

Computer Arkitektur og Operativ Systemer

Hukommelseshierarkiet

Forelæsning 8
Brian Nielsen

*Credits to
Randy Bryant & Dave O'Hallaron (CMU)*

Spørgsmål

- Hvorfor er program A ca. 10 gange hurtigere end program B??

A

```
int sum_array_rows(int a[M][N])
{
    int i, j, sum = 0;

    for (i = 0; i < M; i++)
        for (j = 0; j < N; j++)
            sum += a[i][j];
    return sum;
}
```

B

```
int sum_array_cols(int a[M][N])
{
    int i, j, sum = 0;

    for (j = 0; j < N; j++)
        for (i = 0; i < M; i++)
            sum += a[i][j];
    return sum;
}
```

Kursusgang 8+10: Hukommelsessystemet 1-2

Cache systemer

- Hukommelsestyper
- Primær og sekundære hukommelser
- Lokalitet
- Hukommelseshierarkier
- CPU Cache
 - Direkte koblet
 - Associativ koblet
- Cache-venlig kode

Virtuel hukommelse

- Fysiske og virtuelle adresser
- Adresserum
- VM til caching
- VM til lager administration
- VM til beskyttelse
- Adresse-oversættelse
- Side-fejl

- **Hukommelsen er lige så vigtig som processoren for yde-evnen!**
 - Tricks til at få hurtig afvikling af program ved kendskab til caching
- **Vigtige datalogiske principper**
 - Lokalitet og hukommelseshierarkier
 - Caching strategi
 - Lave gode abstraktioner med høj effektivitet!



Vigtige læringsmål for dagens kursusgang

CH9

- Forstå principper for opbygning af forskellige hukommelses-typer

PP6.5

- Forstå forhold omkring adgangstider
- Primær og sekundær hukommelse

CH10

- Hukommelses-hierarkiet
- Forstå lokalitetsprincippet
 - Spatial og temporal lokalitet
- Opbygning af CPU L1, L2 caches

PP6.12

PP6.13

- Direct mapped
- Flervejs associative
- Håndtering af Hit, miss,
- Beregning af adgangstid

PP6.17

PP6.7

- Genkende (og skrive) programmer med god/dårlig lokalitet

- Mindre intensivt

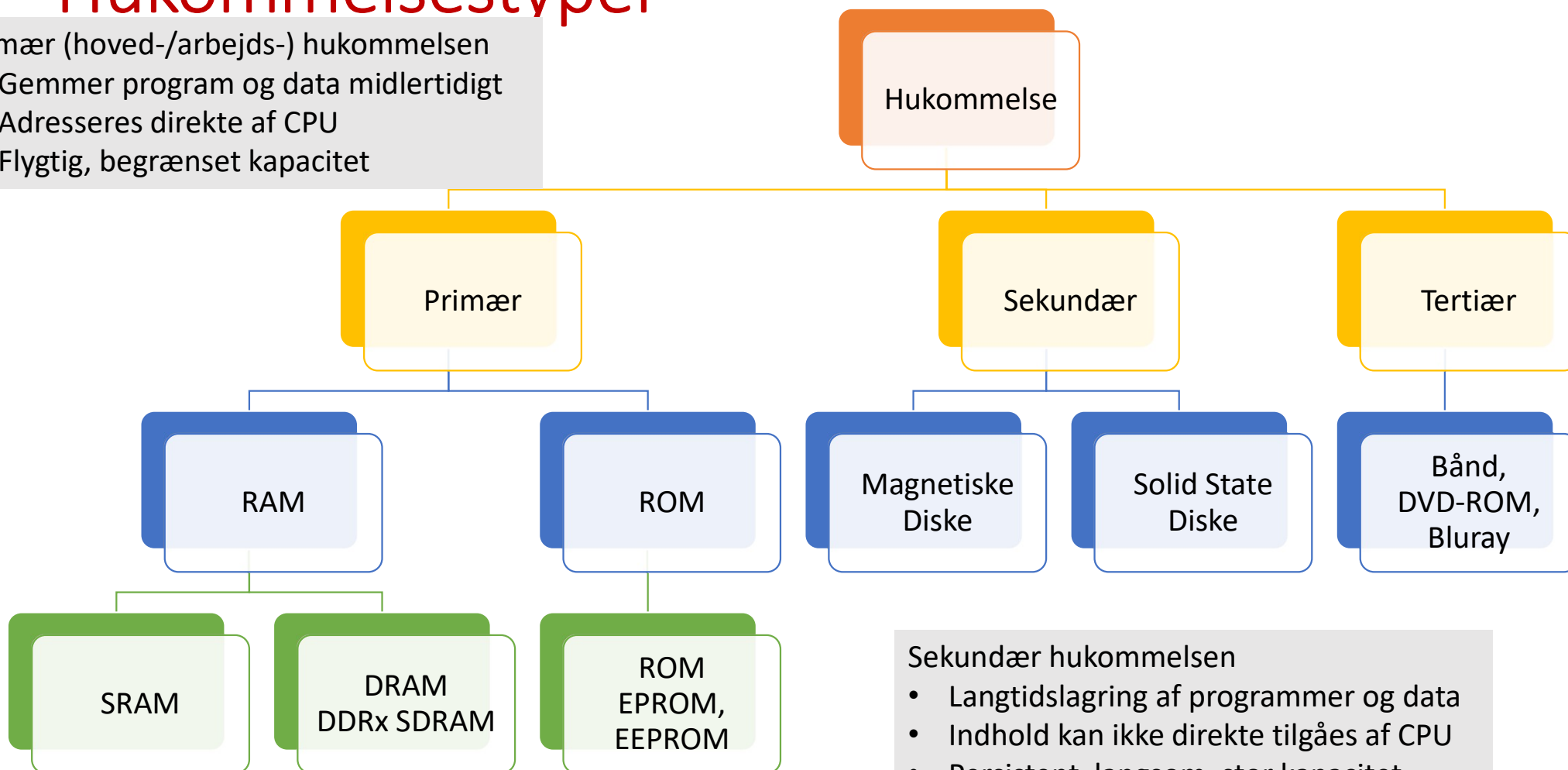
- DRAM opbygning, adgang og forbedringer.
- Opbygning af disk, flash, og SSD
- I/O
- I7 Cache systemet

Hukommelsestyper

Hukommelsestyper

Primær (hoved-/arbejds-) hukommelsen

- Gemmer program og data midlertidigt
- Adresseres direkte af CPU
- Flygtig, begrænset kapacitet

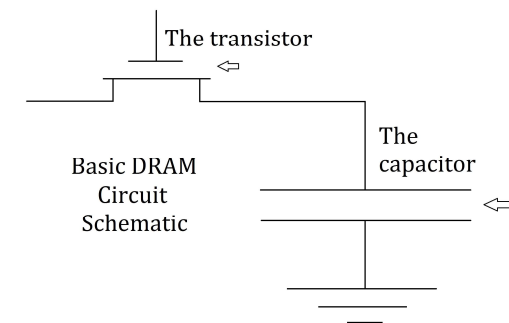


Sekundær hukommelsen

- Langtidslagring af programmer og data
- Indhold kan ikke direkte tilgås af CPU
- Persistent, langsom, stor kapacitet

Random-Access Memory (RAM)

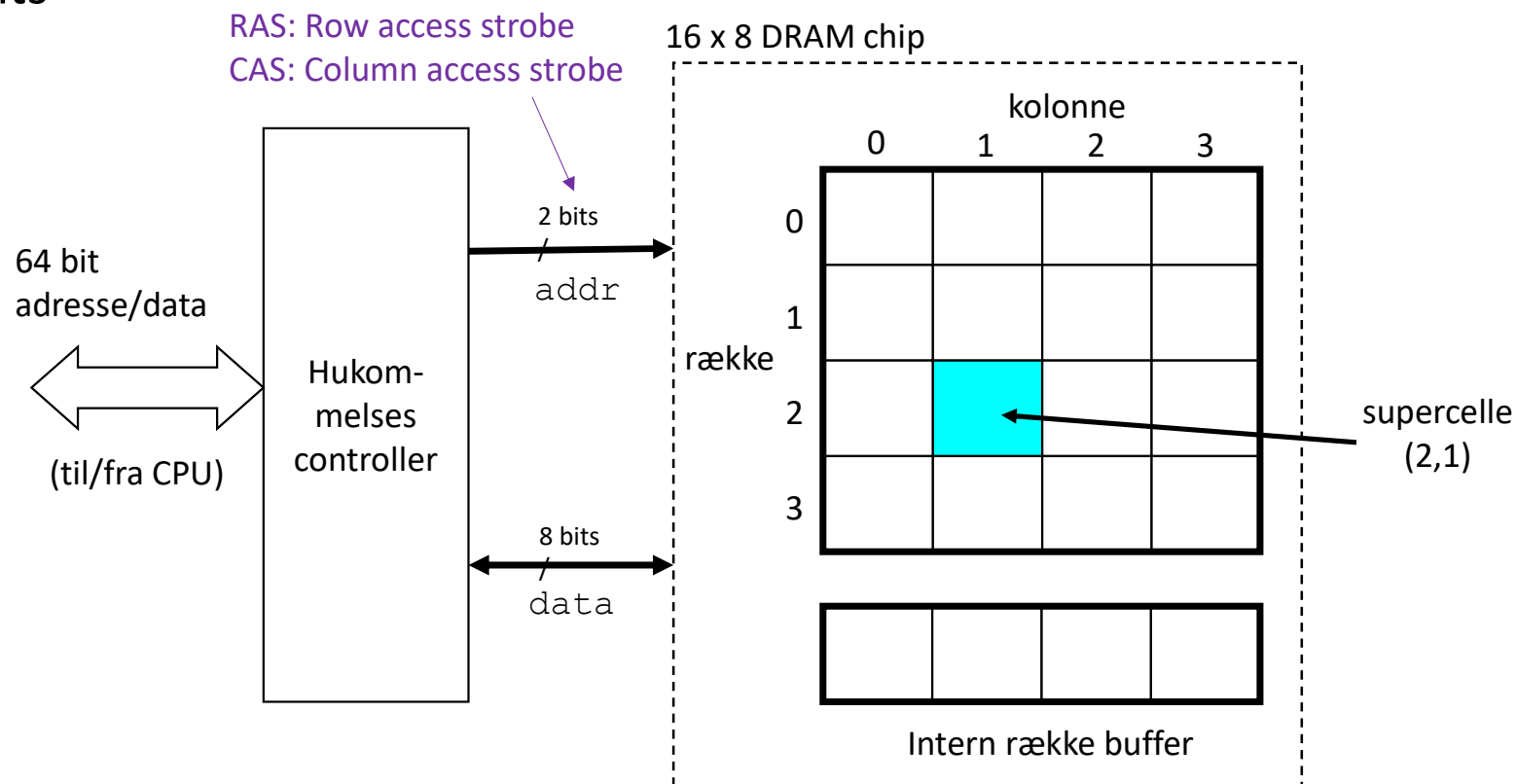
- Nøgle egenskaber
 - Flygtig, læse/skrive specifik byte, angivet med en "adresse"
 - **RAM** "indpakket" i chip.
 - Indeholder et antal **celler** (gemmer en bit per celle).
 - **Primær (hoved-)hukommelsen** består af flere RAM chips sat sammen i moduler
- **SRAM (Statisk RAM)**
 - Hver celle bruger 4-eller 6 transistorer (bi-stabile element)
 - Bevarer bit værdien så længe der er strøm
 - Relativt ufølsomt for elektrisk støj (EMI), stråling, etc
 - **Hurtigere og dyrere** end DRAM
- **DRAM (Dynamic RAM)**
 - Hver celle bruger en kondensator (capacitor) til at gemme en ladning og en transistor til adgang
 - Værdi skal genopfriskes (refreshed) hver 10-100 ms (hukommelses controller foretager en læsning og ny skrivning)
 - Mere følsom for forstyrrelse (EMI, stråling, etc) end SRAM
 - **Langsommere og billigere** end SRAM



https://en.wikipedia.org/wiki/Dynamic_random_access_memory

Konventionel DRAM Struktur

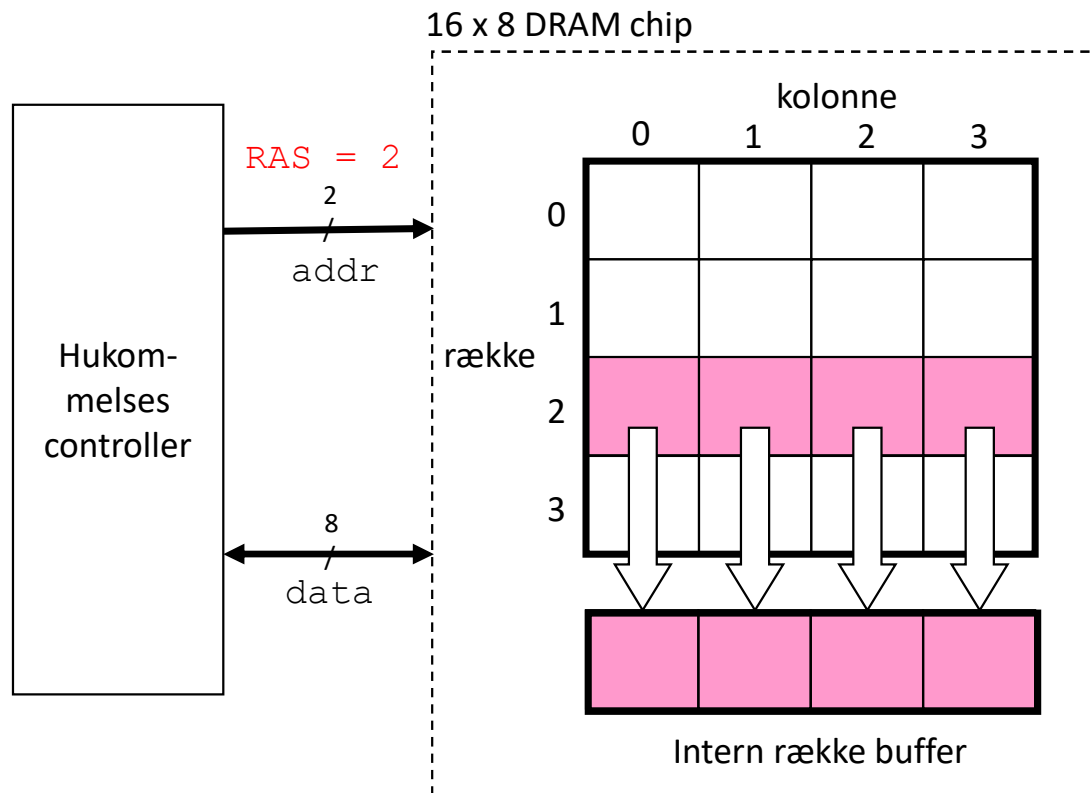
- $d \times w$ DRAM:
 - dw bits totalt: organiseret som 2D array af d **superceller** hver af størrelsen w bits



Aflæsning af DRAM Supercelle (2,1)

Skridt 1(a): Udsend "Row access strobe (**RAS**)" udvælger række 2.

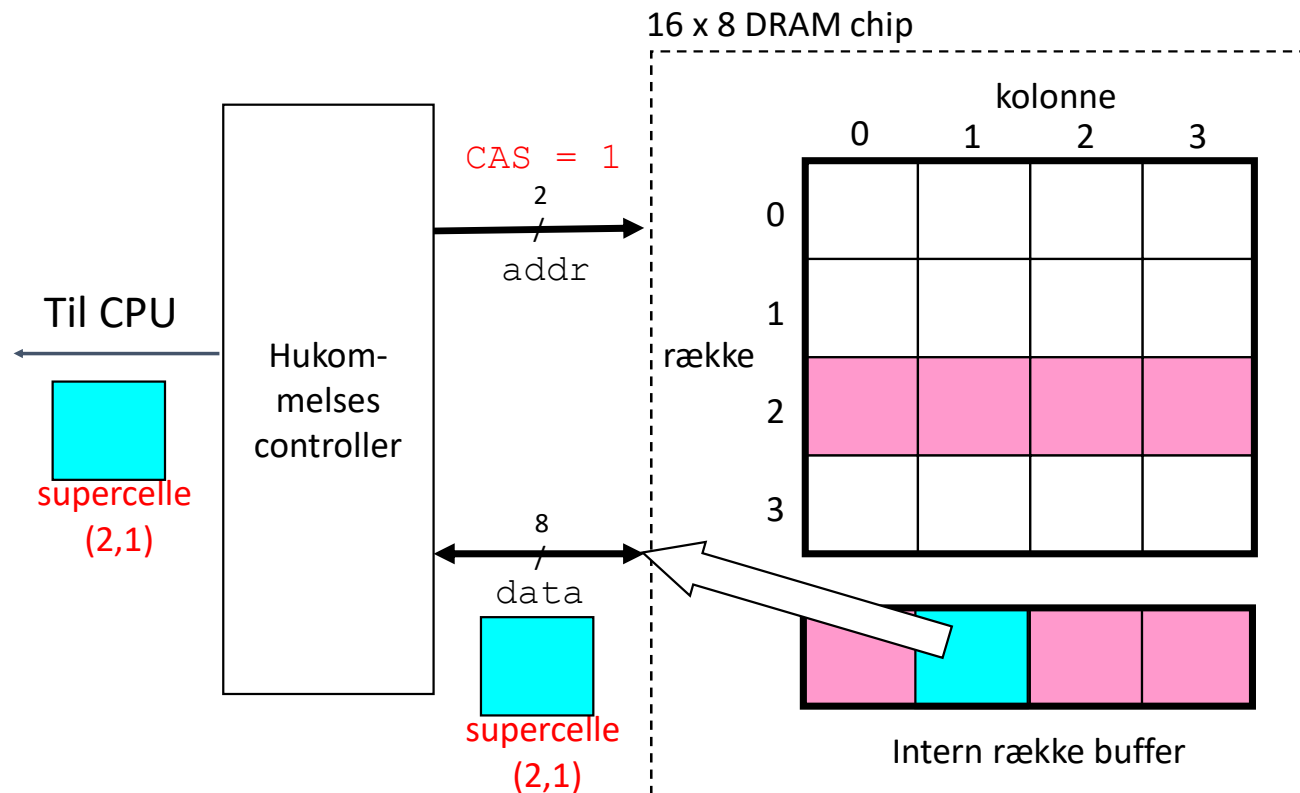
Skridt 1(b): Række 2 kopieres fra DRAM array til række-buffer.



Aflæsning af DRAM Supercelle (2,1)

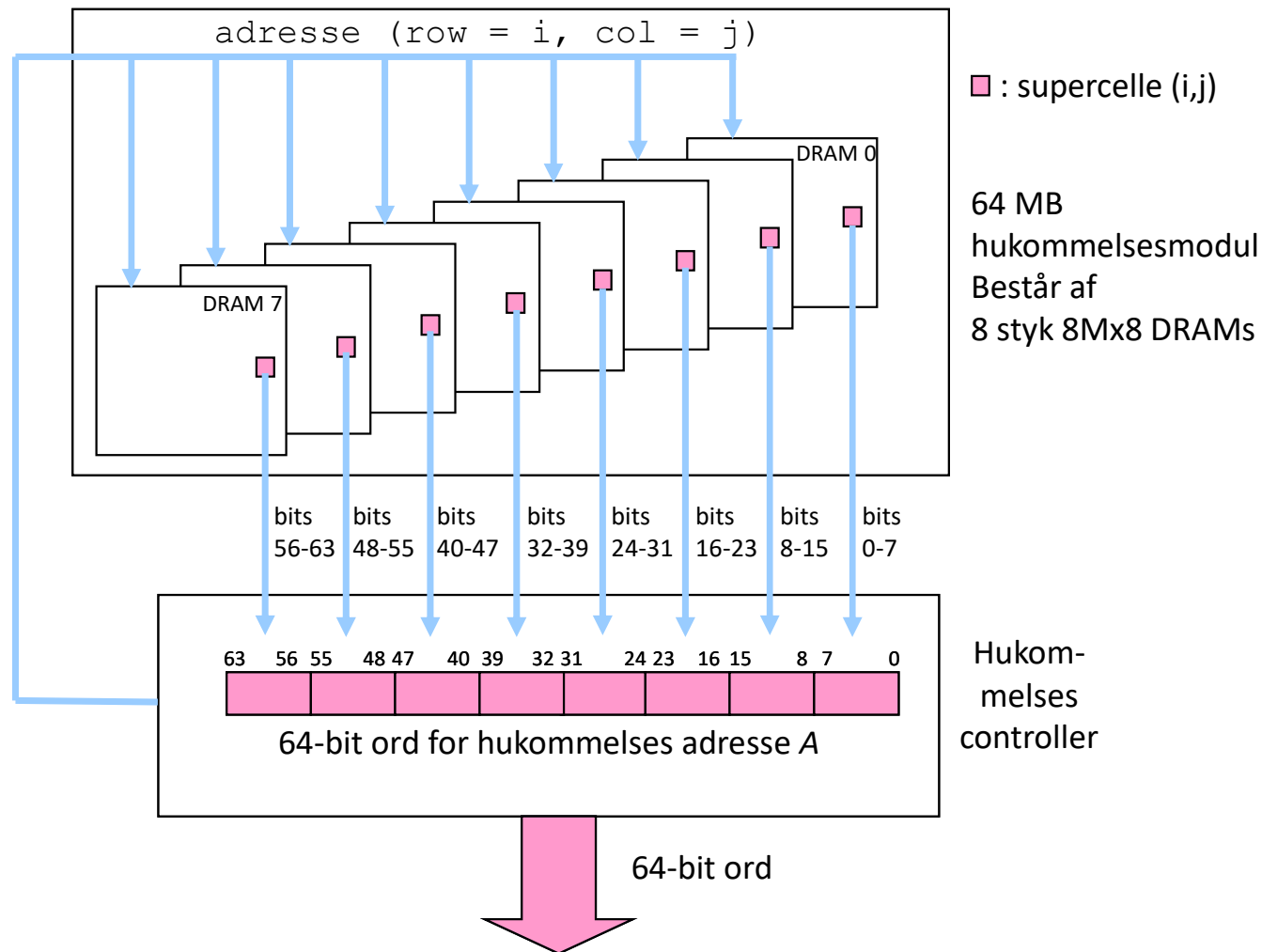
Skridt 2(a): Udsend "Column access strobe (CAS)": vælger kolonne 1.

