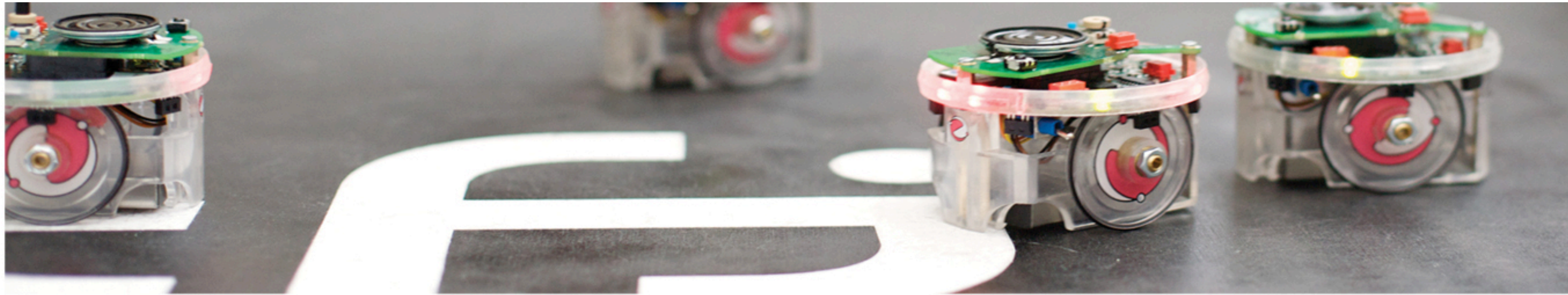


UiO : Institutt for informatikk
Det matematisk-naturvitenskapelige fakultet

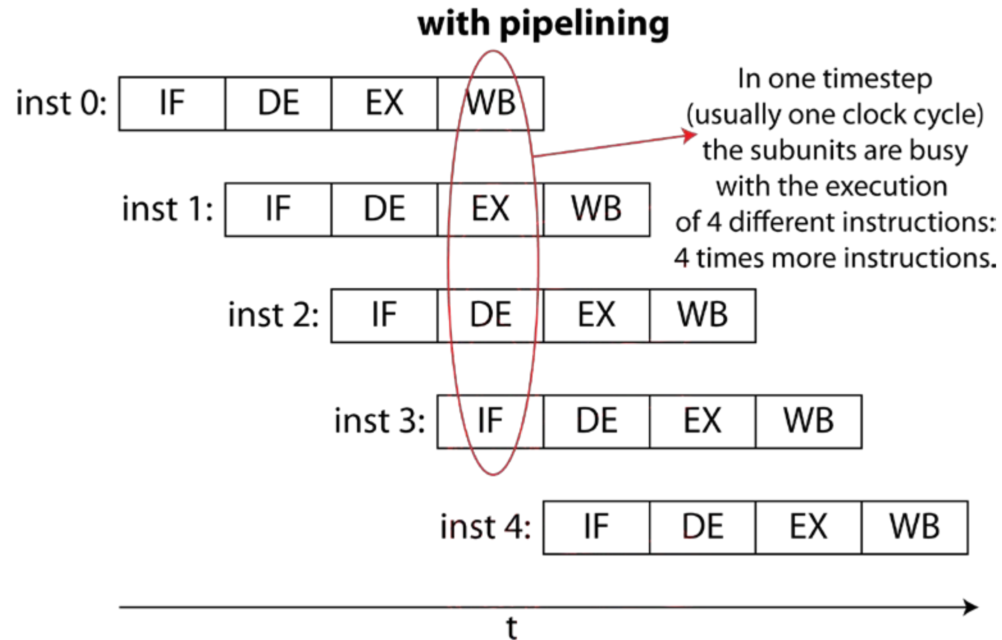
INF2270

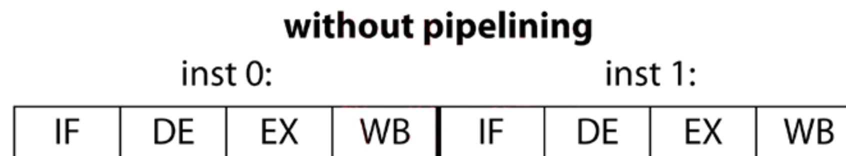
Minnehierarki

Hovedpunkter

- Bakgrunn
  - Kort repetisjon
  - Motivasjon
- Teknikker for hastighetsøkning
  - Multiprosessor
  - Økt klokkehastighet
  - Raskere disker
  - Økt hurtigminne
    - Bruksområder
    - Lagringskapasitet
    - Aksesstider
- Cache
  - Mapping strategier
  - Write strategier
  - Replacement strategier
  - Arkitektur
- Virtuell Minne

Pipeline





Sentrale temaer

- Speed-up
- Hazarder
  - Data
  - Control
  - Resource

Teknikker for hastighetsøkning

- Pipelineing er viktig for å øke antall instruksjoner som utføres pr. sekund

- Andre teknikker som øker hastighet:
  - Flere CPU i en maskin (multiprosessor)
  - Økt klokkefrekvens
  - Stor ordlenge på instruksjoner
  - Raskere disker
  - Hurtigere og større RAM

- Noen av disse er ren forbedring i teknologien, andre er forbedringer av selve arkitekturen.

Multiprosessor

- Ide: Istedenfor å la én CPU utføre instruksjoner, lar man flere CPUer samarbeide om den samme jobben.
- Gir en teoretisk hastighetsøkning direkte proporsjonal med antall ekstra CPUer: det går n ganger raskere med n CPUer enn med én CPU.
- I praksis ikke mulig av flere årsaker:
  - Ikke alltid mulig å dele opp et problem i like store deler som kan løses uavhengig av hverandre.
  - Det kreves administrasjon og koordinering av programeksekveringen før, under og etter at jobben er fordelt på de ulike CPUene, dvs ekstra overhead
- Endel applikasjoner egner seg for lastdeling, f.eks server-applikasjoner, dvs enkelt å parallellisere oppgavene

Økt klokkehastighet og større ordlengde

- Økt klokkehastighet reduserer tiden hver enkelt instruksjon tar (Resultatet av teknologisk utvikling.)
- Hastigheten til alle deler av en datamaskin øker ikke like raskt:
  - Interne data/adressebusser
  - Nettverk
- Klokkehastigheten til en CPU øker mye raskere enn lese/skrivehastigheten til hukommelsen, mao: Effektive hastighets-forbedring blir langt mindre.
- Bredere datapath og ordlengde gjør det mulig å behandle større tall i en operasjon og å lese/skrive mer data fra hukommelsen i én operasjon.
- Det er en øvre grense for hvor mange bit som kan behandles i en operasjon

Raskere disker

- Programmer trenger å aksessere disker av og til ved f.eks innlesning og skriving av data, ved oppstart og virtuell hukommelse
- En harddisk ca 100 000 ganger langsommere enn en CPU, dvs at en CPU kan gjøre 100 000 instruksjoner, mens en harddsik gjør én lese/ skriveoperasjon.
- Teknikker finnes for å lese store blokker av data ad gangen, prøve å gjette hvilke data som skal leses neste gang og lagre disse i hurtigminne, etc
- Harddisker blir både raskere og får mer kapasitet, men allikevel vil hyppig diskaksess være et betydelig problem hvis hastighet er viktig.

