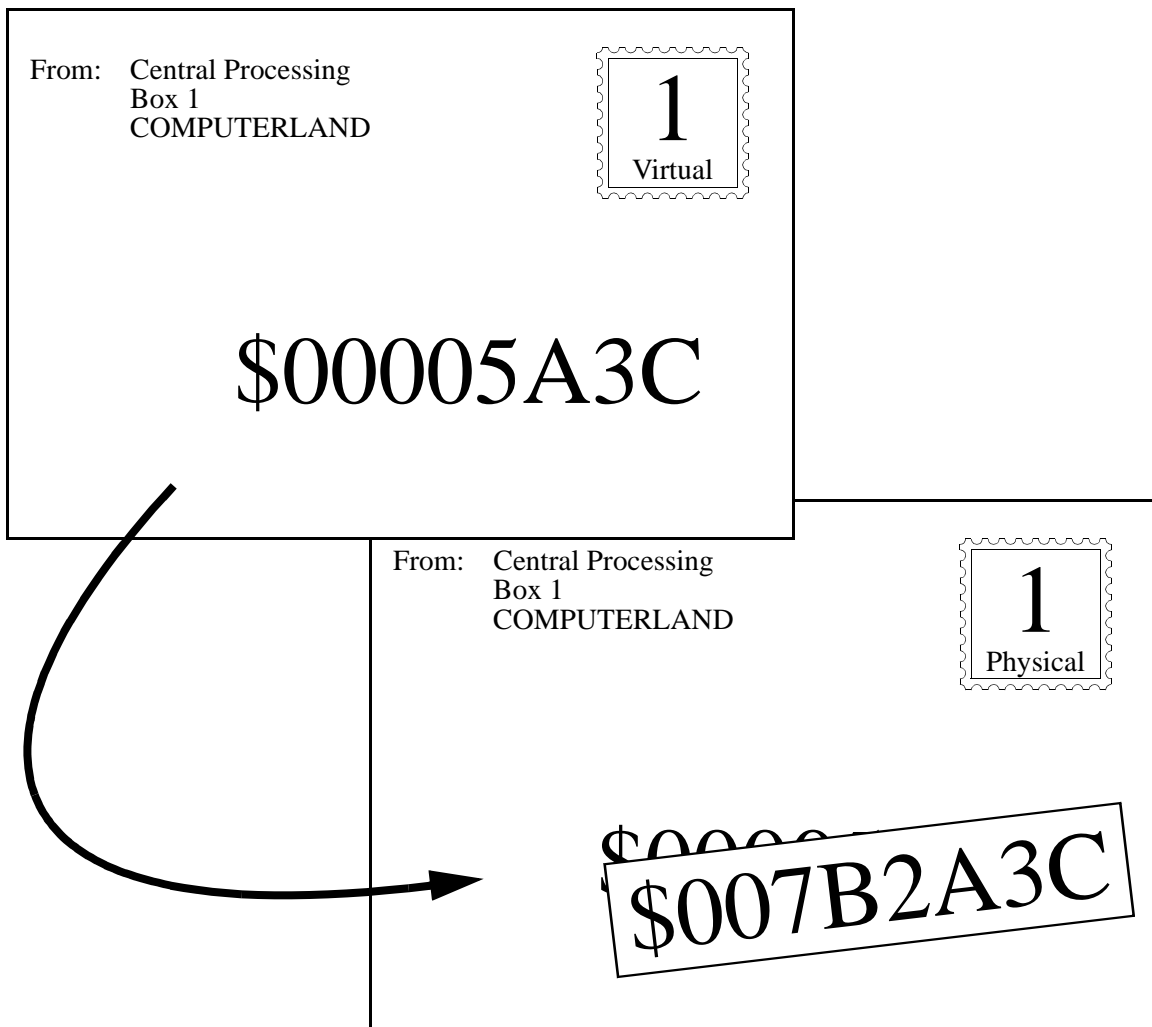


# Cacheminne och adress- översättning



# Innehåll

---

## Cacheminnen

Cacheminnen i korthet . . . . .	5
Det stora, snabba minnet som inte finns . . . . .	5
Processor med cacheminne . . . . .	6
Mer om lokalitet. . . . .	6
Separata instruktions- och datacacheminnen . . . . .	7
Cacheminnet som kombinatorisk krets . . . . .	7
Administrativ information i cacheminnet . . . . .	8
Cacheminnen med flera platser än en . . . . .	8
Cacheminnets parametrar. . . . .	9
Byte-adresserat minne . . . . .	10
Alignment. . . . .	10
Byte-adresseringens följder . . . . .	11
Byte-ordningens problem. . . . .	11
Problem med full associativitet . . . . .	12
Begränsad associativitet. . . . .	12
Ett enkelt och effektivt cacheminne. . . . .	12
Parametrarna hos vårt cacheminne. . . . .	13
Utvärdering av direktmappat cacheminne . . . . .	13
Delvis associativa cacheminnen. . . . .	14
Att utnyttja lokalitet i rummet . . . . .	15
Kombinationer av parametrar . . . . .	16
Multiplexering . . . . .	16
Begränsningar i val av parametrar . . . . .	17
Skrivpolitik: skrivträffar. . . . .	17
Skrivpolitik: skrivmissar . . . . .	18
Sub-block placement . . . . .	19
Write-before-hit . . . . .	19
Skrivmissar: sammanfattning. . . . .	20
Skrivmissar och skrivträffar. . . . .	20
Skrivbuffertar . . . . .	20
Skrivbuffertvarianter . . . . .	21
Sammansmältning . . . . .	21
Exempel på sammansmältning. . . . .	22
Flera ord per skrivbuffertplats . . . . .	22
Skrivning i datorer med separata instruktions- och datacacheminnen. . . . .	23
Flera enheter på bussen . . . . .	23
Primärminnet sett som cacheminne . . . . .	24
Andranivåcacheminne . . . . .	24

---

## Adressöversättning

Adressöversättning i korthet . . . . .	25
Minnesanvändning . . . . .	25
Motiv för adressöversättning . . . . .	26
Livet utan adressöversättning: ett ensamt program. . . . .	26
Livet utan adressöversättning: flera program . . . . .	27
Swapping. . . . .	27
Sidindelning . . . . .	28
Översättningstabell . . . . .	28
Mappning . . . . .	29
Mappningar kan man ha flera av. . . . .	29
Virtuellt minne . . . . .	30
Motiv för sidtabell med flera nivåer . . . . .	31
Exempel på sidtabell med flera nivåer . . . . .	31
Minnesbesparing i trenivåers sidtabell . . . . .	32
Exempel på referens i trenivåers sidtabell . . . . .	32
Cacheminne för adressöversättning . . . . .	33
Virtuell och fysisk adressering . . . . .	33
Tricket . . . . .	34
Trickets konsekvenser. . . . .	34
TLB-missar . . . . .	35
Skilda TLB:er för instruktioner och data . . . . .	35
En utökad minneshierarki . . . . .	36
Tumregel för minneshierarkin . . . . .	36
En stor minneshierarki . . . . .	37



# Cacheminnen

## Cacheminnen i korthet

De flesta program (men inte alla) uppvisar *lokalitet*. Lokalitet innebär att vissa delar av programkod och data används mycket och andra delar lite eller inte alls.

Om de program- och datadelar som används mycket placeras i ett extra snabbt (och dyrt) minne, så går programmet fortare. Hela programmet och alla data finns dessutom i ett billigare, långsamt minne.

Detta kan höja prestanda mycket, utan att höja kostnaderna mer än lite grann.

Det extra snabba minnet kallas cacheminne efter franskans *cache* som betyder "stoppa undan". Cacheminnesmetoden tillämpas på många olika ställen i en dator.

En bra kompilator (för till exempel C eller Pascal) försöker upptäcka eller gissa vilka variabler i programmet som används mest, och ha dessa variabler i processorns snabba register.

Ett bra operativsystem använder en del av primärminnet för data som nyligen lästs från eller skrivits till skivminnet. Ofta behövs samma data snart igen, och primärminnet är mycket snabbare än skivminnet.

En bra mikroprocessor har ett litet, snabbt minne för information från de senaste läsningarna och skrivningarna i primärminnet. Även här kan samma data behövas flera gånger, och det lilla minnet i mikroprocessorn är mycket snabbare än primärminnet.

Det är detta sista fall, med mikroprocessorn, som vi ska koncentrera oss på just nu.

## Det stora, snabba minnet som inte finns

Anta att vi vill bygga en snabb dator, som klarar att utföra  $10^8$  instruktioner per sekund (det vill säga 100 Mips). Då ska  $10^8$  instruktioner per sekund hämtas från datorns minne. Om instruktionerna hämtas en i taget får varje minnesåtkomst alltså ta 10 ns.

Ett minne med en åtkomsttid på 10 ns är snabbt idag. Snabba minnen är dyra, upptar stor kiselyta och har hög effektförbrukning.

Placerar man minnet på ett annat chipp än processorn, så behövs dessutom extra tid, kiselyta och effekt för att driva anslutningsledningarna mellan chippen. Helst ska alltså minnet finnas på processorchippet.

Detta talar för att ett snabbt minne bör vara litet. En rimlig storlek är  $10^5$  bit (16 kbyte), annars blir processorchippet orimligt stort.

Tyvärr är de flesta program idag mycket stora i förhållande till det minne som kan få plats på processorchippet. Att starta textredigeringsprogrammet Emacs kräver åtminstone  $10^7$  bit (1 Mbyte). Större operativsystem (Unix) har minnesbehov i samma storleksordning. En dators primärminne bör därför rymma åtminstone  $10^8$  bit (16 Mbyte).

Så stora minnen kan köpas till rimliga priser, men åtkomsttiden är då 35–50 ns bara i själva minneskretsarna. Överföringen från minne till processor tar också tid, och den totala åtkomsttiden kan närma sig 200 ns.

Om vår dator hämtar sina instruktioner en i taget från ett så långsamt minne, så sjunker hastigheten till 5 Mips.

## Processor med cacheminne

Vi har alltså att välja mellan ett snabbt minne som är för litet eller ett stort minne som är för långsamt. Denna obehagliga situation klarar vi med cacheminnesmetoden: vi installerar båda minnena samtidigt.

Vi placerar ett litet snabbt cacheminne på processorchippet. Dessutom bygger vi ett stort (men långsamt) primärminne utanför processorchippet.

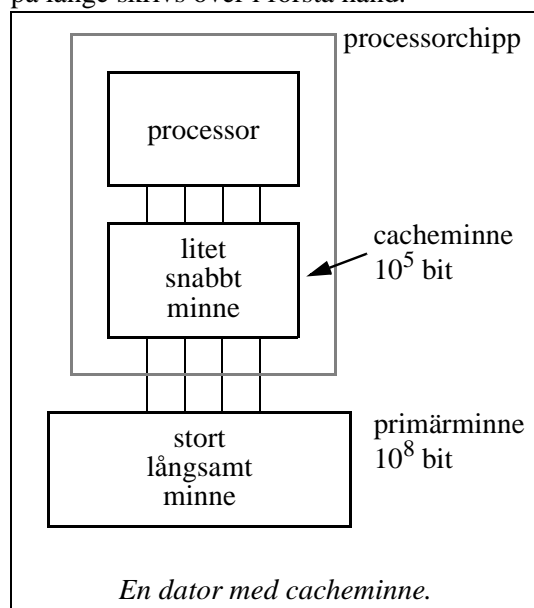
Varje gång processorn hämtar en instruktion från primärminnet (vilket tar lång tid) så arkiveras en kopia i cacheminnet.

Ligger instruktionen inne i en loop (slinga) så behövs den snart igen. Då finns instruktionen redan i det snabba cacheminnet och kan hämtas därifrån.

Motsvarande gäller för läsning och skrivning av data (variabler och liknande). När en variabels värde hämtas från primärminnet, så sparas en kopia i cacheminnet.

Behövs variabelns värde igen så finns det i cacheminnet.

När cacheminnet blir fullt så återanvänds platserna. Cacheminnet konstrueras så att instruktioner och variabler som inte använts på länge skrivs över i första hand.



## Mer om lokalitet

Luddigt uttryckt innebär lokalitet att programmet oftast läser och skriver på ett fåtal minnesadresser i närheten av varandra.

Begreppet lokalitet blir tydligare om man skiljer på lokalitet i tiden (*temporal locality*) och lokalitet i rummet (*spatial locality*).

Hämtning av instruktionerna inne i en loop är ett exempel på lokalitet i tiden: *en minnesadress som nyligen har använts kommer snart att användas igen.*

Med lokalitet i rummet menas att *när en viss minneadress har lästs så läses snart en intilliggande adress.*

Återigen kan instruktionshämtning tas som exempel. Det normala när en instruktion har hämtats är att nästa instruktion snart ska hämtas. Detta mönster bryts ju bara vid hopp i programmet.

Alla cacheminnen utnyttjar lokalitet i tiden: de sparar innehållet i minnesadresser som just lästs eller skrivits, så att värdet finns kvar om det behövs snart igen.

De flesta cacheminnen utnyttjar också lokalitet i rummet. Cacheminnet konstrueras då så att block om 4–32 intilliggande minnesord<sup>†</sup> lagras tillsammans.

När processorn vill läsa något ord i ett block som inte finns i cacheminnet, så hämtas hela blocket från primärminnet. Om nästa ord i nummerordning snart behövs, så är det redan hämtat.

Ett cacheminne blir ganska snabbt fullt vid normal programkörning. Då behövs något sätt att välja ut vad som ska skrivas över när ny information ska in.

En fungerande metod är att skriva över det block som legat oanvänt under längst tid. Har blockets inte lästs eller skrivits på länge så är det inte så sannolikt att det behövs snart igen. Vi återkommer till detta (på sida 9).

<sup>†</sup> Ännu större block förekommer, men är inte vanliga i cacheminnen som sitter på ett processorchipp.

## Separata instruktions- och datacacheminnen

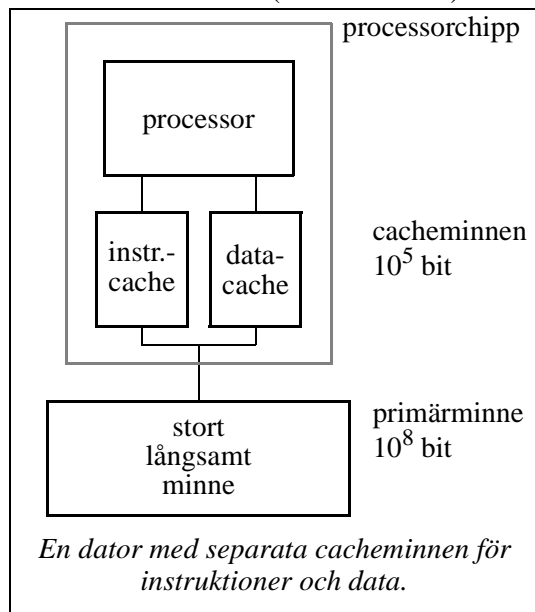
Det är värt att notera att instruktionshämtningar följer ett mycket regelbundet mönster: efter en instruktion hämtas nästföljande. Detta gäller alla instruktioner utom hoppinstruktioner.<sup>†</sup> Läsningar och skrivningar av data inträffar inte alls lika regelbundet.

I många datorer har man olika cacheminnen för instruktioner och data.<sup>‡</sup> Orsaken är att ett cacheminne bara kan göra en sak i taget.

Med gemensamt cacheminne kan ingen instruktion hämtas när läsning eller skrivning av data sker, det vill säga varje gång en LOAD- eller STORE-instruktion utförs. Denna fördröjning av instruktionshämtningen påverkar prestanda ganska mycket eftersom LOAD och STORE är så vanliga.

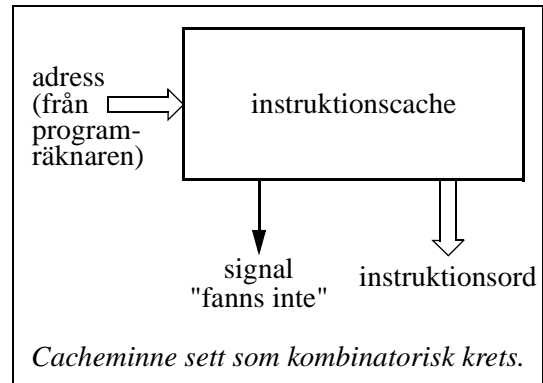
Med separata instruktions- och datacacheminnen görs alla instruktionshämtningar via instruktionscachen. Övriga referenser (på grund av LOAD, STORE och liknande instruktioner) går via datacachen.

Vi ska återkomma till separata instruktions- och datacacheminnen (sida 23 och 24).



<sup>†</sup> Mellan var tredje och var tionde instruktion som utförs är en hoppinstruktion.

<sup>‡</sup> Detta kallas ofta Harvard-arkitektur, efter en serie datorer byggda vid Harvard University i USA åren 1945—1952. Dessa hade helt separata minnen för data och instruktioner.



### Cacheminnet som kombinatorisk krets

Till att börja med koncentrerar vi oss på instruktionshämtningen.

Vi tänker försöka bygga vår snabba dator så att den arbetar med en klockfrekvens på 100 MHz och hämtar en ny instruktion från cacheminnet varje klockperiod.

Så länge alla instruktioner som ska hämtas finns i cacheminnet kan vi betrakta cacheminnet som en kombinatorisk krets, ett läsminne (ROM) om man så vill.

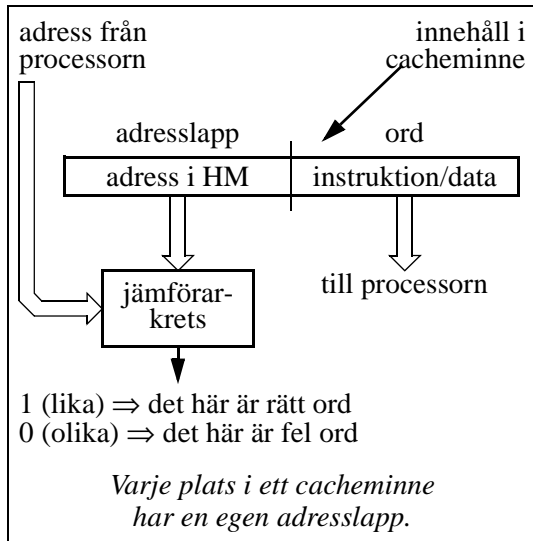
Processorn levererar en adress till cacheminnet och får tillbaka antingen ett instruktionsord eller en signal om att det önskade ordet inte fanns i cacheminnet.