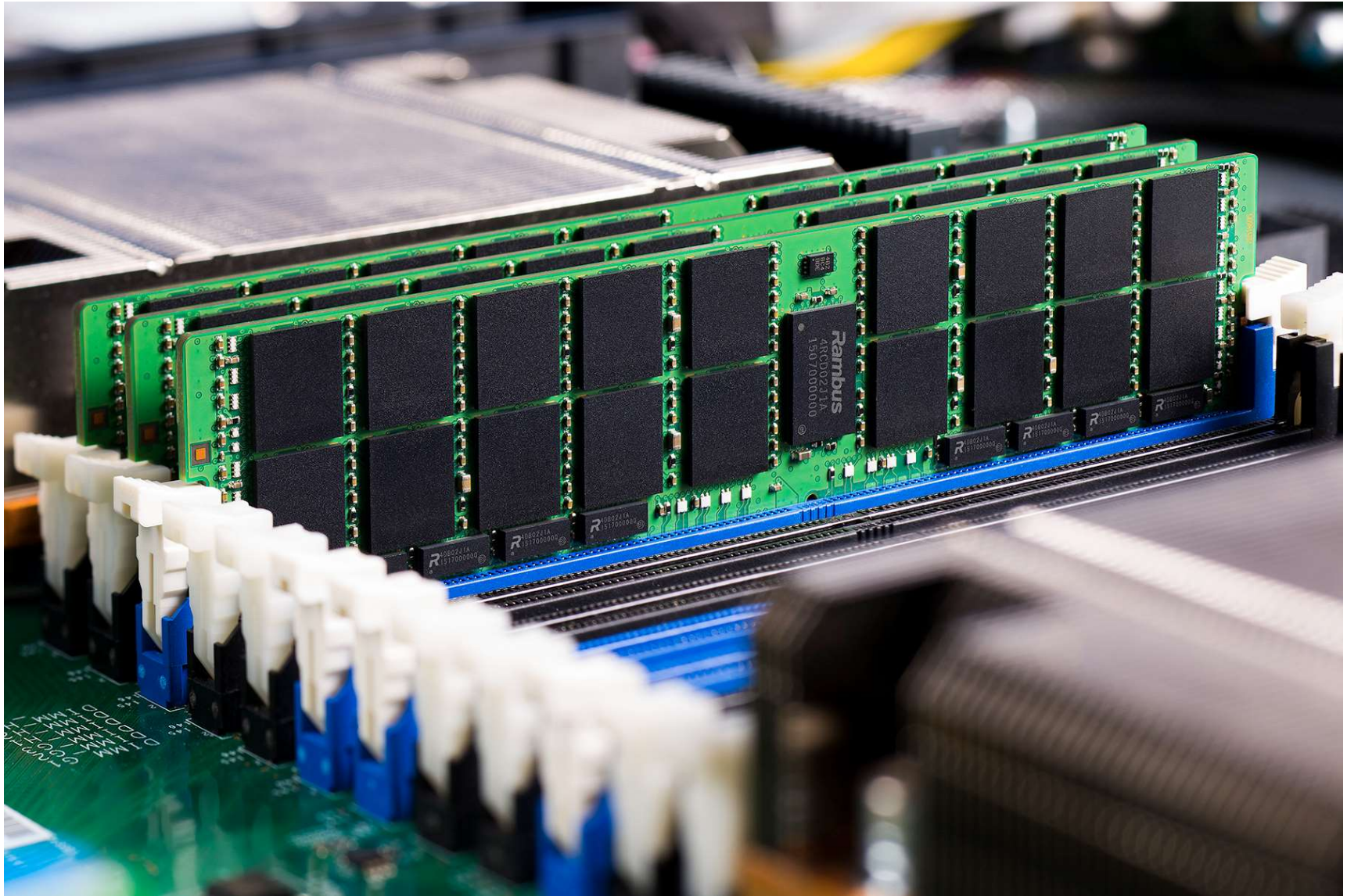
Skridt 2(b): Supercelle (2,1) kopieres fra buffer til data linien, og derefter videre mod CPU.



Optimeret DRAM:
De resterende celler
gemmes, skulle der
ske læsning til same
række.

Hukommelses Moduler



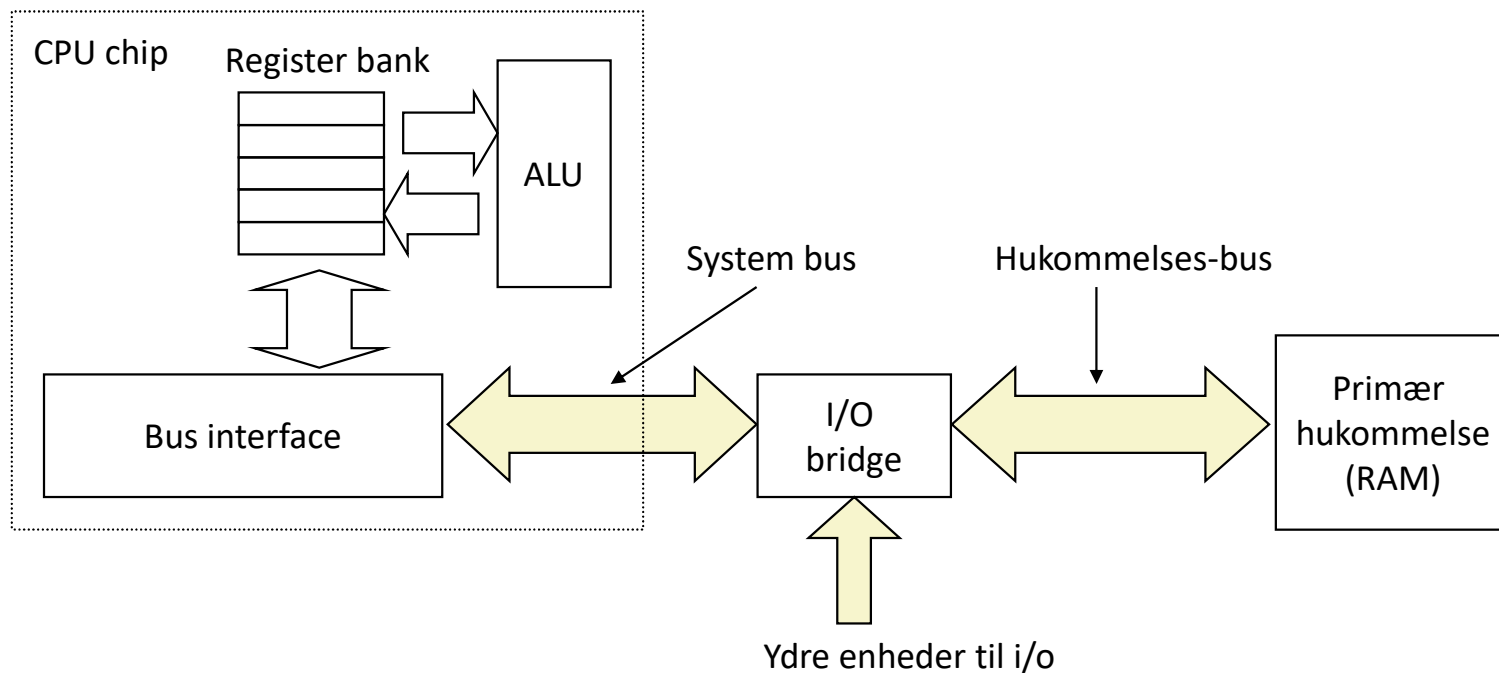


Forbedrede DRAMs

- Basale DRAM celle samme som da den blev opfundet i 1966.
 - Kommercialiseret af Intel i 1970.
- Nye DRAM moduler har bedre interface logik og hurtigere I/O :
 - Synkron DRAM (**SDRAM**)
 - Bruger et konventionelt clock signal i stedet for asynkron styring
 - Tillader genbrug af række-adresse (e.g., RAS, CAS, CAS, CAS)
 - Double data-rate synkron DRAM (**DDR SDRAM**)
 - Dobbelt udlæsning (fx stigende og faldende kant) sender 2 ord per cyklus per ben
 - Forskellige typer afhængigt af størrelsen på en lille buffer til **forudlæsning** ("prefetch buffer")
 - **DDR** (2 ord/adgang), **DDR2** (4 ord/adgang), **DDR3** (8), **DDR4** (8)
 - **Nu DDR5** SDRAM (højere båndbredde, mindre spænding/energi forbrug)

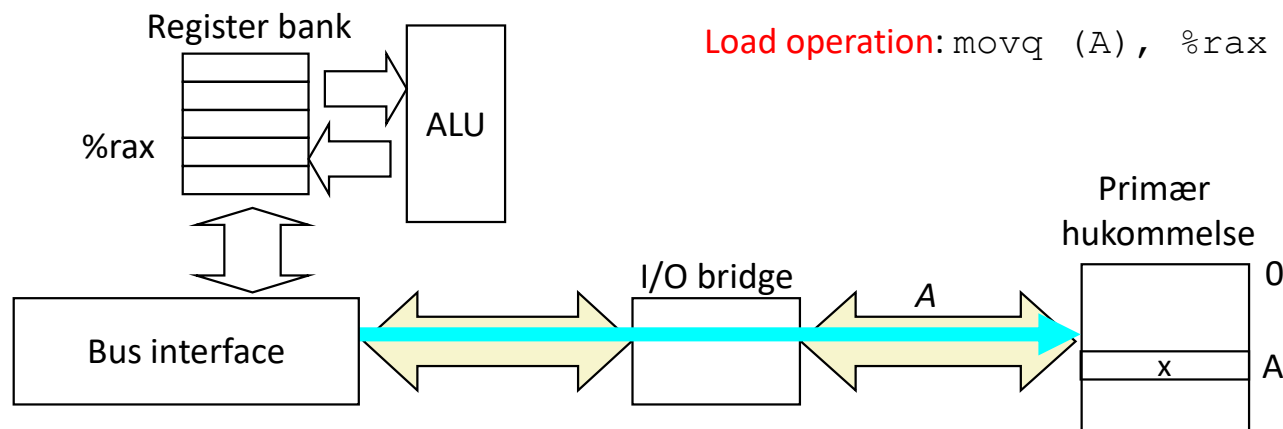
Traditionel Bus Struktur til forbindelse af CPU og hukommelse

- En **bus** er en samling parallelle elektriske forbindelser som overfører adresse, data og kontrol signaler.
- Busser er typisk delte mellem flere enheder.



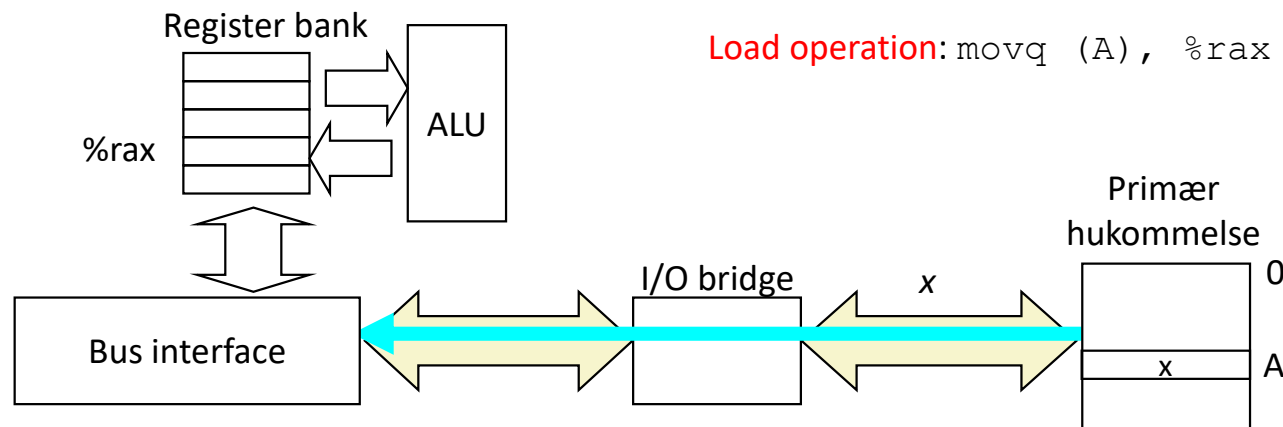
Læse transaktion fra hukommelsen (1)

- CPU placerer adresse A på hukommelsesbussen.
- Kontrol signal angiver "læsning"



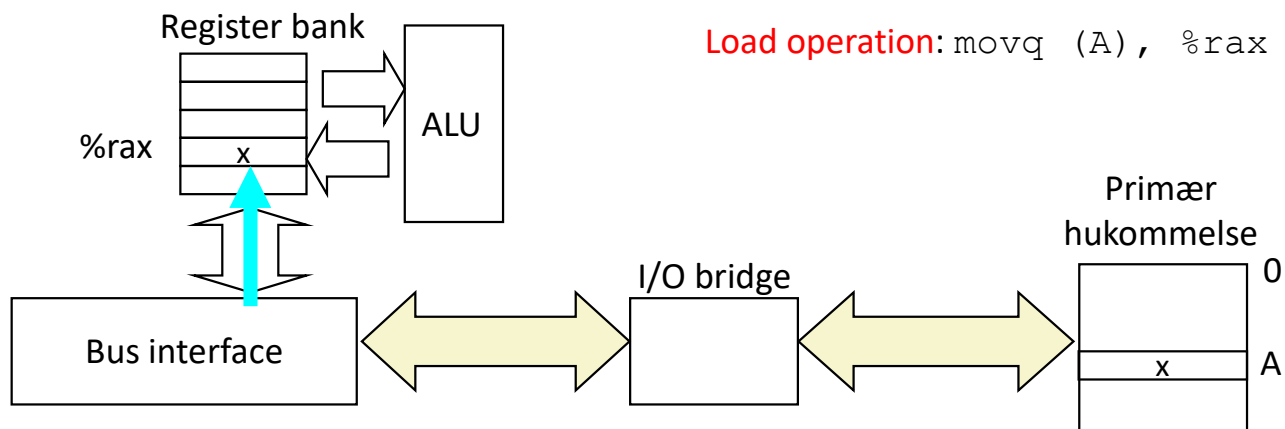
Læse transaktion fra hukommelsen (2)

- Primær hukommelsen læser adressen A fra hukommelses-bussen, udlæser det angivne ord x, og placerer det (efter nogen tid) på bussen.



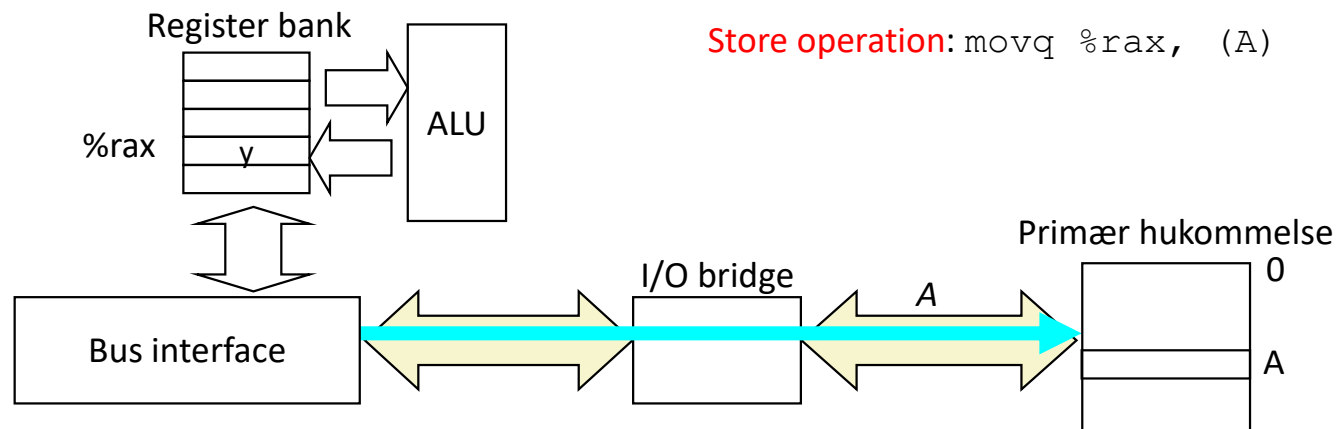
Læse transaktion fra hukommelsen (3)

- CPU læser ord x fra busen og kopierer det ind i register %rax.



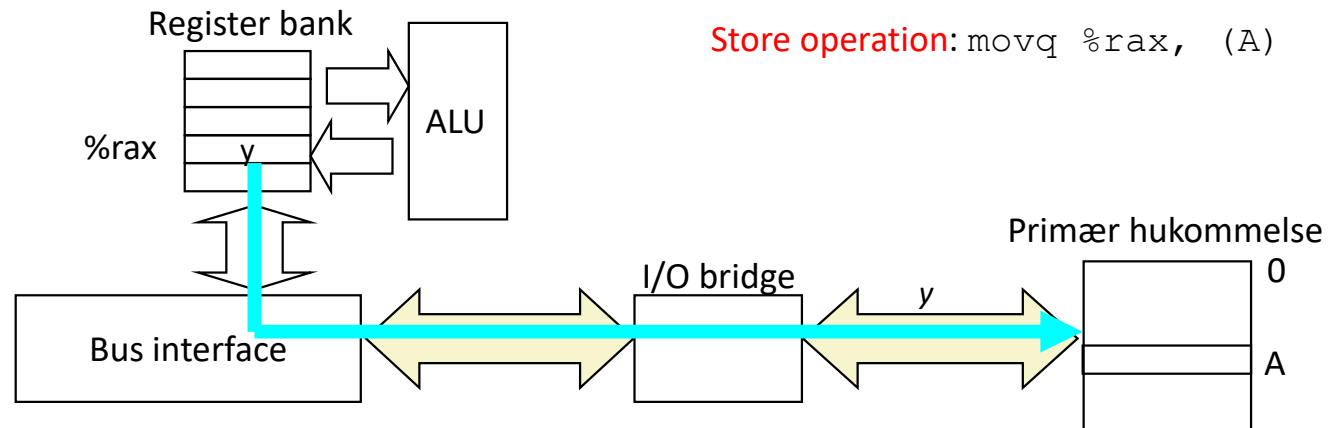
Skrive transaktion til hukommelsen (1)

- CPU placerer adresse A på bussen, og angiver med kontrol signal at der skal skrives
- Primær hukommelsen læser det, og afventer data ord, som skal skrives.



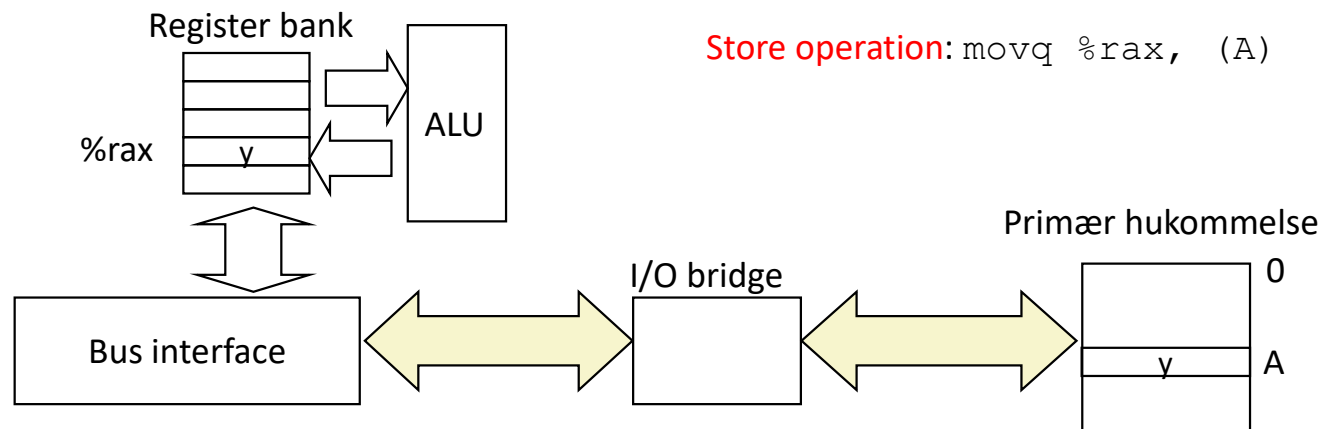
Skrive transaktion til hukommelsen (2)

- CPU placerer data ord y på bussen.