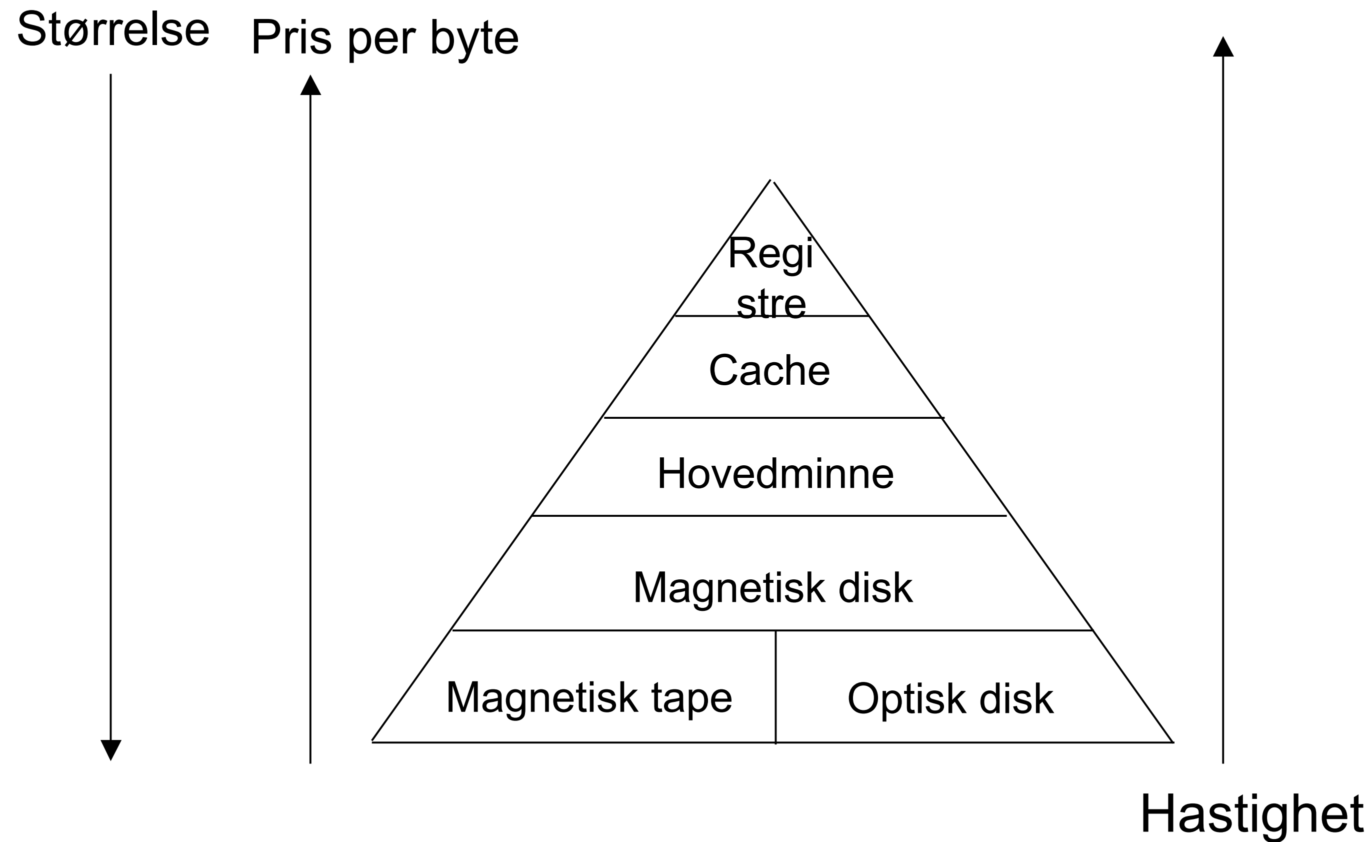
Større og raskere hurtigminne

- Ønsker ubegrenset med minne som er like raskt som CPU'en.

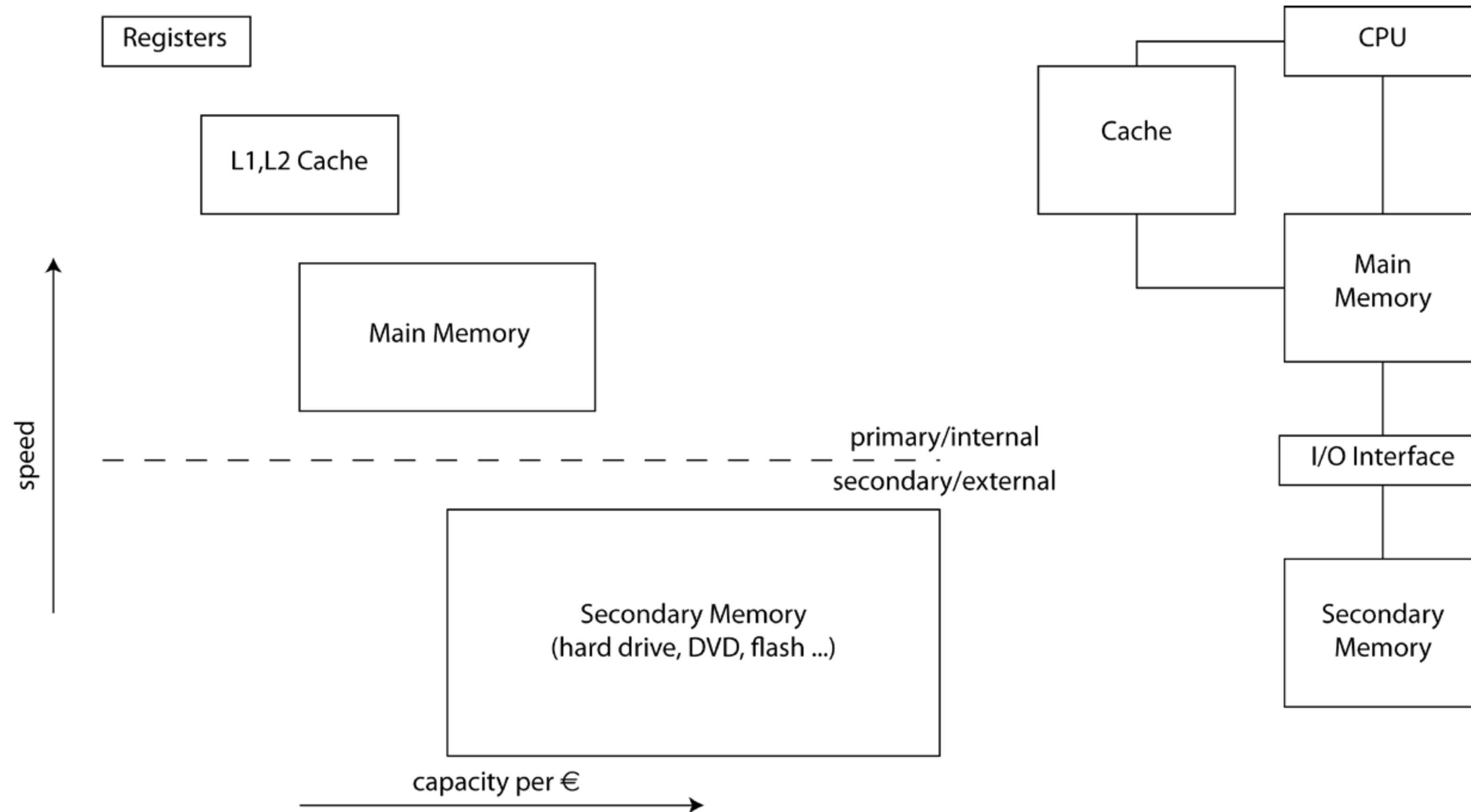
- I praksis må man velge ut fra flere hensyn.

- Tre parametre klassifiserer minnetyper:
  - Antall byte tilgjengelig per enhet
  - Pris per byte
  - Aksesshastighet

Minnehierarki



Minnehierarki



Minnehierarki – Bruksområder

- Register

  Intern kladdeblokk for CPU´en med rask aksess til innholdet.

- Cache

  Hurtig mellomlager for både instruksjoner og data for å jevne ut hastighetsforskjellen mellom CPU´en og hurtigminnet.

- RAM

  Buffer mellom eksternt lagringsmedium og CPU´en med rask både lese- og skrive aksess.

- SSD/Flash/Harddisk

  Høykapasitetsmedium for program/data

Minnehierarki – Lagringskapasitet

- Register

  Integrert på CPU´en, relativt få (32-128 stykker)

- Cache

  Mellomlager internt (L1) eller i nærheten (L2,L3) av CPU´en, typiske kapasiteter er fra 10 KiloByte (L1) til 1 MegaByte (L2) og flere MegaByte (L3).

- RAM

  Internt på hovedkortet i nærheten av CPU´en, størrelser opptil flere GigaByte.

- SSD/Flash/Harddisk

  Ekstern eller intern lagringsenhet i maskinen med kapasitet opptil flere TeraByte.

Minnehierarki – aksesshastighet

Registers	< 1ns	≈ 100 Byte
L1 (på CPU) cache	≈ 1ns	≈ 10 KB
L2,L3 (utenfor CPU) cache	2-10ns	≈ 1 MB
Hovedminne (RAM)	20-100ns	≈ 1 GB
SSD/Flash	100ns-1us	≈ 1 TB
Harddisk	1ms	≈ 1 TB

