

MINIKOMPENDIUM

TDT4160 – DATAMASKINER GK

av Rune Vistnes

Versjon 1.0 – 24. november 2003

Innholdsfortegnelse

1 - Tallsystemer.....	6
1.1 - Bits og bytes.....	6
1.2 - Flyttallsrepresentasjon.....	6
1.3 - Binære tall.....	8
1.4 - Omregning fra ett tallsystem til et annet.....	8
1.4.1 - Omregning til titallsystem.....	8
1.4.2 - Omregning fra titallsystem.....	9
2 - Datamaskinens oppbygning.....	10
2.1 - Hva er en datamaskin?.....	10
2.2 - von Neumann-arkitektur.....	10
2.2.1 - Minnet.....	11
2.2.2 - Prosessoren (ALU og kontrollenhet).....	11
2.3 - Hovedkort.....	12
2.3.1 - Hovedkomponenter.....	12
3 - Prosessoren.....	14
3.1 - Utførende enhet.....	14
3.1.1 - Mikrooperasjoner.....	16
3.2 - Oversikt over prosessoren.....	18
3.3 - Styreenheten.....	19
4 - Instruksjoner.....	21
4.1 - Instruksjonssett.....	21
4.2 - Instruksjonstyper.....	21
4.3 - Representasjon.....	22
4.3.1 - Maskinspråk.....	22
4.3.2 - Assemblyspråk.....	22
4.3.3 - Høynivåspråk.....	23
4.4 - Antall operander.....	23
4.4.1 - Ingen operander.....	24
4.4.2 - En operand.....	24
4.4.3 - To operander.....	24
4.4.4 - Tre operander.....	24
4.5 - Big/Little/Bi endian.....	25
4.6 - Adresseringsmodi.....	25
4.6.1 - Immediate.....	25
4.6.2 - Direkte.....	26
4.6.3 - Indirekte.....	26
4.6.4 - Register.....	26
4.6.5 - Indirekte register.....	27
4.6.6 - Displacement.....	27
4.6.7 - Stakk.....	28
4.7 - Mer om stakk.....	28
4.8 - Instruksjonsformat.....	28
4.9 - Instruksjonssykel.....	30
4.9.1 - Hent.....	30
4.9.2 - Dekod.....	30
4.9.3 - Hent operander.....	30
4.9.4 - Utfør.....	31
4.9.5 - Lager.....	31

4.9.6 - Tilbakeskriving.....	31
4.10 - Instruksjonssykel i maskinvare.....	31
4.11 - En-sykelmaskin.....	31
4.12 - Fler-sykelmaskin.....	31
4.13 - Samlebåndsmaskin.....	31
4.14 - Samlebånd og kontrollflyt.....	33
4.14.1 - Dataavhengigheter.....	35
4.14.2 - RISC og CISC.....	36
5 - I/O (input/output).....	38
5.1 - Kommunikasjon mellom CPU og I/O-kontrollere.....	38
5.1.1 - Programmert I/O (polling).....	39
5.1.2 - Avbruddsdrevet I/O.....	39
5.1.3 - DMA (direct memory access).....	39
5.2 - Busser.....	40
5.2.1 - Bussarkitektur.....	40
5.2.2 - Multiplekset buss.....	41
5.2.3 - Arbitrering.....	41
5.2.4 - Timing (tidsstyring).....	41
5.2.5 - Eksterne busser.....	42
6 - Lagerhierarki.....	44
6.0.1 - Lagertypenes kapasitet.....	46
6.0.2 - Lagertypenes aksessmetoder.....	46
6.0.3 - Lagertypenes ytelse.....	46
7 - Hurtigbuffer.....	47
7.1 - Plasseringsmetoder.....	48
7.1.1 - Direkteavbildning.....	48
7.1.2 - Assosiativ avbildning.....	48
7.1.3 - Sett-assosiativ avbildning.....	48
7.2 - Organisering.....	48
7.3 - Erstatningsalgoritmer for hurtigbufferet.....	49
7.4 - Skrivestrategi.....	49
8 - Primærlageret.....	51
8.1 - RAM.....	51
8.1.1 - Statisk RAM.....	52
8.1.2 - Dynamisk RAM.....	52
8.1.3 - Synkron DRAM.....	52
8.2 - ROM.....	53
8.2.1 - ROM.....	53
8.2.2 - Programmerbar ROM (PROM).....	53
8.2.3 - Slettbar PROM (Erasable PROM - EPROM).....	53
8.2.4 - Elektronisk slettbar PROM (Electronically Erasable PROM - EEPROM).....	53
8.2.5 - Flash-minne.....	53
8.3 - Feilkorrigering i primærlageret.....	53
8.3.1 - Hammingkode.....	54
9 - Sekundærlageret.....	56
9.1 - Magnetplate.....	56
9.1.1 - RAID.....	57
9.1.2 - RAID-nivå.....	58
9.2 - Optisk plate.....	59
9.2.1 - CD-ROM.....	59
9.2.2 - DVD.....	60