Skrive transaktion til hukommelsen (3)

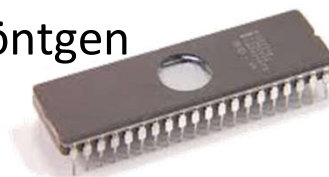
- Primær hukommelsen læser ord y fra bussen, og gemmer det i adresse A.



Sekundær lagring (disks+ssd)

Persistent Hukommelse

- DRAM og SRAM er flygtig (volatile): Taber information når strømmen går.
- Persistent hukommelse bevarer information uden strøm
 - Read-only memory (**ROM**): programmeret når chip er produceret
 - Programmérbar ROM (**PROM**): kan programmeres/skrives én gang vha særligt apparat
 - Sletbar (Erasable) PROM (**EPROM**): kan slettes/genbruges vha. særlig apparat (UV, röntgen stråler)
 - Elektrisk sletbar EPROM (**EEPROM**): elektronisk slette mulighed
 - **Flash-hukommelse**: variant af EEPROMs. Sletning/genbrug af dele ("blok-niveau")



• Sekundær hukommelse



- Harddisk (Magnetisk), Solid State Disk (SSD)
- SD-kort, "USB sticks", (Flash-hukommelse som medie)...
- Anden brug af "hukommelser"
 - Firmware programmer gemmes i ROM (BIOS, disk controllere, netværkskort, grafik acceleratorer, sikkerheds-(del)systemer,...)
 - Flash/EEPROM bruges oftere nu, da firmware bliver mere og mere software tung
 - Konfigurerings-data (EEPROM)

Indmaden i et (magnetisk) harddisk drev!

- Typiske data:
 - Rotationshastighed = 7,200 RPM (8ms/rotation)
 - Gns. søge (seek) time = 9 ms.
 - Gns. # sektorer/spor = 400.
- Udledt:
 - $T_{\text{adgang}} = T_{\text{søg}} + T_{\text{rot}} + T_{\text{læs}}$
 - $T_{\text{adgang}} = 9 \text{ ms} + 4 \text{ ms} + 0.02 \text{ ms}$
- Adgangs-tid domineres af søge-tid og rotationsforsinkelse.
- SRAM adgangs-tid 4 ns/ord,
- DRAM ca. 60 ns
 - Disk er ca. 40000 gange langsommere end SRAM,
 - 2500 gange langsommere DRAM.

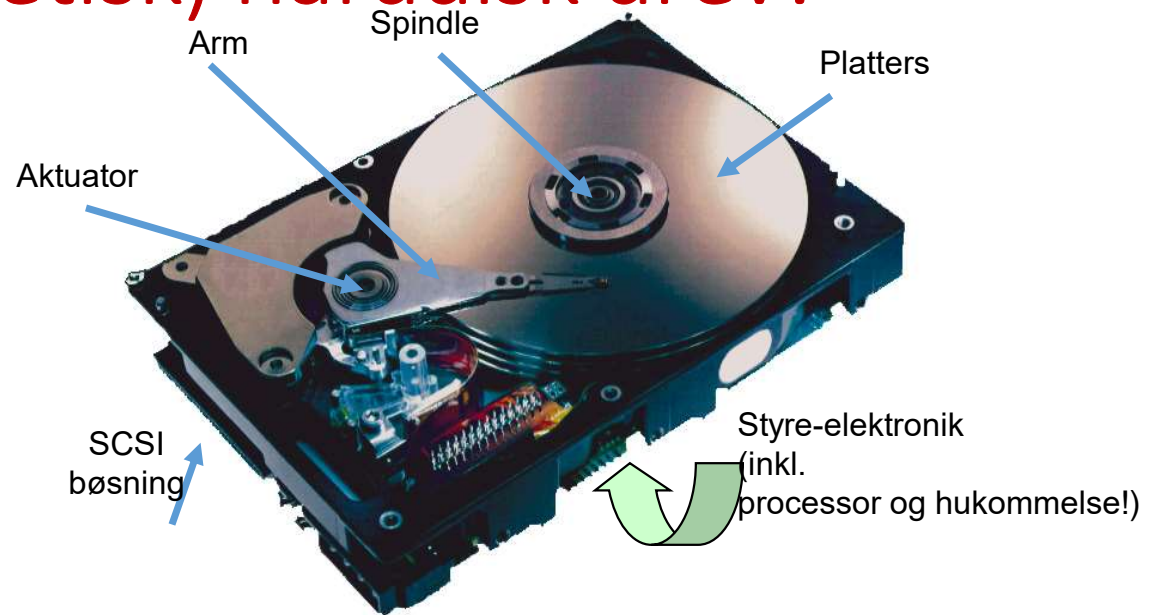
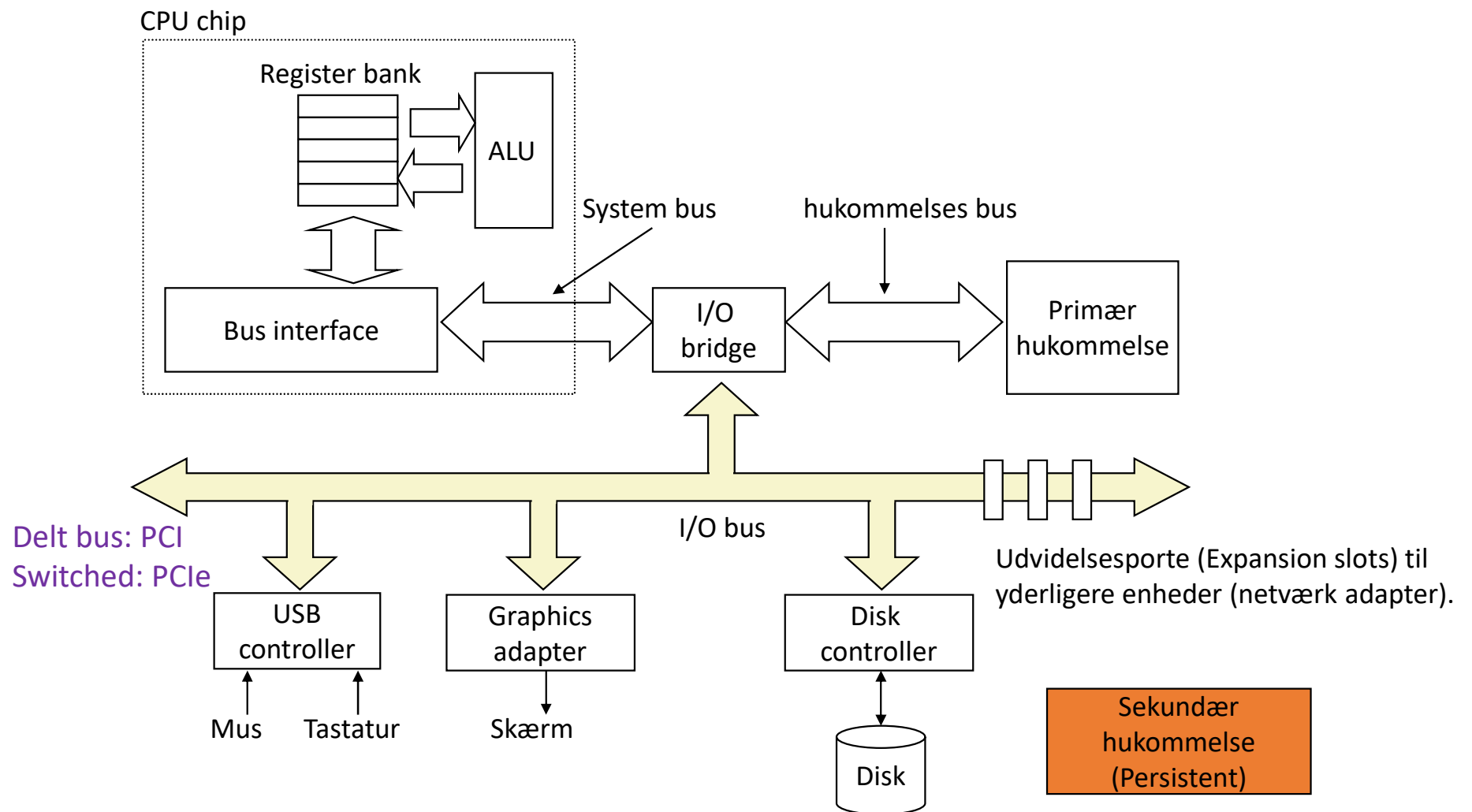


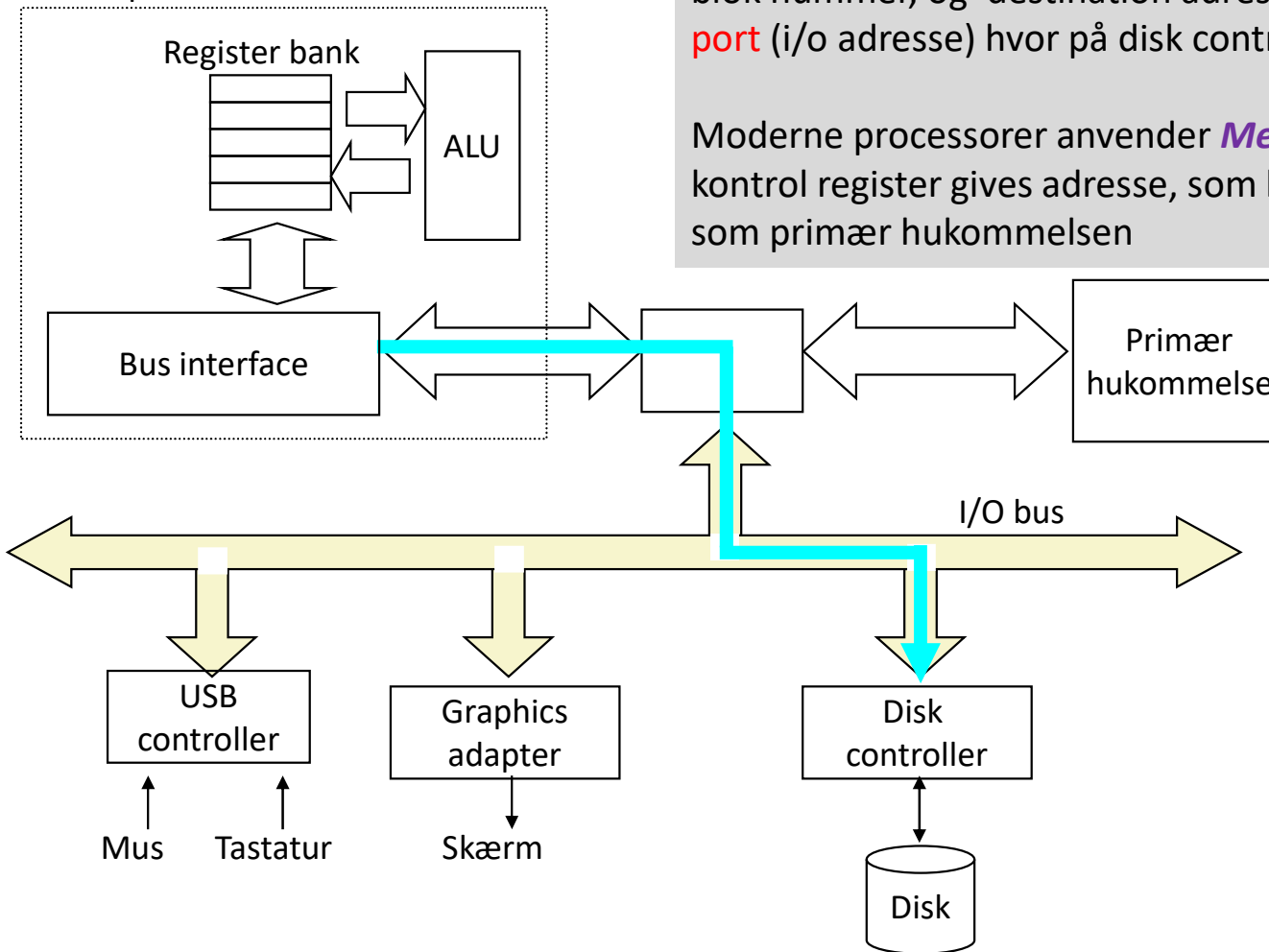
Image courtesy of Seagate Technology

I/O Bus



Læsning af Disk Sektor (1)

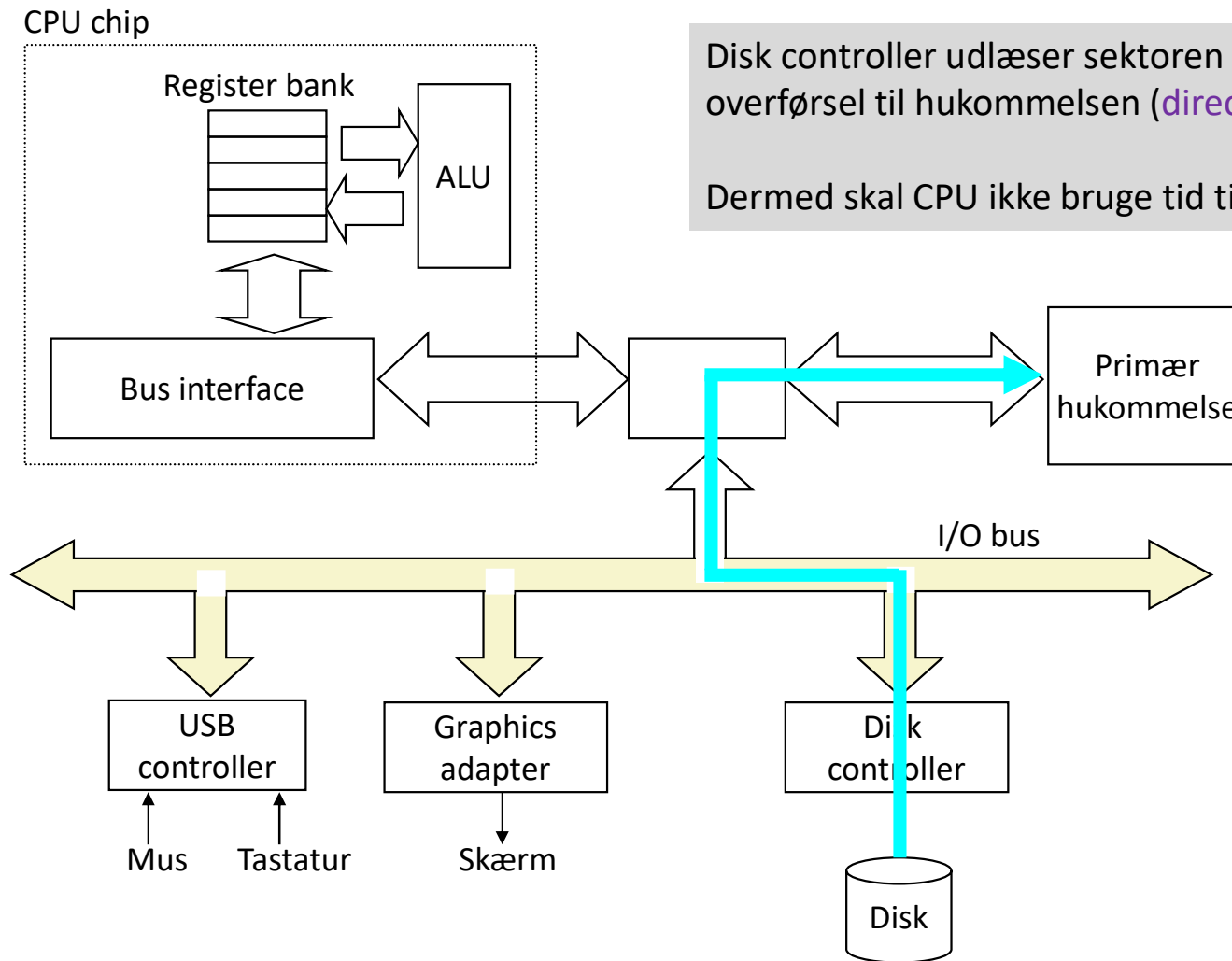
CPU chip



CPU starter en disk-læsning ved at skrive kommandoen, logisk blok nummer, og destination adresse (i hukommelsen) til en **port** (i/o adresse) hvor på disk controller er forbundet

Moderne processorer anvender *Memory Mapped I/O*: enhedens kontrol register gives adresse, som kan tilgås på samme måde som primær hukommelsen

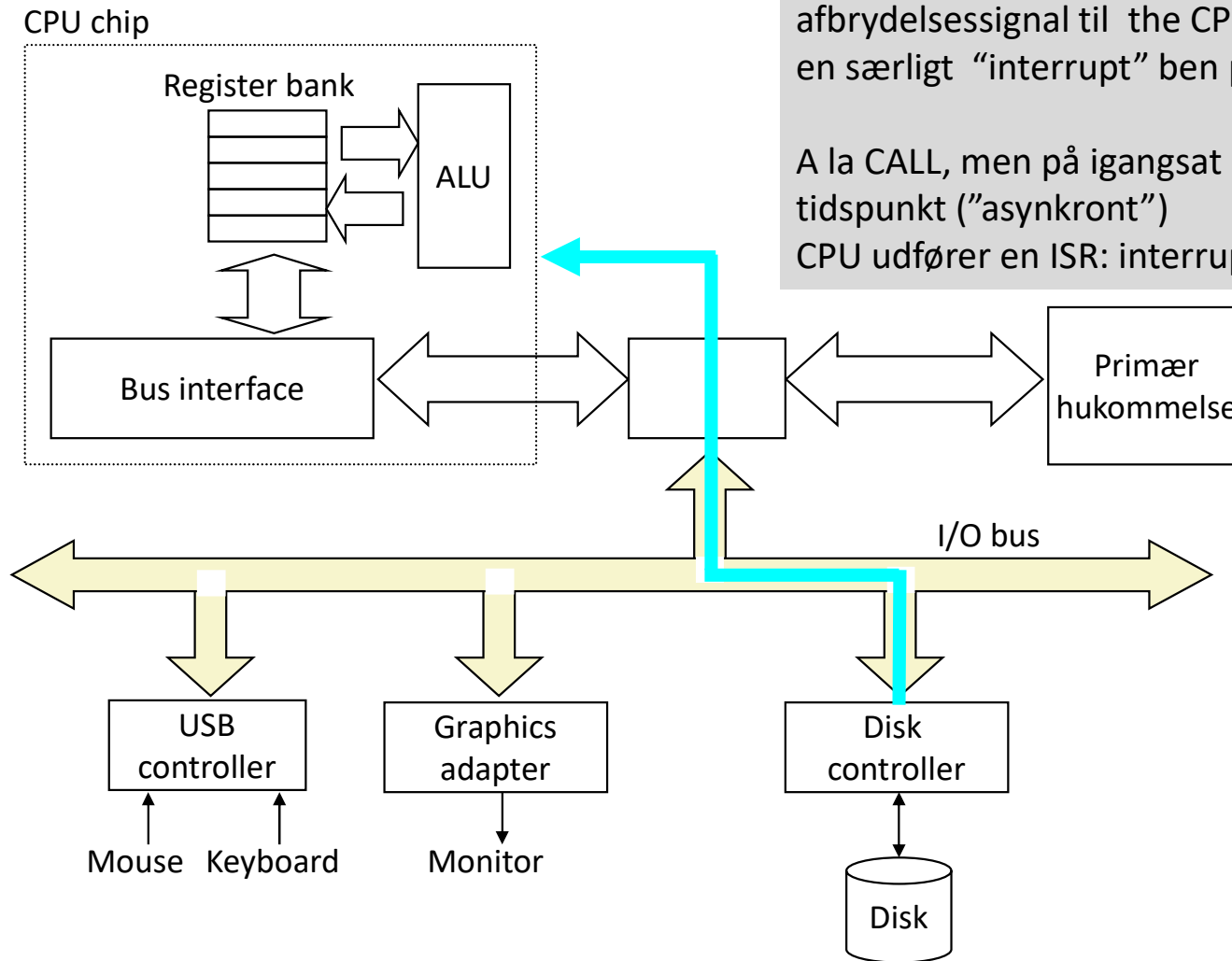
Læsning af Disk Sektor (2)



Disk controller udlæser sektoren og udfører en direkte overførsel til hukommelsen (**direct memory access, DMA**).

Dermed skal CPU ikke bruge tid til at overføre dataene.

