DTK Høst 2018

Innholdsfortegnelse

[Datateknikk 2](#_Toc531290823)

[Datahistorikk 2](#_Toc531290824)

[Maskinklasser 3](#_Toc531290825)

[Tallsystemer 3](#_Toc531290826)

[Virkemåte 4](#_Toc531290827)

[Busser 8](#_Toc531290828)

[Parallelle busser 9](#_Toc531290829)

[Serielle busser 12](#_Toc531290830)

[Hovedbussen 12](#_Toc531290831)

[Svitsjede busser 13](#_Toc531290832)

[Primærminne 13](#_Toc531290833)

[Minnebrikkene 14](#_Toc531290834)

[IO 15](#_Toc531290835)

[Avbruddsystemet 16](#_Toc531290836)

[Flere typer IO 17](#_Toc531290837)

[DMA – Direkte minneaksess 18](#_Toc531290838)

[Sekundærlager 18](#_Toc531290839)

[Magnetiske media 18](#_Toc531290840)

[Optiske media 20](#_Toc531290841)

[Halvlederbaserte sekundærlager 21](#_Toc531290842)

[SSD disker 21](#_Toc531290843)

[Cache 22](#_Toc531290844)

[Virkemåte 23](#_Toc531290845)

[Mapping 24](#_Toc531290846)

[Moderne primærminne 27](#_Toc531290847)

[Timing parametere 28](#_Toc531290848)

[Brikkesettet 29](#_Toc531290849)

[Lagerhierarki 31](#_Toc531290850)

[Digital elektronikk 33](#_Toc531290851)

[CPUens deler og virkemåte 35](#_Toc531290852)

[Kontrollenheten 36](#_Toc531290853)

[Moderne prosessorarkitektur 37](#_Toc531290854)

[RISC og CISC 38](#_Toc531290855)

[Programmering 38](#_Toc531290856)

# Datateknikk

## Datahistorikk

Von Neuman var en av de første som kom opp med designet moderne PCer bruker i dag. Han foreslo en datamaskin arkitektur som inneholdt:

* ­­­Et arbeidslager som skulle inneholde data og instruksjoner, altså minnet
* En aritmetisk/logisk enhet som kunne utføre en del matematiske og logiske operasjoner
* En kontrollenhet som tolket instruksjonene i minnet og sørget for at de ble utført
* Inn/ut enheter som sørget for kommunikasjon med brukeren, tastatur og mus i dag

Denne modellen brukes av alle moderne datamaskiner også i dag selv om von Neuman skrev denne rapporten i 1945/46. Han fikk i etterkant kritikk fordi han ikke nevnte at denne arkitekturen ble foreslått av to av ENIACs konstruktører, John W. Mauchly og J. Presper Eckert jr. von Neuman lagde en maskin, IAS, med plass til 1000 ord i minnet, hver 40 bits. Senere har det blitt vanlig med ordbredde lik en potens av 2, for eksempel 8, 16 eller 32 bits.

Enhver datamaskin forholder seg til strøm, i med at den er elektronisk. Er maskinen analog, noe som er svært uvanlig, vil maskinen operere med glidende overganger mellom spenning, som betyr at den kan ha uendelig mange verdier mellom min og max spenning. Derimot er moderne maskiner digitale og operer bare med hele verdier. For å ha mest mulige distinktive verdier bruker man NULL eller FULL spenning. Altså ingen spenning, eller hvor stor nå enn spenningen er når den er FULL. På denne måten skiller man mellom av (0) og på (1). For å kunne slå av og på spenningen trenger vi en bryter. Man kan kanskje forestille seg en mekanisk bryter, men denne blir for treg. Man trenger en bryter som kan slå seg av og på flere hundre tusen ganger i sekundet. I dag bruker man derfor transistorer. Disse kan slå seg av og på ekstremt hurtig og man klarer i dag å lage disse atomisk små. Det finnes derfor flere hundre millioner av disse i en moderne datamaskin.

**Maskinklasser**

Datamaskiner deles ofte inn etter ytelse. Grensene mellom dem er riktignok ganske så flytende, men vi deler som oftest inn i stormaskiner, minimaskiner, og mikromaskiner. I tillegg har vi supermaskiner. Stormaskiner brukes til å betjene mange brukere. Disse er dyre fordi de krever drifting og store rom der de kan oppbevares. Databaser og servere er eksempler på slike. IBM er den største leverandøren av slike stormaskiner. Minimaskiner var mindre maskiner som ble brukt til blant annet forskning der man ikke trengte en stormaskin. Disse oppsto tidlig på 60-tallet da det ble mulig å fremstille små og billige maskiner. Minimaskiner ble senere utkonkurrert av mikromaskiner som er det vi i dag tenker på som PCer. Det som ble nytt for denne var at den samlet den aritmetiske/logiske enheten og kontrollenheten i en integrert krets, mikroprosessoren. Den siste typen, supermaskiner, er ikke generelle, og er produsert for å gjøre veldig spesifikke oppgaver. Nærmer bestemt vitenskapelige beregninger med høy presisjon. Disse er meget dyre og det finnes ikke mange av dem.

**Viktige historiske trekk**

Etter hvert som maskinene har blitt mindre og mindre og større og større kapasitet har bruken av maskinen endret seg drastisk. Utviklingen kan karakteriseres på følgende måte:

* «Mange brukere – en maskin» (Stormaskinen og minimaskinen 50, 60, 70-tallet)
* «En bruker – en maskin» (Mikromaskiner, PCens inntog, 80-tallet)
* «En bruker – mange maskiner (PC i nettverk, Internett fra 90-tallet)

Ikke bare har antall brukere per maskin endre seg, men bruksområdene til maskinen har endret seg. I dag brukes datamaskiner fra alt til musikk til video til avanserte utregninger.

**Moore’s lov**

Svært lite annet utvikler seg så hurtig som de elektroniske kretsene i datamaskiner. Prosessorene blir raskere og raskere og komponentene mindre og mindre. De siste tiårene har produsentene klart å fordoble regnekapasiteten i prosessorene sine med mellomrom på 18 måneder. Det vil si at prosessorkraften dobles ca. hvert andre år. Denne utviklingen kan fortsette en stund til inntil vi møter på fysiske begrensninger, for eksempel størrelsen på transistorene, som snart ikke kan bli særlig mindre. Moore’s lov sier følgende:

*Tilgjengelig maskinkraft per krone fordobles ca. hvert 1,5 år.*

## Maskinklasser

## Tallsystemer

I datamaskiner benytter vi et binært system med 1ere og 0er. Vi kaller dette for et binært tallsystem. Det er dette datamaskinene benytter når det skal gjøre utregninger. Vi kan si at dette systemet har 2 som grunntall siden det binære tallet 0101 er det samme som 0\*23 + 1\*22 + 0\*21 + 1\*20 = 5 på desimal form, altså den vi vanligvis bruker. Den desimale formen har 10 som grunntall. Likevel finnes det en tredje form, det heksadesimale tallsystemet. Denne har 16 som grunntall. Grunnen til at vi bruker denne er for å kunne skrive lange tall på kortere form. Om man for eksempel skal skrive et stort tall på binær form, kan man måtte bruke mange 0ere og 1ere. I tillegg til å kunne skrive tall på kortere form er det også enklere å regne om fra binær form til heksadesimal enn det er til desimal, siden grunntallet i det heksadesimale systemet er en potens av 2, som er grunntallet i det binære tallsystemet.

**Omregning mellom binær og desimal form**

Når man skal regne om fra binær form til desimal form kan man tenke seg at hver 0 eller 1 bestemmer antallet 1,2,4,8,16,32,64 eller 128ere. Man kan for eksempel skrive 18 på formen 10010, som betyr at vi har tallet 2 en gang, tallet 16 en gang. Om man skal regne det binære tallet 10010 tilbake til desimal form kan man gjøre følgende: 1\*24 + 0\*23 + 0\*22 + 1\*21 + 0\*20 = 16.

**Omregning mellom binær og heksadesimal**

Å regne om fra heksadesimal til binær og omvendt er mye enklere siden de begge har grunntall som er potenser av 2. Dersom man har et binært tall kan man konvertere det til heksadesimal ved å dele det opp i grupper på fire, skrive tallene på desimal form, og deretter slå dem sammen til det heksadesimale tallet. Merk at desimalene 10-15 skrives som A-F på heksadesimal form. For å gjøre om det binære tallet 11010 (26 på desimal form) kan man altså dele det inn i grupper på fire på følgende måte: 0001 1010. Merk at man deler det i grupper av 4 fra høyre og kan sette flere nuller foran det første sifferet dersom det ikke går opp. Disse to gruppene blir på desimal form 1 og 10. Siden tallet 10 på desimal form er A, ender vi opp med heksadesimalen 1A etter å ha slått samen gruppene igjen. Om man skal gjøre om fra heksadesimal til binær reverserer man denne prosessen.

**Omregning mellom heksadesimal og desimal form**

Ved omregning fra heksadesimal form til desimal form kan man gjøre på samme måte som fra binær til desimal form, bare at man bruker 16 som grunntall: 1A = 1\*161 + 10(A)\*160 = 16 + 10 = 26. Desimal til heksadesimal er enklest dersom man først regner om fra desimal til binær.

**Bit, byte, kilo og mega**

Standard er at det er 1000 byte i en kilobyte, men siden en datamaskin operer med 2 tallsystemet, vil operativsystemet si at det er 1024 byte i en kilobyte. Standardiseringsbyrået SI ønsker at man skal bruke standard kilo og mega definisjoner altså at det er 1000 byte i en kilobyte. Derfor har vi i dette faget valgt å skille mellom de to ved å bruke forskjellige betegnelser. Vi skriver liten k som betegnelse på 1000, og stor K som betegnelse på 1024. Altså er 4 kb 4000 bytes, mens 4 Kb er 4096 bytes.

## Virkemåte

**Grunnleggende komponenter i en datamaskin**

Som vi så lengre opp var von Neumann en av de første som beskrev den konstruksjonen av datamaskiner vi har i dag. Han beskrev en datamaskin med binær elektronikk og et minne der både data og instruksjoner lå.

Dette er de grunnleggende komponentene i en datamaskin:

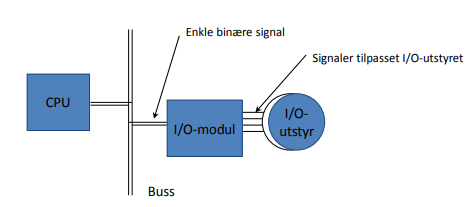
* CPU/prosessor – Dette er datamaskinens hjerne. Denne utfører meget grunnleggende og generelle operasjoner. Disse operasjonene er med andre ord ikke særlig vanskelige, men heller enkle. Eksempler på slike operasjoner er addisjon og sammenlikning av tall og å sjekke om et tall er positivt eller negativt. Disse operasjonen settes i gang ved at prosessoren mottar instruksjoner i form av bitmønstre fra minnet. Disse bitmønstrene sendes over bussen. Slike instruksjoner beskriver både hvilken operasjon som skal utføres og hvilken adresse dataene som skal behandles ligger i. Et dataprogram er en sekvens av slike instruksjoner og de ligger som regel i den rekkefølgen de skal utføre i, i minnet.
* Minnet – Minnet er en enhet som kan lagre et antall ord. Med et ord mener vi en binær kode med et visst antall bits. En bit er en 1er eller 0er. Forskjellig minne har forskjellig ordbredde, men det vanligste er 8, 16, 32, eller 64. Hver plass i minnet kalles en lokasjon og kan holde ett ord. Hver av disse lokasjonene har en adresse og det er sånn datamaskinen holder styr på hvor den skal lese og hvor den skal skrive. Data og instruksjoner befinner seg i samme fysiske minne. Lokasjonene i minnet adresseres uavhengig om det er data eller instruksjoner som befinner seg der. Det som ligger i hver lokasjon er et bitmønster. Det er opp til datamaskinen å tolke om dette er data eller en instruksjon.
* Bussen – En buss er et sett av parallelle ledere, hvert fall hvis det er en parallellbuss, noe minnebussen (bussen mellom minnet og prosessoren) er. Hver enkelt av disse lederne overfører verdien til en bit, ved å enten sende spenning, eller ikke (0 eller 1). En slik buss kan dermed overføre mange bits, en på hver av lederne sine.

**Prosessor, minne og buss samarbeider**

Når man starter et dataprogram på maskinen sin lastes programmets instruksjoner og data over til minnet. Instruksjonene i minnet utgjør hele utgangspunktet for hvordan en maskin fungerer. Disse instruksjonene overføres nemlig sekvensielt slik de ligger i minnet, over til prosessoren der de utføres. Unntaket er dersom det skjer i «hopp» i programmet. Eksempler på slike hopp er løkker der man for eksempel gjentar instruksjoner eller if/else setninger der man for eksempel hopper over noen instruksjoner.

**I/O-utstyr og I/O-moduler**

For å kommunisere med en datamaskin har vi mange slags komponenter som vi kan koble til datamaskinen. Eksempler på slike er tastaturer, mus, skjermer, og til og med harddisker. Siden disse er forskjellige typer I/O utstyr med forskjellige overføringshastigheter trenger vi forskjellige komponenter i maskinen vår som kan styre og kontrollere de forskjellige typene av I/O utstyr. Elektronikk som kan styre en bestemt type IO utstyr monteres på et lite kort som så installeres i datamaskinen. Dette kortet kobles så til bussen. Ofte kalles disse kortene kontrollere, adaptere eller bare kort. Vi installerer kort for akkurat den typen utstyr vi har tenkt til å bruke. Kortene fungerer ved at IO enheten sender eller mottar bitmønstre på den ene siden, mens kortet sender eller mottar bitmønstre fra/til bussen på den andre siden. Vi kan si at kortet kommuniserer med IO enheten på den ene siden og CPU på den andre siden (via bussen).



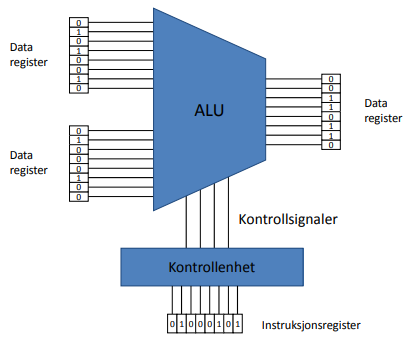
Det finnes tre typer bitmønstre som går mellom CPU og en IO kontrolleren:

* Kommandoer – Dette er bitmønstre som prosessoren sender til kontrolleren. I kontrolleren finnes det elektronikk som gjør at kontrolleren kan utføre kommandoene.
* Statusrapport – Dette er bitmønstre som CPU leser fra kontrolleren. Disse forteller om feilsituasjoner eller annen statusinformasjon.
* Data – For ut-utstyr vil dette være informasjon som prosessoren skriver til kontrolleren, for eksempel skjermer. For inn-utstyr vil dette være informasjons som prosessoren leser fra kontrolleren, for eksempel tastaturer.