Fallet när processorn får det den begär kallas för *träff* (engelska: hit). Fallet när det önskade ordet inte finns i cacheminnet kallas *miss* (samma ord på engelska).

Vid en miss måste instruktionshämtningen stoppas medan ordet hämtas från primärminnet. Då ska ordet också läggas in i cacheminnet så att det finns där nästa gång det behövs.

Resonemanget ovan gäller också för läsning av data. Enda skillnaden är att datahämtning inte sker lika ofta som instruktionshämtning. Skrivning av data är mera komplicerad; detta tar vi upp senare i texten.

Cacheminnet betar sig alltså för det mesta som ett vanligt läsminne: en adress läggs på ingången och efter en stund presenteras data på utgången. Det är bara vid behandlingen av en miss som cacheminnet inte kan betraktas på detta sätt, som en ren kombinatorisk krets.



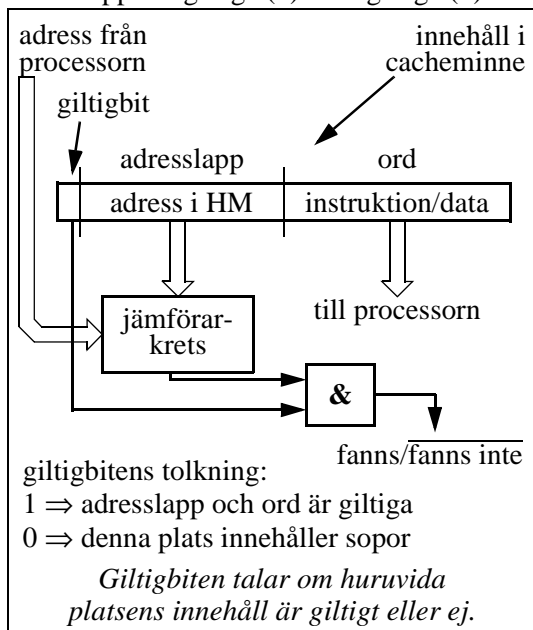
**Administrativ information i cacheminnet**

Varje plats i cacheminnet behöver innehålla en del administrativ information, förutom data- eller instruktionsord. För varje plats behövs åtminstone en *adresslapp* (på engelska: tag) och en *giltigbit* (valid bit).

På adresslappen står den adress i primärminnet som ordet hämtats från. En och samma cacheminnesplats kan ju svara mot olika primärminnesadresser vid olika tillfällen.

När strömmen slås på till datorn finns ingen giltig information i cacheminnet. Det finns också andra tillfällen när vissa ord i cacheminnet måste markeras som ogiltiga.

Cacheminnet har därför en giltigbit för varje plats; giltigbiten anger helt enkelt om ord och adresslapp är ogiltiga (0) eller giltiga (1).



**Cacheminnen med flera platser än en**

Hittills har vi talat allmänt om cacheminnets egenskaper, och beskrivit hur en plats i cacheminnet kan se ut. Nu ska vi fundera över ett cacheminne med flera platser.

Längst ned på sidan ses blockschemat för ett cacheminne med n platser.

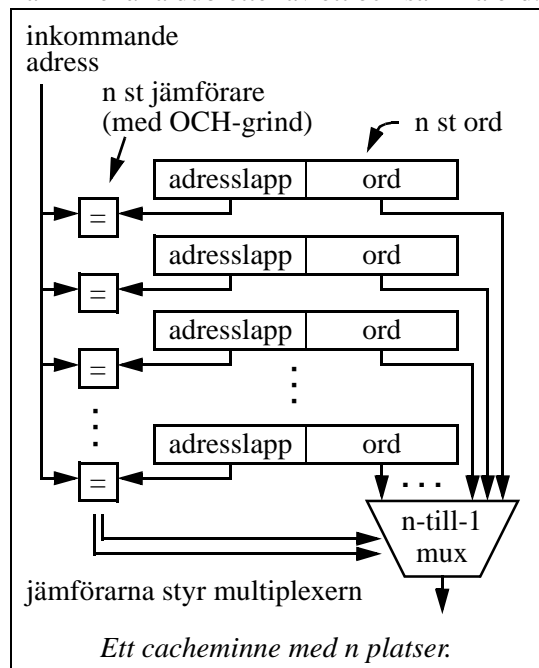
Varje plats har en adresslapp och en jämförare. Observera att varje plats också har en giltigbit med tillhörande OCH-grind, som dock inte har ritats ut.

När processorn begär läsning av en adress, undersöks samtliga cacheminnesplatser på en gång. Alla jämförarkretsarna jämför en och samma adress (den sökta) med innehållet i den egna adresslappen.

Om någon plats innehåller rätt ord, så kommer jämföraren att ge 1:a ut. Detta styr n-till-1-multiplexern så att innehållet i just denna plats levereras till processorn.

Observera att det inte ska kunna inträffa att två av jämförarna anger likhet. Det skulle ju betyda att samma information finns på två ställen i cacheminnet, vilket vore slöseri med dyrbart minnesutrymme.

Den maskinvara som fyller cacheminnet vid en miss konstrueras alltid så att cacheminnet aldrig kan innehålla dubletter av ett och samma ord.



## Cacheminnets parametrar

Vissa uppgifter om ett cacheminne är grundläggande och kommer med i varje diskussion om olika cacheminnets för- och nackdelar. Dessa uppgifter är:

- Storleken, som anges i byte.
- Blockstorleken, som också anges i byte.
- Associativitetstalet, ett dimensionslöst tal.
- Utbytesalgoritmen, ett slags program.
- Skrivpolitiken, som är ett kapitel för sig.

*Storleken* anger hur mycket användbar information cacheminnet rymmer. De administrativa uppgifterna (adresslapp och giltigbit) ingår *inte*.

Detta får den intressanta konsekvensen att två cacheminnen med samma storlek kan kräva helt olika antal minnesbitar när man ska bygga dem.

Vårt cacheminne med  $n$  platser innehåller  $4n$  byte information, eftersom varje instruktions- eller dataord är 4 byte (32 bit).

Om även adresserna är 32 bit stora så krävs ytterligare 33 bit för varje plats, till adresslapp och giltigbit. Det totala antalet minnesbitar som behövs för att bygga cacheminnet blir då  $65n$ .

*Blockstorleken* anger hur mycket användbar information varje plats innehåller. Varje instruktions- eller dataord är som sagt 4 byte, så blockstorleken i vårt cacheminne är 4 byte.

Ur storlek och blockstorlek kan man räkna fram antalet platser i cacheminnet:

$$\text{antal platser} = \frac{\text{storlek}}{\text{blockstorlek}}$$

Vi ser att formeln stämmer för vårt cacheminne med  $n$  platser, ty

$$\frac{\text{storlek}}{\text{blockstorlek}} = \frac{4n}{4} = n$$

*Associativitetstalet* anger hur många av cacheminnets adresslappar som undersöks vid en sökning. I vårt  $n$ -platsers cacheminne undersöker vi alltid alla  $n$  adresslapparna, varför associativitetstalet är  $n$ .

Associativitetstalet kan aldrig bli större än antalet platser. Specialfallet att associativitetstalet är lika med antalet platser kallas *full associativitet*.

Man kan fråga sig vad som händer om man inte undersöker alla adresslappar vid sökning. Vi ska återkomma till detta.

*Utbytesalgoritmen* ska svara på frågan: "vilken cacheminnesplats ska skrivas över, när cacheminnet är fullt och något nytt ska in?".

Några utbytesalgoritmer är:

- Äldsta åtkomst (LRU, least recently used). Cacheminnet bokför tidpunkt för senaste läsning (eller skrivning) för varje plats, och väljer att byta ut innehållet på den plats som varit oläst längst tid.
- Först in, först ut (Fifo, first in first out). Cacheminnet skriver över de data som är äldst, även om de har använts nyligen.
- Slumpmässigt utbyte (random replacement). Platsen väljs slumpmässigt. Detta ger ett jämnt utnyttjande av platserna i cacheminnet.

Äldsta-åtkomst-metoden (LRU) är den intuitivt mest tilltalande med tanke på lokalitet i tiden. Om ett block som lästs nyligen snart kommer att läsas igen, så lär ett block som legat oläst länge inte behövas särskilt snart.

Tyvär kräver LRU komplicerad bokföring, vilket gör den opraktisk för cacheminnen som ju ska byggas med enkla digitala kretsar.

Fifo-principen borde vara näst bäst, men alla undersökningar visar att det inte är så. Den leder nämligen till att gammal information som fortfarande används konsekvent kastas ut före nyare information som inte används.

Slumpmässigt utbyte är i allmänhet bättre än Fifo-metoden (men sämre än LRU). Detta kan motiveras intuitivt på följande sätt.

Om cacheminnet är rimligt stort, så kan det indelas i en mindre del som används mycket, och en större del som används lite. Väljs nu en plats slumpmässigt, så är chansen störst att den ligger i den stora delen — som ju används lite.

Numera används ibland approximationer till LRU-algoritmen. Ett exempel är non-MRU (non-Most Recently Used).

Vid non-MRU reduceras bokföringen så att cacheminnet bara håller reda på vilken plats som användes senast. När det är dags för utbyte väljer man slumpmässigt bland de övriga platserna.

## Byte-adresserat minne

En av orsaken till att storlek och blockstorlek mäts i byte är att de flesta datorer idag är *byte-adresserade*.

Vi ska ägna ett par sidor åt detta begrepp, så att vi ser hur byte-adressering påverkar cacheminnets konstruktion.

Den som redan vet allt om byte-adressering, alignment och byte-ordning kan hoppa framåt till sida 12.

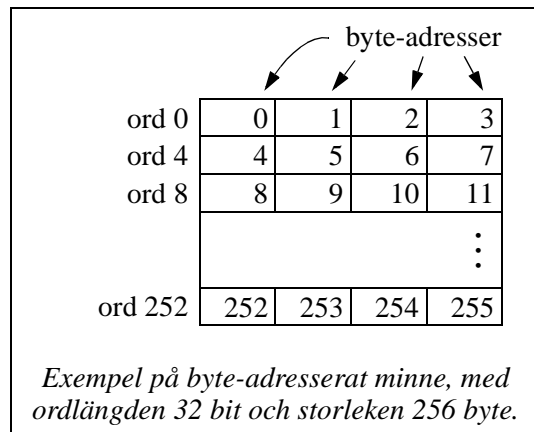
Byte-adressering innebär att den minsta adresserbara enheten i minnet är en åttabits-grupp, en så kallad *byte*, (vilket uttalas "bajt" med kort a-ljud).

Detta innebär inte alls att ordlängden hos minnet (eller hos datorns register) är 8 bit, tvärtom. Dagens datorer har vanligen ordlängder på 32—64 bit.

Ändå adresseras alltså minnet med en byte-adress. En 32-bits läsning läser alltså 4 byte i följd från minnet; en 64-bits läsning läser 8 byte i följd.

Det främsta motivet för byte-adresserat minne är att det snabbar upp program för hantering av text, som ju brukar representeras som en följd av byte.

I bilden här nedanför ser vi ett exempel på byte-adresserat minne. En 32-bits läsning på adress 4 medför alltså att byte nummer 4, 5, 6 och 7 hämtas från minnet.



## Alignment

Ofta (men inte alltid) har datorn krav på *alignment* (svengelska) av byte-adresserna, så att adressen vid exempelvis en 4-bytes (d v s 32-bits) läsning eller skrivning måste vara jämnt delbar med 4.

Om minnet och databussen är 32 bit breda så kan maskinvaran i processorn bara göra *alignade* läsningar. Är adressen inte jämnt delbar med 4 så måste processorn läsa två ord och plocka ut lämpliga delar av dessa för att kunna leverera rätt data.

Skrivningar ger ytterligare problem eftersom bara delar av två olika ord ska uppdateras vid en icke-*alignad* skrivning. De båda orden måste läsas, ändras och skrivas tillbaka.

Detta gör icke-alignade referenser så komplicerade att man gärna väljer att inte utföra dem i maskinvara. I stället gör då processorn ett felavbrott. Felavbrottsrutinen kan simulera den icke-alignade referensen med en lämplig sekvens av alignade referenser.

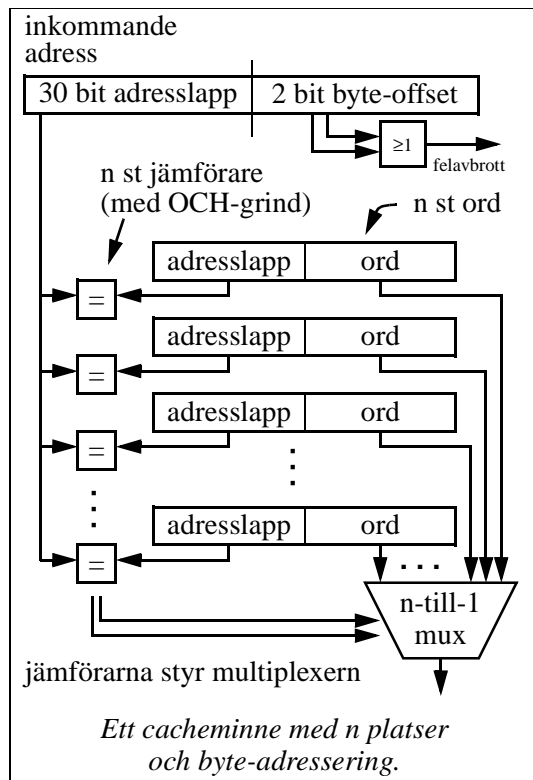
Olika datorer har olika stora krav på alignment. I Motorola 68000 ska den minst signifikanta adressbiten vara noll vid 16- eller 32-bits referenser. Vid 8-bitsreferenser kan alla adressbitar ha valfria värden.

Idag finns datorer anpassade för 128-bits referenser. Tillverkaren rekommenderar att alla referenser i första hand görs *naturligt alignade* (se tabellen nedan), och helst också alignade så att de 4 minst signifikanta adressbitarna är noll.

Om inte tillräckligt många av de minst signifikanta bitarna är noll så är referensen *unaligned* (inte alignad). Då måste två ord skarvas ihop på det sätt som beskrevs ovan.

ordlängd hos referensen	minst signifikanta adressbitar
8 bit	vilka som helst
16 bit	0
32 bit	00
64 bit	000
128 bit	0000

*Adressens utseende för naturligt alignade referenser.*



### Byte-adresseringens följder

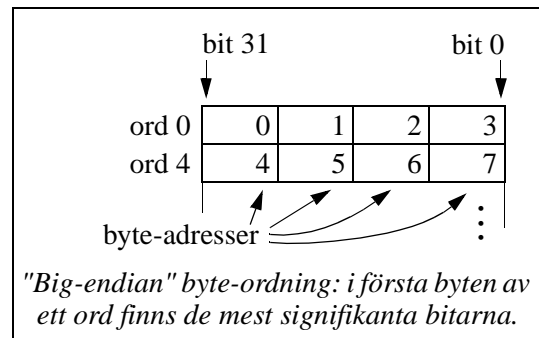
Figuren här ovanför visar hur vårt cacheminne med  $n$  platser ser ut sedan vi byggt om det för byte-adressering. Vi antar att data- och instruktionsord är 32 bit stora. I och med detta ska de två minst signifikanta bitarna vara noll för naturligt alignade referenser.

Vi väljer att göra felavbrott för icke-alignade referenser. Vårt cacheminne kräver alltså att de två minst signifikanta bitarna i adressen är nollställda. Om adresserna är 32 bit stora, så får vi 30 bit kvar som ska jämföras med adresslapparna.

Naturligtvis väljer vi då adresslapparnas storlek till 30 bit. Vi sparar alltså 2 bit på varje adresslapp. Denna besparing har inte så mycket med cacheminnet att göra egentligen. Den beror enbart på att primärminnets maximala storlek har minskat.

En 32-bits byte-adress pekar ut en av  $2^{32}$  8-bitsgrupper. Ett fullt utbyggt minne rymmer då  $8 \cdot 2^{32}$  bit, det vill säga  $2^{35}$  bit.

Tidigare förutsatte vi i stället att varje 32-bits adress pekade ut ett unikt 32-bits ord, varvid ett fullt utbyggt minne kan rymma  $32 \cdot 2^{32}$  bit, det vill säga  $2^{37}$  bit.



### Byte-ordningens problem

Så länge som en adress bara behöver peka ut en adresserbar enhet, så uppstår aldrig några tvivel om hur resultatet ska tolkas. Så till exempel är byte nummer 5 i figuren här ovanför en högst väldefinierad bitgrupp.

Vill vi däremot hämta ett 32-bits ord från adress 4 så uppstår en tveksamhet. Vi vet att vi ska hämta byte nummer 4—7.

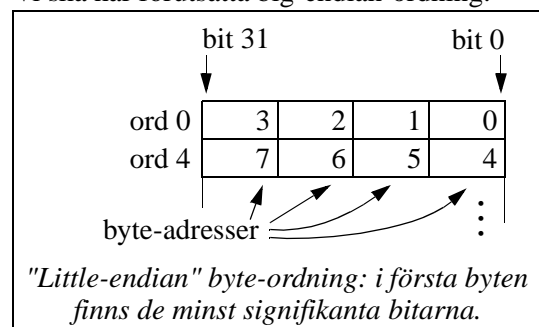
Men i vilken ordning ska 4 byte placeras i ett 32-bits register? Frågan kan också formuleras så här: *innehåller den första byten (nummer 4 i exemplet) de mest signifikanta bitarna eller de minst signifikanta?*

Figuren här ovanför visar hur det ser ut när de mest signifikanta bitarna finns i första byten av ett ord. Denna byte-ordning kallas *big-endian* eftersom de "stora", mest signifikanta bitarna kommer först.

Det motsatta förhållandet visas i figuren här nedanför. Detta kallas *little-endian* eftersom de "små", minst signifikanta bitarna finns på den första adressen (nummer 4 i exemplet).

Byte-ordningen är en petitesse. Tyvärr kan den ställa till stora problem när data kopieras mellan datorer med olika byte-ordning.

Vi ska här förutsätta big-endian-ordning.



### Problem med full associativitet

Fullt associativa instruktions- och data-cacheminnen är ovanliga idag. De är nämligen svåra att bygga, och inte så mycket bättre än cacheminnen med begränsad associativitet.

Huvudproblemet är de bussar som behövs. Den inkommande adressen ska distribueras till ett stort antal ( $n$  stycken) jämförarkretsar. Det kräver en buss med starka drivkretsar.