Læsning af Disk Sektor (3)



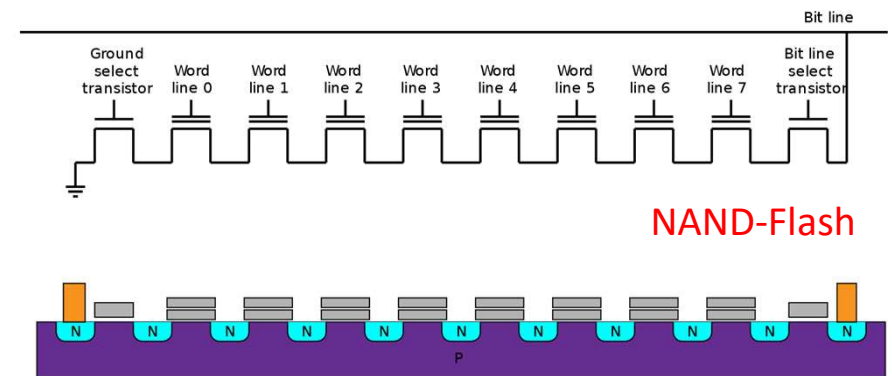
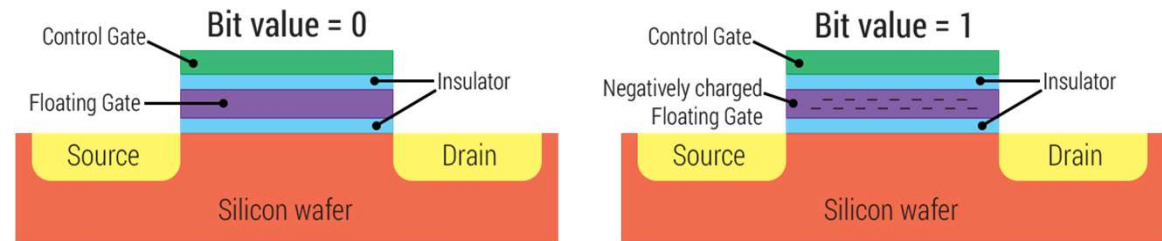
Når DMA overførelsen er færdig, sender disk controller et afbrydelsessignal til the CPU with an *interrupt* (i.e., sætter en særligt "interrupt" ben på CPUen)

A la CALL, men på igangsæt af HW, og på et uforudsigeligt tidspunkt ("asynkront")
CPU udfører en ISR: interrupt service routine

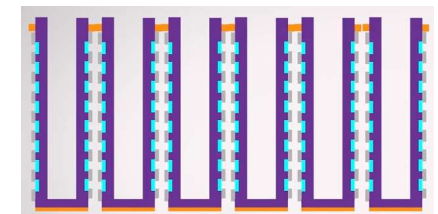
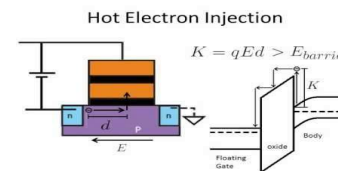
13 ms til læsning af disk
=13M instruktioner (ved
fx 1 ns clock-cyklus)

Flash Hukommelse

- **Flash-hukommelse:** variant af EEPROMs.”
 - Floating gate transistor
 - Pakkes i ”strenger”
 - Samles enheder ”sider” fx 4kB
 - En hel side skal slettes før der kan skrives (selv en enkelt bit)
 - Kan slettes (relativt) hurtigt ”in a flash”
 - Side slides op efter 100.000 skrivninger
- Lagrings-teknologien bruges i
 - Solid State Disk (SSD)
 - SD-kort, USB-sticks, mp3 afspillere,



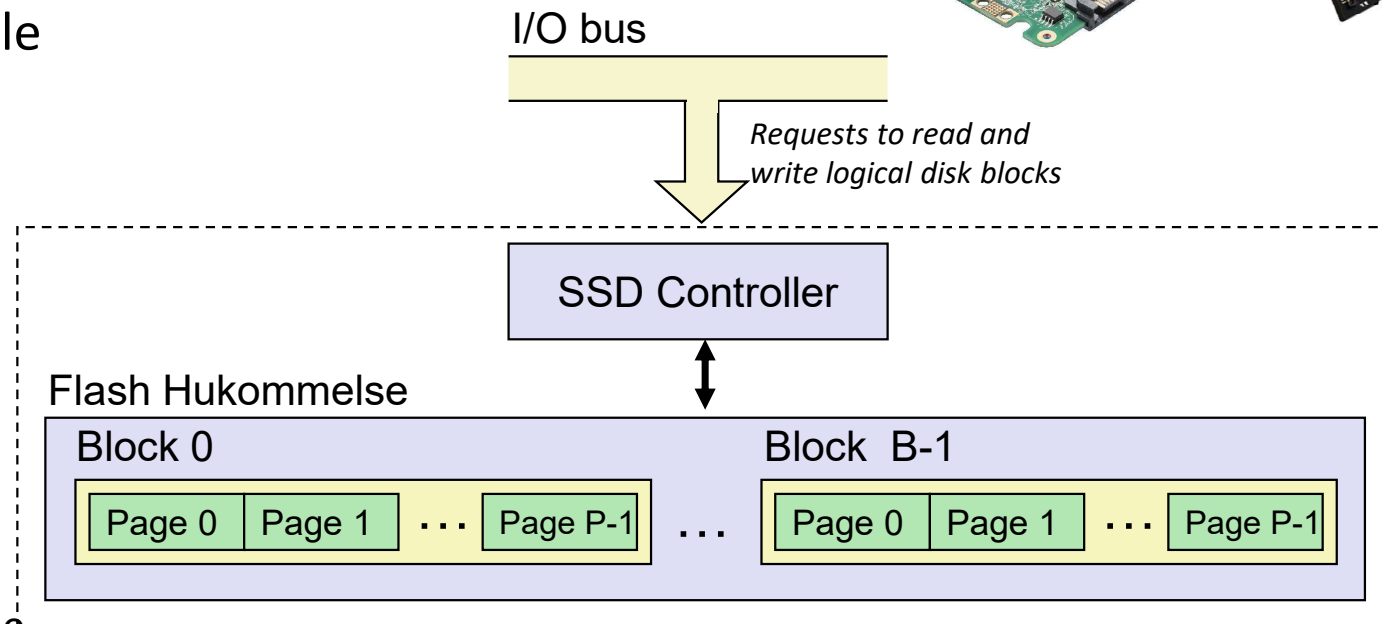
Pakket i 3D øger
lagringstætheden



<https://www.youtube.com/watch?v=s7JLXs5es7I>

Solid State Disks (SSDs)

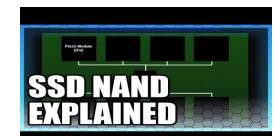
- Nyere alternative til magnetiske diske, uden bevægelige dele



- Bruger Flash-hukommelse
- Opdeles i sider (pages): ~4KB, og Blokkes: ~ 128 sider
- Data læses/skrives i enheder af sider (En Side kan kun skrives efter den er blevet slettet)
- **Avanceret kontrol logik**
 - Hurtig læsning og skrivning
 - Balancering af slitage af flash



<https://www.youtube.com/watch?v=-XZNr7mS0iw&t=485s>



<https://www.youtube.com/watch?v=w3uwpuNY-Ww>

SSD Performance Karakteristika

Kilde: Intel SSD 730 produkt specifikation (2014).

Sekventiel læserate	550 MB/s	Sekventiel skriverate	470 MB/s
Tilfældig læserate	365 MB/s	Tilfældig skriverate	303 MB/s
Gns. Læsetid af blok	50 μ s	Gns. Skrivetid	60 μ s

- Sekventiel læsning hurtigere en tilfældig adgang
 - I øvrigt en almindelig diskussion i hukommelses-systemer
- Tilfældig skrivning er en del langsommere
 - Sletning af en blok tager lang tid (~ 1 ms)
 - Ændring af en side i en blok, kræver at alle andre sider skrives til en ny blok
 - I tidlige SSDer var gabet mellem læsning og skrivning meget større.

Samsung 960 EVO NVMe
1.2 (Non-Volatile Memory Express) grænseflade
Datahastighed: 3200 MBps (læs) / 1500 MBps (skriv),
512 MB LPDDR3 cache,
IOPS: 330000 (læs) / 300000 (skriv),

Ydelse under realistiske brugsscenarier?

SSD vs. Magnetisk (roterende) Diske

- Fordele
 - Ingen bevægelige dele → hurtigere, mindre energi, mere robust
- Ulemper
 - Kan slides op
 - Håndteres af logik til udjævning af slitage (“wear leveling logic”) in flash controlleren
 - Intel SSD 730 garanterer at der kan skrives 128 petabyte (128×10^{15} bytes) før den er udtjent
 - Jo mindre ledig plads, des mindre levetid
 - I 2015, ca. 30 gange så dyrt per byte
- SSD Anvendelser
 - Almindeligt i dyrere bærbare
 - Desktop maskiner: både ssd (især boot disk) og mag. disk
 - Servere, som supplement til mag. disk, til data som kræver særlig hurtig adgang
- Magnetiske diske stadig ”arbejdshesten” til lagring af store data-mængder

Hukommelsesgabet

Lagrings tendenser

SRAM

Metrik	1985	1990	1995	2000	2005	2010	2015	v2015:1985
\$/MB	2,900	320	256	100	75	60	25	116
adgangstid (ns)	150	35	15	3	2	1.5	1.3	115

DRAM

Metrik	1985	1990	1995	2000	2005	2010	2015	v2015:1985
\$/MB	880	100	30	1	0.1	0.06	0.02	44,000
adgangstid (ns)	200	100	70	60	50	40	20	10
typ.størrelse(MB)	0.256	4	16	64	2,000	8,000	16,000	62,500

Mag. Disk

Metrik	1985	1990	1995	2000	2005	2010	2015	v2015:1985
\$/GB	100,000	8,000	300	10	5	0.3	0.03	3,333,333
adgangstid(ms)	75	28	10	8	5	3	3	25
typ.størrelse(GB)	0.01	0.16	1	20	160	1,500	3,000	300,000

CPU Clock Frekvens

Vendepunkt i computerens historie
da designere ramte effekt-muren "Power Wall"

	1985	1990	1995	2003	2005	2010	2015	2015:1985
CPU	80286	80386	Pentium	P-4	Core 2	Core i7(n)	Core i7(h)	
Clock Frekv. (MHz)	6	20	150	3,300	2,000	2,500	3,000	500
Cyklus tid (ns)	166	50	6	0.30	0.50	0.4	0.33	500
Kerner	1	1	1	1	2	4	4	4
Effektiv cyklus tid (ns)	166	50	6	0.30	0.25	0.10	0.08	2,075

Moore's lov:
Antallet af
transistorer i et
integreret
kredsløb
fordobles ca hver
18 md.

ECT= cyklustid / #Cores

(n) Nehalem processor
(h) Haswell processor

Varme



Energi forbruges ikke, men omsættes fra et type til en anden: fra elektrisk energi til varme energi

$P(t)$: Effekt "Power" til tid t .
Energioomsætning per tidsenhed
(1 Watt = 1 Joule/1 sec))

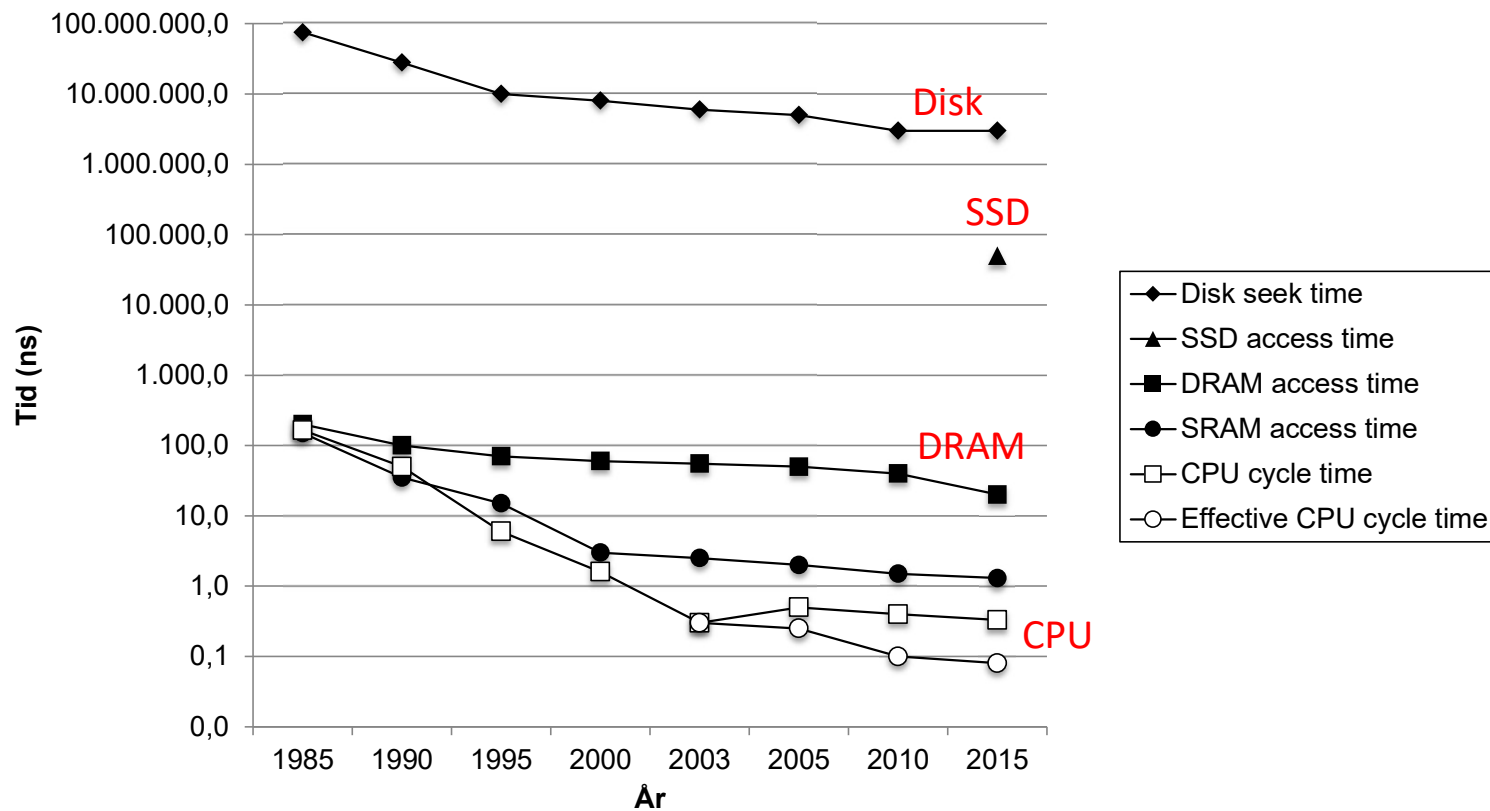
E : energi forbrug (Joule)

http://www.phys.ncku.edu.tw/~htsu/humor/fry_egg.html

Hastigheds-Gabet mellem CPU og hukommelse

Gabet øges mellem DRAM, disk, og CPU hastighed.

NB!
LOG
skala



Din redning: Lokaltet!

Nøglen til at formindske gabet imellem CPU og hukommelse er en fundamental egenskab ved de fleste computer programmer: **lokalitet**

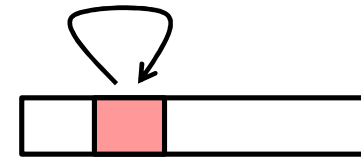
Lokalitet

Lokalitet

- **Lokalitetsprincippet:** Programmer bruger (eller genbruger) ofte data og instruktioner på adresser tæt ved dem den nyligt har brugt

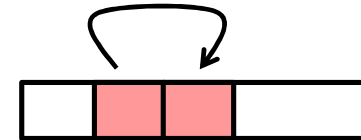
- **Temporal lokalitet**

- Nyligt tilgået data skal sandsynligvis bruges igen i den nærmeste fremtid



- **Spatial (rummelig) lokalitet:**

- Data på nærliggende adresser refereres tit hurtigt efter hinanden



Eksempel på lokalitet

```
sum = 0;
for (i = 0; i < n; i++)
    sum += a[i];
return sum;
```

- Data-referencer

- Tilgår array elementer efter hinanden (reference mønster: skridt længde 1).
- Variablen `sum` tilgås for hver iteration (også `i`, `n`).

Spatial lokalitet

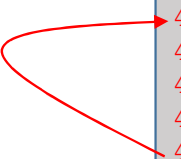
Temporal lokalitet

- Instruktions-referencer

- Udfører instruktioner i sekvens
- Gentager kroppen af løkken.

Spatial lokalitet

Temporal lokalitet

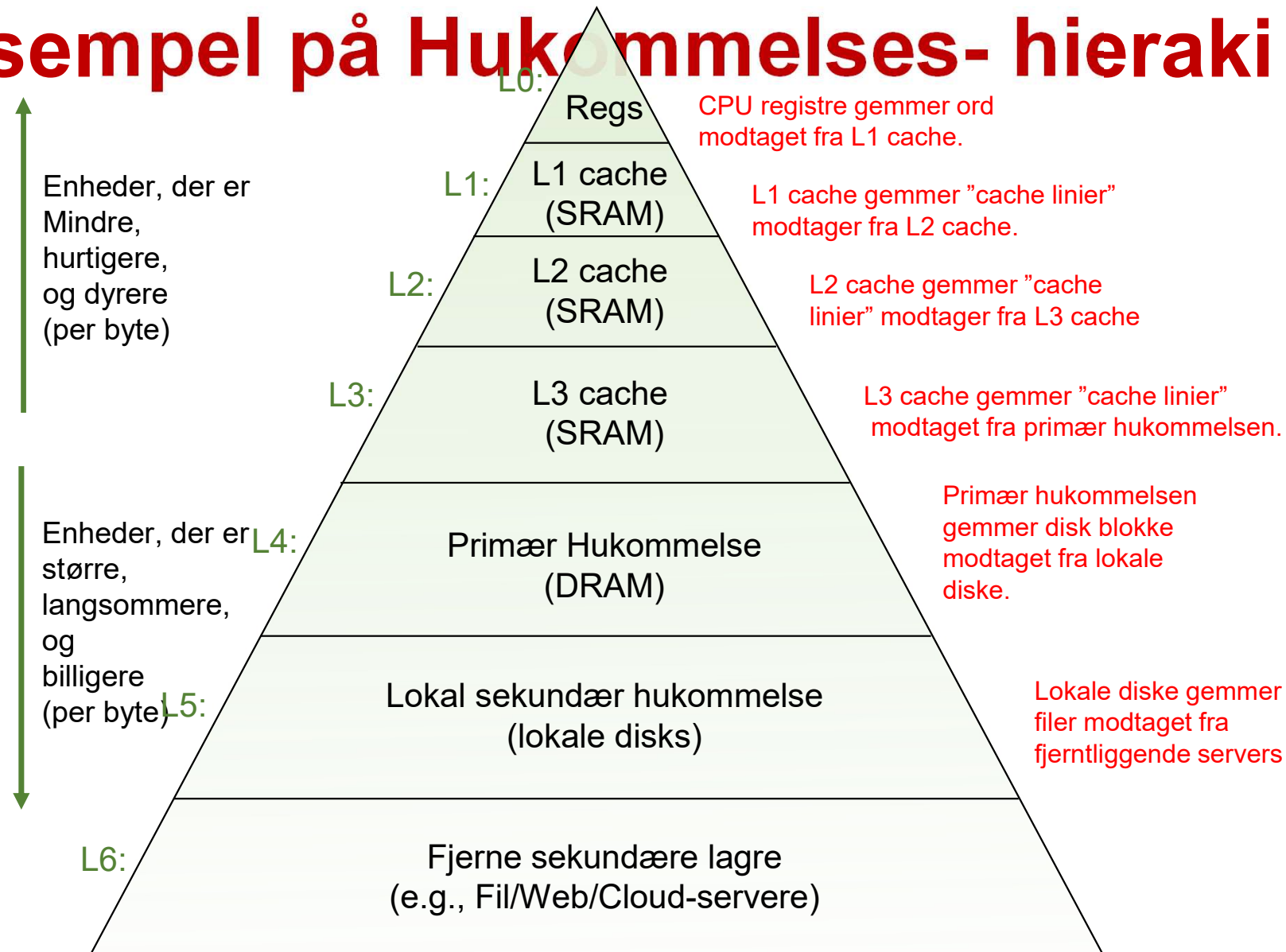


4004b0:	ba 00 00 00 00	mov	\$0x0,%edx
4004b5:	eb 09	jmp	4004c0 <main+0x23>
4004b7:	48 63 f2	movslq	%edx,%rsi
4004ba:	03 0c b4	add	(%rsp,%rsi,4),%ecx
4004bd:	83 c2 01	add	\$0x1,%edx
4004c0:	39 c2	cmp	%eax,%edx
4004c2:	7c f3	jl	4004b7 <main+0x1a>
4004c4:	89 c8	mov	%ecx,%eax
4004c6:	48 83 c4 30	add	\$0x30,%rsp
4004ca:	c3	retq	

Hukommelses Hierarkier

- Nogle fundamentale og varige egenskaber ved hardware og software:
 - Hurtige hukommelsestyper koster mere per byte, har mindre kapacitet, men er hurtigere (og kræver mere effekt (varme!)).
 - Hastigheds-gabet mellem CPU og primær hukommelse øges.
 - Vel-skrevne programmer udviser typisk god lokalitet.
- Disse fundamentale egenskaber komplementerer hinanden perfekt!
 - → Caché hukommelse
- Lægger op til en måde at organisere hukommelses/lagringssystemer på: **hukommelseshierarkiet!**