CPU leser og skriver altså til disse modulene på akkurat samme måte som den gjør til minnet. Siden en datamaskin inneholder flere slike moduler må CPU kunne skille mellom dem. Hver kontroller får derfor sine egne adresser på samme måte som lokasjonene i minnet får sine egne adresser.

**CPUens innhold og virkemåte**

Inne i prosessoren er det binære signaler som bestemmer hva som skjer. I CPUen er det en kontrollenhet som tolker instruksjonene og setter opp de riktige kontrollsignalene. ALUen utfører de riktige operasjonene på de riktige dataene. Disse dataene ligger i et lite minne inne i CPU som kalles et register. Hvert register inneholder ett ord, altså et bitmønster med et fast antall bits. De fleste moderne prosessortyper har ordbredde på 32 eller 64 bits. De fleste prosessorer har noen titalls slike registre. Alle registrene kalles sammen registerblokken. De viktigste delene av CPUen er dermed ALUen, kontrollenheten og registerblokken.



Her ser vi en skisse av hvordan CPU fungerer. Det er ALU som utfører alle de aritmetiske og logiske operasjonene. Det er kontrollenheten som tolker instrukser fra instruksregisteret og sender kontrollsignaler til ALU om hvilke operasjoner den skal utføre. ALUen har to innganger og en utgang. Den tar da inn to bitmønstre og utfører operasjonen den har blitt bedt om å gjøre, før det resulterende bitmønsteret presenteres på utgangen. Både bitmønstrene som går inn og ut av ALU ligger i registre.

Vi deler gjerne inn instruksjonene i tre forskjellige typer instruksjoner:

* Instruksjoner for dataprosessering – Instruksjoner som gjør at CPU utfører en aritmetisk eller logisk operasjon
* Instruksjoner for dataoverføring – Dette er instruksjoner for overføring av data mellom et CPU register og en minnelokasjon. Dette gjelder også overføring av data mellom CPU og en IO modul. Slike overføringer kan gå begge veier.
* Instruksjoner for programkontroll – Instruksjoner som endrer instruksjonsrekkefølgen bort fra den normale sekvensielle rekkefølgen. Dette vil si hopp i programmet.

**Viktige registre i CPU**

De fleste registrene i CPU har et navn, og sin helte egne funksjon. Her er de viktigste:

* Program Counter, PC – Dette registeret inneholder adressen til neste instruksjon. Kan også kalles IP, instruksjonspeker.
* Instruction register, IR – Dette registeret inneholder instruksen som skal utføres
* Accumulator, AC – Dette registeret inneholder data. Her korttidslagres data.

**Instruksjonssyklus**

Vi deler gjerne instruksjonssyklusen i to deler, en hente-del og en utførelses-del. I hente-delen, henter CPU en instruks fra den adressen som står i PC registeret. Når den er hentet inkrementeres PC, altså at den øker med en. Dette gjøres ikke dersom instruksen ber om noe annet. CPU er nå klar til å utføre instruksen. I utførelses-delen utføres instruksjonen i IR registeret. Deretter kan CPU hente neste instruksjon fra den adressen som ligger i PC.

En instruksjon består av en op-kode og en adresse. Op-koden bestemmer hvilken operasjon som skal utføres, og adressen sier hvor dataene som skal brukes skal hentes fra. Om man for eksempel skal addere to tall trenger man tre instrukser. En som leser inn en verdi fra minnet og legger den i et dataregister i minnet. En som adderer tallet i dataregisteret med et tall fra minnet, og deretter legger det nye tallet i dataregisteret. Den siste instruksjonen skriver innholdet av dataregisteret til en adresse i minnet.

## Busser

Denne leksjonen tar opp følgende

* Parallelle busser
* Serielle busser
* Svitsjede busser

Det finnes forskjellige fordeler og ulemper med disse og de har alle forskjellige bruksområder i en datamaskin. Det som avgjør hvilken buss man trenger i en datamaskin, er om man trenger parallell eller seriell overføring, hvor mange enheter som skal kommunisere over bussen, og hva slags informasjon som skal overføres.

**Parallell og seriell overføring**

Når man overfører instrukser og data, er dette i form av bitmønstre. Man trenger aldri å bare overføre en bit, men gjerne flere. Spørsmålet er da om man skal overføre en og en etter hverandre, eller om man skal overføre flere samtidig. Dersom man overfører flere bits samtidig over hver sin egne ledning, sier vi at vi bruker parallell overføring. Om man skal overføre 32 bits, kan man for eksempel overføre disse over en buss med 32 parallelle ledninger. Et annet alternativ er å overføre 32 bits over samme ledning, men etter hverandre. Vi kaller det da seriell overføring.

**Port kontra buss**

Med busser ønsker man gjerne å koble sammen flere enheter. Likevel hender det at man bruker busser som kun går mellom to enheter. I slike tilfeller er det helt klart hvem som skal sende og hvem som skal motta informasjon. Dersom man har en buss mellom kun to enheter, kaller man det en port. Når man skal koble sammen flere enheter må man kunne vite hvilke enheter som skal kommunisere med hverandre. I tillegg til informasjonen som skal overføres må vi også fortelle hvem som skal ha informasjonen. Vi trenger dermed adressering for å skille mellom enhetene.

**Båndbredde**

Med båndbredde mener vi hvor mange bits en buss kan overføre i sekundet. Det er da snakk om hvor mye informasjon som overføres per sekund. Likevel har vi sett at bussen også brukes til å overføre kontrollsignaler og adresser for å finne ut hvilke lokasjoner som skal brukes. I tillegg til å overføre informasjon, brukes altså bussen også til mer administrative formål. Vi ønsker å bruke bussen minst mulig til slike administrative overføringer, for å kunne overføre mest mulig informasjon, på minst mulig tid. Når man oppgir båndbredde, oppgir man derfor den maksimale båndbredden, som vil si at man antar at bussen kun overfører informasjon. I praksis er dette nesten umulig, og man vil ikke få like høy båndbredde som oppgitt.

**Arbitrering/distribuert buss**

Siden bussen er en delt ressurs, dvs. at alle enheter som er koblet til den, overfører signaler over de samme ledningene, kan kun en enhet sende signaler om gangen, ellers blir det kaos. Det at de deler bussen som en ressurs betyr også at alle enheter kan «se» de andre enhetenes signal når de sendes. Vi kan se på det som at «en skriver, flere lytter». Dette betyr at enhet til enhver tid kan «lese» fra busse, altså motta informasjon, mens kun en kan «skrive» til bussen om gangen. Vi trenger derfor en enhet som passer på at kun en skriver til bussen om gangen. Vi skiller mellom to hovedmetoder for å passe på dette. Vi har sentraliserte busser, og distribuerte busser. Forskjellen på disse er teknikken de bruker for å bestemme hvem som skal få skrive på bussen. Dette kalles «arbitrering» og er altså gjort på forskjellige måter i de to bussene.

På en *sentralisert buss* finnes det en elektronisk krets som kalles for en *busskontroller* som gir enhetene tilgang til bussen ved behov.

På en *distribuert buss* kan alle enhetene reservere bussen dersom den er ledig. Dette krever ytterligere elektronikk i hver enhet for tilgangskontroll.

**Vi skiller mellom forskjellige overføringsretninger**

Dersom en enhet bare kan sende, og en annen bare kan motta, kaller vi overføringen mellom disse Simpleks.

Dersom begge enhetene kan både sende, skrive og motta, lese, men ikke samtidig, kaller vi overføringen mellom enhetene for halv dupleks.

Dersom begge enheter kan lese og skrive, og samtidig, kaller vi overføringen mellom disse full dupleks.

**Busshierarki**

Siden forskjellige enheter har forskjellige båndbredder er det ikke gunstig å la alle lese og skrive til samme buss. Vi ønsker derfor å ha flere busser med forskjellige båndbredder. På denne måten blir den totale båndbredden høyere. Det blir derfor et busshierarki der bussene med høyest båndbredde er på toppen. Nedover vil vi finne busser med lavere og lavere båndbredder. Fordelene med dette er som sagt at den totale båndbredden øker, men også at vi kan tilpasse hver enkelt buss til spesielle behov. I tillegg kan vi også ha forskjellige båndbredder på forskjellige busser.

Vi skiller gjerne mellom tre hovedkategorier av busser. Den første og høyest rangerte er minnebussen mellom minnet og CPU. Den må ha ekstrem høy båndbredde siden CPU og minnet er noen av de raskeste komponentene i en datamaskin. Videre har vi bussen som knytter alle IO-moduler sammen. Dette kan være moduler for skjermkort, disker, nettverkskort og flere. Til slutt har vi bussene som går mellom hver enkelt IO-modul, og selve IO enheten. Denne kalles IO buss, og et vanlig eksempel på disse er USB.

### Parallelle busser

En parallell buss består av parallelle ledninger som hver kan overføre en bit om gangen. En typisk buss består av mellom 50 og 100 slike ledninger. Vi deler gjerne bussen inn i tre mindre delbusser med forskjellige overføringer:

* Databuss. Denne overfører data
* Adressebuss. Denne overfører binære adresser
* Kontrollbuss. Denne overfører alle de administrative signalene som styrer og synkroniserer overføringene.

Når en CPU skal lese fra en minnelokasjon må først CPU reservere bussen slik at ingen andre skriver samtidig. Deretter legger CPU adressen på adressebussen og et kontrollsignal på kontrollbussen som betyr les minne. Siden alle enhetene som er koblet til denne bussen kan lese fra den, vil minnet se kontrollsignalet og skjønne at dette er en henvendelse til seg. Elektronikken i minnet leter så opp adressen som var på adressebussen og legger innholdet av adressen ut på databussen slik at CPU kan lese det.

**Databussen**

Dette er den bussen som overfører data og består ofte av 16, 32 eller 64 parallelle ledninger. Dette antallet ledninger kalles databussens bredde, og er ofte en begrensende faktor i en PC. Dette betyr at databussen er en nøkkelfaktor for datamaskinens totale hastighet.

**Adressebussen**

Denne bussen overfører adressen til dataene som skal overføres. Det betyr at CPU legger adressen til den lokasjonen den ønsker å lese fra på denne bussen. Denne bussens bredde avhenger av størrelsen på minnet i maskinen. Den må være så bred at alle adresser kan legges på bussen. Denne bussen kan også brukes til å adressere IO-moduler.

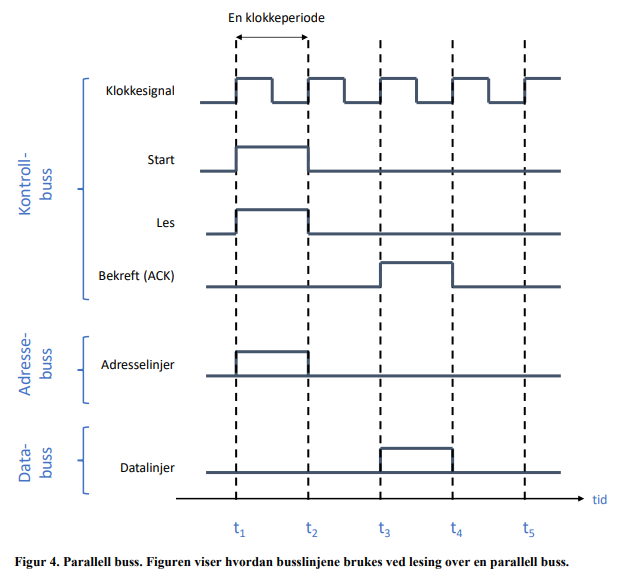
**Kontrollbussen**

Denne bussen brukes for å kontrollere og synkronisere overføring av informasjon. Denne bussen har litt forskjellige standarder for forskjellige busser.

**Datakommunikasjon på en parallell buss**

De fleste parallelle busser er synkrone, dette betyr at de bruker en klokkepuls for å styre aktivitetene på bussen. Poenget med en slik klokkepuls, er at en ny aktivitet kun kan begynne på en ny slik puls. Ofte kan denne pulsen slå flere hundre millioner ganger i sekundet på moderne busser. Vi oppgir antall pulser per sekund som klokkefrekvens (Hz). Dette klokkesignalet overføres på en av ledningene til kontrollbussen. På denne måte har alle enheter tilgang til den.

Vi kan bruke *timing diagram* for å vise hvordan delbussene brukes under en overføring. Den viser spenningen til hver linje i en buss som en funksjon av tid. Figuren på neste side viser et eksempel på et slikt diagram. Det viser linjene til de forskjellige delbussene; kontrollbussen, adressebussen og databussen.



Her ser vi hvordan lesing fra en adresse fungerer i praksis. Det hele starter med at kontrollbussen «starter» en ny prosess som den sier er å «lese» fra minnet. Samtidig legger CPU en adresse i adressebussen. Alt dette skjer på den første klokkepulsen. Siden minnet bruker litt tid på å finne frem til en adresse skjer det ingenting på den andre klokkepulsen. Når minnet finner adressen, legger den innholdet av lokasjonen ut på databussen og setter kontrollsignalet «bekreft» for å vise at CPU nå kan begynne å lese. Dette ser vi skjer på den tredje klokkepulsen. CPU får en klokkepuls på seg til å lese før alle busslinjene resettes og kan begynne på neste operasjon.

**Båndbredde på en parallell buss**

Siden en parallell buss kan overføre på hver klokkepuls kan vi regne ut båndbredde ved å multiplisere klokkefrekvensen og bredden på databussen. Dersom man for eksempel har databuss med 32 i bredde, og 1GHz klokkefrekvens, betyr det at den kan overføre 32 bits hver gang klokkepulsen «slår». Det vil si at den kan overføre 32 bits en milliard ganger i sekundet, altså 32 milliarder bits i sekundet. Dette tilsvarer 32 Gb/s (gigabits per sekund).

Som vi har sett vil vi ikke få så høy båndbredde i realiteten da flere klokkepulser går bort til minnets aksesstid og kontrollsignaler.

