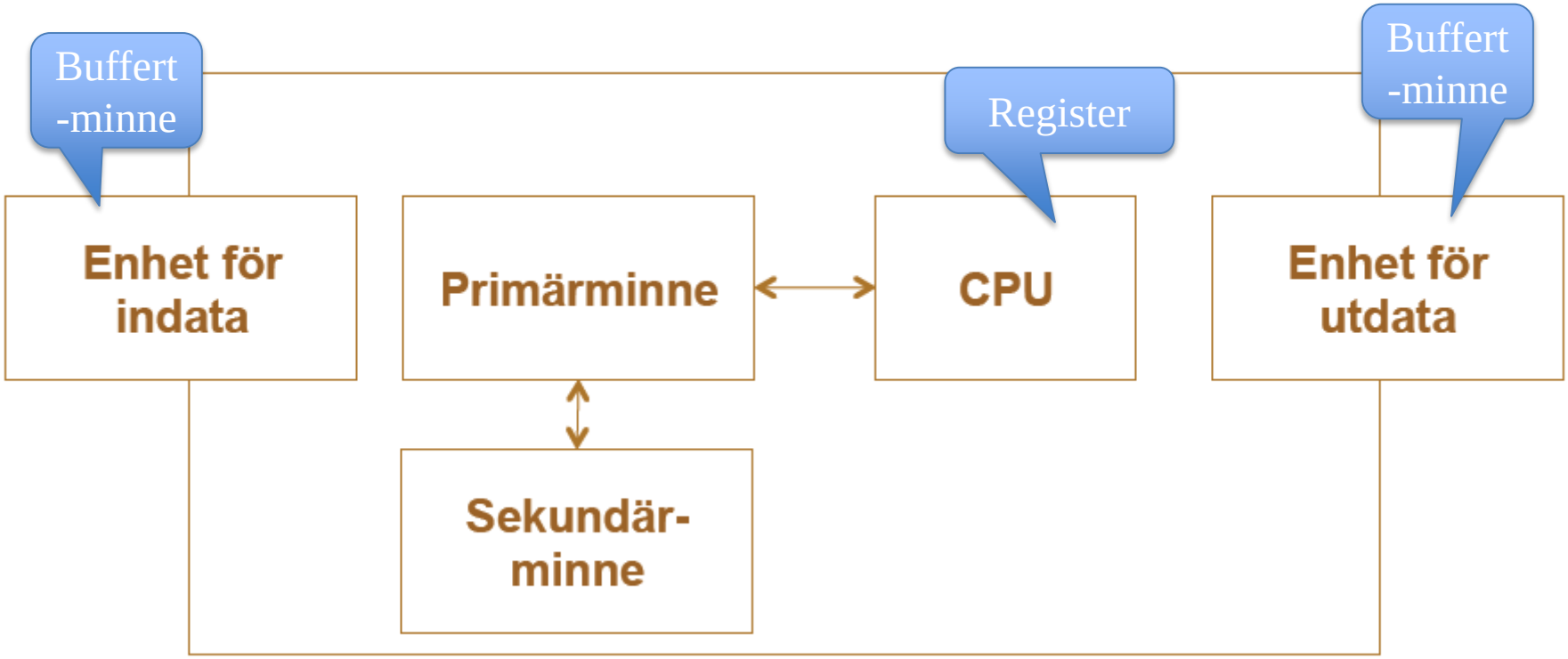
Minneshierarkier

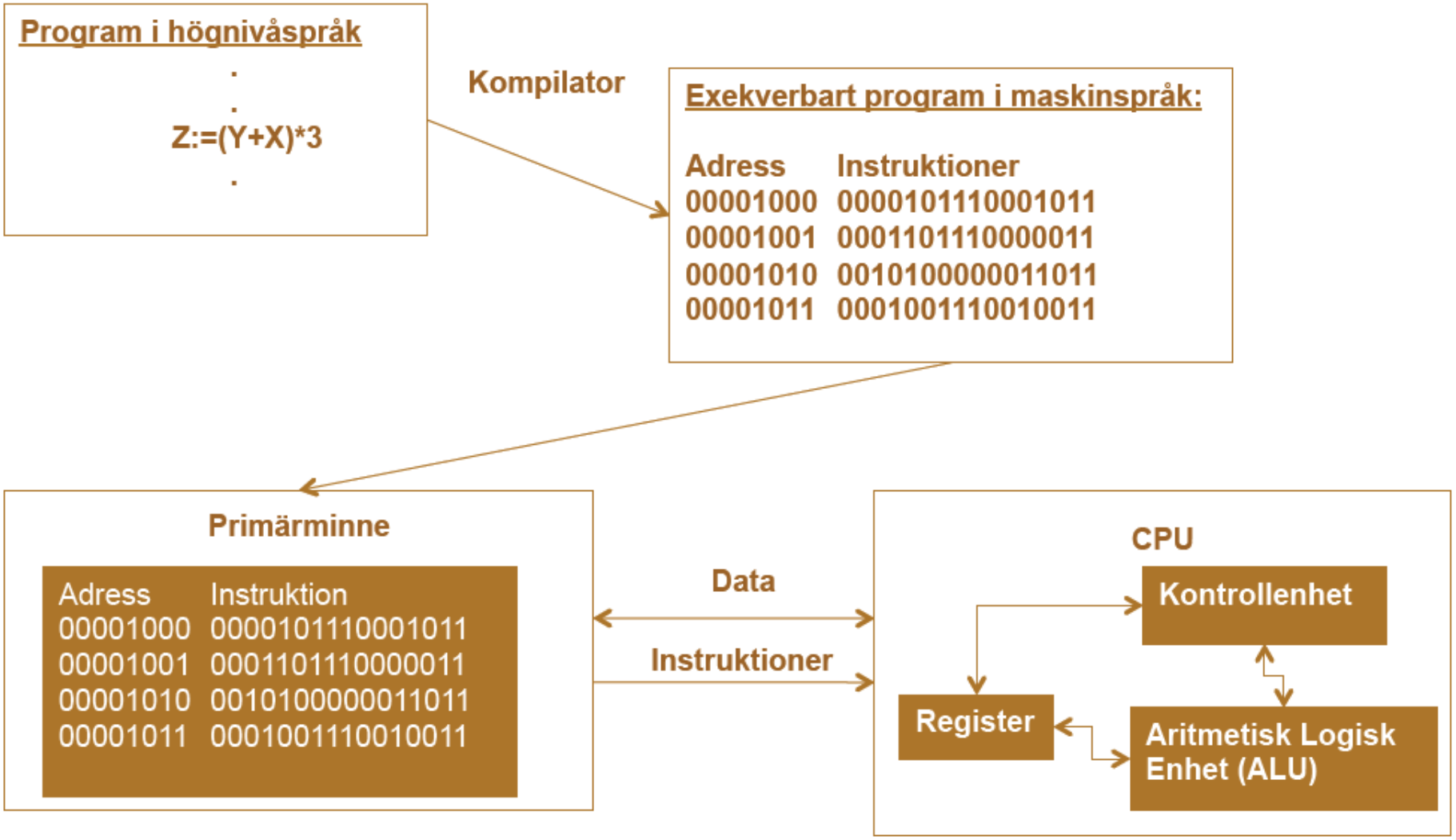
Cache-minne

Lite om virtuellt minne

Vy av datorsystemet ur ett minnesperspektiv

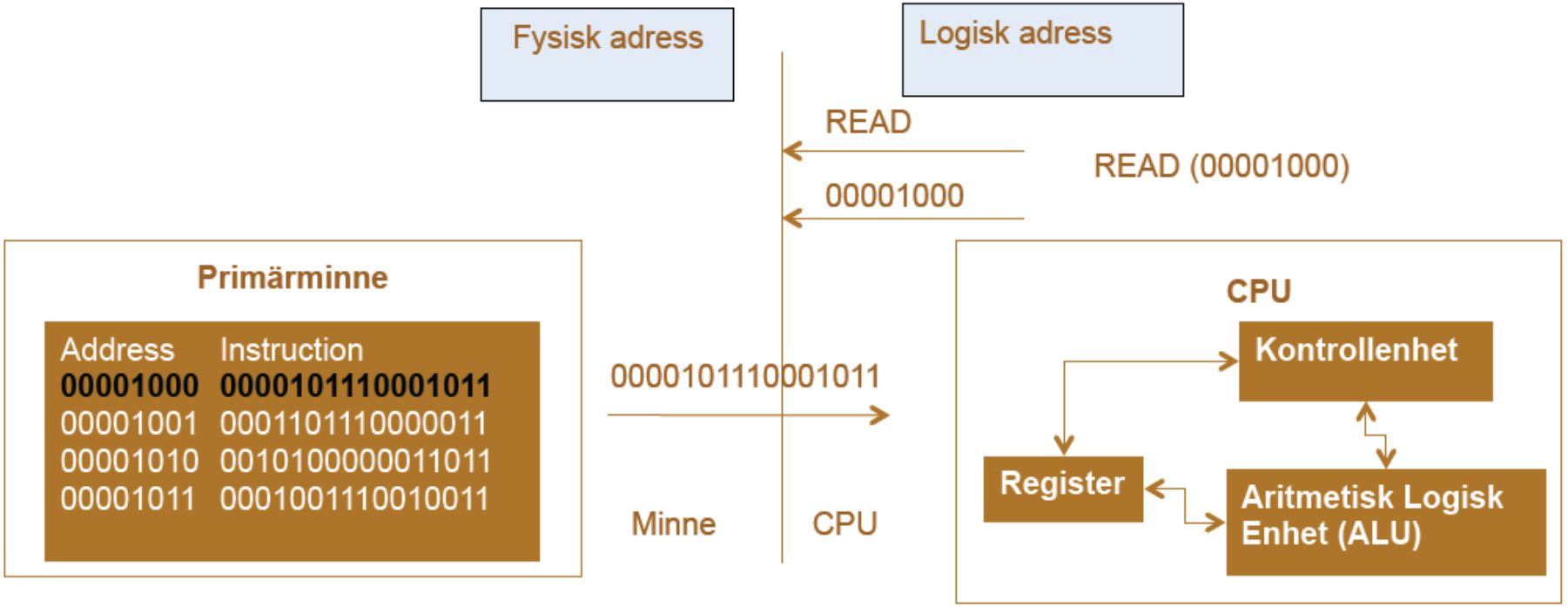


Programexekvering



Minnet ur processorns synvinkel

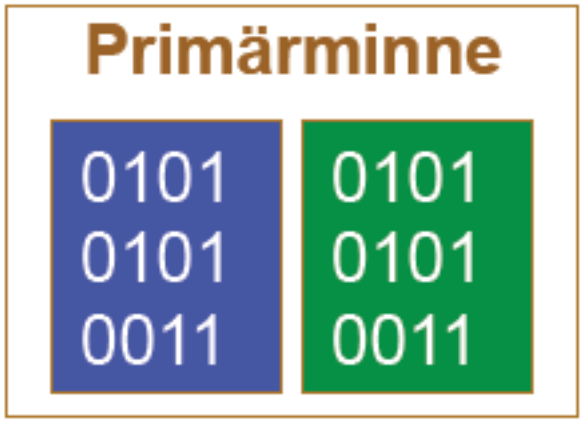
- Processorn ger kommando/instruktion med en adress och förväntar sig data tillbaka



Minneshantering

- Vid multiprogrammering finns flera olika program i primärminnet. Kostar för mycket tid att flytta program till hårddisk vid kontextbyte

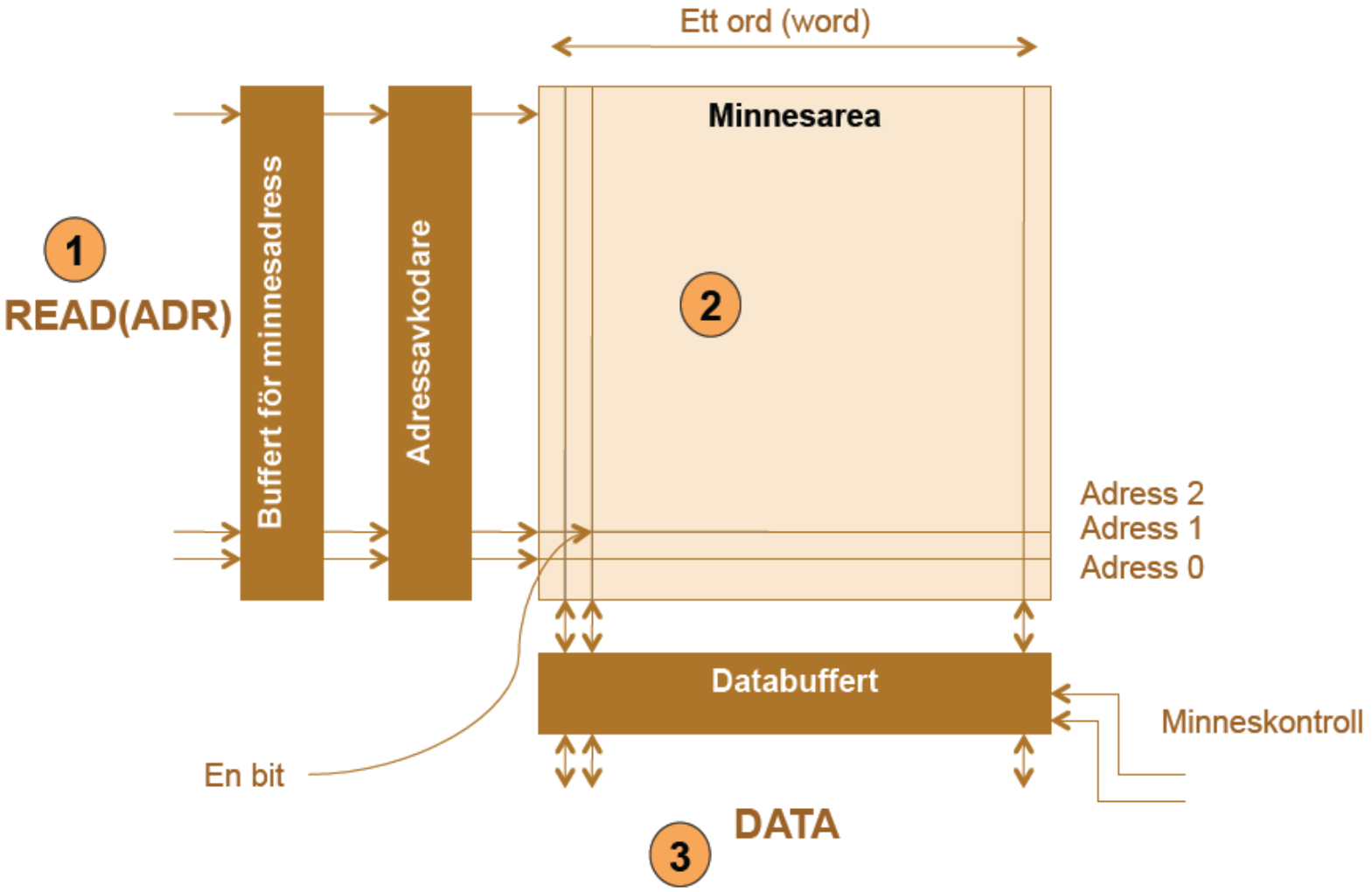
- Ex: Två program körs ”samtidigt”