Eksempel på Hukommelses- hieraki

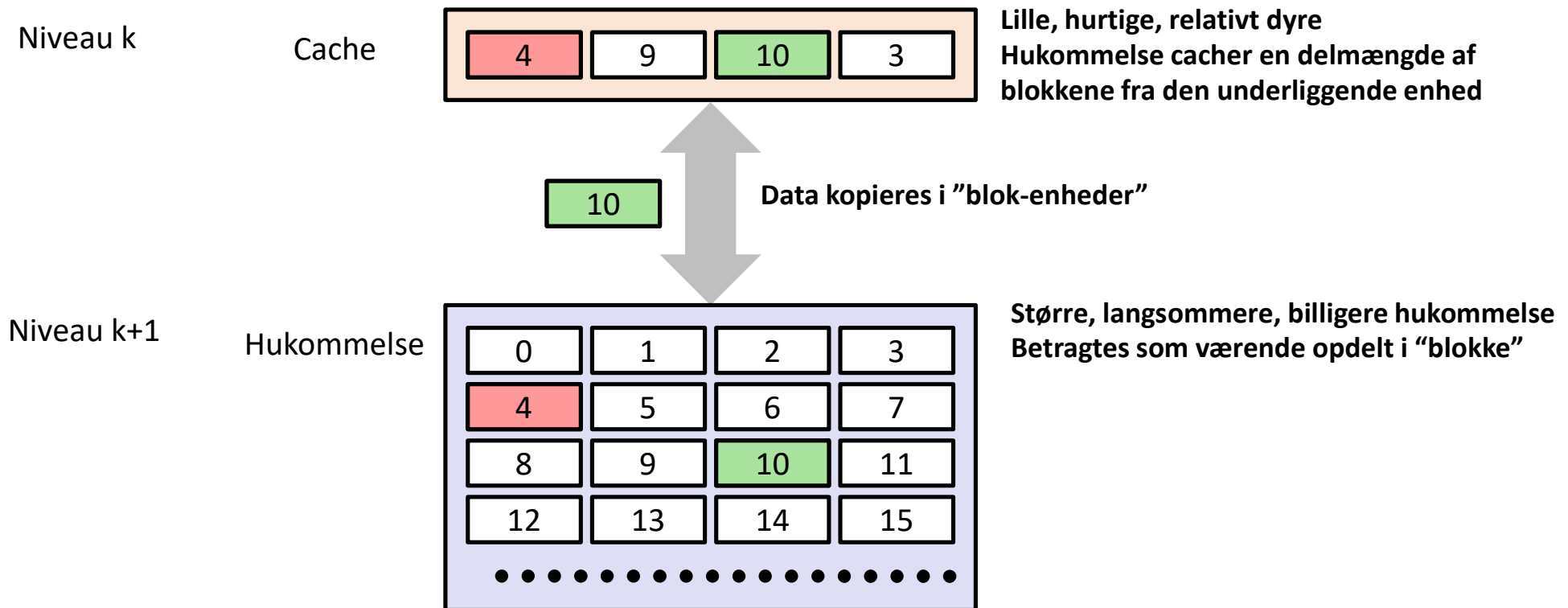


Generelle cache principper

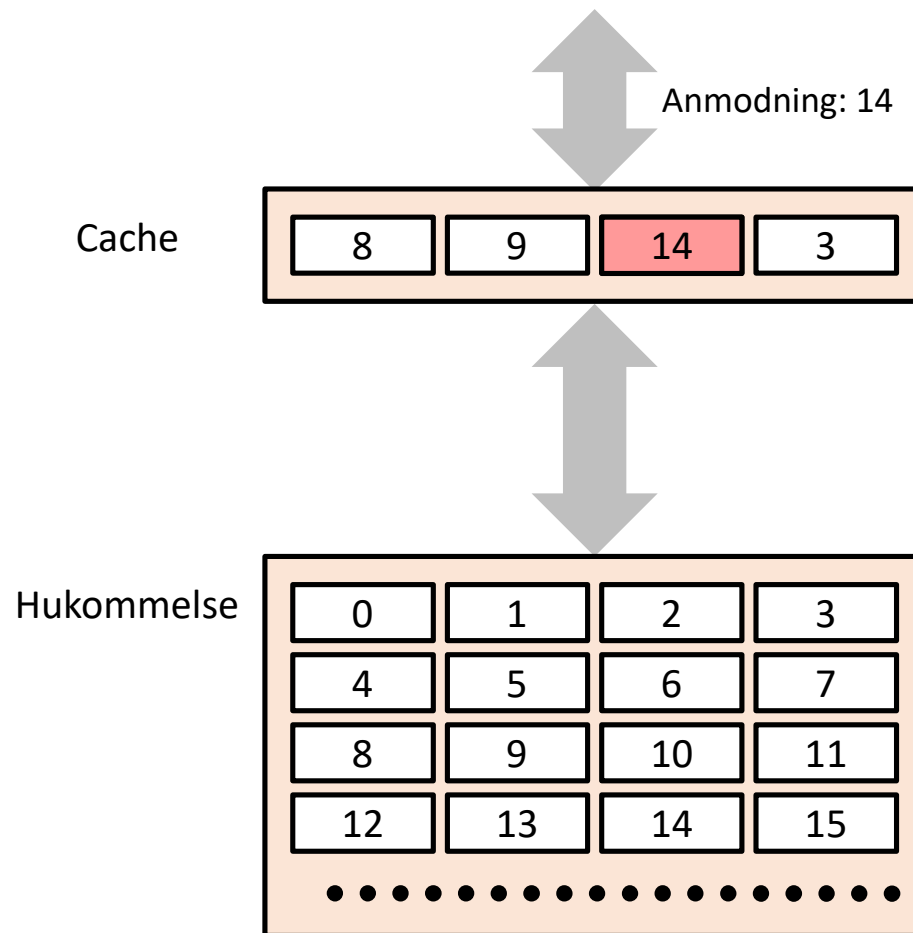
Cache-hukommelse ("Forråd")

- **Cache:** En mindre, hurtig hukommelsesenhed, som fungerer som forråd for en del-mængde af data fra større, langsommere enheder
- Fundamentale ide i et hukommelses-hierarki:
 - For hver niveau k : den hurtigere, mindre enhed på niveau k tjener som cache for den større, langsomme enhed på niveau $k+1$.
- Hvorfor virker hukommelses-hierarkier?
 - Pga. temporal lokalitet tilgår programmer typisk data på niveau k hyppigere end data på niveau $k+1$.
 - Pga. spatial lokalitet er det nemmere at forud indlæse data til niveau k
 - Dermed vi effektivt udnytte niveau $k+1$'s langsommere, større, billigere per bit hukk..
- **Den hellige gral:** Hukommelses-hierarkiet laver en hukommelse der **tilsyneladende** er stort og billigt (som fra bunden), men som leverer data med en hastighed som fra toppen!

Generelle cache begreber:



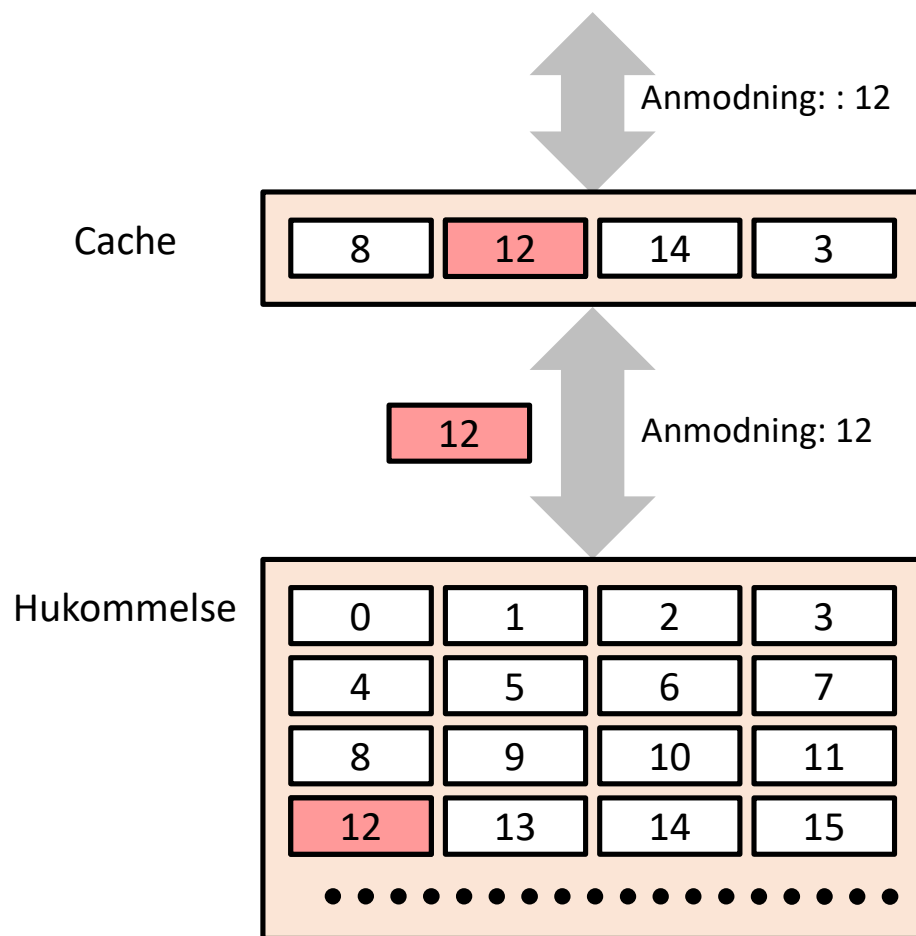
Generelle cache begreber: Hit



Vi skal bruge data i blok b

Blok b findes allerede i cache:
Plets kud (Hit!)

Generelle cache begreber: Miss



Vi skal bruge data i blok b!

***Blok b findes ikke i cache:
Forbier (Miss)!***

***Blok b hentes fra
hukommelsen***

Block b gemmes i cache

- **Placerings politik:**
Afgør hvor b placeres
- **Erstatnings-politik:**
Afgør hvilken (eksisterende cache) blok, der skal smides ud (offer)
FX approximationer til "LRU" (least-recently used): mindst-fornyligt brug blok ofres

Generelle cache begreber :

Cache Miss Typer

- **Kold (tvungen) miss:**
 - Opstår fordi cachen er tom.
- **Konflikt miss:**
 - De fleste cache implementationer har restriktioner for på hvilken plads som blokke fra niveau $k+1$ kan optage i niveau k .
 - Fx. Blokke b skal indplaceres i blok $(b \bmod 4)$ på niveau k
 - Opstår når multiple blokke slås om samme plads, men cachen i øvrigt er stor nok (andre ledige blokke).
 - Fx. Reference til blokkene: 0, 8, 0, 8, 0, 8, ... ville give miss hver gang.
- **Kapacitets miss**
 - Opstår når mængden af "aktive cache blokke" (**working set**) overgår cachen's kapacitet.

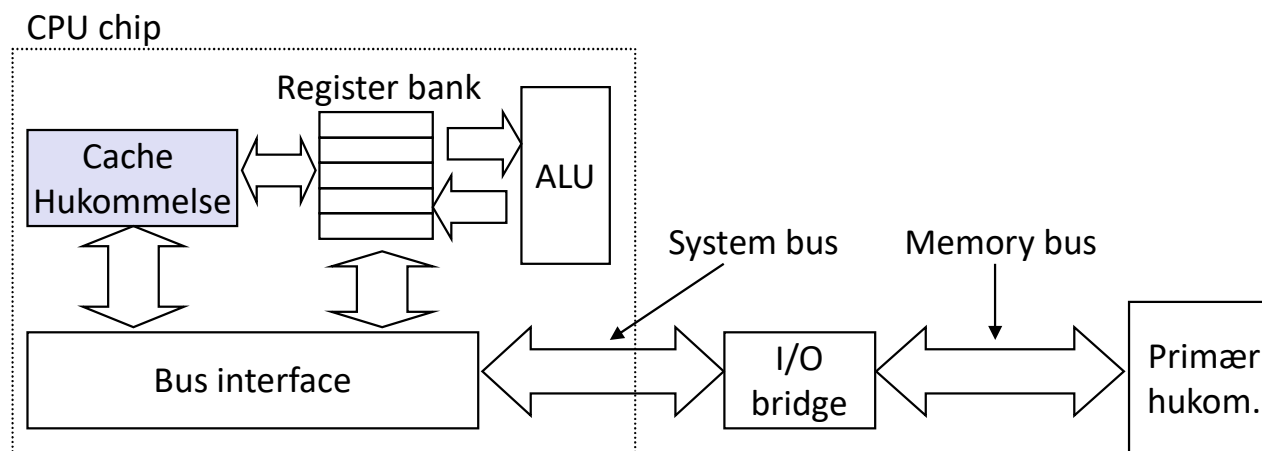
Eksempler på Caching in hukommelses-hierarkiet

Cache Type	Hvad Caches?	Hvor er Cachen?	Latens (cyclers)	Styres af
Registre	4-8 bytes ords	CPU kerne	0	Compiler
TLB	Adresse oversættelse	On-Chip TLB	0	Hardware MMU
L1 cache	64-byte blokke	On-Chip L1	4	Hardware
L2 cache	64-byte blokke	On-Chip L2	10	Hardware
Virtuel Hukom.	4-KB sider	Primær hukommelse	100	Hardware + OS
Buffer cache	Dele af filer	Primær hukommelse	100	OS
Disk cache	Disk sektorer	Disk controller	100,000	Disk firmware
Netværks buffer cache	Dele af filer	Lokal disk	10,000,000	Network File System client
Browser cache	Web sider	Lokal disk	10,000,000	Web browser + web server
Web cache	Web pages	Server disks på (fjern) web server	1,000,000,000	Web proxy server

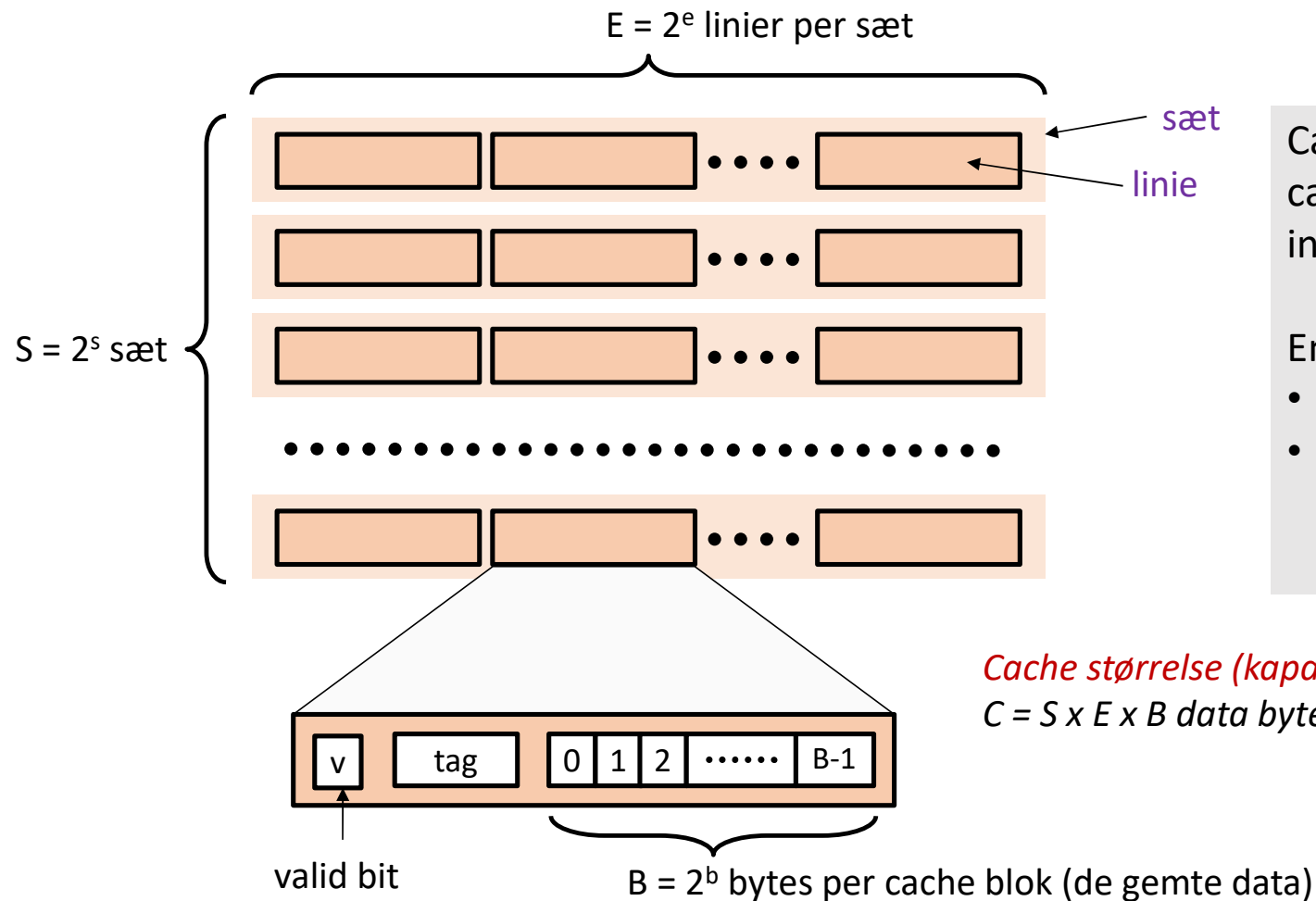
CPU cache typer

CPU Cache Hukommelser

- **CPU Cache hukommelser** er små, hurtige SRAM-hukommelser som administreres automatisk af hardware
 - Gemmer hyppigt anvendte blokke fra primær hukommelse
- Når processor udsteder adresse, slår HW først op i cache efter ordet
- Typisk system struktur



Generel CPU Cache Opbygning (S, E, B)



Cache består af et array af S cache-sæts, hver med E indgange (linier)

En linie består af

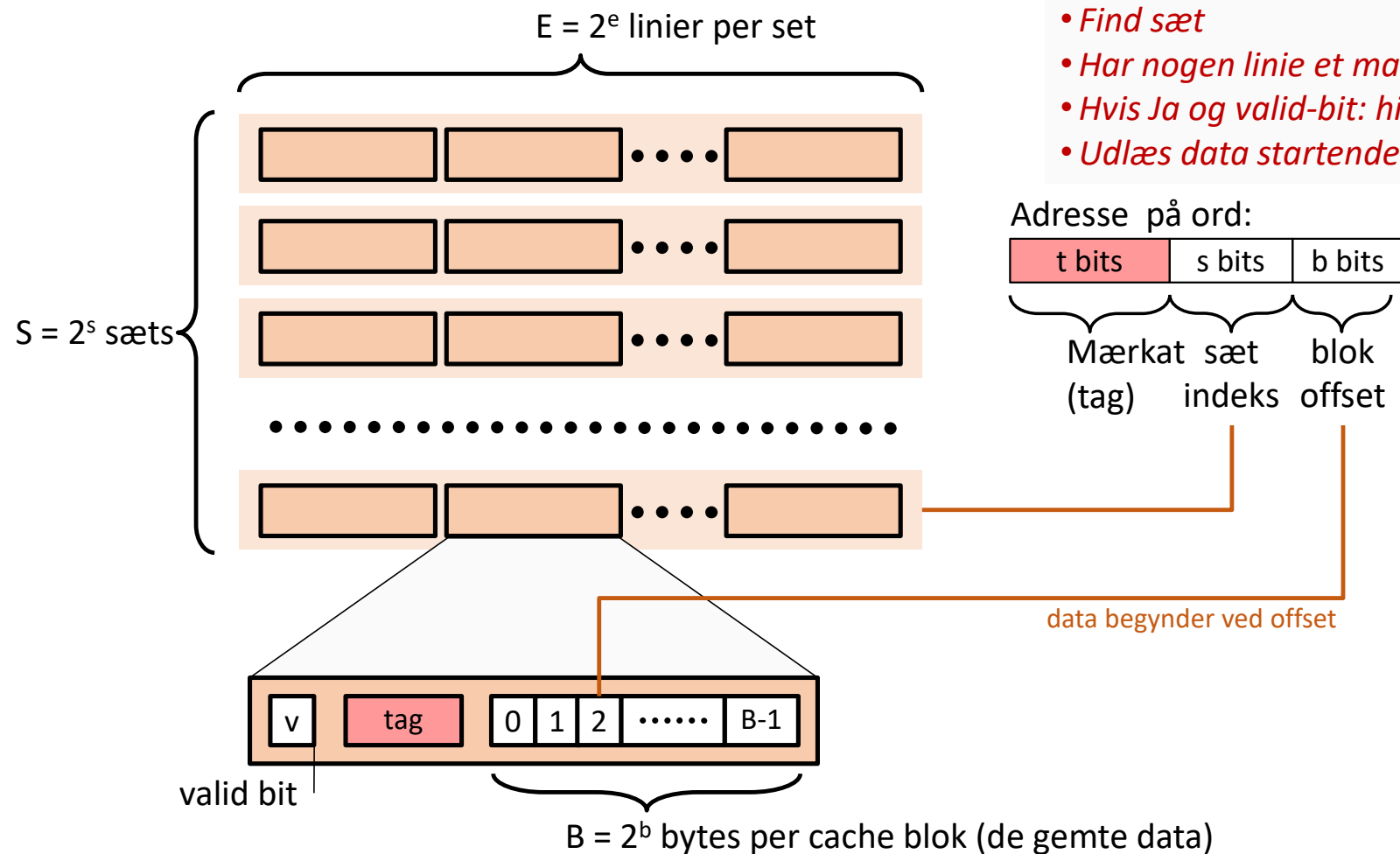
- data-blok på B bytes
- kontrol info
 - valid-bit,
 - mærkat (tag)

Cache størrelse (kapacitet):

$$C = S \times E \times B \text{ data bytes}$$

Cache Læsning

- Opslag skal gå (rasende) hurtigt
- Bruger (struktur) på adressen til at begrænse hvor blokken findes

