs emmagatzemats. S'ha produït un encert (hit).
2. L'MC usa l'offset (0x8) com a selector per extraure del bloc guardat la dada sol·licitada, i la transfereix a la CPU.

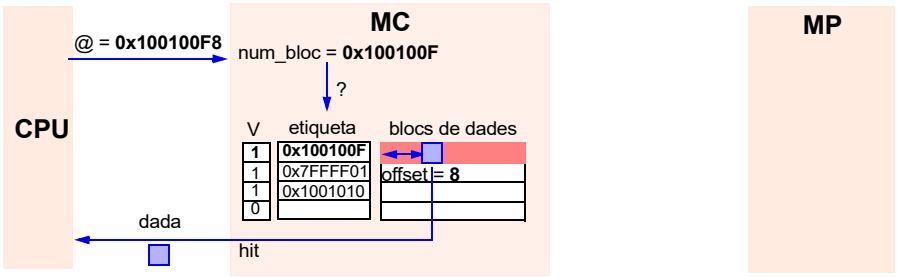
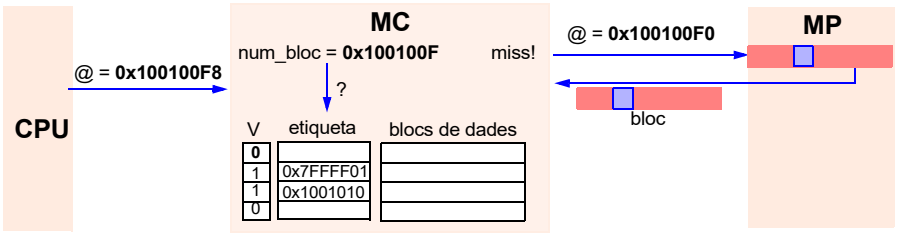


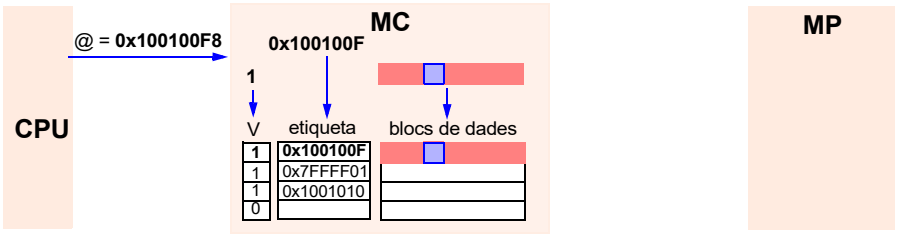
Figura 6.9. Encert en un accés a l'adreça A=0x100100F8: la dada es transfereix a la CPU

En cas de fallada (miss): Suposem que el bloc no hi és a la cache, i l'accés és una fallada. Succeeixen les següents accions:

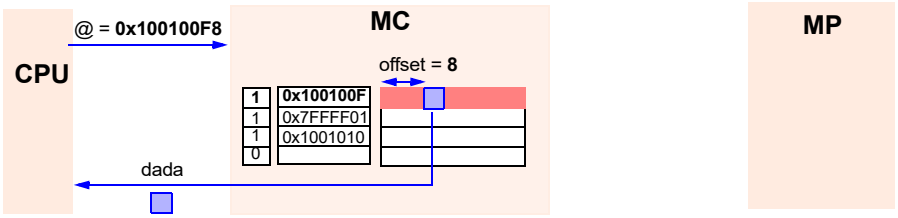
1. L'MC consulta les etiquetes guardades i comprova que el número de bloc de l'adreça A no coincideix amb cap etiqueta dels blocs emmagatzemats. S'ha produït un miss, i ho notifica a la CPU perquè es bloquegi fins que l'MC li transfereixi la dada
2. L'MC sol·licita a l'MP el bloc sencer que conté A, enviant-li l'adreça inicial del bloc, que és 0x100100F0 (s'obté posant a zero els bits d'offset de A).



3. L'MC decideix quina línia ha d'ocupar el bloc obtingut de l'MP i l'hi escriu. També escriu el número de bloc al camp d'etiqueta i posa el bit de validesa a 1.



4. L'MC finalitza l'accés procedint com si es tractés d'un encert: usa l'offset com a selector per extraure del bloc la dada sol·licitada, i la transfereix a la CPU.



2.5 Paràmetres de disseny de la cache

Per al disseny d'una cache definirem els següents paràmetres principals:

1. **Política d'emplaçament**: és l'algorisme que determina en quina línia o línies d'MC pot emmagatzemar-se un bloc. Per tant, també determina en quina línia o línies cal buscar una dada que es vol accedir.
2. **Política de reemplaçament:** és l'algorisme que determina quina línia s'ha d'eliminar per tal de fer espai a un nou bloc.
3. **Política d'escriptura**: determina com es fan les escriptures en cas d'encert, i com es fan en cas de fallada: escriure sols en l'MC? o sols en l'MP? o en els dos llocs?

A les properes seccions estudiarem diverses alternatives per a cada política. Totes elles s'han d'implementar en hardware, i per tant han de ser senzilles i ràpides d'executar.

3. Política d'emplaçament per Correspondència Directa

Ara sabem que l'MC emmagatzema blocs de dades accedits recentment. Però: en quina línia els guarda? com sap si un bloc ja hi és? La part del disseny d'una memòria cache que respon aquestes preguntes s'anomena *política d'emplaçament* (*placement policy*). La política d'emplaçament més senzilla és la de *correspondència directa* o *mapeig directe* (*direct-mapping*).

En una cache de correspondència directa amb capacitat per a NL línies de cache, cada bloc de memòria només pot ser emmagatzemat en una línia preestablerta. El bloc 0 es guarda a la línia 0, el bloc 1 a la línia 1, ... el bloc NL-1 a la línia NL-1. El bloc NL es guarda novament a la línia 0, el bloc NL+1 a la línia 1, etc. Generalitzant, podem dir que l'índex de la línia es determina fent una senzilla operació de mòdul amb el *num_bloc* i el número de línies NL:

$$index = num\_bloc \bmod NL$$

A fi de simplificar la implementació hardware de l'operació mòdul, NL serà sempre una potència de 2 (NL=2^{nl}). Llavors, donada l'adreça d'una dada podrem saber a quina línia de cache buscar-la simplement consultant els *nl* bits de menor pes del *num_bloc*.

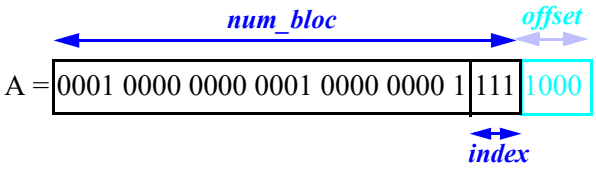


Figura 6.10. Suposem una cache de correspondència directa amb *NL*=8 línies (*nl*=3), i blocs de mida *B*=16 bytes (*b*=4). L'adreça A=0x100100F8 pertany al bloc: *num_bloc* = A div 16 = 0x100100F. Aquest bloc es guardarà a la línia: *index* = *num_bloc* mod 8 = 7

Observem que en una cache de correspondència directa tots els blocs de MP que es mapegen a la mateixa línia tenen per definició el mateix *índex*, és a dir els *nl* bits de menor pes de *num_bloc*. Per tant, l'*etiqueta* que identifica el bloc guardat en aquesta línia pot prescindir d'aquests *nl* bits i formar-se amb la resta de bits del *num_bloc*.

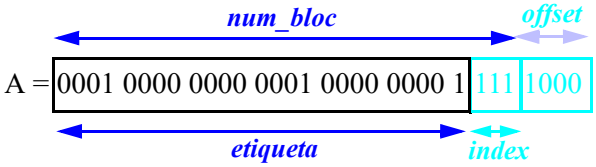


Figura 6.11. Els *nl*=3 bits de menor pes de *num_bloc* no formen part de l'*etiqueta*

Recapitulant, podem desglossar de la següent manera els bits d'una adreça en tres camps de bits: etiqueta, índex i offset:

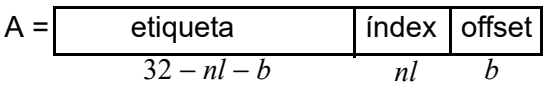


Figura 6.12. Desglossament dels bits de l'adreça en *etiqueta*, *índex* i *offset* per a una cache de NL=2^{nl} línies i amb blocs de mida B=2^{b} bytes

3.1 Diagrama de blocs d'una cache de correspondència directa (lectura)

Podem resumir l'operació bàsica que té lloc durant una lectura en una cache de correspondència directa amb el diagrama hardware de la Figura 6.13. La CPU envia una adreça a la MC, i aquesta en desglossa l'*etiqueta*, l'*índex* de línia i l'*offset* dins el bloc.

Els bits d'índex de l'adreça es connecten a l'entrada d'un decodificador per seleccionar una i solament una línia de la cache. La cache llegeix tots els bits de la línia seleccionada (en vermell, al dibuix): Etiqueta, bit de validesa V i Bloc de dades. Els bits d'offset de l'adreça es connecten a un multiplexor per seleccionar la dada sol·licitada d'entre totes les del bloc. Els bits d'etiqueta de l'adreça es comparen amb l'etiqueta guardada a la línia seleccionada per saber si el bloc que conté correspon al bloc de MP al qual pertany l'adreça sol·licitada. Si són iguals i el bit de validesa val 1, llavors es notifica un encert (hit), i es transfereix a la CPU la dada seleccionada pel multiplexor. Observa que la MC fa la comprovació de l'etiqueta i la lectura de la dada en paral·lel.

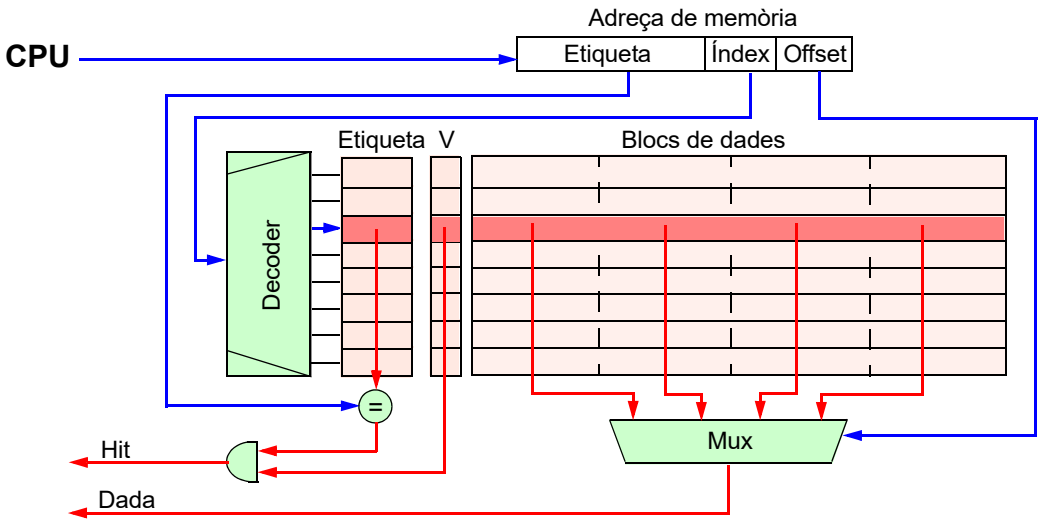


Figura 6.13. Esquema d'una lectura en una cache de correspondència directa

3.2 Impacte de la grandària del bloc en la taxa de fallades i en el temps d'accés

En el moment de dissenyar una cache, el factor determinant sol ser la capacitat total. A major capacitat, més dades podrà emmagatzemar, millorant així la probabilitat d'encert. A l'hora de decidir la llargada dels blocs, en principi sembla lògic pensar que tenir blocs més llargs permet treure major profit de la localitat espacial, i reduir així la taxa de fallades. Però donada una capacitat de cache fixa, si fem els blocs el doble de llargs, haurem de reduir el número de línies a la meitat, i no està clar quin serà l'efecte final sobre la taxa de fallades.