Cache

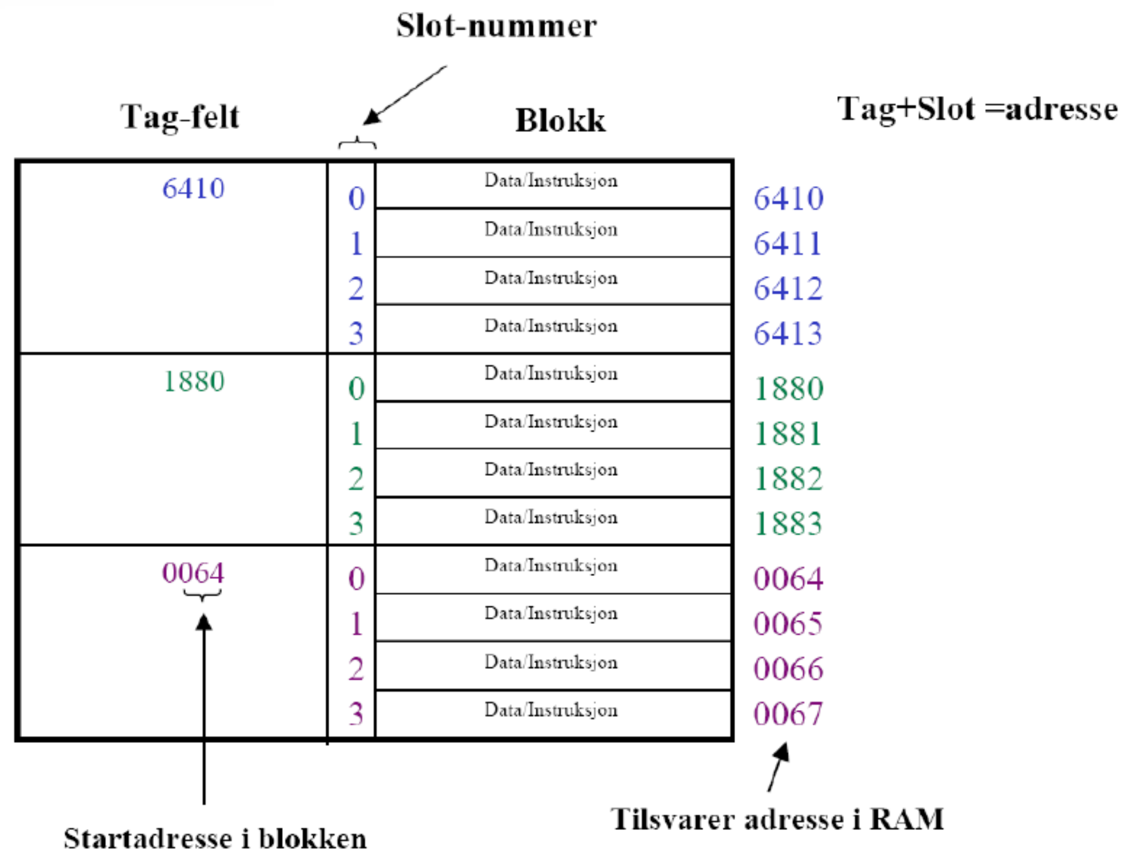
- Cache forbedrer von Neumann arkitekturens flaskehals med hensyn på minneaksessering.
- Cache ligger nærme CPU for å øke hastighet, men små størrelser for å redusere produksjonskost og derfor er det et trade-off diskusjon mellom størrelse og hastighet
- Cache inneholder en kopi av et subsett av hurtigminne (RAM)
- CPU´en henter data og/eller instruksjoner fra cache istedenfor RAM.

Cache

- Siden cache er mindre enn RAM, må det bestemmes hvilke data/instruksjoner som skal ligge i cache

- Cache basere seg på at instruksjoner/data aksesseres:
  - Nær hverandre i tid
  - Nær hverandre i rom

Cache



Cache

- Hvis data/instruksjon CPU'en trenger ligger i cache, kalles det for ***cache hit.***

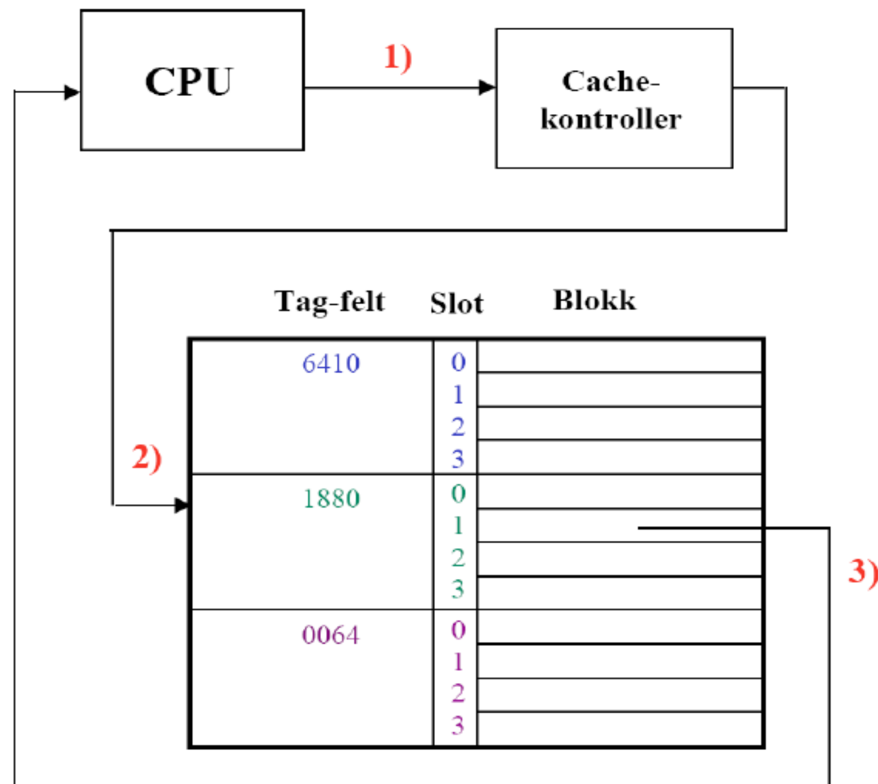
- Hvis data/instruksjon CPU'en trenger ikke ligger i cache, kalles det for ***cache miss.***

- En essensiell del for utnyttelsen av cache er å kunne raskt sjekke for hit eller miss.

Cache hit

- Når en prosessor skal hente en instruksjon eller lese/skrive data, vet den ikke om den skal hente fra RAM eller cache. Hvis det prosessoren ber om ligger i cache, kalles det en cache hit

Eks: hent innholdet
i loaksjon 1881.



Cache miss

- Ved ***cache miss*** må data/instruksjon først kopieres fra hurtigminnet over i cache før CPU'en kan bruke det.

- Cache fylles fra hurtigminnet første gangen en bestemt lokasjon blir referert

Cache miss

Før referanse til Xn
X4
X1
Xn – 2
Xn – 1
X2
X3

Før referanse til Xn

Etter referanse tilXn
X4
X1
Xn – 2
Xn – 1
X2
Xn
X3

Etter referanse tilXn

Cache miss

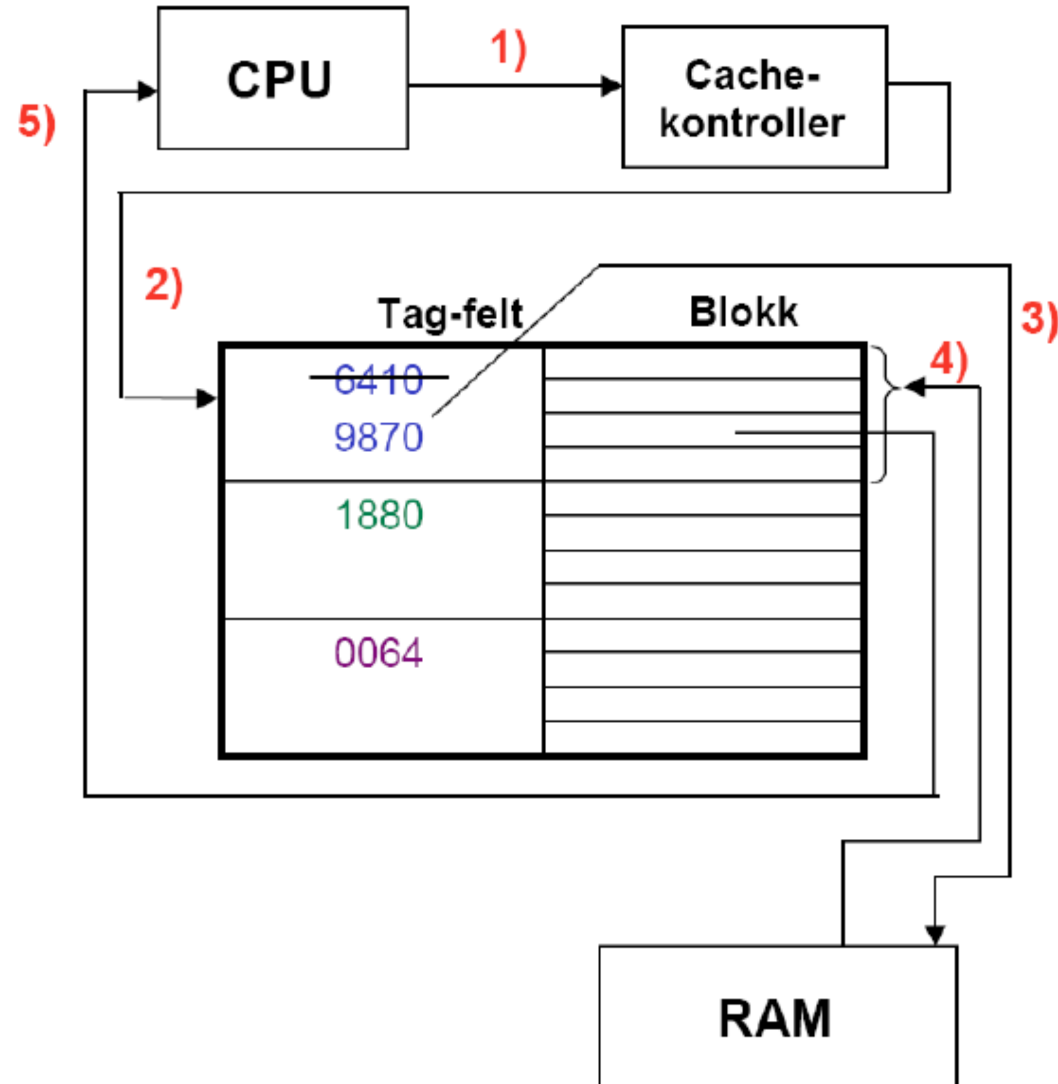
- Hvis data ikke ligger i cache, får man en cache miss

