Estructura de Computadors (EC)

Tema 6. Memòria Cache

Rubèn Tous
Joan Manuel Parcerisa

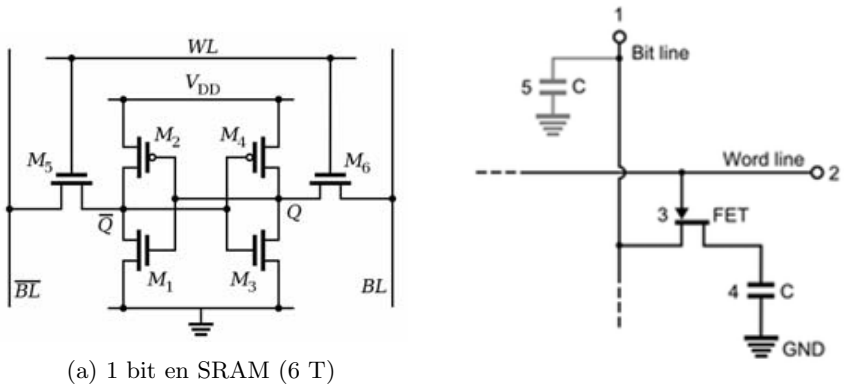
Departament d'Arquitectura de Computadors
Facultat d'Informàtica de Barcelona
Abril 2020

Índex

1 Introducció

Tot computador està proveït 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 0,1 ms en els SSD. Per a un període de rellotge de CPU típic de 0,25 ns, cada accés tardaria 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 (50 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 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)

Figure 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 commutació 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 2), en part degut que la capacitat de memòria instal·lada ha crescut també exponencialment.

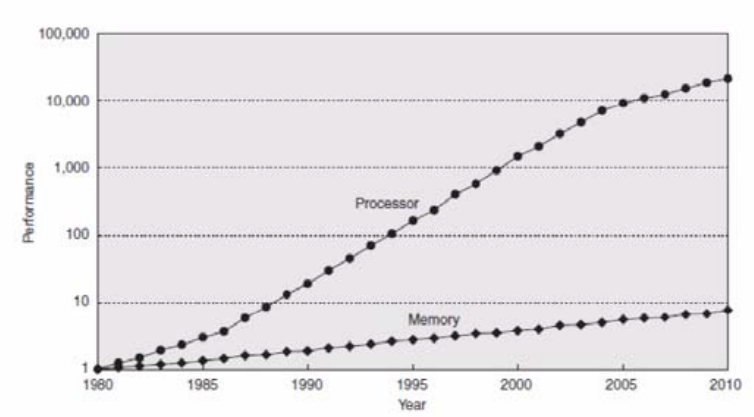


Figure 2: Evolució del rendiment relatiu de processador i memòria.

El resultat és que actualment el temps d'un accés a memòria[1] (t_m) 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 (incloses descodificació, lectura de registres, ALU, etc.), diferència que anomenem gap entre processador i memòria.

Quin és el problema? Sabem que el rendiment del processador és proporcional al CPI (essent CPI = $t_{proc} + t_m$). Suposant que $t_m = 100 * t_{proc}$, la fracció del temps d'execució invertida en el processament de la instrucció (sense comptar l'accés a memòria) és $\alpha = t_{proc}/(t_{proc} + t_m)$, és a dir $\alpha \approx 0,01$. Si solament som capaços de millorar t_{proc} (sense canviar t_m), 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":

[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.

- **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 cerca d'instruccions en memòria, aquesta propietat és deguda al seqüencialment implícit: el processador sempre 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.

Analogia de la biblioteca:
El principi de localitat no només es dona en l'execució de programes d'un computador, també es dona, per exemple, en l'accés als llibres d'una biblioteca. Si agafem un llibre, la probabilitat de que tornem a agafar el mateix llibre en un futur proper és molt més alta que la probabilitat de que agafem un llibre qualsevol (localitat temporal). La probabilitat de que agafem un llibre del mateix prestatge també serà més alta (principi de localitat espacial).

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 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:

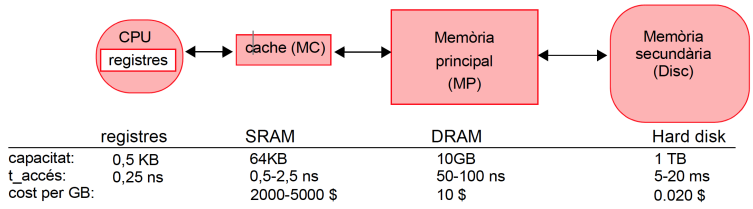


Figure 3: Capacitat, temps d'accés i cost per GB típics

[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

L'MC emmagatzema un subconjunt del contingut de l'MP. L'objectiu de l'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:

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

Analogia de la biblioteca (cache = taula):
Després de fer servir un llibre, podem deixar-lo una estona a la taula on estem treballant, en comptes de tornar-lo a deixar al prestatge. D'aquesta manera ens estalviarem d'haver-lo d'anar a buscar en el cas (cosa probable degut al principi de localitat temporal) de que el tornem a necessitar. Una altra cosa que podem fer és, quan agafem el llibre del prestatge, aprofitar per agafar altres llibres del mateix prestatge que potser necessitarem (cosa probable degut al principi de localitat espacial), i deixar-los temporalment a sobre de la taula. D'aquesta manera, la taula fa de *cache* de llibres, i ens estalvia molts viatges a les prestatgeries.

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 no hi és direm que es produeix una *fallada* (*miss*). En aquest cas, caldrà accedir a la memòria principal en un temps considerablement 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. En cas contrari, si la dada ja es troba a la cache, es produeix un

encert (*hit*) i l'accés té lloc en un temps molt més curt.

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:

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

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

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_m 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 th) 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):

$$t_m = t_h \text{ (en cas d'encert)}$$

$$t_m = t_h + t_p \text{ (en cas de fallada)}$$

1.5 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 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

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

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.

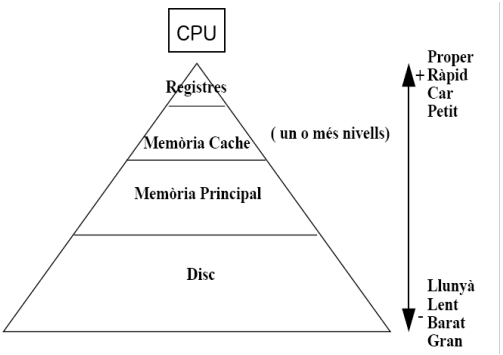


Figure 4: Jerarquia de memòria

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 *TAMBLOC* bytes, essent *TAMBLOC* una potència de 2 (mides típiques són entre 16 i 128 bytes). L'espai d'adreçament de memòria es subdivideix lògicament en blocs de *TAMBLOC* bytes, de manera que l'adreça inicial de cada bloc és un múltiple de *TAMBLOC*. 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 d'MP* (o també adreça de bloc), i li serveix a la cache per identificar els blocs que té guardats.

Analogia de la biblioteca (bloc = prestatge):
A l'analogia de la biblioteca el *bloc* podria correspondre, per exemple, a tots els llibres d'un prestatge. Cada cop que necessitéssim un llibre portaríem a la taula tots els llibres del prestatge.

Quan la CPU sol·liciti una dada, es copiarà a la memòria cache tot el bloc al que aquesta pertanyi (les altres dades del bloc seran les dades "properes"). Si numerem els blocs en que podem dividir la memòria principal de 0 a N-1, podem esbrinar el número de bloc al que pertany una adreça A (el *número de bloc MP*) fent:

$$\textit{número de bloc MP} = \frac{A}{TAMBLOC}$$

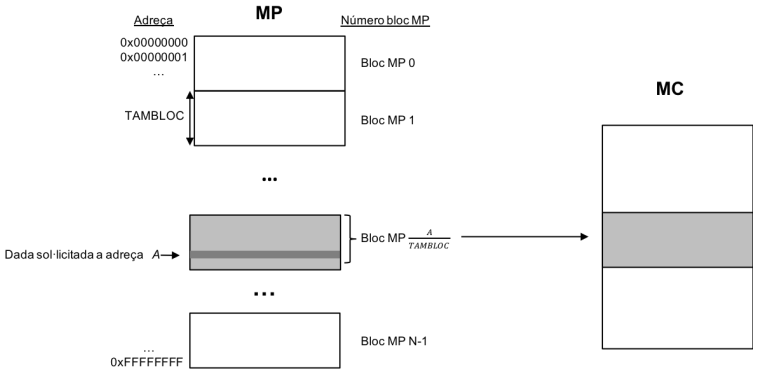


Figure 5: Blocs de memòria principal.

Com $TAMBLOC = 2^t$, això equival a descartar els t bits de menys pes de l'adreça. A aquests bits els anomenarem *block offset* (número de byte dins el

bloc):

$$block\ offset = A \mod TAMBLOC$$

Totes les adreces d'un bloc tindran diferents els t bits de menys pes (el *block offset*) però idèntics els $32 - t$ bits alts. De vegades, en comptes de parlar de *block offset*, es diu que els 2 bits de menys pes són el *byte offset* (número de byte dins la paraula) i els altres $t - 2$ bits el *word offset* (número de paraula dins el bloc).

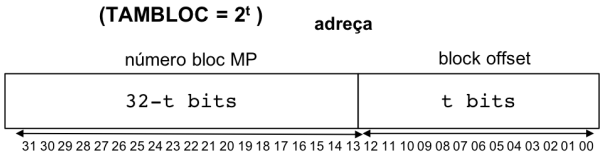


Figure 6: Número de bloc MP i block offset.

Anem a veure un exemple. Imaginem-nos blocs de dos paraules cadascun ($TAMBLOC = 8$ bytes). Donat que $8 = 2^3$ tindrem que $t = 3$ (el *block offset* seran els 3 bits de menys pes de l'adreça). Imaginem-nos que es produeix una lectura a l'adreça 0x0000000C. Els 29 bits de més pes ens indicaran el número de bloc MP, el número 1. Els 3 bits de menys pes ens indicaran el block offset (byte 4 dins el bloc). O si volem, word offset 1 (paraula 1 dins el bloc) i byte offset 0 (byte 0 dins la paraula 1).

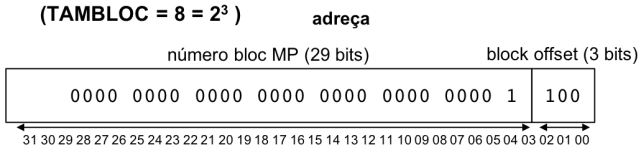


Figure 7: Número de bloc MP i block offset (A=0x0000000C).

2.2 Cache de correspondència directa

Ara sabem que l'MC emmagatzema blocs de dades accedits recentment, però, ón els guarda? cóm sap si un bloc ja hi és? La part del disseny d'una memòria cache que respon a aquestes preguntes s'anomena política de *ubicació* (*placement*). La política de ubicació més senzilla és la de *correspondència directa* (*direct-mapped*).

Una cache de correspondència directa consta d'un número de *línies* determinat (NUM_LíNIES). A cada *línia* hi pot anar un bloc de memòria principal. Per exemple, podem imaginar-nos una MC de 4 línies i blocs de 2 paraules cadascun.