Els dissenyadors de memòries determinen la grandària òptima del bloc empíricament mitjançant experiments amb un simulador capaç d'interpretar les instruccions d'un conjunt de *benchmarks* (programes representatius). En cada experiment, el simulador modela una possible configuració de cache i n'obté mesures de rendiment. La Figura 6.14. mostra la taxa de fallades (a l'eix d'ordenades) per a blocs de diferents llargades (eix

d'abscises), i per a diferents capacitats de cache (de 4KB a 256KB). Els experiments s'han fet amb el benchmark SPEC92.

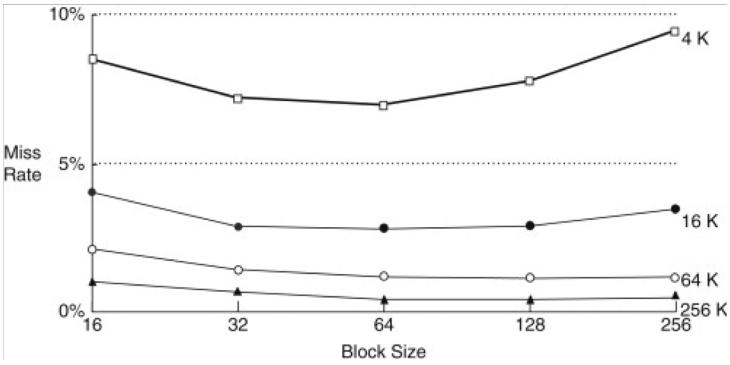


Figura 6.14. Taxa de fallades amb diferents llargades de bloc i capacitats de cache, mesurada per al benchmark SPEC92 (Font: D.A.Patterson and J.L.Hennessy, *Computer Organization and Design, the Hardware/Software Interface*, 5th. ed., Morgan Kaufmann, 2014).

A la figura s'observa que a mesura que augmenta la mida del bloc disminueix la taxa de fallades, ja que la cache treu major profit de la localitat espacial. Però s'observa també que la taxa de fallades comença a augmentar a partir d'un cert punt, quan la mida de bloc esdevé una fracció significativa de la capacitat total de la cache, perquè llavors el nombre de línies disponibles a la cache és massa petit i es produeix una competència important entre els blocs per ocupar-les. Com a resultat, un bloc pot ser expulsat de la cache abans que totes les seves dades hagin estat accedides, impedint així que es pugui aprofitar la localitat espacial existent.

A més de l'impacte que té sobre la taxa de fallades, la grandària del bloc té també un impacte sobre el temps que es triga en transferir blocs entre els diferents nivells de la jerarquia de memòria, i per tant en el *temps de penalització de les fallades*, que estudiarem més endavant.

4. Polítiques d'escriptura

Fins ara, per simplicitat, hem suposat que tots els accessos a memòria (a dades o instruccions) eren lectures. Anem a veure ara com afecta al funcionament de la memòria cache el fet de considerar també les escriptures. Copiar blocs de dades de la memòria principal a la memòria cache provoca que, temporalment, hi hagi dades duplicades (redundància de dades). Això no és cap problema si les dades no canvien mai, però si les escriptures modifiquen tan sols la versió que tenim en MC haurem d'assegurar-nos que les modificacions no es perden (per exemple quan reemplacem un bloc d'MC), i que les lectures posteriors sempre obtenen la versió més actualitzada de les dades. És a dir, haurem de *gestionar la coherència* de les dades.

Quan la CPU sol·licita una escriptura a memòria (per exemple durant l'execució d'una instrucció `sw`) es pot produir un encert o bé una fallada, i per a cada cas existeixen diverses alternatives de gestió de l'escriptura:

4.1 Polítiques d'escriptura en cas d'encert (gestió de la coherència)

Quan una escriptura produeix un encert de cache, ens trobem davant la disjuntiva entre escriure només a la cache, i que hi hagi dues versions diferents de la mateixa dada, o escriure a la cache i a la memòria principal, assegurant així que ambdues còpies de la dada són iguals. Aquestes dues possibilitats donen lloc a dues maneres diferents de gestionar la coherència de les dades:

- **Escriptura immediata** (*write-through*): Si hi ha un encert, s'escriu la dada simultàniament a la cache i a l'MP. Ambdues còpies romanen iguals, i el funcionament bàsic de la cache no es veu afectat. Quan calgui reemplaçar el contingut d'una línia, es podrà fer directament, rebutjant el bloc que conté ja que es té la certesa que a la memòria principal es conserva la mateixa informació.
- **Escriptura retardada** (*write-back* o *copy-back*): Si hi ha un encert, s'escriu la dada únicament a la cache. Llavors hi haurà al mateix temps dues versions diferents de la dada, una a la memòria cache i una a la memòria principal.

  Quan posteriorment calgui reemplaçar un bloc (ja sigui per una fallada de lectura o d'escriptura), s'haurà de saber si aquest ha estat modificat o no. Amb aquesta finalitat, afegirem un bit a cada línia de la memòria cache, el bit D (*dirty bit*) que valdrà 1 si el bloc ha estat modificat (Figura 6.15.). Si el bloc que cal reemplaçar conté una dada modificada, caldrà procedir de la següent manera:

  1. Copiar **tot el bloc** modificat (el que volem reemplaçar) a la memòria principal, per preservar les modificacions que contingui.
  2. Transferir el nou bloc (el que volem accedir) de la memòria principal a la cache, sobreescrivint el bloc antic.

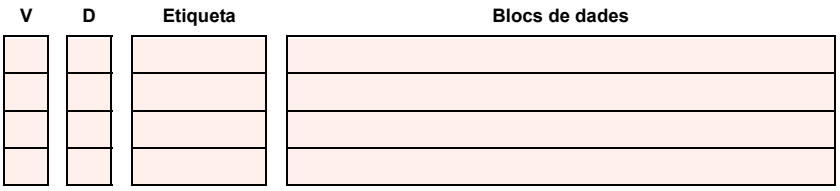


Figura 6.15. Cache d'escriptura retardada, amb un Dirty Bit associat a cada línia

4.2 Polítiques d'escriptura en cas de fallada

Quan es produeix una fallada de lectura sempre s'ha de copiar el bloc corresponent d'MP a l'MC. En el cas de les escriptures, hi haurà dues possibles alternatives (de vegades anomenades polítiques d'escriptura en cas de fallada o *write-miss policies*):

- **Escriptura amb assignació** (*write allocate*): Copiem el bloc de MP a la memòria cache, de la mateixa manera que ho faríem amb una lectura, i després prosseguim escrivint la dada en MC.
- **Escriptura sense assignació** (*no-write allocate*): NO copiem el bloc de MP a MC, sinó que escrivim la dada a l'MP directament, com si no hi hagués cache.[4]

4.3 Polítiques d'escriptura combinant opcions per a encerts i fallades

Combinant una política de gestió de la coherència (escriptura immediata o retardada) amb una política en cas de fallada (escriptura amb o sense assignació) obtenim el que s'anomena la política d'escriptura (*write policy*). En el context de l'assignatura treballarem principalment tres combinacions o polítiques diferents:

1. Escriptura immediata amb assignació
2. Escriptura immediata sense assignació
3. Escriptura retardada amb assignació

Al laboratori, tan sols usarem l'escriptura immediata amb assignació ja que és l'única que implementa el simulador Mars.

4.4 Impacte de la política d'escriptura en el rendiment

Per un costat, el contingut de la cache —i per tant la taxa de fallades— no és el mateix si la *política d'escriptura en cas de fallada* és amb assignació o sense assignació. La influència d'aquesta política sobre la taxa de fallades —i per tant sobre el rendiment— dependrà del programa o conjunt de programes que s'executi. En canvi, la política escollida per a la *gestió de la coherència* (escriptura immediata o retardada) no altera per a res la taxa de fallades ja que, sigui quina sigui l'alternativa escollida, a la cache sempre hi haurà els mateixos blocs, i l'única diferència entre les dues alternatives és "en quin moment" s'actualitza la informació a la memòria principal. On sí que la gestió de la coherència pot tenir un impacte sobre el rendiment és en el temps d'accés en cas d'encert o en el de penalització de les fallades. Vegem-ho en més detall:

- **Escriptura immediata**: Mantenir la mateixa versió de les dades a MP i MC simplifica molt les coses però pot implicar un problema de rendiment si a cada escriptura hem d'esperar que s'escrigui a l'MP. Per aquest motiu, normalment es fa servir un *buffer d'escriptura*, on queden emmagatzemades les escriptures pendents de portar a MP. Un cop escrites les dades a l'MC i al buffer, el proces-

4. En els processadors actuals és rar que una cache apliqui la política d'escriptura sense assignació genèricament a tots els stores. La intenció d'aquesta política és evitar contaminar la cache amb blocs que no s'espera que siguin llegits pròximament, com succeeix quan es copien o inicialitzen extenses regions de memòria, tals com buffers d'entrada/sortida (com els usats en gràfics), o quan l'adreça que s'escriu pertany a un dispositiu mapejat en memòria. Per a aquests casos específics, alguns processadors inclouen en el seu repertori certes instruccions de store especials que apliquen la política d'escriptura sense assignació, a diferència de la resta de stores normals.

sador pot continuar l'execució mentre el buffer les escriu en MP, en segon pla (un cop escrites, s'eliminen del buffer per tal de fer lloc a dades posteriors). Si quan s'executa un store el buffer està ple, la CPU ha d'aturar l'execució i esperar fins que hi hagi lloc. Si el ritme d'escriptures promig de la CPU és major que el d'escriptures del buffer en MP, aquest s'omplirà inevitablement i la CPU s'haurà d'aturar. Però encara que aquest ritme sigui menor, també es poden produir aturades de la CPU si les escriptures es presenten en ràfegues que saturen el buffer. En aquest cas, les aturades es poden minimitzar dimensionant la capacitat del buffer de forma convenient.

Per un altre costat, un aspecte positiu de l'escriptura immediata amb assignació és que permet accedir a les etiquetes per comprovar si hi ha encert o fallada i fer l'escriptura en el mateix cicle de rellotge, com hem suposat fins ara per a les lectures. Si es produeix una fallada haurem fet una escriptura debades sobre un bloc diferent al que volíem escriure, però no importa ja que la informació que conté es troba ja en MP, i el bloc pot ser reemplaçat directament.