Minnet

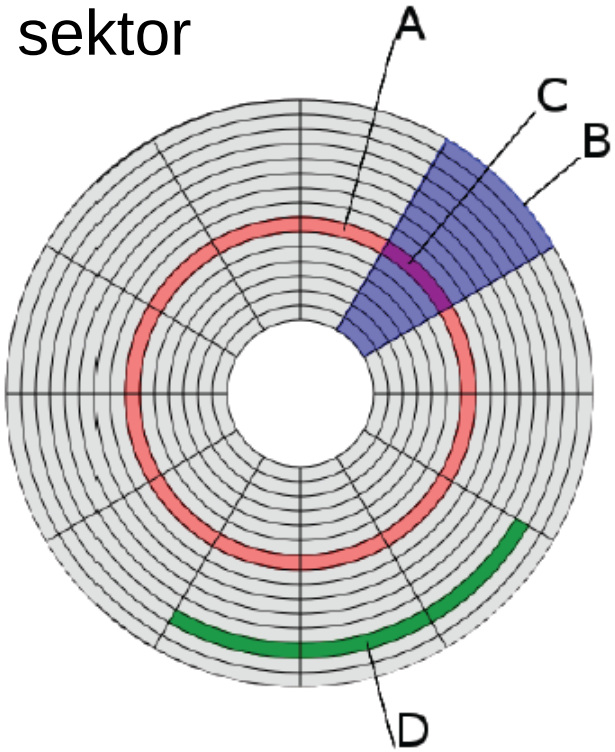
- Minnet kan delas upp i primärminne och sekundärminne
- Primärminnet förlorar sitt innehåll när strömmen bryts. Minnet är flyktigt (*volatile*)
  - RAM (*Random Access Memory*)
    - Dynamiska (DRAM), Statiska (SRAM)
- Sekundärminnet behåller sitt innehåll när strömmen stängs av. Minnet är icke-flyktigt (*non volatile*)
  - Hårddisk, flashminne, magnetband
  - CD, DVD

Primärminne



Sekundärminne, diskar

- A – spår, (B – geometrisk sektor), C – sektor
  D – kluster av sektorer



- En sektor kan vara 512 – 4096 byte och består av *sector header*, *data area* och *error correction code*