La característica principal d'una cache de correspondència directa és que es determina a quina línia d'MC s'ubiquen els blocs de memòria principal en funció

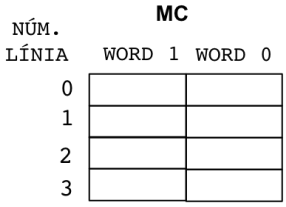


Figure 8: Exemple d'MC de correspondència directa amb 4 línies de 2 paraules cadascuna.

del número de bloc MP. El primer bloc d'MP anirà sempre a la primera línia, el segon a la segona, etc., fins al bloc MP número NUM_LíNIES, que tornarà a anar a la primera línia i així successivament. És a dir, el número de línia d'MC serà el residu de dividir el número de bloc MP per NUM_LíNIES:

$$\textit{número de línia MC} = \textit{número de bloc MP} \mod \text{NUM\_LíNIES}.$$

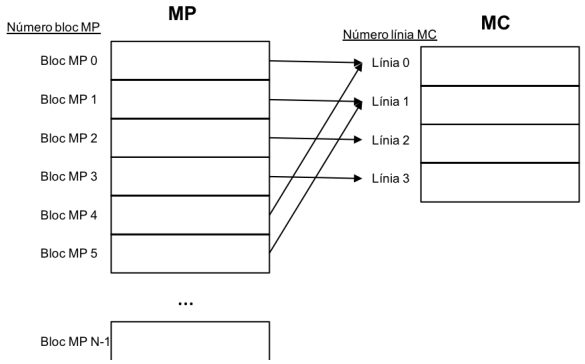


Figure 9: Assignació de línies d'MC a blocs d'MP en una cache de correspondència directa.

Donat que NUM_LíNIES sempre serà un potència de 2 (NUM_LíNIES $= 2^l$), donada l'adreça d'una dada podrem saber a quina línia de cache hem de buscar-la consultant els l bits de menys pes dels bits de *número de bloc MP* de l'adreça. La resta de bits dels bits del *número de bloc MP* de l'adreça s'anomenen *etiqueta* (*tag*). Donada una adreça, l'etiqueta serà el que ens permetrà distingir entre blocs d'MP diferents, encara que corresponguin a la mateixa línia d'MC.

Anem a veure un exemple. Seguim amb la cache de 4 línies i blocs de 2 paraules cadascun. Imaginem-nos que es produeix una lectura a l'adreça 0x0000000C. Sabem, d'un exemple anterior, que els 29 bits de més pes ens indiquen el número de bloc MP, el número 1 en aquest cas. Els 3 bits de menys pes ens indiquen el block offset (el byte 4 dins el bloc en aquest cas). Donat que l'MC té 4 línies ($4 = 2^2$), tindrem que $l = 2$ (el *número de línia* seran els 2 bits

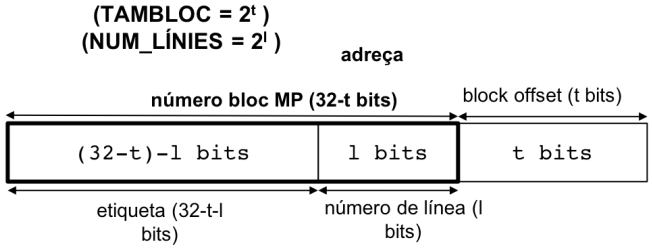


Figure 10: Els l bits de menys pes dels bits del *número de bloc MP* de l'adreça indiquen el número de línia de cache on s'ha ha d'anar el bloc al que correspon la dada.

de menys pes de la part del *número de bloc MP* de l'adreça). La resta de bits, 27 bits, seran l'etiqueta.

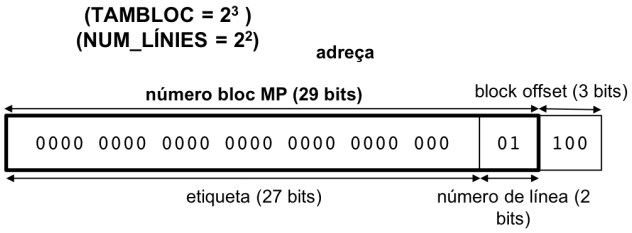


Figure 11: Amb una cache de $4 = 2^2$ línies, els 2 bits de menys pes dels bits del *número de bloc MP* de l'adreça indiquen el número de línia.

D'aquesta manera, podem determinar si el bloc al que pertany la dada que necessitem (el bloc amb número de bloc MP = 1) està a la nostra cache de correspondència directa simplement mirant l'adreça, que també ens dirà a quina línia hauria d'estar (línia 1). En cas que el bloc no hi sigui, serà copiat a aquesta línia.

El *bit de validesa* (V)

Però com sabrem si el bloc ja és a la línia o no? Hi haurà un *bit de validesa* (V) a cada línia, inicialitzat a zero, que ho indicarà:

L'*etiqueta* (tag)

Però què passaria si a continuació es produeix una lectura a l'adreça 0x00000028? Inspeccionem primer l'adreça:

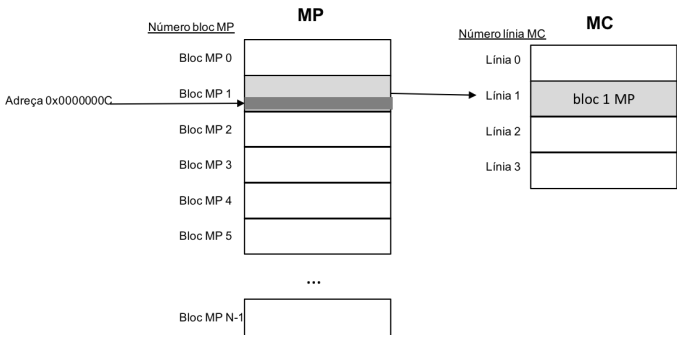


Figure 12: L'adreça 0x0000000C pertany al bloc MP 1. Amb una cache de $4 = 2^2$ línies, els 2 bits de menys del *número de bloc MP* ens indiquen el número de línia al que correspon el bloc, que en aquest cas serà la línia 1.

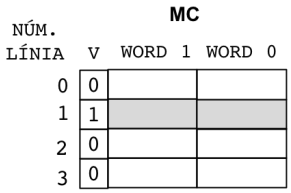


Figure 13: El *bit de validesa* (V) indica si una línia de cache conté un bloc.

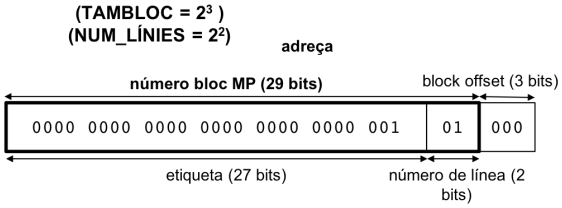


Figure 14: L'adreça 0x00000028 pertany al bloc MP 5, que també correspon a la línia 1 de la nostra cache de 4 línies.

L'adreça correspon al bloc 5 d'MP però aquest també correspon a la línia 1 de la cache:

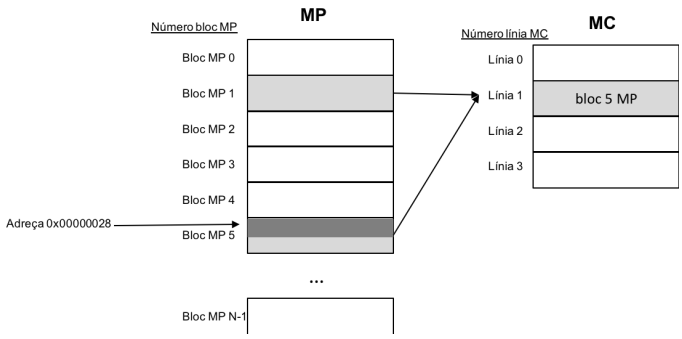


Figure 15: El bloc MP 5 i el bloc MP 1 corresponen a la mateixa línia d'MC. Ens caldrà afegir a la memòria cache alguna cosa que ens permeti distingir uns blocs del altres.

Donat que hi haurà molts blocs d'MP amb el mateix número de línia MC ens caldrà afegir a la cache informació sobre l'etiqueta de cada bloc:

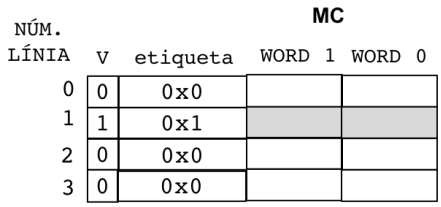


Figure 16: A la memòria cache guardarem també l'etiqueta de cada bloc. Això ens permetrà distingir uns blocs dels altres.

Reemplaçaments

Quan s'ha de copiar un bloc a una línia de la cache però la línia ja conté un altre bloc (amb una etiqueta diferent) es reemplaçarà el bloc antic pel bloc nou (més endavant, quan parlem d'escriptures, veurem que els reemplaçaments poden implicar més accions). A l'exemple anterior, el bloc 5 d'MP reemplaça el bloc 1 a la línia 1.

Exemple

Are que coneixem amb més detall el funcionament de la cache, anem a revisar l'exemple anterior pas a pas. Seguim amb la cache de 4 línies i blocs de 2 paraules cadascun, inicialment buida.

Primer fem una lectura de l'adreça 0x0000000C (suposem un *lw*). Com hem vist abans, l'adreça correspon al bloc 1 d'MP, i aquest correspon a la línia 1 d'MC. Inicialment la línia 1 té el bit de validesa a 0, de manera que es produirà una fallada de cache. Això provocarà que es copïi el bloc 1 d'MP a l'MC, es posi el bit V a 1 i s'escrigui un 0 a l'etiqueta (veure Figura 17).

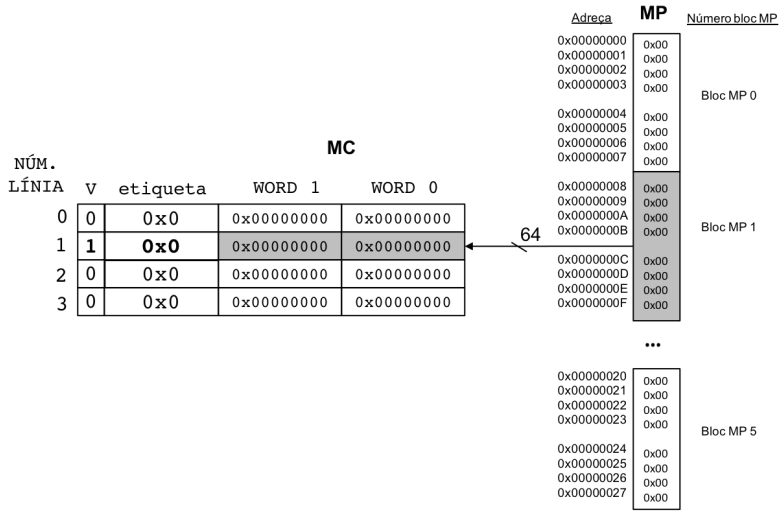


Figure 17: La fallada provocada per l'adreça 0x0000000C provoca que es transfereixi el bloc 1 d'MP a l'MC.

Un cop s'ha copiat el bloc, el processador accedeix a la dada (una paraula) a la memòria cache tal i com es mostra a la Figura 18.

A continuació, es produeix una lectura de l'adreça 0x00000028. Com hem vist abans, aquesta adreça correspon al bloc 5 d'MP, que també correpon a la línia 1 de l'MC. Aquesta línia té el bit V a 1, però té una etiqueta diferent (la 1) de manera que es produirà una fallada d'MC. Això provocarà que el contingut de la línia 1 (el bloc 1 d'MP), sigui reemplaçat pel bloc 5 d'MP tal i com mostra la Figura 19.