Starka drivkretsar är stora (tar plats på chippet) eller långsamma (en liten och klen drivkrets behöver god tid på sig för att driva busen till rätt nivåer).

När jämförelsen väl har gjorts ska rätt block väljas ut med multiplexern. Den ska ha en ingång per cacheminnesplats. Grindar med fler än 5–10 ingångar existerar inte, varför multiplexern också måste byggas som en buss.

Varje plats har då en egen drivkrets med tri-stateutgångar<sup>†</sup> som slås på om jämförarkretsen anger likhet. Även dessa drivkretsar är stora eller långsamma. Jämförarkretsarna själva tar naturligtvis också plats på chippet.

<sup>†</sup> Utgångar som kan stängas av med en särskild signal. En avstängd utgång är högimpediv och påverkar inte bussen. Tri-state är ett registrerat varumärke för National Semiconductor Corporation.

### Begränsad associativitet

Associativitetstalet anger hur många adresslappar som undersöks samtidigt (se sida 9). För varje adresslapp som ska undersökas behövs en jämförarkrets, så associativitetstalet anger också antalet jämförarkretsar.

Anta nu att ett cacheminne har färre jämförare än platser. Vid varje sökning måste då en delmängd av platserna väljas ut för adresslappsjämförelse.

Om det sökta ordet finns i cacheminnet så *måste* den plats där det finns komma med vid valet. Detta utesluter slumpval och liknande.

Den metod som används, och som fungerar bra, är att använda en del av den inkommande adressen för att peka ut en plats, eller en grupp av platser, i cacheminnet.

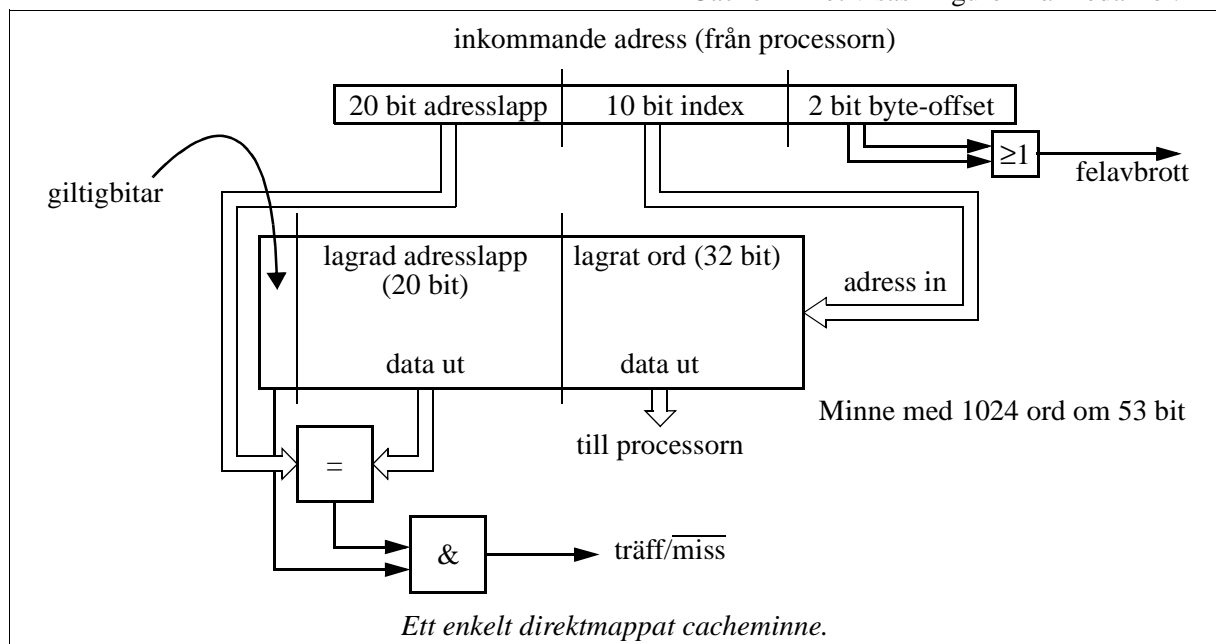
Låt oss studera hur detta kan se ut!

### Ett enkelt och effektivt cacheminne

Det cacheminne som beskrivs här är i någon mening motsatsen till det fullt associativa.

Vi låter även här adresser samt data- och instruktionsord vara 32 bit långa. Cacheminnesstorleken bestäms till 4096 byte (4 kbyte) och blockstorleken till 4 byte. Minnet är byte-adresserat.

Cacheminnet visas i figuren här nedanför.



De två minst signifikanta bitarna av den inkommande adressen ska vara noll, annars är referensen icke-alignad.

10 bitar av adressen används som index i cacheminnet. Dessa bitar går direkt till adressgångarna på en vanlig minneskrets.

Minneskretsen innehåller 1024 ord om 53 bit. Varje 53-bits ord är uppdelat i tre delar: 32 bit data (eller instruktion), 20 bit adresslapp och 1 giltigbit.

Är giltigbiten ettställd så är datadelen av cacheminnesordet giltig. Stämmer dessutom adresslappen med den inkommande adressens 20 mest signifikanta bitar så är datadelen en kopia av det sökta minnesordet, och cacheminnet signalerar "träff" (cache hit).

Om giltigbiten är noll, eller adresslappen i minnet skiljer sig från den inkommande, så har cacheminnet ingen kopia av det sökta ordet. Då måste detta ord hämtas från primärminnet. Hämtningen sköts av ett sekvensnät, som inte finns med i figuren. Dessutom behövs datavägar som inte heller är utritade.

När ett ord hämtas ser sekvensnätet till att de 20 mest signifikanta bitarna av primärminnesadressen lagras i cacheminnesordets adresslappsfält.

! Observera att de 10 adressbitar som utgör index inte behöver lagras i adresslappen. Platsen i cacheminnet avgör entydigt vilka värden dessa bitar har.

### Parametrarna hos vårt cacheminne

Låt oss nu undersöka parametrarna hos denna cacheminneskonstruktion. Storlek och blockstorlek har vi redan bestämt.

Eftersom bara en jämförare finns, och bara en adresslapp undersöks vid sökning, är associativitetstalet 1. Detta specialfall kallas *direkt-mappat* (direct-mapped) cacheminne.

Utbytesalgoritmen blir trivial. En del av den inkommande adressen används direkt som adress till cacheminnets minneskretsar. Någon ytterligare valmöjlighet finns inte.

Alltså kan varje ord i primärminnet bara placeras på *en* bestämd plats i cacheminnet. När ett nytt ord ska in måste det nya ord ligga på en viss plats och det som fanns på den platsen förut kastas följaktligen ut.

### Utvärdering av direktmappat cacheminne

Direktmappade cacheminnen har en framträdande nackdel: varje ord i primärminnet kan bara placeras på exakt ett ställe i cacheminnet. Om två ofta använda ord ligger på adresser med samma indexdel, så kan inte båda finnas i cacheminnet samtidigt.

Om två subrutiner finns i minnesceller med samma index, så tvingas cacheminnet att kasta ut den ena subrutinen och hämta in den andra varje gång en av rutinerna anropas.

Subrutinen måste alltså läsas in från primärminne till cacheminne varje gång den körs. Detta fenomen kallas *thrashing* (bra svenskt ord saknas). Samma fenomen kan drabba datareferenser.

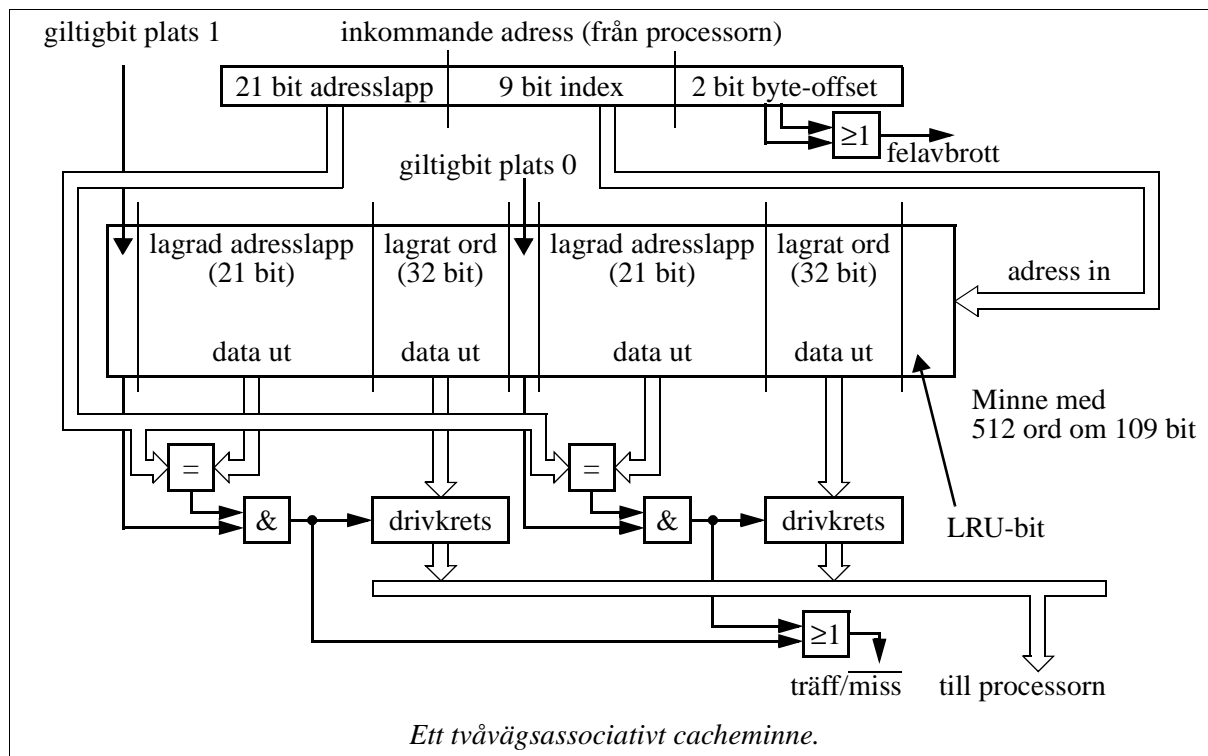
Mer allmänt kan direktmappade cacheminnen drabbas av att cacheminnets utrymme inte utnyttjas helt. Vissa platser med "fel" index kan förbli oanvända, medan andra används mycket.

Ändå används direktmappade cacheminnen mycket i dagens processorer. Ett huvudskäl är att direktmappade cacheminnen kan göras snabbare än associativa.

Ett direktmappat cacheminne kan leverera sitt ord till processorn redan innan jämförarkretsen har jämfört färdigt. Signalen om att det levererade värdet är giltigt (eller ogiltigt) kan komma lite senare, vilket kan spara flera nanosekunder. Processorn konstrueras då så att inga registerinnehåll ändras innan jämförelsen är klar.

De inbesparade nanosekunderna kan direkt användas för att förkorta processorns klockcykel, så att hela processorn går snabbare. Detta uppväger mer än väl att processorn oftare får vänta på primärminnesläsningar.

Det direktmappade cacheminnet är dessutom enklare att bygga och kräver mindre kisel, vilket gör hela processorn billigare.



### Delvis associativa cacheminnen

Det finns mellanformer mellan direktmappade och fullt associativa cacheminnen.

Vi ska illustrera detta genom att ändra vårt direktmappade cacheminne så att associativitetstalet blir 2 i stället för 1. Efteråt är cacheminnet inte direktmappat längre, utan *tvåvägsassociativt* (two-way set associative).

Associativitet 2 innebär att varje ord i primärminnet kan placeras på två platser i cacheminnet. Detta betyder att två platser, med varsin adresslapp och giltigbit, ska lagras på varje index i cacheminnet.

Minneskretsarnas bredd ska alltså fördubblas. Detta innebär en fördubbling av storleken, om vi inte samtidigt halverar antalet möjliga index.

Vi väljer att hålla storleken konstant, och halverar alltså antalet möjliga indexvärden. Detta innebär att en adressbit flyttas från index till adresslapp, se bilden här ovanför.

Inkommande index pekar nu ut två platser i cacheminnet. De båda platserna har varsin adresslapp, och båda adresslapparna måste

jämföras med adresslappsdel av den inkommande adressen. För det behövs två jämförarkretsar.

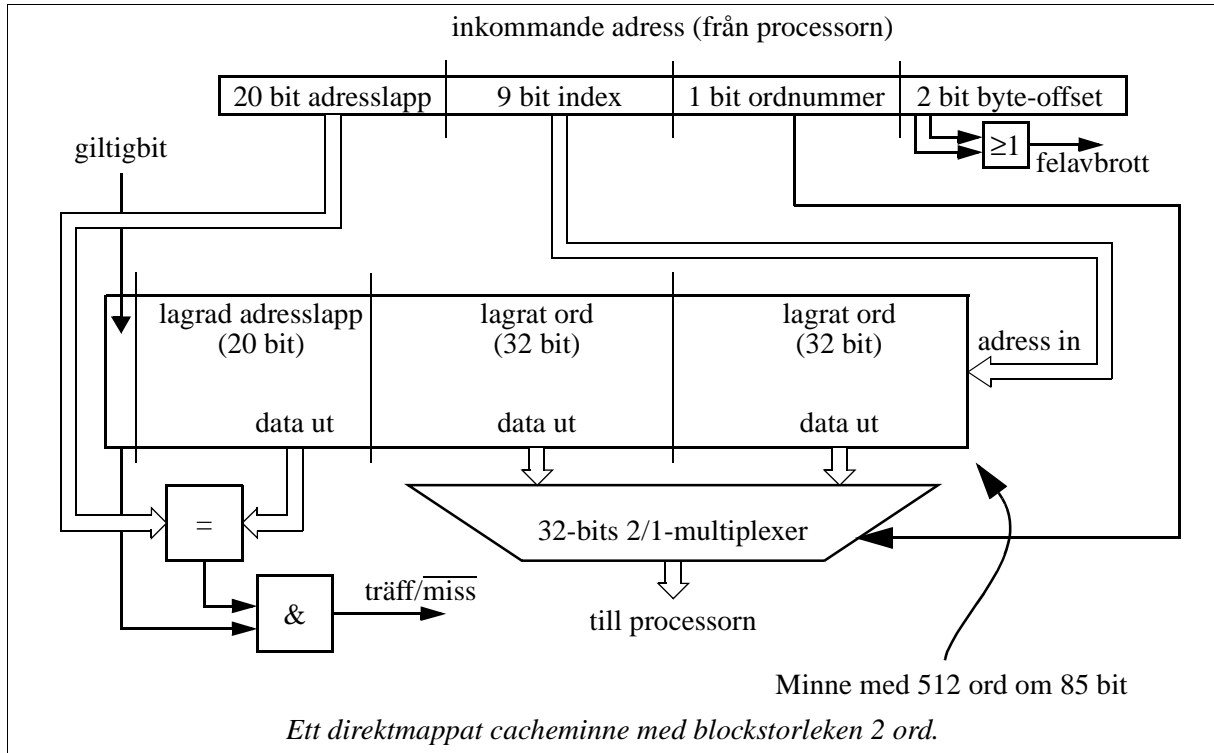
Om någon av jämförarna anger likhet ska innehållet i den ena platsen skickas vidare till processorn. I figuren används en drivkrets för varje plats. Drivkretsen styrs av jämförarkretsens utsignal (och av giltigbiten).

Det går att ersätta drivkretsarna med en multiplexer; den utför samma sak men har annan kodning av sina styrsignaler.

Eftersom vårt cacheminne inte längre är direktmappat är utbytesalgoritmen inte längre trivial. Väljer vi LRU så måste varje index förses med information om vilken av dess platser som användes senast, näst senast, näst-näst senast, och så vidare.

I ett tvåvägsassociativt minne räcker en bit för att ange vilken plats som använts senast. Denna bit visas längst till höger i minnet i figuren. Vid större associativitetstal än 2 blir administrationen besvärlig. LRU brukar då ersättas med någon approximation, se sida 9.

Mängden platser med rätt index kallas *set* på datorsvengelska. *Set size* är därför ett annat namn på associativitetstalet. Ett direktmappat cacheminne har *set size* = 1, och så vidare.



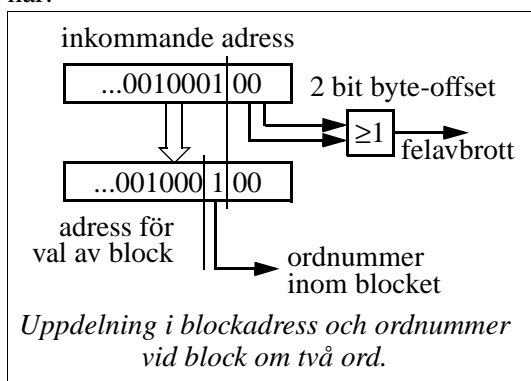
### Att utnyttja lokalitet i rummet

I beskrivningen har vi hittills förutsatt att cacheminnesplatserna bara rymmer ett ord. Vi nämnde dock att lokalitet i rummet kan utnyttjas om cacheminnet i stället hanterar block om flera ord.

Det betyder att en del av adressen ska användas för att peka ut ett visst ord inom det utvalda blocket. Antalet ord i varje block måste därför vara en jämn tvåpotens.

Anta att vi väljer block om två ord. Cacheminnet hanterar då alltid två ord i taget, och varje tvåordsblock upptar *en* cacheminnesplats med *en* adresslapp och *en* giltigbit.

För att välja ord inom ett block åtgår då en adressbit, förutom de två som anger bytenummer inom ett ord. Uppdelningen blir så här:



Bilden här ovanför visar ett direktmappat cacheminne med 2 ord i varje block. Cacheminnets datadel är lika stor som tidigare.

Notera hur uppdelningen av den inkommande adressen har ändrats. Antalet indexbitar i vårt ursprungliga direktmappade cacheminne var 10.

Vårt tidigare tvåvägsassociativa cacheminne hade två ord (med varsin adresslapp) på varje index. För att behålla konstant storlek halverade vi antalet möjliga index och minskade därmed antalet indexbitar till 9.

Det direktmappade cacheminnet med 2-ordsblock har också två lagrade ord på varje index, men orden i ett block delar på en adresslapp. För valet mellan de två orden används i stället en del av adressen. När blockstorleken ökas från 1 ord till 2, så flyttas därför en bit från index till ordnummer.

Vi kan notera att multiplexern, som väljer rätt ord ur blocket, styrs direkt av den inkommande adressen. Multiplexern kan därför göras snabbare än multiplexeringen i ett associativt cacheminne.