(1) CPU ber om innholdet i lokasjon 9872

(2) Cachekontrolleren finner ut at denne ikke finnes i cachen

(3) Blokken som starter på 9870 leses inn fra RAM

(4) Blokken som starter på 6410 overskrives for å gi plass til 9870



Indeksering av cache og blokkoverskrivning

(1) **Hvordan** cache kontrolleren indekserer (dvs. leter fram) riktig blokk ved lesing/ skriving

(2) **Hvilken** blokk som skal overskrives(kastes ut når cache er full

Mapping strategier for cache

1. Direct-mapped cache
2. Set-associative cache
3. Full associative cache

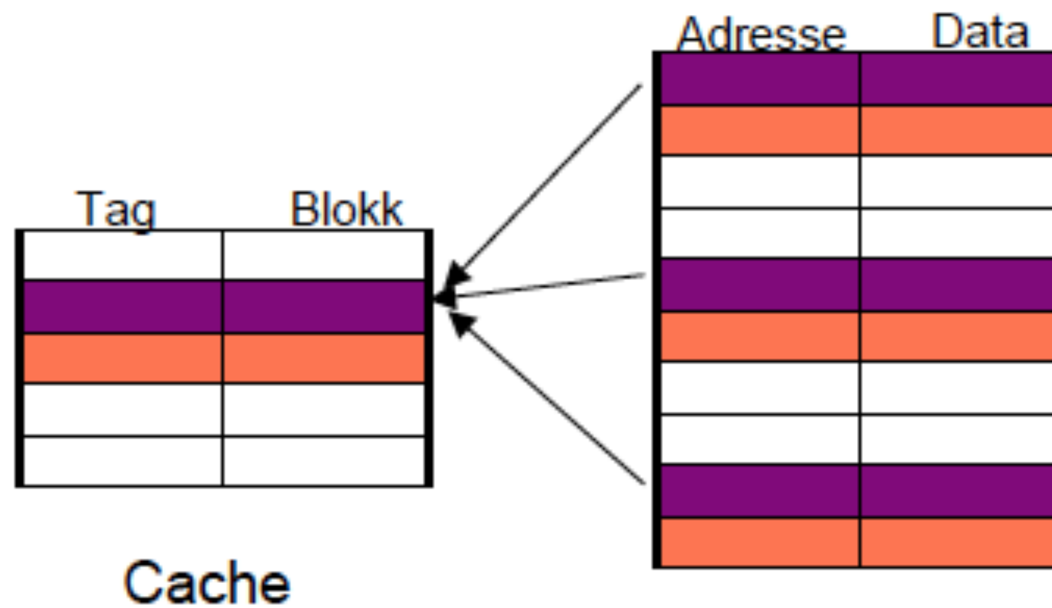
Metodene skiller seg fra hverandre ved:

- Hvor enkle de er å implementere
- Hvor raskt det går å finne en blokk
- Hvor god utnyttelse man får av cache
- Hvor ofte man får en cache miss

Direct-mapped Cache

- En bestemt blokk fra RAM kan bare plasseres i en bestemt blokk i cache. Flere RAM-lokasjoner må “konkurrere” om samme blokk i cache.
  - Fordel: Lett å sjekke om riktig blokk finnes i cache: Sjekk kun ett tag-felt og se om denne inneholder blokken man leter etter.
  - Ulempe: Høy miss-rate (selvom det er ledig plass andre steder, kan en blokk kun plasseres ett bestemt sted)

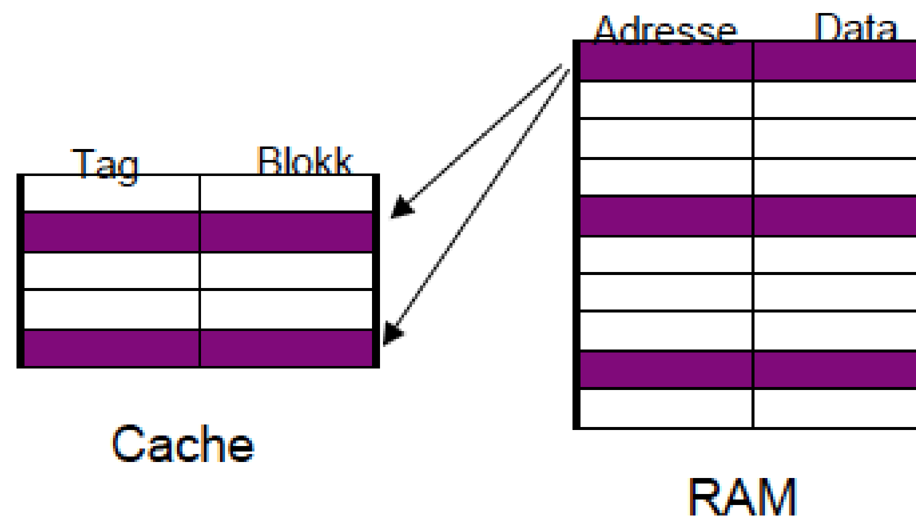
Direct-mapped Cache



Set-assosiative

- En bestemt blokk fra RAM kan plasseres i et begrenset antall blokk-lokasjoner i cache.
  - Fordel: Lett å sjekke om riktig blokk finnes i cache: Søk gjennom et begrensset antall tag-felt og se om blokken man leter finnes i cache.
  - Ulempe: Lenger søketid enn ved direkte-mappet (med mindre man søker i parallell gjennom tag-feltene)

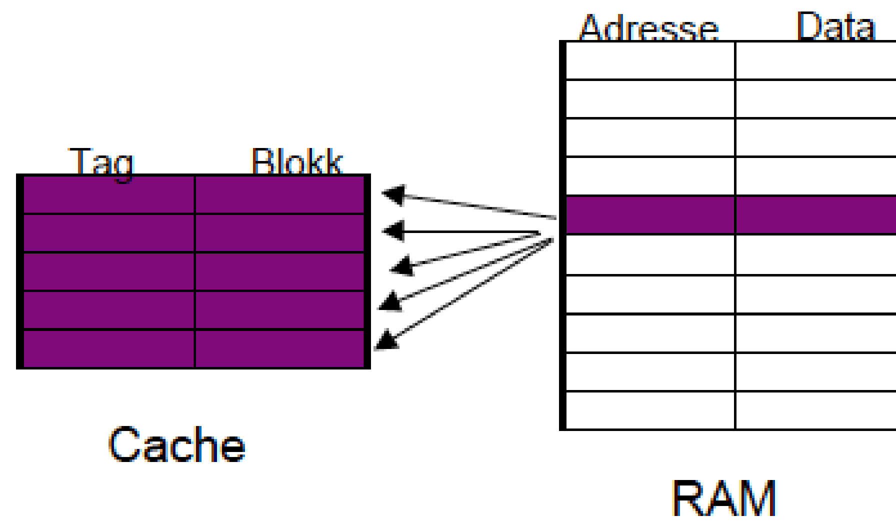
Set-assosiative



Full assosiative

- En bestemt blokk fra RAM kan plasseres hvor som helst i cache
  - Fordel: Cachen utnyttes meget godt, har minst sjanse for cache miss av alle de tre metodene
  - Ulempe: Søket for å finne en blokk kan bli tidskrevende. Man kan lage mekanismer for å forenkle søk, f.eks. hashing, men dette kompliserer cache-kontrolleren og krever ekstra hardware.