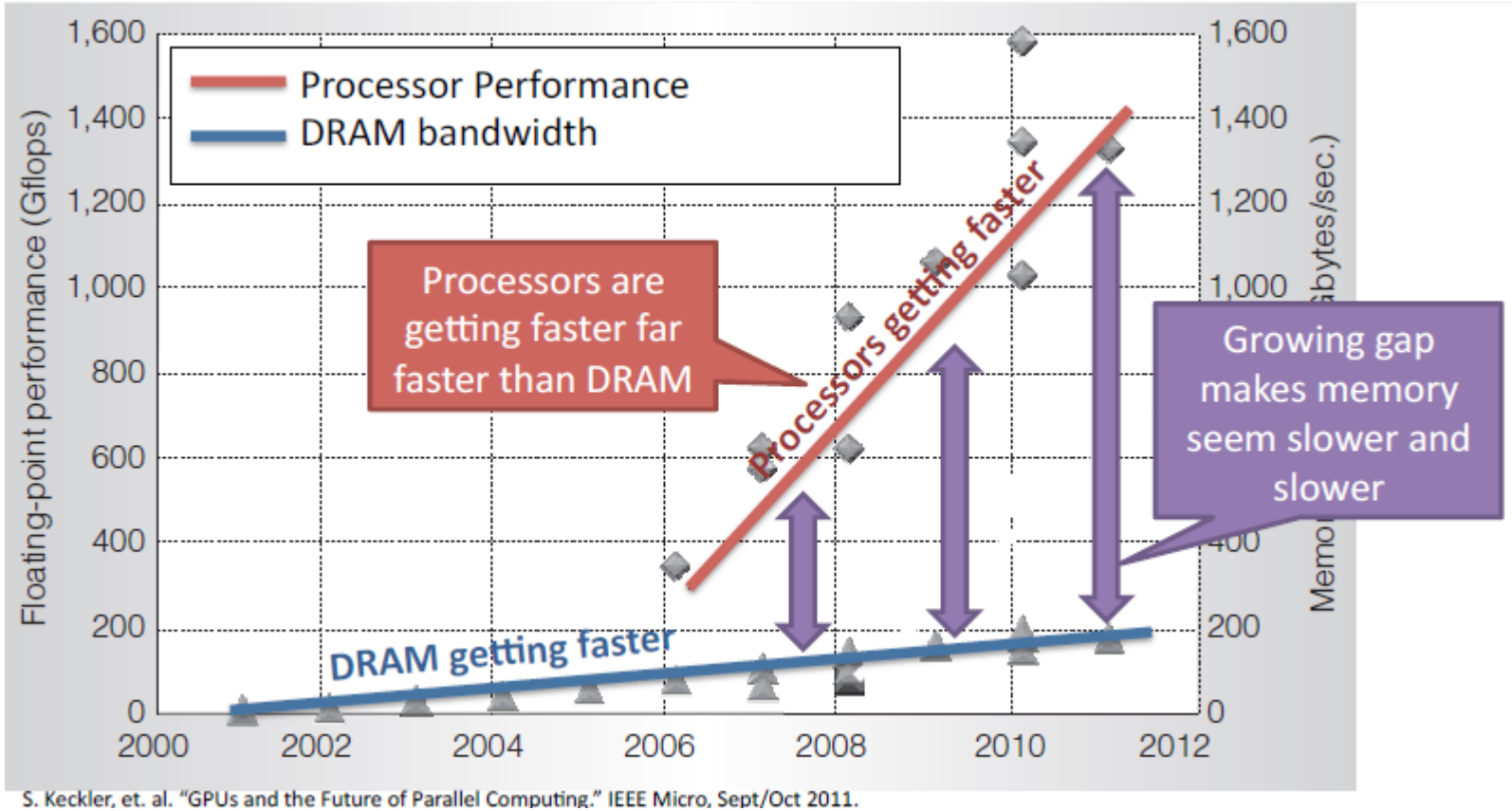
Design av minnessystem

- Vad vill vi ha?
  - Ett minne som har plats för stora program och som är lika snabbt som processorn

- Problem
  - Processorer arbetar i hög hastighet och behöver stora minnen
  - Minnen är mycket långsammare än processorer

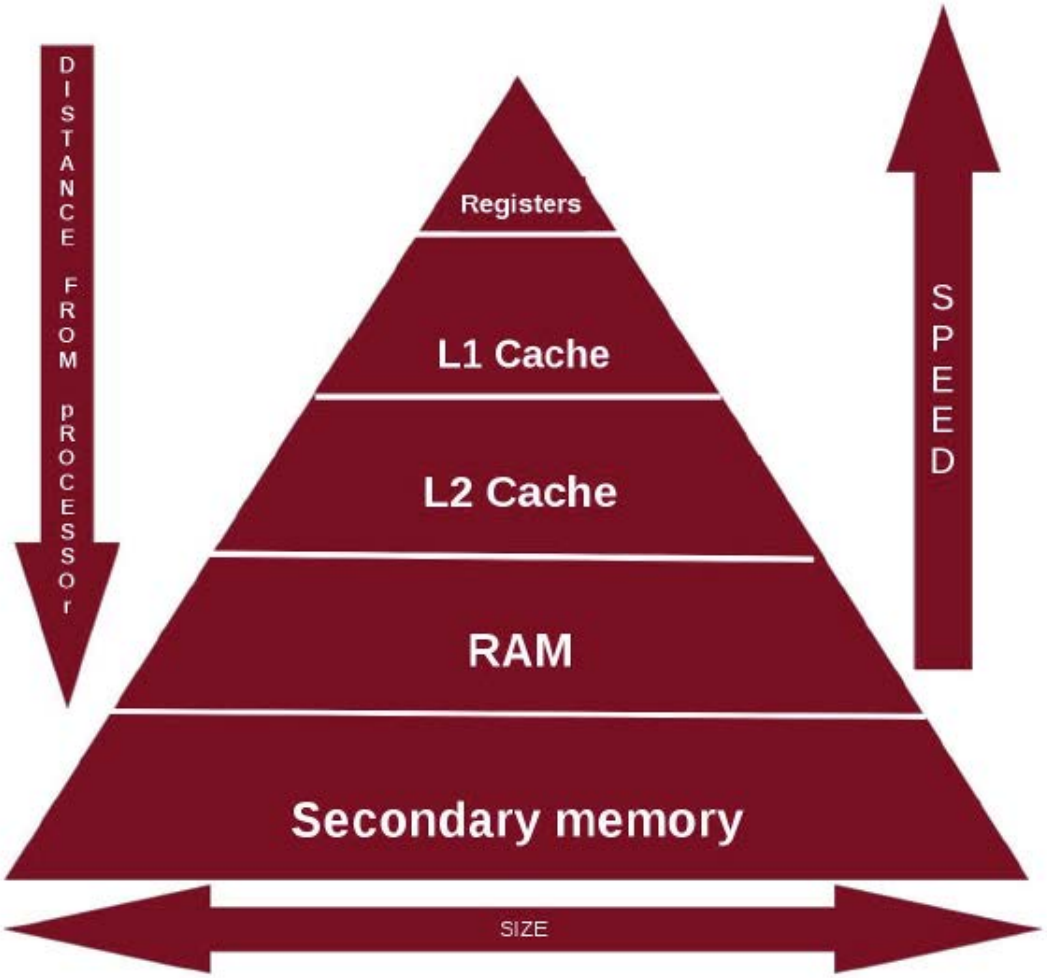
- Fakta
  - Stora minnen är långsammare än små
  - Snabbare minnen kostar mer per bit

Prestandautveckling



S. Keckler, et. al. "GPUs and the Future of Parallel Computing." IEEE Micro, Sept/Oct 2011.

Minneshierarki



Minneshierarki, prestandajämförelse

- Processorregister:
  8-32 register på (32-64 bitar => 32 – 256 byte)
  accesstid: få ns (1 klockcykel)

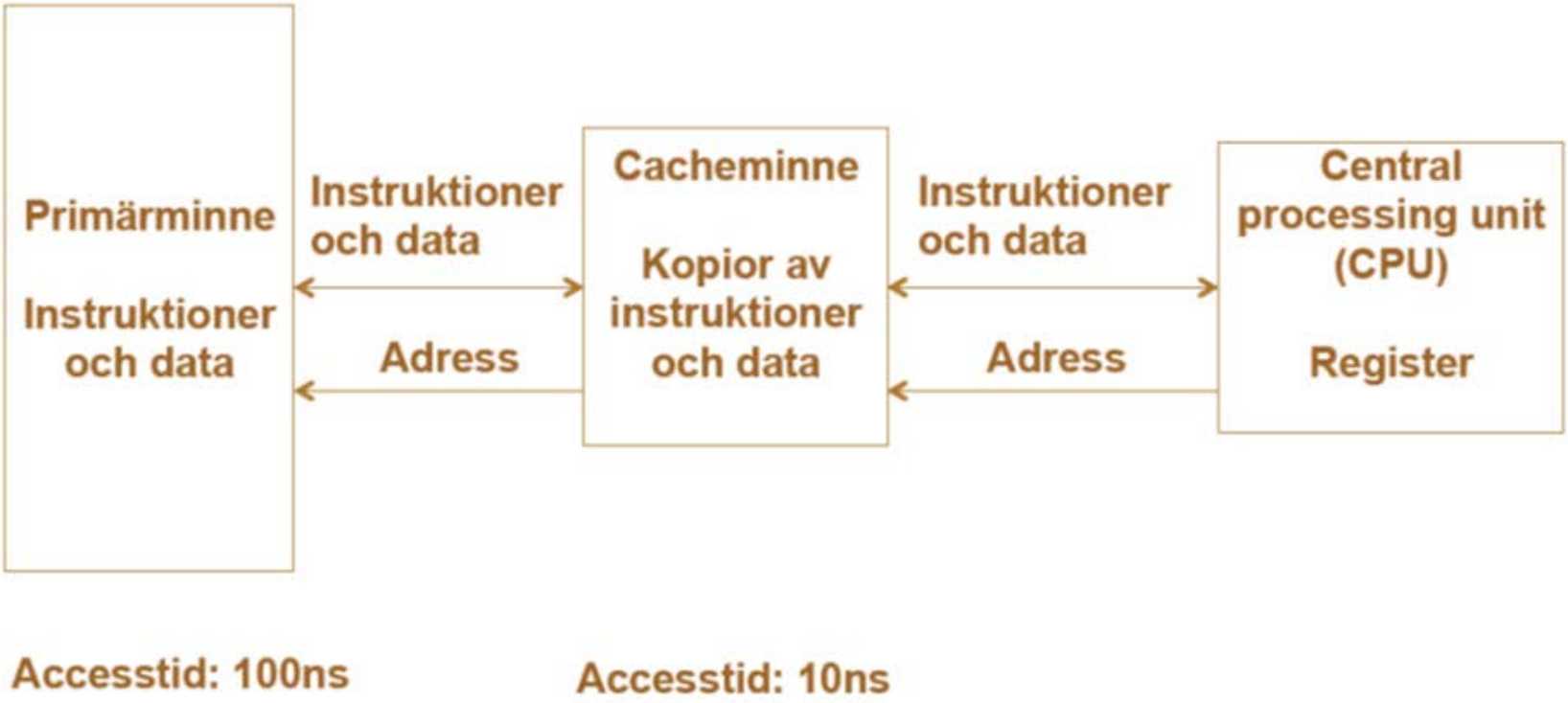
- On chip cache memory:
  32 byte – 128 KiB
  accesstid: ~10ns (3 klockcykler)