- **Escriptura retardada**: Mantenir diferents versions de les dades a MP i MC afecta al rendiment de diverses formes. Per un costat, l'escriptura retardada té l'avantatge de suportar un temps d'accés més curt, ja que sols escriu en MC. A més a més, redueix el nombre total d'escriptures en MP, ja que múltiples escriptures del mateix bloc en MC poden comportar una única escriptura en MP.

  Però per un altre costat, quan una fallada de cache (de lectura o escriptura) requereix el reemplaçament d'un bloc i aquest conté dades anteriorment modificades (bit D a 1), caldrà copiar prèviament tot aquest bloc a la memòria principal, allargant així el temps de penalització per fallada. Per reduir aquest temps, es pot fer servir també un buffer específic, on queda emmagatzemat el bloc pendent de ser escrit posteriorment en MP, en segon pla, de manera que es pot iniciar immediatament la còpia del nou bloc de MP a l'MC i completar l'escriptura de la dada.

  Un altre inconvenient de l'escriptura retardada és que no podem fer l'escriptura en el mateix cicle de rellotge que comprovem si hi ha encert o fallada, com sí que passa amb l'escriptura immediata, ja que les línies d'MC poden contenir informació que encara no és present a MP i no podem escriure-hi fins estar segurs que contenen el bloc que volem modificar. Retardar l'escriptura al cicle següent afecta al rendiment perquè allarga el temps d'accés per a totes les escriptures. Per resoldre aquest problema s'acostuma a fer servir un petit buffer d'escriptura on s'escriu la dada de manera provisional, en el mateix cicle que la comprovació de l'etiqueta, i es difereix l'escriptura de la dada a la MC fins al cicle següent, però sense fer esperar a la CPU en cas d'encert.

Totes les solucions que impliquen buffers d'escriptura requereixen diverses complicacions del hardware per assegurar la correcció dels programes. Amb el propòsit de simplificar la realització d'exercicis en aquest curs d'EC, definirem a la secció 5.1 un Model de Temps que concretarà quina configuració de hardware assumirem per a cada una de les diferents polítiques d'escriptura (per exemple quins buffers suposarem), i els temps de penalització de fallada corresponents.

4.5 Exemple 1: Cache d'escriptura immediata amb assignació

- Suposem un processador de 8 bits ⇒ rang d'adreces de 0 a 255
- Mida de bloc B = 4 bytes ⇒ 2 bits d'*offset* + 6 bits de *num_bloc*
- Cache de 2 línies (corresp. directa) ⇒ 1 bit d'*índex*

Fem una seqüència inicial de 5 lectures (loads):

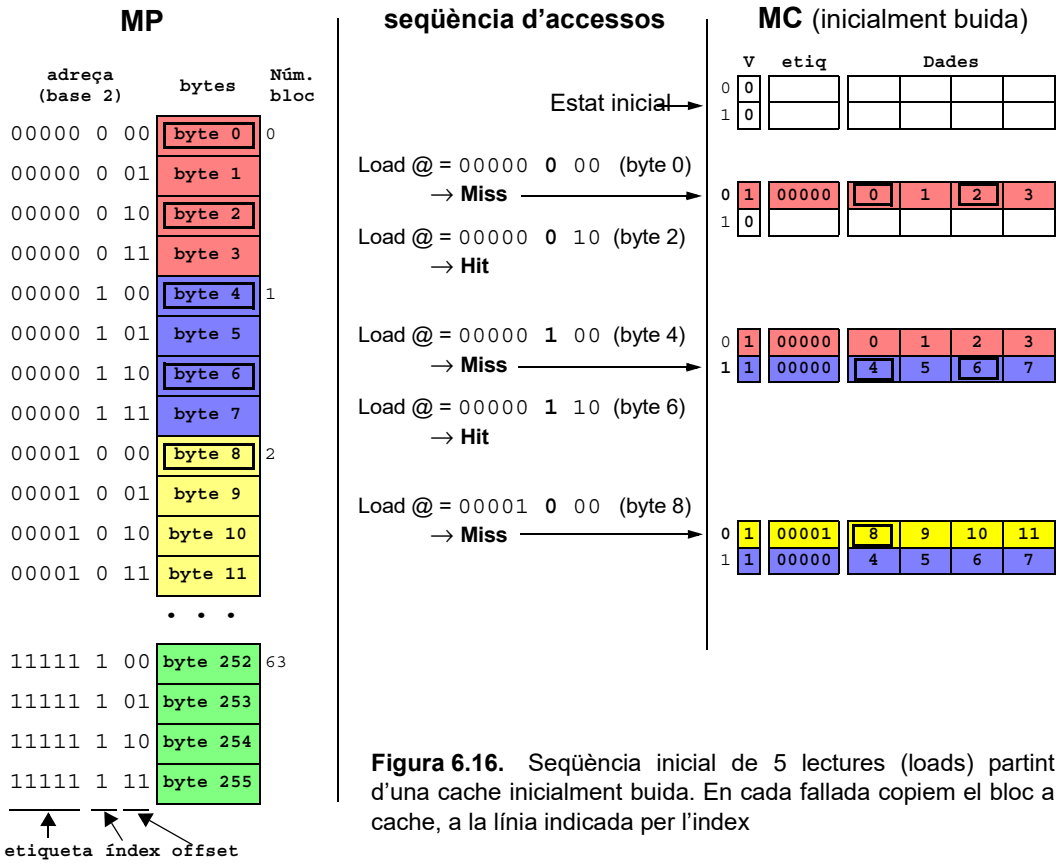


Figura 6.16. Seqüència inicial de 5 lectures (loads) partint d'una cache inicialment buida. En cada fallada copiem el bloc a cache, a la línia indicada per l'index

A continuació, fem 2 escriptures (stores) sobre la mateixa cache: amb la política d'escriptura *immediata amb assignació*, s'escriu la dada en MC i en MP. I en cas de fallada, s'ha de copiar abans el bloc de MP a MC.

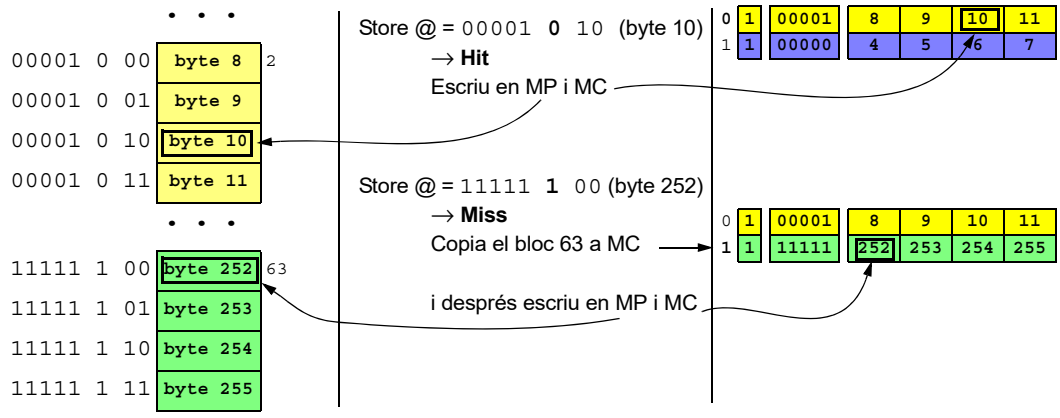


Figura 6.17. Escriptura immediata amb assignació: 2 escriptures (stores), partint de la cache inicialitzada a l'exemple anterior

4.6 Exemple 2: Cache d'escriptura immediata sense assignació

Suposem que la cache ha quedat inicialitzada amb les 5 lectures de la Figura 6.16. (amb els blocs 1 i 2 carregats). Suposem que fem les mateixes 2 escriptures de l'exemple anterior, però ara la política d'escriptura és *immediata sense assignació*: En cas d'encert, s'escriu tant en MC com en MP. Però en cas de fallada, solament s'escriu en MP.

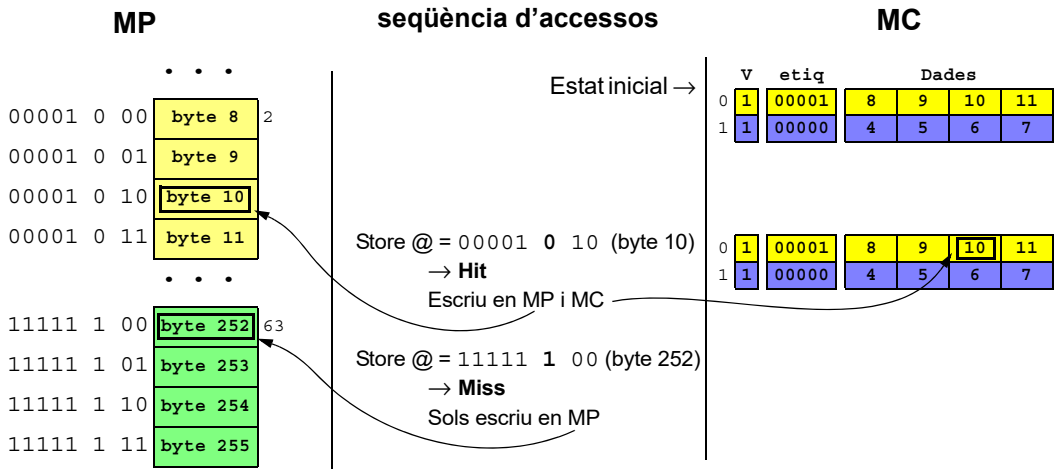


Figura 6.18. Escriptura immediata sense assignació: seqüència de 2 escriptures (stores)

4.7 Exemple 3: Cache d'escriptura retardada (amb assignació)

Suposem que la cache ha quedat inicialitzada amb les 5 lectures de la Figura 6.16. (amb els blocs 1 i 2 carregats, i ambdós amb el bit Dirty a 0). Suposem que fem 2 escriptures i 1 lectura, però ara la política d'escriptura és *retardada amb assignació*: en qualsevol fallada, si s'ha de reemplaçar un bloc modificat, abans cal actualitzar-lo en MP.