I de associativa minnet styrs multiplexeringen av jämförarkretsarna, vars resultat inte är tillgängligt omedelbart. Detta bidrar till att göra associativa cacheminnen något långsammare än direktmappade.

### Kombinationer av parametrar

Nu har vi sett hur ett direktmappat cacheminne kan ändras till ett tvåvägsassociativt. Vi har också sett hur blockstorleken hos ett cacheminne kan ändras från 1 ord till 2 ord.

Om båda dessa ändringar görs tillsammans får vi cacheminnet i figuren längs ned på denna sida.

Varje index i detta cacheminne innehåller två platser med var sin adresslapp och giltigbit. Två jämförarkretsar finns, en för vardera platsen.

Utsignalerna från de två jämförarkretsarna används för att välja från vilken plats som data ska skickas till processorn.

På varje plats finns två lagrade ord. En bit ur den inkommande adressen används för att välja mellan dessa två.

Cachestorleken ska fortfarande hållas konstant. Eftersom det nu finns fyra lagrade ord per index ska antalet index minskas till 256.

Antalet indexbitar i den inkommande adressen blir 8. En bit väljer ord inom blocket och som vanligt har vi en 2-bits byte-offset som ska vara noll.

### Multiplexering

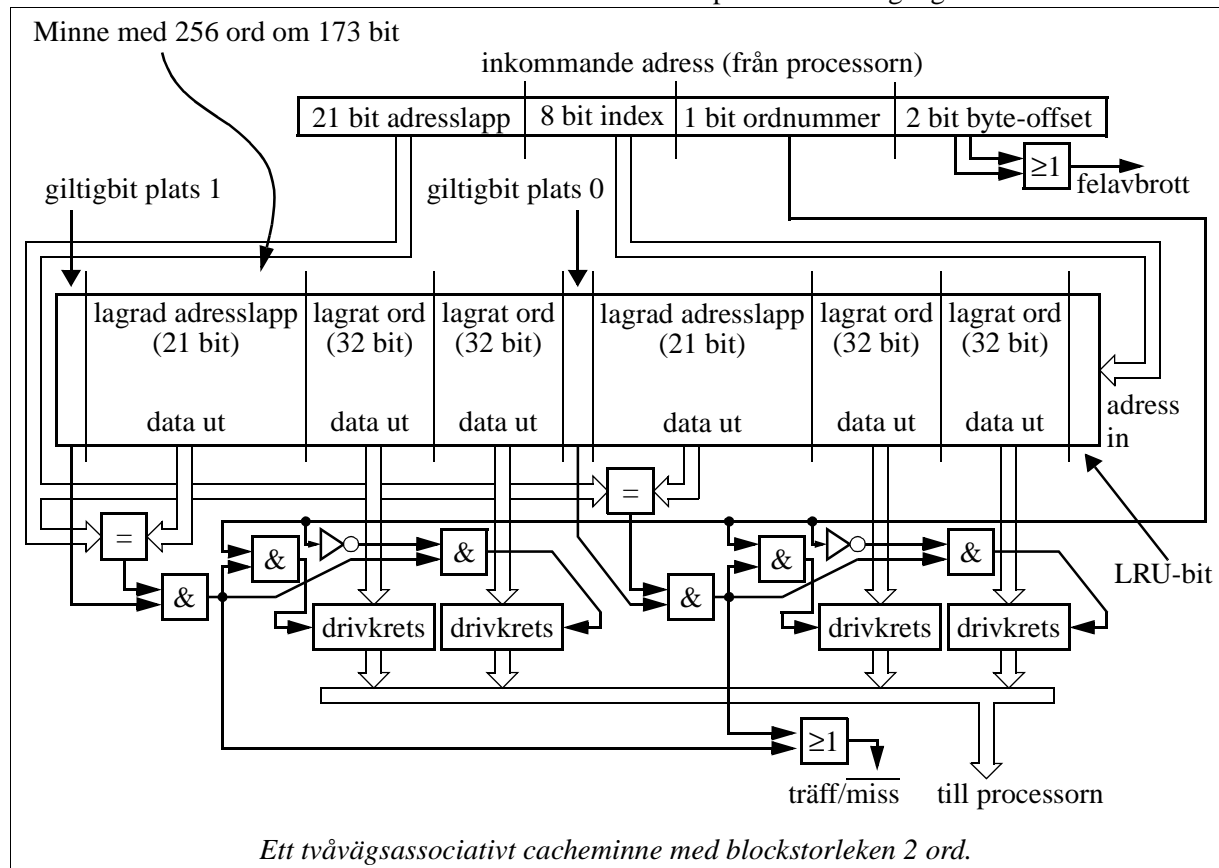
Vi har två olika val som ska göras av maskinvaran vid träff: *vilken plats* och *vilket ord*.

Multiplexering kan allmänt göras antingen med multiplexorkretsar eller med drivkretsar. Det tvåvägsassociativa cacheminnet på sida 14 har drivkretsar; cacheminnet med blockstorleken 2 ord på sida 15 har multiplexorkretsar. Funktionen är likartad och man kan välja det alternativ som är bäst i andra avseenden. I cacheminnet här nedanför sker all multiplexering med drivkretsar.

Låt oss anta att vi får träff i plats 1. Då kommer jämförarkretsen längst till vänster att ge 1:a ut. Giltigbiten är 1 (det var ju en träff) och OCH-grinden längst till vänster ger 1:a ut den också.

OCH-grindens utsignal kan öppna någon av drivkretsarna i det vänstra paret, som hör till plats 0. Vilken av dessa två drivkretsar det slutligen blir beror på ordnumret i den inkommande adressen.

Om ordnumret är 0 så blockeras den vänstra drivkretsen. Den högra öppnas eftersom OCH-grinden som styr dess enable-signal får 1 på båda sina ingångar.



## Begränsningar i val av parametrar

I alla cacheminnen som inte är fullt associativa används en del av adressen som index i något slags minneskrets. Antalet möjliga index är då

$$2^{\text{antal indexbitar}}$$

Om blockstorleken är större än 1 ord så ska en del av adressen välja ut ett av orden i varje block. Detta medför att

$$\text{blockstorlek} = 2^{\text{antal ordvalsbitar}}$$

Varje index måste peka ut (minst) en hel cacheminnesplats. Detta kan också formuleras som att en cacheminnesplats inte kan vara utspridd över flera index i cacheminnet. Följaktligen måste antalet platser vara proportionellt mot antalet index.

Vi har sett att både ökad blockstorlek och ökat associativitetstal medför att flera informationsord lagras på varje index. Det korrekta uttrycket för ett cacheminnes storlek är

$$\text{storlek} = \text{blockstorlek} \cdot \alpha \cdot 2^{\text{antal indexbitar}}$$

där  $\alpha$  är associativitetstalet. Eftersom

$$\text{antal platser} = \frac{\text{storlek}}{\text{blockstorlek}}$$

så gäller också att

$$\text{antal platser} = \alpha \cdot 2^{\text{antal indexbitar}}$$

Eftersom alla ingående värden i ekvationerna måste vara heltal, så kan inte ett cacheminne ha vilka associativitetstal som helst om antalet platser är givet.

Traditionellt har storleken brukat vara en jämn tvåpotens. Eftersom blockstorleken *måste* vara en jämn tvåpotens, så blev också antalet platser en jämn tvåpotens. Två jämna tvåpotenser dividerade med varandra ger ju alltid en ny.

I uttrycket

$$\text{antal platser} = \alpha \cdot 2^{\text{antal indexbitar}}$$

kan vi nu notera att även associativitetstalet  $\alpha$  måste vara en jämn tvåpotens för att det hela ska gå ihop. Detta har lett till att mycken tankevärdhet har ägnats åt att undersöka associativitetstalen 2, 4, 8, ...

Idag förekommer exempelvis associativitetstalen 3 och 5 i aktuella, högpresterande mikroprocessorer.

## Skrivpolitik: skrivträffar

Frågan om vad som händer vid skrivning är relativt oberoende av de övriga parametrarna. Vi ska dela upp svaret i två fall.

- **skrivträff** (write hit): skrivningen sker till ett ord som finns i cacheminnet.
- **skrivmiss** (write miss): skrivningen sker till ett ord som inte finns i cacheminnet.

Vid en skrivträff är frågan om skrivningen ska uppdatera primärminnet, eller om bara cacheminnesinnehållet ska ändras.

Om en skrivning alltid ska ändra i primärminnet är cacheminnet av *genomskrivningstyp* (write-through). Med denna typ av cacheminnen är primärminnesinnehållet alltid giltigt; cacheminnet kan dock innehålla en kopia för snabb åtkomst.

Nackdelen med genomskrivning är att skrivningar till primärminnet tar lång tid. I vissa fall skrivs och läses en variabel flera gånger i snabb följd. Då skulle det vara effektivare att bara ändra variabeln i cacheminnet, och vänta med att uppdatera primärminnet.

Om primärminnet *inte* uppdateras vid varje skrivning så är cacheminnet av *återskrivningstyp* (write-back). I ett sådant cacheminne kan en plats innehålla data som är aktuellare än de som finns i primärminnet. Platsens innehåll måste därför skrivas tillbaka till primärminnet innan det skrivs över (när ny information ska in).

I cacheminnen av återskrivningstyp har varje plats en extra bit som talar om huruvida platsens innehåll ändrats sedan det hämtades in till cacheminnet. Ett block som inte har ändrats är *rent* (clean) och kan kastas utan att skrivas tillbaka till primärminnet.

Har ett eller flera ord i blocket ändrats är blocket *smutsigt* (dirty) och måste kopieras till primärminnet innan dess cacheminnesplats kan återanvändas. Den bit som talar om huruvida blocket är rent eller inte kallas *dirty bit*.

### Skrivpolitik: skrivmissar

Vid en skrivmiss finns ingen plats reserverad i cacheminnet för det ord som ska skrivas. Valmöjligheterna är nu betydligt flera.

En möjlighet är att reservera plats i cacheminnet för det som ska skrivas och hämta in det block som ordet tillhör. Därmed omvandlas skrivmissen till en skrivträff och kan behandlas på samma sätt som en sådan. Detta kallas *fetch-on-write* (hämtning vid skrivmiss).

En annan möjlighet är att endast skriva till primärminnet, utan att cacheminnet påverkas. Efter skrivningen finns ordet bara i primärminnet, och en senare läsning kommer att ge miss. Detta alternativ kallas *write-around*.

Låt oss nu i detalj studera de olika alternativ som finns.

*Platsreservation vid skrivmiss* (allocate on write miss) innebär att en plats väljs ut. I ett direktmappat cacheminne är detta trivialt; i associativa cacheminnen är det aningen krångligare. Är cacheminnet av återskrivningstyp kan platsens gamla innehåll behöva skrivas tillbaka till primärminnet.

När platsen är reserverad (och dess gamla innehåll vid behov återskrivet) ändras platsens adresslapp så att den anger adressen för det som ska skrivas.

Write-around, som nämndes ovan, innebär bland annat att ingen plats reserveras vid skrivmiss.

*Hämtning vid skrivmiss* kan användas om en plats har reserverats. Då kan data hämtas upp till denna plats från adressen i primärminnet. En del av dessa just hämtade data kommer förstås strax att skrivas över eftersom vi ju behandlar en skrivmiss.

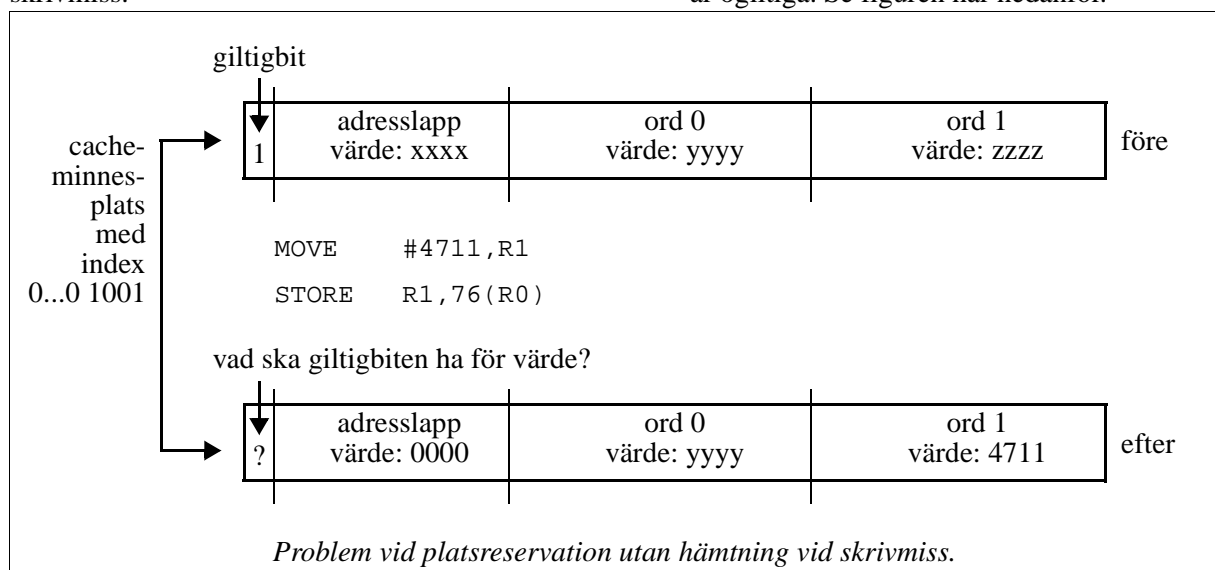
Poängen med att hämta upp data vid skrivmiss är att missen därigenom förvandlas till en träff. När hämtningen är klar kan skrivningen (STORE-instruktionen) startas om. Den ger då träff och behandlas på samma sätt som andra skrivträffar.

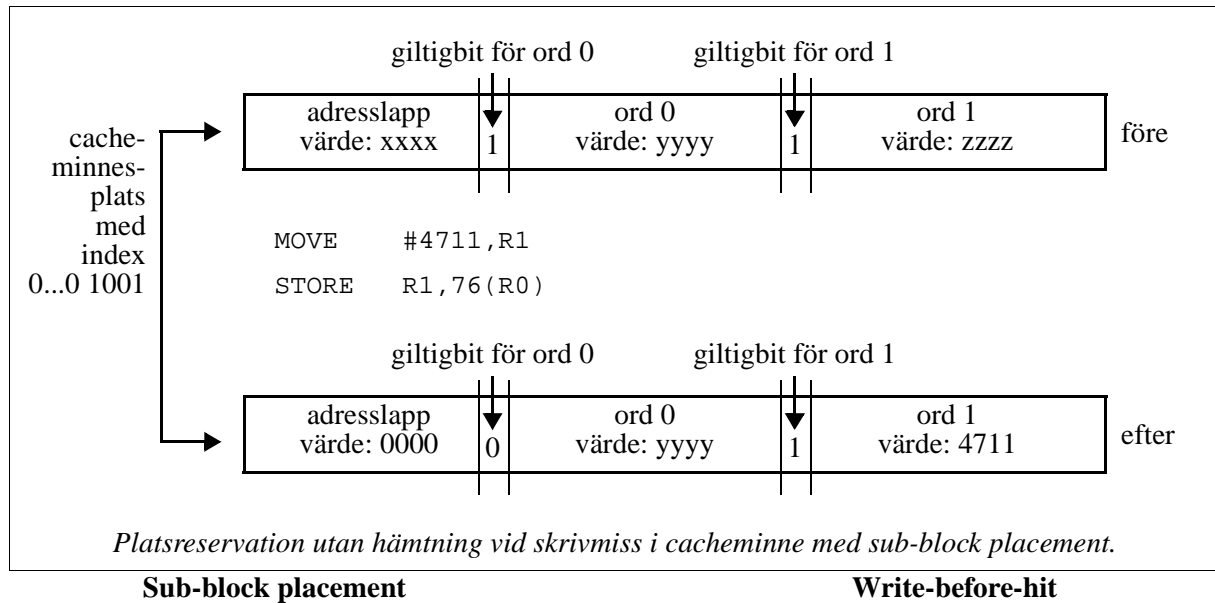
*Platsreservation utan hämtning vid skrivmiss* innebär att man reserverar en plats, men inte hämtar upp data från primärminnet. Den engelska beteckningen på detta fall är *allocate on write miss with no fetch-on-write*.

Platsen som reserverats innehåller då inga giltiga data. När STORE-instruktionen som orsakade skrivmissen sedan utförs skrivs giltiga data från processorn till cacheminnet, så platsen förblir inte tom särskilt länge.

Men en cacheminnesplats rymmer vanligtvis mer data än vad som skrivs med en enda STORE-instruktion. Därför blir inte hela datadelen av cacheminnesplatsen giltig utan bara en del.

Det gör att en giltigbit per plats inte räcker till. En ensam giltigbit kan ju inte ange vilka delar av platsen som är giltiga och vilka som är ogiltiga. Se figuren här nedanför.





Cacheminnet med platsreservation utan hämtning vid skrivmiss måste ha en indelning av cacheminnesplatserna i delblock (engelska: *sub-block placement*).

Delblocken är i allmänhet 32 eller 64 bit stora. Varje delblock har sin egen giltigbit.

Nu kan STORE-instruktionen som gav miss i cacheminnet utföras utan problem, se figuren här ovanför.

En komplikation är att skrivning av 1 byte är tillåten i alla moderna datorer. Om varje byte i cacheminnet har en egen giltigbit så går alldeles för mycket chippyta åt till giltigbitar.

Datorer med platsreservation utan hämtning vid skrivmiss för 32- och 64-bits skrivningar använder därför platsreservation *med* hämtning för byte-skrivningar.

Vid sub-block placement behöver alltså inte alla data på en cacheminnesplats vara giltiga. Detta kan utnyttjas för att snabba upp läsmisar: endast det eftersökta ordet hämtas, men inte de övriga i blocket.

Det medför förstås att lokaliteten i rummet inte utnyttjas, vilket är tveklaktigt. En kompromiss kan vara att hämta in några stycken delblock, men inte alla.

Cacheminnet kan normalt bara utföra en läsning eller skrivning i taget. För att avgöra om en referens ger miss eller träff måste datorns styrlogik läsa cacheminnets adresslapp. Först därefter kan skrivning ske.

I vissa fall kan man bygga styrlogiken så att skrivningen till cacheminnet görs direkt, utan föregående läsning av adresslappen. Vid en skrivträff spelar detta ingen roll, men vid skrivmiss så förstör detta förfarande det gamla innehållet i cacheminnesplatsen. Metoden kallas *write-before-hit*.