- Off-chip cache memory:
  128 KiB – 12 MiB
  accesstid: ~tiotals ns (10 clockcykler)

- Primärminne (main memory):
  250 MiB – 4 GiB
  accesstid: ~100 ns (100 klockcykler)

- Hårddisk:
  1 GiB – 1 Tib
  accesstid: 10-tals ms (10 000 000 klockcykler)

Cache-minne


Primärminne
Instruktioner och data
Instruktioner och data
Adress
Cacheminne
Kopior av instruktioner och data
Instruktioner och data
Adress
Central processing unit (CPU)
Register
Accesstid: 100ns
Accesstid: 10ns

Cache-minne

- Ett cacheminne är mindre och snabbare än primärminnet
  - Hela program får inte plats
  - Data och instruktioner ska vara tillgängliga när de behövs

- Om man inte har cacheminne:
  - Accesstid för att hämta en instruktion=100ns

- Om man har cacheminne:
  - Accesstid för att hämta en instruktion=100+10=110 ns
    - Först ska instruktionen hämtas till cacheminne och sedan hämtas instruktionen från cacheminnet till CPU

Cache exempel 1

- Program: Assemblerinstruktioner

```
x = x + 1;      ADD    r0, r0, #1
y = x + 5;      ADD    r1, r0, #5
z = y + x;      ADD    r2, r1, r0
```

- Om man inte har cacheminne:
  - Accesstid för att hämta en instruktion=100ns
    - Tid för att hämta instruktioner från RAM: 3*100=300ns
- Om man har cacheminne:
  - Accesstid för att hämta en instruktion=100+10=110ns
    - Tid för hämta instruktioner från RAM: 3*110=330ns

Cache exempel 2

Antag:

- 1 maskininstruktion per rad
- 100 ns för minnesaccess till primärminnet
- 10 ns för minnesaccess till cacheminnet

• Programmet och dess maskininstruktioner (**x** är initierad till 1000 och **y** till 500)

Exempelprogram:

```
while (x > 0){
     x = x - 1;
     printf("x = %d", x);

}



while (y > 0){
    y = y - 1;
    printf("y = %d", y);

}
```

Assembler (förenklad):

```
          BAL  lLabel1          //instr1
lCond1 :
          SUB r2, r2, #1        //instr2
          MOV  r1, r2           //instr3
          BL  printf            //instr4
lLabel1 :
          CMP  r2, #0           //instr5
          BGT lCond1            //instr6
          BAL  lLabel2          //instr7
lCond2 :
          SUB  r3, r3, #1       //instr8
          MOV  r1, r3           //instr9
          BL   printf           //instr10
lLabel2 :
          CMP  r3, #0           //instr11
          BGT  lCond2           //instr12
```

Utan cache, exempel 2
(exekveringsordning)

Antal instruktioner

1
2
3
4
5
6
7
:
4999
5001
5002
5003
5004
5005
5006
5007
:
7504
7505
7506

Minnesaccess till en instr: 100 ns

Totalt 6006 instr

Total accesstid: 7506*100ns = 750,6µs

Instruktioner som exekveras:

```
BAL  lLabel1              //instr1
CMP  r2, #0               //instr5
BGT  lCond1               //instr6
SUB  r2, r2, #1           //instr2
MOV  r1, r2          //instr3
BL   printf          //instr4
CMP  r2, #0          //instr5

MOV  r1, r2          //instr3
BL   printf          //instr4
CMP  r2, #0          //instr5
BGT  lCond1          //instr6
BAL  lLabel2         //instr7
CMP  r3, #0          //instr11
BGT   lCond2         //instr12
SUB   r3, r3, #1     //instr8

BL    printf         //instr10
CMP  r3, #0          //instr11
BGT   lCond2         //instr12
```

Med cache, exempel 2

Antal instruktioner	Instruktioner som exekveras:	
1	`BAL lLabel1`	`//instr1`
2	`CMP r2, #0`	`//instr5`
3	`BGT lCond1`	`//instr6`
4	`SUB r2, r2, #1`	`//instr2`
5	`MOV r1, r2`	`//instr3`
6	`BL printf`	`//instr4`
:		
4999	`MOV r1, r2`	`//instr3`
5001	`BL printf`	`//instr4`
5002	`CMP r2, #0`	`//instr5`
5003	`BGT lCond1`	`//instr6`
5004	`BAL lLabel2`	`//instr7`
5005	`CMP r3, #0`	`//instr11`
5006	`BGT lCond2`	`//instr12`
5007	`SUB r3, r3, #1`	`//instr8`
5008	`MOV r1, r3`	`//instr9`
5009	`BL printf`	`//instr10`
:		
7502	`BL printf`	`//instr10`
7503	`CMP r3, #0`	`//instr11`
7504	`BGT lCond2`	`//instr12`

Minne + cache: 100 ns + 10 ns

Cache-access: 10 ns

Minne + cache: 100 ns + 10 ns

Cache-access: 10 ns

Totalt 7506 instr

Total accesstid: 12*100ns + 7506*10ns = 76,26µs

Cacheminne - bakgrund

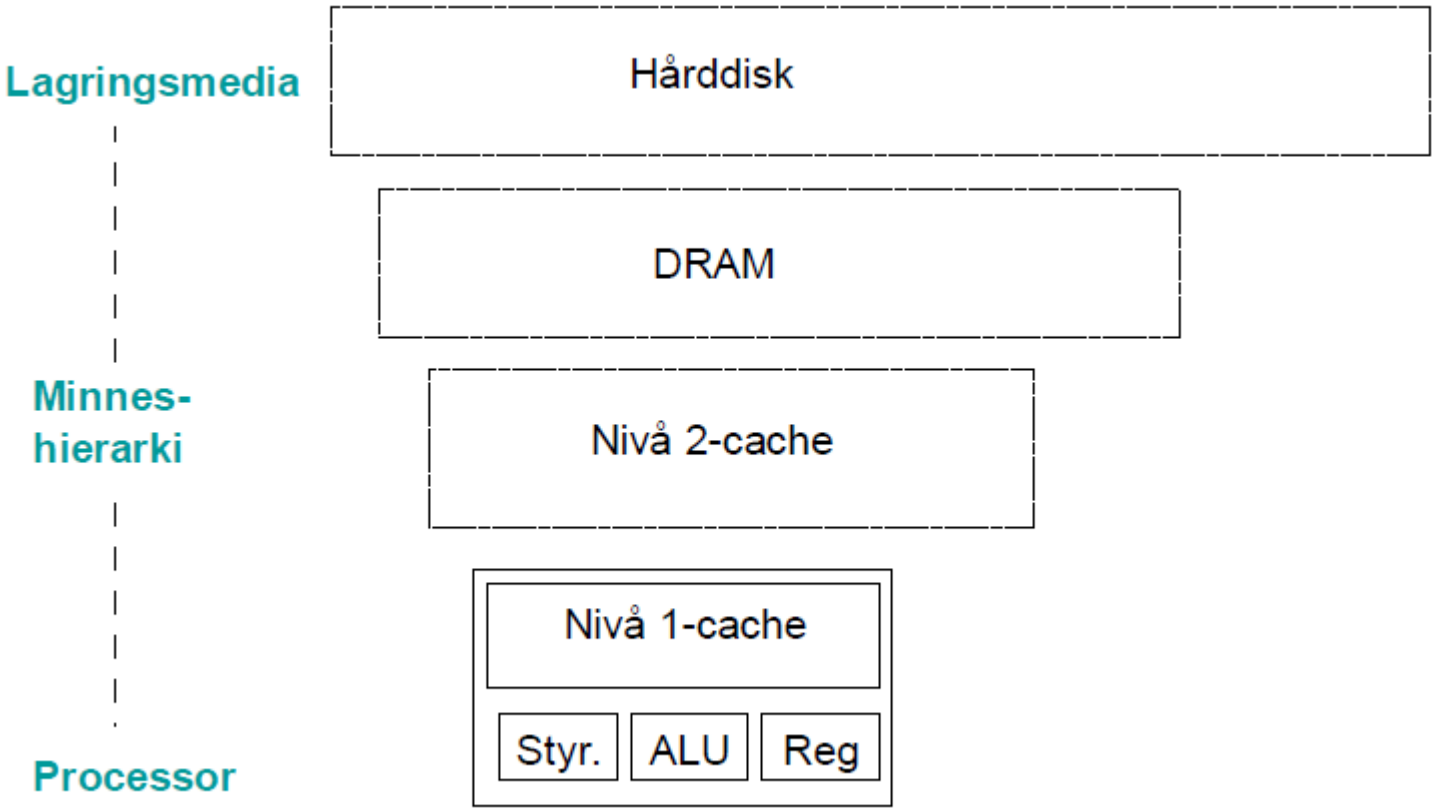
- Minnesreferenser tenderar att gruppera sig under exekvering
  - både instruktioner (t ex loopar) och data (datastrukturer)

- Lokalitet av referenser (*locality of references*):
  - Temporal lokalitet – lokalitet i tid –
    - om en instruktion/data blivit refererad nu, så är sannolikheten stor att samma referens görs inom kort igen
  - Spatial lokalitet – lokalitet i rum -
    - om instruktion/data blivit refererad nu, så är sannolikheten stor att instruktioner/data vid adresser i närheten kommer användas inom kort

Utnyttja lokalitet

- Minneshierarki
  - Lagra allt på hårddisk
  - Kopiera nyligen refererade och närliggande delar från disk till det mindre primärminnet
  - Kopiera de mest nyligen refererade och närliggande delarna från primärminne till cacheminne
    - Cacheminne kopplat till CPU