**Hva begrenser båndbredden til parallelle busser**

Som vi har sett vil båndbredden til en parallell buss være frekvensen ganget med antall linjer på databussen. Siden vi får raskere og raskere prosessorer, ønsker vi også busser med meget høy båndbredde. Man skulle da kanskje tro at man bare kan øke bussbredden elle øke klokkefrekvensen for å få høyere båndbredde. Likevel følger det visse komplikasjoner med å gjøre dette. Første og fremst er det ikke gunstig å øke bussbredden ytterligere da en buss med for eksempel 128 i bredde, vil ta veldig stor fysisk plass og gjøre at tilkoblingen dens også vil det. For det andre vil det å øke frekvensen mye også følge med en del komplikasjoner. Det er nemlig sånn at når man øker frekvensen så vil signalene i ledningene ikke bruke nøyaktig like lang tid gjennom ledningene. Dette gjelder alltid, men blir mye tydeligere, og viktigere, med høy frekvens. Jo lengre ledningene til bussen er, jo lengre vil ulikhetene bli i hvor lang tid et signal bruker. Jo bredere bussen er, jo større individuelle forskjeller blir det. Problemet med høy frekvens er at bitene kan komme så tett tiden mellom dem kan nærme seg forskjellen i gangtid. Dette viser også at korte busser kan ha mye høyere båndbredde enn lange busser.

**Hvorfor bruker vi parallelle busser**

Grunnen til at vi bruker parallell buss i dag, er at den er ekstremt rask. Den brukes derfor som minnebuss, bussen mellom CPU og minnet. Dersom man ser inne i en PC, ser man at minnet ikke er plassert langt unna prosessoren. Dette er for å gjøre bussen kortest mulig, og dermed kunne ha høy frekvens. Ellers i PCer serielle busser mye brukt. En moderne minnebuss er vanligvis en 64 bits parallellbuss med frekvens i GHz-området.

### Serielle busser

Serielle busser overførere som nevnt tidligere bits over samme ledning, en etter en. Man skulle da tro at en seriell buss vil være mye tregere enn en parallell buss som kan overføre mange bits samtidig. På den annen side tar serielle busser opp mye mindre fysisk plass og kan ha mindre tilkoblinger.

**Båndbredde på serielle busser**

Selv om en parallell buss i teorien kan være 16 ganger raskere, for en 16 bits parallellbuss, vil dette ikke helt stemme. Grunnen til dette er, som vi har sett, at parallelle busser ofte må vente litt for å unngå at signaler kommer frem på forskjellige tidspunkter osv. Vi kan si at parallellbusser kommer med begrensninger når det gjelder frekvens. Dette gjør ikke serielle busser da de overfører over en ledning. Disse bussene kan ha høy frekvens og lange ledninger uten noen problemer. Grunnet god teknologi har disse bussene nå også relativt høy båndbredde og har erstattet flere parallellbusser de siste årene.

**Hvor bruker vi serielle busser**

Vi bruker stort sett serielle busser til IO utstyr. Dette kommer av at serielle busser har blitt såpass raske, og at tilkoblingsmatriellet deres er såpass smått, noe som kommer til nytte når man skal ha mange IO enheter. Vi snakker gjerne om tre ting som er annerledes med de moderne IO bussene enn de gamle:

* Serielle busser i stedet for parallelle. Serielle busser tar mindre plass, og kan være lengre noe som er en stor fordel med det meste av IO utstyr.
* Hot swap. Nye busser tillater å koble til og fra PC-en mens den er på. Dette er ikke spesielt for serielle busser, men moderne busser generelt. Dette er en stor fordel da man trenger relativt mye IO utstyr for diverse oppgaver, tastatur, mus, hodetelefoner, tegnebrett og mer.
* Tettere samarbeid mellom bussen og operativsystemet. Dette kommer litt som en konsekvens av hot swap. Det er greit å få beskjed av operativsystemet når en ny enhet er koblet til, og når driveren er ferdig nedlastet.

**USB**

Står for «Universal Serial Bus». På USB kan man koble til inntil 127 enheter, og den moderne USB typen kommer med båndbredde på inntil 5Gbits/s.

### Hovedbussen

Dette er bussen som kobler alle IO modulene sammen. Tidligere har dette vært en 32 bits parallellbuss kalt PCI, men ettersom man har trengt å koble sammen flere og flere IO enheter, og båndbredde kravene har økt, har det vært behov for å gå vekk fra en delt buss.

### Svitsjede busser

Nye datamaskiner er utstyrt med standarden PCI-Express, som baserer seg på svitsje-teknologi.

**Svitsjet kommunikasjon**

Å ha delte busser hadde, som vi har sett, den ulempen at kun en enhet kunne skrive til bussen om gangen. Dette betyr at to IO moduler ikke kunne kommunisere med hverandre dersom en helt annen IO modul ønsket å kommunisere med for eksempel CPU. Løsningen på dette er svitsjet kommunikasjon. Den fungerer litt som en gammeldags telefonsentral der to og to kan snakke sammen over en lukket forbindelse. Det vil si at så mange moduler som mulig kan snakke med hverandre, gitt at ingen er med i to «samtaler» samtidig. Likevel er det flere spørsmål man må stille seg når man bruker svitsjet buss. Skal tilkoblingene være serielle eller parallelle, og hvordan skal svitsjen prioritere forskjellige enheter dersom det er flere moduler som vil snakke med en enhet. Vi trenger altså en standard.

På PCI-Express som brukes i alle moderne PC-er er koblingen mellom svitsjen og komponentene seriell. Likevel er det to koblinger mellom alle komponentene og svitsjen, dette gjort for at forbindelsene skal være full duplekse. Det som er smart med PCI-Express er at man kan gi to komponenter flere tilkoblinger dersom de trenger høyere båndbredde. I PCI-Express har hver seriell kobling en frekvens på 2.5 GHz. En slik kobling kalles en *lane*. Grafikkortet kan for eksempel bruke 8 lanes. Merk at det er PCI-Express 1 som har 2.5 GHz i båndbredde per tilkobling. Nyere versjoner har 5 eller 8.

## Primærminne

Sammen med cache og registrene, er primærminnet en av de lagertypene som CPU kan aksessere direkte. Ytelsen til minne påvirker ytelsen til hele datamaskinen sterkt. Dette kommer av at et programs instrukser må hentes fra minnet, og kjøringen av programmet dermed avhenger av hvor raskt minnet klarer å hente ut instruksene.

Vi skiller mellom to typer minne, ROM og RAM. ROM er en forkortelse for «Read Only Memory», mens RAM står for Random Access Memory. En datamaskin inneholder begge deler, men RAM er det man vanligvis tenker på når man sier minne. ROM brukes blant annet for å lagre operativsystemet på. Dersom man slår av datamaskinen, forsvinner ikke innholdet på ROM.

**RAM**

RAM er avhengig av kontinuerlig drivspenning og mister derfor sitt lagrede innhold om man slår av datamaskinen. Grunnen til at det heter «random» er at man kan lese eller skrive til en vilkårlig plass i minnet, i stedet for å måtte starte for eksempel på starten hver gang.

**DRAM og SRAM**

DRAM står for dynamisk RAM og her fungerer hver celle ved at en kondensator «husker» hva som er lagret i cellen. Den husker dette så lenge den har spenning. Det betyr at man trenger elektronikk som hele tiden gjenoppfrisker hukommelsen dens, ved å tilføre ny spenning. SRAM står for statisk RAM og her er hver enkelt celle en stabil transistorkrets. Her huskes informasjon så lenge ikke drivspenningen ikke slås av. Man skulle da tro at SRAM er mye billigere og bedre, siden det ikke krever konstant «gjenoppfriskning». Likevel er ikke dette tilfellet da DRAM sine celler er langt mindre enn SRAM sine, og man får dermed DRAM med langt større lagringskapasitet. Samtidig er DRAM langt billigere. Ulempen med DRAM er at den er mye tregere enn SRAM. Dette er grunnen til at vi bruker DRAM i primærminnet, mens vi bruker SRAM i cache.

### Minnebrikkene

Vi ser vanligvis på minnet som en enhet. Denne enheten består av mange adresser og hver av disse kan lagre et ord, men forskjellig ordbredde fra minne til minne. Vi oppgir lagringskapasiteten til minnet i bytes, selv om ordbredden gjerne er større enn 8 bits. Nå skal vi «åpne» minnet og se på komponentene inne i minnet, som vi kaller for minnebrikker

**Minneceller**

Minnebrikkene er integrerte kretser med mange celler. Hver av disse cellen kan lagre en bit, og har hver tre tilkoblingspunkter:

* Valg. Dette signalet settes når denne cellen skal leses eller skrives fra
* Kontroll. Dette signalet avgjør om det skal leses eller skrives til cellen
* Data. Dette signalet er det som faktisk leser en bit fra cellen eller skriver en bit til cellen

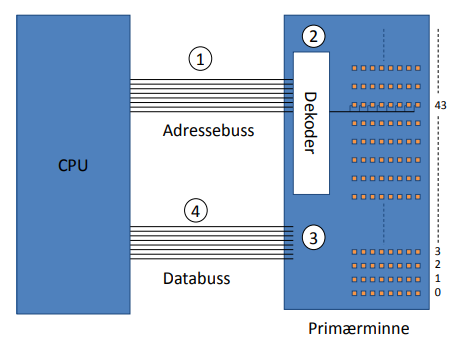
Fra utsiden er minnet bygd opp av en lang rekke lokasjoner. Hver lokasjon holder et ord med en bestemt ordbredde, altså et visst antall bits. Dette betyr at det finnes like mange celler i hver lokasjon som det er bits i et ord. Når vi da skal lese eller skrive til en enkelt minnelokasjon settes velg-signalet til cellene i denne lokasjonen. Kontrollsignalet settes også til cellene og forteller om det skal leses eller skrives. Datasignalet settes fra enten den som skal skrive til cellene, eller fra cellene om det skal leses fra dem.

**Dekoder**

Når for eksempel CPU legger en adresse på adressebussen, må minnet vite hvilke celler i minnet den skal sette velg-signalet for. Dette er det en dekoder som gjør. En dekoder fungerer ved å ha et inngangssignal, en adresse, og like mange utgangssignal som det finnes antall adresser i minnet. Det er riktignok bare en av disse utgangssignalene som kan være høye, altså ha spenning. Dersom for eksempel CPU etterspør noe i adresse 0101, som på desimal form er adresse 5, vil utgangssignal nr. 5 være høyt, mens de andre 15 signalene vil være lave.

**Ordbreddeorganisering**

Om man lagrer alle cellene på en brikke i like stor bredde som ordbredden i hver lokasjon, vil man få plass til alle bitene i et ord, på en brikke. Ulempen er at man trenger en meget avansert dekoder dersom man skal ha lagre en del informasjon.



Her ser vi et eksempel der ordbredden i minnet er 8 bits. Dersom minnet totalt skulle hatt 1 kilobyte lagring, ville man trengt en 10 bits dekoder for å kunne dekode alle adressene. Dette kommer av at 1 kilobyte ville betydd 1024 adresser dekoderen hadde måttet kunne aksessere. Man trenger med andre ord meget avanserte dekodere for å sette opp cellene på denne måten.

**Kvadratisk organisering**

Det å lage den integrerte kretsen med alle cellene i minnet er meget enkelt rent produksjonsteknikk. Som vi har sett går det an å få alle cellene på en brikke, men dette krever avanserte dekodere. Disse derimot er meget avanserte å dermed vanskeligere å lage. Derfor sprer vi gjerne minnecellene utover flere brikker.

Ytterpunktet her blir å ha en brikke for hver bit i et ord. Det vil si at en brikke inneholder bit 0 for alle lokasjoner, en brikke inneholder bit 1 for alle lokasjoner, osv. Selv om vi da trenger like mange brikker som vi har bits i hvert ord, kan dekoderne nå være mye enklere. Vi plasserer cellene i kvadratisk form i rekker og søyler, og deler adressen i to. Den ene halvdelen velger ut rekke, og den andre søyle. Dette er grunnen til at dekoderne nå blir mye enklere, fordi vi nå trenger dekodere som kan dekode færre bits. Det å ha like mange brikker som man har bits i ordene, er en meget vanlig oppbygning vi kan da oppgi lagringskapasiteten til en brikke i antall bits som lagres på den. Vanlige størrelser er 16, 64, 265 megabits osv. Ettersom vi klarer å lage mer avanserte dekodere, kan vi også utvide antall rekker og søyler, og dermed øke lagringskapasiteten til minnet. Ved å for eksempel kunne dekode en ekstra bit, vil man kunne legge til dobbelt så mange rader og dobbelt så mange søyler, som resulterer i 4 ganger så mange celler. Dette skjer ca. hvert tredje år.

**Andre organiseringer av minnebrikkene**

Det går an å organisere annerledes, men noen standard for dette finnes ikke.

**SIMM og DIMM**

SIMM brikker var ferdige lagde minnebrikker med 32 bits minne. På denne måten trengte man ikke tenke på organiseringen av cellene. DIMM kom etter at et lyst hode innså at man kunne ha celler på begge sider av brikken. Man ser dermed forskjell på de ved å se om det er lagerbrikker på begge sider eller ikke. DIMM er også dermed dobbelt så stor lagring som SIMM.

## IO

**IO utstyr**

Det finnes mange forskjellige IO enheter. Alt fra tastaturer til skjermer. Disse har forskjellige bruksområder, men også helt forskjellig kompleksitet. Skjermer er relativt kompliserte, og helt vitale, i forhold til hodetelefoner.

**Virkemåte**

IO utstyr kommuniserer med CPU gjennom IO moduler som er koblet til en egen buss, hovedbussen. Her kommuniserer CPU med modulene på ganske lik måte som med minnet. Hver IO modul har to porter (lokasjoner, men kalles porter eller IO-registre når det er snakk om IO), en dataport og kontrollport. CPU forteller IO utstyret hva den skal gjøre ved å skrive bitmønster til kontrollporten, men kan også motta statusinformasjon fra denne porten. Dataporten brukes når CPU skal lese data fra IO enheten eller når den skal sende data til IO modulen. Disse portene adresseres på samme måte som lokasjoner i minnet for at CPU skal kunne skille mellom dem, men det kan skje at flere IO moduler får samme adresser. Da finnes det to muligheter:

* Man kan ha isolert IO, som vil si at man har en egen linje på bussen som settes dersom adressen på adressebussen er en IO port. Her må altså bussen ta hensyn til om det er en adresse eller en port.
* Man kan sette av egne adresser i minnet som skal adressere IO porter. Dette tar opp plass i minnet, men bussen slipper å ta hensyn til om det er en adresse eller en port som adresseres.

Problemet med å lese/skrive fra IO utstyr er at det ofte er veldig tregt sammenliknet med prosessoren. Dette betyr at prosessoren må bruke mye tid på å vente til IO utstyret er ferdig med sine oppgaver før den kan begynne på noe nytt. Det er derfor vi trenger avbrudd og programmert IO (polling) som sikrer synkronisering mellom programutførelse og IO-operasjoner.