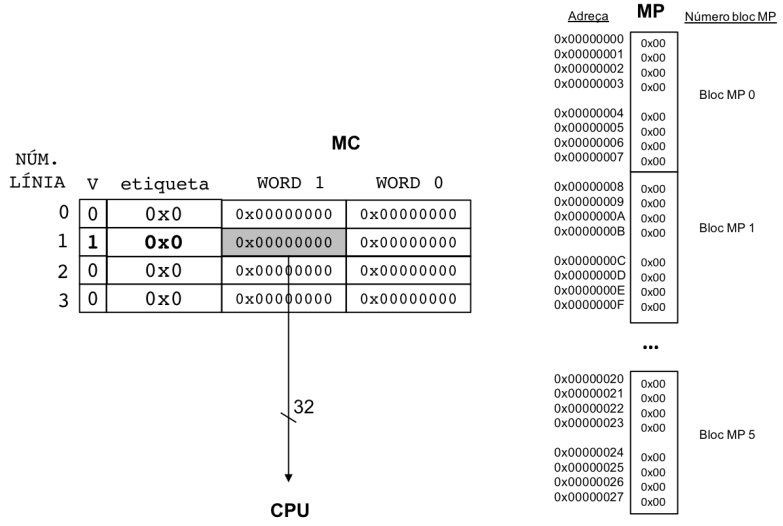


Figure 18: La fallada provocada per l'adreça 0x0000000C provoca que es transfereixi el bloc 1 d'MP a l'MC.

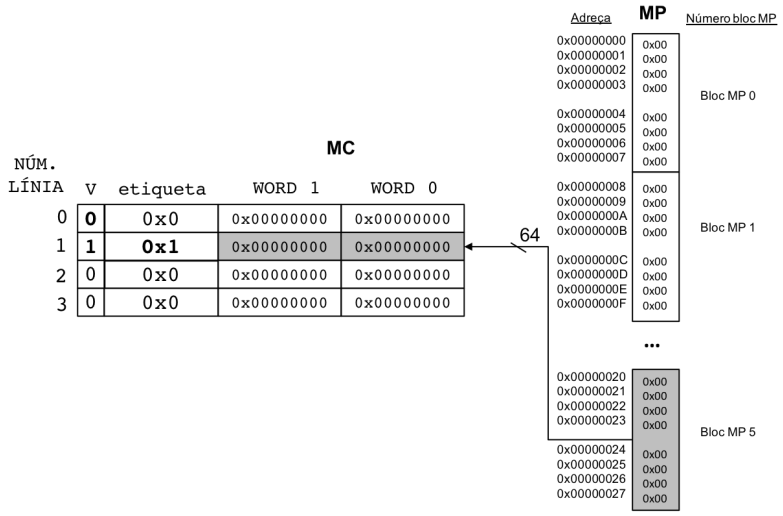


Figure 19: La fallada provocada per l'adreça 0x0000000C provoca que es transfereixi el bloc 1 d'MP a l'MC.

2.3 Diagrama de blocs (cache de correspondència directa)

La Figura 20 mostra el diagrama hardware d'una cache de correspondència directa i només lectura com la de l'exemple anterior, amb 4 línies i blocs de 2 paraules.

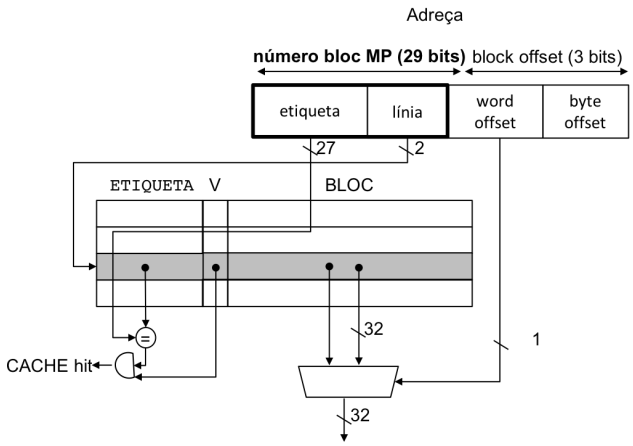


Figure 20: Diagrama hardware d'una cache de correspondència directa amb 4 línies i blocs de 2 paraules.

L'etiqueta emmagatzemada en la cache es compara amb l'etiqueta de l'adreça de memòria (els 27 bits de més pes) per saber si el bloc que conté la línia correspon al bloc d'MP al que pertany la dada. Si les etiquetes coincideixen i el bit de validesa està activat, aleshores es produirà un encert. El word offset (un bit en aquest cas ja que tenim blocs de només dues paraules) serveix de selector d'un multiplexor que s'encarrega de deixar la dada corresponent a la sortida.

2.4 Grandària de bloc. Impacte en la taxa de fallades i en el temps de penalització de les fallades

Copiar a la cache blocs amb moltes dades permet treure profit de la localitat espacial i reduir així la taxa de fallades. Però, com es tria la grandària d'un bloc? En principi, quan més gran sigui el bloc més encerts fruit de la localitat espacial tindrem, i menor serà la taxa de fallades. No obstant, hi ha múltiples blocs competint per ocupar espai a l'MC. Si incrementem massa la grandària del bloc respecte a la grandària total de la memòria cache arribarà un moment en que la taxa de fallades començarà a créixer. Els dissenyadors de memòries determinen la grandària òptima del bloc mitjançant *benchmarking*, és a dir executant un conjunt de programes representatius i mesurant el rendiment de les diferents configuracions. La Figura 21 mostra la taxa de fallades (eix d'ordenades) del benchmark SPEC92 amb blocs (eix d'abscisses) i caches de diferents grandàries. A la figura s'observa que quan la grandària del bloc és massa gran respecte a la grandària de l'MC la taxa de miss deixa de disminuir i comença a créixer.

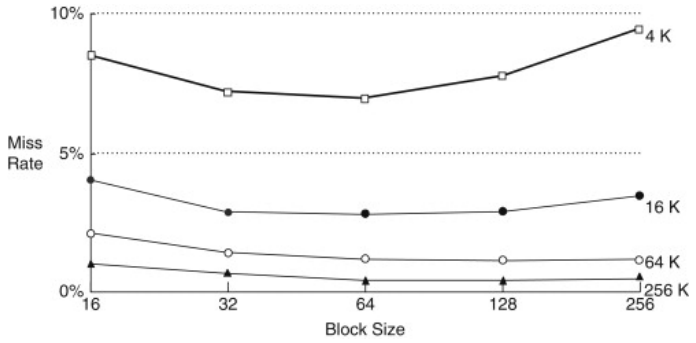


Figure 21: Taxa de miss, grandària del bloc i grandària de la cache. Benchmark SPEC92. Font: Hennessy and Patterson, Computer Architecture: A Quantitative Approach, 5th ed., Morgan Kaufmann, 2012.

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.

2.5 Gestió de les escriptures

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 en el moment en que comencem a modificar algunes dades haurem d'assegurar-nos de que les modificacions no es perden (per exemple quan reemplacem un bloc d'MC), i de que les lectures 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 poden produir dues situacions diferents: que la dada que es vol escriure es trobi a la memòria cache (encert d'MC) o que no hi sigui (fallada d'MC). Cadascuna d'aquestes situacions es pot resoldre mitjançant diferents tècniques, la combinació de les quals dona lloc al que s'anomena *política d'escriptura* (*write policy*).

Situació 1: encert d'escriptura (gestió de la coherència de dades)

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 es produeix un encert de cache, s'escriu **la dada** simultàniament a la memòria cache i a la memòria principal. D'aquesta manera, ambdues còpies de dada romanen iguals, i el funcionament bàsic de la memòria 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 de que a la memòria principal es conserva la mateixa informació.

- **Escriptura retardada** (*write-back* o copy-back): Si es produeix un encert de cache, s'escriu **la dada** únicament a la memòria cache. D'aquesta manera, hi haurà al mateix temps dues versions diferents de la dada, una a la memòria cache i una a la memòria principal. Això té un impacte sobre el funcionament bàsic de la memòria cache que hem estudiat fins ara. Quan calgui reemplaçar el contingut d'una línia, s'haurà de conèixer si la línia conté alguna modificació o no. Per aquest motiu, caldrà afegir un bit a la memòria cache, el bit D (*dirty bit*) que valdrà 1 si el bloc ha estat modificat (Figura 22).

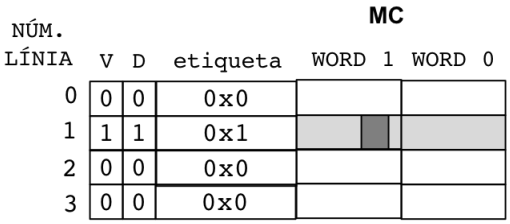


Figure 22: El bit D (*dirty bit*) assenyala quines línies contenen dades modificades si s'utilitza una escriptura retardada.

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 a la memòria cache el nou bloc, sobreescrivint el bloc antic.

És important adonar-se de que aquest canvi afecta de la mateixa manera als reemplaçaments provocats per lectures que als provocats per escriptures. De forma que la tècnica d'escriptura retardada té també un impacte en el funcionament de la cache quan es produeixen lectures.

Situació 2: fallada d'escriptura

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

- **Escriptura amb assignació** (write allocate): Si es produeix una fallada de cache, copiem **el bloc** a la memòria cache de la mateixa manera que ho fem amb una lectura.
- **Escriptura sense assignació** (no-write allocate): Si es produeix una fallada de cache, escrivim **la dada** a la memòria principal, com si no hi hagués cache.

Motivació de la política d'escriptura sense assignació:
De vegades, els programes escriuen seqüencialment molts blocs sencers de dades. Per exemple, de vegades el Sistema Operatiu ha d'escriure zeros a totes les paraules d'una pàgina de memòria (veurem que és una *pàgina* al Tema 7, però per ara podem pensar que són molts blocs). En aquests casos, portar els blocs a la memòria cache resulta contraproduent.

Combinacions de tècniques. La política d'escriptura

Els dos enfocs per a la gestió de la coherència (escriptura immediata o retardada) es poden combinar amb qualsevol dels enfocs per a la gestió de les fallades d'escriptura (escriptura amb assignació o sense assignació). La combinació escollida dona lloc al que s'anomena *política d'escriptura* (*write policy*). En el context de l'assignatura treballarem principalment dues combinacions o polítiques diferents:

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

Una excepció serà el laboratori, ja que el Mars fa servir escriptura immediata amb assignació.

Gestió de les escriptures i rendiment

La política d'escriptura escollida tindrà un impacte en el rendiment. Per un costat, la *política de fallada d'escriptura* (amb assignació o sense assignació), tindrà un impacte sobre la taxa de fallades, que dependrà del programa o conjunt de programes que s'executi. L'enfoc escollit per a la gestió de la coherència (immediata o retardada), per contra, no alterarà la taxa de fallades, ja que, sigui quin sigui l'enfoc escollit, a la cache sempre hi haurà els mateixos blocs d'MP. On sí impactarà l'enfoc de gestió de la coherència serà en el temps de penalització de les fallades:

- **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 processador pot continuar l'execució.