Full-assosiative



Write strategier

- **Write hit**
  - Det skal skrives til hukommelsen og blokken med lokasjonen det skal skrives til ligger i cache
  - Data skrives til riktig lokasjon i blokken i cache
- **Write miss**
  - Det skal skrives til hukommelsen og blokken med lokasjonen det skal skrives til ligger ikke i cache
  - Cache-kontrolleren må bestemme
    - Hvilken blokk i cache som kan/skal overskrives
    - Kopiere inn riktig blokk fra RAM til cache
    - Besørge skriving til riktig lokasjon i cache
- Ved begge write-operasjoner vil ikke innholdet i RAM og innholdet i kopien av blokkene i cache være identiske -> **ikke OK**

Write inkoherens

- Hvis det bare skrives til cache og ikke RAM, vil ikke RAM inneholde gyldige data
- Hvis en blokk det er skrevet til kastes ut fordi cache er full, mister man data
- To strategier benyttes for å håndtere write-operasjonen korrekt:
  1) Write-through
  2) Write-back

Write-through

- Etter hver gang det skrives til en blokk, skrives innholdet i blokken også tilbake til RAM
  - **Fordel:**
    - Enkelt å implementere i cache-kontrolleren
  - **Ulempe:**
    - Hvis det skal skrives flere ganger rett etterhverandre til samme blokk må prosessoren vente mellom hver gang

Write-back

- Innholdet i en blokk i cache kopieres kun tilbake til RAM hvis blokken skal overskrives i cache og det har blitt skrevet til den i cache

- For raskt å detektere om en blokk har blitt skrevet til benyttes et kontroll-bit i cache (kalles dirty-bit'et)

  – **Fordel:**
    - Gir ingen hastighetsreduksjon ved flere påfølgende write-operasjoner til samme blokk siden CPU'en ikke trenger vente mellom hver skriving

  – **Ulempe:**
    - Hvis arkitekturen støtter direkte lesing/skriving mellom RAM og Input/Output-enheter uten å gå via CPU'en (kalt Direct Memory Access eller DMA), risikerer man inkonsistente data hvis ikke DMA-kontrolleren kommuniserer med cache-kontrolleren

Replacement strategy

- Ved en cache miss må noen ganger en eksisterende blokk kastes ut for å gi plass til en ny blokk. Hvilken blokk som fjernes kan ha stor betydning for hastigheten til programmet som eksekveres.

- First-in-First-out (FIFO)
  - Blokkene er organisert som en ringbuffer.
  - Blokken som har ligget lengst i minne blir overstrevet (uavhengig om den er mye brukt)

- LRU (Least Recently Used):
  - Blokken som har ligget lengst i cache uten å ha blitt skrevet til eller lest skrives over, pga. Lokalitetsprinsippet
  - Ren LRU benyttes sjelden fordi det medfører mye administrasjon som i seg selv er tidkrevende (bl.a må et tidsstempel oppdateres hver gang en blokk aksesseres)

- Random:
  - Kaster man ut en tilfeldig blokk når man trenger å frigi plass til en blokk.

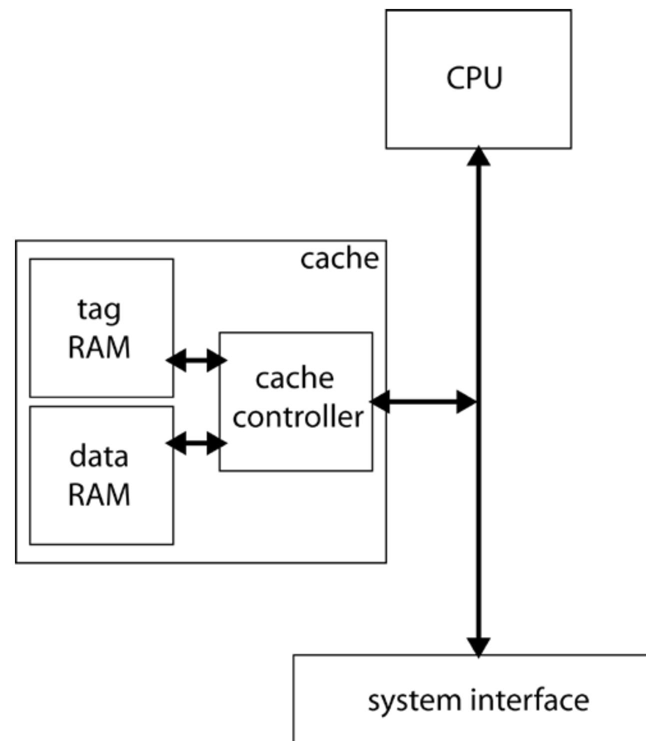
- Hybrid:
  - Deler inn blokker i tidsgrupper, og kaster så ut en tidfeldig valgt fra den gruppen som har ligget lengst i cache uten å bli brukt.

Arkitektur

- Cache kan plasseres på (minst) to måter i forhold til CPU og RAM
  (1) Look-aside
  (2) Look-through
- De to organiseringene påvirker hvordan lesing håndteres (”read architecture”)
- ”Read architecture” og ”write policy” er det viktigste karakteristika til cache
- Cache koherens: Hvorvidt innholdet i cache og RAM er identisk
- For å bevare koherens kan cache gjøre
  - ”Snoop”: Cache overvåker adresselinjer for å se etter transaksjoner som angår data den sitter på
  - ”Snarf”: Cache tar data fra datalinjer og kopierer dem inn
- Inkoherens kan være
  - ”Dirty data”: Data i cache er modifisert, men ikke tilsvarende data i RAM
  - ”Stale data”: Data i RAM er modifisert, men ikke tilsvarende data i cache.

Look-aside arkitektur

- Cache er plassert med RAM
- Både RAM og cache leser adresselinjene samtidig

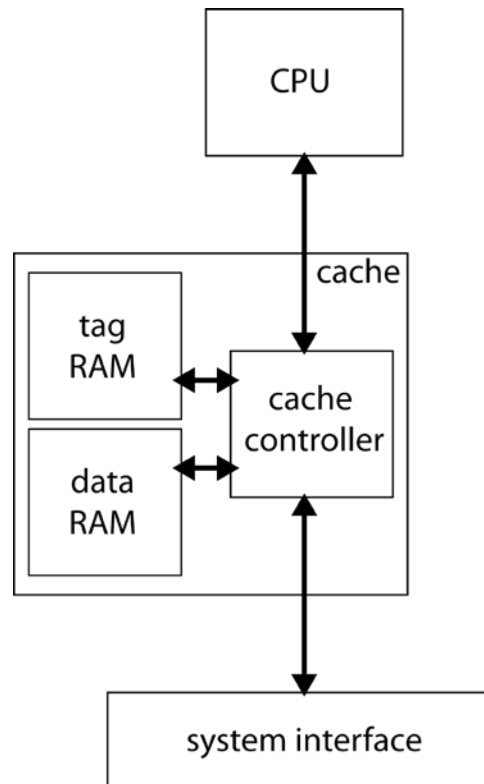


Look-aside arkitektur

- Når prosessorer starter en lesing, sjekker cache adressen (snooping) for å se om det er en ”read hit”
- Hvis ”read hit”:
  - Cache svarer prosessoren og stopper videre buss-syklus for å hindre lesing fra hovedminnet (RAM)
- Hvis ”read miss”:
  - Hovedminnet (RAM) vil besvare henvendelsen fra prosessoren. Cache ”snarfer” data slik at de er i cache neste gang prosessoren ber om dem.
- Fordeler:
  - Mindre komplisert enn look-thorugh
  - Bedre responstid (RAM og CPU ser samme buss-syklus)
- Ulempe
  - Prosessoren kan ikke aksessere cache hvis en annen enhet aksesserer RAM samtidig

Look-through

- Cache sitter fysisk mellom CPU og internbussen
- Cache-kontrolleren vil avgjøre om buss-syklus skal slippe videre til RAM



Virtuell Minne

- Lagringskapasiteten i RAM må deles mellom flere ulike prosesser:
  - Operativsystemet
  - Driver-rutiner
  - Bakgrunnsjobber (eks utskrifter)
  - Brukerprogrammer som kjøres samtidig
  - Ulike brukere som deler samme RAM