10 - Innvevde (innebygde) systemer.....	61
10.1 - Designhensyn.....	61
10.1.1 - Kostnad.....	61
10.1.2 - Strømforbruk.....	61
10.1.3 - Sanntid.....	61
10.1.4 - Levetid.....	61
10.1.5 - Størrelse og vekt.....	62
10.1.6 - Vanskelig miljø.....	62
10.1.7 - Hastighet.....	62
10.2 - Komponenter - Mikrokontroller.....	62
10.3 - Utvikling.....	63
11 - Oppsummering.....	64
11.1 - Primærkunnskaper.....	64
11.2 - Prosessoren.....	65
11.3 - Instruksjoner.....	66
11.4 - I/O.....	67
11.5 - Lagerhierarkiet.....	68
11.6 - Hurtigbuffer.....	69
11.7 - Primærlageret.....	70
11.8 - Sekundærlageret.....	71
11.9 - Innvevde systemer.....	72

1 - Tallsystemer

1.1 – Bits og bytes

I en datamaskin snakker vi ofte om bits og bytes. En bit er den elementære lagringsenheten i en datamaskin. Denne kan være enten 0 (nei, usann, av) eller 1 (ja, sann, på). Bits (BInary digiT) representeres derfor med totallsystemet. En byte er en samling med 8 bits. Dette er ofte den minste enheten man kan håndtere uavhengig i en datamaskin. 8 bits kan settes sammen på 256 forskjellige måter og kan derfor representere alle tall fra 0 til 255. Heksadesimale tall er veldig praktiske til å representere grupper på 4 bits, og benyttes derfor ofte til å representere innholdet i en byte (som består av to grupper på fire bits).

Her kommer en liten oversikt over hva bits og bytes er for noe:

En bit (forkortes «b») er enten en 0-er eller en 1-er.

1 byte (forkortes «B») består av 8 bits.

1 KB (kilobytes) = 1024 bytes

1 MB (megabytes) = 1024 kB = 1024 · 1024 bytes = 1048576 bytes

1 GB (gigabytes) = 1024 MB

1 TB (terrabytes) = 1024 GB

Grunnen til at man multipliserer med 1024 i stedet for det mer naturlige 1000, er at man ønsker tall som passer med totallsystemet og ikke titallsystemet. $1024 = 2^{10}$.

1.2 – Flyttallsrepresentasjon

Tall kan representeres på flere måter. Det vanligste og enkleste i en datamaskin er heltall. En byte kan inneholde alle hele tall fra 0 til 255, eller dersom man ønsker tall med fortegn; fra -128 til 127. Tilsvarende kan man lage større heltall ved å benytte seg av dataord med 16, 32 eller 64 bits.

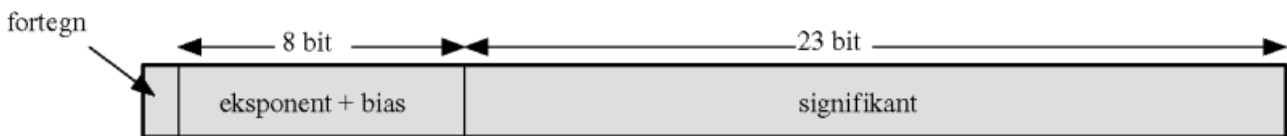
En annen representasjon er fastpunkt (fixed point). Her setter man av et fast antall bits til tall før og etter komma. For eksempel kan man dele en byte inn i 5 + 3, der man har 5 bits til å representere heltallet før komma, og 3 bits til å representere tallet etter komma. Dette er en enkel måte å lage kommatall på i en datamaskin som dessuten kan behandles raskt og nøyaktig av en datamaskin.

Dersom ingen av disse metodene strekker til, kan man benytte flyttall. Dette er den mest kompliserte måten å representere tall på for en datamaskin. Heltall og fastpunkt har et begrenset tallintervall, men like god presisjon over hele intervallet. Bruker man fastpunkt med god presisjon (mange bits for tallet etter komma), har man få bits igjen til tallet før komma, og intervallet blir lite. Tanken bak flyttall er å prøve å gi både i pose og sekk: Både stort intervall (kunne representere store tall), og god presisjon for små tall.

Flyttall benytter seg av samme metode som vi gjør med penn og papir: Vi innfører en eksponent. Skal vi skrive store tall, for eksempel 149 000 000 000 000, skriver vi ikke tallet ut fullstendig men benytter en tierekspont: $1,49 \cdot 10^{14}$. Dette tallet er svært stort, men har svært liten presisjon. Neste tall vi kan representere med samme antall siffer er $1,50 \cdot 10^{14}$, som er 10^{12} større! Det positive er at små tall har

god presisjon. Tallene $1,49 \cdot 10^{-14}$ og $1,50 \cdot 10^{-14}$ har bare en differanse på 10^{-16} . Dette kalles flyttall fordi posisjonen til komma endrer seg etter hvor stort tallet er. Store tall har få eller ingen siffer etter komma, mens små tall har mange siffer etter komma. Samme triks benyttes av flyttall i en datamaskin bortsett fra at man bruker totallsystemet.

Hvert tall kan skrives på formen: $\pm S \cdot B^{\pm E}$, hvor S står for signifikand (kalles av og til mantisse), B for basen og E for eksponent. Når man representerer flyttall i en datamaskin (binært) bruker man en base på $B = 2$.



Figuren viser hvordan et flyttall kan lagres i en datamaskin. Dette formatet følger IEEE's 32-bits flyttallsstandard. Vi har tre felt:

- fortegn
- signifikand
- eksponent + bias

Den våkne leser vil se at basen ikke lagres, den er nemlig 2 for alle tall i en datamaskin.