Write-before-hit kan bara användas om cacheminnet är direktmappat och av genomskrivningstyp (*write-through*). Om cacheminnet inte är direktmappat så finns det flera platser med samma index. Då måste adresslapparna läsas, så att skrivningen sker till rätt plats vid träff.

Om cacheminnet är av återskrivningstyp (det vill säga *inte* av genomskrivningstyp), så kan cacheminnet innehålla aktuella data som ännu inte har skrivits tillbaka till primärminnet. Dessa data får inte skrivas över innan återskrivning gjorts.

Därför måste styrlogiken först kontrollera om referensen ger träff eller miss; vid miss görs återskrivning av data från cacheminnet till primärminnet före skrivningen till cacheminnet.

**Skrivmissar: sammanfattning**

Vid skrivmiss finns följande alternativ:

	hämtning	ingen hämtning	
plats-reservation	fetch-on-write	write-validate	ej write-before-hit
			write-before-hit
ingen plats-reservation	meningslös kombination	write-around	ej write-before-hit
		write-invalidate	write-before-hit

*Alternativ vid skrivmiss.*

*Fetch-on-write* är det tankemässigt enklaste fallet. En plats reserveras och data hämtas från primärminnet precis som vid en läsmiss. Därefter har skrivmissen förvandlats till en träff och kan startas om.

*Write-validate* är en kortare (men inte vedertagen) beteckning på platsreservation utan hämtning vid skrivmiss. Detta fall kräver sub-block placement (utom i det sällsynta fallet att cacheminnet blockstorlek är 1 ord).

*Write-before-hit* är ett konstruktionssätt som innebär att cacheminnet skrivs utan föregående kontroll av adresslappen. Skrivningen görs alltså innan man vet om det var träff eller miss. *Write-before-hit* kan bara användas i direktmappade cacheminnen av genomskrivningstyp.

För såväl *fetch-on-write* som *write-validate* gäller att *write-before-hit* inte påverkar vilken information som finns i cacheminnet. Eftersom en plats ändå reserveras, så spelar det ingen roll om dess innehåll skrivs tidigt (*write-before-hit*) eller sent (ej *write-before-hit*). I cacheminnen utan platsreservation vid skrivmiss har det däremot betydelse om *write-before-hit* används eller ej.

*Write-invalidate* är kombinationen av *write-before-hit* och ingen platsreservation. Det innebär att data skrivs i cacheminnet även vid en miss (när ingen plats reserveras); data förstörs alltså data vid miss. Det verkar dumt, men kan möjliggöra kortare klockcykeltid.

*Write-around* innebär att träff/miss kontrolleras först (ingen *write-before-hit*). Vid miss lämnas cacheminnets innehåll opåverkat.

**Skrivmissar och skrivträffar**

Vad som händer vid skrivträff och vad som händer vid skrivmiss är i princip oberoende. Vi kan ställa upp denna tabell:

		skrivträff	
		genomskrivning	återskrivning
skrivmiss	fetch-on-write		
	write-validate		
	write-around		
	write-invalidate		

*Alternativ vid skrivning.*

I praktiken använder cacheminnen av återskrivningstyp ofta *fetch-on-write*. Detta innebär att skrivningar till primärminnet konsekvent skjuts upp tills de blir absolut nödvändiga. Dessutom finns ju möjligheten att samma ord ska läsas eller skrivas igen snart, och då är det bra om ordet redan finns i cacheminnet.

I cacheminnen av genomskrivningstyp är *write-around* vanligast.

*Write-validate* är krångligare än såväl *fetch-on-write* som *write-around*, men ger bättre prestanda. Den extra komplexiteten medför dock att *write-validate* är ovanlig i praktiken.

*Write-invalidate* används bara när andra metoder skulle ge förlängd klockcykeltid.

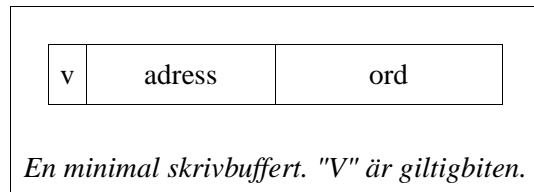
**Skrivbuffertar**

Ett cacheminne av genomskrivningstyp skickar varje skrivning vidare till närmast lägre nivå i minneshierarkin, exempelvis till huvudminnet. Om cacheminnet blockeras medan skrivningen skickas vidare, så tar varje *STORE*-instruktion mycket lång tid.

För att inte cacheminnet ska blockeras finns nästan alltid en *skrivbuffert* mellan cacheminnet och närmast lägre nivå. Skrivbufferten lagrar skrivningen medan den vidarebefordras. Cacheminnet kan då arbeta vidare med nya minnesreferenser.

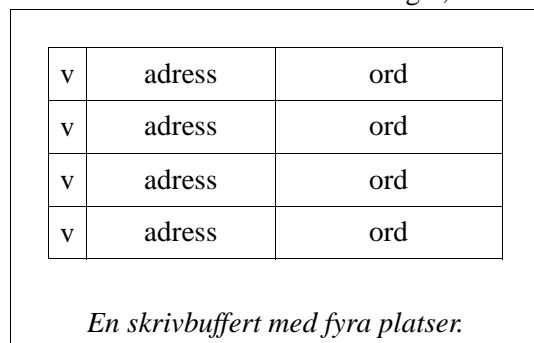
## Skrivbuffertvarianter

En skrivbuffert kan vara mer eller mindre sofistikerad. Enklast är en skrivbuffert som lagrar ett enda ord. Den består av en adresslapp, en giltigbit samt plats för det ord som ska skrivas.



Med en 1-ords skrivbuffert uppstår lätt väntetider, eftersom det ofta händer att flera STORE-instruktioner utförs med korta tidsmellanrum.

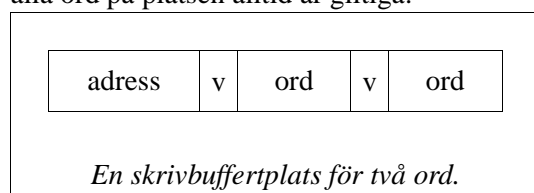
Skrivbufferten kan göras större på två sätt. Det ena är att ha flera platser som bildar en lista eller kö av väntande skrivningar, så här:



Det andra sättet att få en större skrivbuffert är att låta varje skrivbuffertplats lagra mer än ett ord. Ofta används båda sätten samtidigt.

Att lagra mer än ett ord per plats fungerar likadant som när blockstorleken ökas i ett cacheminne. Alla orden på en plats måste ligga intill varandra i adressordning, eftersom de delar på samma adresslapp.

Det kan noteras att varje ord på en skrivbuffertplats har en egen giltigbit, eftersom inte alla ord på platsen alltid är giltiga.



## Sammanmältning

En skrivbuffert med plats för flera ord kan vara *sammansmältande* (på engelska: coalescing). Vi ska studera tre fall: ingen sammansmältning, begränsad sammansmältning samt full sammansmältning.

En skrivbuffert *utan* sammansmältning bör se ut som den med fyra platser här till vänster. Varje skrivning vidarebefordras nedåt i minshierarkin. Skrivbufferten kan alltså inte minska busstrafiken utan bara sprida ut den i tiden.

När en ny skrivning anländer till en skrivbuffert utan sammansmältning ska en ny skrivbuffertplats alltid reserveras. Det ger en enkel konstruktion.

I ett flerprocessorsystem är det viktigt att minska busstrafiken så mycket som möjligt, eftersom det där finns flera processorer som var och en bidrar med busstrafik. Skrivbuffertar utan sammansmältning är därför lämpligast i enprocessorsystem.

Vi övergår nu till en skrivbuffert med *begränsad* sammansmältning. För att inte krångla till det för mycket på en gång låter vi även denna ha fyra platser med ett ord per plats.

När en ny skrivning anländer till en skrivbuffert med begränsad sammansmältning undersöks den senast reserverade skrivbuffertplatsens adresslapp. Stämmer adresslappen så skrivs det gamla ordet på platsen över med det nya.

Denna skrivbuffert slår alltså ihop flera skrivningar till samma adress i följd, vilket minskar busstrafiken.

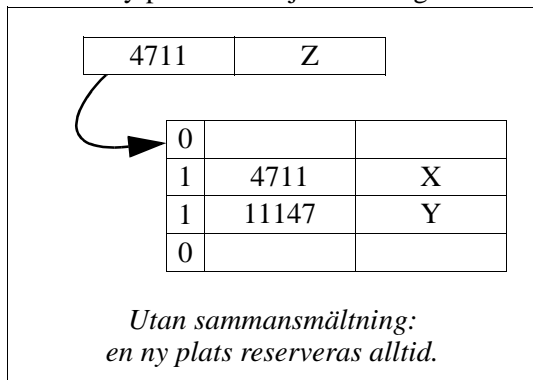
I en skrivbuffert med *full* sammansmältning undersöks *samtliga* adresslappar när en ny skrivning anländer. Stämmer adresslappen så skrivs det gamla ordet på platsen över med det nya. Denna skrivbuffert slår ihop flera skrivningar till samma adress, även om de inte kommer i följd.

Skrivbufferten med full sammansmältning har en intressant egenskap. Ett visst adresslappsvärde kan bara förekomma en gång i taget i denna skrivbuffert. I de andra typerna kan samma adresslappsvärde upprepas.

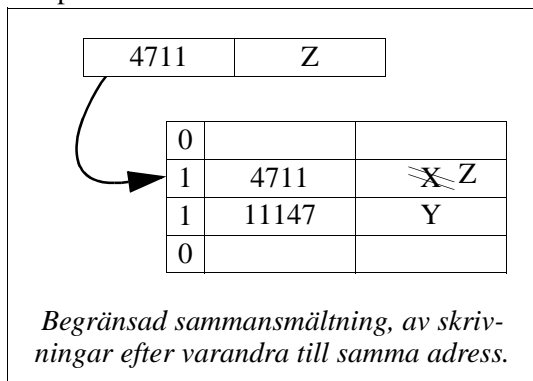
### Exempel på sammansmältning

Låt oss nu se på några konkreta exempel.

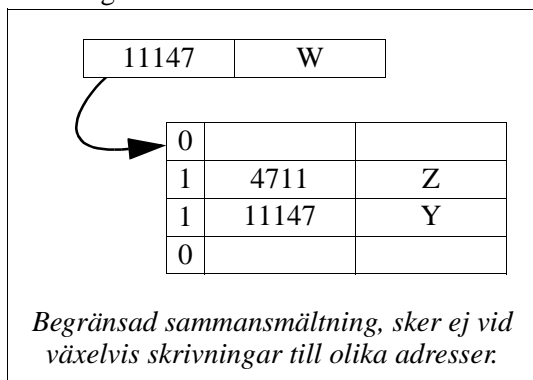
En skrivbuffert utan sammansmältning reserverar en ny plats vid varje skrivning.



Har skrivbufferten begränsad sammansmältning så reserveras ingen ny plats om skrivningens adress stämmer med adresslappen på den plats som senast reserverades.



Om skrivningens adress inte stämmer med den sist reserverade platsens adresslapp så reserveras en ny plats, även om adressen finns någon annanstans i skrivbufferten.



Med full sammansmältning så sammansmälts skrivningar alltid, så länge adressen finns någonstans i skrivbufferten.

### Flera ord per skrivbuffertplats

En skrivbuffert som lagrar mer än ett ord på varje plats måste ha någon form av sammansmältning; annars blir den meningslös. Vi ska strax se varför.

Låt oss betrakta fallet med flera ord per plats, men *ingen* sammansmältning.

Vi finner då att när en skrivning anländer ska en ny plats omedelbart reserveras, utan att någon adresslapp undersöks. Även om skrivningen avser en adress som ligger intill den senast skrivna adressen, så reserveras alltså en ny skrivbuffertplats.

Då finns det ingen möjlighet att fylla ut en plats så att alla dess ord är giltiga. I stället kommer högst ett ord per plats att ha giltigbitten ettställt.

Har skrivbufferten i stället *begränsad* sammansmältning så undersöks den senast reserverade skrivbuffertplatsens adresslapp.

Om adresslappen stämmer, så skrivs det nya ordet in på rätt position och positionens giltigbitten ettställs.

I det fall processorn skriver flera ord efter varandra — med intilliggande adresser — så kommer en och samma plats att fyllas upp med de skrivna orden. Det är precis det fall som ska klaras av med en skrivbuffert med begränsad sammansmältning.

I en skrivbuffert med full sammansmältning undersöks samtliga adresslappar vid varje skrivning.

Det ger mer komplicerad maskinvara, men fångar upp flera fall. Ett tänkbart exempel är att processorn skriver flera ord med intilliggande adresser, men gör andra skrivningar då och då.

På den här beskrivningen kan det låta som om full sammansmältning vore det enda rätta. Men ett motiv för att begränsa möjligheterna till sammansmältning är att skrivningar inte kan ångras.

Om en skrivning ger felavbrott så blir det svårt att ta om hand felet om processorn redan har gjort klart skrivningar som hör till instruktioner senare i programmet.

### Skrivning i datorer med separata instruktions- och datacacheminnet

Det kan vara värt att notera att data är både läs- och skrivbara, medan instruktioner bara läses av processorn. Eftersom ett cacheminne för instruktioner inte behöver klara av skrivningar kan det byggas något enklare än datacacheminnet.

Ett instruktionscacheminne som inte klarar skrivning kräver dock "ren" kod, det vill säga att programmet inte modifierar sig självt. Allmänt gäller idag att "normala" program endast innehåller ren kod.

Men vissa funktioner kräver att instruktionscacheminnets innehåll kan ändras på kommando. Det gäller framför allt programladdningen när ett nytt program ska startas.

Vid laddningen kopieras programmet från skivminne till ett område i primärminnet, antingen med hjälp av DMA eller med en programslinga. När primärminnet är uppdaterat måste instruktionscacheminnets innehåll ogiltigförklaras (invalideras), så att primärminnets nya innehåll verkligen hämtas.

Görs kopieringen med en programslinga, så innebär detta att information hämtas från skivminnets styrenhet med en LOAD-instruktion och sparas i primärminnet med en STORE-instruktion.

Härvid passerar informationen datacacheminnet — processorn "vet" ju inte att det är programkod den kopierar. Processorn använder alltid datacacheminnet för alla minnesreferenser utom hämtfasens instruktionshämtning.

Är datacacheminnet av återskrivningstyp så kan informationen fastna där, utan att nå primärminnet. En invalidering av instruktionscacheminnet medför då bara att samma gamla föråldrade information hämtas igen från primärminnet.

För att undvika detta måste datacacheminnet tömmas, så att alla smutsiga ord (med ett-ställd dirty-bit) skrivs till primärminnet.

I allmänhet finns särskilda instruktioner för att tömma och/eller invalidera instruktions- och datacacheminnet. Dessa instruktioner används i de subrutiner för programladdning som brukar finnas i datorns operativsystem.

### Flera enheter på bussen

Hittills har vi antagit att processorn är ensam om att kunna modifiera primärminnet. Om det finns andra enheter som ändrar i minnet så finns risk att cacheminnet levererar föråldrad information till processorn.

I flerprocessorsystem är det uppenbart att alla processorerna kan ändra i primärminnet. Men även enprocessorsystem har ofta en enhet för direkt minnesåtkomst (DMA), som kan läsa och skriva i minnet på egen hand.

Problemet är något mindre om cacheminnet är av genomskrivningstyp, eftersom primärminnet då alltid innehåller den senaste informationen. Trots detta kan följande oönskade händelseförlopp inträffa:

1. Processorn läser (eller skriver) en minnescell, vilket medför att dess innehåll läggs i cacheminnet.
2. En DMA-överföring uppdaterar primärminnet, men inte cacheminnet.
3. Processorn försöker läsa minnescellens nya innehåll, men cacheminnet levererar det gamla föråldrade innehållet.

Med cacheminnet av återskrivningstyp finns dessutom risken att processorn skriver aktuella data till cacheminnet, varefter DMA-enheten läser gamla data från primärminnet.

Problemet kallas *the stale-data problem* (problemet med gamla data). Det har många lösningar, alla med sitt pris i form av ökad maskinvarukostnad, minskade prestanda och/eller problem för programmeraren.

En möjlighet är att cacheminnet undersöker alla adresser som passerar på databussen. Om en adress finns i cacheminnet så uppdateras (eller ogiltigförklaras) cacheminnesplatsen. Det gör att cacheminnet alltid innehåller aktuell information (eller ingen alls).

Denna metod, där cacheminnet spionerar på vad som händer på bussen, brukar på datorsvengelska kallas *snooping cache*.

### Primärminnet sett som cacheminne

Hittills har vi betraktat cacheminnen som sitter på ett processorchipp. Den bärande idén är att minnen är antingen stora eller snabba, men inte båda för det har man inte råd med.

Ett litet men snabbt cacheminne används därför som komplement till ett stort långsamt minne, för att lagra information som används ofta (eller förväntas användas ofta).

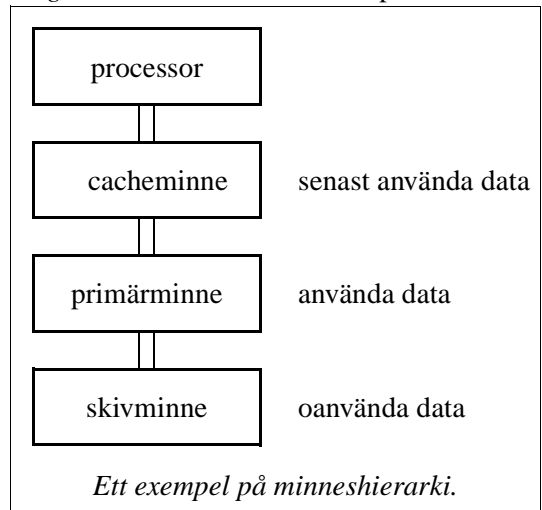
Man kan utvidga detta resonemang till andra sammanhang. Exempelvis är primärminne snabbt och dyrt om man jämför med skivminne (disk).

Primärminnet borde därför endast innehålla de data och instruktioner som används något sånär ofta (eller förväntas användas ofta). Övrig information får ligga på skivminnet. På detta sätt får vi en hierarki med tre nivåer förutom processorn, se figuren här nedanför.

Flyttning av data och instruktioner mellan skivminne och primärminne kan ske på olika sätt. Med en kombination av maskinvara och programvara (i operativsystemet) kan denna del av minneshierarkin gömmas för programmerare som skriver "vanliga" program.