### Avbruddsystemet

Som vi har sett bruker IO utstyr mye lengre tid på forskjellige oppgaver enn det prosessoren gjør på å utføre instrukser. Dersom man for eksempel skal skrive ut noe på en printer kan man tenke seg at CPU og skriver kommuniserer på følgende måte: CPU sender et tegn til skriveren og venter på at den skal bli ferdig med å skrive tegnet, før den sender ett nytt. Måten skriveren kan si fra om den er ferdig eller ikke, er ved å ha en egen linje på IO modulen som settes dersom den er ferdig. Dermed kan CPU bare sjekk om linjen er satt eller ikke for å vite om den skal sende neste tegn. Likevel er dette en løsning som funger svært dårlig i realiteten. Dette kommer av at CPU da hele tiden må sjekke om denne linjen er satt eller ikke. Den tiden CPU bruker på å sjekke om den kan fortsette, burde ha blitt brukt på andre ting. Det er her avbruddsystemet kommer inn. Vi ønsker nemlig at skriveren selv kan si fra når den er klar til å fortsette. På denne måten kan CPU gjøre andre ting inntil den får klartegn fra skriveren.

**Polling**

Eksempelet over der CPU stadig måtte spørre skriveren om den var klar, viser at CPU hele tiden måtte «polle» skriveren. Dette kalles polling, eller stadig spørring. Denne metoden fungerer godt dersom CPU har lite annet å holde på med, men meget dårlig i moderne maskiner der mange prosesser kjører samtidig.

**Avbruddsdrevet IO**

I stedet for at CPU selv må sjekke når skriveren er klar, burde det være sånn at skriveren selv kan si fra. På denne måten kan CPU i mellomtiden gjøre andre nyttige oppgaver. Når skriveren endelig er klar, kan IO modulen varsle CPU ved å sende et *avbruddssignal*. Dette betyr at CPU ikke gjør akkurat som den vil, som man kanskje kunne ha fått inntrykk av før. Den må nemlig kunne ta hensyn til at noen etterspør den gjennom avbruddssignaler. Avbruddssignalet får nemlig prioritering. CPU legger til side det den holder på med, og fullfører det som kreves av den. Men hvordan vet den hva den skal gjøre når den får et avbruddssignal? Jo, det finnes noe som heter avbruddsrutiner (ISR – Interrupt Service Routine). Dette er instrukser CPU skal utføre når det kommer avbrudd. CPU vil altså legge til side det den holder på med og starte på avbruddsrutinen, dersom det kommer et avbrudd. Disse avbruddsrutinene er relativt korte program fordi vi ikke ønsker at CPU skal bruke mye tid på hvert avbrudd. Selve avbruddssignalet sendes over en egen linje på kontrollbussen. Når denne linjen aktiveres, vil CPU fullføre den instruksjonen den holder på med, for så å begynne på avbruddsrutinen.

Tidligere så vi på at prosessorens instruksjonssyklus der CPU henter en instruksjon og deretter utfører den. Nå legger vi til et trinn i denne syklusen. Etter at den har utført en instruksjon, skal nemlig CPU sjekke om den har fått noen avbrudd. Siden avbruddsrutiner også er små program som ligger i minnet, vil alt som skjer ved avbrudd være at adressen i PC endrer seg til startadressen for avbruddsprogrammet. Likevel må vi huske at CPU skal gå tilbake til å utføre instrukser fra det programmet den holdt på med før avbruddet. Det betyr altså at CPU lagrer innholdet av PC registeret ved et avbrudd. Deretter settes PC lik startadressen til avbruddsprogrammet. Når dette er ferdig laster CPU tilbake den adressen som lå i PC før avbruddet, og CPU kan fortsette med dette.

**Forskjellige typer avbrudd**

Vi har nå sett at avbrudd brukes for IO utstyr, men avbrudd brukes i mange flere sammenhenger. Nå skal vi se på noen av disse.

Klokkeavbrudd – En klokkekrets gir avbrudd, med meget høy prioritet og med fast frekvens. Operativsystemet benytter disse avbruddene til å utføre administrasjon.

Utstyrsfeil – Både CPU og IO utstyr har elektronikk som sier fra dersom noe er feil. Disse feilene formidles gjennom avbrudd. Eksempler kan være bortfall av spenning eller paritetsfeil i minnet.

Programfeil – Mange feil i et program kan gi avbrudd. Eksempler er divisjon med 0, forsøk på å utføre ulovlige instruksjoner, ulovlig operand eller forsøk på å lese eller skrive til/fra beskyttet del av minnet.

**Programmert avbrudd – kall til operativsystemet**

Alle programmer trenger noen ganger at operativsystemet gjør dem en tjeneste. Dette er da gjerne tjenester som programmet selv ikke har tillatelse, **privilegier**, til å gjøre. Eksempler på dette er innlesing fra tastatur, skriving på skjermen eller skriver, eller å skrive og lese på disk. I stedet ber programmet operativsystemet om å utføre tjenesten. Vi sier at programmet starter en systemrutine. Måten et program starter en systemrutine er ved å foreta et avbrudd. Dette betyr at programmet utførere en instruksjon som får CPU til å oppføre seg som om det hadde skjedd et avbrudd. Dermed setter den i gang med en avbruddsrutine som i vårt tilfelle er systemrutinen vi ønsker å kjøre. Et slikt imitert avbrudd kaller vi for et programmert avbrudd. Grunnen til at dette lar programmet utføre de tjenestene det ønsker, er at CPU settes i privilegert modus ved avbrudd. Dette betyr ikke at et program kan gjøre hva som helst ved å bruke programmerte avbrudd, det er nemlig nødvendig med en liten sjekk før tjenesten utføres. Dette kan for eksempel være at operativsystem sjekker om bruker har lov til å skrive filen.

### Flere typer IO

Tidligere har vi sett på IO fra maskinvaren sitt ståsted. Nå skal vi se på IO fra programvaren sitt ståsted. Det finnes ulike måter å gjennomføre IO operasjoner. De tre metodene belaster CPU i ulik grad:

* Programmert IO. Dette kalles også polling.
* Avbruddsdrever IO
* Direkte minneaksess.

Programmert IO belaster CPU i stor grad da den stadig må sjekke statusen på IO utstyret. Denne metoden egner seg godt for elektronikk som ikke har så mange arbeidsoppgaver. Eksempler på dette kan være fjernkontroller eller overvåkningssystemer.

Avbruddsdrevet IO avlaster CPU betydelig fordi IO modulene klarer å jobbe mer selvstendig. Denne metoden kan være mer kostbar, men CPU kan i mellomtiden benytte tiden til mer nyttige oppgaver.

Disse to metodene har begge noen ulemper: For det første medfører begge at prosessoren må utføre mange instruksjoner. I tillegg går all data via prosessoren. Selv om avbruddsdrevet IO er bedre fordi CPU får gjort andre oppgaver mellom avbruddene, går også mye av CPUens tid bort til administrasjon. Spesielt dersom avbruddene kommer tett på hverandre. Det man trenger, er en metode som lar IO overføre data til og fra minnet uten å gå via CPU. På denne måten sparer CPU mye tid.

### DMA – Direkte minneaksess

Denne metoden brukes på IO moduler som overfører store datamengder. Eksempler kan være diskkontrollere, høyhastighets nettverk, grafiske skjermer med høy ytelse osv. Slike IO moduler kan vi gi ansvaret for å overføre hele blokker med data direkte mellom minnet og modulen. Det eneste CPU må gjøre er å fortelle modulen hva som skal gjøres, hvor det skal gjøres, og hvor mye som skal gjøres. Dette kan være at modulen skal lese et bestem antall ord fra et bestemt spornummer på en disk. Deretter setter modulen i gang med hele overføringen, og sender ikke avbruddsignal til CPU før den er helt ferdig. En slik modul vil være mer avanser og dermed dyrere, men til gjengjeld avlaste CPU. DMA er en enhet som bygges inn i IO modulen.

## Sekundærlager

Når vi snakker om sekundærlager, snakker vi om det folk flest kanskje tenker på når de hører lagring. Det vil for eksempel si harddisken. Det er denne CPU leser fra når den skal laste et program inn i minnet.

### Magnetiske media

Magnetiske sekundærlagre er basert på at informasjon lagres i form av ulik magnetisk polarisasjon. Slike lagertyper er billige per GB og kan ha høy lagerkapasitet.

Det viktigste magnetiske lagringsmediet på en moderne datamaskin er harddisken. Deres ytelse er i stor grad med på å påvirke en maskins totale ytelse. De har nemlig en svært viktig funksjon:

* Harddisken brukes som arkiv for både data og programvare. Dette vil si at alle programmet leses herfra og legges i minnet når de skal kjøres. Vi er derfor avhengig av at harddisken ikke har så særlig lang aksesstid, og at den kan lese relativt hurtig.
* Siden moderne operativsystemer bruker virtuelt minne, legger den de delene av primærminnet som brukes minst på harddisken, dersom primærminnet begynner å bli fullt. Dersom maskinen plutselig trenger disse dataene igjen, må de leses inn på nytt herfra. Dette kalles swapping.

En disk er en rund plate med magnetisk belegg. Disse diskene er laget av aluminium, og kan leses fra eller skrives til ved at et diskhode beveges over disken og leser eller skriver. Dette diskhodet beveges radielt over disken, og er egentlig en spole. En harddisk består som oftest av mange disker som er koblet på samme akse. Hver disk har to diskhoder som kan lese eller skrive, en på undersiden, og en på oversiden. Alle disse diskhodene sitter på samme arm og beveger seg dermed sammen. Skriving til disken skjer ved at det sendes en elektrisk puls til spolen i diskhodet. Det dannes da et elektrisk felt rundt spolen. Dette feltet danner et magnetisk mønster på diskens overflate. Dette mønsteret blir forskjellig avhengig om den elektriske pulsen var positiv eller negativ. Lesing fra disken foregår ved at det magnetiske mønsteret på disken beveges forbi spolen, og dermed induserer en elektrisk strøm i den. Denne elektriske pulsen vil da ha samme polaritet som den som skrev på disken hadde.

**Organisering**

En disk lagrer informasjon i spor langs disken. Disse sporene er sirkulære og har litt mellomrom mellom seg for å sikre at diskhodet ikke leser fra feil spor. Dersom diskhodet bommer litt på sporet vil signalnivået svekkes, og det er derfor det kreves ekstremt høy presisjonsnivåer av diskhodet. Hvert spor er igjen delt opp i sektorer. Diskhodet kan dermed lese spesifikk data ved å få oppgitt spornummer og seksjon. Hver slik seksjon lagrer vanligvis 512 bytes, altså 0.5KB. Dette er også den minste datamengden som leses/skrives på disk, da diskhodet vanligvis skriver/leser flere seksjoner enn dette. Disse diskene roterer med konstant rotasjonshastighet. Denne hastigheten oppgis i rpm, rotasjoner per minutt. Vanlige hastigheter er 5400, 7200 eller 10 000 o/min. På engelsk kaller de det for CAV (Constant Angular Velocity).

**Lagerkapasitet, aksesstid og overføringsstakt**

Lagringskapasiteten på disker øker stadig, men aksesstiden har stort sett vært den samme de siste 20 årene.

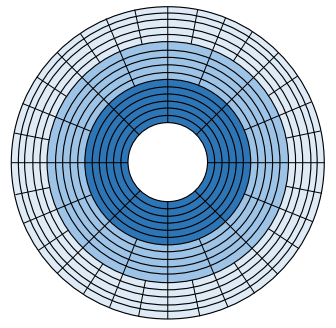
Nyere magnetiske disker har gjerne lagringskapasitet som måles i TB, terrabyte, altså 1000 GB. Prisen per GB med slik lagring er altså mye lavere enn prisen per GB med RAM. Grunnen til at man kan ha stadig større lagring er at lagringstettheten stadig øker. Man kan ha sporene på diskene tettere å tettere, og dermed lagre mer på en disk. Som nevnt tidligere stiller dette store krav til diskhodets presisjon.

Aksesstiden til disken bestemmes av to ting. Hvor lang tid det tar for diskhodet å bevege seg over rett spor, og hvor lang tid det tar for disken å rotere slik at rett data er under diskhodet. Aksesstiden kan altså variere ut i fra hvor langt diskhodet må «reise», og hvor mye disken må rotere. Resultatet av dette er at vi på de fleste disker må vente mellom 8 og 12 ms på første byte. Denne tiden avhenger som sagt av disken, men langt mer av hvordan data er organisert på disken. Det er derfor viktig å organisere data slik at så mye ligger på samme spor som mulig, for å unngå at diskhodet må reise for mye. Vi sier at det er viktig å *defragmentere* disken, det vil si å organisere informasjonen så den ligger sammenhengende på fortløpende sektorer.

Hvor fort kan man lese data når diskhodet første er i riktig spor, og riktig sektor? Dette avhenger av hvor raskt disken roterer, og hvor mange sektorer det er på disken. Vi så tidligere at det er vanlig å lagre 0.5 KB i hver sektor. På denne måten er det relevant hvor mange sektorer en disk har. La oss si at en disk har 300 sektorer, og rotere 7200 ganger i minuttet. For å finne ut hvor mye data vi kan lese fra disken i sekundet må vi først finne ut hvor mange ganger disken roterer i sekundet: 7200 / 60 = 120 o/min. Samtidig må vi finne ut hvor mye data som er lagret på en i et helt spor. Vi hadde 300 sektorer med 0.5 KB lagret i hver sektor. Dette gir oss 300 \* 0.5 KB = 150 KB i hvert spor. Da kan vi til slutt finne ut hvor mye data som leses i sekundet ved å gange antall rotasjoner i sekundet med antall bytes i hvert spor: 120o/sek \* 150KB = 18 000KB/s = 18 MB/s. Dette er altså overføringshastigheten fra diskhodet til IO modulen. Om det leses like raskt til minnet, avhenger av bussen mellom IO modulen og minnet. Disker oppgir ofte en overføringstakt ved vedvarende lesing i området 100-150 MB/s. Dette er ofte oppgitt fra ytterste spor, da man kan lese data vedvarende i en lengre periode her.

**ZBR, Zoned Bit Recording**

Som dere kanskje har skjønt, er det litt rart å dele opp i sektorer, når dette vil bety at lagringstettheten er mye større inne ved sentrum av disken enn ytterst. Vi deler derfor inn i grupper. De innerste gruppene har færre sektorer enn de ytterst. På denne måten kan vi ha ca. lik lagringstetthet over alt, og dermed få plass til mer data per disk.