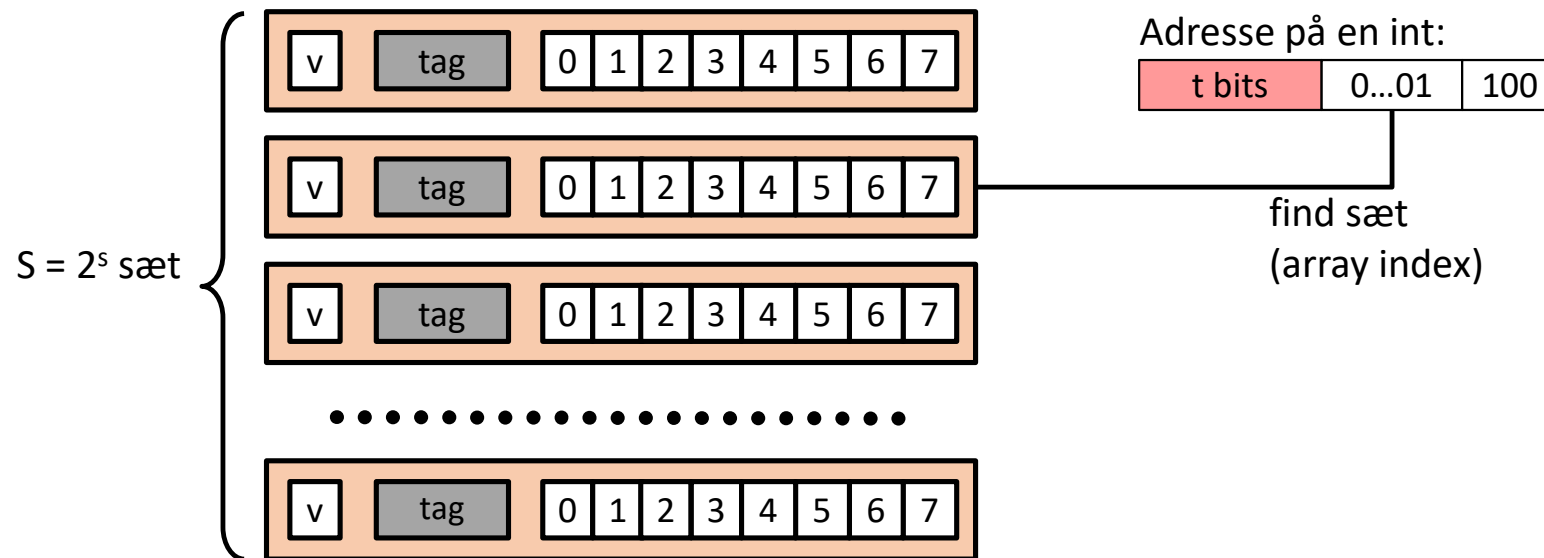


- *Find sæt*
- *Har nogen linie et matchende mærkat?*
- *Hvis Ja og valid-bit: hit*
- *Udlæs data startende på offset*

Eksempel: Direkte kobling ($E = 1$)

Direkte kobling "Direct mapped cache": Én linie per sæt

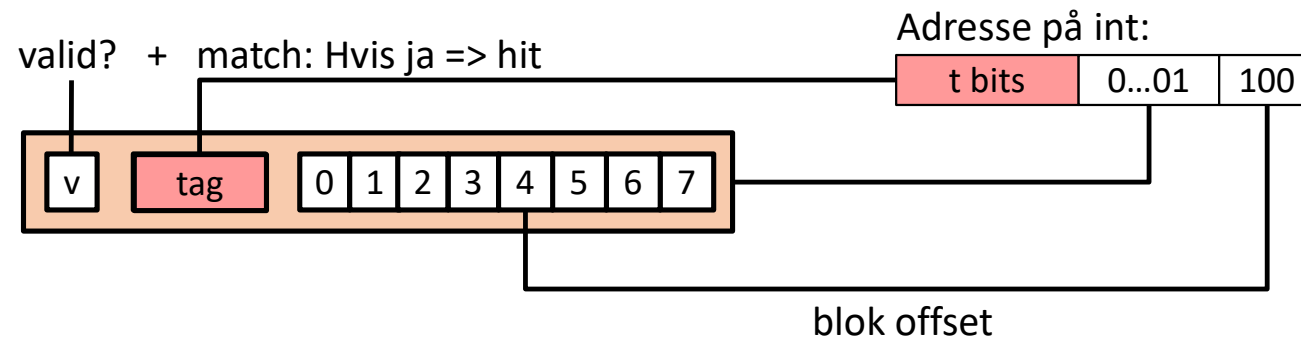
Antag at hver cache blok er 64-bits (8 bytes): $\Rightarrow b=3$ bits til blok offset



Eksempel: Direkte kobling ($E = 1$)

Direkte kobling "Direct mapped cache": Én linie per sæt

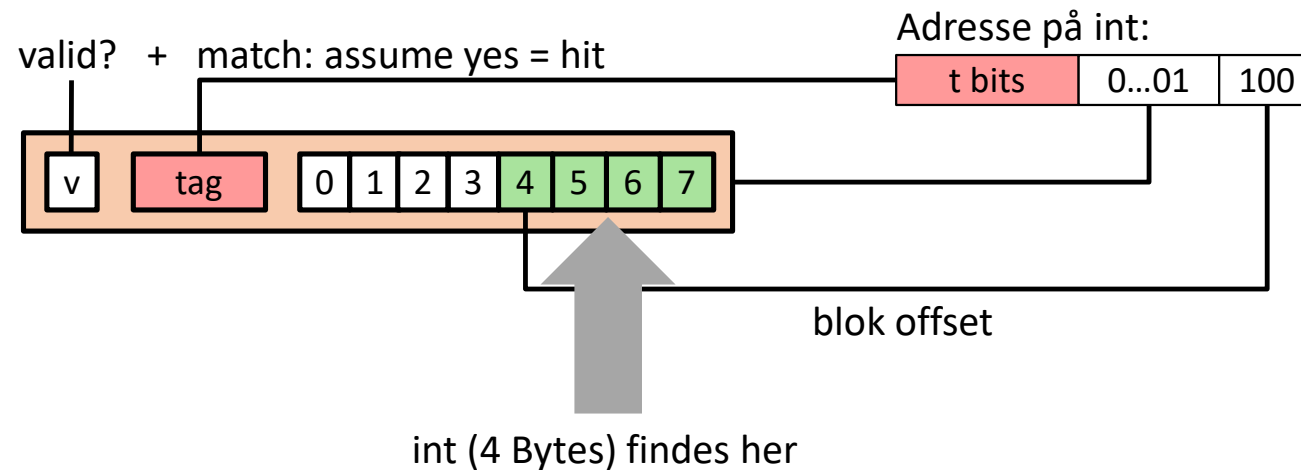
Antag at hver cache blok er 64-bits (8 bytes): $\Rightarrow b=3$ bits til blok offset



Eksempel: Direkte kobling ($E = 1$)

Direkte kobling "Direct mapped cache": Én linie per sæt

Antag at hver cache blok er 64-bits (8 bytes): $\Rightarrow b=3$ bits til blok offset



Hvis mærket ikke matcher: gammel line forkastes (eviction) og erstattes

Simulering af direkte koblet cache (E = 1)

t=1	s=2	b=1
x	xx	x

Antage 4-bit adresser:

Primær hukommelse M=16 bytes

Cache: B=2 bytes/blok, S=4 sæts, E=1 Blokke/sæt, C=4*1*2 bytes

	v	Tag	Blok
Sæt 0	1	0	M[0-1]
Sæt 1			
Sæt 2			
Sæt 3	1	0	M[6-7]

Adresse serie (læsning, en byte per læsning):

0	[0000] ₂ ,	miss
1	[0001] ₂ ,	hit
7	[0111] ₂ ,	miss
8	[1000] ₂ ,	miss
0	[0000] ₂	miss

Konflikt misses!

E-vejs Sæt Associativ Cache (Her: E = 2)

Normalt, færre
konflikt-misses

2-vejs: linie per sæt

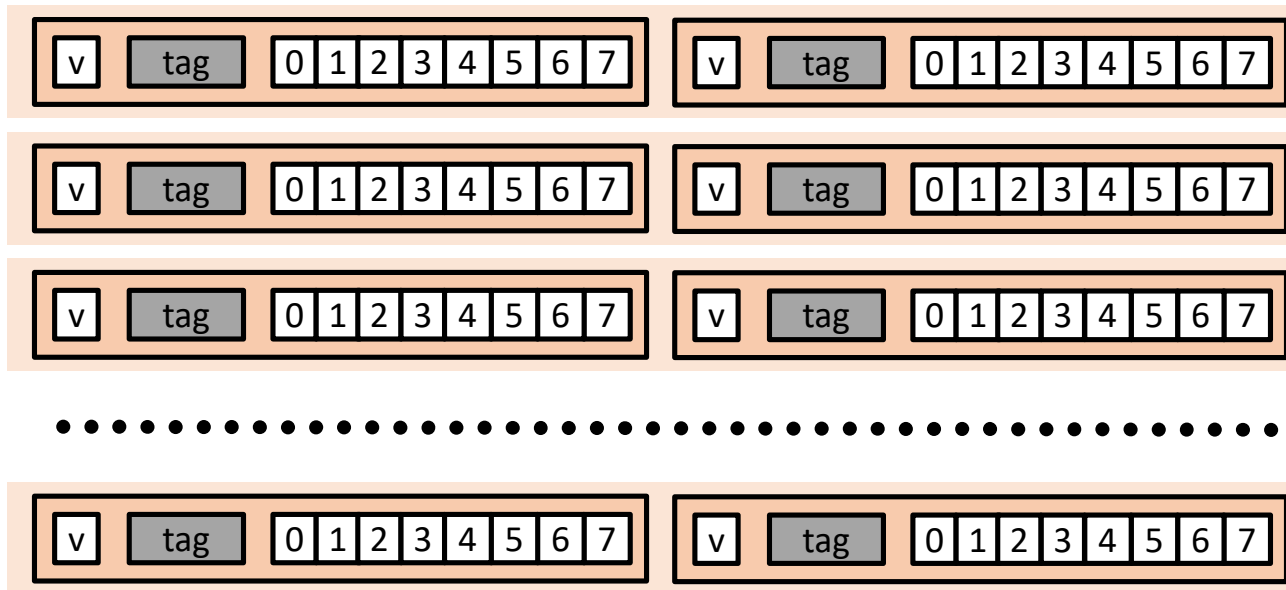
Associativ: Inden for hver sæt skal vi finde et matchende mærkat

Antag at hver cache blok er 64-bits (8 bytes): \Rightarrow b=3 bits til blok offset

Adresse på short int:

t bits	0...01	100
--------	--------	-----

find sæt
(array index)

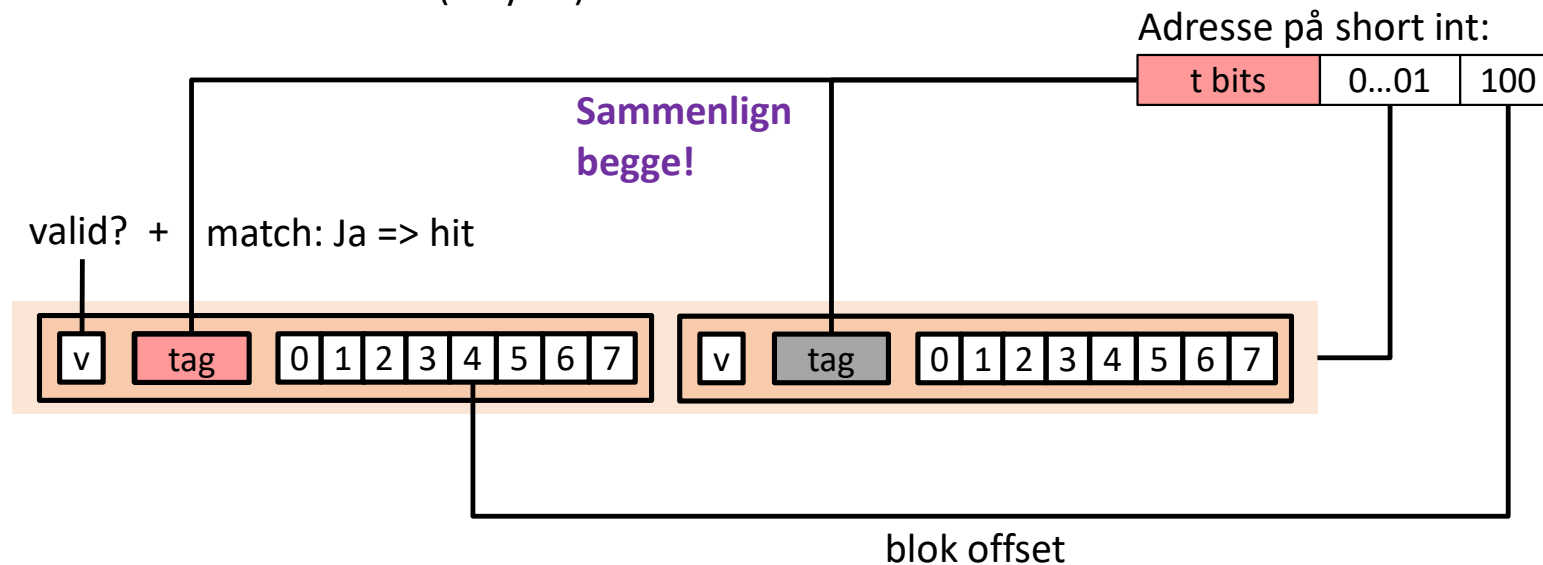


E-vejs Sæt Associativ Cache (Her: E = 2)

2-vejs: linie per sæt

Associativ: Inden for hver sæt skal vi finde et matchende mærkat

Antag at hver cache blok er 64-bits (8 bytes): => b=3 bits til blok offset

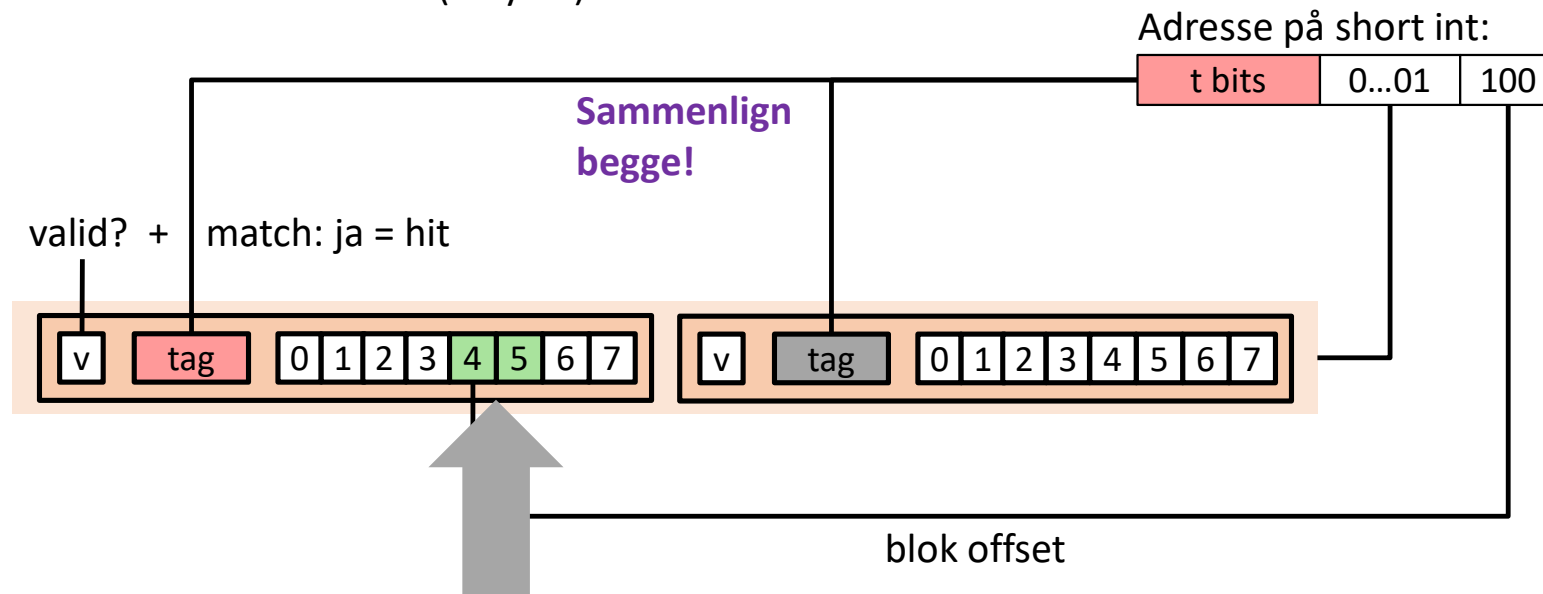


E-vejs Sæt Associativ Cache (Her: E = 2))

2-vejs: linie per sæt

Associativ: Inden for hver sæt skal vi finde et matchende mærkat

Antag at hver cache blok er 64-bits (8 bytes): => b=3 bits til blok offset



short int (2 Bytes) findes her

Ingen match:

- En linie i sættet vælges til ofring (eviction) og erstatning; hvis en findes med v=0 et godt offer
- Erstatningspolitikker: arbitrær, least recently used (LRU), ...

Simulering af E-vejs Sæt Associativ Cache (Her: E = 2)

t=2	s=1	b=1
xx	x	x

Antag 4-bit adresser:

Primær hukommelse M=16 bytes

Cache: B=2 bytes/blok, S=1 sæts, E=2 Blokke/sæt,
C=2*2*2 bytes

	v	Tag	Block
Sæt 0	1	00	M[0-1]
	1	10	M[8-9]
Sæt 1	1	01	M[6-7]
	0		

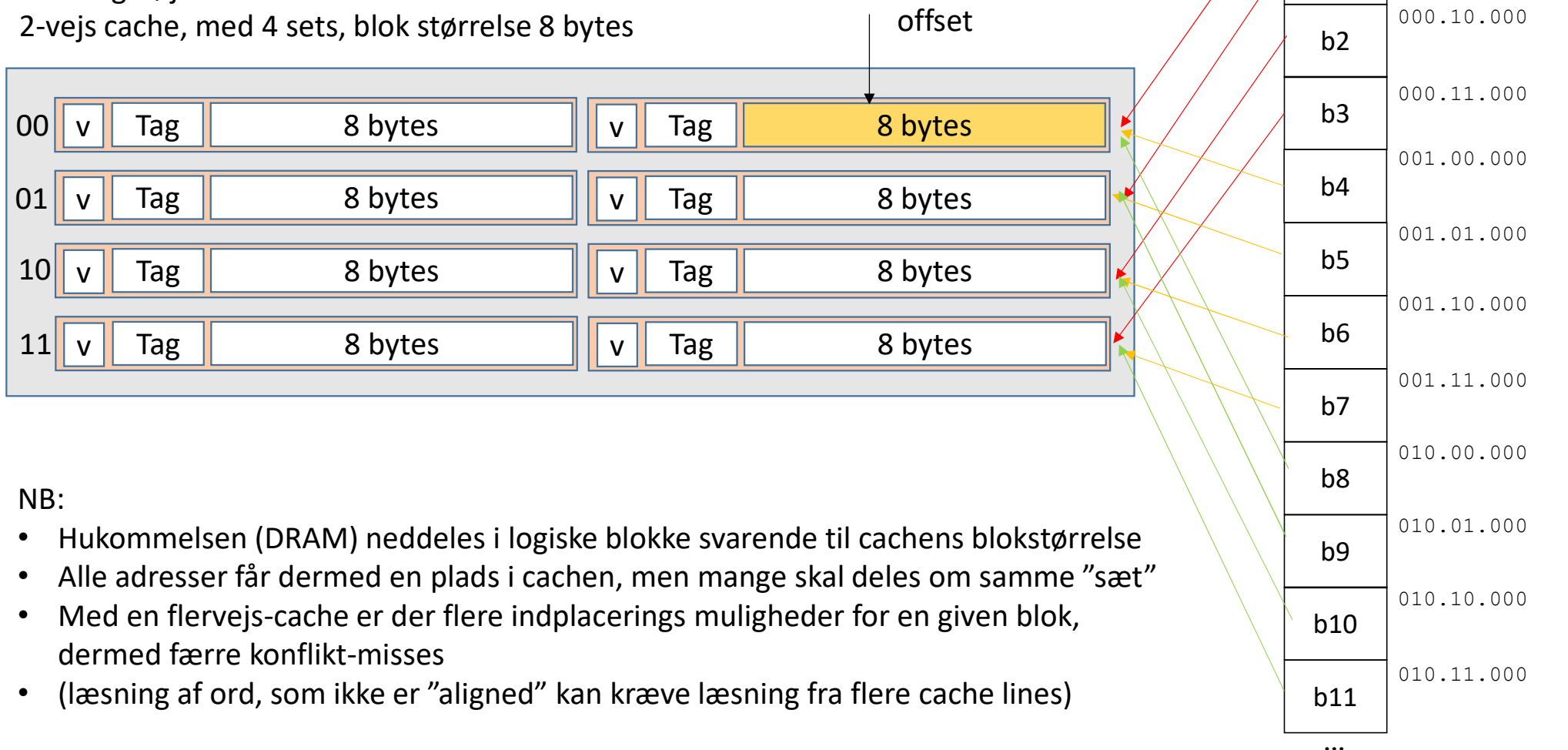
Adresse serie (læsning, en byte per læsning):