Programmeraren kan då arbeta som om datorn var utrustad med ett mycket stort primärminne. Processorn och operativsystemet kopierar information mellan primärminne och skivminne vartefter programmet behöver dem. Resultatet kallas *virtuellt minne*.

För att virtuellt minne ska fungera behövs maskinvara och program för *adressöversättning*, som är ämnet för nästa kapitel.



### Andranivåcacheminne

Att bygga en hierarki av cacheminnen låter sig göras även mellan processor och primärminne. Det blir faktiskt allt vanligare.

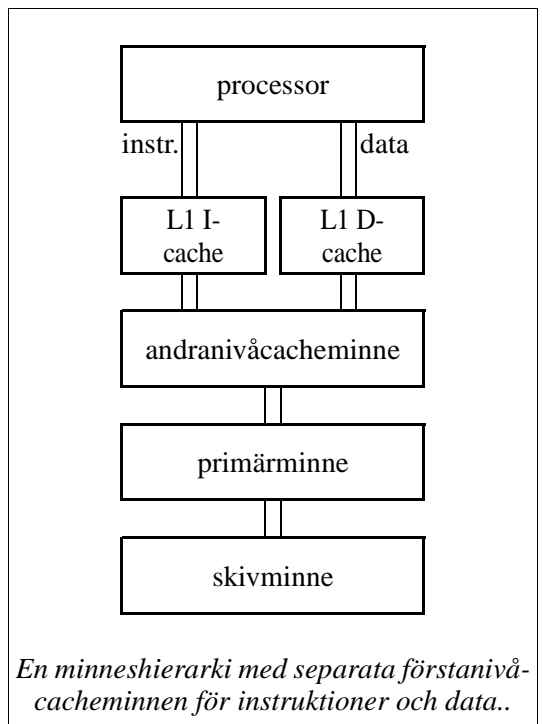
Orsaken är att klockfrekvensen hos nya processorer ökar mycket snabbare än åtkomsttiden för minneskretsar. Det gör att skillnaden i hastighet mellan processor och primärminne i nya datorer blir större för varje år.

För att inte en miss i cacheminnet närmast processorn ska ta alltför lång tid lägger datorkonstruktören till ett extra cacheminne på vägen till primärminnet. Detta extra cacheminne kallas andranivåcacheminne. Cacheminnet närmast processorn kallas då förstanivåcacheminne.

Andranivåcacheminnet görs alltid betydligt större än förstanivåcacheminnet. Andranivåcacheminnet kan också vara mer associativt.

Det är vanligt med separata förstanivåcacheminnen för instruktioner och data. Med ett gemensamt cacheminne närmast processorn blockeras instruktionshämtningen vid varje LOAD- eller STORE-instruktion (se sida 7).

En mera realistisk minneshierarki kan därför se ut som i figuren här nedanför. "L1" är förkortning för "level 1", det vill säga första nivån. "I-cache" är instruktionscacheminne och "D-cache" är datacacheminne.



# Adress- översättning

## Adressöversättning i korthet

Primärminnet i en dator är en resurs, vars användning ska fördelas mellan flera olika program. Fördelningen görs av ett övervakningsprogram, som kallas operativsystem.

De flesta persondatorer och arbetsstationer har maskinvara för adressöversättning. Denna maskinvara räknar om de adresser som används i ett program till andra adresser, som används av primärminnet.

Adresserna i programmet kallas *virtuella*, medan de adresser som primärminnet använder kallas *fysiska*.

När adressöversättning används förenklas programmerarens och operativsystemets arbete på följande sätt.

Alla program skrivs som om de hade tillgång till adresser från 0 och uppåt så långt som behövs, vilket underlättar programmeringen.

Maskinvaran för adressöversättning, som styrs av operativsystemet, räknar sedan om dessa adresser. Varje program kan placeras på den plats i primärminnet som operativsystemet finner mest lämplig.

Det är enkelt att bygga in en mekanism för minnesskydd i adressöversättningslogiken. Minnesskydd innebär att ett program inte kan läsa eller skriva i minnesceller som hör till andra program. Programfel i ett program kan då inte medföra felaktiga variabelvärden i andra program.

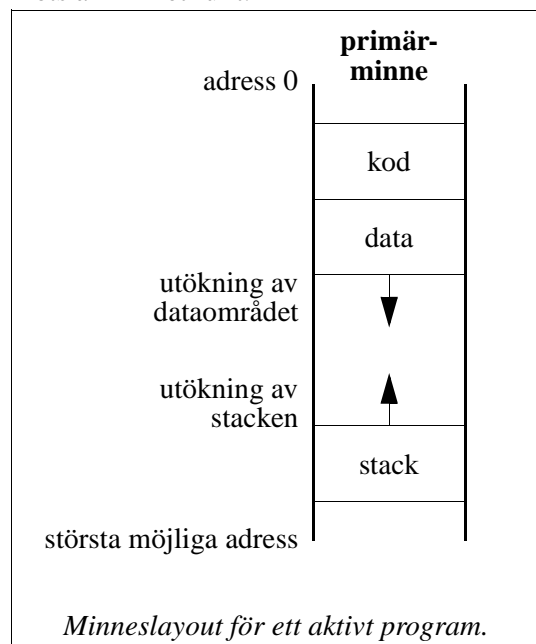
Både den friare placeringen av olika program och minnesskyddet dem emellan underlättar operativsystemets arbete.

## Minnesanvändning

Det minnesutrymme som ett aktivt program använder kan delas in i tre delar: programkod, data och stack. Figuren nedan visar hur dessa delar kan placeras.

Programkoden börjar på en adress nära noll och fortsätter så långt som behövs. Efter programkoden kommer data och längs bort stacken.

Eftersom både datamängd och stack kan behöva utökas under programmets körning, finns plats reserverad för detta i mitten. Då kan data växa mot högre adresser (nedåt i figuren) och stacken växa mot lägre adresser (uppåt i figuren). När dataarea och stackarea möts är minnet fullt.



### Motiv för adressöversättning

Adressöversättning är onödig om datorn *dels* bara kör ett program i taget, och *dels* har tillräckligt stort primärminne för detta programs behov.

När endast ett program körs kan detta få tillgång till alla giltiga minnesadresser. Så länge programmet får plats i primärminnet behöver minnet inte administreras speciellt.

Det normala numera är dock att datorn har flera program aktiva samtidigt, och att alla dessa program inte får plats samtidigt i primärminnet.

Till exempel har den Unix-arbetsstation där detta skrivs just nu nära 60 aktiva program, varav hälften ingår i operativsystemet. Programmens sammanlagda minnesbehov är omkring 100 Mbyte vilket är tre gånger primärminnets storlek (32 Mbyte).

Olika kombinationer av minnesstorlek och antal program visas i nedanstående tabell.

		Minnesstorlek	
		tillräcklig	för liten
Antal program	ett	fall 1	fall 2
	flera	fall 3	fall 4

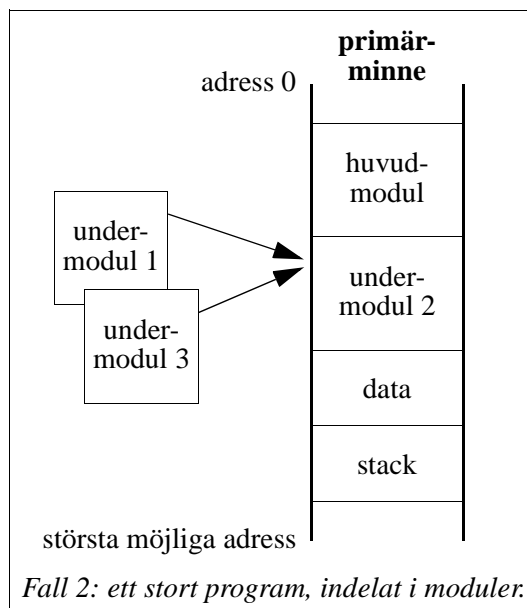
*Olika krav på minnesresurser.*

*Fall 1 är enklast och fall 4 är svårast.*

Om primärminnet är stort nog för att rymma alla aktiva program kan man till nöds klara sig utan adressöversättning, även med mer än ett aktivt program i datorn.

Utan adressöversättning måste programmen vara positionsoberoende (relokerbara). Detta kräver extra tankearbete vid kompilering, assemblering och länkning, men är fullt genomförbart.

En viktig detalj är minnesskyddet. Minnesskydd kräver extra maskinvara, som inte behöver kombineras med adressöversättning. I praktiken klarar dock nästan alla datorers maskinvara antingen *både* minnesskydd och adressöversättning eller ingetdera.



### Livet utan adressöversättning: ett ensamt program

*Fall 1* i tabellen här intill innebär att datorn kör ett ensamt program, som får plats i minnet. Då kan minnet organiseras helt enligt figuren på sida 25.

I *fall 2* körs ett ensamt program som *inte* får plats. Då kan programmeraren försöka dela in programmet i en huvudmodul och några undermoduler, på ett sådant sätt att inte alla undermodulerna används samtidigt.

Huvudmodulen kompletteras sedan med programkod som hämtar in olika undermoduler efter behov.

Figuren ovan visar ett modulindelad program med en huvudmodul och tre undermoduler. Av undermodulerna kan bara en i taget finnas i primärminnet. Detta kallas på datorsvengelska för att programmet har *overlay-struktur*.

Omskrivningen med modulindelning innebär ett betydande extraarbete för programmeraren. Det kan också hända att programmeraren inte hittar den bästa möjliga indelningen.

I värsta fall går det inte att dela in programmet i lämpliga moduler. Då kan programmet inte köras alls på den aktuella datorn.

### **Livet utan adressöversättning: flera program**

*Fall 3* innebär att datorn kör flera program, som tillsammans ryms i minnet.

Var och ett av programmen kan då tilldelas ett visst minnesutrymme, och inom detta används en layout enligt figuren på sida 25.

Detta system har flera nackdelar.

Den första nackdelen är att stack- och data-areorna måste placeras ut redan när ett program hämtas till minnet. Om dessa läggs tätt, så får data och stack inte särskilt mycket tillväxtutrymme.

Om data och stack läggs glest, så åtgår mera minne totalt — minne som kanske inte utnyttjas. Det är svårt att från början veta hur mycket minne ett program behöver, eftersom behovet brukar variera starkt med indata.

Den andra nackdelen är att programmen måste vara relokert skrivna.

Olika kombinationer av program ska ju kunna finnas i minnet samtidigt. Då måste varje program kunna placeras på vilken plats som helst i minnet. Annars blir möjligheterna att kombinera olika program starkt beskurna.

Den tredje nackdelen är att minnesskydd saknas. Utan minnesskydd kan ett fel i ett program lätt leda till ändringar av innehållet i minnesceller som tillhör andra program.

Adressöversättning med kontroll av minnesskyddet eliminerar alla tre nackdelarna.

Man kan fråga sig, om adressöversättning inte har några nackdelar. Svaret är att den huvudsakliga nackdelen är att det behövs extra maskin- och programvara för att åstadkomma översättningen.

Den extra maskinvaran kostar pengar, och den extra programvaran "kostar" i form av extra körtid.

### **Swapping**

Som avslutning på vår studie av livet utan adressöversättning betraktar vi nu *fall 4*: flera aktiva program i ett primärminne som inte har plats för dem alla samtidigt.

När primärminnet är för litet måste en del program placeras någon annanstans, vanligtvis på ett skivminne. Ett program som placeras på skivminne i stället för i det (fulla) primärminnet kallas på datorsvengelska för *utswappat* (uttalas "utsvåppat").

För ett utswappat program finns varken programkod, data eller stack i primärminnet. Operativsystemet har dock kvar viss information om programmet i sina tabeller.

När ett utswappat program ska fortsätta köras, måste det först kopieras in till primärminnet. Processorn är ju konstruerad för att hämta instruktioner och data från primärminnet, inte direkt från skivminnet.

Att primärminnet är fullt medför att något program måste kopieras ut till skivminnet innan ett utswappat program kan kopieras in till primärminnet. De båda programmen byter alltså plats.

Swapping har en betydande nackdel: den kopiering som krävs tar lång tid. Väntetiden när ett utswappat program ska hämtas in är några sekunder.

Denna svarstid upplevs som mycket lång, särskilt för datorer med grafiska fönstersystem. Datorer som styrs med skrivna kommandoord kan ofta ha längre reaktionstid innan de upplevs som långsamma.

I det fall att ett ensamt program är för stort för primärminnet så är inte swapping tillräckligt. Programmeraren kan då införa en moduluppdelning (overlay-struktur) på det sätt som beskrivs på sida 26.

Swapping och moduluppdelning kan alltså användas var för sig och tillsammans. Swapping löser problemet med att få minnet att räckta till flera program samtidigt; moduluppdelning löser problemet med att få minnet att räckta till ett enda program som är för stort.

## Sidindelning

Vi koncentrerar oss nu på hur adressöversättning fungerar och hur översättningen kombineras med minnesskydd.

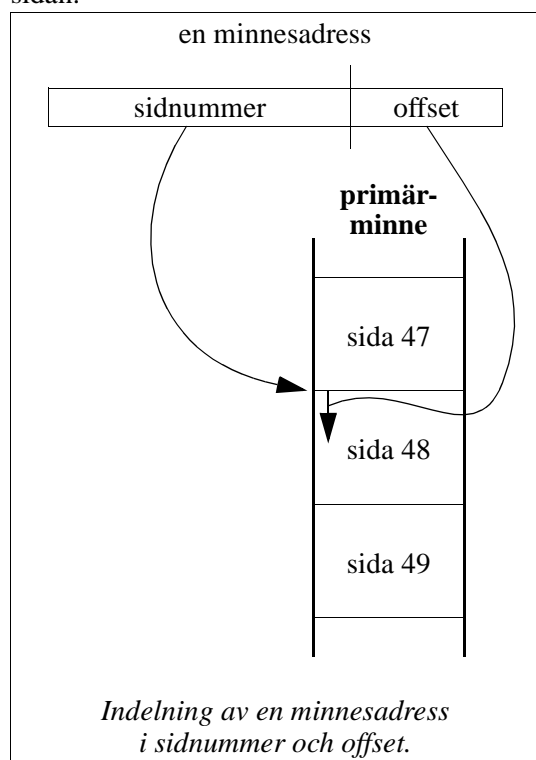
För att göra adressöversättningen enklare och effektivare delar vi in minnet i *sidor* (pages) med en viss konstant storlek.

Denna sidstorlek väljs när dator och operativsystem konstrueras och är sedan svår att ändra. En ändring påverkar nämligen både maskinvaran och operativsystemet, och normalt vill man kunna byta ut dessa oberoende av varandra (mot nyare versioner).

Sidstorleken (räknad i antal minnesord) är alltid en jämn tvåpotens, och första sidan börjar alltid på adress 0. Därför kan en minnesadress delas in i en del för sidval och en del som väljer plats inom sidan.

Den del som väljer sida kallas *sidnummer* (page number); den del som anger en plats inom en sida kallas *offset*.

I figuren här nedanför ser vi hur sidnumret pekar ut en viss sida i primärminnet. Offseten anger sedan en plats inom den utpekade sidan.



## Översättningstabell

Den adress som används av det aktiva programmet kallas virtuel. Därför används beteckningen *virtuellt sidnummer* (Virtual Page Number, VPN) för sidnumret i den adress programmet producerar.

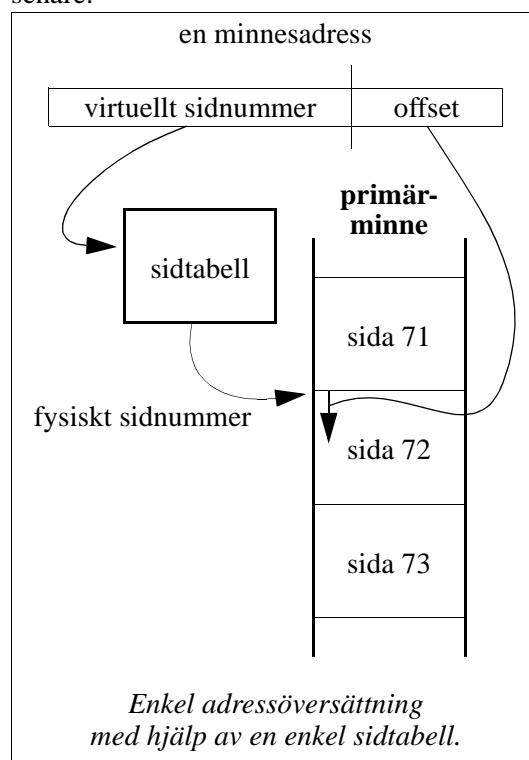
Den adress som entydigt bestämmer en plats i det verkliga primärminnet kallas ju fysisk adress. Sidnummerdelen av en sådan kallas följaktligen *fysiskt sidnummer* (Physical Page Number, PPN).

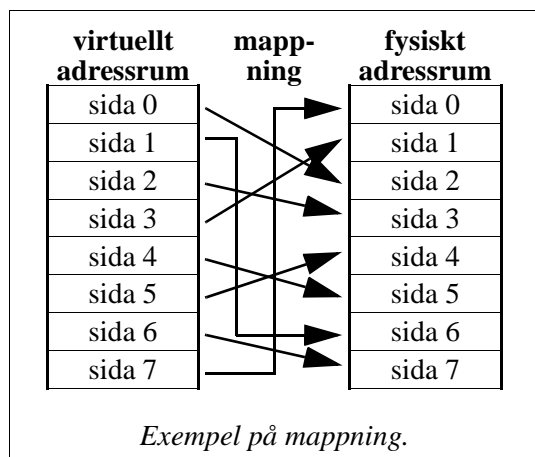
Det virtuella sidnumret används som index i en *sidtabell*. Sidtabellen innehåller det fysiska sidnumret.

Vid översättningen ändras endast sidnumret: det virtuella sidnumret byts mot ett fysiskt sidnummer. Offseten är oförändrad vid översättningen. Det hela kan illustreras med figuren nedan.

I verkligheten är sidtabellen mera komplicerad än figuren antyder. Dess funktion är dock alltid densamma: virtuellt sidnummer översätts till fysiskt.

En mer verklighetstrogen sidtabell beskrivs senare.





### Mapping

Ordet mappning är svengelska. Det används ofta om översättningen sedd som funktion i matematisk mening.

Här ovan visas ett exempel på mappning från virtuellt till fysiskt adressrum. Virtuellt sidnummer 0 översätts i exemplet till fysiskt sidnummer 2, virtuell sida 1 översätts till fysisk sida 6, och så vidare.