Her ser vi hvordan de forskjellige fargene, gruppene, har forskjellig antall sektorer. Den innerste sporet har åtte, mens det ytterste har 32. Vanlige disker har langt flere sektorer i hver gruppe, men denne illustrasjonen viser tydelig hvordan ZBR fungerer.

**Winchester teknikk**

Siden man ønsker mest mulig data per disk, ønsker man også at sporene skal ligge så tett som mulig. Signalstyrken avhenger av hvor nærme diskhodet er disken, og om vi ønsker tette spor er det viktig at diskhodet ligger meget nært disken, slik at den ikke leser fra feil spor, eller avgir ett svakt signal. Armen som diskhodet er festet på er derfor utformet litt som en vinge slik at lufttrykket disken lager når den roterer, holder armen oppe. Armen vil da «sveve» ekstremt nær disken, uten å treffe den. Avstanden mellom diskhodet og disken er mellom 500 og 1000 ganger mindre enn et hårstrå. Dette er grunnen til at man skal være forsiktig med en PC når den er på, for at armen ikke skal røre disken.

### Optiske media

Med optiske media mener vi lagringsmedia som CD-ROM og DVD.

**CD-ROM**

De første CD platene for musikk kom i 1983. Dagens CD-ROM bygger på den samme teknologien, men har en mer avansert form for feilkorreksjon. En CD disk har en reflekterende overflate. Denne leses med en laserstråle ved at laseren lyser ned på platen og deretter reflekteres av flaten og inn i en fotodiode. Det er da intensiteten av det reflekterte lyset som spiller en rolle. CD flaten har nemlig noen forhøyninger, disse kalles på engelsk *pits*, ellers er overflaten plan. Binær informasjon lagres ved at fotodioden måler høy lysintensitet der CD-en er plan, men ikke måler noen ting dersom laseren treffer en pit. Disse forhøyningene er også grunnen til at CD plater er Read Only, det er ikke mulig for en laser å lage nye forhøyninger, eller fjerne gamle.

I motsetning til harddisken der all informasjon er lagret på separate spor, er informasjon på CD-ROM lagret i et spor som går i spiral ut fra sentrum. Dette kommer av at all informasjon på en CD er ment å leses fra start til slutt, som for eksempel musikk. På denne måten kan laseren lese fortløpende. Lagringstettheten er lik over hele platen.

Til forskjell fra harddisken roterer CD platen med varierende hastighet. Dette kommer av at bitsene som leses fra platen må komme med jevnt mellomrom for at for eksempel musikken skal spille jevnt. Dette løses ved at platen roterer raskere når hodet er nærmere sentrum. Likevel er hastigheten langs sporet, alltid den samme. Dette kalles CLV – Constant Linear Velocity.

**Lagringstetthet**

På CD-ROM har spiralen 22 188 omdreininger, og sporbredde på 0.8 mikrometer. Om man da skulle rettet ut spiralen ville den hatt en lengde på over 5.5 km.

Datablokkene på en CD-ROM angis med tidskode: minutt, sekund og blokknummer. Dette er også arvet fra musikk-CDen. En vesentlig forskjell mellom en musikk-CD og en CD-ROM er at det ikke gjør så mye om hodet leser en bit fra musikk-CDen feil, i med at vi ikke hører det. Det er derimot fatalt om en feil bit leses fra CD-ROMen siden dette gjør at programmet vårt ikke funger som det skal. CD-ROM har derfor langt mer feil-korreksjon.

**Aksesstid**

Aksesstiden på CD-ROM er mye lengre enn på harddisk. Dette kommer av at hodet som leser fra CDer er mye tyngre og tregere. Sidene blokkene ikke starter på konstante vinkler må hodet «lytte» seg frem til rett spor og start av blokk. Aksesstiden ligger som regel i området 100-300 ms.

**DVD**

DVDen er relativt lik CDen, men ble laget for å tilpasse film. Forskjellen er at DVD har høyere lagringstetthet. Dette er mulig siden laseren som leser fra DVDen har kortere bølgelengde. Dermed blir forhøyningene mindre og sporene kan ligge tettere.

**Brenning**

Man kan brenne egne CD eller DVDer, ved å ha en brenner og en brennbar CD-plate. Brenneren fungerer ved at man har en litt kraftigere laser på hodet. Denne er såpass sterk at den kan endre farge på et fargestoff som ligger på platen. Fargestoffet endrer farge fra gjennomsiktig til svart. Det er altså der det blir svart at man etterligner pits. Når man skal lese fra platen bruker man en mindre sterk laser.

### Halvlederbaserte sekundærlager

Hittil har vi sett på lagringsformer med mekanisk, bevegelige, deler. Dette er grunnen til at deres aksesstid er såpass høy, sammenliknet med for eksempel aksesstiden til primærminnet. Om vi ønsker lagringsmedier med kortere aksesstid, må vi bruke lagerteknologier som er full-elektroniske. Slike lagringsmedier er basert på halvlederteknologi. Som vi har sett er dette samme type teknologi som brukes i primærminnet. RAM og ROM er eksempler på slike lagringsmedier. Men vi må også huske at det som lagres i sekundærminnet skal «huskes» når vi slår av datamaskinen. Vi må da enten bruke ROM, eller utstyre datamaskinen med et batteri som opprettholder drivspenningen. I med at ROM er Read Only Memory, må vi i så fall finne en form for ROM man enkelt kan skrive til. Det er produksjonsmetodene for ROM typen som i senere år har blitt billigere og dermed mer vanlig. SSD er et eksempel på slik lagring. Denne typen lagring er fortsatt mye dyrere og har lavere lagringskapasitet enn magnetiske harddisker.

### SSD disker

Solid State Drive er et lagringsmedium som er fult elektronisk. Den burde dermed ikke kalles disk, men siden den har bruksegenskaper som ligner på harddisken sine, kaller vi den for en disk. I med at den er helt elektronisk er aksesstiden for å lese fra den, mye kortere, og man kan lese data langt hurtigere ved fortløpende lesing. Siden den ikke har noen bevegelige deler tåler den flere støt og bruker dessuten mindre strøm. Den er bare dyrere og har mindre lagringskapasitet enn harddisk.

**Oppbygging**

SSD er bygget opp av to deler, lagerenheten og styreelektronikken. Lagerenheten er satt sammen av minnebrikker som lagrer data, mens styreelektronikken sørger for at informasjonen kan aksesseres på en slik måte at operativsystemet skjønner at det er en hurtig harddisk.

**Lagerenheten**

Det vanligste i SSD lagre er å bruke ROM-typen flash-ROM for å lagre informasjon. Som vi så i primærminnet var hver minnebrikke bygget opp av mange celler. Disse cellene kunne aksesseres individuelt. Disse kalles da NOR-celler. Det er *ikke* disse som brukes i SSD. Her brukes nemlig NAND-celler der vi må lese, skrive, eller slette mange celler om gangen. I tillegg brukes ofte MLC (Multi-Level Cell) i SSD. Dette betyr at hver celle kan lagre flere bits, for eksempel 2.

Som sagt må man lese, skrive eller slette flere celler om gangen på SSD fordi den bruker NAND-celler. På SSD er derfor den minste datamengden man kan lese eller skrive 4096 bytes, altså 4 KB. Dette kalles en side (page). Når man skal slette celler kan man ikke bare slette en side. Man må nemlig slette hele «blokken» siden ligger i dersom man ønsker å slette siden. Vanligvis inneholder en blokk 128 sider. Ikke bare er dette en utfordring, men cellene slites også hver gang noe slettes fra dem. Disse to utfordringene gjør at styreelektronikken må være meget avansert i tillegg til at vi trenger programvare.

**Styreelektronikken**

Denne elektronikken har to oppgaver. Den ene er å administrere bruken av flash-minnet, for eksempel sånn at data ikke slettes fra de samme cellene om og om igjen, eller å passe på at data lagres spredt i lageret. Den andre er å kommunisere med resten av datamaskinen gjennom et egnet IO grensesnitt.

Når det kommer til administrasjonen må styreelektronikken må den passe på at SSDen lagrer informasjon jevnt utover minnebrikkene sine. Dette kommer av at den kan aksessere alle samtidig og dermed kan aksessere mer informasjon raskere om den ligger spredt. I tillegg må den passe på at cellene brukes jevnt og at sletting fra celler ikke påvirker ytelsen for mye.

Dersom en SSD skal erstatte en harddisk må den kommunisere på samme måte som harddisken. SSD bruker vanligvis SATA versjon 2.0 eller 3.0 som IO-busser. Den teoretiske båndbredden for disse er 300 MB/s og 600 MB/s. Enda raskere SSDD’er bruker kanskje PCI-Express X16.

SSD bruker litt lengre tid på å skrive enn å lese, og lengre tid på å slette. Likevel er den mye raskere enn harddisker. Særlig når det kommer til håndtering av mange små filer. Dette kommer av at harddisken her må flytte mye på den mekaniske armen sin. Her får vi så store forskjeller som 1 MB/s (harddisken) mot 50 MB/s (SSD).

## Cache

Som vi har sett tidligere klarer ikke minnet å levere instrukser like raskt som CPU utfører dem. Vi bruker derfor en teknikk som vi kaller cache.

Det er ofte praktisk å ting man bruker ofte, i nærheten av seg. Sånn fungere cache også. Vi laster nemlig de instruksjonene som brukes ofte, inn i et raskere minne. Nemlig et minne som er raskt nok til å holde tritt med CPU. Dette vil si at vi i stedet for å hente en instruksjon og utføre den, henter mange instruksjoner. Når CPU har utført en instruks sjekker den først cache for å se om neste instruksjon ligger der. Om den ikke gjør det må den vente til riktig instruksjon blir lastet inn i cache. I praksis ligger riktig instruksjon nesten alltid i cachen. På denne måten får vi utnyttet CPUens utrolige hastighet.

**Prinsippet om lokalitet**

Som vi har snakket om tidligere utføres et program sekvensielt. Det vil si at instruksjoner utføres i rekkefølge, en etter en. Unntaket er hvis programmet gjør «hopp» for eksempel ved å utføre en instruksjon flere ganger, eller ved å hoppe over noen instruksjoner. Disse instruksjonene lagres også sekvensielt i minnet. På denne måten kan vi si at det er meget sannsynlig at instruksjonen i adresse nr. 2 skal utføres etter at instruksjonen i adresse nr. 1 er utført. Det samme gjelder, i tillegg til instruksjoner, også data. Dette kommer av at man ofte bruker arrays og records. På denne måten lagres data som brukes i samme program nært hverandre.

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

**Romlig og temporal lokalitet**

Vi skiller mellom to typer lokalitet:

* Romlig lokalitet betyr at minnereferansene ligger nært hverandre. Altså at instrukser som utføres ligger nært hverandre i minnet. Sekvensiell utførelse er et eksempel på dette
* Temporal lokalitet betyr at minnereferanser brukes med korte mellomrom. Eksempler på dette er løkker der instruksjoner utføres flere ganger.

**Størrelse**

Cache er i dag ca. en promille av primærminnet, altså 1/1000. Denne lagringen skal altså inneholde de delene av minnet som er mest aktive. Størrelsen til cachen sammenliknet med minnet sier også noe om hvor effektivt prinsippet om lokalitet er.

### Virkemåte

CPU går til cachen får å se om lokasjonen allerede er kopiert fra minnet. Dersom den er det, leser CPU herfra. Dersom den ikke er kopiert til cachen, vil CPU gå direkte til minnet. Når den først finner frem til riktig lokasjon her, vil den ikke bare lese akkurat den lokasjonen den er ute etter, men også et fast antall lokasjoner i tillegg. Alle disse lokasjonene kaller vi en «blokk», og en slik inneholder altså et fast antall lokasjoner. Denne kopieres så over i cachen. Grunnet prinsippet om lokalitet, vil sjansen nå være stor for at de neste lokasjonene CPU trenger ligger i cache.

Siden CPU skal kunne ha såpass rask tilgang til cachen ligger cachen faktisk inne i CPU, den er en integrert del av prosessorkretsen. Cachen er bygget opp av SRAM, statisk RAM, som er mye raskere enn DRAM, den typen vi finner i primærminnet.

**Treffrate**

CPU aksesserer altså lokasjoner enten i cache eller i minnet. Dersom CPU må ut og hente blokker i minnet, kaller vi det bom. Her må CPU først lese en blokk inn i cachen, og deretter lese lokasjonen fra cache. Aksesstiden blir her lik aksesstiden til minnet pluss aksesstiden til cache. Dersom CPU finner det den leter etter i cache, har vi et treff. Aksesstiden vil da være lik aksesstiden til cachen. Vi ønsker hele tiden at cachen skal være så effektiv at vi får flest mulig treff. Vi kaller antall treff av totale aksesser, for treffrate. Denne vil over tid være mellom 80 og 98%. Når vi åpner et nytt program derimot, er den null. Dette kommer av at vi ikke har brukt noen av lokasjonene programmet bruker, tidligere. Vi må derfor laste inn de lokasjonene programmet bruker. Etter en kort stund vil treffraten være meget høy igjen.

Aksesstid ved treff er lik aksesstiden til cache Tc

Aksesstiden ved bom er lik aksesstiden til minnet pluss aksesstiden til cache (Tp + Tc)

Treffraten skriver vi som H. Denne er et tall mellom 0 og 1, der 0 bare er bom, og 1 betyr 100% treffrate.

Effektiv aksesstid er det vi bruker for gjennomsnittlig aksesstid over mange aksesser (Te). Denne kan vi sette opp som en formel:

Der Te er effektiv aksesstid, Tc er cachens aksesstid, H er treffraten og Tp er minnets aksesstid.

Av denne formelen kan vi se at treffrate lik 100% vil være det samme som Tc, og treffrate på 0% vil være det samme som Tc + Tp.

**Mer om virkemåte**

Når vi nå går videre må vi lære noen nye ting om hvordan minnet egentlig fungerer. I steder for å tenke oss at ord er lagret enkeltvis i minnet kan vi nå se for oss at enhver lokasjon tilhører en blokk. Når denne lokasjonen skal aksesseres hentes hele denne blokken inn i cache. I tillegg må vi vite at cache er bygget opp av linjer som hver kan hold en blokk. Cache kan altså holde flere blokker selv om den er mye mindre enn minnet. For at cachen skal hole styr på hvilke blokker som ligger på de forskjellige linjene må hver blokk også ha en form for indikator. Denne kalles en «tag» og lagres sammen med blokken.