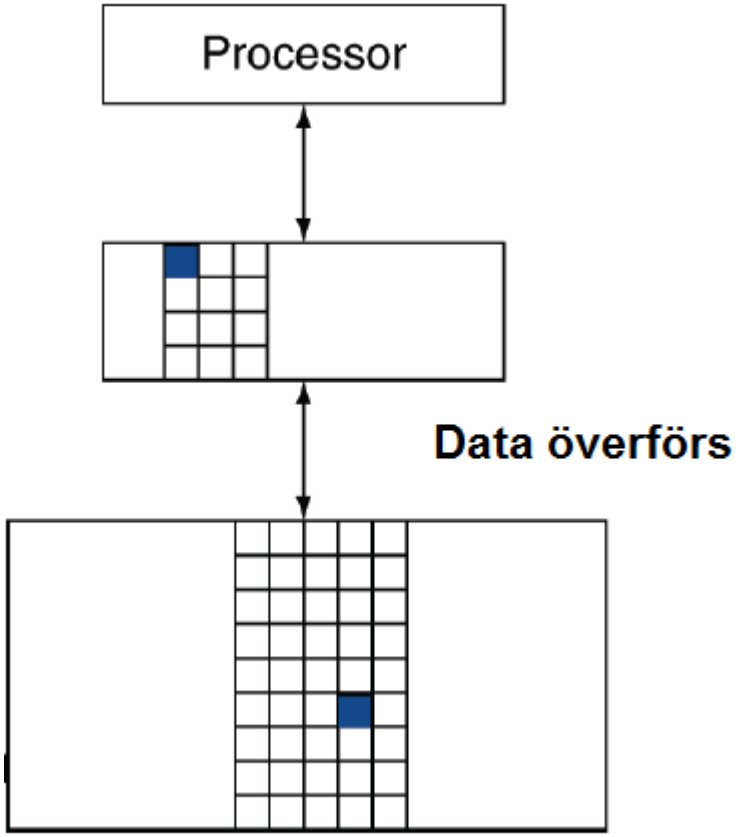
Minneshierarki - Systemnivå



Disk i storleksordningen 10 000 000 gånger långsammare än cache och 100 000 gånger långsammare än DRAM

Minneshierarki - nivåer

- Block (*line*): enhet som kopieras
  - Kan vara flera ”words”
- Om ”accessed data” finns i översta nivån (upper level)
  - Hit: access ges av högsta nivå
    - Hit ratio: hits/antal accesser
- Om accessad data inte finns på aktuell nivå
  - Miss: block kopieras från lägre nivå
  - Tid: miss penalty,
    Miss ratio: antal missar/antal accesser = 1 – hit ratio
  - Därefter kan data nås från högre nivå



Cacheminne - liknelse

- Liten byrå (4 lådor) och källarförråd (plats för 16 lådor)

- Plats för alla kläder finns i källarförrådet, men vissa kläder som används mycket förvaras i byrån

Läsning med cache-minne

- Antag att CPU vill läsa en adress som hör till ett visst block i primärminnet
- Finns blocket i cache-minnet?
- Vid träff fås svar från cache-minnet
- Vid miss läses blocket över från primärminnet till cache-minnet
- Gammalt block kastas ut om cache-minnet är fullt

Skrivning med cache-minne

- Antag att CPU vill skriva till en adress som hör till ett visst block i primärminnet
- Läs in blocket från primärminne till cache-minne på samma sätt som vid läsning
- Alternativ 1: Skriv alltid både till cacheminnet och till primärminnet (write-through)
- Alternativ 2: Vänta med att skriva till primärminnet tills detta block kastas ut ur cache-minnet (write-back)

Cache-minne

Valid-bit

”Adresslapp”

Datablock

Block-index

Index	V	Tag	Data
000	N		
001	N		
010	N		
011	N		
100	N		
101	N		
110	N		
111	N		

Det här cache-minnet har plats för 8 cache-block (*cache lines*)

Cache-minne, forts

- Hur vet vi att specifik datadel finns i cache?
- Om den finns där, hur hittar vi den?

Direkt adressmappning:

Varje minnesblock mappas till ett cache-block

- **många minnesblock mappas till *samma* cache-block**

Direkt cache-mappning

Återfinns genom adressmappning

Byteadressen kan delas in i tre delar:

Tag	Index	Offset

Tag = En adresslapp som innehåller den adressinformation (den övre delen av adressen) som inte ryms i cache-adressen.

Index = vilket cacheblock informationen hamnar i

Offset = anger vilken byte* i varje cacheblock

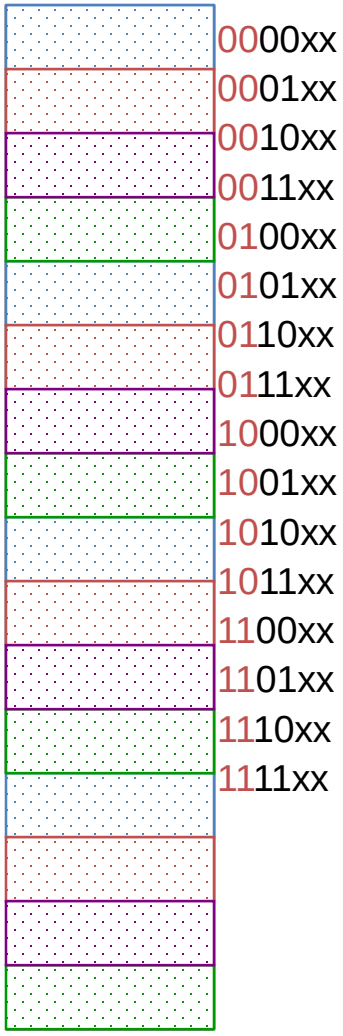
**Om minnet är byteadresserat*

Enkelt cache-exempel

Cache

Index	Valid	Tag	Data
00			
01			
10			
11			

Primärminne



Block om ett word (4 byte)
De två lägsta bitarna definierar vilken byte i ordet det är frågan om

Hur hittar vi den?

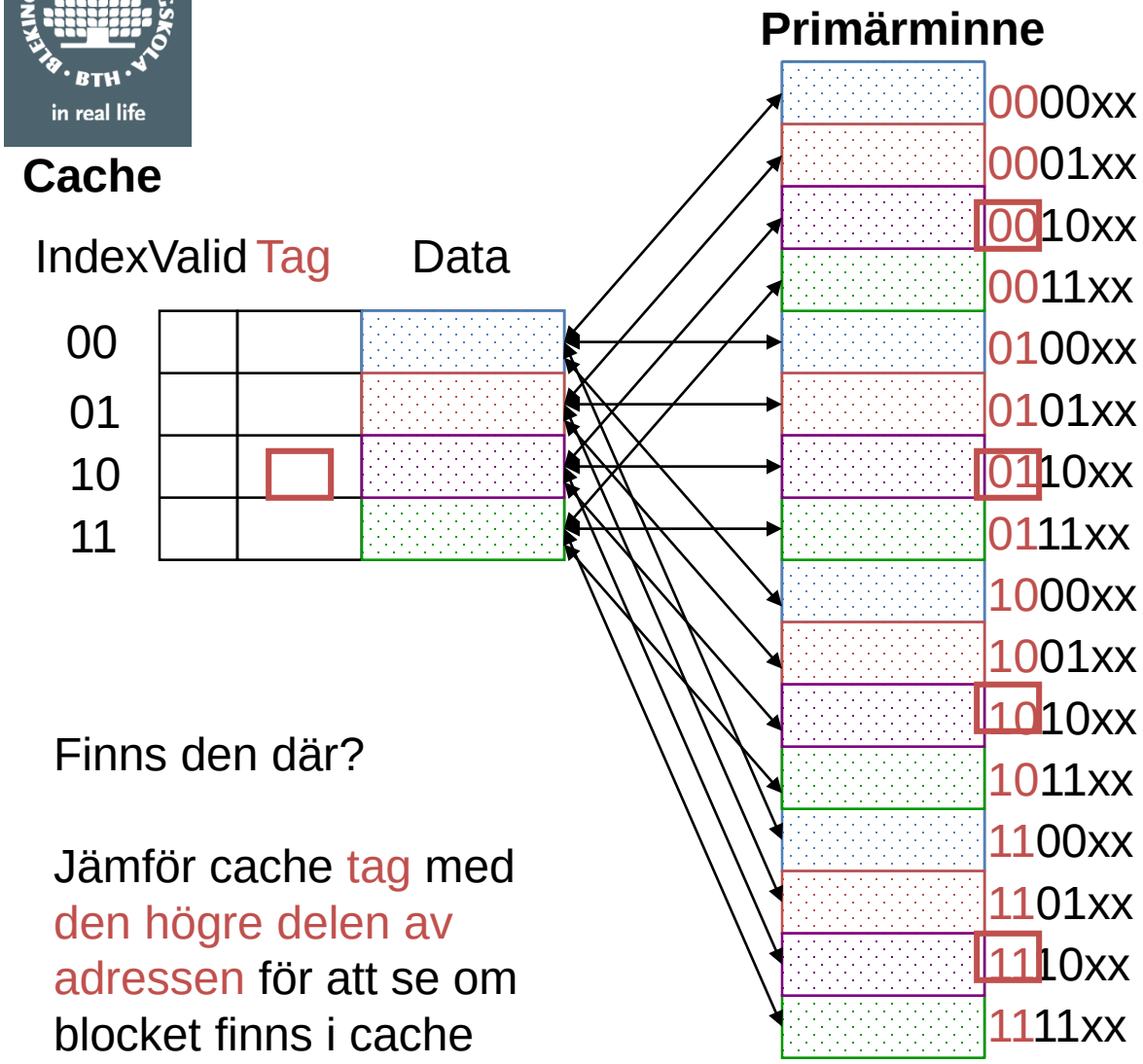
Använd de nästa två bitarna – för att se vilket cacheblock (dvs, cacheindex = (blockadress) modulo (antal block i cachen)

Finns den där?
→Gå in på rätt index.