Den mappning som visas i figuren stämmer inte med typiska mappningar i verkliga datorer. Vi ska ta upp två viktiga skillnader.

*Det fysiska adressrummet brukar vara mindre än det virtuella.* I figuren är adressrummen lika stora.

I verkligheten anges en virtuell adress med 32 eller 64 bitar, vilket med byte-adresserat minne ger adressrum på 4 Gbyte respektive 18 Pbyte ( $18 \cdot 10^{18}$  byte). Mycket få datorer idag har primärminnen större än 1 Gbyte.

*Mappningsfunktionen är oftast partiell, och endast sällan total.* I figuren finns en fysisk sida för varje virtuell; den visade mappningen är alltså en total funktion.

I verkligheten fyller de flesta program inte ut det virtuella adressrummet. De virtuella sidor som programmet inte använder markeras som ogiltiga. För detta ändamål finns en särskild giltigbit på varje plats i sidtabellen.

Om processorn begär läsning eller skrivning på en ogiltig virtuell sida så görs ett felavbrott (exception). I Unix kallas detta "segmentation fault" och drabbar gärna den som skriver sina första C-program.

### Mappingar kan man ha flera av

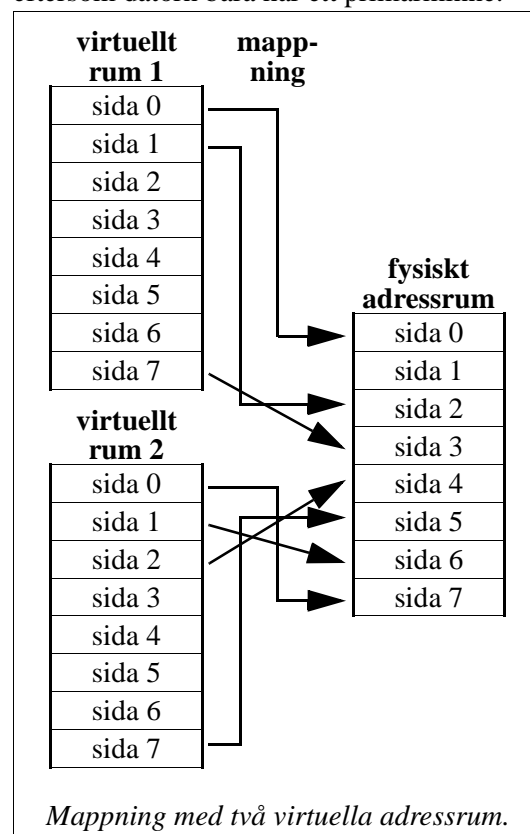
En viktig användning för adressöversättning är att *ge varje program ett eget virtuellt adressrum*. Detta innebär att varje program har sin egen mappning, och att sidtabellens innehåll måste bytas när datorn växlar mellan olika aktiva program.

Vart och ett av programmen kan då vara skrivet för att börja på adress 0 och uppåt, precis enligt figuren på sida 25. Detta underlättar programmeringen.

I det fysiska primärminnet kan programmets sidor placeras på fullständigt godtyckliga platser, och i godtycklig ordning. Sidtabellens innehåll justeras sedan så att sidorna hamnar rätt i det virtuella adressrummet.

När ett program ska stoppas, och ett annat fortsätta sin körning, så ändrar operativsystemet i sidtabellen så att det nya programmens virtuella adressrum mappas på rätt sätt.

Figuren nedan visar två virtuella adressrum med olika mappningar. Några virtuella sidor är ogiltiga och saknar översättning (sida 2—6 i det ena adressrummet och sida 3—6 i det andra). Det finns bara ett fysiskt adressrum eftersom datorn bara har ett primärminne.



### Virtuellt minne

Låt oss återgå till våra funderingar från sida 26–27. Det finns ju två fall när det fysiska minnet är för litet för att rymma alla program: fall 2 och fall 4.

Maskinvaran för adressöversättning kan hjälpa oss att hantera dessa fall. Det går till på följande sätt.

Vi inför en fil som lagrar sidor som inte får plats i primärminnet. Av historiska skäl brukar filen kallas swapfil.

När antalet lediga sidor i primärminnet sjunker under en viss nivå, söker operativsystemet upp sidor som inte använts nyligen. Innehållet i dessa sidor kopieras till swapfilen, vilket frigör plats i primärminnet.

I samband med kopieringen ändrar operativsystemet också sidtabellen för den process som äger de sidor som flyttats till swapfilen. De rader i processens sidtabell som avser bortflyttade sidor markerar som ogiltiga.

Om processen försöker använda någon sida som kopierats ut till swapfilen så medför det felavbrott (exception). Operativsystemet tar emot felavbrottet och kan kopiera in sidan från swapfilen till primärminnet.

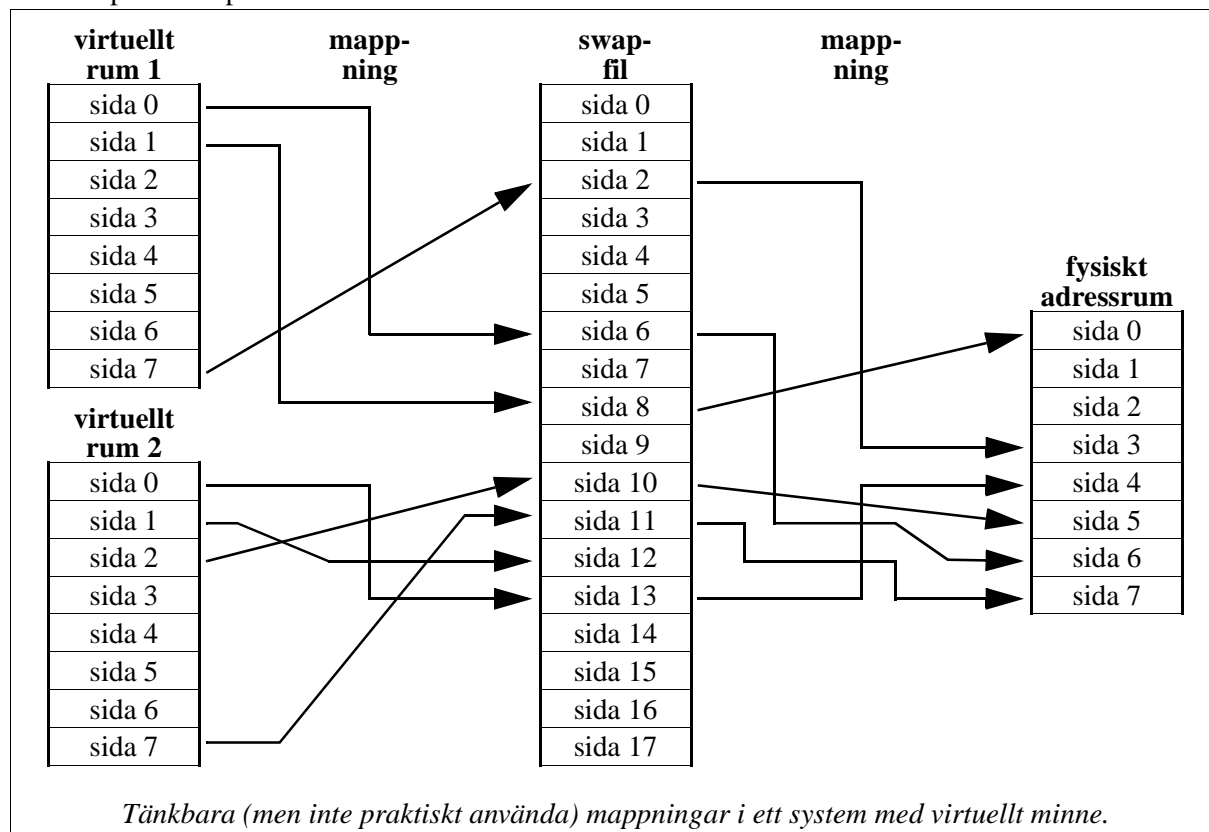
När sidan återställts till primärminnet ändrar operativsystemet i sidtabellen så att sidan åter markeras som giltig. Därefter kan processen fortsätta.

Denna hantering underlättas mycket av adressöversättningen. Översättningen gör det ju möjligt att placera en sida på godtycklig plats i primärminnet. Operativsystemet behöver till exempel inte kopiera in en sida från swapfilen till samma plats som sidan hade innan den flyttades ut, till swapfilen.

I vissa operativsystem finns plats reserverad i swapfilen även för sidor som har plats i primärminnet. I sådana system kan man tänka sig nedanstående dubbla mappning: från virtuellt adressrum till swapfil och sedan från swapfil till primärminne.

Observera att mappningen bara är tänkt. Inget känt operativsystem använder datastrukturer som liknar denna mappning.

Istället är mappningarna uppdelade på flera olika datastrukturer. Sidtabellen översätter från virtuellt till fysiskt adressrum. En annan (mer komplicerad) datastruktur översätter från virtuellt adress till plats i swapfilen, ifall sidan inte finns med i sidtabellen.



### Motiv för sidtabell med flera nivåer

Låt oss införa lite siffror. När detta skrivs är en vanlig sidstorlek i existerande datorer 4096 byte ( $2^{12}$  byte). Vi förutsätter 32-bits virtuella adresser.

Det virtuella adressrummet (för ett enda program) består då av  $2^{20}$  sidor. En sidtabell med så många platser skulle bli 4 MB stor (4 194 304 byte).

Operativsystemet utnyttjar 10–20 hjälpprogram, som ständigt är igång även när inga användarprogram körs. Bara sidtabellerna för dessa hjälpprogram skulle fylla större delen av primärminnet i en normal dator.

I datorer med 64-bits virtuella adresser blir sidtabellen för en process mer än 1000 GB stor (alltså  $10^{12}$  byte). Att lagra *en enda* sådan sidtabell skulle kräva 30 stora hårddiskar.

Dessutom använder nästan inga program hela sitt virtuella adressrum. Den fullständiga sidtabellen för en process innehåller därför många ogiltigmarkerade platser.

Sidtabellen delas därför in i flera delar, som kallas nivåer.

Nivåindelningen gör att alla ogiltiga sidor inte behöver ha var sin plats i sidtabellen. I stället kan hela block med ogiltiga sidor dela på en plats.

### Exempel på sidtabell med flera nivåer

Här nedanför visas ett exempel på en sidtabell med tre nivåer. Sidnumret delas in i tre delar, som här har kallats index 1, 2 och 3.

Förstanivåsidtabellen innehåller inte sidnummer, utan istället pekare till andranivåsidtabeller.

Index 1 anger en plats i förstanivåsidtabellen; på denna plats finns en pekare till den andranivåsidtabell som visas i figuren.

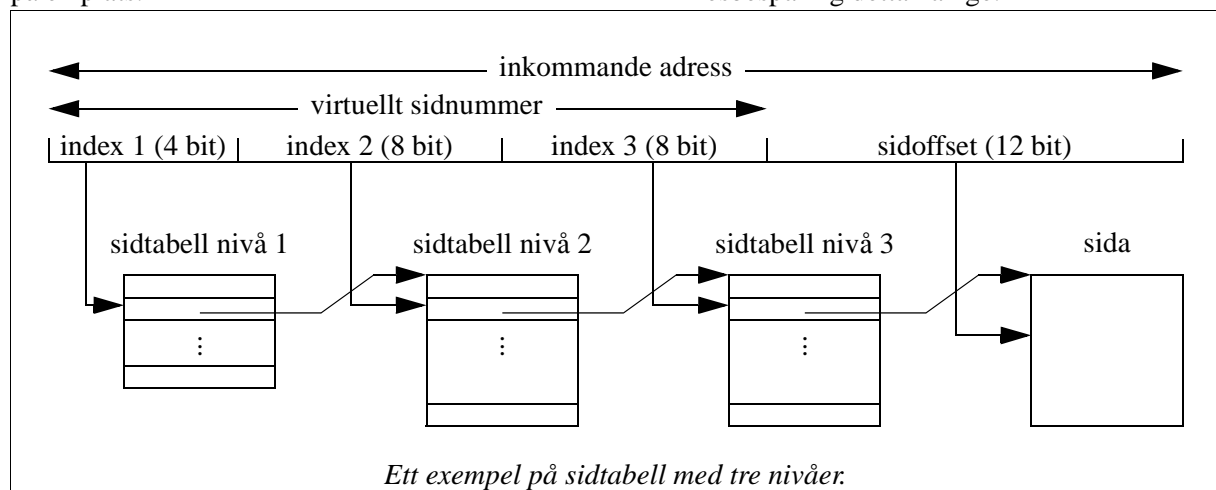
Index 2 anger en plats i den utpekade andranivåsidtabellen. På denna plats finns en pekare till en tredjenivåsidtabell.

I tredjenivåsidtabellen finns fysiska sidnummer. Index 3 anger en plats i den utpekade tredjenivåsidtabellen; på den platsen finns det fysiska sidnummer som ska användas för referensen.

Poängen med detta är att många av sidtabellerna i nivå 2 och 3 kan utelämnas, eftersom programmet inte använder hela sitt virtuella adressrum.

En sidtabell som utelämnats ersätts med en ogiltigmarkering på nivån ovanför. Om en sidtabell på nivå 3 utelämnats, så finns alltså en ogiltigmarkering i stället för en pekare i sidtabellen på nivå 2.

Vi ska ta ett exempel för att se hur stor minnesbesparing detta kan ge.



### Minnesbesparing i trenivåers sidtabell

En sidtabell på nivå 3 har 256 platser i vårt exempel. Eftersom en sida är 4096 byte stor, mappar sidtabellen på nivå 3 ett minnesområde som är 1 MB stort (1 048 576 byte).

Om programmet använder 10 MB minne av de 4000 MB som utgör hela dess adressrum, så räcker det med 10 sidtabeller på nivå 3. Dessutom behövs några sidtabeller på nivå 2, och en på nivå 1.

Totalt blir detta högst 15 sidtabeller, var och en med 256 platser. Om varje plats är 4 byte tar varje sidtabell 1 KB (1024 byte) i anspråk, och den totala minnesåtgången blir ungefär 15 KB.

Detta kan jämföras med de 4 MB som skulle behövas för samma sidtabell, om vi använde bara en nivå.

Många tredjenivåsidtabeller (och ganska många andranivåsidtabeller) behövs alltså inte, utan ersätts med en ogiltigmarkering på nivån ovanför.

Detta gör att en flernivåers sidtabell nästan alltid tar mycket mindre minnesutrymme än en sidtabell med bara en nivå.

I det sällsynta fallet att en process verkligen använder hela sitt virtuella adressrum tar sidtabellen med flera nivåer lite extra plats. Men eftersom nästan inga processer i en dator använder hela sitt virtuella adressrum, så blir vinsten mycket större än denna lilla förlust.

### Exempel på referens i trenivåers sidtabell

Låt oss ta ett exempel till, med bestämda adressangivelser. Se figuren här nedanför!

Vi kör ett program vars programkod börjar på adress 0 och sträcker sig till adress 22 FFFF (hexadecimalt). Programkoden upptar alltså drygt 2 MB.

Data börjar på adress 1000 0000 (hexadecimalt). Vi antar att programmet använder 200 KB data till att börja med.

Stacken börjar på adress 3FFF FFFC (hexadecimalt). Den växer nedåt, mot lägre adresser. Vi antar att stacken till att börja med får plats på en enda sida.

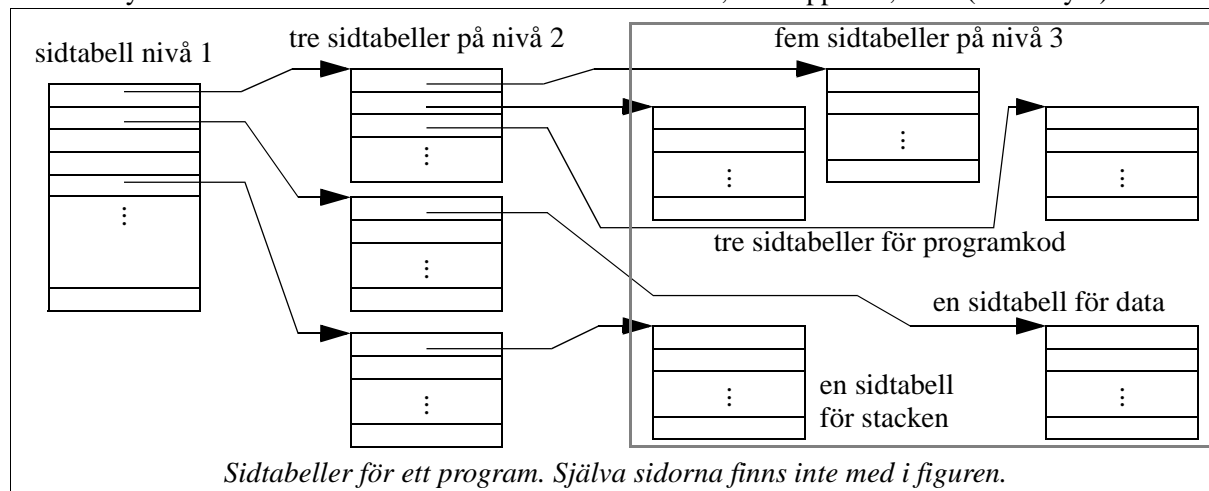
I förstanivåsidtabellen kommer endast platserna 0, 1 och 3 att användas. Platserna 2–6 och 8–15 är markerade som ogiltiga.

I den andranivåsidtabell som pekats ut av plats 0 i förstanivåsidtabellen används de tre första platserna. På dessa tre platser finns pekare till tredjenivåsidtabeller. De övriga 253 platserna i andranivåsidtabellen är markerade som ogiltiga.

Plats 1 i förstanivåsidtabellen pekar ut en andranivåsidtabell, där endast plats 0 innehåller en pekare till en tredjenivåsidtabell. De övriga platserna är ogiltigmarkerade.

Plats 7 i förstanivåsidtabellen pekar också ut en andranivåsidtabell. I den andranivåsidtabellen används plats 255; de övriga platserna är ogiltigmarkerade.

Vårt program använder alltså förstanivåsidtabellen, tre andranivåsidtabeller, och fem tredjenivåsidtabeller. Totalt behövs nio sidtabeller, som upptar 8,4 KB (8 448 byte) minne.



### Cacheminne för adressöversättning

Även när flernivåers sidtabell används, så består ett virtuellt adressrum av så många sidor att tabellen inte får plats inne i processorn. Sidtabellen läggs därför i primärminnet.