  Per un altre costat, un aspecte positiu de l'escriptura immediata és que podem seguir comprovant 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 aquest bloc contindrà la mateixa informació que hi haurà a MP i podrà ser reemplaçat directament.

- **Escriptura retardada**: Mantenir diferents versions de les dades a MP i MC introdueix problemes de rendiment diferents. Per un costat, quan fem servir escriptura retardada, no podem comprovar si hi ha encert o fallada i fer l'escriptura en el mateix cicle de rellotge, com passa amb l'escriptura immediata. Les línies d'MC poden contenir informació que encara no és present a MP i no podem escriure-hi fins estar segurs de que contenen el bloc que volem modificar. Per resoldre aquest problema,

també s'acostuma a fer servir un **buffer**, on s'escriu la dada de manera provisional.

Per un altre costat, quan una fallada de cache (de lectura o escriptura) requereix el reemplaçament d'un bloc i aquest conté dades modificades (bit D a 1), caldrà copiar tot el bloc a la memòria principal. Per minimitzar l'impacte d'això en el rendiment, també s'acostuma a fer servir un **buffer**, on queda emmagatzemat el bloc pendent de portar a MP, permetent així continuar l'execució.

Més endavant, quan s'especifiqui el **Model de Temps**, es concretarà quina serà la configuració de cache que farem servir nosaltres (per exemple quins buffers suposarem) de manera que puguem calcular quants cicles de rellotge trigarà una escriptura en les diferents situacions.

2.6 Exemple amb política d'escriptura immediata sense assignació

Imaginem-nos que tenim una cache de correspondència directa amb 4 línies de 2 paraules cadascuna i política d'escriptura immediata sense assignació. Imaginem-nos que, en un moment donat, l'MC i l'MP tenen els continguts que es mostren a la Figura 23:

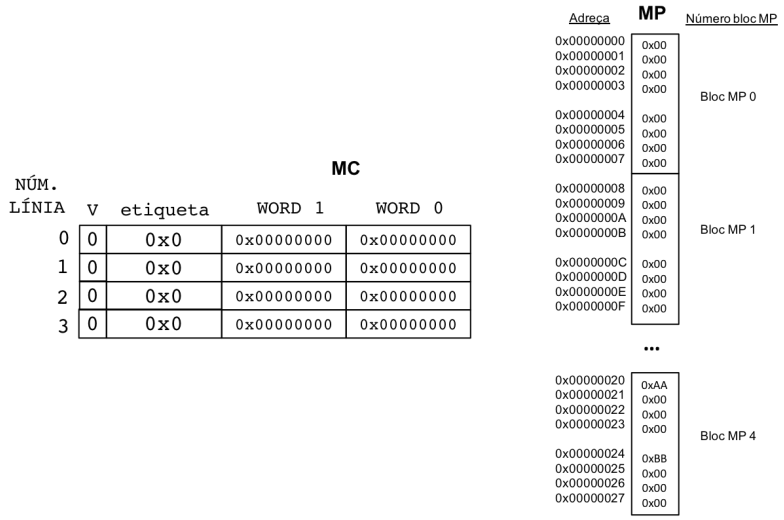


Figure 23: Estat inicial de l'MC i l'MP.

Lectura amb fallada

A continuació s'executa el següent codi:

```
li $t0, 36
lw $t1, 0($t0)
```

L'adreça 36 correspon a la segona paraula del bloc 4 d'MP, que alhora correspon a la línia 0 de l'MC (veure Figura 24).

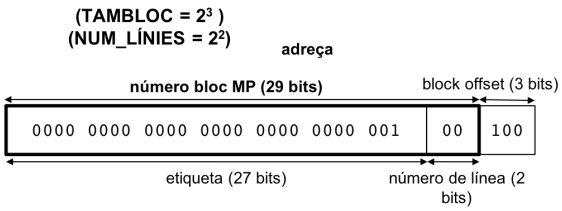


Figure 24: Anàlisi de l'adreça 36.

El que succeirà al subsistema de memòria serà el següent:

1. Donat que la línia 0 té el bit V = 0 es produirà una fallada de lectura.
2. Es copiarà el bloc 4 d'MP a la línia 0. Es posarà el bit V de la línia 0 a 1. I s'escriurà l'etiqueta 1 a la línia 0.
3. Es servirà la dada a la CPU.

La Figura 25 mostra l'estat final de l'MC i l'MP.

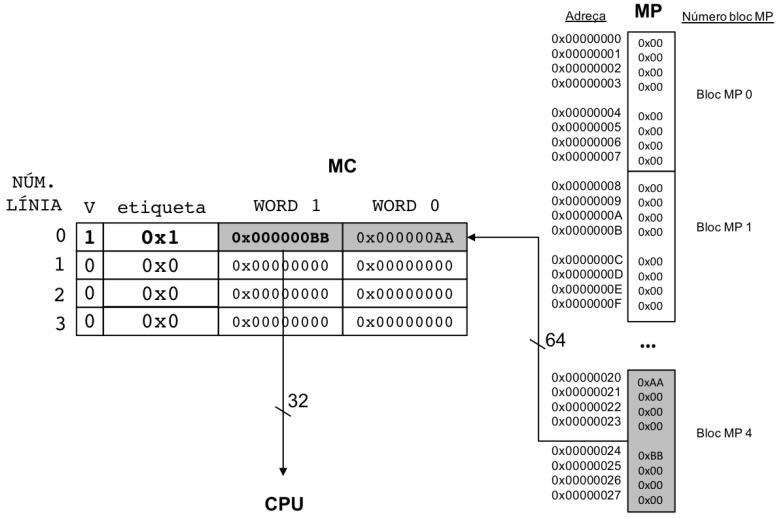


Figure 25: Estat final de l'MC i l'MP després d'una lectura amb fallada.

Lectura amb encert

A continuació s'executa el següent codi:

```
li $t0, 32
lw $t1, 0($t0)
```

L'adreça 32 correspon a la primera paraula del bloc 4 d'MP. El que succeirà al subsistema de memòria serà el següent:

1. Donat que la línia 0 té el bit V = 1 i l'etiqueta 1 (la mateixa que el bloc 4) es produirà encert de lectura.

2. Es servirà la dada a la CPU.

L'accés a les etiquetes per comprovar si la referència és un encert i el servei de la referència es poden realitzar en el mateix cicle. En general direm que totes dues coses es realitzen durant el temps t_h.

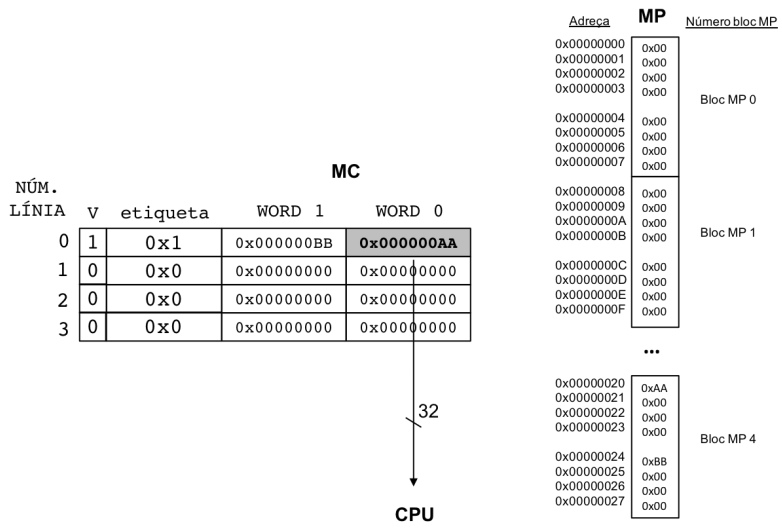


Figure 26: Estat final de l'MC i l'MP després d'una lectura amb encert.

Escriptura amb encert

A continuació s'executa el següent codi:

```
li $t0, 32
li $t1, 5
sw $t1, 0($t0)
```

El que succeirà al subsistema de memòria serà el següent:

1. Donat que la línia 0 té el bit V = 1 i l'etiqueta 1 (la mateixa que el bloc 4) es produirà encert d'escriptura.

2. S'escriurà la dada en paral·lel a l'MC i a la memòria.

Figure 27: Estat final de l'MC i l'MP després duna escriptura amb encert.

Escriptura amb fallada

A continuació s'executa el següent codi:

```
li $t0, 0
li $t1, 6
sw $t1, 0($t0)
```

L'adreça 0 correspon a la primera paraula del bloc 0 d'MP. El que succeirà al subsistema de memòria serà el següent:

1. Tot i que la línia 0 té el bit V = 1 també té etiqueta =1, diferent a l'etiqueta que correspon a l'adreça (etiqueta 0). Per tant, es produirà una fallada d'escriptura.

2. Donat que tenim una política d'escriptura **sense assignació**, no modificarem res a l'MC, simplement escriurem la dada a la memòria principal.

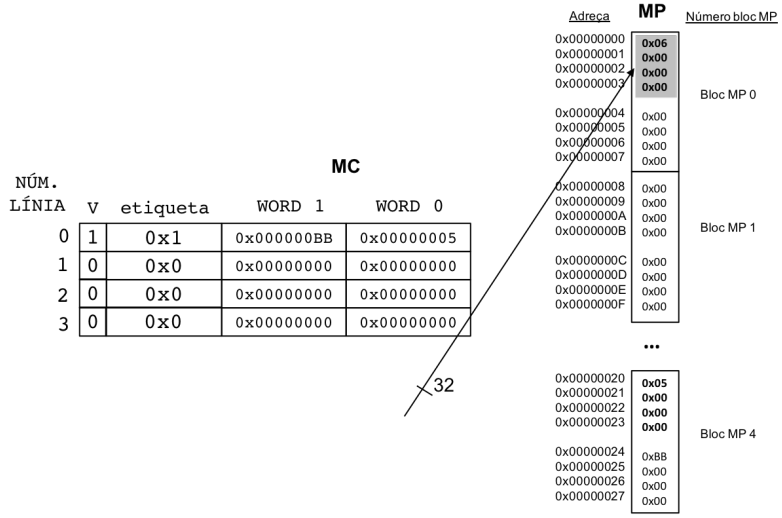


Figure 28: Estat final de l'MC i l'MP després d'una escriptura amb fallada.

2.7 Exemple amb política d'escriptura retardada amb assignació

Repetim ara els passos de l'exemple anterior però amb una política d'escriptura retardada amb assignació. Partim de nou d'un estat de l'MC i l'MP tenen els continguts que es mostren a la Figura 29:

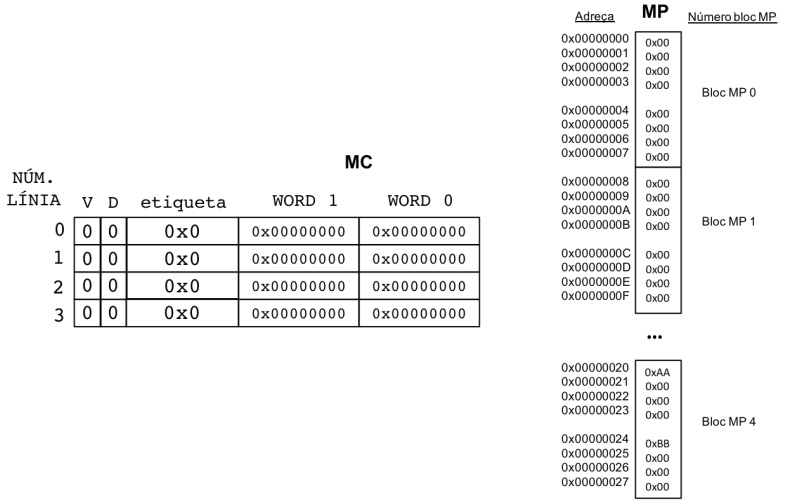


Figure 29: Estat inicial de l'MC i l'MP.