0	[0000] ₂ ,	miss
1	[0001] ₂ ,	hit
7	[0111] ₂ ,	miss
8	[1000] ₂ ,	miss
0	[0000] ₂	hit

Mapping af adresser til cache

EX: "Legetøjscache"

2-vejs cache, med 4 sets, blok størrelse 8 bytes



NB:

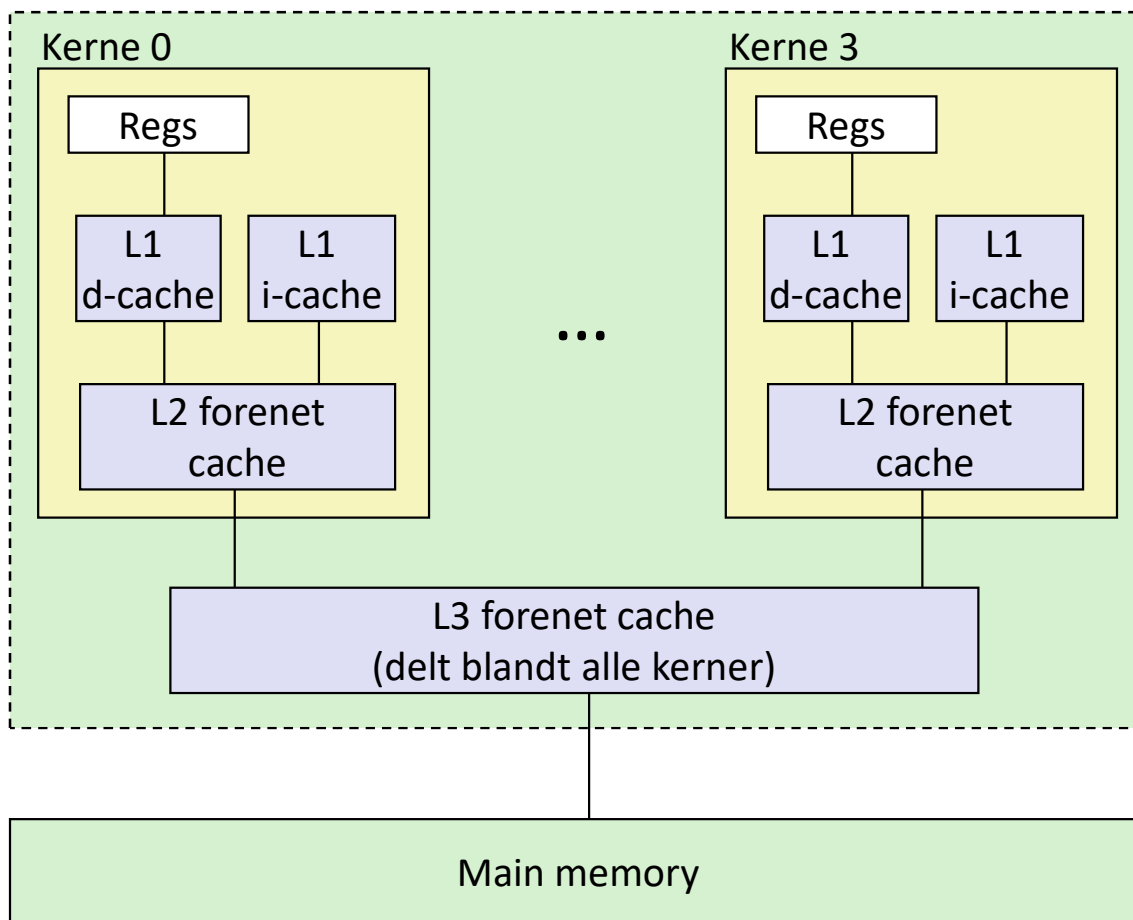
- Hukommelsen (DRAM) neddeles i logiske blokke svarende til cachens blokstørrelse
- Alle adresser får dermed en plads i cachen, men mange skal deles om samme "sæt"
- Med en flervejs-cache er der flere indplacerings muligheder for en given blok, dermed færre konflikt-misses
- (læsning af ord, som ikke er "aligned" kan kræve læsning fra flere cache lines)

Hvad med skivninger

- Data findes i flere kopier:
 - L1, L2, L3, Primær hukommelse, Disk
- Hvad skal vi gøre ved "write-hit" (ord allerede i cache)?
 - **Gennemskrivning (Write-through)**: opdatér cache og skriv øjeblikkeligt til hukommelsen
 - **Tilbage-skrivning (Write-back)**: opdatér cache, men udsæt skrivning til linien skal erstattes
 - Cache linie udvides med "modificeret (dirty) bit" (sat hvis linien er forskellig fra hukommelsen)
- Hvad skal vi gøre ved "write-miss"?
 - **Skrive-allokering (Write-allocate)**: vælg plads til ord i cache, opdatér linien)
 - God, hvis der efterfølgende kommer flere skrivning til samme lokation (linie)
 - **Uden-skrive-allokering (No-write-allocate)**: skriver direkte til hukommelsen udenom cache
- Typiske kombinationer
 - Write-through + No-write-allocate
 - **Write-back + Write-allocate**

Intel Core i7 Cache Hierarki

Processor pakke (CPU-chip)



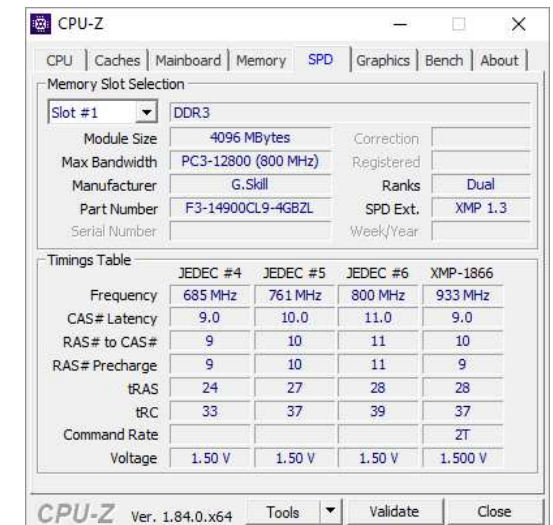
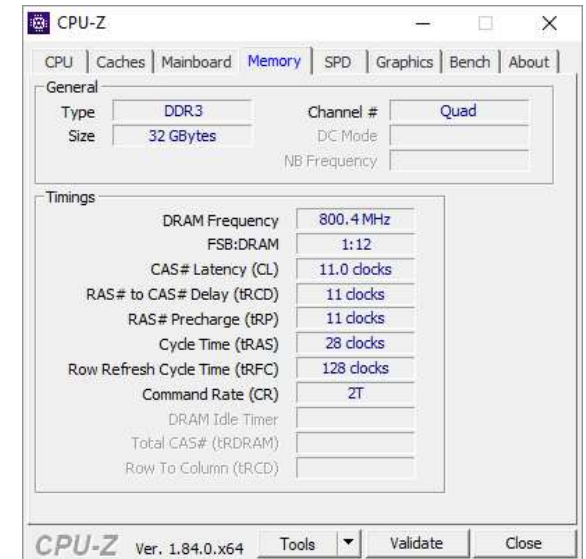
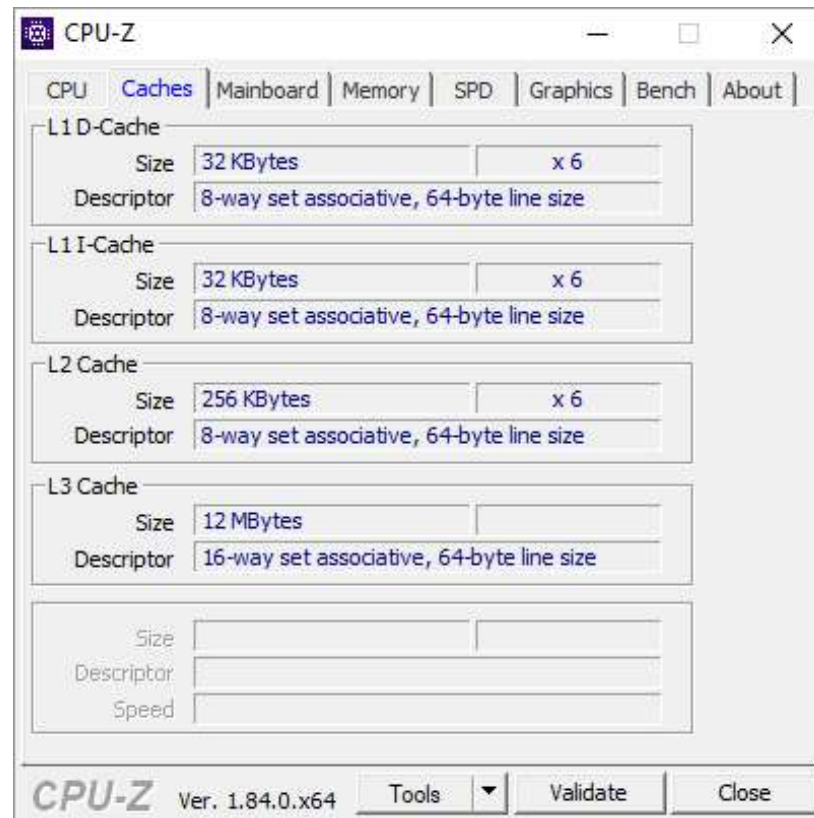
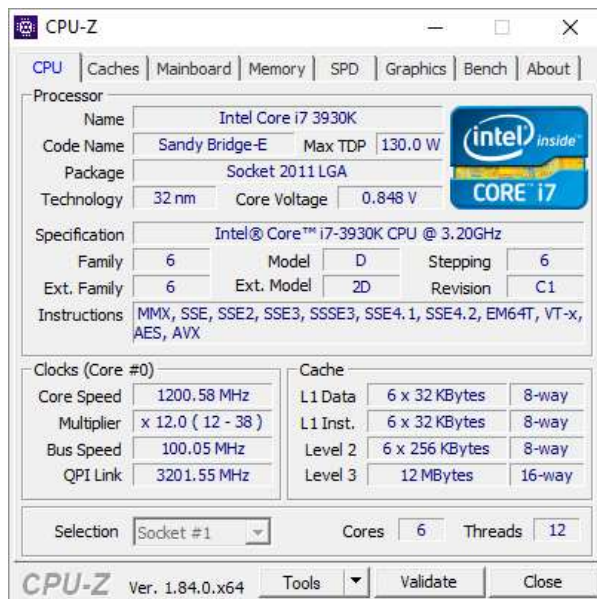
L1 i-cache og d-cache:
32 kB, 8-vejs,
Latens: 4 cykler

L2 unified cache:
256 kB, 8-vejs,
Latens : 10 cykler

L3 unified cache:
8 MB, 16-vejs,
Latens : 40-75 cykler

Blok størrelse: 64 bytes i alle
tilfælde.

Demo CPU-Z værktøj



Metrikker for Cache Performance

- Miss Rate
 - Andel af referencer til hukommelsen som ikke findes i cache (misses / accesses)
= $1 - \text{hit rate}$
 - Typiske tal:
 - 3-10% for L1
 - Kan være ganske lille lille (fx., $< 1\%$) for L2, afhængigt af størrelse, etc.
- Hit Tid
 - Tid til at levere data fra en cache-linie til processoren
 - Inkluderer tiden til at afgøre om linien findes i cache
 - Typiske tal:
 - 4 clock cykler for L1
 - 10 clock cykler for L2
- Ekstra omkostning ved Miss (Miss Penalty)
 - Yderligere krævet tid grundet miss: skal hente data fra underliggende hukommelse
 - typisk 50-200 cykler for primær hukommelsen



Eksempel analyse

- Kæmpe tidsforskel imellem hit og miss
 - Kan fx nemt være 100x, hvis vi kun har L1 og primær hukommelse
- 99% hit rate er dobbelt så godt som 97%!!

- Eksempel
 - cache hit tid 1 cyklus
 - miss penalty: 100 cykler

Vi skal altid læse fra cachen + nogle gange skal opdatere den fra primær hukommelse

- Gennemsnitlig adgangstid:
 - 97% hits: 1 cyklus + 0.03 * 100 cykler = **4 cykler**
 - 99% hits: 1 cyklus + 0.01 * 100 cykler = **2 cykler**

- Derfor fokuserer man på “miss rate” i stedet for “hit rate”

1. (4 pts)

Antag at svartiden ved cache hit er 8 cycles (hit time), og hovedhukommelsen har en svartid på 100 cycles (miss penalty). Hvis miss raten er på 2%, hvad bliver da den gennemsnitlige svartid ? Angiv svaret med én decimals precision.

$$\frac{98 \cdot 8 + 2 \cdot 100}{100} = 9.8 \text{ cycles}$$

$$\frac{(100 \cdot 8 + 2 \cdot 100)}{100} = 10.0 \text{ cycles}$$

Cache-venlige programmer

Skriv cache venlig kode

- Få det hyppige tilfælde (common case) til at køre hurtigt
 - Fokuser på de indre løkker i centrale funktioner
- Minimér antal cache miss i de indre løkker
 - Gentagende referencer til variable er godt (**temporal lokalitet**)
 - Skridt længde (stride-1) reference mønster er godt (**spatial lokalitet**)

Nøgle idé: Vi kan bruge vores viden om caches til at kvantificere vores kvalitative forståelse af lokalitet

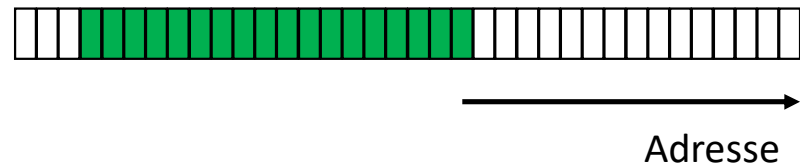
Kvalitativt Estimat af Lokalitet

- **Påstand:** Det er vigtigt for en professionel programmør at kunne kigge på koden og vurdere om den viser god/dårlig lokalitet!!!
- **Spørgsmål:** Udviser denne funktion god lokalitet mht. array a?

```
int sum_array_rows(int a[M][N])
{
    int i, j, sum = 0;

    for (i = 0; i < M; i++)
        for (j = 0; j < N; j++)
            sum += a[i][j];
    return sum;
}
```

Husk Række-baseret layout!
Alt data i cache-line udnyttes
(fx 64 bytes= 16 ints)

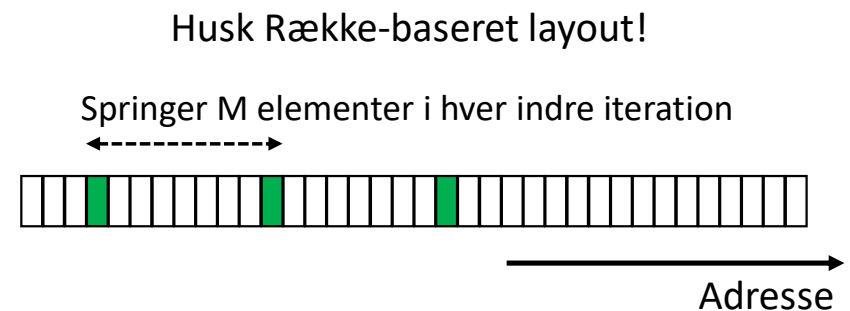


Eksempel på lokalitet

- **Spørgsmål:** Udviser denne funktion god lokalitet mht. array a?

```
int sum_array_cols(int a[M][N])
{
    int i, j, sum = 0;

    for (j = 0; j < N; j++)
        for (i = 0; i < M; i++)
            sum += a[i][j];
    return sum;
}
```



Hukommelses-bjerget

- **Læse-rate** (read bandwidth)
 - Antal bytes der kan læses fra hukommelsen per sekund (MB/s)
- **Hukommelses-bjerget**: Målt læse-rate som funktion af spatial og temporal lokalitet.
 - Kompakt måde at karakterisere performance af et hukommelsessystem.

Test funktion til måling af hukommelses-bjerget

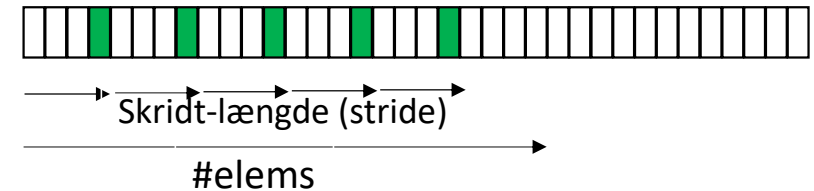
```
long data[MAXELEMS]; /* Global array to traverse */

/* test - Iterate over first "elems" elements of
 * array "data" with stride of "stride", using
 * using 4x4 loop unrolling.
 */
int test(int elems, int stride) {
    long i, sx2=stride*2, sx3=stride*3, sx4=stride*4;
    long acc0 = 0, acc1 = 0, acc2 = 0, acc3 = 0;
    long length = elems, limit = length - sx4;

    /* Combine 4 elements at a time */
    for (i = 0; i < limit; i += sx4) {
        acc0 = acc0 + data[i];
        acc1 = acc1 + data[i+stride];
        acc2 = acc2 + data[i+sx2];
        acc3 = acc3 + data[i+sx3];
    }

    /* Finish any remaining elements */
    for (; i < length; i++) {
        acc0 = acc0 + data[i];
    }
    return ((acc0 + acc1) + (acc2 + acc3));
}

mountain/mountain.c
```



Kald `test()` med mange kombinationer af `elems` og `stride`.

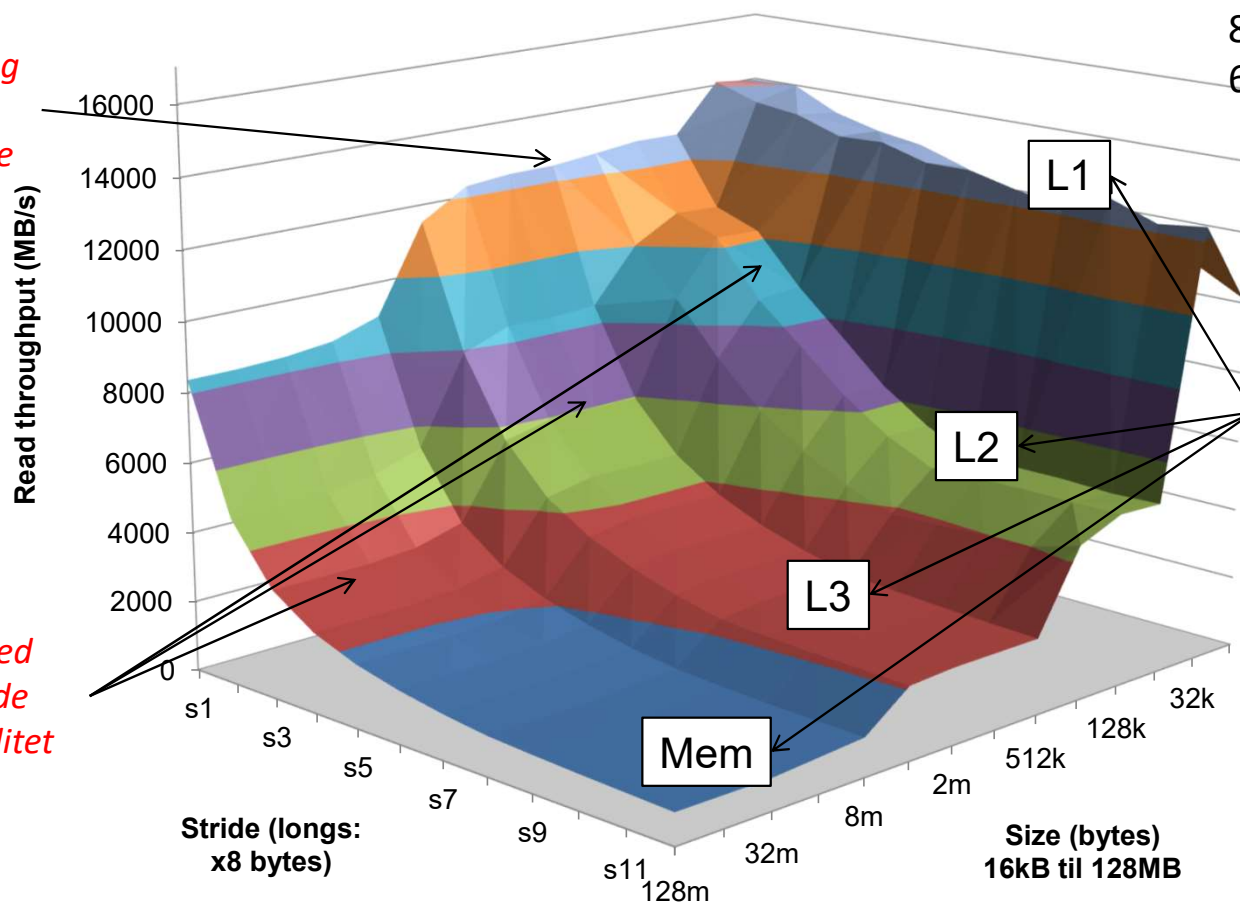
For hver kombination:

1. Kald `test()` en gang for at "opvarme" caches.
2. Kald `test()` igen og mål læserate (MB/s)

Hukommelses-bjerget

Aggressiv
forudindlæsning
(prefetching)
ved skridtlængde
1

Skrænter med
nedadgående
spatial lokalitet



Core i7 Haswell
2.1 GHz
32 KB L1 d-cache
256 KB L2 cache
8 MB L3 cache
64 B blok størrelse

Højderygge med
temporal lokalitet

Matrix Multiplikation Eksempel

- Beskrivelse

- Multiplikation af N x N matricer
- Matrix elementer: doubles (8 bytes)
- $O(N^3)$ total operationer
 - N^2 resultat elementer i "C" skal beregnes
 - Hver kræver N mult+sums fra source matricerne A,B

- Antag

- Stort matrix ifht L1-cache
- 4 doubles pr cache linie (32 Bytes)

- Analyse Metode:

- Undersøg adgangs mønstret i den indre løkke for program varianter

```
/* ijk */
for (i=0; i<n; i++) {
    for (j=0; j<n; j++) {
        sum = 0.0;
        for (k=0; k<n; k++)
            sum += a[i][k] * b[k][j];
        c[i][j] = sum;
    }
}
```

matmult/mm.c

Variablen *sum* gemmes i register

$$\begin{array}{|c|c|} \hline C & \\ \hline C_{11} & C_{12} \\ \hline C_{21} & C_{22} \\ \hline \end{array} = \begin{array}{|c|c|} \hline A & \\ \hline A_{11} & A_{12} \\ \hline A_{21} & A_{22} \\ \hline \end{array} * \begin{array}{|c|c|} \hline B & \\ \hline B_{11} & B_{12} \\ \hline B_{21} & B_{22} \\ \hline \end{array}$$

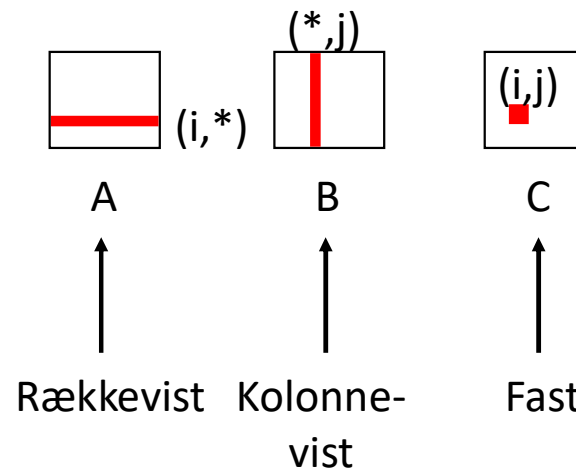
$$C_{11} = A_{11} * B_{11} + A_{12} * B_{21}$$

Matrix Multiplikation (1/6: ijk)

```
/* ijk */
for (i=0; i<n; i++) {
    for (j=0; j<n; j++) {
        sum = 0.0;
        for (k=0; k<n; k++)
            sum += a[i][k] * b[k][j];
        c[i][j] = sum;
    }
}
```

matmult/mm.c

Indre løkke:



Antal miss per iteration af indre løkke :

<u>A</u>	<u>B</u>	<u>C</u>
0.25	1.0	0.0

Skridtlængde 1

Skridtlængde N

Matrix Multiplikation (2/6: jik)

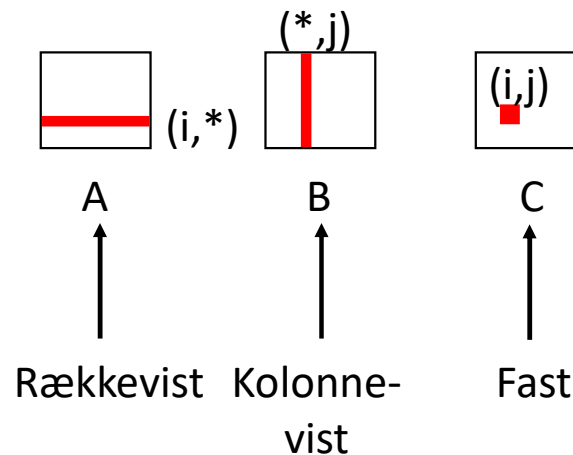
```
/* jik */  
for (j=0; j<n; j++) {  
    for (i=0; i<n; i++) {  
        sum = 0.0;  
        for (k=0; k<n; k++)  
            sum += a[i][k] * b[k][j];  
        c[i][j] = sum  
    }  
}
```

matmult/mm.c

Antal miss per iteration af indre løkke

<u>A</u>	<u>B</u>	<u>C</u>
0.25	1.0	0.0

Indre løkke:



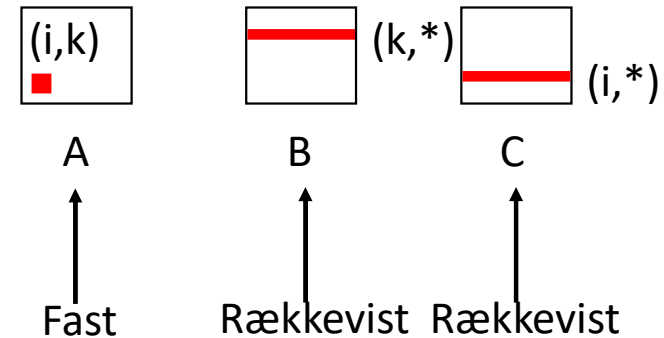
Byttet ydre i,j rundt: Samme mønster som foregående

Matrix Multiplikation (3/6: kij)

```
/* kij */
for (k=0; k<n; k++) {
    for (i=0; i<n; i++) {
        r = a[i][k];
        for (j=0; j<n; j++)
            c[i][j] += r * b[k][j];
    }
}
```

matmult/mm.c

Indre løkke:



Antal miss per iteration af indre løkke :

A
0.0

B
0.25

C
0.25

Skridtlængde 1 Skridtlængde 1

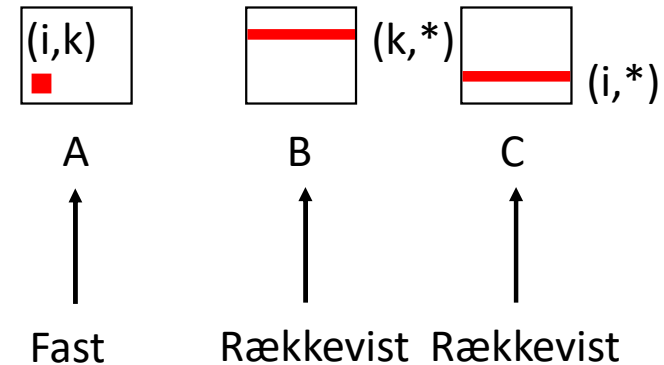
Holder element r fra A fast,
summerer r-elementets bidraget i
række i C

Matrix Multiplikation (4/6: ikj)

```
/* ikj */  
for (i=0; i<n; i++) {  
    for (k=0; k<n; k++) {  
        r = a[i][k];  
        for (j=0; j<n; j++)  
            c[i][j] += r * b[k][j];  
    }  
}
```

matmult/mm.c

Indre løkke:



Antal miss per iteration af indre løkke :

<u>A</u>	<u>B</u>	<u>C</u>
0.0	0.25	0.25

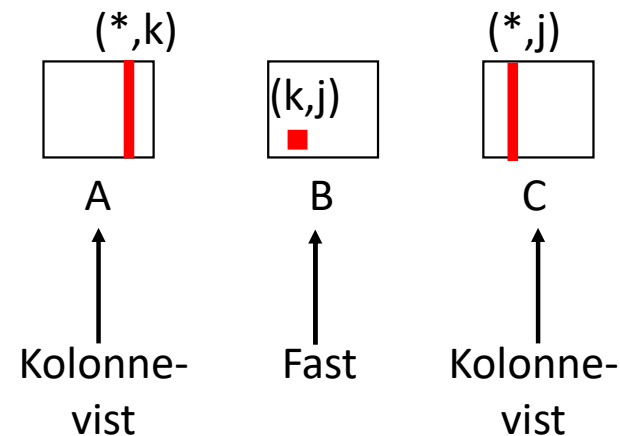
Ombyttet ydre løkker: Samme mønster som foregående

Matrix Multiplikation (5/6: jki)

```
/* jki */
for (j=0; j<n; j++) {
    for (k=0; k<n; k++) {
        r = b[k][j];
        for (i=0; i<n; i++)
            c[i][j] += a[i][k] * r;
    }
}
```

matmult/mm.c

Indre løkke:



Antal miss per iteration af indre løkke

<u>A</u>	<u>B</u>	<u>C</u>
1.0	0.0	1.0

Skridtlængde N

Skridtlængde N

Holder element r fra B fast, summerer r -elementets bidraget til kolonne i C

Matrix Multiplikation (6/6: kji)

```
/* kji */  
for (k=0; k<n; k++) {  
    for (j=0; j<n; j++) {  
        r = b[k][j];  
        for (i=0; i<n; i++)  
            c[i][j] += a[i][k] * r;  
    }  
}
```

matmult/mm.c

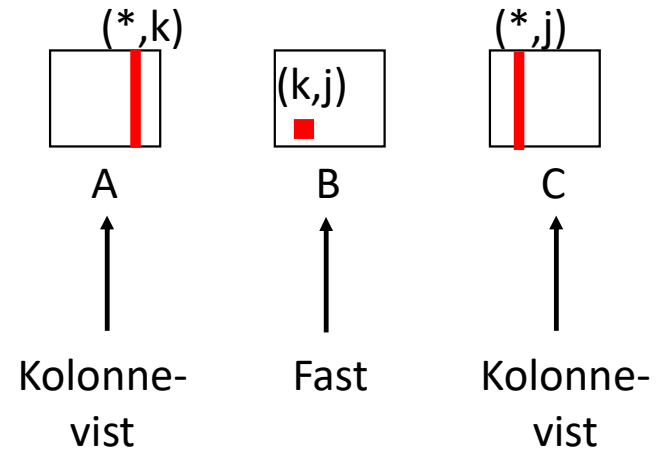
Antal miss per iteration af indre løkke :

A
1.0

B
0.0

C
1.0

Indre løkke:



Ombyttet ydre løkker: Samme mønster som foregående

Resumé af Matrix Multiplikation

```
for (i=0; i<n; i++) {  
    for (j=0; j<n; j++) {  
        sum = 0.0;  
        for (k=0; k<n; k++)  
            sum += a[i][k] * b[k][j];  
        c[i][j] = sum;  
    }  
}
```

ijk (& jik):

- 2 loads, 0 stores
- misses/iter = 1.25

```
for (k=0; k<n; k++) {  
    for (i=0; i<n; i++) {  
        r = a[i][k];  
        for (j=0; j<n; j++)  
            c[i][j] += r * b[k][j];  
    }  
}
```

kij (& ikj):

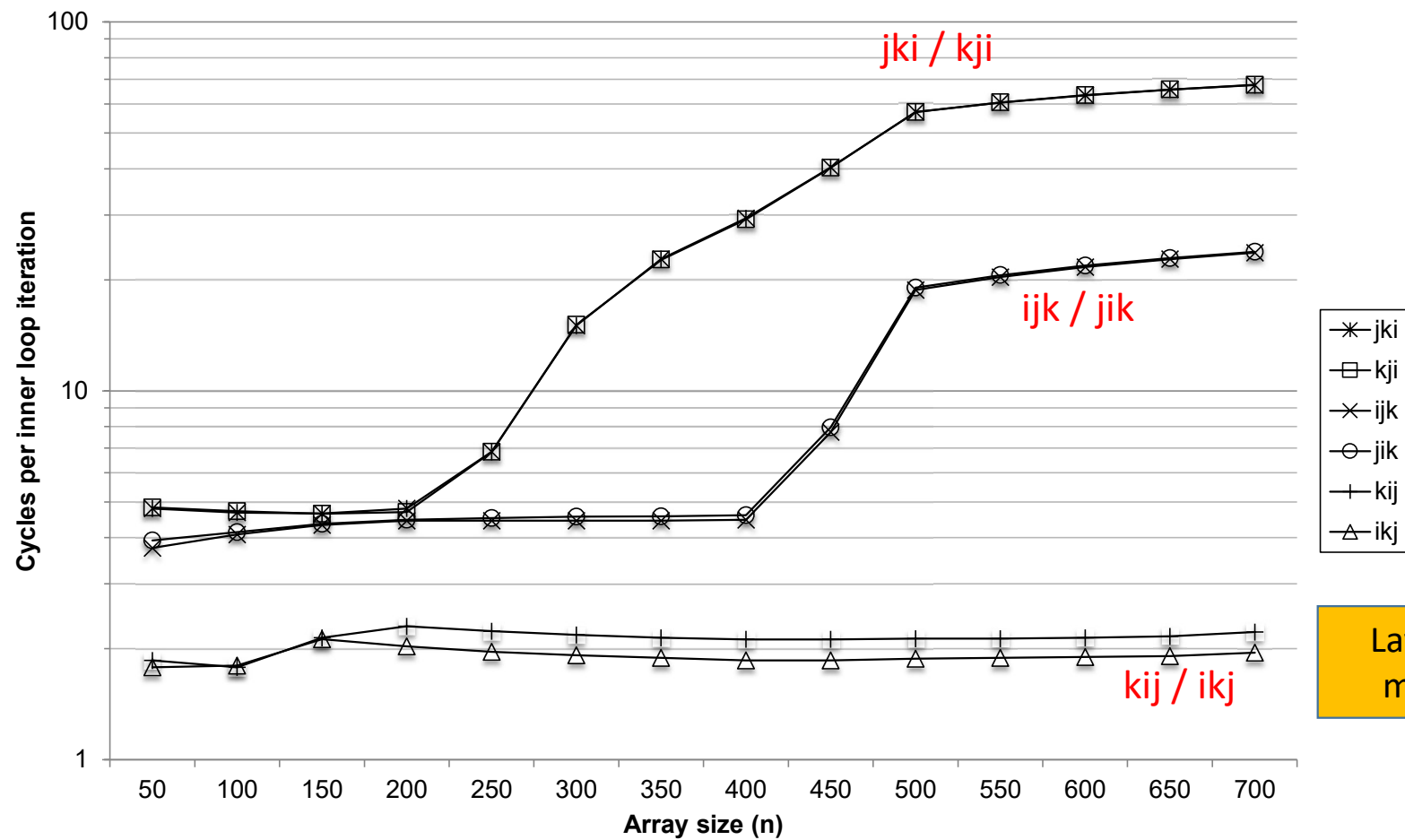
- 2 loads, 1 store
- misses/iter = 0.5

```
for (j=0; j<n; j++) {  
    for (k=0; k<n; k++) {  
        r = b[k][j];  
        for (i=0; i<n; i++)  
            c[i][j] += a[i][k] * r;  
    }  
}
```

jki (& kji):

- 2 loads, 1 store
- misses/iter = 2.0

Performance af Matrix Multiplikation på Core i7

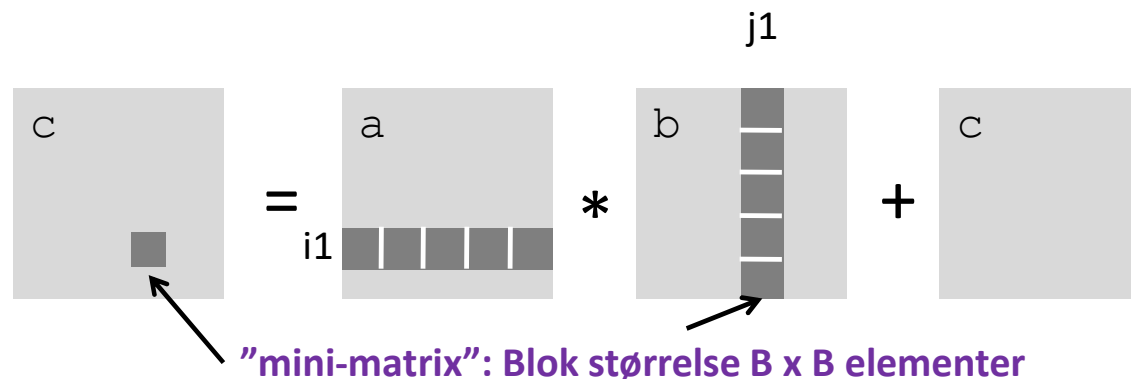


Optimering af Matrix Multiplikation vha. "Blocking"

```
c = (double *) calloc(sizeof(double), n*n);

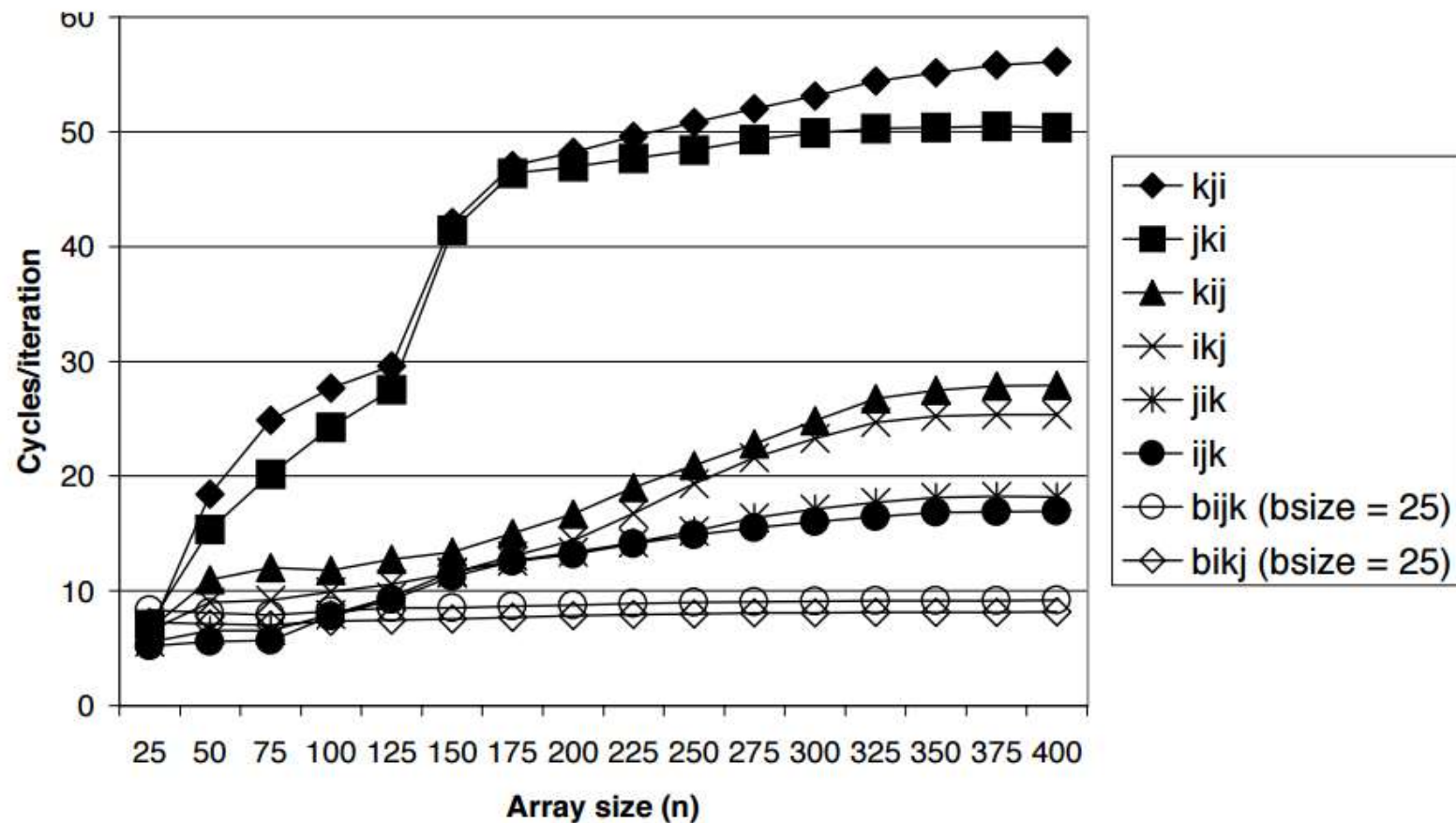
/* Multiply n x n matrices a and b */
void mmm(double *a, double *b, double *c, int n) {
    int i, j, k;
    for (i = 0; i < n; i+=B)
        for (j = 0; j < n; j+=B)
            for (k = 0; k < n; k+=B)
                /* B x B mini matrix multiplications */
                for (i1 = i; i1 < i+B; i++)
                    for (j1 = j; j1 < j+B; j++)
                        for (k1 = k; k1 < k+B; k++)
                            c[i1*n+j1] += a[i1*n + k1]*b[k1*n + j1];
}
```

matmult/bmm.c



Kan være yderst effektivt, men kræver en del omskrivning af koden

Gevinst med Blocking (Pentium 3 Xeon)



<http://csapp.cs.cmu.edu/3e/waside/waside-blocking.pdf>

Cache Resumé

- Lokalitet, lokalitet og lokalitet
- Cache hukommelse har betydelig indflydelse på programmers hastighed
- Skriv programmer så du udnytter dette!
 - Fokuser på de indre løkker, hvor det meste beregning og hukommelsesadgang forekommer
 - Maksimér spatial lokalitet ved at læse data-objekter sekventielt ved at læse med skridt-længde 1.
 - Maksimér temporal lokalitet ved at genbruge data objektet så ofte som muligt når det først er læst (dvs. gør det færdigt en gang for alle).