**Blokkstørrelse**

En cache kan enten ha mange linjer med små blokker, eller få linjer med store blokker. Hva som lønner seg avhenger av flere ting, blant annet hva slags oppgaver prosessoren brukes til å løse.

Dersom man har stor grad av romlig lokalitet er det en fordel med store blokker slik at CPU kan jobbe lenge i en blokk. Dersom man nå får en bom, straffes man hardt fordi det tar lengre tid å hente inn en ny stor blokk. Dersom blokkene var mindre ville det tatt kortere tid å laste inn en blokk til cache og man ville dermed ikke straffes like hardt. På den annen side måtte man da kanskje oftere ut i primærminnet og hente inn nye blokker.

Små blokker lønner seg dersom man har større grad av temporal lokalitet. Altså at man hopper litt frem og tilbake i minnet, men bruker flere lokasjoner om igjen. På denne måten kan cache ha flere blokker fra forskjellige steder i minnet, og bruke lokasjonene i hver blokk om igjen og om igjen.

Moderne PC-er har landet på å bruke blokkstørrelse på mellom 32 og 64 bytes.

### Mapping

Når man skal kopiere en blokk til cache, må det bestemmes hvilken linje blokken skal ligge i. På den ene siden ønsker man enkelthet, men på den andre siden er det viktig at man ikke unødig kaster blokker ut av cachen

**Full-assosiativ cache**

Dette er cache der blokker som hentes inn kan legges i hvilken som helst linje. Dette er lite effektivt fordi det betyr at CPU må sjekke hver eneste linje i cachen på søk etter riktig blokk.

**Ikke-assosiativ/direkte-mappet cache**

Dette er det stikk motsatte av assosiativ cache. Her har hver blokk i minnet en fast plass. Siden det er flere blokker i minnet enn linjer i cachen betyr det at mange blokker har en linje som sin faste linje. På denne måten trenger CPU bare sjekke en linje når den skal finne ut om lokasjonen den søker ligger i cache. Her oppstår det en konflikt dersom to blokker som tilhører samme linje skal brukes. Da må man hele tiden kaste ut den ene og laste inn den andre. Dette er veldig lite effektivt, og kalles *thrashing*.

**Sett-assosiativ cache**

For å unngå trashing kan vi organisere linjene i cachen i «sett». Dette betyr at blokkene ikke kan ligge hvor som helst i cachen, men at de heller ikke bare kan ligge i en linje. Dette gjør at konkurrerende blokker kan ligge i cache samtidig ved at de deler på et sett. Vi kaller en cache 2-veis dersom den har 2 linjer i hvert sett. Den kan også være 4-veis. Da har den 4 linjer per sett. Det er slik sett-assosiativ vi i praksis nesten alltid bruker i en datamaskin. Det vanligste er å bruke 2, 4, 8 eller 16-veis cacher.

**Utskiftningsmetoder**

Når cachen er full eller bare når man må bytte ut en blokk i cachen, må man velge hvordan dette skal gjøres. Det finnes en del metoder for dette:

LRU: Least Recently Used bytter ut den det er lengst siden ble aksessert.

FIFO: First In, First Out bytter ut den som har vært i cachen lengst, altså den som kom inn i cachen først.

LFU: Least Frequently Used bytter ut den blokken som har blitt aksessert færrest ganger.

Random: Bytter ut en tilfeldig blokk.

Det er viktig at denne utskiftningen skjer raskt dersom cache skal lønne seg. Derfor bruker man vanligvis enkle versjoner av disse. En form for tilfeldig utskiftning er meget vanlig.

**Skriving til minne**

Vi har nå sett på lesing fra cache. Men hvordan fungere skriving til minnet nå som vi har cache. Det finnes to løsninger:

* Write Through: CPU skriver til både cache og minnet. Fordelen her er at minnet alltid er oppdatert. Ulempen er at cachen ikke avlaster CPU. Den må fortsatt skrive direkte til minnet, og det hadde vært det samme som å ikke ha cache.
* Write Back: CPU skriver bare til cache. Fordelen med dette er at det er raskt, fordi vi får lav buss-trafikk. Her oppdateres ikke innholdet i minnet før en linje skiftes ut i cachen. Ulempen er at denne utskiftningen tar lengre tid, og at CPU må vente i mellomtiden. Dessuten møter vi på problemer med IO utstyr.

**Cache koherens**

Eksempler på problemer med Write Back er IO utstyr som skriver data til minnet. Dersom CPU har skrevet noe til minne, men dette fortsatt bare ligger i cache, vil det ikke være samsvar mellom primærminnet og cachen. Vi sier at det ikke er koherens. Et slikt problem er en av grunnene til at cache-mekanismen er ofte er like komplisert som resten av CPUen.

En vanlig løsning er å la cachen tømme seg (cache flush) før slike IO-operasjoner utføres.

Man kan nå lure på når cache egentlig lønner seg. Dersom man har helt tilfeldig data-aksess hjelper cache lite, men dersom man har høy grad av lokalitet, vil cache hjelpe betydelig. Grunnet at det varierer såpass når cache er nyttig, har vi utviklet flere typer cache.

**Fler-nivå cache**

I stedet for å øke størrelsen på cachen kan man ha flere nivåer på cache. Man kan ha en mindre cache og en større. Den minste er den raskeste og kalles Level 1 eller L1 cache. Den litt større cachen er litt tregere, men fortsatt mye raskere enn DRAM i minnet. Denne kalles L2 cache. Noen nyere datamaskiner har også L3 cache som er enda litt større og tregere enn L2 cachen. Denne er også fortsatt mye raskere enn minnet. Algoritmen for hvilke av cachene som brukes er meget avanserte, men i grove trekk er det slik at lokasjoner som blir brukt oftest lagres i L1 cachen, og lokasjoner som brukes mindre lagres i høyere nivåer cache.

**Splittet cache kontra enhetlig cache**

Tradisjonell cache brukes både til data og instruksjoner. Dette kalles en enhetlig cache. Denne avpasser automatisk antall instruksjoner kontra data i cachen. Denne har fordelen at den er dynamisk og lett tilpasser seg program som enten har mest data eller mest instruksjoner. I tillegg er det enklere å designe og implementere en cache enn to.

En annen mulighet, motstykket, blir da å ha en splittet cache. Her har vi to selvstendige del-cacher, en til instruksjoner og en til data. Dette er en stor fordel dersom prosessoren får bruk for både data og instruksjoner samtidig. Dette kan være gunstig i tilfeller der en prosessor utfører flere instruksjoner samtidig. For eksempel i superskalare prosessorer, prosessorer som inneholder mer enn en prosesserende enhet.

**Egen kontra felles (delt) cache**

En moderne prosessor har gjerne flere kjerner bygget inn i samme krets. Operativsystemet vil oppfatte dette som 4 prosessorer. Spørsmålet er da om de skal dele cache eller ha hver sin. På nyere prosessorer har hver kjerne sin egen L1 og L2 cache mens alle deler L3 cachen.

**Sett størrelse**

Å ha stor sett størrelse vil resultere i mindre sannsynlighet for thrashing. Dette kommer av at flere konkurrerende blokker kan ligge i cachen samtidig. Likevel krever dette mer kontrollelektronikk og cachen blir vanskeligere å lage. Alt i alt resulterer dette i at straffen ved å bomme blir større ved å øke sett-størrelsen. Det oppstår dermed en avveining mellom straffen for å bomme kontra gevinsten for å treffe.

**Utvikling av cache**

I senere år har man tatt i bruk mer og mer cache, og flere nivåer. Man bruker større og større total mengde cache, og antall nivå øker.

## Moderne primærminne

De forskjellige bussene har forskjellig overføringshastigheter. Som vi har sett er for eksempel minnebussen en synkron buss. Det betyr at den har en klokkepuls som slår med fast frekvens. På hvert slag kan den overføre data. En slik klokkepuls har også prosessoren, men denne er mye raskere. Vi skal nå se hvordan primærminnet, minnebussen og L2 cachen i prosessoren samarbeider.

**Rask repetisjon**

Vi husker hvordan henting av instruksjoner kunne deles inn i tre trinn:

* Adressebussen overfører en adresse til minnet. Dette er adressen CPU ønsker å lese fra eller skrive til. Denne overføringen tar en bussklokkesyklus.
* Deretter må vi vente til minnet finner frem til adressen. Dette tar minst en klokkesyklus, men ofte flere.
* Til slutt må data overføres fra minnet tilbake til CPU. Dette tar en bussklokkesyklus.

Vi ser at den totale tiden dette tar, avhenger av to ting. Aksesstiden til minnet, og klokkefrekvensen til bussen. Som vi så i leksjonen om primærminnet vil DRAM typisk ha aksesstid rundt 35-70 ns. Minnebussen har som regel frekvens i GHz området. Man kan kanskje tenke ut i fra disse trinnene at en prosessor bruker ca. 4-5 klokkesykluser per instruksjon. Det vi ikke har tatt høyde for i denne modellen er cache. Vi må nemlig huske at prosessoren ikke henter inn en og en instruksjon, men hele blokker av dem.

**Prosessoren med cache**

Når vi skal se på overføringen fra minnet, men denne gangen med cache, er det greit å vite hvor stor en typisk blokk i cache er. På moderne PCer brukes vanligvis blokkstørrelse på 64 bytes.

Når det kommer til minnebussene, er det vanligvis 64-bits busser. Det vil si at de har 64 linjer som alle overfører 1 bit. Bussen kan altså overføre 8 byte per klokkepuls. Men siden en blokk er 64 bytes ønsker vi jo å overføre 64 bytes. Vi må altså overføre 8 bytes, 8 ganger. Tidligere så vi at CPU brukte ca. 2-4 klokkepulser på å overføre bare en instruksjon. Hvis vi nå skal gjøre 8 overføringer tar jo dette veldig lang tid. Likevel fungerer ikke overføringene helt på samme måte. Nå som vi skal overføre en hel blokk holder det at vi vet startadressen. Vi overfører nemlig de neste minnelokasjonene fortløpende så lenge det er plass i blokken. Hver overføring etter den første tar da bare en klokkepuls. Denne kontinuerlige overføringen kalles *burst mode*, på norsk *kontinuerlig aksess*. Dette betyr at prosessen nå består av følgende trinn:

* Overføre en startadresse. Dette tar en klokkepuls.
* Vente på at minnet finner frem. Dette avhenger av aksesstiden til minnet, men vi må vente et helt antall klokkepulser.
* Overføring skjer kontinuerlig fra startadressen. En minnelokasjon for hver klokkepuls inntil blokken er full.

Hvor høy bussfrekvens vi har påvirker ikke aksesstiden til minnet annet enn at høy bussfrekvens fører til at vi får flere slag mens vi venter på minnet. Derimot vil overføringen gå mye fortere med høy frekvens når minnelokasjonen først er funnet. Dette kommer av at vi kan overføre en lokasjon på hver klokkepuls noe som gjør at høy frekvens bare er en fordel.

**Utviklingstrekk for RAM-minne**

Vi har sett at prosessorene blir dobbelt så kraftige annethvert år ca. De kan altså utføre dobbelt så mange instrukser hver 2. år. Dette forutsetter at minnet også kan levere dobbelt så mange instruksjoner hvert år. Det tar likevel hele 10 år før man klarer å halvere aksesstiden til DRAM-minnet. Grunnen til at man da likevel klarer å holde tritt med prosessorene er at man kan gjøre styreelektronikken såpass god at den takler høye bussfrekvenser. Dette betyr i praksis at aksesstiden ikke forbedres noe særlig, men at minnet takler såpass høy bussfrekvens at den kan overføre minnelokasjoner raskt nok til at prosessoren mates med nok instruksjoner.

**SDRAM**

Får å få burst til å bli mest mulig effektiv ønsker vi høyest mulig bussfrekvens. Dette krever at samarbeidet mellom minneelektronikken og minnebussen er meget godt. Derfor har vi såkalt synkron DRAM. Den er synkron i den betydning at bussen og styreelektronikken i minnet samarbeider godt. Dette betyr at vi kan øke frekvensen på minnebussen uten noen store konsekvenser. Det er slike synkrone typer RAM som brukes på PCer i dag. Hver enkelt minnecelle er fortsatt DRAM, vi kaller derfor disse DRAM brikkene for SDRAM, synkron dynamisk RAM. SDRAM minnebrikkene er ikke lengre i vanlig bruk siden de ble avløst av en nyere type DDR-SDRAM

**DDR-SDRAM**

Denne RAM typen er dobbel så rask som SDRAM og DDR står derfor for Double Data rate. Dette kommer av at denne typen RAM kan overføre data under burst, både i overgangen mellom lav og høy spenning, men også i overgangen tilbake fra høy til lav. Den kan dermed gjøre kontinuerlig overføring dobbelt så raskt som vanlig SDRAM. Aksesstiden derimot er den samme, da økt frekvens ikke påvirker denne. Selv om vi kan overføre på «ned-pulser» oppgir vi fortsatt tiden det tar å komme i gang på i hele pulser, altså bare opp-pulsene.

Senere har det kommet DDR2, DDR3 og DDR4. Disse fungerer på samme måte, men tillater bare høyere klokkepulser.

Merk at vi skiller mellom klokkepuls og overføringstakt når vi snakker om DDR. Dersom en DDR tillater klokkepuls på 400 MHz, har den en overføringstakt på 800 MT/s (Million transfers per second).

Det er brikkesettet i en maskin som bestemmer hva slags RAM du kan bruke.

### Timing parametere

Som vi vet fra før aksesseres hver enkelt celle i minnet ved at man første setter en rad, og deretter en kolonne. Når man har valgt riktig rad, settes et kontrollsignal som kalles RAS (Row Access Strobe). Dette signalet varsler om at rekke-adressen er gyldig. Deretter kan man dekode kolonnen. Når riktig celle er funnet settes et kontrollsignal som varsler at kolonne-adressen er gyldig. Dette signalet heter CAS (Column Access Strobe).

Tiden et tar fra RAS er satt til raden er aktivert kalles RAS-to-CAS-delay, og angis i klokkesykluser. VI skriver dette TRCD.

Tiden det tar fra CAS er satt til riktig celle er tilgjengelig kaller vi CAS-latency. Også denne oppgis i klokkesykluser. Denne skriver vi CL. Tiden det tar å lese en celle dersom ingen rekker er aktivisert vil da være lik: TRCD + CL.

Dersom feil rekke blir aktivisert vil det ta litt tid, før man kan prøve å aktivere en ny rekke. Tiden det tar før man kan aktivere en ny rekke kalles TRP å aksessere en celle. Den totale tiden det tar å aksessere en celle dersom feil rekke aktiveres blir da: TRP + TRCD + CL.

