1. Cache

CPU er så kjapp i forhold til primærminnet at vi trenger et raskere minne hvis instruksjoner skal leveres i samme hastighet som prosessoren utfører de. Cache kalles ofte en hurtigbuffer. Aksesstiden på cache er så kort at det holder tritt med CPUen.

**Lokalitet**

*Prinsipp*: dersom en minnelokasjon er benyttet en gang, er det svært sannsynlig at den- eller en lokasjon like ved siden av- snart vil bli benyttet gang til.

* Derfor overføres instruksjoner i blokker, altså flere instruksjoner på en gang.
* Hvis prosessoren først må hente noe i primærminnet kan det hente flere instruksjoner som en blokk.
* Disse instruksjonene er de som ligger nært den instruksjonen som skal utføres i prosessoren.

*Romlig lokalitet*: minnereferansene ligger nært hverandre. Eks. sekvensiell utføring.

*Temporal lokalitet*: minnereferanse gjerne brukes gjentatte ganger. Eks. løkker (while og for).

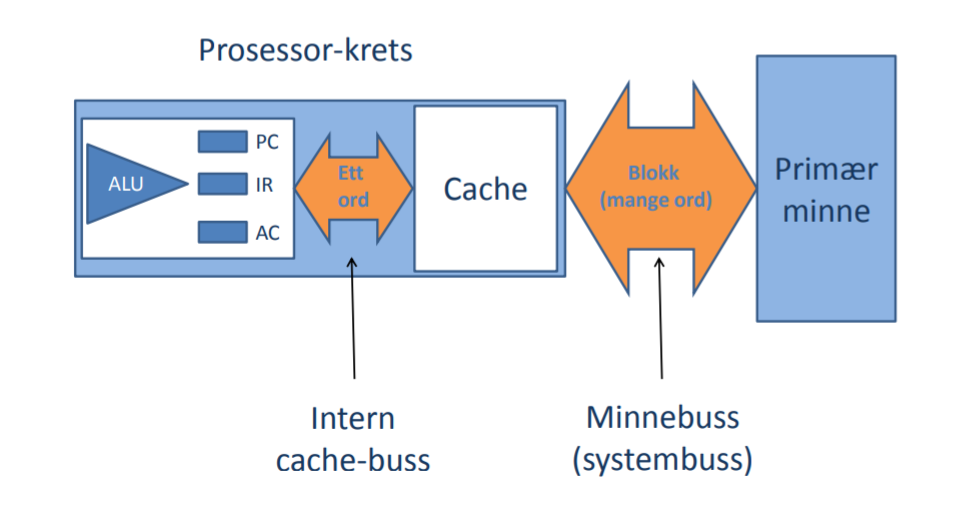
**Virkemåte til cache**

Før prosessoren skal lese en instruksjon fra primærminnet sjekker den om den ligger i cachce, siden denne er mye raskere enn primærminnet. Hvis instruksjonen ikke er kopiert til cache, vil den hente en blokk med instruksjoner (f.eks. 64 bytes) og legger den i cache på grunn av lokalitet prinsippet. Cache er oppbygget av statisk ram (SRAM), med aksesstid 1-20ns, i forhold til primærminne med DRAM som har 40-70ns aksesstid.

1. Nå med cache ser vi hovedsakelig på primærminnet som en samling av blokker. Det er disse blokkene som blir overført i cache. Vi nummerer blokkene fra 0 og oppover. Hvis en instruksjon skal hentes i minnet, hentes hele blokken og legges i cache.
2. Cachen er liten, men har plass til mer enn en blokk, derfor har den linjer. Hver linje inneholder en blokk.
3. Hver linje har også en «tag» som er en identifikator som sier hvilken blokk som ligger på hver enkelt linje i cachen.

**Plassering**

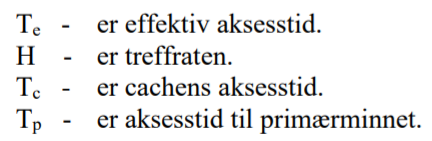
Cache må være så rask som mulig for at den skal holde følge med prosessoren, derfor ligger den så nært som mulig prosessoren. I moderne prosessorer er cache en integrert del av prosessorkraften.



**Treffrate**

Siden noen instruksjoner ikke vil ligge i cachcen vil prossesoren måtte vente på primærminnet. Andelen av aksessene som finner data i cachen kalles treffrate, denne er vanligvis 80-98%.

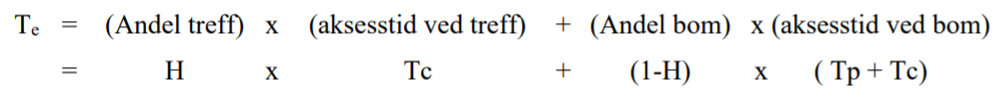
0 treffrate: kalles bom



Aksesstid ved treff: TC

Aksesstid ved bom: (TP+TC)

Bomraten: (1-H)

*Treffraten:* 



**Blokkstørrelse**

Vil variere etter forhold til hva som er mest optimalt. Kan deles inn i lite antall store blokker eller motsatt.