Lectura amb fallada d'un bloc no modificat

A continuació s'executa el següent codi:

```
li $t0, 36
lw $t1, 0($t0)
```

L'adreça 36 correspon a la segona paraula del bloc 4 d'MP (línia 0 d'MC). El que succeirà al subsistema de memòria serà el següent:

1. Donat que la línia 0 té el bit V = 0 es produirà una fallada de lectura.

2. Copiarem el bloc 4 d'MP a la línia 0. En aquest cas la línia encara no contenia cap bloc, però si n'hagués contingut un (V = 1) però amb una etiqueta diferent, l'haguéssim pogut reemplaçar directament si D = 0.

3. Es servirà la dada a la CPU.

La Figura 30 mostra l'estat final de l'MC i l'MP.

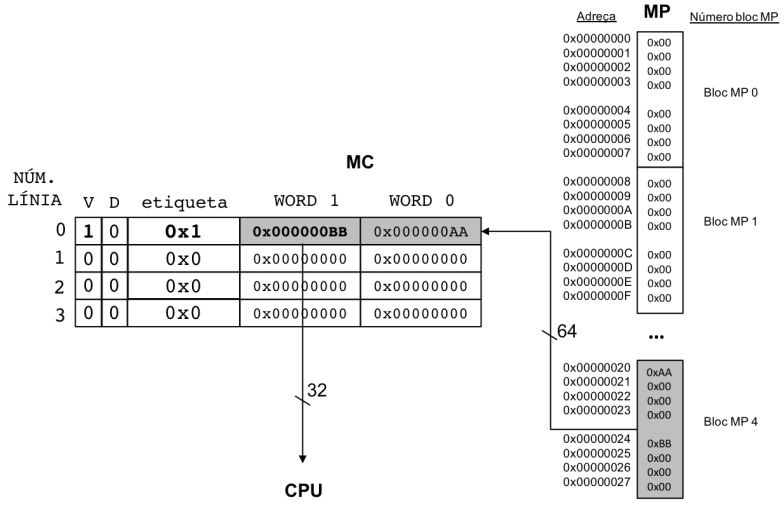


Figure 30: Estat final de l'MC i l'MP després d'una lectura amb fallada d'un bloc no modificat.

Lectura amb encert

A continuació s'executa el següent codi:

```
li $t0, 32
lw $t1, 0($t0)
```

L'adreça 32 correspon a la primera paraula del bloc 4 d'MP. El que succeirà al subsistema de memòria serà el el següent (no hi ha cap diferència amb l'exemple amb escriptura immediata sense assignació):

1. Donat que la línia 0 té el bit V = 1 i l'etiqueta 1 (la mateixa que el bloc 4) es produirà encert de lectura.
2. Es servirà la dada a la CPU.

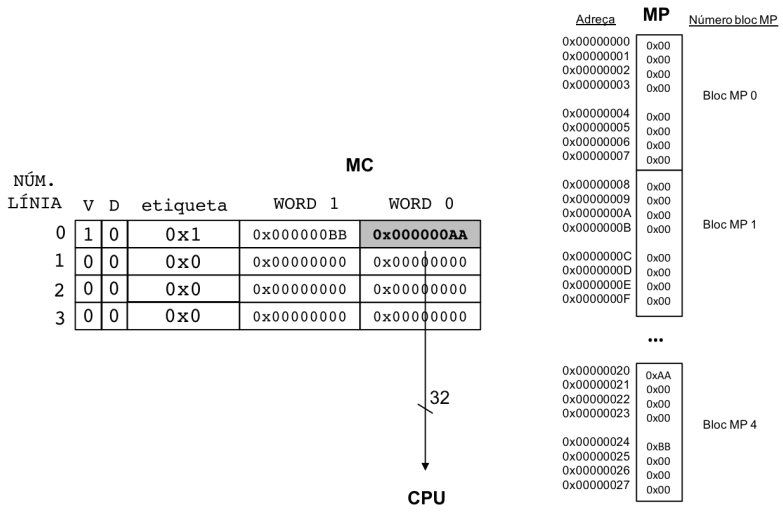


Figure 31: Estat final de l'MC i l'MP després d'una lectura amb encert.

Escriptura amb encert

A continuació s'executa el següent codi:

```
li $t0, 32
li $t1, 5
sw $t1, 0($t0)
```

El que succeirà al subsistema de memòria serà el següent:

1. Donat que la línia 0 té el bit V = 1 i l'etiqueta 1 (la mateixa que el bloc 4) es produirà encert d'escriptura.

2. S'escriurà la dada **només** a l'MC i es marcarà la línia com a modificada (D = 1).

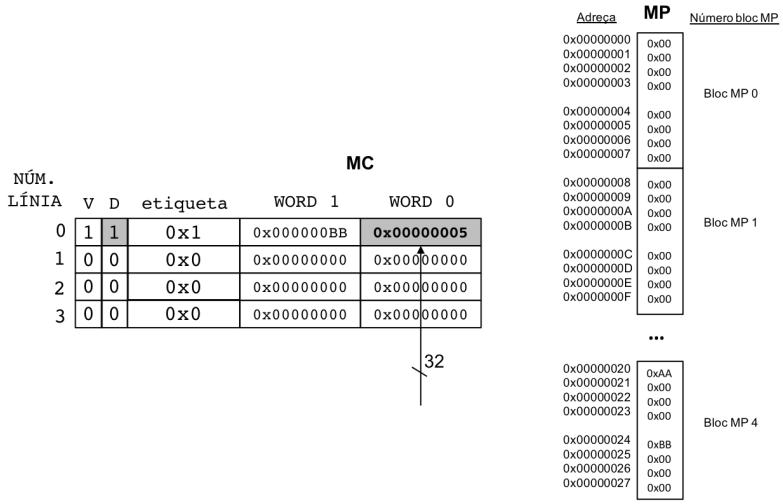


Figure 32: Estat final de l'MC i l'MP després d'una escriptura amb encert.

Lectura amb fallada d'un bloc modificat

A continuació s'executa el següent codi:

```
li $t0, 0
lw $t1, 0($t0)
```

L'adreça 0 correspon a la primera paraula del bloc 0 d'MP. El que succeirà al subsistema de memòria serà el següent:

1. Tot i que la línia 0 té el bit V = 1 també té etiqueta = 1, diferent a l'etiqueta que correspon a l'adreça (etiqueta 0). Per tant, es produirà una fallada de lectura.

2. Donat que la línia 0 conté un bloc modificat (D = 1) no podem reemplaçar el seu contingut directament. Ens cal primer copiar el bloc modificat (era el bloc 4 d'MP) a l'MP.

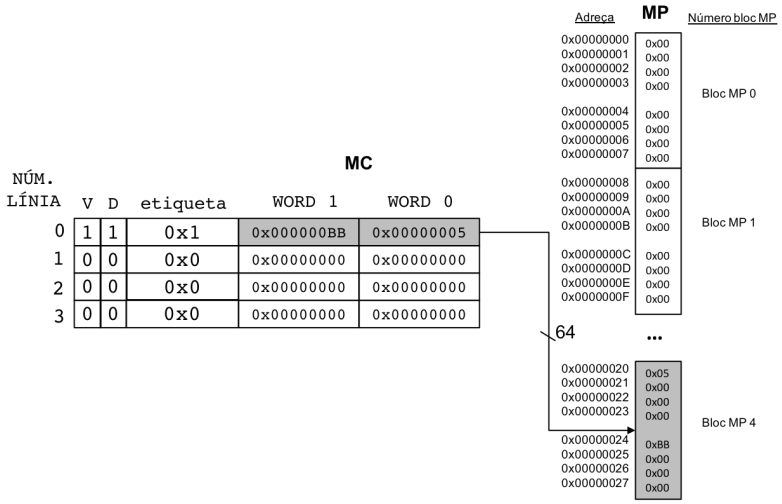


Figure 33: Una lectura amb fallada d'un bloc modificat provoca la transferència del bloc modificat a l'MP.

3. Un cop copiat el bloc modificat a l'MP podem copiar el nou bloc, el bloc 0 d'MP, a la línia 0.

4. Finalment podem servir la dada.

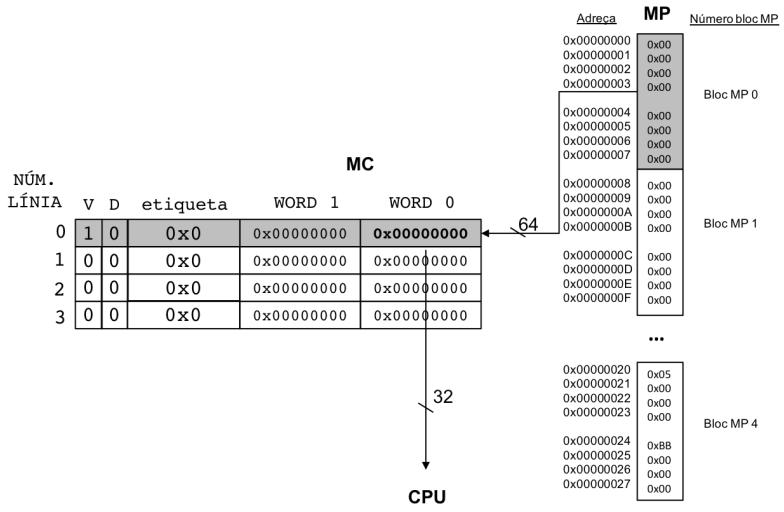


Figure 34: Estat final de l'MC i l'MP després d'una lectura amb fallada d'un bloc modificat.

Escriptura amb fallada d'un bloc no modificat

A continuació s'executa el següent codi:

```
li $t0, 36
li $t1, 7
sw $t1, 0($t0)
```

L'adreça 36 correspon a la segona paraula del bloc 4 d'MP (línia 0 d'MC). El que succeirà al subsistema de memòria serà el següent:

1. Tot i que la línia 0 té el bit V = 1 també té etiqueta = 0, diferent a l'etiqueta que correspon a l'adreça (etiqueta 1). Per tant, es produirà una fallada d'escriptura.

2. Donat que la línia té el bit D = 0, copiarem el bloc 4 d'MP a la línia 0 directament, reemplaçant el seu contingut.

3. Escriurem únicament a MC i marcarem la línia com a modificada (D = 1).

La Figura 35 mostra l'estat final de l'MC i l'MP.