Jämför cache tag med den högre delen av adressen för att se om blocket finns i cache

(blockadress) = adressbitarna ovanför offseten

Enkelt cache-exempel, forts



Block om ett word (4 byte)
De två lägsta bitarna definierar vilken byte i ordet det är frågan om

Hur hittar vi den?

Använd de nästa två bitarna – för att se vilket cacheblock (dvs, dvs, cacheindex = (blockadress) modulo (antal block i cachen)

Finns den där?

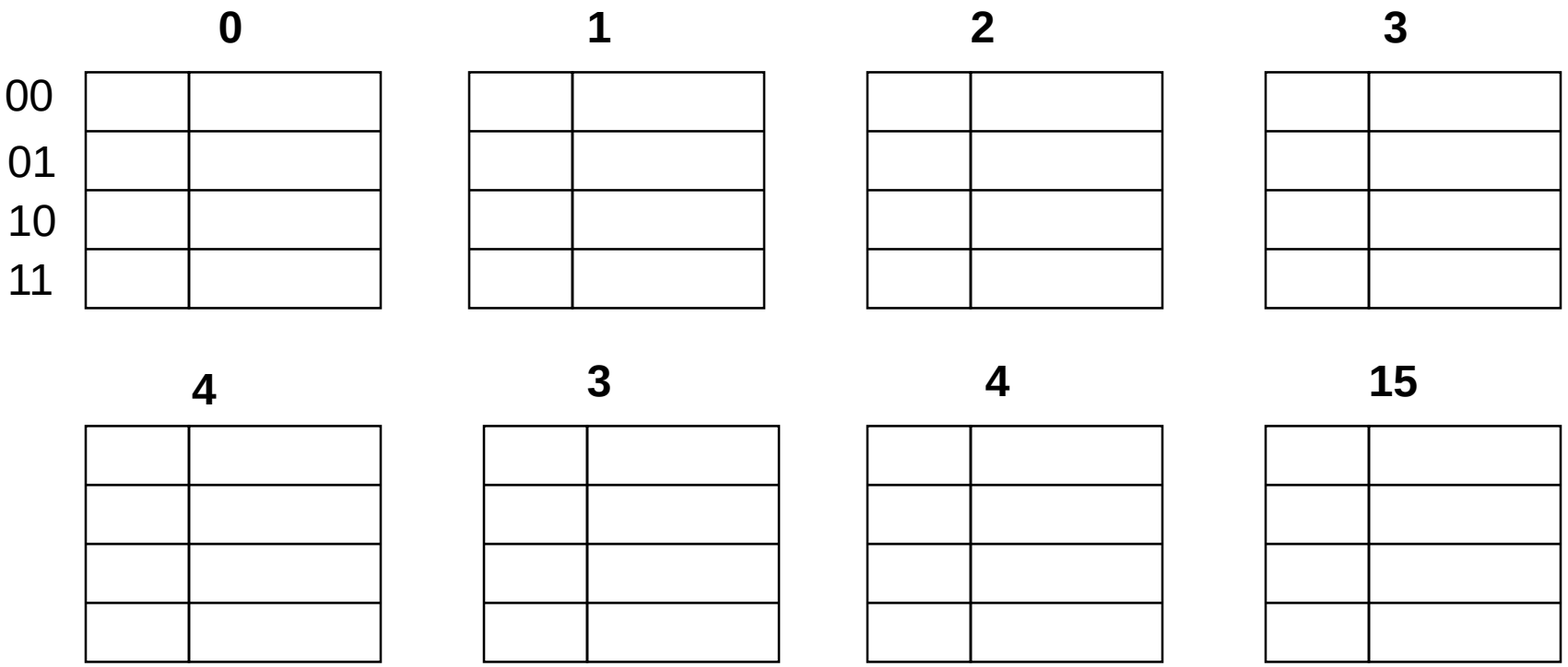
Jämför cache tag med den högre delen av adressen för att se om blocket finns i cache

(blockadress) = adressbitarna ovanför offseten

Direktmappad cache

Antag att vi vill nå minnesdresser i följden **0 1 2 3 4 3 4 15**

Starta med en tom cache - alla block markerade som “not valid” vid start



Direktmappad cache, forts

Antag att vi vill nå minnesceller i följden **0 1 2 3 4 3 4 15**

Starta med en tom cache - alla block markerade som “not valid” vid start
Adresserna har 4 bitar => 2 bitar tag och 2 bitar index

0 miss

00	00	Mem(0)
01		
10		
11		

1 miss

00	00	Mem(0)
01	00	Mem(1)
10		
11		

2 miss

00	00	Mem(0)
01	00	Mem(1)
10	00	Mem(2)
11		

3 miss

00	00	Mem(0)
01	00	Mem(1)
10	00	Mem(2)
11	00	Mem(3)

01 **4** miss 4

00	~~00~~	~~Mem(0)~~
01	00	Mem(1)
10	00	Mem(2)
11	00	Mem(3)

3 hit

00	01	Mem(4)
01	00	Mem(1)
10	00	Mem(2)
11	00	Mem(3)

4 hit

00	01	Mem(4)
01	00	Mem(1)
10	00	Mem(2)
11	00	Mem(3)

15 miss

00	01	Mem(4)
01	00	Mem(1)
10	00	Mem(2)
11	~~00~~	~~Mem(3)~~

11 15

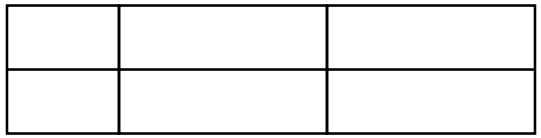
- 8 förfrågningar, 6 missar

Dra nytta av spatial lokalitet

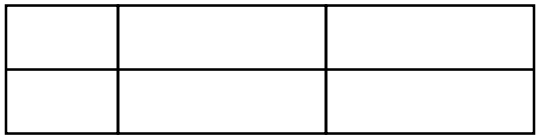
Låt cache-blocken vara större än ett dataord **0 1 2 3 4 3 4 15**

Starta med en tom cache - alla block markerade som “not valid” vid start
Nu får vi 2 bitar tag 1 bit index 1 bit offset.

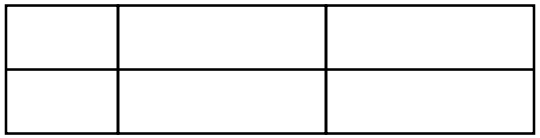
0



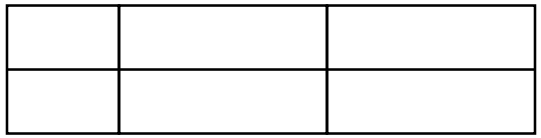
1



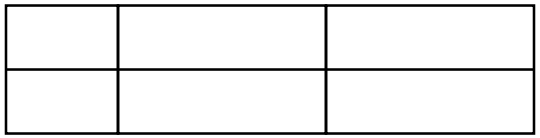
2



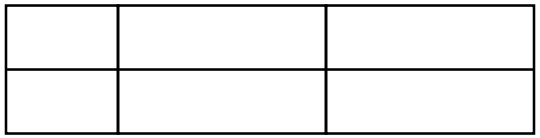
3



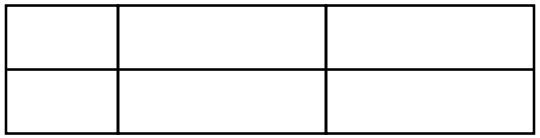
4



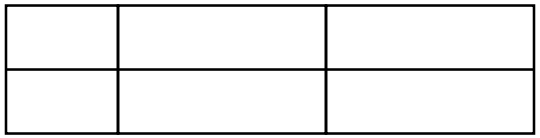
3



4



15



Dra nytta av spatial lokalitet, forts

Låt cache-blocken vara större än ett dataord **0 1 2 3 4 3 4 15**

Starta med en tom cache - alla blockmarkerade som “not valid” vid start

0 miss

0	00	Mem(1)	Mem(0)
1			

1 hit

00	Mem(1)	Mem(0)

2 miss

00	Mem(1)	Mem(0)
00	Mem(3)	Mem(2)

3 hit

0	00	Mem(1)	Mem(0)
1	00	Mem(3)	Mem(2)

4 miss

01

~~00~~	Mem(~~1~~) 5	Mem(~~0~~) 4
00	Mem(3)	Mem(2)