* Programmer med stor grad av romlig lokalitet vil det lønne seg med store blokker, for å holde blokka lengst mulig i cachen.
* Ved bom med store blokker vil det ta lengre tid å overføre fra minnet enn ved mindre blokker.
* Ofte er forskjellen mellom 32 og 64 bytes.

**Mapping funksjonen**

Når en blokk kopieres til cache må den legges på en linje, hvor den skal ligge avgjør mapping funksjonen.

Hello in traffic man Open my apps yeah. The network there Hey Cortana open new phone on my way caught behind off meet are you crazy call me right on

*Full-assosiativ cache:*

En blokk kan legges i en hvilken som helst linje. Derimot når CPU skal hente data, må den sjekke hver eneste tag i cachen for å finne blokken. Dette er tidskrevende og krever komplisert elektronikk. Derfor brukes ikke dette i praksis. Prosessorer opererer i nanosekunder, derfor vil den minste sammenligning i denne grad gjøre at cachen blir bortkastet i første plass.

*Ikke-assosiativ cache (direkte mapping)*

For å slippe å sammenligne, kan man legge en blokk på en bestemt plass i cachen. Det ligger flere blokker på hver linje, men prosessoren vet hvilken linje en bestemt blokk ligger, derfor trenger den bare å lete etter tagen til den blokken på den bestemte linjen, ikke alle.

*Trashing*

Ulempen er at to ulike ord som brukes samtidig kan komme til å havne opp i samme linje. For eksempel kan det hende at rett etter at en instruksjon er hentet inn må den vike plassen igjen for en data-verdi som havner i samme linje. Hvis dette skjer ofte kan verdien av caching reduseres betydelig. Denne situasjonen hvor to blokker gjensidig tvinger hverandre ut av cache hele tiden kalles trashing.

*Sett-assosiativ cache*

Løsningen på trashing.

**Utskiftingsalgoritmer**

Metoder for å velge mellom de ulike linjene der data fra minnet kopieres til cache. Disse må være enkle og raske for å ikke forsinke CPU.

**LRU**: *Least Recently Used.* Bytter ut den blokka som det er lengst siden ble aksessert. Dette gjøres ved å bruke en teller som økes for hver minne-aksess, linjen med høyest verdi offres først.

**FIFO**: *First In First Out*. Bytter ut den blokka som har være lengst i cachen. Telleren økes kun når en ny blokk hentes fra minnet. Nye blokken sin teller settes til 0.

**LFU**: *Least Frequently Used*. Bytter ut den blokka som har vært aksessert færrest antall ganger. Telleren økes kun for referanser til dette ordet. Etter en stund blir disse verdiene tilbakestilt.

**Random**: Vilkårlig utskifting. En tilfedlig blokk velges.

**Skriving til minnet**

*Problem: CPU vil endre verdien i en lokasjon i minnet. Hva skal skje hvis en kopi av lokasjonen ligger i cachen?*

To forskjellige løsninger:

*Write through:* Endringen av verdien/skrivingen skjer samtidig i både cachen og minnet. Fordelen er at minnet er alltid oppdatert, ulempen er at dette kan forsinke CPU.

*Write back:* Endringen av verdien/skrivingen gjøres kun i cachen, minnet oppdateres først når innholdet av en cache-linje må vike plassen. Fordelen er redusert buss trafikk, ulempen er lengre ventetid for CPU ved utskifting av innhold i linje.

**Cache-koherens**

Noen ganger kan flere enheter endre innholdet i minnet. For eksempel, IO utstyr kan endre på innholdet i minnet, samtidig som innholdet som nettopp ble endret ligger i cache.

For å unngå at lokasjoner som er kopiert til en cache blir endret, må avanserte mekanismer til. Hvis innholdet i cachen samsvarer med minnet sier vi at de ikke er konsistent. Derimot samsvaret mellom minnet og cache kalles cache-coherence.

1. Moderne cache-design

**Flernivå caching**

Behoved for cache øker bare mer og mer siden forskjellen på prossesorens ytelse og minnets aksesstid vokser hele tiden. Derfor ønsker man å bruke større cache-størrelse, men samtidig opprettholde den effektive aksesstiden. Med større cacher trenger man mer elektronikk for å styre den, derfor kan ofte større cache være tregere enn mindre cache, selv om den er bygget opp med samme teknologi.

Løsningen på dette er å bruke det to cacher, en stor og en liten. Den lille raske cachen lagrer de meste brukte lokasjonene, og den store lagrer den mindre brukte. Den store cachen vil være tregere enn den mindre, men fremdeles raskerere enn primærminnet. Dette prinsippet kalles to-nivå-caching.

Lille cachen: *nivå 1 cache (L1-cache)*

Store cachen: *nivå 2 cache(L2-cache)*