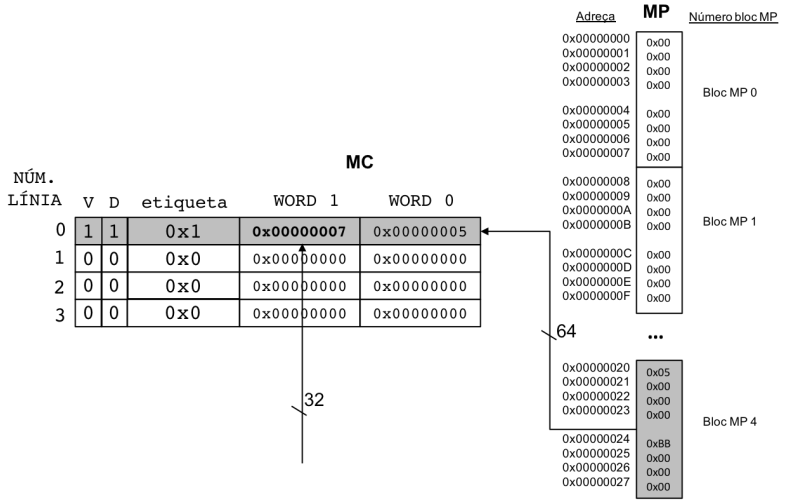


Figure 35: Estat final de l'MC i l'MP després d'una escriptura amb fallada d'un bloc no modificat.

Miss. Si la línia a reemplaçar no ha estat modificada simplement es porta la nova línia a l'MC, s'escriu la paraula a MC i es marca la línia com modificada.

Escriptura amb fallada d'un bloc modificat

A continuació s'executa el següent codi:

```
li $t0, 0
li $t1, 3
sw $t1, 0($t0)
```

L'adreça 0 correspon a la primera paraula del bloc 0 d'MP. El que succeirà al subsistema de memòria serà el següent:

1. Tot i que la línia 0 té el bit V = 1 també té etiqueta = 1, diferent a l'etiqueta que correspon a l'adreça (etiqueta 0). Per tant, es produirà una fallada d'escriptura.

2. Donat que la línia 0 conté un bloc modificat (D = 1) no podem reemplaçar el seu contingut directament. Ens cal primer copiar el bloc modificat (era el bloc 4 d'MP) a l'MP.

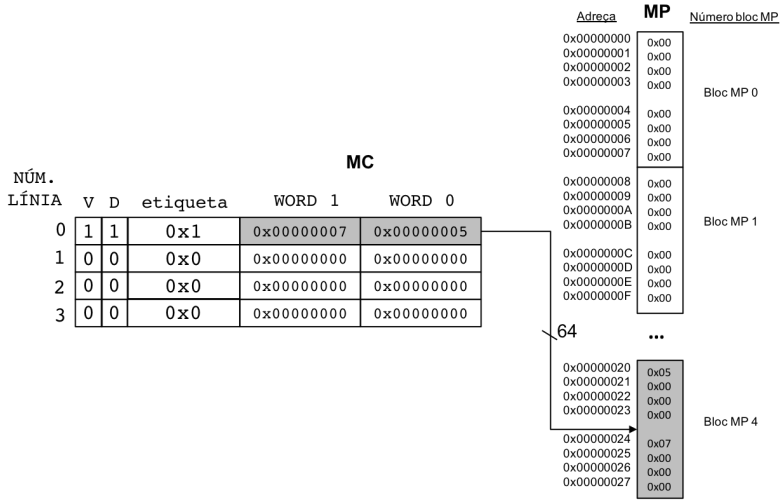


Figure 36: Una escriptura amb fallada d'un bloc modificat proca la transferència del bloc modificat a l'MP.

3. Un cop copiat el bloc modificat a l'MP podem copiar el nou bloc, el bloc 0 d'MP, a la línia 0.

4. Finalment escriurem la dada (1 paraula) només a l'MC.

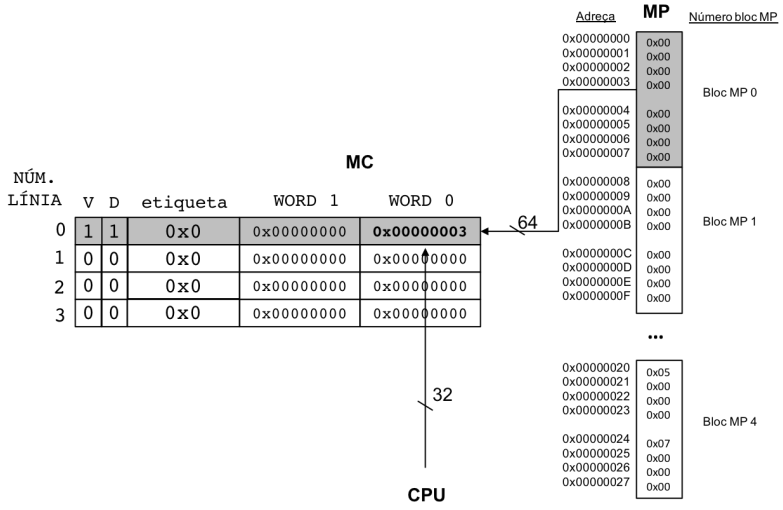


Figure 37: Estat final de l'MC i l'MP després d'una escriptura amb fallada d'un bloc modificat.

Miss. La línia a reemplaçar ha estat modificada, s'escriu a MP. Posteriorment, es porta la nova línia a MC. Finalment, s'escriu la paraula a MC.

2.8 Disseny de la memòria per suportar caches

Hem vist que el temps de penalització de les fallades (t_p) depèn en gran mesura del que fins ara hem anomenat t_{block}. Aquest temps inclou diferents aspectes, com el temps que triga la memòria en llegir/escriure el bloc o el temps que es triga en transferir-lo pel bus. Anem a veure com es dissenya la memòria per poder minimitzar el temps de penalització sense encarir-ne massa la producció.

El nostre sistema amb memòria cache té dues interconnexions principals: per un costat el processador ha de poder transferir dades a/des de l'MC i per l'altre l'MC ha de poder transferir dades a/des de l'MP. Aquestes dues interconnexions es realitzen mitjançant *busos* de dades. Normalment, l'amplada en bits de cada bus de coincidirà amb l'amplada de la sortida de la memòria a la que va connectat. Considerem, de moment, que l'amplada de la sortida de la memòria coincideix amb el que la memòria pot llegir/escriure cada vegada. Més endavant veurem que això no és necessàriament així.

Una primera possibilitat (Figura 38) seria que l'MC, l'MP i els busos tinguessin una amplada d'una paraula. Suposem, per exemple, que tenim els següents paràmetres:

- Temps que triga el processador en enviar l'adreça = 1 cicle.
- Temps que triga l'MP en llegir **una paraula** = 15 cicles (caldrà fer-ho 4 vegades).
- Temps que es triga en enviar **una paraula** pel bus = 1 cicle (caldrà fer-ho 4 vegades).
- Grandària del bloc: 4 paraules.

NOTA: El subsistema de memòria i els seus busos funcionen amb una freqüència de rellotge molt inferior a la del processador. Els cicles que es mencionen en aquest apartat seran cicles de rellotge del subsistema de memòria i, per tant, no els podríem fer servir directament en els càlculs de rendiment que hem vist en apartats anteriors.

En base a aquestes dades, podem calcular el temps necessari per a copiar un bloc d'MP a MC així:

$$t_p = 1 + 4 * 15 + 4 * 1 = 65 \text{ cicles}$$

El nombre de bytes transferits per cicle seria:

$$\frac{4 * 4}{65} = 0.25 \text{ bytes/cicle}$$

Una segona possibilitat (Figura 39) seria disposar d'una MC i una MP de major "amplada", per exemple capaces de llegir/escriure dues paraules de cop. Això ens permetria reduir el nombre de transferències però ens obligaria a ampliar el bus entre l'MP i l'MC proporcionalment, i també a posar un multiplexor entre el processador i l'MC. Refem l'exemple anterior per a aquest disseny:

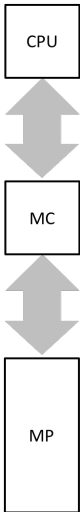


Figure 38: Disseny en el que tots els components (MC, MP i busos) tenen 1 paraula d'amplada.

- Temps que triga el processador en enviar l'adreça = 1 cicle (igual que abans).
- Temps que triga l'MP en llegir **dues** paraules = 15 cicles (caldrà fer-ho 2 vegades).
- Temps que es triga en enviar **dues** paraules pel bus = 1 cicle (caldrà fer-ho 4 vegades).

En base a aquestes dades, podem calcular el temps necessari per a copiar un bloc d'MP a MC així:

$$t_p = 1 + 2 * 15 + 2 * 1 = 33 \text{ cicles}$$

El nombre de bytes transferits per cicle seria:

$$\frac{4 * 4}{33} = 0.48 \text{ bytes/cicle}$$

Tot i que aquest disseny suposa una reducció considerable en el temps necessari per transferir bloc, implica alguns desavantatges. Per un costat, és costós incrementar la grandària del bus. Per altra banda, la lògica de multiplexació podria fer créixer el temps d'accés a la cache (sempre, també quan hi ha encerts).

Una tercera possibilitat (Figura 40) consistirà en fer servir una memòria *entrellaçada*. Fins ara hem suposat que l'amplada de la sortida de la memòria és igual al número de paraules que la memòria pot llegir/escriure cada vegada. En la pràctica això no és necessàriament així. Un chip de memòria pot consistir en b bancs de memòria (per exemple $b = 4$), d'amplada p paraules cadascun

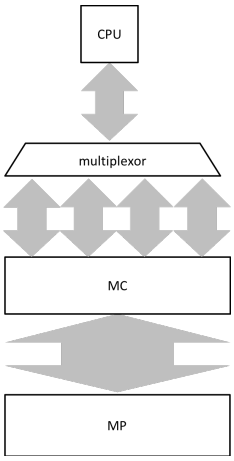


Figure 39: Disseny en el que tots els components (MC, MP i busos) tenen 1 paraula d'amplada.

(per exemple $p = 1$), capaços de treballar simultàniament. Aquesta memòria pot accedir simultàniament a $b * p$ paraules (per exemple 4 paraules), però en comptes de tenir una sortida de $b * p$ paraules pot tenir una sortida de menor amplada (per exemple 1 paraula) de forma que hi puguem connectar un bus menys costós (per exemple d'una paraula d'amplada).

Refem l'exemple anterior per a aquest disseny:

- Temps que triga el processador en enviar l'adreça = 1 cicle (igual que abans).
- Temps que triga l'MP en llegir **quatre** paraules = 15 cicles.
- Temps que es triga en enviar **una** paraula pel bus = 1 cicle (caldrà fer-ho 4 vegades).

En base a aquestes dades, podem calcular el temps necessari per a copiar un bloc d'MP a MC així:

$$t_p = 1 + 1 * 15 + 4 * 1 = 20 \text{ cicles}$$

El nombre de bytes transferits per cicle seria:

$$\frac{4 * 4}{33} = 0.80 \text{ bytes/cicle}$$

Aquest tercer disseny serà el que suposarem d'ara en endavant (memòria entrellaçada i bus de dades d'una paraula).

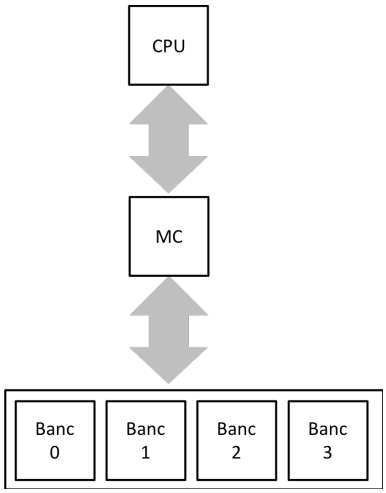


Figure 40: Disseny en el que tots els components (MC, MP i busos) tenen 1 paraula d'amplada.