TRAS er tiden fra en bank aktiviseres til man kan sette RAS. Den totale ventetiden dersom feil bank blir aktiviser vil da være: TRAS + TRP + TRCD + CL.

Det at vi har disse forskjellige tidene betyr at en helt vilkårlig aksess i minnet sannsynligvis vil ta TRAS + TRP + TRCD + CL klokkepulser. Dette er verste fall. På den andre siden, vil høy grad av lokalitet gjøre at vi bare må vente CL klokkepulser. Dette vil tilsvare langt mindre enn 35-70 ns som er standard aksesstid for minnet. Grunnen til at vi da oppgir aksesstiden til minnet som 35-70 ns er at man må oppgi aksesstiden til en helt vilkårlig lokasjon.

Vi kan dermed konkludere med at det tar litt tid å klargjøre minnebrikken, men at DDR-SDRAM klarer å spy ut data relativt radig når alle overføringene skjer fra minnelokasjoner som ligger tett. Hvor hurtig den kan spy ut avhenger av hvor sist aksess var.

**Hvordan alle disse tidene oppgis**

Ofte ser man CL, TRCD, TRP, og TRAS oppgitt på formen 5-5-5-15. Det er altså snakk om antall klokkepulser.

## Brikkesettet

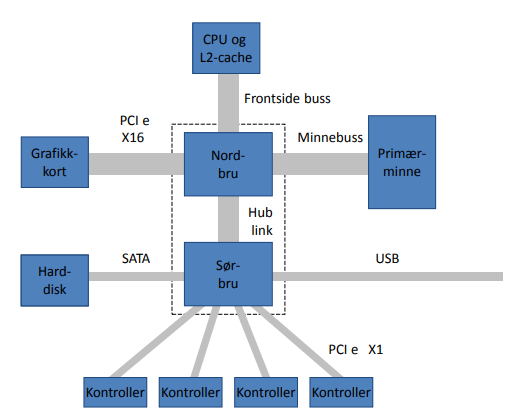
Grunnet at moderne datamaskiner har utrolig mange komponenter for dataoverføring og datalagring, i tillegg til mange IO enheter, trenger vi elektronikk som sørger for at alle disse komponentene blir utnyttet effektivt. Brikkesettet er delen so kobler sammen delene og som sørger for god administrasjon. Brikkesettet har mange funksjoner:

* Prosessorstøtte. Ulike brikkesett støtter ulike prosessorer med ulike klokkefrekvenser.
* Primærminnestøtte. Brikkesettet støtter ulike typer minnetyper som SDRAM, DDR, DDR2…
* Busstøtte. Brikkesettet støtter bussene.
* Tastaturkontoller
* USB-kontroller
* Diskkontroller. Nye brikkesett inneholder en SATA-harddiskkontroller
* Nettverks-kontroller, slik at man slipper eget nettverkskort
* Lyd-kontroller, slik at man slipper eget lydkort.

De siste er eksempler på kontrollere man har kunnet integrere i en og samme brikke grunnet at man har fått mindre transistorer. Tidligere var lyd og nettverk egen IO moduler.

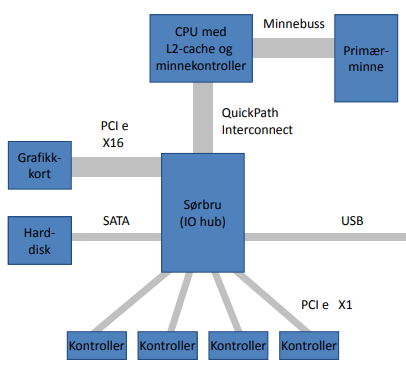
På moderne maskiner består gjerne brikkesettet av to hovedkretser. En nord-bro som tar seg av den raskeste kommunikasjonen, den som går mellom minnet, CPU og for eksempel grafikkortet. Den andre, Sør-broen, tar seg av kommunikasjon med IO kontrollere. Vi kaller gjerne bussen mellom nord-broa og L2 cachen for *frontside buss* og bussen mellom de to broene for *hub-link*.

Da vi lærte om hovedbussen, så vi at dette var en svitsjet buss. Her kunne to og to komponenter kommunisere gjennom en svits som tildelte forbindelser. Dersom to komponenter trengte raskere båndbredde kunne svitsjen tildele flere forbindelser, flere *lanes*. Denne svitsjen sitter i brikkesettet. Den ene forandringen vi gjør er at vi har to slike svitsjer. En på nord-broen, og en på sør-broen. På denne måten kan vi beholde den modellen vi snakket om tidligere, der nord-broen tar seg av raskere forbindelser, for eksempel X16 (16 lanes), mens sør-broen tar seg av de litt tregere, for eksempel X1 (1 lane). Dette illustreres på figuren under.



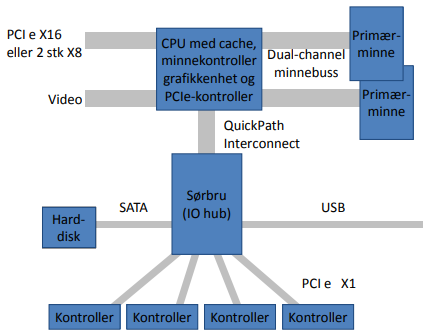
**Integrert minnekontroller**

Som vi ser på figuren over blir frontside bussen lett en flaskehals i denne maskinen. Det kommer av at både primærminnet og grafikkortet kommuniserer gjennom denne bussen, og at begge disse krever høy båndbredde. Som nevnt tidligere lar små transistorer oss integrere mer og mer inn i prosessoren. Derfor har vi flyttet spå godt som hele nord-broen inn i prosessoren og integrert en egen minnekontroller. På denne måten kan vi ha to busser inn til prosessoren, en 64 bits minnebuss, og bussen fra sør-broa, der all annen trafikk kommer inn; det vil si all IO. Det nye oppsettet vil da ser mer ut som på figuren under. Merk også at sør-broa her har PCIe X16.

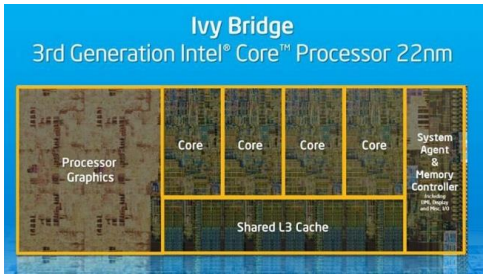


Forskjellige produsenter har valgt ulike løsninger for denne IO bussen som går mellom sør-broa og CPU.

I de nyeste prosessorarkitekturene har vi integrert alle kontrollere for hurtig trafikk inn i prosessorkretsen. Nå sitter både kontroller for grafikk og PCIe X16 kontrolleren i prosessoren. I tillegg benyttes det flerkanals minnebuss, vanligvis 2-kanals. Det vil si at vi har 2 minnebusser, og dermed kan kommunisere med to minnekort samtidig. Dagens arkitektur ligner på figuren under.



En moderne prosessor er ofte utformet som på figuren under. Prosessoren under har fire kjerner, 3-nivå caching, integrert grafikk-enhet, og en integrert 2-kanals minnekontroller. Hver kjerne har sin egen L1 og L2 cache, men de deler alle L3 cachen. System Agent er det intel har valgt å kalle de integrerte delene som tidligere lå på nord-broa.



## Lagerhierarki

Når vi nå skal se på lagerhierarki er det viktig at vi ser på noen begreper i forbindelse med lagerteknologier vi tidligere har sett på.

Overføringsenhet: Mens primærminnet overfører noen få ord ved lesing eller skriving, overfører sekundærminnet store blokker når det først overføres. Vanlige størrelser på disse blokkene er 512, 1024 og 2048 bytes.

Sekvensiell aksess: For noen lagringstyper er det eneste effektive, å aksessere data sekvensielt. Eksempler på dette kan være kassetter eller CD-er, der innholdet også er ment til spilles av i rekkefølge. Direkte vilkårlig aksess ville i slike tilfeller tatt mye tid, fordi det er vanskelig å finne frem til riktig «lokasjon».

Direkte aksess: For de fleste vanlige sekundærlagringstyper kan vi aksessere data direkte. På for eksempel en disk kan vi flytte diskhodet akkurat dit vi ønsker å lese data, og deretter starte en sekvensiell avlesning. Det å finne frem til rett startlokasjon kan ta litt tid, men selve lesingen går raskt.

Vilkårlig aksess: I primærminnet må vi ikke posisjonere oss før vi kan lese/skrive. Her kan vi lese eller skrive hvor som helst i minnet i vilkårlig rekkefølge. Den eneste sekundærminne-teknologien som gir oss denne muligheten er SSD-disker.

Aksesstid: Tiden det tar fra CPU etterspør data, til denne dataen er tilgjengelig. For tregt sekundærminne kan det være av betydning om aksesstiden er oppgitt fra når data begynner å strømme fra sekundærminnet, eller når data er ferdig med å strømme inn fra sekundærlageret.

Overføringshastighet ved vedvarende lesing: Uansett hvor lang tid det tar å aksessere en lokasjon, vil det ved store overføringer spille en viktig rolle hvor fort lagringsmediet klarer å overføre når den først begynner.

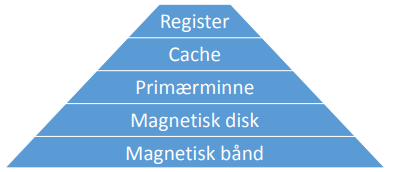
**Lagerhierarki**

Vi har sett at det finnes forskjellige lagringsmedier på en PC. Registrene i CPU lagrer data i det de brukes, minnet lagrer data i det et program kjører, mens sekundærminnet lagrer data så lenge vi vil, på en disk. Vi kan altså skille mellom hvor lenge de lagrer noe, hvor mye de kan lagre, og hvor lang tid det tar å aksessere dataene de lagrer.

Generelt kan vi komme med følgende påstander:

* Jo hurtigere lagertype, desto dyrere er den per bit.
* Jo høyere lagringskapasitet, desto billigere er den per bit.
* Jo høyere lagringskapasitet, desto langsommere er den.

Vi må altså bruke flere typer lagring i en maskin. Vi kan dermed sette opp et hierarki for lagringstypene. Vi kjennetegner lagrinstypene ved at:

* De synker i pris per bit de lagrer
* De synker i hastighet, dvs aksesstiden øker
* Øker i lagringskapasitet
* Blir sjeldnere aksessert av CPU

Tidene for aksessering kan variere veldig. Det tar godt under 100 ns å aksessere primærminne, mens det tar ca 10 ms å aksessere sekundærminnet. Bånd kan det ta sekunder eller minutter å aksessere.

Prisene på lagertypene ser vi generelt synker, men på kort sikt ser vi at markedet spesielt for primærminne er ustabilt, og at prisene kan variere.

**Bruk av forskjellig lagerteknologier.**

Man skulle ønske at man kunne brukt en type lagerteknologi, men dette går ikke da vi ikke har funnet en type som har stor nok lagringskapasitet, er rask nok, billig nok, og beholder data når vi slår av maskinen. Derfor har vi flere typer lagerteknologier. Og siden forskjellene blir større mellom den hurtigste og den langsomste lagerteknologien, har vi stadig flere trinn mellom dem

Registrene. Minnet som er direkte aksesserbart av CPUen. Disse er en viktig del av CPU.

L1-cache. Cache som holder tritt med CPUens hastighet. Denne er innebygget i CPUens kjerne. Aksesstiden kan være rundt 1ns. Denne er bygget opp av hurtig SRAM.

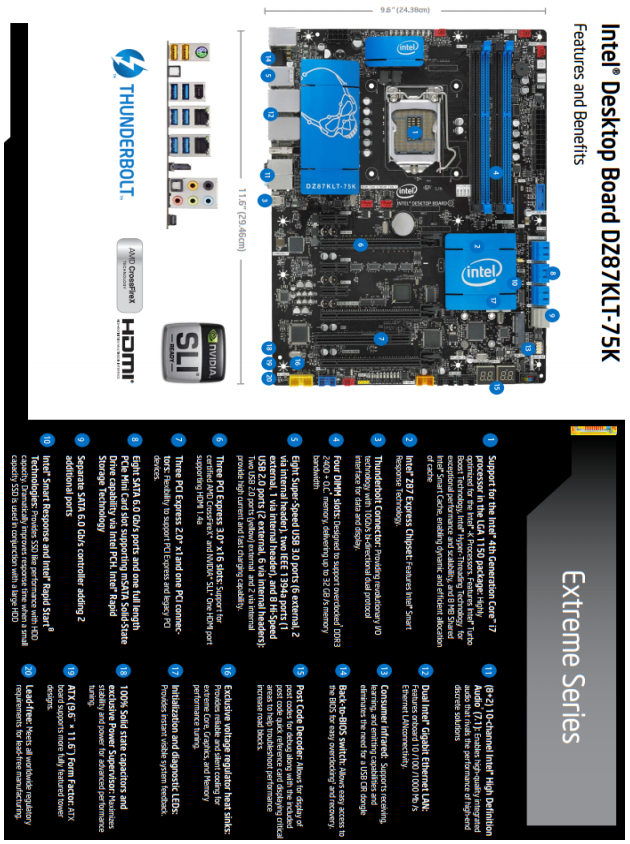
L2-cache. Dette er en større cache som er bygget opp av litt tregere SRAM. Denne har kanskje en aksesstid på mellom 2 og 10 ns.

Primærminne. Bygget opp av DRAM, men mye billigere enn SRAM. Aksesstider er typisk mellom 35 og 70 ns.

Sekundærlager. SSD disker, harddisker og optiske teknologier som DVD-drev. Mye billigere enn DRAM, men mye langsommere. Typisk aksesstid for harddisker er ca. 10 ms. SSD er ca. 100 ganger raskere, altså 0.1 ms. Optiske drev er omtrent 10 ganger langsommere enn harddiseker.

Huska at prinsippet om lokalitet gjelder for alle lagertyper og dermed kan påvirke akesesstidene.

**Et moderne hovedkort**



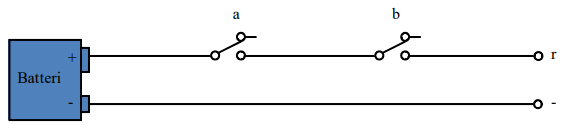
Her ser vi alle komponentene på et moderne hovedkort. Vi vil kjenne igjen flesteparten av dem. Nr.1 er prosessor sokkelen der prosessoren går. Den andre store blå brikken er brikkesettet. De fire minnesoklene ser vi på nr.4. De er farget i to forskjellige farger fordi det er dual channel DDR3. Dette husker vi fra tidligere betyr at vi i praksis har to minnebusser som leser fra to forskjellige minnebrikker samtidig. Brikkesettet har kontrollere for USB og firewire som vi ser i punt 5. Sokler for PCIe X16 finner vi i punkt 6 og PCIe X2 er nr. 7. Brikkesettet har også SATA diskkontroller med fleire sokler (8) og (9). Brikkesettet har også innebygget lydkort (11) og ethernett nettverkskort (12).