Alle flyttall i en datamaskin lagres normalisert. Dette vil si at de er på formen $1,xxx \cdot 2^{xxx}$. Dette gjør at vi ikke trenger å lagre sifferet før komma, fordi det alltid er 1. Signifikanden representerer altså sifrene etter komma. Eksponenten representeres ofte forskyvet (engelsk: biased). Det vil si at man lagrer tallet $E + \text{bias}$ i stedet for å lagre E direkte. Dersom eksponenten er på 8 bit vil man typisk ha en bias på 127. Dette gjøre at man kan ha negative eksponenter og likevel klare seg med å lagre bare positive tall (0 til 255) i eksponentfeltet. Bias er konstant og trenger derfor ikke lagres sammen med hvert tall.

Som et eksempel tar vi tallet 4,25.

- Først må vi forandre dette til binær form: 100,01
- Deretter må det normaliseres: $1,0001 \cdot 2^2$
- Fortegnsbitet blir 0
- Eksponenten er 2, og vi lagrer derfor $127 + 2 = 129(10) = 1000001(2)$
- Signifikanden blir 0001
- Vi ender da opp med dette: 0 1000001 0001000000000000000000

Her er noen andre eksempler på hvordan vi lagrer noen gitte binære flyttall:

1.1010001	· 2^{10100}	= 0 10010011 101000100000000000000000
-1.1010001	· 2^{10100}	= 1 10010011 101000100000000000000000
1.1010001	· 2^{-10100}	= 0 01101011 101000100000000000000000
-1.1010001	· 2^{-10100}	= 1 01101011 101000100000000000000000

Eksempler binærform av flyttall

Flyttall kan være veldig nyttig. Men man skal være svært påpasselig og restriktiv med å bruke disse i et dataprogram fordi avrundingsproblemer kan føre til uventede feil. Som oftest vil det være bedre å benytte heltall eller fastpunkt hvis det er mulig fordi du da har bedre kontroll på tallenes presisjon.

1.3 – Binære tall

Binære tall, det vil si tall som benytter base 2, finnes overalt rundt oss. De er blant annet brukt i den datamaskinen du har framfor deg nå. Binære tall brukes i logiske kretser og boolsk algebra. Hvor kom de fra og hvor var de utviklet? Binære tall har to opphavsmenn, den mest kjente er Leibniz så tidlig som 1666 og den mindre kjente er Pingala 1500 år før Leibniz.

Leibniz var en tysk matematiker som trengte en måte å representere logikk på ved hjelp av matematikk. Binære tall representerte boolske uttrykk for Leibniz, slik som ja/nei, sann/usann og av/på. Gjennom hans arbeid «kunsten å kombinere» introduserte han ideen om «tankenes lov», men denne ble ikke godt mottatt av folket i hans tid. Senere fikk han støtte av den kinesiske boken «I Ching» (Boken om endringer). Denne boken hjalp Leibniz til å omformulere arbeidet sitt. Leibniz mente at de binære tallene representerte skapelsen; tallet 1 var et bilde av Gud og tallet 0 var et bilde på det ugudelige.

På Pingalas tid brukte man binære tall til å beskrive musikk. «Chhandahshastra» av Pingala var kilden til denne oppdagelsen. Pingala dannet en matrise for å gi en unik verdi til hvert mål. Pingala brukte «Big Endian» skrivemåte på tallene sine, det vil si at han har det mest signifikante sifferet til høyre, i motsetning til slik vi bruker 10-tallsystemet med det mest signifikante sifferet til venstre (Little endian).

I senere tid har oktale tall (8-tallsystemet) og hexadesimale tall (16-tallsystemet) kommet til, noe som gjør hverdagen enklere for noen og vanskeligere for andre. Binære tall benyttes fordi logiske uttrykk kan være enten ja eller nei (1 eller 0). Derfor passer tall med base 2 fint. Oktale tall er nyttige fordi de kan brukes til å representere grupper på 3 binære siffer (3 binære siffer kan benyttes til å telle fra 0 til 7). Hexadesimale tall er tilsvarende nyttige fordi de kan brukes til å representere grupper på 4 binære siffer.

1.4 – Omregning fra ett tallsystem til et annet

1.4.1 – Omregning til titallsystem

Bruk av andre tallsystemer enn titallsystemet kan virke forvirrende hvis man ikke er vant til det. Heldigvis finnes det en enkel formel for å konvertere et hvilket som helst tall til titallsystemet:

$$\sum_{i=-n}^{m-1} d_i r^i$$

- r er radix (base)
- m er antall tall til venstre for komma
- n er antall tall til høyre for komma
- d_i er det i -te tallet

Som et forklarende eksempel på denne formelen brukes det hexadesimale tallet $4d20_{(16)}$. Dette tallet har ingen tall til høyre for komma, så vi begynner med $i = 0$.

- Ta minst signifikante tall (0) og multipliser med basen (16) opphøyd i 0
- Ta nest minst signifikante tall (2) og mult. med basen (16) opphøyd i 1
- Ta nest, nest minst signifikante tall ($d = 13$) og multipliser med basen (16) opphøyd i 2
- Ta mest signifikante tall (4) og multipliser med basen (16) opphøyd i 3
- Legg alle produktene sammen

Skrevet ut blir det slik:

$$(0 \cdot 16^0) + (2 \cdot 16^1) + (13 \cdot 16^2) + (4 \cdot 16^3) = 19744$$

Formelen fungerer for alle tallsystemer.