3 Mesures de rendiment

3.1 Model de temps

Als apartats anteriors, s'han descrit múltiples opcions de disseny d'una memòria cache (polítiques d'escriptura, buffers, etc.). De cara a especificar una configuració estàndard per als exercicis de rendiment que veurem més endavant, a continuació es descriu un *model de temps* que ens permetrà determinar de manera unívoca quants cicles de rellotge trigarà una referència a memòria en cadascuna de les diferents situacions.

A nivell general cal considerar que el temps de servei d'una referència a memòria és el temps en determinar si la referència és un encert o una fallada a memòria cache i servir la referència en cas d'encert (t_h) més el temps de penalització per resoldre la referència en accedir al següent nivell de la jerarquia de memòria (t_p):

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

Cal tenir en compte que:

- L'accés a les etiquetes per comprovar si la referència és un encert i el servei de la referència en cas d'encert es realitzen seqüencialment durant el temps t_h: en la primera meitat d'aquest temps es fa l'accés a les etiquetes i en la segona meitat es fa la lectura o escriptura a memòria cache.

- Per aquelles configuracions de memòria cache amb política d'escriptura immediata es considera l'existència d'un **buffer d'escriptura** amb llargada il·limitada on queden emmagatzemades les escriptures pendents de portar

a MP. També es considera que cap referència a memòria entra en conflicte amb escriptures pendents en aquest buffer. Cal adonar-se que el contingut d'aquest buffer es porta a MP en paral.lel amb l'execució de les instruccions que vénen a continuació de l'accés a memòria.

A continuació es mostra una taula amb el temps de penalització (t_p) associat a una referència segons sigui lectura/escriptura; encert/fallada i segons la memòria cache contempli una política d'escriptura immediata sense assignació/retardada amb assignació:

t_p	Immediata amb assignació (laboratori)	Immediata sense assignació	Retardada amb assignació
Lectura - Encert	0	0	0
Lectura - Fallada	$t_{block} + t_h$ [1]	$t_{block} + t_h$ [1]	bloc modif.: $2 * t_{block} + t_h$ [2] bloc no mod.: $t_{block} + t_h$ [1]
Escriptura - Encert	0 [3]	0 [3]	0 [5]
Escriptura - Fallada	$t_{block} + t_h$ [1]	0 [4]	bloc modif.: $2 * t_{block} + t_h$ [2] bloc no mod.: $t_{block} + t_h$ [1]

[1]Es porta el bloc d'MP i es reinicia l'accés a memòria.
[2]Es porta el bloc modificat a MP; es porta el nou bloc d'MP guardant-lo a l'entrada que es reemplaça i es reinicia l'accés a memòria.
[3]L'escriptura es fa a MC i a MP. L'actualització d'MP a partir del buffer d'escriptura es fa de forma concurrent amb la continuació de la referència a memòria.
[4] L'escriptura es fa únicament a MP. L'actualització d'MP a partir del buffer d'escriptura es fa de forma concurrent amb la continuació de la referència a memòria.
[5] L'escriptura es fa únicament a MC.

Càlcul de t_p

En funció del *model de temps* descrit, tindrem diferents valors de t_p. En el cas de la l'*escriptura immediata sense assignació* el *model de temps* especifica que les fallades d'escriptura no tenen penalització (degut a que suposarem l'ús d'un buffer d'escriptura). Per això, l'única penalització introduïda per les fallades serà la penalització de les fallades de lectura. Si ens diuen que la proporció d'escriptures és p_e:

$$t_p = (1 - p_e) * (t_{block} + t_h)$$

En el cas de la l'*escriptura retardada amb assignació*, suposant que la proporció de blocs modificats és p_m, el *model de temps* especifica la següent penalització de les fallades:

$$t_p = p_m * (2 * t_{block} + t_h) + (1 - p_m) * (t_{block} + t_h)$$

3.2 Mesures de rendiment

Càlcul del temps mitjà d'accés a memòria

Una forma de mesurar el rendiment d'una memòria cache serà el *temps mitjà d'accés a memòria* (Average Memory Access Time o AMAT), t_{ma}. Aquest serà el temps mitjà d'una referència a memòria i el calcularem així:

$$t_{ma} = t_h + m * t_p$$

En funció de la política d'escriptura, el *model de temps* ens indicarà de quina manera haurem de calcular t_p.

De vegades, treballarem amb dues cache diferents, una per les instruccions i una per les dades. En aquests casos, a més de donar-nos valors diferents per als diferents paràmetres per a cadascuna de les cache (t_h, t_{block}, etc.) ens proporcionaran el *número de referències a memòria per instrucció*:

$$nr_{instr} = nr/I$$

nr_{instr} serà sempre més gran que 1, donat que per a tota instrucció es realitza com a mínim un accés a memòria, el fetch. A més, algunes instruccions (per exemple *lw* o *sw*)fan lectures o escriptures de dades. D'aquesta manera tindrem que la proporció de referències de lectura o escriptura de dades:

$$nr_{instr} - 1$$

En aquests casos, el càlcul del t_{ma} seria:

$$t_{ma} = \frac{t_{ma:fetch} + (nr_{instr} - 1) * t_{ma:dades}}{nr_{instr}}$$

Càlcul del temps d'execució en funció de t_p i CPI_{ideal}

Com ja sabem, el nombre de cicles d'execució es calcula:

$$n_{cicles} = n_{ins} \cdot CPI$$

essent n_{ins} el nombre d'instruccions executades i CPI la mitja de cicles de rellotge que tarda cada instrucció. Sovint, ens caldrà calcular n_{cicles} per a un determinat programa sota una varietat de configuracions de memòria, on n_{ins} és constant però el CPI varia segons sigui el nombre de fallades de cache, ja que en cada fallada es tarden t_p cicles addicionals.

Si anomenem n_{cicles_ideal} el que tarda una execució ideal (sense fallades de memòria cache), podem definir:

$$CPI_{ideal} = \frac{n_{cicles\_ideal}}{n_{ins}}$$

D'aquesta manera, podem calcular el temps d'execució t_p així:

$$t_{exe} = n_{ins} \cdot CPI \cdot t_c = (n_{ins} \cdot CPI_{ideal} + n_{fallades} \cdot t_p) \cdot t_c$$

Normalment, no ens donaran directament $n_{fallades}$, sinó el número de referències a memòria per instrucció (nr_{instr}) i la taxa de fallades (m). Amb aquestes dades el càlcul quedaria així:

$$t_{exe} = (CPI_{ideal} + m \cdot nr_{instr} \cdot t_p) \cdot n_{ins} \cdot t_c$$

4 Millores: Associativitat i Multinivell

4.1 Associativitat total o per conjunts

Fins ara hem ubicat els blocs d'MP dins la memòria cache mitjançant la tècnica de *correspondència directa*. Aquesta tècnica ofereix un bon rendiment, ja que podem saber a quina línia correspon 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. Anem a veure ara algunes tècniques alternatives.

4.2 MC completament associativa

Una tècnica totalment diferent és el mètode completament associatiu. Consisteix en ubicar els blocs d'MP a qualsevol posició de la memòria cache que estigui lliure. 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 entrades una per una.

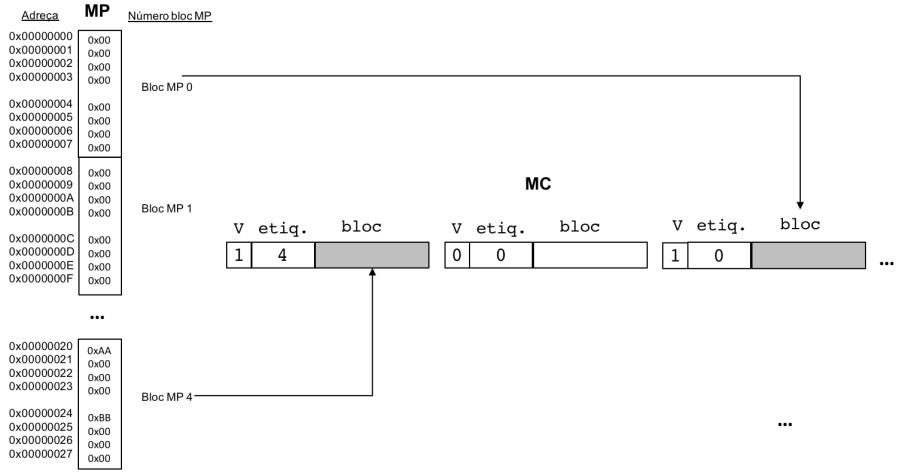


Figure 41: Exemple de memòria cache completament associativa

4.3 MC 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*. Com a les caches de correspondència directa, el número de bloc MP (que trobem a l'adreça) determinarà

a quin *conjunt* ha d'anar el bloc (els c bits del número de bloc MP si hi ha 2^c conjunts). No obstant, dins de cada conjunt no només hi haurà espai per a un bloc, sinó que hi podrem encabir n blocs. cadascun dels blocs que admetrà un conjunt l'anomenarem *via*. Si tenim n vies parlarem d'una cache *associativa d'n vies* (*n-way associative*). Dins d'un conjunt, la ubicació d'un bloc a una via o una altra no dependrà de l'adreça, sinó que s'agafarà la primera via que estigui lliure (després veurem que es fa si no n'hi ha cap). És a dir, el conjunt el determinarà l'adreça, com en correspondència directa, però la via dins el conjunt es decidirà en funció de la disponibilitat, com en el mètode completament associatiu.

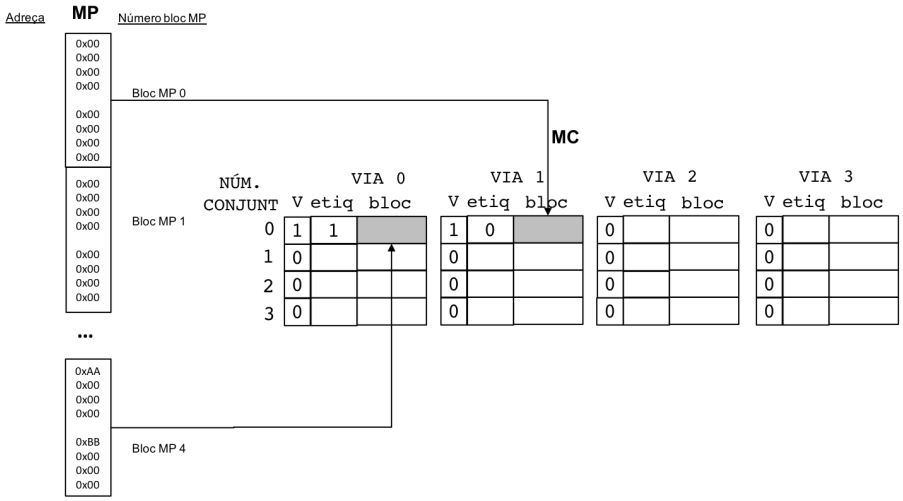


Figure 42: Exemple de memòria cache associativa de 4 vies i 4 conjunts (blocs de dos paraules).

L'anàlisi de l'adreça (veure Figure 43) en una cache associativa per conjunts la podem fer de la mateixa manera que en una cache de correspondència directa però tenint en compte només una via.