## Digital elektronikk

Med digital elektronikk mener vi binær elektronikk. Som vi vet bruker datamaskinene 0ere, ingen spenning, og 1ere, spenning, for å overføre data. Likevel legger dette bare til rette for enkle operasjoner som addisjon og subtraksjon. For at prosessoren skal kunne utføre mer avanserte operasjoner som multiplikasjon og divisjon, trenger vi porter. De vanligste portene er OG-, ELLER- og NOT-porten.

**OG-porten**

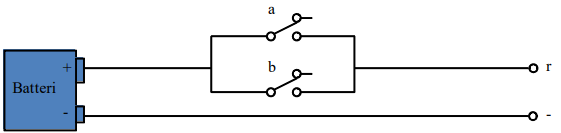
Denne porten tar inn to signaler og en utgang. Den setter utgangssignalet til 1 bare dersom begge inngangssignalene er 1. En krets som viser dette kan illustreres på følgende måte:



Her ser vi at både a og b må være 1 (sann, føre spenning) for at r, utgangssignalet skal være 1.

**ELLER-porten**

Denne porten setter utgangssignalet sitt til 1 så lenge minst ett av inngangssignalene er 1. Dette kan illustreres med følgende krets.



Her ser vi at bare en av de to bryterne a eller b trenger å føre spenning for at r skal få verdi lik 1. Dersom b ikke er lukket, går bare all spenningen gjennom a.

**NOT-porten**

Alt denne porten gjør er å sette utgangssignalet til det motsatte av inngangssignalet.

**Reléer**

I stedet for å ha brytere vi selv må skru av og på ønsker vi å ha brytere som slår seg av og på ved hjelp av elektriske signaler. Et rele er en mekanisk bryter der bryterarmen åpnes og lukkes ved hjelp av et magnetfelt som slås av eller på. Dette er bedre enn en bryter vi selv må slå av og på, men problemet er at vi trenger en bryter som kan slås raskere av og på. Problemet med reléet er at den bare vi stå stille i en posisjon midt mellom av og på, dersom man prøver å slå den av og på flere 1000 ganger i sekundet.

**Transistoren**

Løsningen på dette, er transistoren. Svitsjetiden til en slik er nede i pikosekunder. En transistor fungerer ved at man ved hjelp av en styrestrøm kan velge hvor godt den kan lede strøm. Transistoren kan enten brukes som forsterker eller som bryter.

**Transistoren som forsterker**

Transistoren er brukt i forsterkere til musikkanlegg. Den fungerer ved å ta et svakt inngangssignal som styresignal. Den får også inn et sterkt signal. Grunnet at man kan velge hvor godt transistoren skal lede strøm kan den bruke det svake signalet til å variere det sterke signalet. På denne måten vil de to signalene være helt like, men det ene sterkere enn det andre. Det sterke signalet kan nå for eksempel gå til en høyttaler.

**Transistor som bryter**

Her lar vi styresignalet være av eller på, ikke noe imellom. Transistoren vil da fungere som en god leder, eller som en isolator. I en prosessor kan du finne flere millioner transistorer, på et areal på størrelse med en fingernegl. Disse transistorene virker som brytere. I slike prosessorer finner vi også millioner av logiske porter sånn som OG, ELLER og NOT-porten. Disse slås sammen i såkalte kombinatoriske kretser.

Måten man har klart å gjøre prosessorer raskere og raskere er ved å lage transistorene mindre. En prosessor fra 2014 har flere milliarder transistorer, og nå er det tallet økt ytterligere. Nå begynner vi å nærme oss en grense der transistorene begynner å nærme seg dimensjonene til enkeltatomer. Dette gjør at andre fysiske lover gjelder. For å komme videre fra dette, er nano-teknologi vår neste mulighet.

## CPUens deler og virkemåte

Vi har nå sett litt på alle komponentene i en datamaskin. Nå er det på tide å gå dypere inn på datamaskinens hjerne, CPUen.

Som vi så på tidligere består instruksjonssyklusen til CPUen av to hovedtrinn, men også ett trinn til. De to første består av å første hente en instruksjon, og deretter utføre den. Det tredje trinnet består av å sjekke om det har kommet noen avbrudd. Siden dette skjer relativt sjelden sier vi at de to første trinnene er de trinnene vi vanligvis veksler mellom.

**De viktigste delene i CPUen**

Vi har sett litt på hvordan CPUen er satt sammen tidligere, men skal nå gå mer i dybden. De viktigste delene av CPU er:

* Registerblokken, altså samlingen av alle registrene på CPU.
* ALU (Aritmetisk/logisk enhet)
* Kontrollenheten
* Klokkekretsen
* Intern CPU-buss

De ulike komponentene kommuniserer over den interne CPU-bussen. De ulike instruksjonene utføres på starten av en klokkepuls. Hvor mange klokkepulser det tar å utføre en instruks varierer fra instruks til instruks. Hele utføringen skjer under kontroll av kontrollenheten som setter opp de rette styresignalene til enhver tid. Sett fra en programmerers ståsted er registerblokken den viktigste da det er denne en må forholde seg mest til.

**Registerblokken**

Når vi sier registerblokken mener vi alle registre i en CPU. Hvor mange registre en CPU har, varierer.

Registre som er tilgjengelige for brukerprogrammene:

* Generelle registre. Disse brukes til data som CPU trenger under utføring.
* Flagg-registeret. Dette er et register som bare kan leses og brukes til å angi statusen til CPU.
* Adresseregistre. Dette er registre som peker på forskjellige lokasjoner i minnet. Det finnes forskjellige typer av disse.

Registre som ikke er tilgjengelige for brukerprogrammene:

* Programteller
* Instruksjonsregister
* Registre som brukes av operativsystemet som kun kan aksesseres av privilegerte instruksjoner
* Interne registre som brukes til bufring under utførelse av instruksjoner.

**Kontrollenheten**

Alle aktiviteter som skjer inne i CPU styres av kontrollsignalene til kontrollenheten. Dette kan være signaler for hvilken operasjon ALU skal utføre, eller signaler som åpner/lukker for dataflyt mellom to registre.

**ALU**

ALU er den komponenten som faktisk utfører operasjoner i CPU. Den har to innganger og en utgang. Både inngangene og utgangen er registre. ALU gjør forskjellige operasjoner med bitmønstrene som den får via inngangen, og presenterer resultatet i registeret koblet til utgangen. Det er kontrollsignaler fra kontrollenheten som bestemmer hvilken operasjon som skal utføres.

**Klokkefrekvensen**

Enhver aktivitet i CPU settes i gang av en klokkepuls i form av en elektrisk puls. Frekvensen til de fleste prosessorer er i området 300-4000 MHz. Dette betyr at det aktiviseres en ny handling mellom 300 millioner og 4 milliarder ganger i sekundet.

**Intern CPU-buss**

Dette er en parallell buss med samme bussfrekvens som klokkehastigheten til CPUen. Dette kan den gjøre fordi den er utrolig rask og fordi den overfører data over veldig korte avstander. Den brukes til å frakte data mellom registre og ALUen sine innganger og utganger.

**Mikrooperasjoner**

Mikrooperasjoner er den fineste inndelingen vi bruker på CPUens arbeid inntil videre. Slike mikrooperasjoner kan være å flytte data mellom registre, å gjøre enkle operasjoner i ALU eller å flytte data mellom registre og minnet eller mellom registre og IO-porter.

Å flytte data mellom registre skjer ved at kontrollsignaler som tillater flyt av data inn/ut av et register, settes. Det kan for eksempel være at signaler som tillater flyt ut av register 1 og inn i register 2 settes.

Denne metoden kan blant annet brukes for å overføre data til ALUens innganger som også er registre. Deretter kan kontrollsignaler som bestemmer hvilken operasjon som skal utføres, settes.

Når CPU skal kommunisere med resten av datamaskinen gjennom systembussen bruker den egne register som har som eneste oppgave å gjøre linjene på systembussen ledig for CPU.

* ABR-register (AdresseBuss Register) register som CPU kan legge en adresse på dersom den ønsker data fra en lokasjon i minnet.
* DBR-register (DataBuss Register) register som CPU kan skrive/lese data på/fra.

## Kontrollenheten

Som vi så i forrige kapittel er det kontrollenheten som sørger for at riktige operasjoner blir gjort i CPU. Den har hovedsakelig to oppgaver. Den ene er sekvensiering. Dette går ut på å passe på at CPU utfører en rekke mikrooperasjoner i rett rekkefølge. Den andre er eksekvering (execution). Denne går ut på å sørge for at mikrooperasjonene blir utføre. Det kan enheten gjøre ved å sette styresignaler til ALU, eller åpne/lukke for datastrøm mellom registre.

Selve kontrollenheten virker litt på samme måte som ALUen, men har litt andre oppgaver. Den har nemlig forskjellige innganger som bestemmer hvilke utgangssignaler som skal settes. En av disse inngangssignalene er IR, altså selve instruksen som CPU skal utføre. Det andre inngangssignalet er klokkesignalet som bestemmer når kontrollenheten skal sette signaler for utførelse.

Selve kontrollenheten er enten hardkodet eller mikroprogrammert. Dersom den er hardkodet har den et bestemt utfall for alle mulige inngangssignaler. Det vil si at man har en kombinatorisk krets som er hardkodet for alle mulige utfall. En mikroprogrammert kontrollenhet er laget slik at den bryter en instruks ned i mikroinstrukser, og deretter setter de kontrollsignalene som har spenningsverdien 1 og bryter de som har spenningsverdi 0. Fordelen med hardkodede kontrollenheter er at de er meget raske. Ulempen er at de er kompliserte. Mikroprogrammerte kontrollenheter er litt tregere, men det er til gjengjeld enklere å implementere kompliserte instruksjoner. Tidligere har det vært et stort skille mellom de to typer prosessorene CISC (Complex instruction Set Computer) og RISC (Reduced Instruction Set Computer). I dag benytter man en form for blanding av de to.

## Moderne prosessorarkitektur

**Pipeline** er en teknikk der en prosessor kan utføre flere instruksjoner samtidig, men hver instruksjon er på forskjellige trinn. Dette kan sammenliknes med et samlebånd. Mange ting kan ligge på samlebåndet samtidig, men bare en av disse gjenstandene kan gå gjennom et trinn om gangen. Vi kan utvide instruksjonssyklusen til 5 trinn. Grunnet pipeline teknikken kan CPU utføre alle trinnene samtidig, men bare for en instruksjon på hvert trinn. Vi skiller mellom trinnene:

FI (fetch instruction), DI (Decode instruction), FO (Fetch Operand, bare dersom instruksen krever data utenfor registrene), EI (Execute instruction), WO (Write out).

Cache gjorde det enklere å ha instruksjoner klare. Dette kan sammenliknes med å ha matretter klare på en restaurant.

Senere kom **superskalare** prosessorer. Dette betød at de hadde to «samlebånd», to pipelines. De kunne altså ekspeder to «kunder» samtidig. Man skiller mellom forskjellige grader av superskalar, altså hvor mange parallelle samlebånd den har. Superskalar prosessor av grad to har to pipelines.

**Dynamisk utføring** betyr at prosessoren legger til side instruksjoner dersom de krever å hente data fra primærminnet. På denne måten kan CPU fortsette med instruksjoner fra cache inntil de nødvendige dataene har ankommet fra primærminnet.

I en pipeline utførelse kan det hende to instrukser trenger å aksessere minnet samtidig. Dette kalles en *hasard* og her vil pipelinen «hikke». Den siste instruksen vil stoppe opp litt og pipelinen venter noen klokkesykluser før den fortsetter.

**Flere kjerner**

Flere kjerner gjør at man får flere «prosessorer». Disse telles som en prosessor fordi de ligger på samme krets og fordi de deler på alle forbindelseslinjer, men de jobber alle uavhengig av hverandre. Elektronikken som sikrer at alle kjernene har koherens i cachene sine er mye enklere når de ligger på samme krets. Med flere kjerner blir det operativsystemets oppgave å fordele oppgaver og for eksempel takle hasarder.

## RISC og CISC

Forskjellen mellom disse er om de bruker hardkodede kontrollenheter eller mikrprogrammerbare kontrollenheter. Tidligere var det nærmest enn krig om hva man skulle bruke. I dag bruker vi det beste fra begge leire. Grunnet at man kan lage mer avanserte kretser har man klart å hardkode flere kompliserte operasjoner inn i kontrollenheter. Dermed er det nå også sterke innslag av RISC i CISC prosessorer. En typisk forskjell på de to er registerblokka. På CISC er denne mindre, og hvert register har en spesifikk oppgave. På RISC finnes det flere registre med generelle oppgaver.

Den største forskjellen mellom de to er instruksjonsformatet. CISK har lange og kompliserte instrukser. Her er dekodingen mye mer avansert, og mønsteret på instruksjonene forskjellig fra gang til gang. Instruksjonene til RISC er enkle og korte, og følger et mønster. Dette betyr at dekodingen er mye enklere.

En annen forskjell på de to er hvordan de håndterer funksjonskall. Dette er kall på funksjoner(metoder) der man hopper i koden, og deretter returnerer der man slapp. I tillegg har man ofte parametere som skal huskes, eller lokale variabler inni funksjonen. I CISC legges disse dataene på stakken – en del i minnet som brukes som midlertidig lagring/kladdeark for programmer og benytter LIFO prinsippet. Fordelen med dette er at vi kan lagre mye data, ulempen er at det blir mye minnetrafikk. PÅ RISC lagrer man alle dataene i en funksjon i registre. På denne måten slipper vi unødig minnetrafikk og får et raskere resultat. Unntaket, dersom man går tom for registre er å benytte minnet.

## Programmering

Vi skiller mellom symbolsk program og assembly program ved at assemblyprogram benytter variabler i stedet for spesifikke adresser. I stedet for å si LOAD 940, kan et assemblyprogram si LOAD A.

Man bruker sjelden assemblerprogrammering, men i to tilfeller er det nyttig. Ved utstyrsnær programmering, eller i tidskritiske deler av et program.

Java er veldig mye brukt fordi det oversetter programmet til bytekode, og deretter til kode som en spesifikk CPU kan skjønne når den første er lastet ned på en PC.