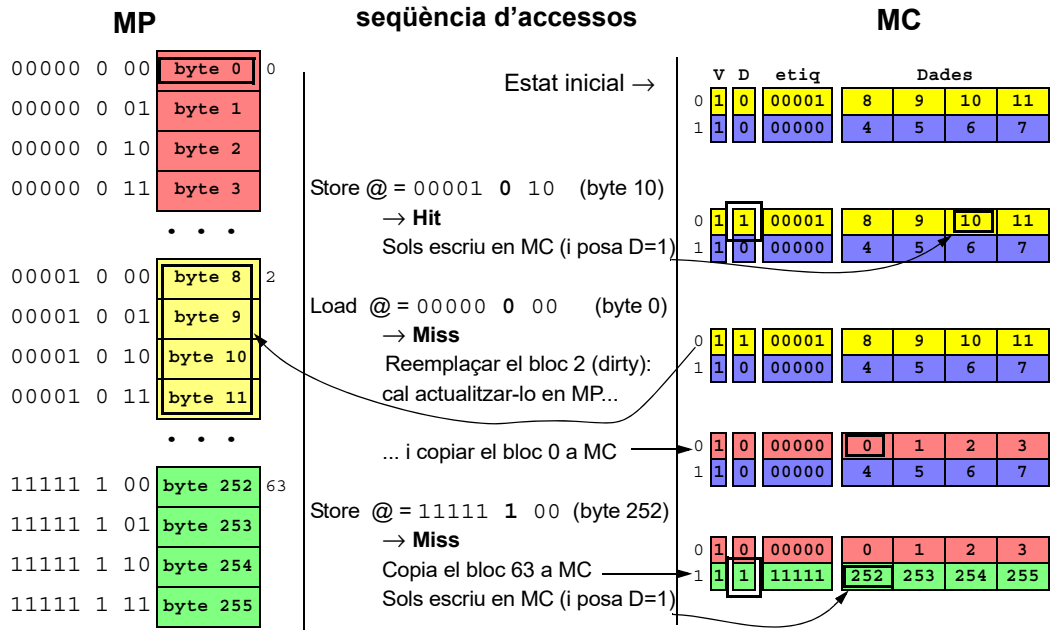


Figura 6.19. Escriptura retardada amb assignació: 1 lectura (load) i 2 escriptures (stores)

5. Mesures de rendiment

Als apartats anteriors, s'han descrit múltiples opcions de disseny d'una memòria cache (polítiques d'escriptura, buffers, etc.), però necessitem un criteri per avaluar-les que es basi en el rendiment. En primer lloc definirem el *temps d'accés a memòria* ($t_{accés}$), un paràmetre constructiu que caracteritza el comportament de la cache, i que tindrà el major impacte en el càlcul de les dues mètriques principals de rendiment: el *temps d'accés mitjà a memòria* (t_{am}) i el *temps d'execució* (t_{exe}).

5.1 Temps d'accés a memòria. Un model simplificat

Podem definir el *temps d'accés a memòria* ($t_{accés}$) com el temps que transcorre entre el moment que la CPU fa una sol·licitud d'accés (lectura o escriptura) al subsistema de memòria i el moment que la dada és efectivament transferida, de forma que la CPU pot prosseguir l'execució. Durant aquest temps, la CPU ha de restar a l'espera.

A fi de facilitar la realització d'exercicis en aquest curs d'EC, hem especificat una sèrie de configuracions de cache estàndard, per a les quals hem definit un model de temps simplificat que ens permetrà determinar de manera unívoca quants cicles de rellotge trigarà una referència a memòria en cadascuna de les diferents situacions.

En cas d'encert (*hit*), el temps d'accés a memòria és $t_{accés} = t_h$. Aquest temps inclou l'accés a les etiquetes per comprovar que la referència és un encert, així com el temps necessari per fer la lectura o escriptura de la dada a la cache[5]. Aquest temps és major com més grans siguin la capacitat de la cache així com el grau d'associativitat de l'algorisme d'emplaçament (que s'estudiarà a la secció 6), però per als exercicis d'aquest curs suposarem que és un simple paràmetre constructiu fix, ja donat.

En cas de fallada, el temps d'accés també inclou el temps necessari per comprovar amb l'etiqueta que efectivament es tracta d'una fallada, i que suposarem igual al *temps d'encert* (t_h), però en aquest cas caldrà afegir-hi un *temps de penalització* extra (t_p) necessari per resoldre la referència en accedir al següent nivell de la jerarquia. És a dir,

$$t_{acces} = t_h + t_p$$

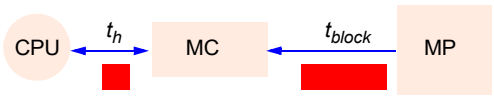
El temps de penalització (t_p) està format per dues parts: la primera pot incloure la còpia de blocs de MP a MC o viceversa, depenent de la política d'escriptura[6]; la segona consisteix a transferir la dada de MC a CPU o a la inversa. Per tal de modelar la primera part, definim t_{block} com el temps que cal per transferir un bloc entre MC i MP. I suposarem que la segona part tarda sempre t_h (igual al temps d'encert). A continuació examinem cas per cas el desglossament del temps de penalització per a diverses polítiques d'escriptura.

5. En la major part de casos podem suposar que comprovació i escriptura es fan simultàniament. Tal com s'ha discutit a la secció 4.4, per a la política d'escriptura retardada l'escriptura s'ha de fer en el cicle següent que la comprovació. Però si suposem que fem l'escriptura en un petit buffer temporal, podem assumir també en aquest cas que escriptura i comprovació d'etiqueta són simultanis.

6. Per aquelles configuracions de memòria cache amb política d'escriptura immediata considerarem l'existència d'un buffer d'escriptura amb capacitat il·limitada on queden emmagatzemades les escriptures pendents de portar a MP. També considerarem que cap referència a memòria posterior entra en conflicte amb escriptures pendents en aquest buffer. Cal adonar-se que el contingut d'aquest buffer s'escriu a MP en paral.lel amb l'execució de les instruccions que venen a continuació de l'accés a memòria.

1. Escriptura immediata amb assignació

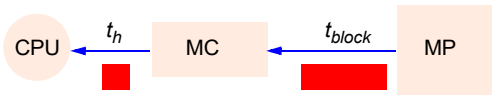
- Fallada de lectura o d'escriptura (la penalització és igual en els dos casos): cal copiar el bloc de MP a MC (t_{block}) i transferir la dada a/de la CPU (t_h)



$$t_p = t_{block} + t_h$$

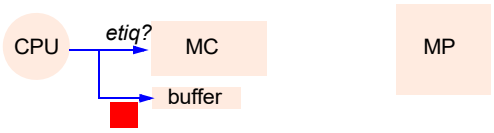
2. Escriptura immediata sense assignació

- Fallada de lectura: cal copiar el bloc de MP a MC (t_{block}) i servir la dada a la CPU (t_h)



$$t_p = t_{block} + t_h$$

- Fallada d'escriptura: s'escriu la dada al buffer[6] durant la comprovació inicial de l'etiqueta, de manera que no cal cap temps extra de penalització



$$t_p = 0$$

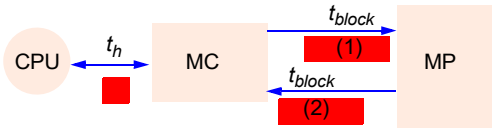
3. Escriptura retardada (amb assignació)

La penalització és igual per a fallades de lectura o d'escriptura. En realitat, depèn de si el bloc que s'ha de reemplaçar està modificat o no:

- Fallada que reemplaça un bloc no-modificat: cal copiar el bloc de MP a MC (t_{block}) i transferir la dada a/de la CPU (t_h)

$$t_p = t_{block} + t_h$$

- Fallada que reemplaça un bloc modificat: primer cal copiar el bloc a reemplaçar en MP (t_{block}), ja que serà l'única còpia que en quedarà; a continuació, cal copiar el nou bloc de MP a MC (t_{block}); i finalment transferir la dada a/de la CPU (t_h)



$$t_p = 2 \cdot t_{block} + t_h$$

Taula resum: la taula a continuació resumeix el temps de penalització per a les tres polítiques d'escriptura, segons el model de temps que hem definit per defecte:

t_p	Immediata amb assignació	Immediata sense assignació
Lectura	$t_{block} + t_h$	$t_{block} + t_h$
Escriptura	$t_{block} + t_h$	0

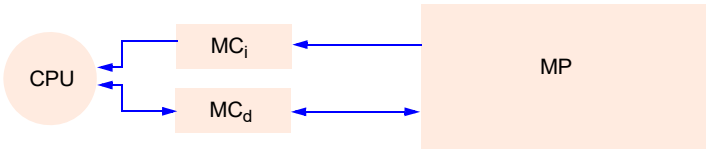
t_p	Retardada amb assignació
Bloc NO modificat	$t_{block} + t_h$
Bloc modificat	$2 \cdot t_{block} + t_h$

5.2 Temps d'accés mitjà a memòria (t_{am})

Com hem vist, el temps d'accés d'una referència a memòria és diferent en cada cas segons si és un encert o una fallada, si és una lectura o escriptura, etc. A fi d'obtenir un únic valor representatiu d'aquest temps que ens serveixi per avaluar el rendiment de la cache usarem el *temps d'accés mitjà a memòria* (t_{am}) també anomenat *average memory access time* (AMAT), el qual en calcula el promig, ponderant el temps de cada tipus d'accés pel seu nombre d'ocurrències al llarg d'un programa o conjunt de programes[7]. En una primera aproximació, podem fer el promig ponderant el temps d'encert (t_h) per la taxa d'encerts ($h = 1 - m$) i el temps de fallada ($t_h + t_p$) per la taxa de fallades (m):

$$\begin{aligned} t_{am} &= (1 - m)\cdot t_h + m\cdot(t_h + t_p) \\ &= t_h + m\cdot t_p \end{aligned} \qquad [1]$$

En algunes arquitectures, la CPU està connectada a dues caches diferents. Una cache d'instruccions (MC_i) que és de sols lectura i s'usa durant l'etapa de fetch per a llegir les instruccions de memòria; i una cache de dades (MC_d) que s'usa amb els loads i stores per a llegir o escriure dades de memòria[8].



En aquest cas, per a promitjar el temps d'accés a memòria haurem de ponderar adequadament els temps de penalització causats per cada cache (que denotarem t_{pi} i t_{pd}) segons la fracció d'accessos a cada una i llurs taxes de fallades corresponents (que denotarem m_i i m_d). Si és n_{dat} el nombre d'accessos a MC_d (igual al nombre de loads i stores) i n_{ins} el nombre d'accessos a MC_i (tants com instruccions), la suma total de referències a memòria és doncs $n_{refs} = n_{dat} + n_{ins}$. Llavors, la fracció d'accessos a MC_d és n_{dat}/n_{refs}, que també podem escriure com $(n_{refs} - n_{ins})/n_{refs}$, i la fracció d'accessos a MC_i és n_{ins}/n_{refs}. El temps d'accés mitjà queda

$$t_{am} = t_h + (n_{ins}/n_{refs}) \cdot m_i \cdot t_{pi} + (n_{refs} - n_{ins})/n_{refs} \cdot m_d \cdot t_{pd} \qquad [2]$$

És habitual que, per a un programa donat, el nombre de referències per instrucció ($n_r = n_{refs}/n_{ins}$) sigui un paràmetre estadístic conegut. Observem que n_r és sempre major que 1 ja que per a tota instrucció es realitza com a mínim un accés a memòria. Llavors podem reformular l'anterior expressió en funció d'aquest paràmetre

$$t_{am} = t_h + (1/n_r) \cdot m_i \cdot t_{pi} + (1 - 1/n_r) \cdot m_d \cdot t_{pd} \qquad [3]$$

Com ja hem vist a la secció 5.1, el temps de penalització no és igual per a totes les fallades, i això depèn de la política d'escriptura (naturalment, això no afecta a la cache d'instruccions). Així doncs, s'hauran de refinar les anteriors expressions [1], [2] i [3] per ponderar adequadament els diversos temps de penalització.