1.4.2 - Omregning fra titallsystem

Tilsvarende er det viktig å kunne konvertere fra titallsystemet til et annet tallsystem. Formelen over kan omformes til å gå fra desimaltall til en annen base. Algoritmen vil se slik ut:

- Ta tallet og del med basen (D / r , der D er opprinnelig tall og r er basen). Resten vil da være d_0 (minst signifikante tall i svaret)
- Ta heltallsdelen av svaret i punktet over og del med basen. Resten vil da være d_1
- Gjenta forrige punkt helt til heltallsdelen blir 0

Vi bruker samme eksempel: Regn ut hva $19744_{(10)}$ blir hexadesimalt.

- $19744 / 16 = 1234$, rest = 0
- $1234 / 16 = 77$, rest = 2
- $77 / 16 = 4$, rest = 13
- $4 / 16 = 0$, rest = 4

Vi får da svaret $4d20_{(16)}$.

2 - Datamaskinens oppbygning

2.1 - Hva er en datamaskin?

Det de fleste ser for seg når de tenker på en datamaskin er en PC med skjerm, tastatur og mus. Hvis man så fjerner skjermen, tastaturet og musen fra bildet, mener du fremdeles at det er en datamaskin? Vi kan ta det enda lengre, hva hvis vi fjerner harddisk, cd-rom diskettstasjon og lignende, vil det fremdeles være en datamaskin? Hva hvis du tok ut innmaten av kabinettet og brukte den uten kabinett?

De færreste tenker over hva som ligger i begrepet datamaskin, og begrepet har helt sikkert endret seg gjennom tiden. Hvis man skal forsøke å gi en tidløs definisjon på en datamaskin ville en prøvd seg på noe følgende: En datamaskin er en maskin (elektronisk, mekanisk, biologisk eller lignende) som tar inn en form for data, prosesserer denne og gir ut behandlede data. Ordbøker definerer uttrykket på følgende måte:

Websters

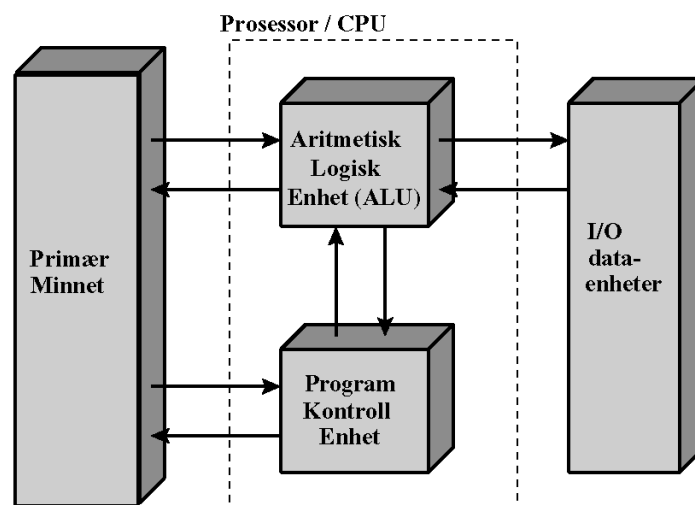
comput'er: also called processor: Electronic device designed to accept data, perform prescribed mathematical and logical operations at high speed, and display the result of these operations

Oxford

computer: An electronic machine that can store, organize and find information, do calculations and control other machines.

I løpet av det siste tiåret har det vært stor vekst innenfor en spesiell type datamaskiner, nemlig innebygde (embedded) systemer. Dette er gjerne små datamaskiner, typisk på en brikke, som gjerne er spesialdesignet til å utføre en spesiell oppgave. Typiske eksempler på innebygde systemer er mobiltelefoner, mp3-spillere og digitale planleggere.

2.2 - von Neumann-arkitektur



I dette faget er arkitekturen og virkemåten til en vanlig PC et sentralt tema. Denne virkemåten er veldig lik det som vi finner i andre systemer, men kan variere litt i kompleksitet. Over kan du se en oversikt over hvordan en von Neumann Maskin er bygd opp. Den består av fire deler:

- Primærminne, som inneholder data og instruksjoner (program)
- En aritmetisk og logisk enhet (engelsk: ALU) som utfører selve beregningene
- En kontrollenhet som tolker instruksjonene og sørger for at de blir utført
- Inn og ut data enheter, som f.eks. kan lagre data over lengre tid slik som eksterntminnet.

Von Neumann arkitekturen ligger til grunn for nesten alle datamaskiner som har blitt designet etter dette tidspunktet. Hovedpunktet i utgivelsen var at programmer ble lagret på samme måte som data og kunne endres av datamaskinen selv, mye i samsvar med det som originalt ble foreslått av Babbage. Konseptet som ligger til grunn for von Neumann-arkitekturen kaller vi "stored-program" konseptet. Det innebærer at programmer kan bli representert i en form som passer å bli lagret sammen med data i minnet. Datamaskinen kan da få sine instruksjoner ved å lese dem fra minnet.