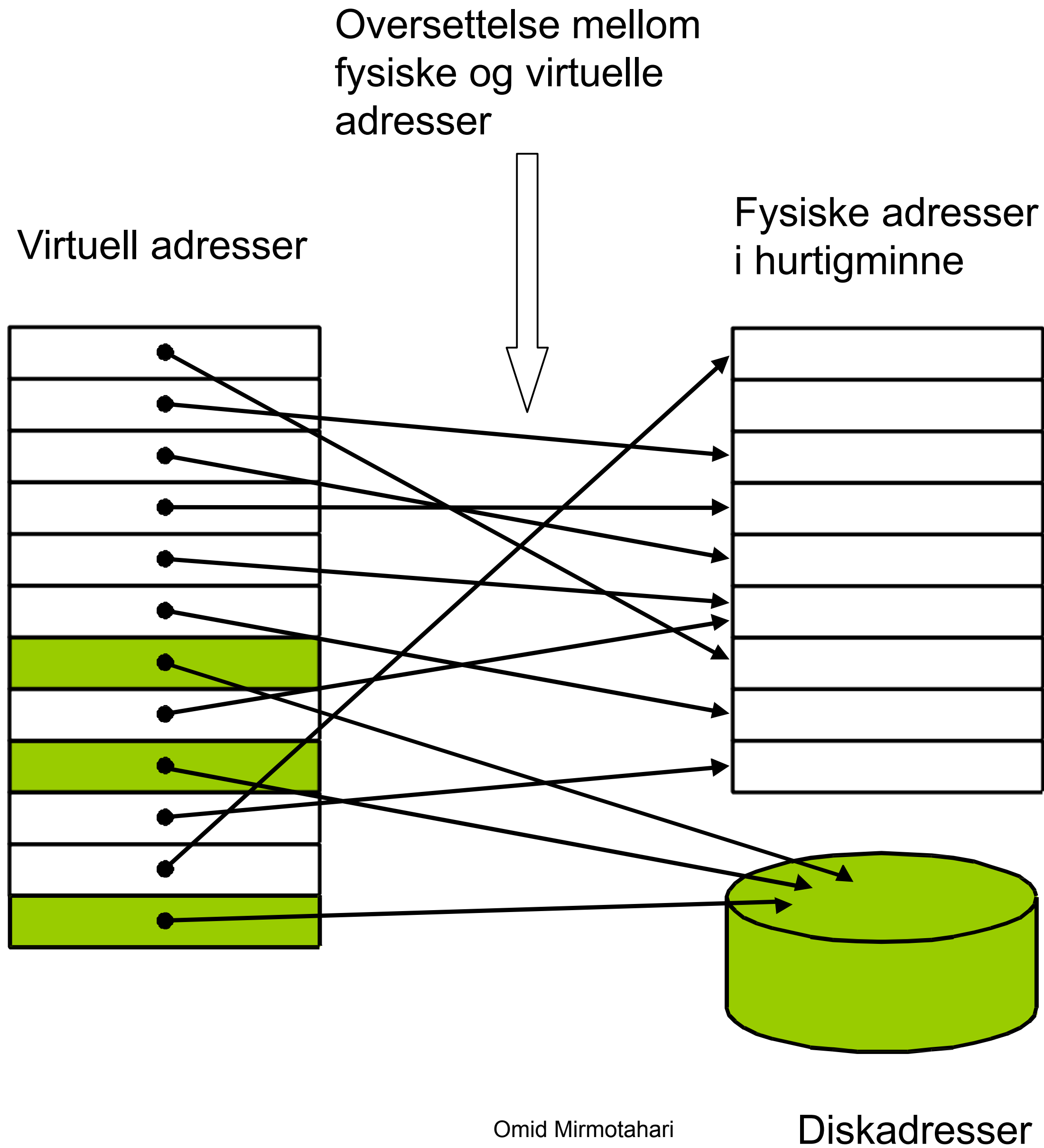
- Dette får to konsekvenser:
  - Hver prosess har mye mindre hurtigminne til rådighet enn den totale kapasiteten.
  - De ulike prosessenes minneområder må beskyttes mot overskrivning av andre prosesser.

- Virtuelt minne løser disse to problemene ved å
  - Gi prosesser tilgang til nesten ubegrenset minne ved å bruke deler av harddisken som hurtigminne (dvs. RAM).
  - Holde styr på hvilke prosesser som får lov til å skrive til hvilke områder i det virtuelle minnet.

- Virtuelt minne implementeres som en del av operativsystemet.

- En prosess bruker virtuelle adresser som så oversettes til fysiske adresser.

- Et program må ha relokerbar kode


Oversettelse mellom fysiske og virtuelle adresser
Virtuell adresser
Fysiske adresser i hurtigminne
Diskadresser

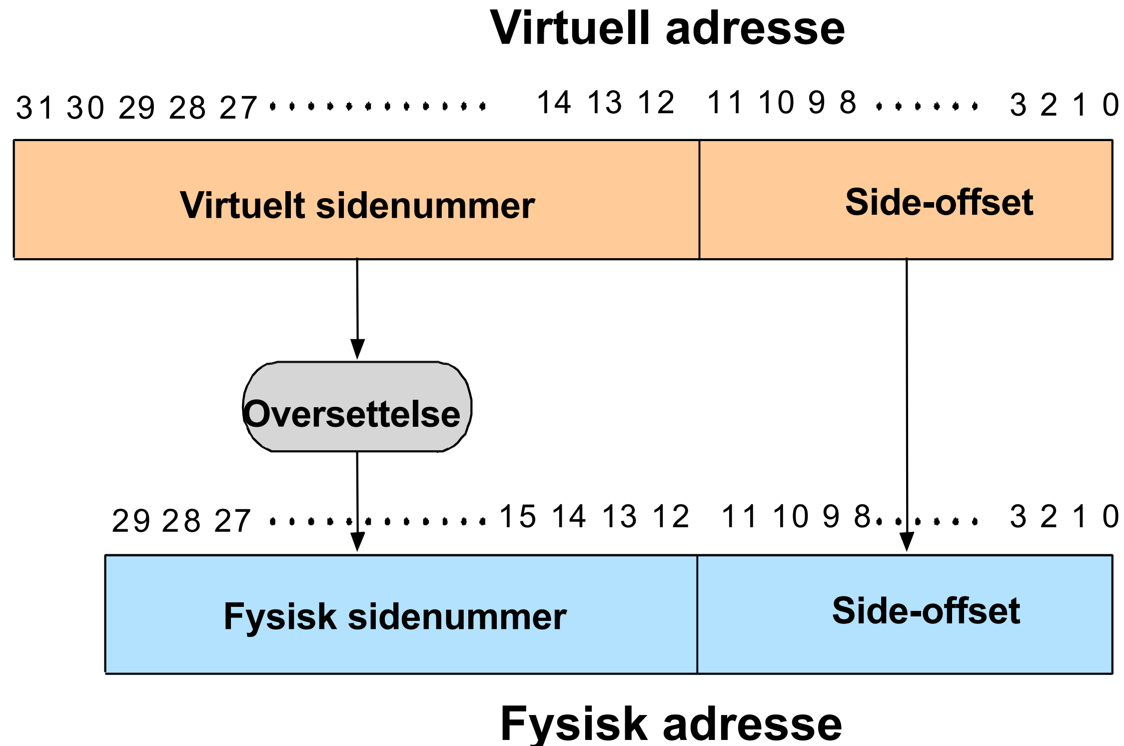
- Operativsystem-delen som håndterer virtuelt minne må ta seg av flere oppgaver:
  - Plassering i hurtigminne (RAM).
  - Plassering på harddisk.
  - Konsistens mellom data i hurtigminne og på harddisk ved skriving
  - Flytting mellom harddisk og hurtigminne (begge veier) hvis data ikke ligger i hurtigminne.

- Det finnes to hovedtyper virtuelt minne:

  - Paging: Det virtuelle minnet deles opp i blokker med fast størrelse og et én-dimensjonalt adresserom.

  - Segmentering: Det virtuelle minnet deles opp i blokker med variabel størrelse (uavhengig segmentnummer og offset).

- Avbildning fra fysiske til virtuelle adresser ved paging:



- Sidestørrelsen er på 2^{12} = 4 KB
- Maksimum antall sider i fysisk minne er 2^{18}
- Det fysiske adresserommet er maksimalt 1GB.
- Det virtuelle adresserommet er maksimalt 4 GB

- Kostnaden ved ’page fault’ (dvs at en side ikke finnes i hutigminnet, men på disk) er svært stor.

- Dette får følgende konsekvenser:
  - Sidestørrelsen må være tilstrekkelig stor (> 4 KB).
  - Bruker full asossiativ plassering av sider.
  - Bruker SW-agoritmer for å bestemme plasseringen.
  - Bruker ’write-back’ istedenfor ’write-through’.