---

7. La pràctica de l'enginyeria acostuma a avaluar el rendiment d'un disseny de cache usant conjunts de programes estàndard (*benchmarks*) representatius del tipus de càrrega de treball al qual se sotmetrà el processador.

8. D'aquesta manera, cada cache es pot dissenyar de forma més específica a la seva funció, i fins i tot poden operar en paral·lel, en les arquitectures segmentades que s'estudiaran en cursos posteriors.

Si la política d'escriptura és immediata amb assignació, totes les fallades tenen idèntica penalització

$$t_p = t_{block} + t_h$$

En canvi, si la política d'escriptura és immediata sense assignació, la penalització depèn de si la fallada és una lectura ($t_{p\_lectura}$) o una escriptura ($t_{p\_escriptura}$). Així doncs, suposant que les escriptures representen una fracció p_e de totes les fallades a memòria[9]

$$\begin{aligned} t_p &= (1 - p_e)\cdot t_{p\_lectura} + p_e \cdot t_{p\_escriptura} \\ &= (1 - p_e)\cdot (t_{block} + t_h) + p_e \cdot 0 \\ &= (1 - p_e)\cdot (t_{block} + t_h) \end{aligned}$$

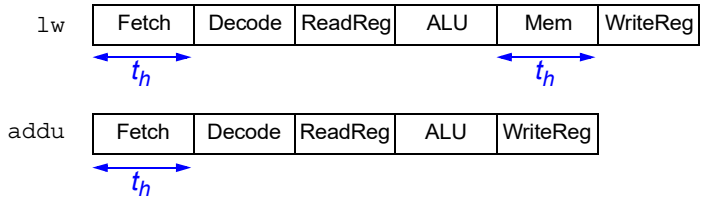
I si la política d'escriptura és retardada, la penalització depèn de si la fallada (ja sigui lectura o escriptura) causa el reemplaçament d'un bloc modificat ($t_{p\_bloc\_modificat}$) o bé d'un bloc no-modificat ($t_{p\_bloc\_no\text{-}modificat}$). Així doncs, suposant que els blocs reemplaçats modificats representen una fracció p_m de totes les fallades a memòria

$$\begin{aligned} t_p &= (1 - p_m)\cdot t_{p\_bloc\_no\text{-}modificat} + p_m \cdot t_{p\_bloc\_modificat} \\ &= (1 - p_m)\cdot (t_{block} + t_h) + p_m \cdot (2 \cdot t_{block} + t_h) \\ &= (1 + p_m)\cdot (t_{block} + t_h) \end{aligned}$$

5.3 Impacte de les fallades en el rendiment (t_{exe})

El temps d'accés mitjà és una mètrica útil per comparar l'eficiència de diferents alternatives de disseny d'una cache, però l'objectiu final és obtenir el màxim rendiment possible del sistema en conjunt. Per tant, ens interessa calcular com influeixen els diferents paràmetres de disseny de la cache en el rendiment, és a dir en el temps d'execució.

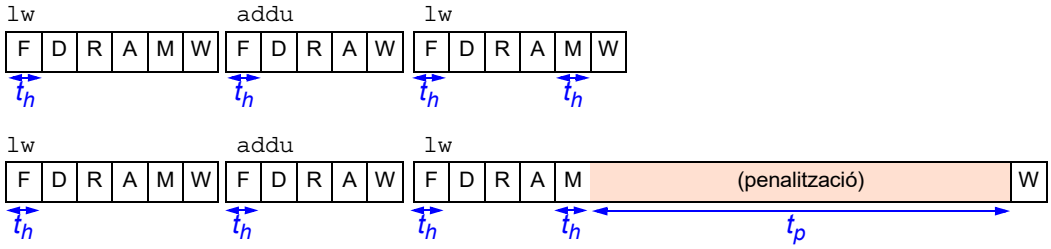
El temps d'accés a memòria constitueix una part del temps d'execució de la CPU. El següent diagrama il·lustra com es descomponen en el temps les etapes d'execució típiques d'instruccions com els loads (`lw`) o les sumes (`addu`): Fetch, Descodificació, Lectura de registres font, càlcul a la ALU, Accés a Memòria i Escriptura del registre destinació. En aquelles etapes on s'accedeix a memòria (Fetch, Mem), s'hi indica també el temps d'accés a memòria suposant que són encerts de cache ($t_{acces} = t_h$).



En el següent diagrama veurem dos exemples d'execució d'una seqüència de 3 instruccions: `lw`, `addu` i `lw`. En el primer exemple suposarem que s'executa amb una cache IDEAL, és a dir una cache on tots els accessos són màgicament encerts. El segon exemple mostra la mateixa seqüència amb una cache REAL, on alguns accessos causen fallades.

9. Si no s'especifica el contrari en un exercici, suposarem per simplificar que la proporció d'escriptures p_e és la mateixa entre les fallades a memòria que entre totes les referències a dades en conjunt, és a dir que és igual al nombre de stores dividit entre la suma de loads i stores.

Concretament, l'accés a la dada del segon load és una fallada, i es mostra ombrejat el corresponent temps de penalització (t_p).



Com es pot observar, les etapes assenyalades en blanc són iguals en els dos exemples. L'única diferència està en el temps de penalització extra causat per les fallades de cache. Aquest exemple ens servirà per il·lustrar la influència de les fallades de cache en el càlcul del temps d'execució. En primer lloc, definim alguns termes:

n_{ins}	=	nombre d'instruccions executades
n_{cicles_ideal}	=	cicles d'execució en el cas ideal (part blanca del diagrama)
n_{cicles_penal}	=	cicles de penalització per fallades (part ombrejada del diagrama)
n_{cicles}	=	$n_{cicles\_ideal} + n_{cicles\_penal}$ (cicles totals d'execució)
CPI_{ideal}	=	$n_{cicles\_ideal} / n_{ins}$
CPI_{penal}	=	$n_{cicles\_penal} / n_{ins}$
CPI	=	$n_{cicles} / n_{ins} = \text{CPI}_{ideal} + \text{CPI}_{penal}$
$n_{fallades}$	=	nombre de fallades a cache
n_{refs}	=	nombre de referències a memòria
n_r	=	n_{refs} / n_{ins} (referències a memòria per instrucció)
t_c	=	temps de cicle de rellotge

Recordem el temps d'execució

$$t_{exe} = n_{cicles} \cdot t_c = \text{CPI} \cdot n_{ins} \cdot t_c = (\text{CPI}_{ideal} + \text{CPI}_{penal}) \cdot n_{ins} \cdot t_c$$

Si refinem el càlcul de CPI_{penal}

$$\begin{aligned} \text{CPI}_{penal} &= n_{cicles\_penal} / n_{ins} \\ &= t_p \cdot n_{fallades} / n_{ins} \\ &= t_p \cdot m \cdot n_{refs} / n_{ins} \\ &= t_p \cdot m \cdot n_r \end{aligned}$$

Llavors, el temps d'execució quedarà

$$t_{exe} = (\text{CPI}_{ideal} + t_p \cdot m \cdot n_r) \cdot n_{ins} \cdot t_c$$

Observem que el terme CPI_{ideal} depèn del temps que tarden en processar-se les diverses etapes de l'execució (etapes en blanc al diagrama), i per tant de la microarquitectura de la CPU. En canvi, el terme CPI_{penal} (part ombrejada al diagrama) depèn de les fallades de cache en accessos a memòria. Podem millorar t_p i m amb un bon disseny de la cache, però també podem millorar m i n_r amb un compilador que optimitzi el programa reduint els accessos a memòria, o millorant la localitat.

Amb el temps, les millores de la microarquitectura de la CPU han reduït el CPI_{ideal} molt més ràpidament que la reducció del CPI_{penal} assolit per les millores en la cache. Però, d'acord amb la llei d'Amdhal, resulta imprescindible millorar aquesta component del temps d'execució per evitar que esdevingui el terme dominant si volem que les tècniques d'arquitectura o compilació proposades es tradueixin en augments de rendiment significatius del sistema. Vegem doncs algunes tècniques bàsiques que permetran reduir CPI_{penal} i millorar el rendiment de la cache:

- Com millorar la taxa de fallades m? $\Rightarrow$ Amb associativitat
- Com millorar el temps de penalització t_p? $\Rightarrow$ Amb caches multinivell

6. Millores de rendiment: Associativitat

Fins ara hem ubicat els blocs d'MP dins la memòria cache mitjançant la política d'emplaçament de *correspondència directa*. Aquest algorisme ofereix un bon rendiment, ja que podem saber en quina línia cercar o emmagatzemar el bloc d'MP associat a una adreça de memòria simplement mirant els bits de l'adreça. No obstant, aquesta tècnica no té en compte el nivell d'ocupació de la resta de línies de l'MC. Podríem tenir tota la cache buida tret d'una línia i veure'ns obligats a reemplaçar-la si els bits de l'adreça així ho indiquen. L'associativitat consistirà en ubicar els blocs dins la cache de forma més flexible, permetent que un bloc pugui ser emmagatzemat en més d'una línia.

6.1 Cache completament associativa

Portant a l'extrem aquest criteri, la màxima flexibilitat s'assoleix amb una cache *completament associativa*. Aquesta política d'emplaçament consisteix a ubicar els blocs d'MP a **qualsevol línia** que estigui lliure (després veurem què es fa si no n'hi ha cap). Això maximitza l'aprofitament de l'espai, reduint així la taxa de fallades, però fa que localitzar un bloc sigui molt més costós i lent. El número d'un bloc d'MP ja no ens donarà informació sobre la ubicació teòrica del bloc dins la cache, i ens veurem obligats a inspeccionar totes les línies una per una. El nombre de línies ja no té per què ser potència de 2.

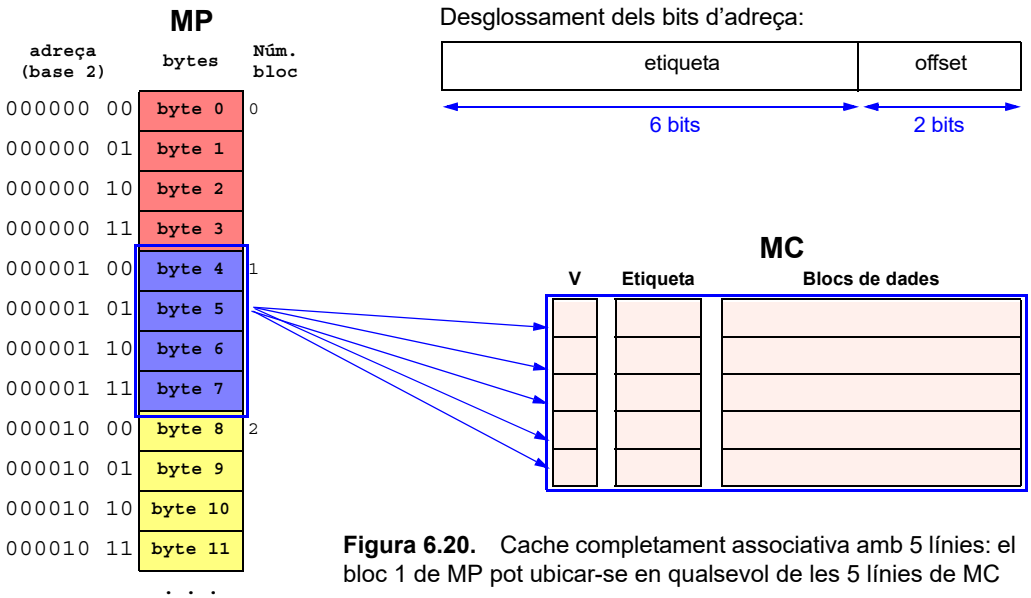


Figura 6.20. Cache completament associativa amb 5 línies: el bloc 1 de MP pot ubicar-se en qualsevol de les 5 línies de MC

6.2 Cache associativa per conjunts

Una solució intermèdia són les caches *associatives per conjunts*. Aquestes caches tenen un número determinat d'entrades, com les caches de correspondència directa, però no s'anomenen *línies* sinó *conjunts*. Dins de cada conjunt no només hi haurà espai per a un bloc, sinó que hi podrem encabir N blocs. Cadascun dels blocs que admet un conjunt l'anomenem *via*. Si els conjunts tenen N vies llavors l'anomenem *cache associativa per conjunts d'*N *vies* (N-*way associative*) i direm que té un grau d'associativitat N.