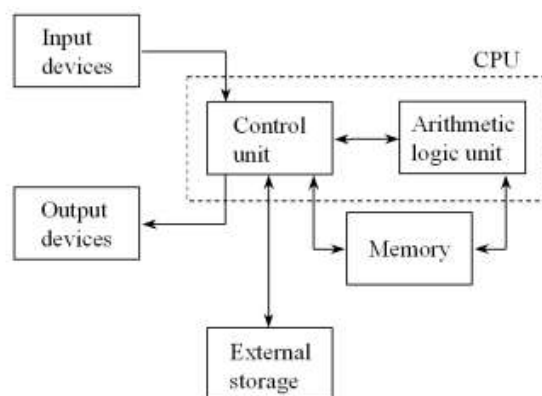
2.2.1 – Minnet

Minnet inneholder både data og instruksjoner. Både minnelokasjoner med data og instruksjoner kan leses fra og skrives til, og det i en hvilken som helst rekkefølge. Det er det du ser foregår hele tiden, om enn i en litt forenklet versjon enn i den virkelige verden.

2.2.2 – Prosessoren (ALU og kontrollenhet)

Prosessoren, eller Central Processing Unit (CPU), er den enheten som styrer hele systemet. Mens minnet lagrer data og busser frakter data frem og tilbake, tar CPU-en imot data og instruksjoner, utfører instruksjoner på dataene og sender resultatet ut igjen på bussene.

Prosessoren er delt i to enheter; en utførende enhet og en kontrollenhet. Den beregnende enheten kan utføre både aritmetiske og logiske operasjoner på data, og kalles derfor for den aritmetisk-logiske enheten. Kontrollenheten har ansvar for å tyde en instruksjon som den får fra minnet og velge alternative aksjonsveier basert på resultatet av tidligere utførte operasjoner.



2.3 – Hovedkort

Hvis du åpner opp kabinettet, det vil si den boksen som er selve datamaskinen, finner du hovedkortet. Under ser du en figur av et hovedkort hvor sentrale komponenter er navngitte.



2.3.1 – Hovedkomponenter

- Prosessorsokkelen er av typen socket A. Prosessoren er ikke koblet til på bildet over.
- Primærminnet kan kobles til DIMM-soklene (DIMM er en standard for tilkobling av RAM) som er merket "minnesokler".
- BIOS (Basic Input/Output System) er (for vanlige PC-er) navnet på et program som ligger i ROM (evt. flash). En av BIOS-ens viktigste oppgaver er å sørge for å hente operativsystemet fra sekundærlageret.



- Strøm: Her kobles strømmen til hovedkortet. Eksterne enheter og sekundærminnet har egen strømtilkobling.
- Kontakter: Her er et grensesnitt for eksterne komponenter slik som tastatur og printer
- ATA, Floppy: Her kan man koble til sekundærminnet (harddisk, cd-rom). Kontakten øverst til venstre kan du koble til diskettstasjonen. De resterende kontaktene i denne ruten er IDE-kontakter til sekundærminnet. Grunnen til at det er 4 tilkoblinger i motsetning til 2 som er normalt er for

at dette spesielle hovedkortet inneholder en RAID-kontroller. Nyere hovedkort har en mulighet for tilkobling av serial-ATA som er en ny seriell utgave av ATA.

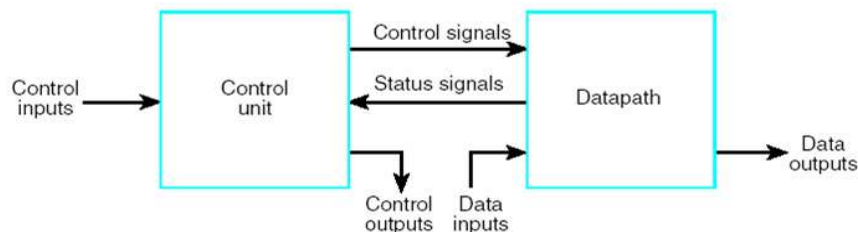
- Busser: Her kan ekspansjonskort som nettverkskort, grafikkort og lydkort tilkobles. Det brune tilkoblingspunktet helt til venstre er en AGP slot som er laget for grafikkort. De hvite tilkoblingspunktene til høyre er PCI-slottet som er standarden for de fleste ekspansjonskortene i stasjonære PC-er

3 - Prosessoren

En prosessor er en kompleks enhet i en datamaskin som er i stand til å utføre en algoritme spesifisert ved et lagret program. En ting du må huske på når du leser om prosessoren, er at det finnes mange måter å lage denne på. Det som presenteres her er én mulig løsning, men andre kilder vil presentere en annen løsning. Det du bør sitte igjen med er en generell forståelse, ikke detaljer om hvordan en spesifikk prosessor fungerer.