Plassering av sider og gjenfinning av data

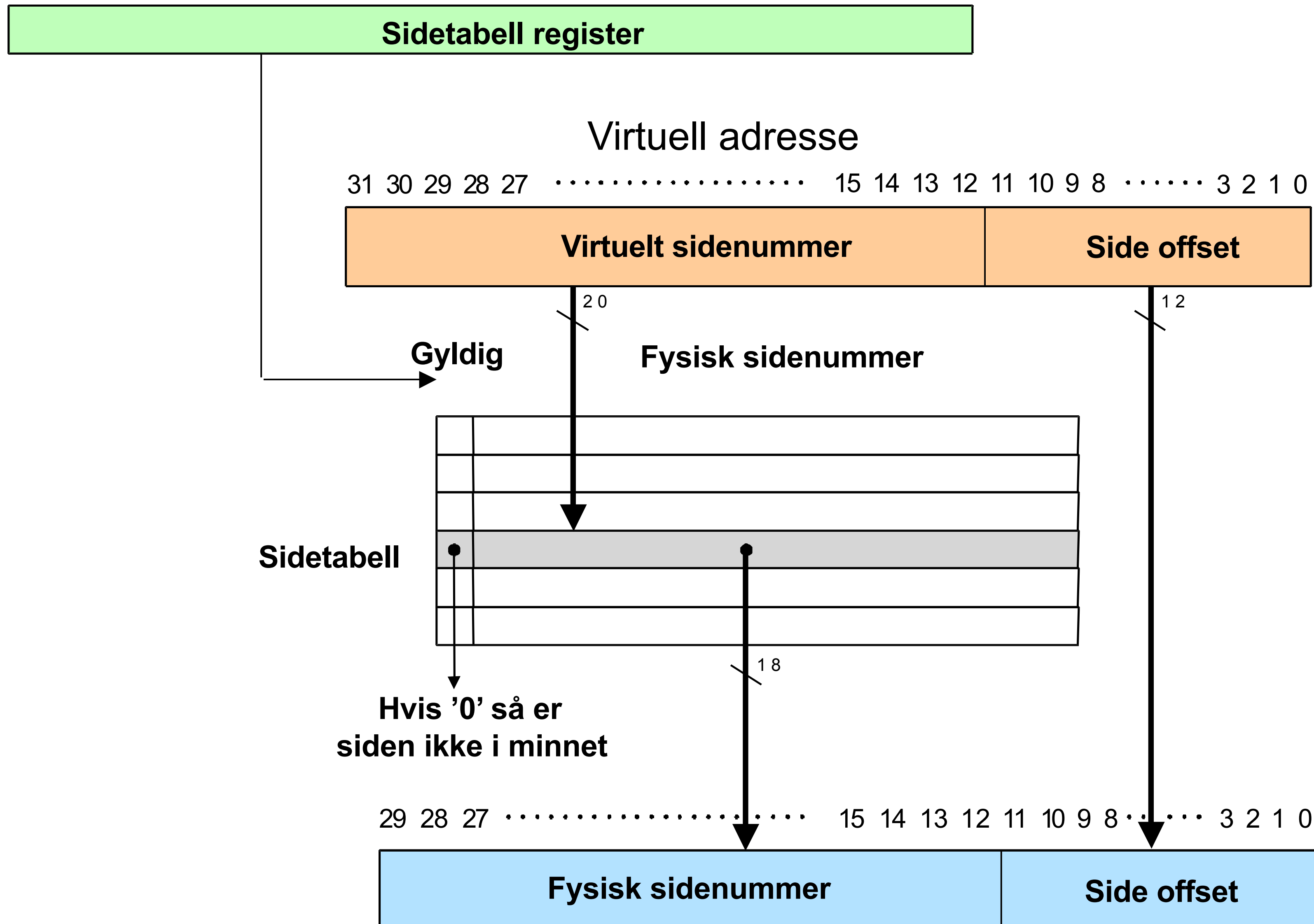
- For å redusere sjansen for 'page fault' brukes full assosiativ plassering.

- Når en ny blokk hentes inn fra harddisken kan den plasseres på en vilkårlig ledig plass.

- For å finne og plassere en bestemt blokk bruker man en komplett *sidetabell* ('page register'), slik at man slipper et fullt søk hver gang.

- Sidetabellen gir opplysning om
  - Hva den fysiske adressen er gitt den virtuelle
  - Informasjon om innholdet på en gitt fysisk adresse tilsvarer den virtuelle.

- Sidetabellen ligger i en egen enhet som kalles MMU (Memory Management Unit).


Sidetabell register
Virtuell adresse
31 30 29 28 27 .... 15 14 13 12 11 10 9 8 ...... 3 2 1 0
Virtuelt sidenummer
Side offset
20
12
Gyldig
Fysisk sidenummer
Sidetabell
18
Hvis '0' så er
siden ikke i minnet
29 28 27 .... 15 14 13 12 11 10 9 8 ...... 3 2 1 0
Fysisk sidenummer
Side offset

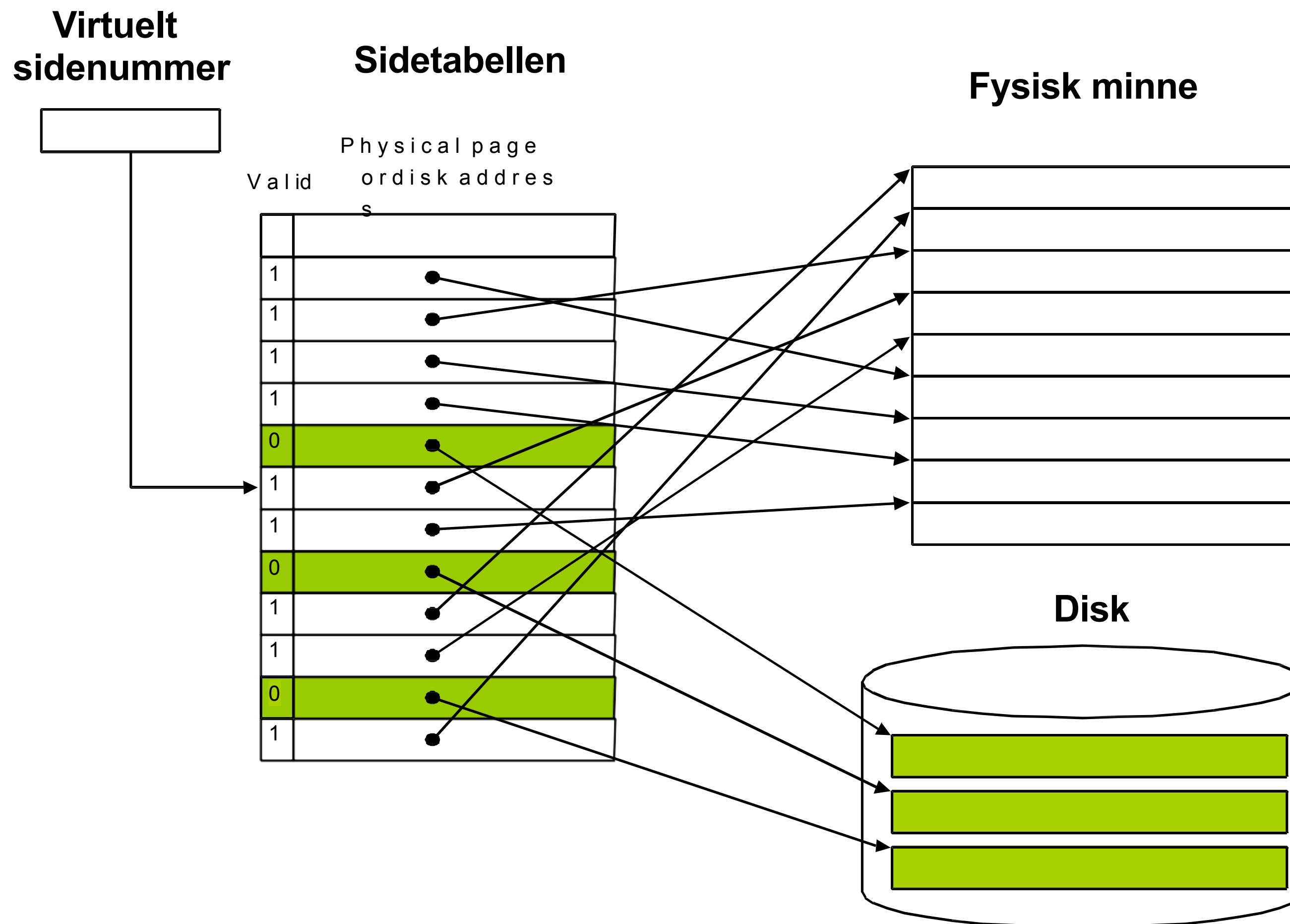
- Hvis riktig side ikke finnes i det fysiske minnet (dvs. 'page fault') skjer følgende:

1) Operativsystemet tar over kontrollen.

2) En intern datastruktur aksesesere for å finnes siden på harddisken.

3) Sidetabellen oppdateres.

4) Det fysiske minnet oppdateres med riktig innhold fra harddisken.


Virtuelt sidenummer
Sidetabellen
Fysisk minne
Physical page ordisk addres s
Valid
1
1
1
1
0
1
1
0
1
1
0
1
Disk

- Hvis det fysiske minnet ikke inneholder noen ledig plass, må en side fjernes for å få plass til en ny.
- Siden som erstattes, må skrives tilbake til disk.
- Mest vanlig er å erstatte den siden som har ligget lengst i fysisk minne *uten å bli brukt* (Least Recently Used, LRU).
- LRU implmenteres ved et eget bit som settes når blokken aksesseres, og som slettes ved jevne mellomrom av opertivsystemet.
- Operativsystemet inneholder en tabell sidene sortert etter LRU (kan også sorteres etter andre kriterier, f.eks FIFO).

- Samme problem oppstår ved skriving til fysisk minne som ved skriving til cache.
- ’Write-through’ er upraktisk fordi skriving til harddisk er meget langsomt.
- Mest vanlig er ’write-back’ som oppdaterer siden på harddisken kun ved en ’page fault’
- En blokk som erstattes trenger kun å skrives til harddisk hvis innholdet er modifisert (unødvendig hvis f.eks blokken kun inneholder instruksjoner)
- Et eget bit, ’dirty bit’, settes hvis siden er skrevet til etter at den ble hentet fra harddisken.