De forma anàloga a la correspondència directa, el número de bloc d'MP determina a quin conjunt s'emmagatzema el bloc. Suposant una cache amb capacitat per a NC = 2^{nc} conjunts, l'índex del conjunt és *index* = *num_bloc* mod NC, i es calcula seleccionant simplement els *nc* bits de menys pes de *num_bloc*.

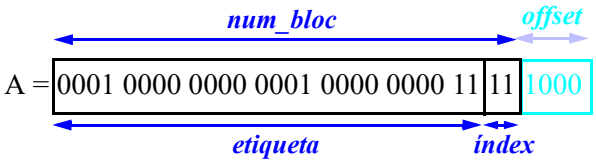


Figura 6.21. Desglossament d'una adreça de 32 bits amb blocs de mida B=16 bytes, suposant que la cache té NC=4 conjunts (*nc*=2 bits).

No obstant, dins d'un mateix conjunt, l'assignació d'un bloc a una via o una altra no depèn de l'adreça, sinó que es pot emmagatzemar a qualsevol via que estigui lliure (després veurem què es fa si no n'hi ha cap), com en l'emplaçament completament associatiu. Com que el nombre de vies no s'usa per al càlcul de l'índex, aquest no ha de ser necessàriament potència de 2. Vegem un altre exemple, amb adreces de 8 bits i blocs de 4 bytes, en una cache associativa per conjunts de 3 vies (amb 4 conjunts):

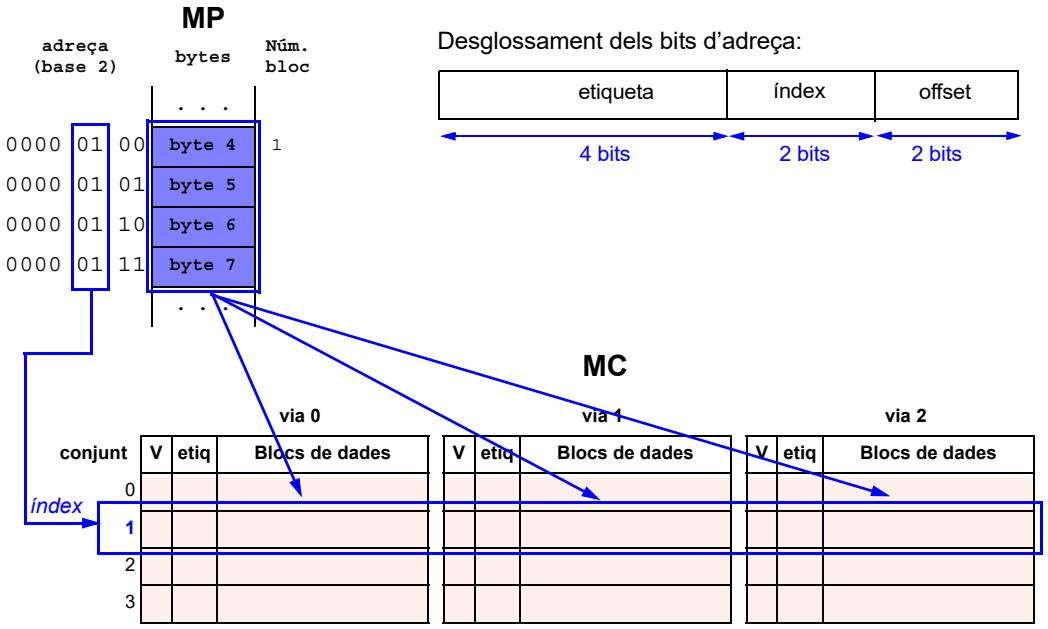


Figura 6.22. Suposem adreces de 8 bits, i blocs de B=4 bytes. La cache és associativa per conjunts, i té 4 conjunts, cadascun amb 3 vies. El bloc 1 de MP ha d'ubicar-se al conjunt 1, però pot triar qualsevol de les 3 vies que estigui lliure.

6.3 Algorisme de reemplaçament

En una cache associativa per conjunts, un bloc pot ser ubicat en qualsevol via dins el conjunt que li pertoqui. Una cache totalment associativa pot ser considerada com un cas extrem d'associativitat en què la cache té 1 sol conjunt, amb tantes vies com línies. En qualsevol dels dos casos, a l'hora de seleccionar una via, òbviament sempre seleccionarem aquella que estigui lliure (V=0). Si no n'hi ha cap de lliure, llavors la cache utilitza un *algorisme de reemplaçament* per seleccionar el *bloc víctima* que reemplaçarà.

- L'algorisme LRU (Least Recently Used) és un dels algorismes que produeix la menor taxa de fallades, i consisteix a reemplaçar el bloc que faci més temps que no s'accedeix. Aquest serà l'algorisme que usarem a la major part dels exercicis del curs. La implementació d'aquest algorisme és simple quan hi ha només dues vies, ja que es gestiona amb un sol bit per conjunt. Per a 4 vies, ja és més complex i, a la pràctica, s'usa un algorisme pseudo-LRU: agrupem les vies en 2 parelles i un bit decideix quina parella conté el bloc menys recentment usat; un altre bit en cada parella decideix quin bloc de la parella ha estat accedit abans.

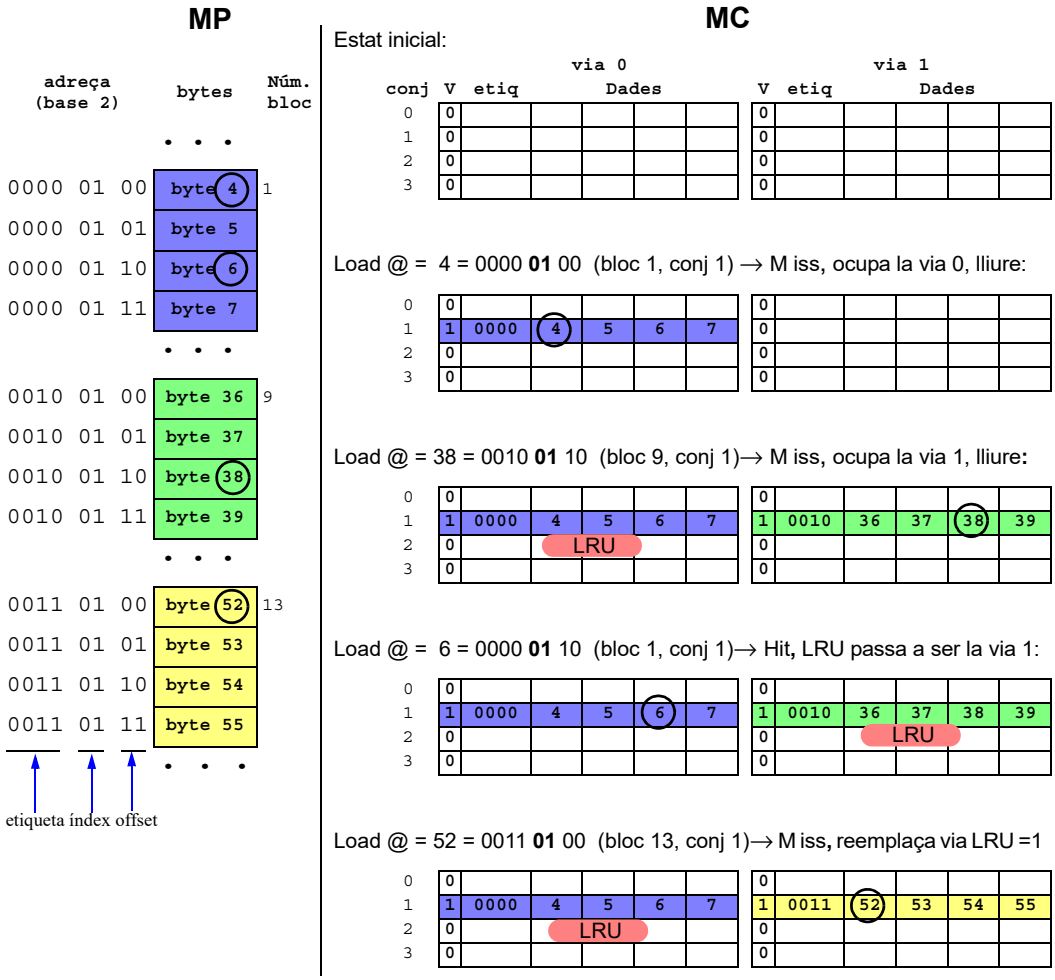


Figura 6.23. Suposem adreces de 8 bits, i blocs de B=4 bytes. La cache és associativa per conjunts, i té 4 conjunts, cadascun amb 2 vies. Fem 4 accessos, tots al conjunt 1. Les 2 primeres adreces (4, 38) fallen i ocupen vies lliures. L'adreça 6 encerta a la via 0 i fa que LRU passi a ser la via 1. L'adreça 52 falla i reemplaça el bloc LRU (el de la via 1).

- L'algorisme Aleatori (o Random) selecciona el bloc víctima a l'atzar (sovint usant algun senzill mecanisme hardware). No és tan eficient com LRU però la seva taxa de fallades s'aproxima molt a la que obté LRU quan el grau d'associativitat és molt alt. En aquests casos és una alternativa atractiva tenint en compte la simplicitat de la seva implementació.
- L'algorisme FIFO, menys eficient, selecciona com a víctima el bloc que fa més temps que ha "ingressat" en el conjunt (no confondre amb el criteri LRU que selecciona el que fa més temps que ha estat "usat").
- L'algorisme OPT selecciona com a víctima el bloc que serà usat més tard en el futur. S'ha demostrat teòricament que OPT aporta la taxa de fallades òptima possible, però per desgràcia no és implementable ja que requereix conèixer els accessos a memòria futurs! No obstant, en entorns d'experimentació per mitjà de simulació, conèixer la taxa de fallades mínima permet avaluar quin grau de millora es pot perseguir en la recerca de nous algorismes. Resulta interessant observar que de fet, LRU és un intent intuïtiu de predir el comportament futur en base a l'observació del comportament passat: "quin bloc fa més temps que no s'usa" intenta predir "quin bloc s'usarà més tard", i serà sovint correcte!