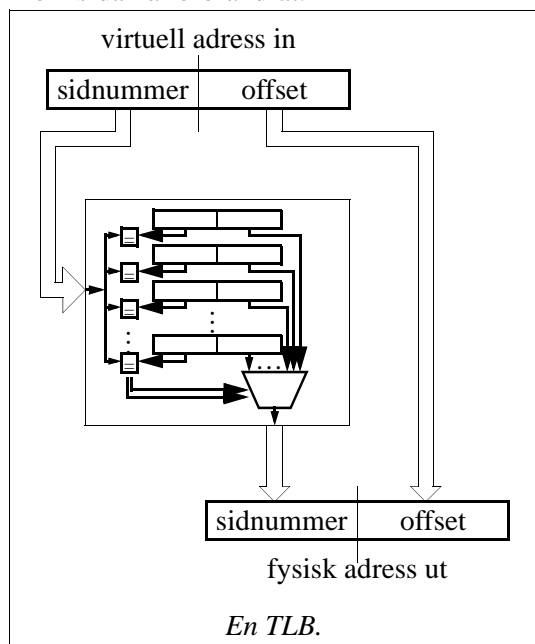
Sidtabellen används vid varje läsning, instruktionshämtning och skrivning i primärminnet. Processorn måste alltså läsa i sidtabellen (som finns i primärminnet) varje gång den ska läsa eller skriva någonting i primärminnet.

Om inget knep används så kommer datorn därför att gå hälften så fort som utan adressöversättning. Varje minnesåtkomst medför ju en extra läsning i sidtabellen.

Det knep som används är att ha ett cacheminne för adressöversättningar. Detta cacheminne kallas för TLB (*Translation Lookaside Buffer*).

En TLB är ofta fullt associativ och har cirka 100 platser eller mindre.

Vid träff i TLB:n översätter denna det inkommande sidnumret från programmet till ett nytt sidnummer som kan användas för den verkliga primärminnesåtkomsten. Offset inom sidan är oförändrat.



### Virtuell och fysisk adressering

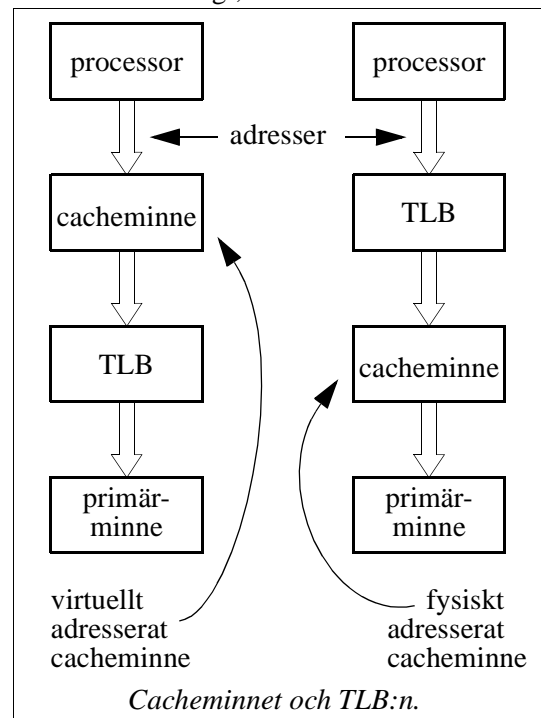
En intressant fråga är om cacheminnet ska placeras mellan processor och TLB, eller mellan TLB och primärminne. De båda möjligheterna visas i figuren här nere till höger.

Ett cacheminne som sitter mellan processor och TLB kallas för *virtuellt adresserat*. Denna placering har två nackdelar:

- Alla platser i cacheminnet måste ogiltigförklaras (invalideras) om innehållet i översättningstabellerna ändras. Ändringar sker 10—1000 gånger per sekund i datorer som kör flera program samtidigt.
- Flera olika virtuella adresser kan översättas till en och samma fysiska adress av TLB:n. Då kan cacheminnet lagra samma primärminnesinnehåll flera gånger med olika adresslapp. Skrivning till en av adresserna ändrar bara en av kopiorna, men inte alla.

Cacheminnen som sitter mellan TLB och primärminne kallas *fysiskt adresserade*. De har som främsta nackdel att en tidsödande översättning sker vid varje referens. Detta förlänger tiden för en cacheminnesläsning högst märkbart, vilket kan kräva lägre klockfrekvens hos processorn.

Lyckligtvis finns ett sätt att gå runt denna nackdel hos fysiskt adresserade cacheminnen, nämligen att göra uppslagning i cache och TLB samtidigt, det så kallade *tricket*.



### Tricket

Lägg märke till att en del av adressen passerar oförändrad genom TLB:n. Offset inom en sida ska inte ändras vid översättningen, utan bara sidnumret. Det kan utnyttjas för att bygga ett fysiskt adresserat cacheminne lika snabbt som ett virtuellt adresserat.

Offset kan användas för att indexera cacheminnet samtidigt som sidnumret används för uppslagning i TLB:n. De två långsamma uppslagningarna (i cacheminne respektive TLB) görs alltså samtidigt i stället för i tur och ordning.

Figuren här nedanför visar vårt direktmappade cacheminne från sida 12, ombyggt för parallell TLB-uppslagning.

För att detta trick ska fungera ställs följande krav på cacheminnets storlek och associativitetstal i förhållande till virtuelltminnessystemets sidstorlek:

$$\frac{\text{cacheminnesstorlek}}{\text{associativitetstal}} \leq \text{sidstorlek}$$

Om inte kravet är uppfyllt, så kommer en eller flera bitar ur sidnumret att behövas för att indexera cacheminnets minneskretsar. I och med det kan adressen inte längre översättas samtidigt med cacheminnesåtkomsten.

### Tricket konsekvenser

Det är en stor fördel att ha ett fysiskt adresserat cacheminne, om det är lika snabbt som ett med virtuella adresser. Därför används tricket i många moderna processorer.

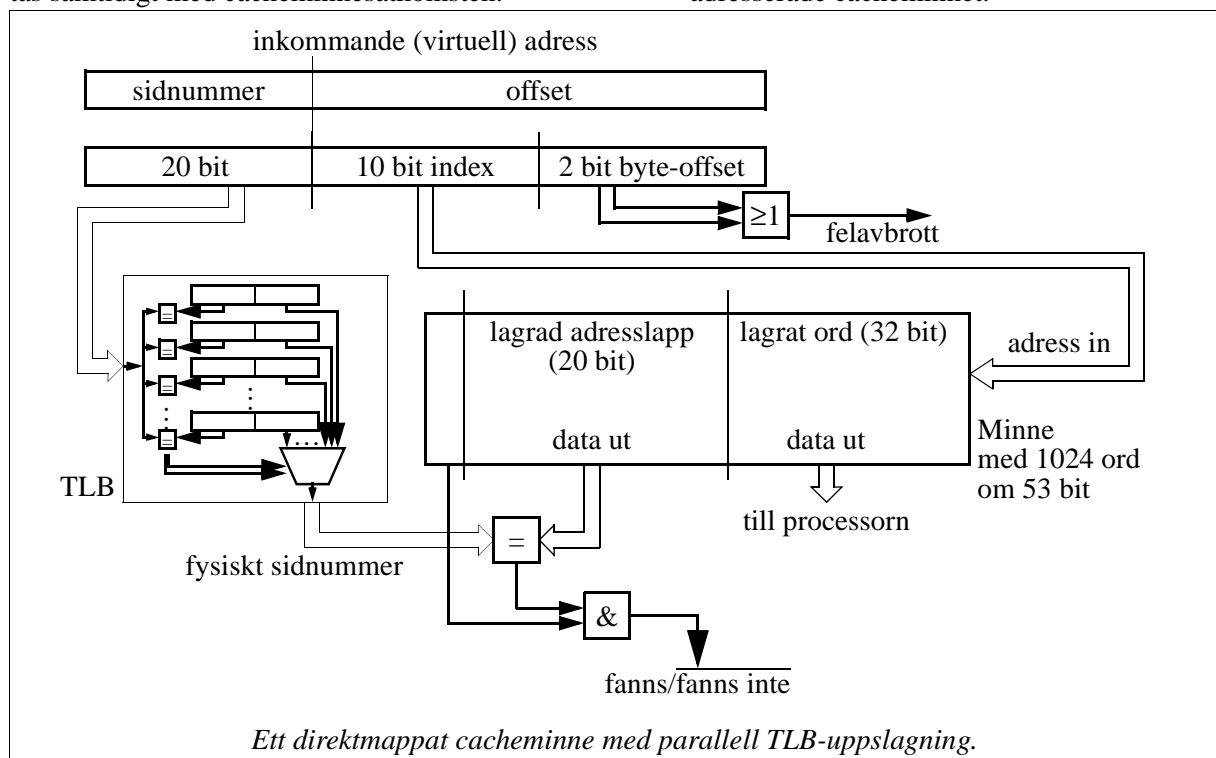
Men olika datorer i en serie ska kunna köra samma operativsystem. Det betyder att sidstorleken inte får ändras mellan datorerna i serien, eftersom operativsystemet måste modifieras om sidstorleken ändras.

Med tiden har det blivit möjligt att ha stora cacheminnen. Det går däremot inte att öka sidstorleken eftersom operativsystemet då skulle behöva ändras.

Ett direktmappat cacheminne kan inte rymma mer än en sida om tricket ska användas. Vissa mikroprocessorer har cacheminnen med höga associativitetstal (3–5), för att kunna ha stort cacheminne trots liten sidstorlek.

De senaste åren har virtuellt adresserade cacheminnen börjat förekomma i avancerade mikroprocessorer. Då kan cacheminnet göras stort, vilket ger en prestandaförbättring.

Tydligen väger denna förbättring tyngre än problemen med invalidering vid processbyte och risken för dubbla kopior i det virtuellt adresserade cacheminnet.



## TLB-missar

I äldre datorer fanns maskinvara (eller mikroprogram) som aktiverades vid TLB-miss, läste sidtabellen i primärminnet och uppdaterade innehållet i TLB:n.

Läsning av sidtabellen är så komplicerat att det inte gärna låter sig göras i maskinvara. Risc-datorer konstrueras därför så att TLB-miss medför felavbrott.

En programrutin som hör till operativsystemet startas vid felavbrottet. Denna programrutin läser sidtabellen och skriver lämpliga värden i en av platserna i TLB:n.

Metoden att hantera TLB-missar med en programrutin gör att TLB:ns utbytespolitik och en del andra parametrar kan varieras utan ändringar i maskinvaran.

I Intels x86-processorer används dock mikroprogram (eller något som påminner om mikroprogram) för att sköta TLB-missar. Orsaken är att processorerna måste anpassas till äldre program, som förutsätter det beteendet.

Låt oss studera en minnesreferens i en dator som använder tricket. Vi antar att referensen är en instruktionshämtning.

TLB:n innehåller inte det inkommande sidnumret utan vi får en TLB-miss. Instruktionen kan då inte hämtas eftersom processorn inte vet var i det fysiska minnet instruktionen finns. Den lagrade adresslapp som cacheminnet levererar kan inte användas, eftersom det inte finns något fysiskt sidnummer att jämföra adresslappen med.

Processorn gör ett felavbrott och startar programrutinen för TLB-missar. Denna programrutin slår upp sidnumret i sidtabellen, som finns i primärminnet. Låt oss anta att sidtabellen innehåller en översättning för det efterfrågade sidnumret.

Då väljer programrutinen ut en lämplig plats i TLB:n och lägger in översättningen där. Sedan startas instruktionshämtningen om.

Eftersom TLB:n nu innehåller en lämplig översättning får vi TLB-träff. TLB:n levererar ett fysiskt sidnummer, som jämförs med det lagrade från cacheminnet. Stämmer adresslappen så har vi fått träff även i cachen.

## Skilda TLB:er för instruktioner och data

I verkligheten är det vanligt att processorn har två TLB:er. Den ena används då vid instruktionshämtning och kallas I-TLB; den andra används vid läsning och skrivning av data och kallas D-TLB.

Motivet för uppdelningen i I-TLB och D-TLB är att dataöversättning och instruktionsöversättning måste kunna ske samtidigt. Med bara en TLB så skulle instruktionshämtningen få vänta när en översättning sker för en datareferens, det vill säga vid varje LOAD- eller STORE-instruktion.

Det skulle fördröja instruktionshämtningen varje gång en LOAD- eller STORE-instruktion utförs, precis som i datorer med gemensamt cacheminne för instruktioner och data. Se sida 7.

Ett undantag är om cacheminna är virtuellt adresserade. I så fall kan en gemensam TLB användas, eftersom den bara utnyttjas vid cachemiss.

Separata instruktions- och datacacheminnen med tillhörande I-TLB och D-TLB brukar användas närmast processorn, det vill säga för första cachenivån. Vid miss i något av förstanivåcacheminna gör processorn ett försök i andranivåcacheminnet.

Andranivåcacheminnet brukar vara gemensamt för instruktioner och data, men det ger ingen nämnvärd försämring av prestanda.

Ett andranivåcacheminne används ju bara vid *miss* i något av förstanivåcacheminna, vilket är relativt sällan. Med gemensamt förstanivåcacheminne uppstår konflikt både vid träff och vid miss.

Det förekommer experiment med två TLB-nivåer. Det innebär att datorn förses med förstanivå- och andranivå-TLB:er, där andranivå-TLB:n avsökts automatiskt vid miss i första nivån.

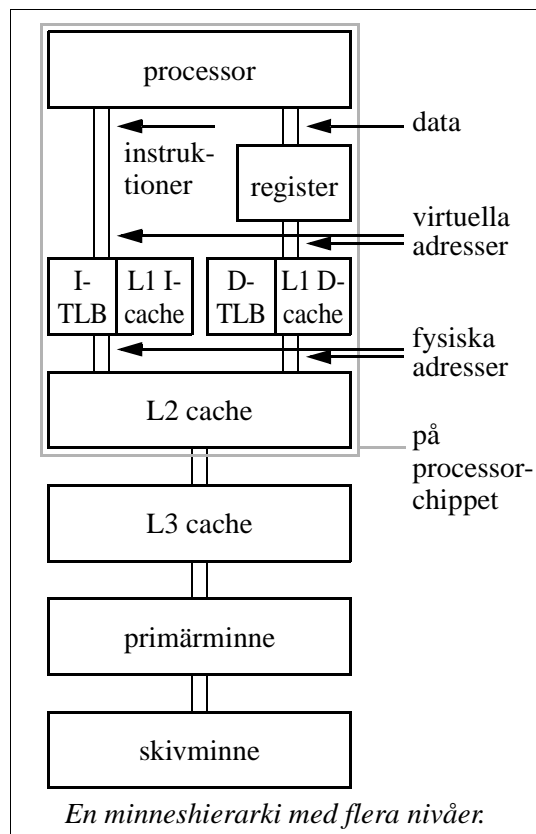
## En utökad minneshierarki

Registren kan även de ses som ett slags cacheminne. Deras användning definieras av det maskinkodsprogram som körs.

I viss mening är det förstås programmeraren som hanterar registren. De flesta program skrivs dock i något komplicerat språk, varvid kompilatorn avgör hur registren används.

Registren finns på processorchippet. Det samma gäller nästan alltid förstani-våcacheminnet. Andranivåcacheminnet brukar dock ofta byggas upp med andra chipp. Eftersom anslutningar mellan två chipp kräver långsamma drivkretsar får andranivåcacheminnet längre åtkomsttid än förstani-våcacheminnet.

För att det ska vara meningsfullt att införa två nivåer av cacheminne måste förstas andranivåcacheminnet vara snabbare än primärminnet. Det är det också, eftersom cacheminnet byggs upp med statiska minneskretsar, som är snabba men dyra och strömslukande. För primärminnet används dynamiska minnen. Dessa är strömsnåla, stora och billiga, men inte lika snabba som statiska minnen.



## Tumregel för minneshierarkin

En intressant fråga är hur många nivåer en minneshierarki bör ha.

Som tumregel anger man ofta att åtkomsttiden bör ändras en faktor tio mellan två intilliggande nivåer. Regeln gäller oftast för de översta paren i hierarkin på föregående sida:

- register — L1 cache,
- L1 cache — L2 cache,
- L2 cache — L3 cache, och
- L3 cache — primärminne.

Skivminnet, som finns längst ned i hierarkin, är betydligt billigare per lagrad bit än primärminnet. Åtkomsttiden för skivminnet är cirka 1000 gånger så lång som för primärminnet.

Det förefaller lämpligt att ha ett cacheminne mellan skivminne och primärminne. Sådana cacheminnen är också vanligt förekommande, men de brukar göras i programvara.

Operativsystemet, som administrerar alla överföringar mellan skivminne och primärminne, sparar nyligen lästa skivminnesblock i en reserverad del av primärminnet. Denna reserverade primärminnesdel fungerar som cacheminne för skivminnesåtkomster.

Det är ingen särskild skillnad mellan den reserverade primärminnesdelen och det övriga primärminnet. Uppdelningen görs helt i operativsystemets programvara; laddas ett nytt operativsystem så ändras uppdelningen.

Åtkomsttiden för den reserverade primärminnesdelen blir lite längre än för det vanliga primärminnet eftersom varje åtkomst måste passera operativsystemets styrprogram (på engelska: device driver) för skivminnet.<sup>†</sup>

Moderna skivaggregat (engelska: disk drives) brukar fördes med cacheminne på samma sätt. De har redan från början en inbyggd styrdator. Numera fördes styrdatorn ofta med extra minne och programvara som håller de senast lästa delarna av skivan snabbt tillgängliga. Funktionen kallas ofta *track buffer*.

<sup>†</sup> En plats i detta programvarubaserade cacheminne kallas på engelska *disk buffer*. Hela cacheminnet kallas *buffer cache*; det består ju av en samling *disk buffers*.

## En stor minneshierarki

Vi kan utöka vår minneshierarki till att omfatta två cacheminnesnivåer på processorchippet och en tredje mellan processorchipp och primärminne. Tar vi med buffertarna för skivminnet så får vi figuren här nedanför.

Processorkretsen sitter i en sockel på ett kretskort, och på detta kretskort finns minneskretsar för tredjenivåcacheminnet. Där finns också primärminnet, som av operativsystemet delas in i en "vanlig" del och en del med buffertar för skivminnesåtkomster.

Skivaggregatet innehåller även det en buffert, förutom själva skivan.

En notering: endast skivan behåller sitt innehåll utan matningsspänning. Därför måste operativsystemet utföra ett antal olika instruktioner och kommandon för att tvinga data nedåt i hierarkin innan datorn stängs av.