3 hit

01	Mem(5)	Mem(4)
00	Mem(3)	Mem(2)

4 hit

01	Mem(5)	Mem(4)
00	Mem(3)	Mem(2)

15 miss

11

01	Mem(5)	Mem(4)
~~00~~	Mem(~~3~~) 15	Mem(~~2~~) 14

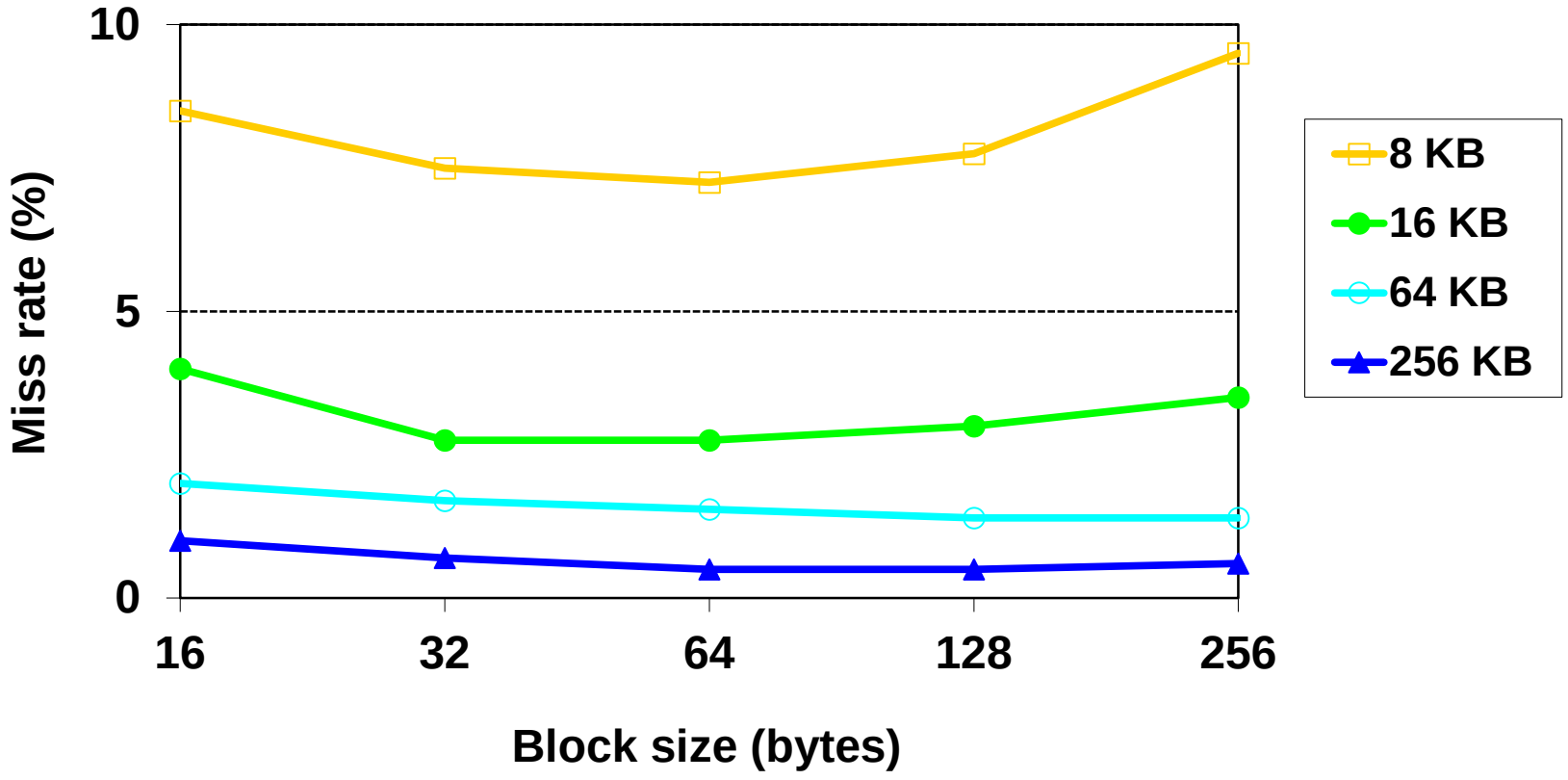
- 8 förfrågningar, 4 missar

Inverkan av blockstorlek i cache

- Större block minskar andelen cache-missar
  - spatial lokalitet

- Fix storlek på cache: större block => färre block
  - kan leda till högre frekvens av missar

Frekvens av cache-missar vs cache-storlek



Vad händer vid en cache-miss?

- Vid cache hit fortsätter CPU som vanligt, men vid miss …
  - stanna pipe-linen
  - hämta minnesinnehåll från primärminnet
  - Om det var en instruktionshämtning som stannades => börja om hämtfasen
  - Om det var en datacache-miss => fullfölj datahämtningen

Effekt av cache-miss

With 1 cycle cache

With 100 cycle DRAM

Skrivning till minne Write-Through

Innehållet i cache finns i två versioner (även i DRAM)

- Principen Write-Through uppdaterar DRAM varje gång en skrivning görs till cache
  - Det gör att skrivningar tar lång tid (jfr föregående bild)

- Kompromiss – skrivbuffert
  - Håller data som ska skrivas, och stannar bara pipe-linen för skrivning när bufferten är full

Skrivning till minne Write-Back

- Uppdatera bara data i cache
  - håll reda på att data förändrats i blocket med en så kallad ”**dirty bit**”

- När ett block som markerats med ”dirty” ska slängas ut från cache
  - skriv tillbaka till minnet

Data vs programkod

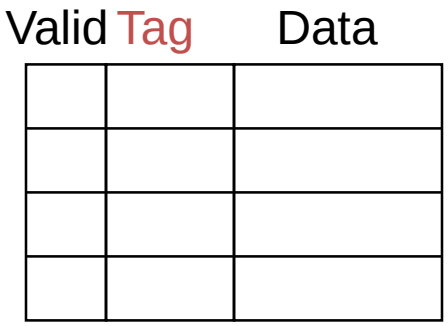
- Många datorer har olika cache för data och instruktioner (jfr D-cache och I-cache hos ARM)
- Det är nödvändigt för att möjliggöra instruktionshämtning och läsning/skrivning till minne samtidigt (full utnyttjandemöjlighet av pipe-line, Harvard-lik arkitektur)

Fler principer - associativ mappning

- Finns fler principer för hur minnesmappning till cache kan gå till
- Associativ mappning innebär att ett minnesblock kan placeras på mer än en plats i cacheminnet
- Tvåvägsassociativ mappning, betyder att varje minnesord kan placeras på en av *två olika* platser. Man ordnar det genom att ta bort en indexbit och lägga den i ”tag” (adresslappen) istället så att två block nu har samma index
- Nu blir inte utbytesalgoritmen längre trivial (FIFO,LRU eller random, kräver extra hårdvara)

Fullt associativ mappning

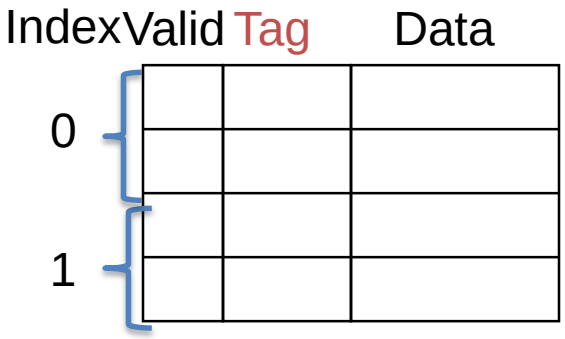
- Nu finns ingen indexdel i adressen som talar om var i cacheblocket minnesinnehållet ska läggas, det kan hamna var som helst.



- Vid access måste cachen sökas igenom för att se om aktuell tag ligger där

Tvåvägsassociativ mappning

- Två cacheblock har samma index, en adressbit har flyttats från indexdelen till tag-delen



- Nu måste man söka igenom två platser innan man vet om adressens innehåll finns i cache eller inte

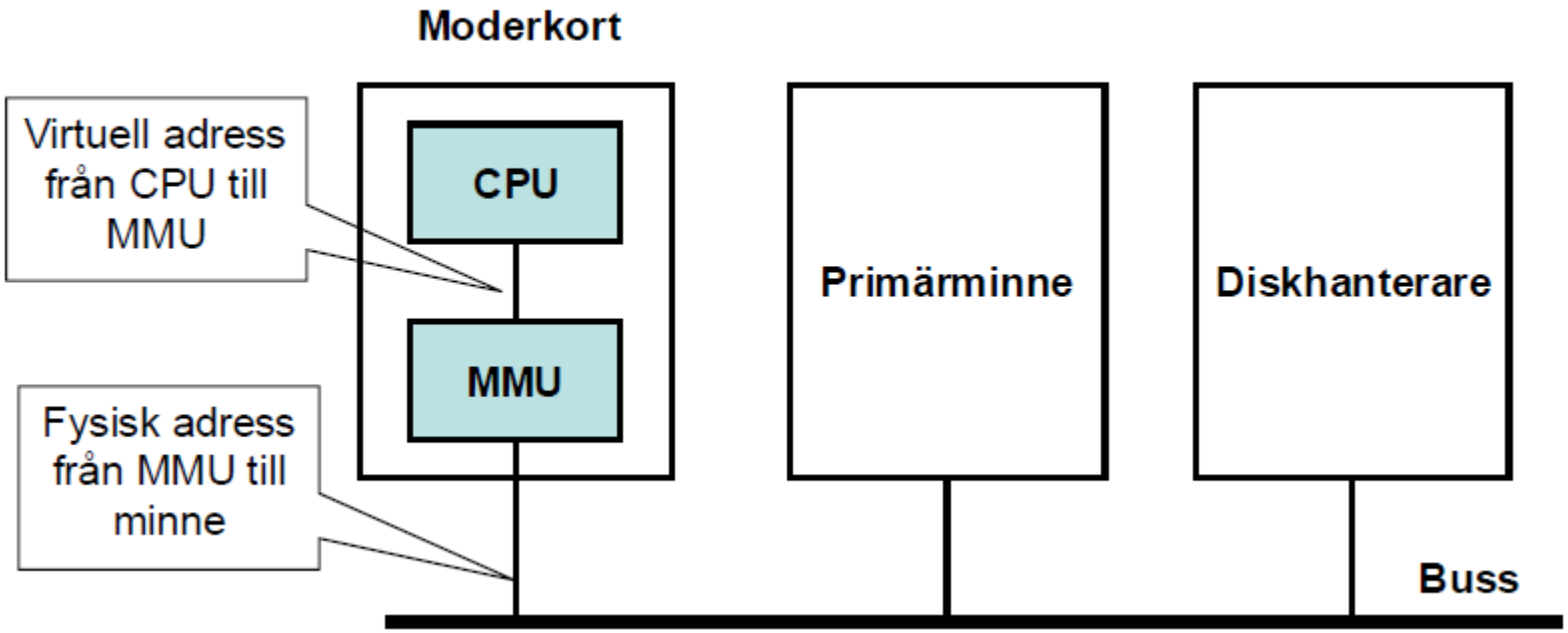
Virtuellt minne

- CPU arbetar med instruktioner och data som ligger i primärminnet
- Vad händer om ett program behöver större minnesutrymme än primärminnet?
- Virtuellt minne ”lurar” CPU att primärminnet är större än det egentligen är

Virtuellt minne, forts.