6.4 Diagrama de blocs

La Figura 6.24. mostra el diagrama de blocs d'una cache associativa per conjunts amb 256 conjunts de 4 vies. Les adreces són de 32 bits i els blocs de 4 bytes. S'observa que calen 4 comparadors per determinar quina via (si n'hi ha cap) conté la mateixa etiqueta que la del bloc d'MP que s'està cercant. A més de servir per determinar si es produeix encert o fallada (mitjançant una porta OR) la sortida dels comparadors fa de selector en un multiplexor 4 a 1 que selecciona una de les quatre vies.

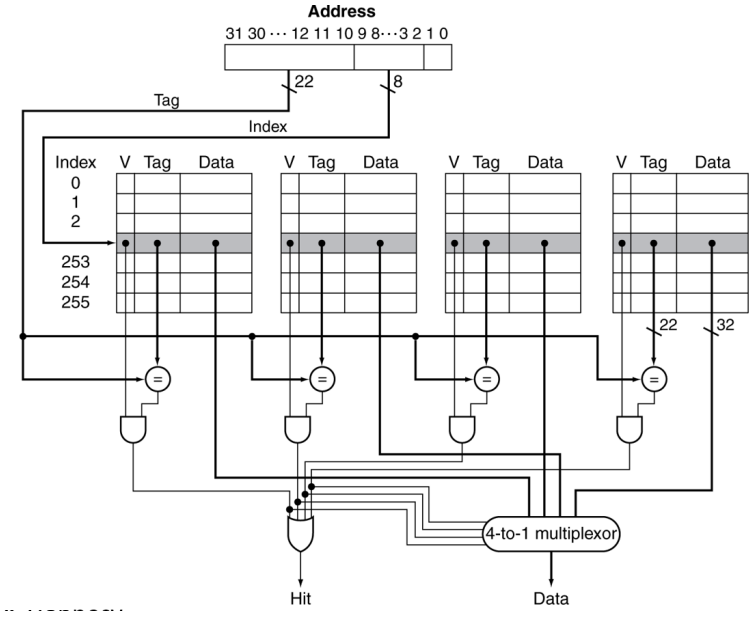


Figura 6.24. Cache associativa per conjunts amb 256 conjunts de 4 vies i blocs de 4 bytes. Font: David A. Patterson and John L. Hennessy, Computer Organization and Design: the Hardware/Software Interface, 5th ed., Morgan Kaufmann, 2012.

7. Millores de rendiment: caches multinivell

Podem desglossar els cicles d'execució (CPI) en dos termes: CPI_{ideal}, que no depèn de les fallades, i CPI_{penal}, que són els cicles d'espera causats per les fallades.

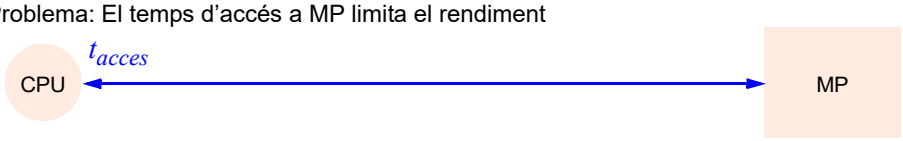
$$\text{CPI}_{penal} = m \cdot t_p \cdot n_r$$

Ja hem vist com es pot reduir la taxa de fallades (m) augmentant el grau d'associativitat de la cache. No obstant, el temps que tarda la memòria principal (DRAM) en completar un accés (i per tant la penalització t_p) no ha disminuït amb els anys al mateix ritme que el temps de processament d'una instrucció en la CPU i, en conseqüència, el temps de penalització per fallada ha esdevingut una fracció dominant del temps d'execució.

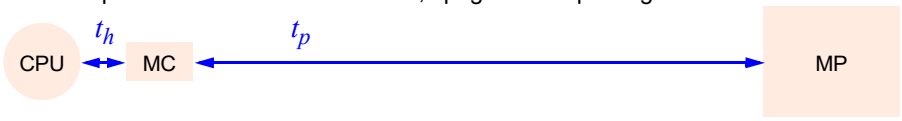
La Figura 6.25 il·lustra com un sistema simple (apartat a) pot millorar el temps d'accés mitjà a memòria incloent una cache (apartat b) de manera que només pateixi la penalització t_p durant les fallades, tal com s'ha explicat al llarg d'aquest tema.

A fi de reduir l'impacte d'aquesta penalització en el rendiment podríem reduir-ne el nombre d'ocurrències (m) si augmentéssim la capacitat de la cache, però això també augmentaria el temps d'accés en cas d'encert (t_h) i podria incrementar el CPI_{ideal}, reduint paradògicament el rendiment. En canvi, per a reduir l'impacte de la penalització resulta més efectiu seguir una estratègia anàloga a la que ens ha portat a incloure una cache: afegint un segon nivell de cache L2 a la jerarquia de memòria (apartat c). Donada una referència de la CPU a memòria, s'intentarà primer resoldre-la en L1. Si falla, s'anirà a buscar el bloc a L2, en comptes d'anar a memòria principal. I només si el bloc falla en L2, aleshores s'anirà a buscar a memòria principal. Aquest bloc ingressarà en L2 (i també en L1[10]). Observem que les fallades de L1 que encertin en L2 causaran un temps de penalització igual a t_{h_L2}, un temps molt menor que t_p, i s'haurà reduït així la penalització promig.

(a) Problema: El temps d'accés a MP limita el rendiment



(b) Solució: explotar la localitat amb una MC, i pagar el temps llarg sols en les fallades



Encara hi ha un problema: la penalització de les fallades limita el rendiment

(c) Solució: apliquem la mateixa idea però no a les peticions de la CPU, sinó a les peticions del conjunt CPU+L1 (és a dir, a les fallades de L1), afegint-hi una cache L2

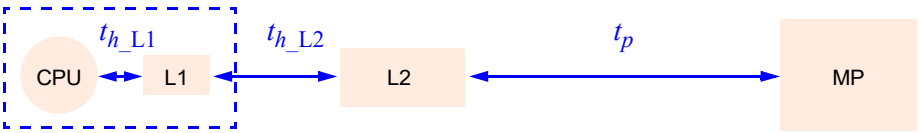


Figura 6.25. Les caches multinivell com a mètode per reduir el temps de penalització de les fallades de L1: aquelles que encertin en L2 tindran penalització t_{h_L2}

10. Hi ha múltiples opcions de disseny per a les caches multinivell. En aquest curs d'EC donarem per fet que la cache de nivell 2 és sempre *inclusiva* (conté sempre tots els blocs de la cache de nivell 1)

Tenir dos nivells diferents de cache ens permetrà orientar el disseny de cadascuna a un objectiu diferent. El disseny de la cache de nivell 1 l'orientarem a **reduir el temps d'encert** (t_{h_L1}) a un nivell que no afecti CPI_{ideal}. Això ho aconseguirem amb:

- Un nivell baix d'associativitat
- Blocs no molt grans
- Capacitat moderada

Si tinguéssim un sol nivell de cache, cada fallada accediria a la memòria principal i aquestes decisions de disseny incrementarien molt la taxa de fallades. Per contra, el fet de tenir una segona cache ho fa possible, donat que podem orientar el disseny d'aquesta precisament a **minimitzar la taxa de fallades**, a canvi de sacrificar lleugerament el temps d'encert t_{h_L2}, que serà major que t_{h_L1}. Per aquest motiu, la cache de nivell 2 té:

- Un grau alt d'associativitat
- Blocs grans
- Molta capacitat

L'enorme augment en la complexitat i rendiment dels processadors, així com la integració de múltiples nuclis (*cores*) en el xip, ha donat peu a un creixement enorme de la capacitat de la memòria principal (DRAM). A mesura que han anat creixent el nombre de processadors, ha augmentat també el tràfic sobre la DRAM, saturant l'ample de banda disponible de la interconnexió. A fi i efecte de reduir aquest tràfic, ha fet falta incorporar a la jerarquia de memòria un nou nivell de cache L3, compartit entre tots els processadors, i normalment integrat dins del propi xip (veure Figura 6.26).

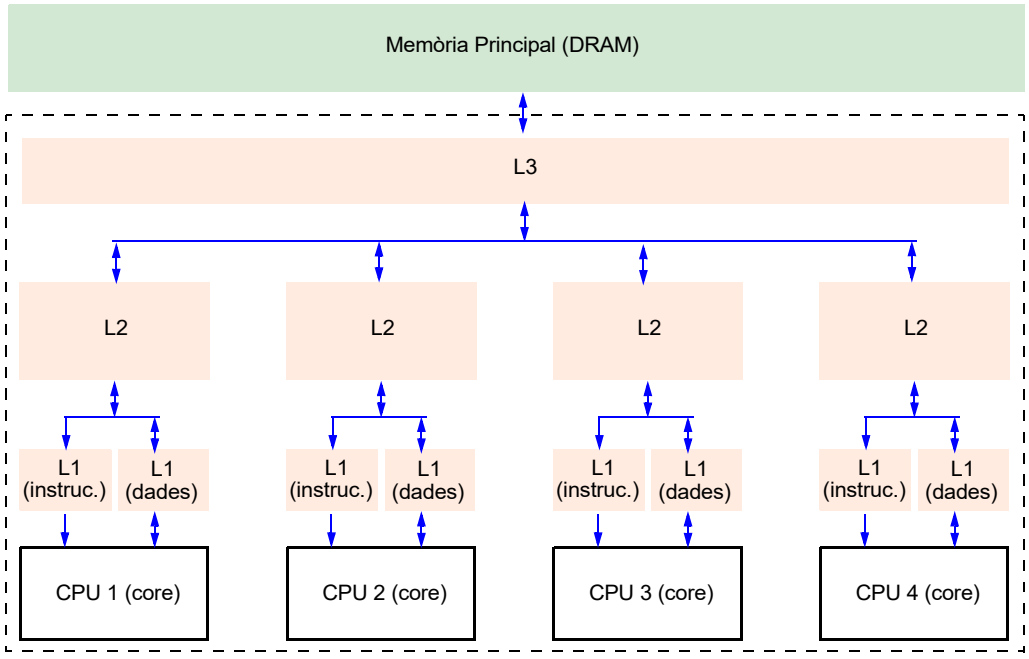


Figura 6.26. Cache multinivell en un xip (línia de punts) amb 4 processadors o nuclis

8. Tipologia de les fallades de cache

Les fallades de cache poden classificar-se en tres categories (les tres 'C'). Classificar-les permet raonar sobre les seves causes i idear solucions per evitar-les.

1. **C**old-start (arrencada en fred) o **C**ompulsory (obligatòria)

   Es produeixen el primer cop que s'accedeix a un bloc de memòria. Per la seva naturalesa, no es poden evitar augmentant el grau d'associativitat o la capacitat de la cache.