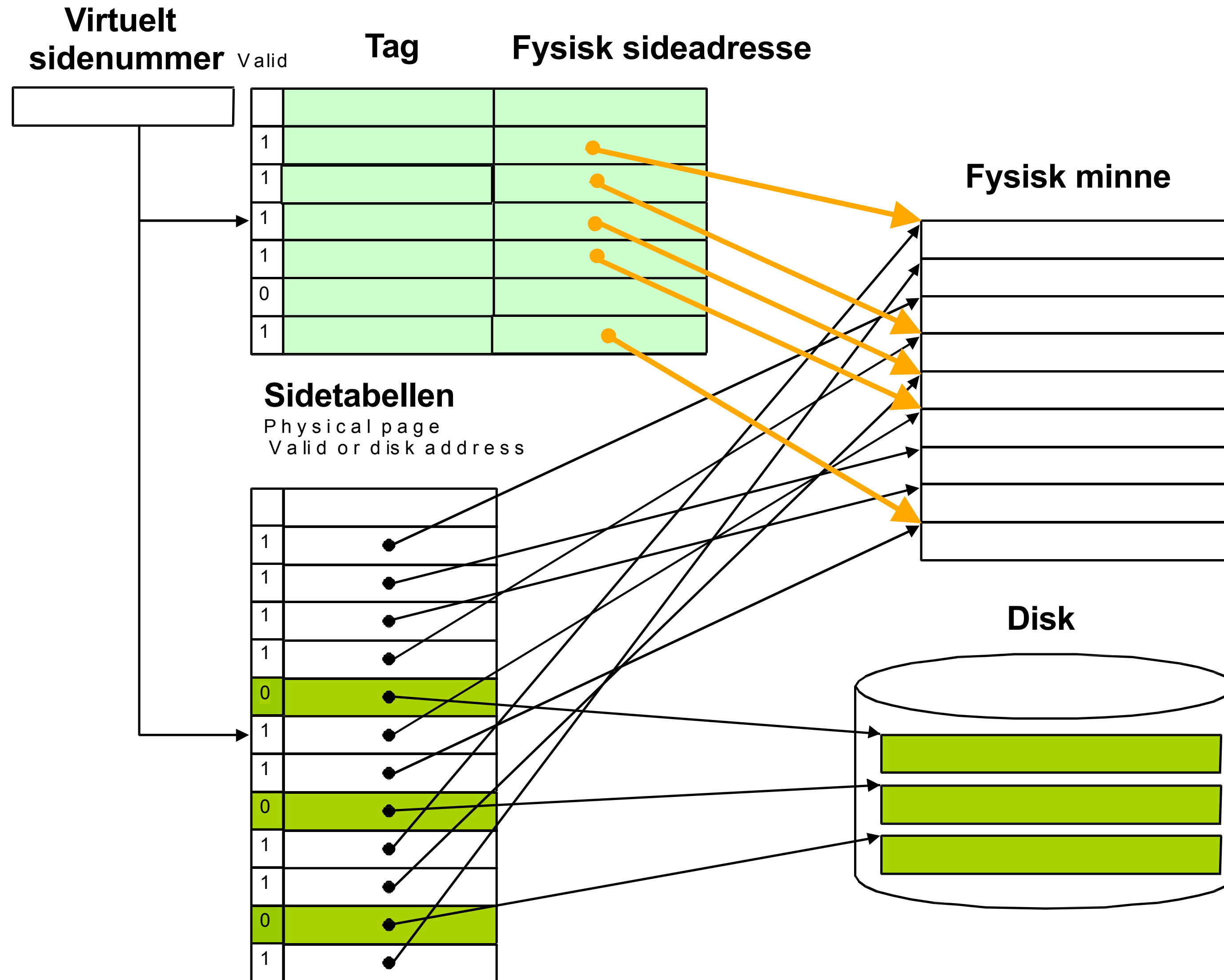
- Virtuelt minne kan føre til at minneaksess tar dobbelt så lang tid:
  1) Slå opp i sidetabellen.
  2) Les fra fysisk minne.

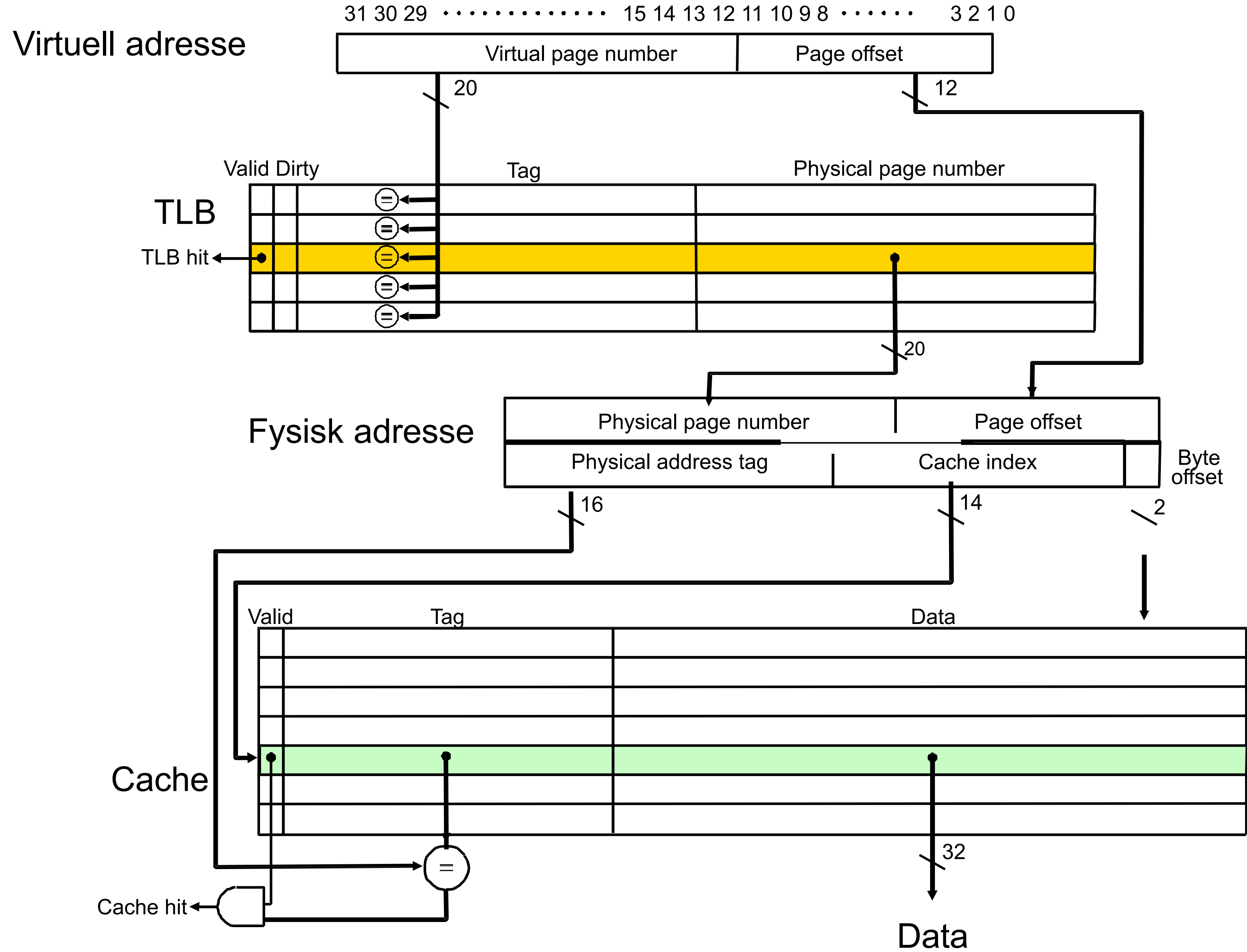
- Ved å benytte en egen cache, TLB (Translation Lookaside Buffer), for de siste oppslagene, slipper man oppslag i sidetabellen.

- Tag-feltet i TLB inneholder deler av et virtuelt sidenummer, mens datafeltet inneholder det fysiske sidenummeret.

- Ved hver minne-referanse gjøres oppslag i TLB:
  1) Hvis lokasjonen er indeksert av TLB, hentes innholdet direkte.
  2) Hvis lokasjonen ikke finnes i TLB, må det først sjekkes om den finnes i fysisk minne.
  3) Finnes den i fysisk minne, oppdateres TLB og man fortsetter fra 1).
  4) Hvis lokasjonen verken finnes i TLB eller fysisk minne, genereres en 'page fault', og deretter går man videre fra 2).

TLB




31 30 29 .... 15 14 13 12 11 10 9 8 ...... 3 2 1 0
Virtuell adresse
Virtual page number
Page offset
20
12
Valid Dirty
Tag
Physical page number
TLB
TLB hit
20
Fysisk adresse
Physical page number
Page offset
Physical address tag
Cache index
Byte offset
16
14
2
Valid
Tag
Data
Cache
32
Cache hit
Data