Ofte deles prosessorer inn på denne måten:

- Styreenhet (engelsk: control unit) som styrer den utførende enheten og bestemmer hva som skal utføres
- Utførende enhet (engelsk: datapath) der data flyter gjennom og blir behandlet

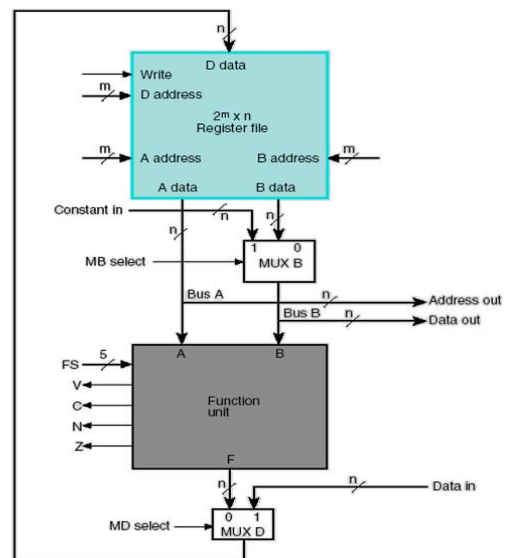


Figuren over viser forholdet mellom utførende enhet og styreenhet. Styreenheten finner ut hva den skal gjøre ved å se på alle sine inngangssignaler. Basert på dette setter styreenheten opp kontrollsignalene (kalles også styresignalene) som er nødvendig for å få den utførende enheten til å gjøre det den skal. Utførende enhet tar inn data, prosesserer disse (for eksempel gjør en logisk operasjon) og sender svaret ut på en utgang. Den forteller også styreenheten enkelte ting den har behov for å vite om resultatet.

3.1 - Utførende enhet

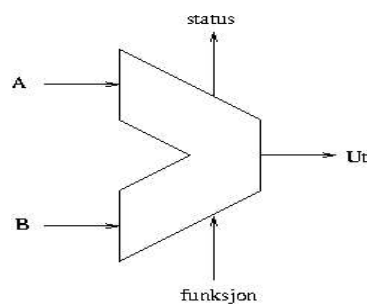
Utførende enhet (engelsk: datapath) er ansvarlig for å prosessere data. En utførende enhet består hovedsaklig av en registerblokk og en funksjonell enhet. Under er en figur som viser et eksempel på en utførende enhet. Dens oppbygning varierer mellom prosessortyper, men vil i hovedsak ligne på denne.

En prosessor er en synkron enhet. Alt som skjer i prosessoren skjer synkront, det vil si i takt med et klokkesignal. Klokkesignalet veksler mellom verdiene 0 og 1, og en slik veksling kalles en klokkesyklus. Alt som skjer innenfor samme klokkesyklus, skjer i parallell.



Registerblokk består av et sett med registre. Et register er et lite og hurtig lager som kan lagre ett ord (ofte 32 bits). Prosessoren har mulighet til å lese ut to registre hver klokkesykel: A og B. Ved å sette opp registeradresser (registernumrene) på A- og B-adresseinngangene vil de tilsvarende registrene komme ut på datautgangene. Tilsvarende har prosessoren mulighet til å skrive tilbake data til ett register hver klokkesyklus. Dette gjøres ved å sette adressen til registeret det skal skrives til på D-adresse-inngangen, og skrive data til D-datainngangen. I tillegg finnes et «write»-signal som indikerer at man ønsker å skrive data til registeret denne klokkesyklusen. I en del prosessorer er ikke register 0 et ekte register. I stedet er det satt konstant lik 0. Dette fordi det ofte er ønskelig å ha en lett måte å få tilgang til denne konstanten på.

Den funksjonelle enheten tar i mot to dataenheter (A og B) og gjør en gitt operasjon på disse. Resultatet sendes ut til en utgang (F). Hvilken operasjon som skal utføres bestemmes av et kontrollsignal merket FS i figuren. Den funksjonelle enheten kan gjøre aritmetiske og logiske operasjoner, for eksempel addisjon, subtraksjon, AND og OR. Derfor er det vanlig å kalle denne for en ALU (Arithmetic Logic Unit). Det er dette navnet som vil bli brukt senere i kompendiet. En ALU tegnes ofte med symbolet vist i figuren under. Dette for å fremheve at en ALU har to hovedinngangssignaler som den behandler, for så å sende resultatet til utgangen.



Resultatet av hver ALU-operasjon medfører at visse statussignaler (kalles ofte for flagg) blir satt. Disse statusflaggene benyttes av styreenheten for å avgjøre hva som skal skje neste sykel.

Vanlige statusflagg er:

- **Mente (C for Carry):** Forteller at resultatet ble større enn det tilgjengelige antall bit. For eksempel med 8-bits tall vil $250 + 10$ «gå rundt», og resultatet blir 4 og C-flagget satt.
- **Overflyt (V):** Nesten likt C-flagget, men for tall med fortegn. For eksempel med 8-bits tall vil $120 + 10$ gå rundt, og resultatet vil være (i toerskomplement) -126 og V-flagget satt.
- **Negativ (N):** Forteller at svaret er negativt
- **Null (Z for Zero):** Forteller av svaret er 0

Utførende enhet kobles opp slik at A- og B-utgangene fra registerblokken kobles inn på A- og B-inngangene til ALU-en. Utgangen på ALU-en kobles inn på D-inngangen til registerblokken. Når man nå ønsker å gjøre en operasjon tar man ut to registre fra registerblokken, lar ALU gjøre en operasjon på disse, for eksempel legge de sammen, og så skrive svaret tilbake til et register i registerblokken.

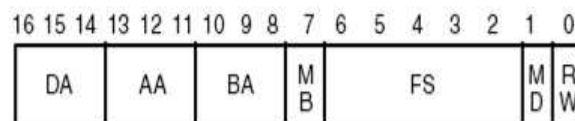
Ofte er det ønskelig å hente data andre steder fra enn fra registerblokken. MUX B gjør det mulig å velge noe annet enn register B som inngang til ALU, og MUX D gjør det mulig å skrive noe annet enn svaret fra ALU til registerblokken.