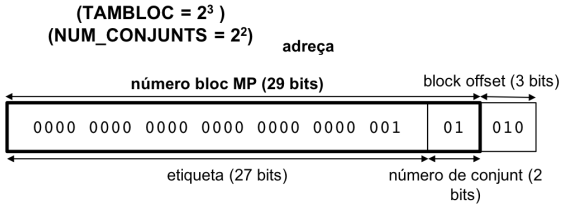


Figure 43: Anàlisi de l'adreça en una memòria cache associativa de 4 vies i 4 conjunts (blocs de dos paraules).

4.4 Reemplaçament LRU (Least Recently Used)

Ara ja sabem que en una cache totalment associativa o en una associativa per conjunts (dins el conjunt corresponent) ubicarem el bloc a l'espai que estigui lliure. Però què farem quan no quedi espai lliure? L'algorisme més utilitzat és l'anomenat *Least Recently Used* (LRU), consistent en reemplaçar el bloc que faci més temps que no s'utilitza. La implementació d'aquest algorisme serà fàcil quan hi hagi només dues vies per conjunt (o només dues entrades en una cache completament associativa), ja que amb un bit per conjunt ho podrem gestionar. No obstant, en el cas general, quan tinguem més de dues vies per conjunt, la implementació serà més complexa. Existeixen altres algorismes de reemplaçament (per exemple l'aleatori) però, per defecte, nosaltres suposarem LRU.

4.5 Exemple

Imaginem-nos que tenim una cache associativa per conjuts amb 4 conjunts, 2 vies, i blocs de 2 paraules. Imaginem-nos que, en un moment donat, l'MC i l'MP tenen els continguts que es mostren a la Figura 44:

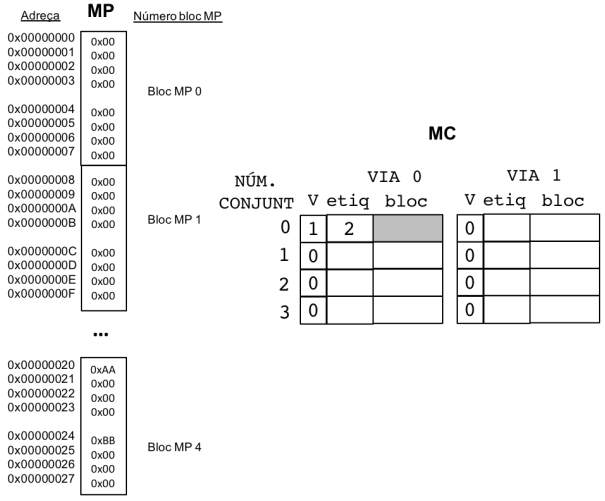


Figure 44: Estat inicial de l'MC i l'MP.

A continuació s'executa el següent codi:

```
li $t0, 0
lw $t1, 0($t0)
```

L'adreça 0 correspon a la primera paraula del bloc 0 d'MP, que alhora correspon al conjunt 0 de l'MC (veure Figura 24). Donat que el conjut 0 encara té una

via lliure (la 1), el bloc serà ubicat en aquesta via. La Figura 45 mostra l'estat final de l'MC i l'MP.

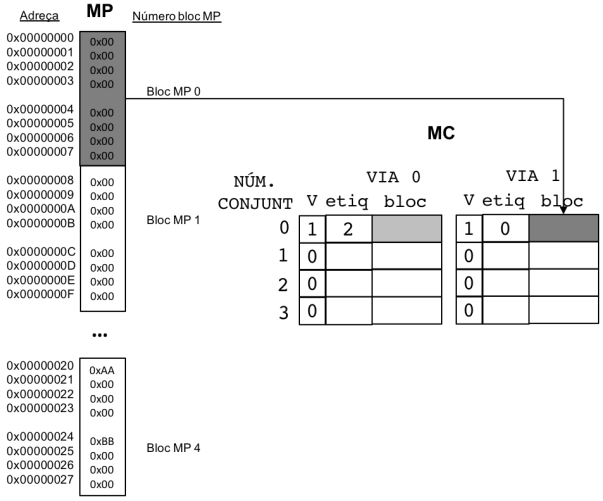


Figure 45: Estat final de l'MC i l'MP.

A continuació s'executa el següent codi:

```
li $t0, 32
lw $t1, 0($t0)
```

L'adreça 32 correspon a la primera paraula del bloc 4 d'MP, que també correspon al conjunt 0 de l'MC. Ara ens trobem que no hi ha cap via lliure i hem de decidir quin bloc reemplaçar. L'algorisme LRU ens diu que reemplacem el que fa més temps que no s'utilitza, per tant reemplaçarem el bloc que hi ha a la via 0. La Figura 46 mostra l'estat final de l'MC i l'MP.

4.6 Diagrama de blocs

La Figura 47 mostra el diagrama de blocs d'una cache associativa per conjunts amb 4 vies, 256 conjunts i blocs d'una paraula. S'observa que calen 4 comparadors per determinar quina via (si n'hi ha cap) conté la mateixa etiqueta que la del bloc 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 la dada entre les quatre vies.

És fàcil veure que, si incrementéssim en nivell d'associativitat del disseny anterior, necessitaríem més comparadors. Si a més ho féssim sense incrementar la capacitat total, ens caldria reduir el nombre de conjunts i per tant, s'incrementaria també el nombre de bits de l'etiqueta. Per aquest motiu, una

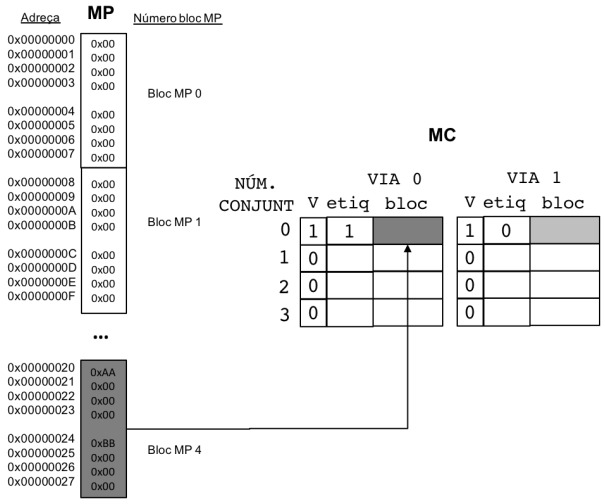


Figure 46: Estat final de l'MC i l'MP.

major associativitat, que ens ajuda a reduir taxa de fallades (m), implica, com a contrapartida, un increment del temps de servei en cas d'encert (t_h).

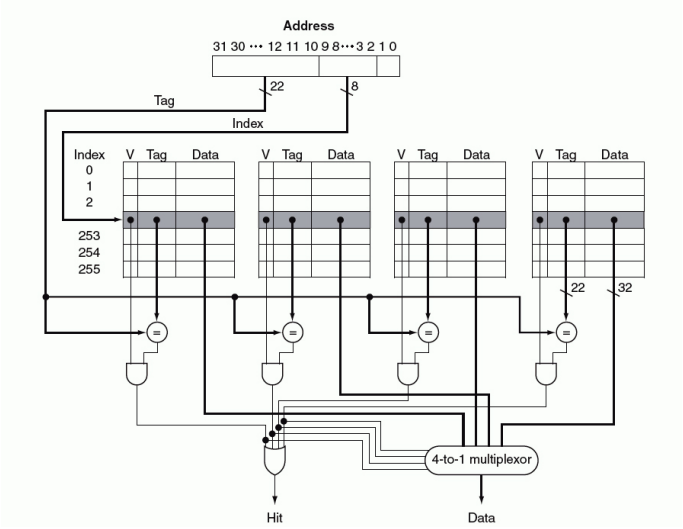


Figure 47: Diagrama de blocs d'una cache associativa per conjunts amb 4 vies, 256 conjunts i blocs d'una paraula. Font: David A. Patterson and John L. Hennessy, Computer Organization and Design: the Hardware/Software Interface, 5th ed., Morgan Kaufmann, 2012.

4.7 Reducció de la penalització de fallada: cache multinivell

Avui dia, molts processadors inclouen un segon nivell de memòria cache (L2 cache). Donada una referència a memòria, s'intentarà primer resoldre a la cache de primer nivell (L1 cache). Si es produeix una fallada, s'anirà a buscar el bloc a la cache de nivell 2, en comptes d'anar-lo a buscar a la memòria principal. Si el bloc no és a la cache de nivell 2, aleshores s'haurà d'anar a buscar a la memòria principal. La penalització de fallada de la cache de nivell 1 serà el temps d'accés a la cache de nivell 2. La Figura 48 mostra el diagrama d'un processador amb dos nivells de memòria cache.

Però, quin avantatge té això? Tenir dos nivells diferents ens permetrà orientar el disseny de cadascuna de les caches a un objectiu diferent. El disseny de la cache de nivell 1 l'orientarem a reduir el temps d'encert Això ho aconseguirem amb:

- Un nivell baix d'associativitat.
- Blocs de poques paraules.
- Poca capacitat.

Aquestes decisions de disseny, per contra, incrementaran molt la taxa de fallades i serien inviables si cada fallada impliqués un accés a la memòria principal. El fet de tenir una segona cache ho fa possible, donat que podem orientar el disseny

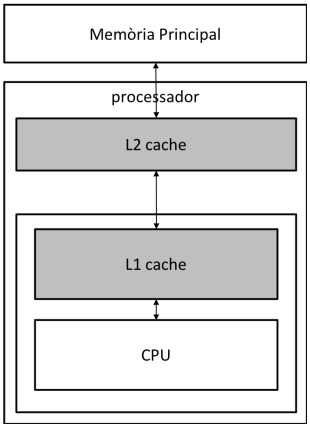


Figure 48: Cache multinivell.

d'aquesta precisament a minimitzar la taxa de fallades. Per aquest motiu, la cache de nivell 2 té:

- Un nivell alt d'associativitat.
- Blocs grans.
- Molta capacitat.

Hi ha múltiples opcions de disseny per a les caches multinivell. De cara als exercicis, per simplificar, donarem per fet que tots els accessos són de lectura i també que la cache de nivell 2 és sempre *inclusiva* (conté sempre tots els blocs de la cache de nivell 1).

Avui dia és freqüent també tenir dues caches de nivell 1 diferenciades, una per les instruccions i una per les dades. En un processador multi-nucli (multi-core), cada nucli (core) disposa de la seva pròpia cache de nivell 1, mentre que la cache de nivell 2 acostuma a ser compartida entre els diferents nuclis (veure Figura 49).

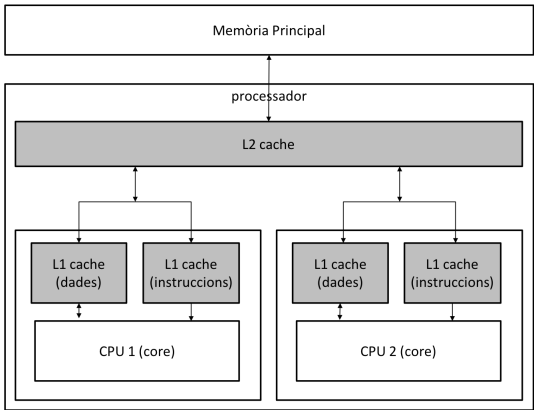


Figure 49: Cache multinivell en un processador multi-nucli.