- Minnet delas in i sidor
- Sidor som inte får plats i primärminnet flyttas till sekundärminnet av Memory Management Unit (MMU, integrerat med processorn)
- ”Osynligt” för CPU
- Principen liknar cache-minne

Adressöversättning



MMU = Memory Management Unit

Virtuellt minne, forts

- Primärminnet fungerar som en ”cache” för sekundärminnet (t ex disk)
  - Tillåter effektiv och säker delning av minne mellan flera program
  - Gör det lätt att köra program som kräver mer minne än primärminnets storlek
  - Underlättar laddning av program för exekvering genom att tillåta relokering av kod (dvs går att placera var som helst i primärminnet)

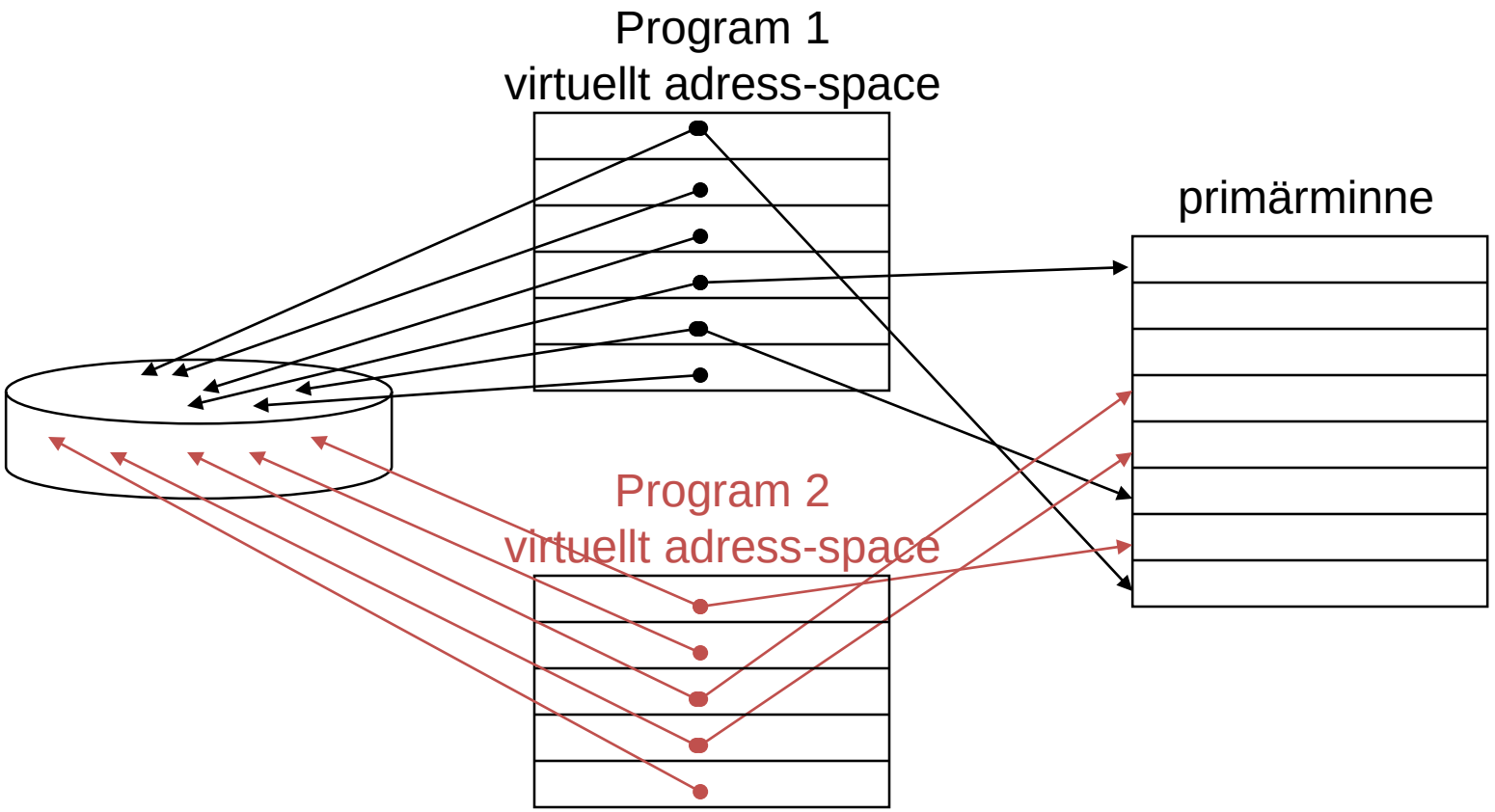
- Fungerar pga lokalitet
  - ett program arbetar troligen med en begränsad del av sitt adress-”space” under en viss tid

- Varje program kompileras till sitt eget adress-space, ett virtuellt sådant
  - vid exekvering översätts de virtuella adresserna till fysiska adresser i primärminnet

Två program som delar fysiskt minne

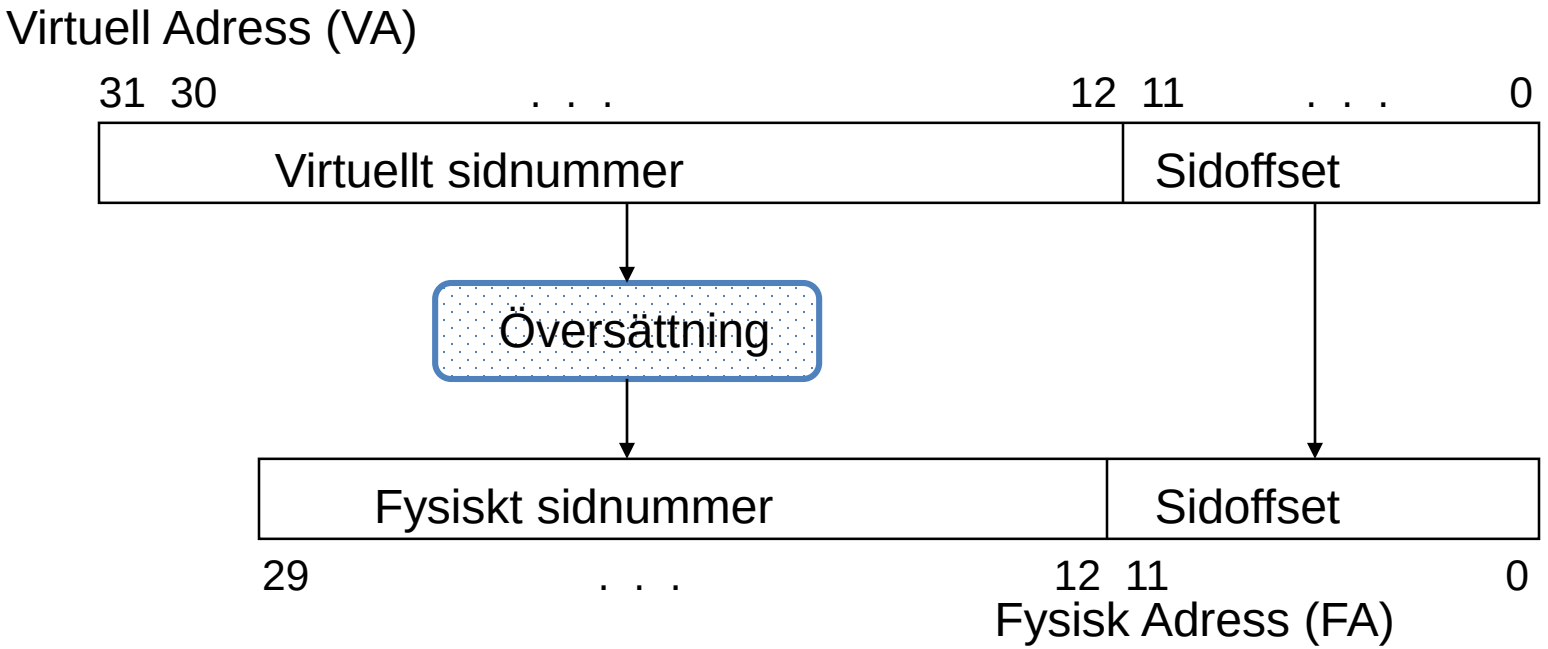
Ett programs adress-space delas in i sidor (pages)

- start på varje sida finns dokumenterad i programmets sidtabell (page table)



Adressöversättning

- En virtuell adress översätts till en fysisk adress med hjälp av en kombination av hårdvara (CPU) och mjukvara (OS)



- Varje minnesförfrågan kräver först en översättning från virtuell till fysisk adress
  - en virtuell minnesmiss kallas sidfel (page fault)

Adressöversättning, forts

- Sidtabellen som används vid översättning av adresser är ligger i primärminnet. Det betyder att varje minnesaccess (läsning/skrivning) kräver en extra läsning i minnet (segt!!!)

- Knep: Ett särskilt cache-minne för adressöversättning.
  - Kallas TLB (Translation Lookaside Buffer)

Vilken sida ska slängas ut?

Olika strategier:

- **First-In, First-Out (FIFO):**
  Släng ut den sida som funnits längst tid i primärminnet (enklast att bygga)

- **Least-Recently Used (LRU):**
  Släng ut den sida som inte använts på längst tid (utnyttjar lokaliteten bättre)

- **Random**

- **Mer om virtuellt minne i Realtids- och operativsystem**