Konstanten som kommer inn til MUX B vil komme fra instruksjonsregisteret, som du vil lære mer om senere. Utenfor ligger det et datalager (RAM), som kobles til linjene «Address out», «Data out» og «Data in». Dette kan man skrive til i stedet for å la data gå gjennom ALU, eller man kan lese data fra datalageret og skrive til registerblokken i stedet for det som kommer ut fra ALU.

For å sette opp den utførende enheten til å gjøre en bestemt operasjon, må alle styresignaler settes opp riktig. Her er en oversikt over styresignaler:

- DA: Registeradresse til registerblokkens skriveport
- AA: Registeradresse til registerblokkens leseport A
- BA: Registeradresse til registerblokkens leseport B
- MB: Kontrollsignal til MUX B (velger inngang)
- FS: Velger funksjon som skal utføres av ALU
- MD: Kontrollsignal til MUX D
- WR: Angir om data skal skrives til registerblokk

Styresignalene grupperes for enkelhets skyld sammen til et styreord, som er det styreenheten gir inn til den utførende enheten. I AOC vil formatet på styreordet være som vist i figuren under.



3.1.1 – Mikrooperasjoner

Hva er en mikrooperasjon?

Før vi går inn på prosessorens oppbygning bør du vite litt om hva en prosessor er i stand til å gjøre for noe. Som et hjelpemiddel til å analysere prosessorens funksjon har vi innført begrepet mikrooperasjon.

En mikrooperasjon er en elementær operasjon som blir utført på dataceller. De er som oftest på følgende form: Svarcelle ← celle1 OP celle2. Man tar to dataelementer (celle1 og celle2), gjør en operasjon på disse, og lagrer svaret (i svarcelle). Datacellene er stort sett alltid registre, som du vil lære mer om senere, men kan også være fra datalageret (RAM). Register skrives på denne måten: R_n, hvor n angir hvilket register som benyttes.

Poenget er at vi kan beskrive oppførselen til prosessoren ved hjelp av slike mikrooperasjoner.

De fire vanligste mikrooperasjonene er:

- Overføringsmikrooperasjoner overfører data.
- Aritmetiske mikrooperasjoner som utfører aritmetikk.
- Logiske mikrooperasjoner som gjør logiske operasjoner.
- Skift-mikrooperasjoner som forskyver data i registrene.

Aritmetiske mikrooperasjoner

Notasjon	Beskrivelse
$R0 \leftarrow R1 + R2$	Innholdet til R1 blir plussset med innholdet til R2 og overført til R0
$R2 \leftarrow \sim R2$	Komplementet av innholdet til R2(enerkomplement)
$R2 \leftarrow \sim R2 + 1$	2-ers komplement av innholdet til R2
$R0 \leftarrow R1 + R2 + 1$	R1 pluss 2-ers komplementet til R2 overført til R0 (subtraksjon)
$R1 \leftarrow R1 + 1$	Inkrementer innholdet til R1 (teller opp)
$R1 \leftarrow R1 - 1$	Dekrementerer innholdet til R1 (teller ned)

Logiske mikrooperasjoner

Notasjon	Beskrivelse
$R0 \leftarrow \sim R1$	Logisk bitvis NOT
$R0 \leftarrow R1 \text{ AND } R2$	Logisk bitvis AND
$R0 \leftarrow R1 \text{ OR } R2$	Logisk bitvis OR
$R0 \leftarrow R1 \text{ XOR } R2$	Logisk bitvis XOR

Skift-mikrooperasjoner

Skifter data i et register enten mot høyre eller mot venstre. Venstreskift medfører at alle bit i et binært tall flyttes en plass mot venstre, og minst signifikante bit settes til 0. Dette er ekvivalent med å multiplisere med 2. Høyreskift skifter data mot høyre, som er det samme som å dele med 2.

Vi skiller gjerne mellom logisk skift og aritmetisk skift. Logisk venstreskift er akkurat som forklart over. Logisk høyreskift er tilsvarende, men man skifter alltid inn 0 på mest signifikante bit. Aritmetisk skift brukes når man ønsker å ta vare på fortegnet. Aritmetisk høyreskift er som forklart over, men i stedet for å skifte inn 0 blir fortegnsbitet (mest signifikante bit) tatt vare på. Tilsvarende for aritmetisk venstreskift; fortegnsbitet blir ikke skiftet vekk.

Notasjon	Beskrivelse
$R0 \leftarrow R1 \ll R2$	Aritmetisk venstreskift
$R0 \leftarrow R1 \gg R2$	Aritmetisk høyreskift
$R0 \leftarrow R1 \lll R2$	Logisk venstreskift
$R0 \leftarrow R1 \ggg R2$	Logisk høyreskift

3.2 – Oversikt over prosessoren

Nå vet du hva den utførende enheten er og hvordan denne kontrolleres av styresignalene i styreordet. Du har sett at ved å sette opp styreordet riktig kan du få utførende enhet til å utføre mikrooperasjoner. Neste skritt nå er å legge til de delene av prosessoren som leser inn et program og setter opp utførende enhet til å utføre dette programmet.