2. **C**onflicte o **C**ol·lisió

   Són fallades que succeeixen en caches de correspondència directa (o associatives per conjunts) quan múltiples blocs competeixen per la mateixa línia (o conjunt) de la cache i s'expulsen mútuament. Els conflictes tendeixen a disminuir si augmentem el grau d'associativitat. Aquestes fallades serien encerts en una cache completament associativa d'igual capacitat.

3. **C**apacitat

   Són fallades que succeeixen perquè a la cache no hi caben tots els blocs que necessita el programa durant la seva execució (el *working set*). Es produeixen en blocs que són reemplaçats i més tard visitats novament. Aquestes fallades no es poden evitar augmentant l'associativitat, solament es podrien reduir augmentant la capacitat de la cache.

La Figura 6.27 il·lustra la relació entre els tres tipus de fallades, l'associativitat i la mida de la cache per a un programa determinat d'exemple. Com es pot observar, la quantitat de *cold-start* misses és fixa, ja que sols depèn del nombre total de blocs diferents accedits pel programa. Per a una mida de cache determinada (línia de punts), les fallades de conflicte disminueixen a mesura que augmentem l'associativitat. En una cache completament associativa no hi ha conflictes, per definició. En canvi, les fallades de capacitat sols es poden reduir augmentant la mida de la cache, sigui quin sigui el grau d'associativitat.

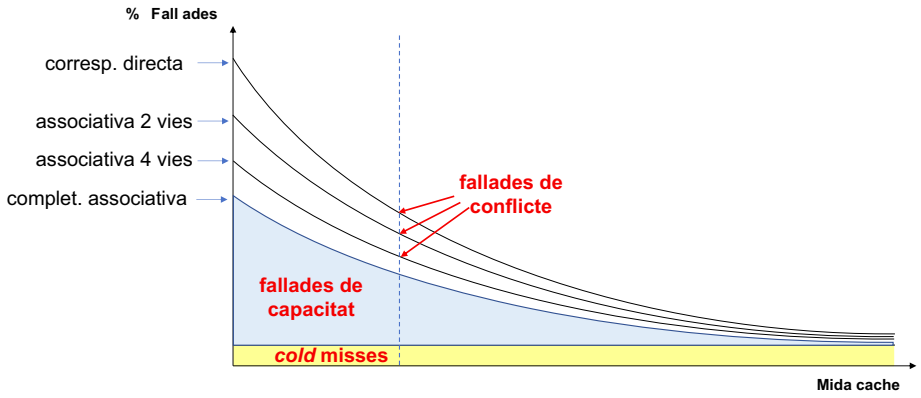


Figura 6.27. Percentatge de fallades segons el grau d'associativitat i la mida de la cache, per a un programa donat.

8.1 Exemple: Fallades de conflicte i d'arrencada en fred (cold-start)

Suposem la següent funció escrita en C, executada en un processador MIPS de 32 bits, amb blocs de 8 bytes (2 words per bloc)

```
int f(int A[8], int B[8]) {
        int i, s=0;
        for (i=0; i<8; i++)
                s = s + A[i] * B[i];
        return s;
}
```

Suposem una MC de correspondència directa, amb 4 línies, inicialment buida.

La Figura 6.28 mostra els canvis d'estat de la MC durant l'execució de les primeres dues iteracions del bucle de la funció `f`. En la primera iteració (`i=0`), els blocs de `A[0]` i de `B[0]` són accedits per primer cop i fan fallades per *arrencada en fred* (*cold-start*). Observem que, independentment del valor de `i`, `A[i]` i `B[i]` resideixen sempre en blocs mapejats a la mateixa línia 0 de MC, de manera que el bloc de `B[0]` reemplaça el bloc de `A[0]`. En la segona iteració aquests blocs s'accedeixen per segon cop, tornen a fallar i s'expulsen mútuament. No obstant, aquestes fallades no són d'arrencada en fred, ja que els blocs havien estat visitats abans, són fallades de *conflicte*.

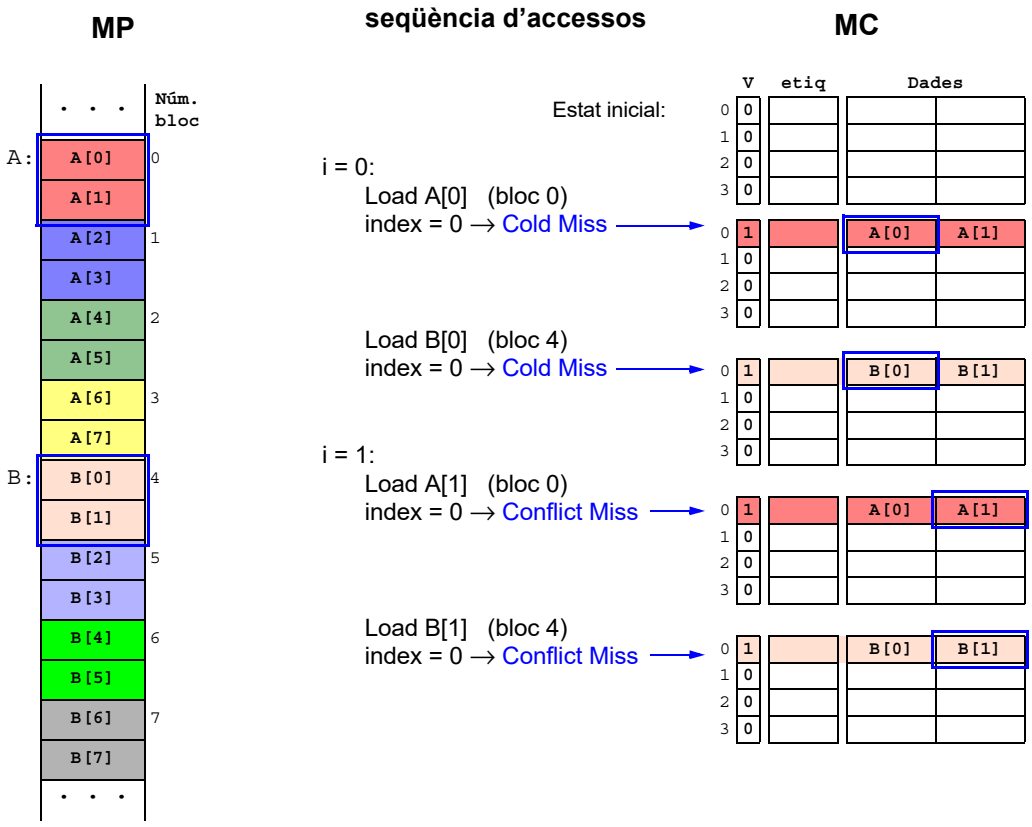


Figura 6.28. Canvis d'estat de la MC durant l'execució de les 2 primeres iteracions del bucle de la funció f.

8.2 Exemple: Fallades de capacitat

Ara executem `g` amb el mateix context, però amb una **MC totalment associativa**:

```
int g(int V[16]) {
        int i, s=0;
        for (i=0; i<16; i++)  s = s + V[i];
        for (i=0; i<16; i++)  s = s + V[i]*3;
        return s;
}
```

A la cache sols hi cap la meitat del vector V. Els blocs de la segona meitat (V[8] a V[15]) reemplacen els de la primera meitat V[0] a V[7] (Figura 6.29). Quan el segon bucle els torna a visitar, els blocs ja no hi són i els accessos fallen: són fallades de *capacitat*.

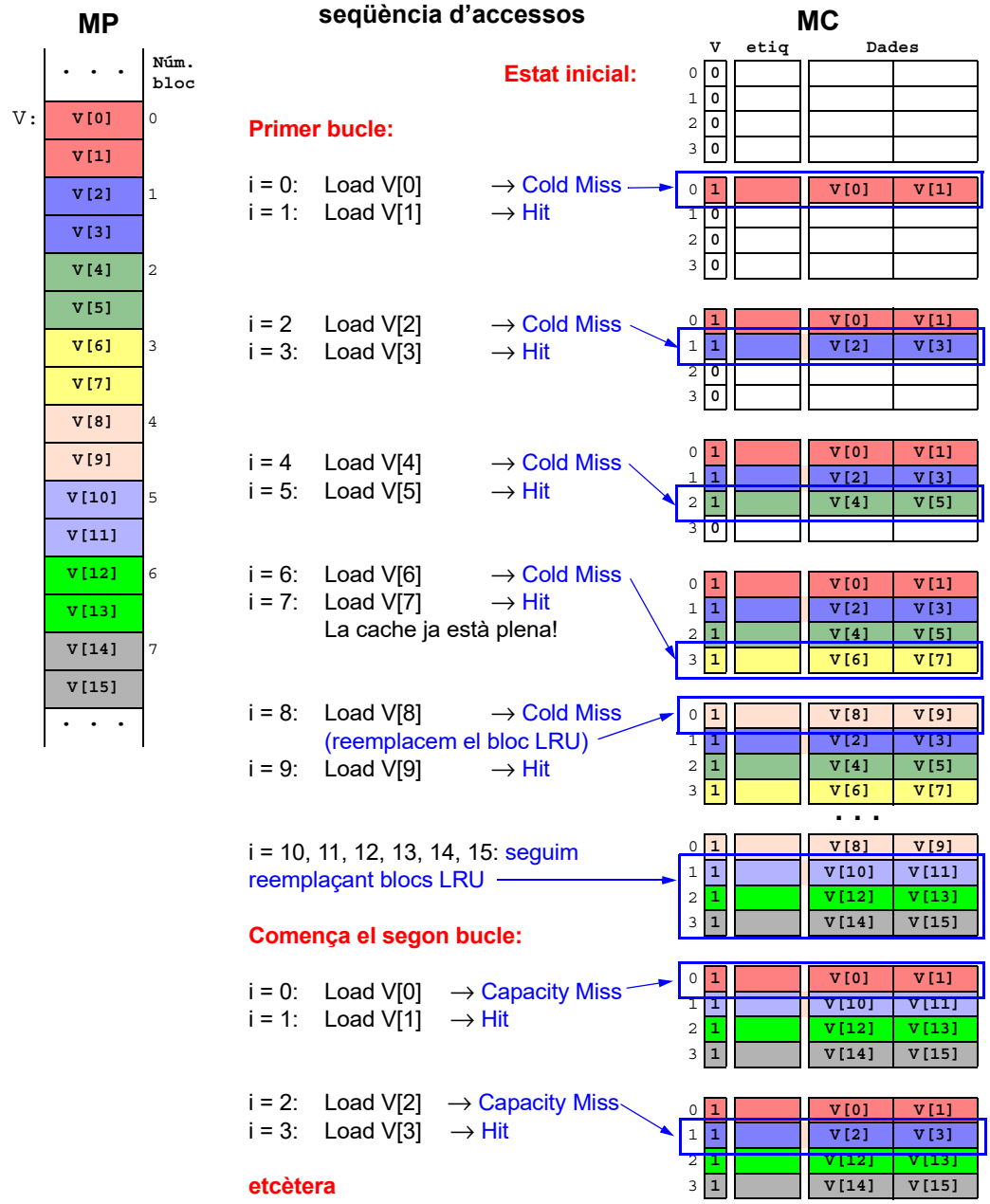


Figura 6.29. Canvis d'estat de la MC totalment associativa durant l'execució de la funció g