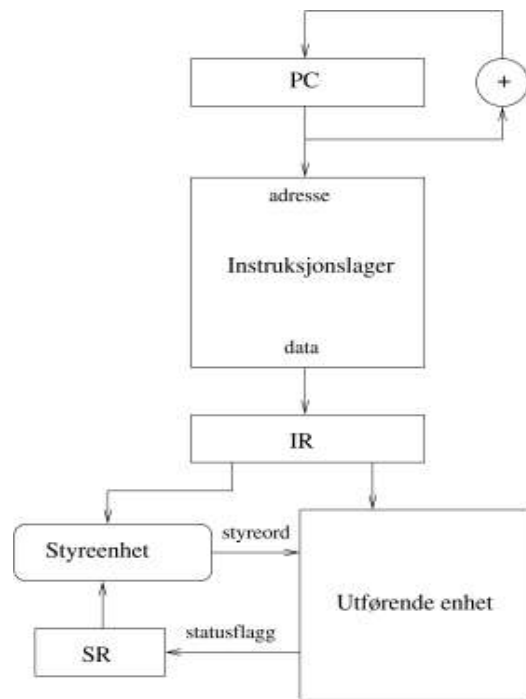
Som du sikkert husker om von Neumann-arkitekturen ligger dataprogrammet lagret i et instruksjonslager (i RAM). Disse er lagret i form av en rekke instruksjoner som ligger sekvensielt i lageret. Hver instruksjon er en liten oppgave som prosessoren skal gjøre. Ofte tilsvarende dette en mikrooperasjon, men ikke alltid. Prosessoren henter instruksjoner fra instruksjonslageret og finner ut hvordan disse skal oversettes til et styreord for utførende enhet, som da gjør selve jobben.

Du husker sikkert at det finnes en registerfil som inneholder generelle dataregistre brukt til utregninger. Dette er såkalte «brukersynlige» registre fordi disse brukes direkte av brukeren (det vil si programmet). I tillegg finnes det noen spesialregistre i prosessoren; registre som har helt spesielle oppgaver.

Et av de viktigste er programtelleren (PC for Program Counter). Programtelleren inneholder til enhver tid adressen til neste instruksjon som skal hentes fra instruksjonslageret. Hver gang en instruksjon er ferdig utført, hentes en ny instruksjon fra instruksjonslageret (med PC som adresse) og PC inkrementeres. Unntaket er for hoppinstruksjoner, da disse medfører at prosessoren skal hente instruksjoner fra en helt annen plass. PC vil da få en helt ny verdi.

Programtelleren benyttes til å slå opp i instruksjonslageret for å finne neste instruksjon. Denne instruksjonen lagres så i et instruksjonsregister (IR). IR inneholder altså til enhver tid den gjeldende instruksjonen.

Nå kan vi legge til de delene av prosessoren som ligger utenfor utførende enhet. Dette er forenklet vist i figuren under.



Som du ser går PC rett inn på adresseinngangen til instruksjonslageret. Dette medfører at vi i neste sykel får overført instruksjonen som ligger på denne adressen til IR. Når vi har en instruksjon i IR, kan selve jobben begynne. Styreenheten undersøker instruksjonen og finner ut hva denne skal gjøre for noe. Basert på dette setter den opp styreordet til utførende enhet. Noen av kontrollsignalene (og også datasignalene) inn til utførende enhet kan komme direkte fra IR. Dette vil du lære mer om når instruksjoner og instruksjonssett forklares mer i detalj.

I tillegg til spesialregistrene nevnt ovenfor, finnes det et register som kalles statusregister (SR). Dette inneholder rett og slett statusflaggene fra ALU-en. I tillegg kan det inneholde noen andre flagg, for eksempel et som sier om prosessoren er i system-modus (for operativsystemet) eller i bruker-modus (for alle brukerprogrammer). SR benyttes av styreenheten fordi utføring av visse instruksjoner er avhengig av innholdet i SR. Dette gjelder for eksempel betingete hoppinstruksjoner, der man bare skal utføre et hopp hvis forrige instruksjon ga et gitt resultat.

3.3 – Styreenheten

Styreenheten (engelsk: control unit) får inn data fra instruksjonsregisteret og benytter dette, sammen med statusregisteret, til å avgjøre hvordan styreordet til den utførende enheten skal settes opp. Den sørger også for at resten av prosessoren, det vil si henting av instruksjoner, skjer på riktig måte og til riktig tid.

Noen instruksjoner svarer direkte til en mikrooperasjon. Er det tilfelle kan styreenheten sette opp styreordet og gjøre seg ferdig med instruksjonen samme syklen instruksjonen kom til instruksjonsregisteret (IR). Andre instruksjoner kan være mer komplekse, og bestå av flere mikrooperasjoner. Er det tilfelle må styreenheten bruke flere sykler på å gjøre seg ferdig med instruksjonen, og sette opp forskjellige styreord for hver sykel.

Styreenheten er enten fastprogrammert eller mikroprogrammert.

I en fastprogrammert (hardwired) styreenhet er oppførselen bestemt av måten MUX-er, vipper og kombinatorisk logikk er koblet opp på. Dersom alle instruksjoner bare trenger én sykel (de utfører bare én mikrooperasjon i datastien), kan en fastprogrammert styreenhet bestå av ren kombinatorikk. Trengs flere sykler, kan styreenheten kobles opp ved hjelp av en tilstandsmaskin (FSM).

En mikroprogrammert styreenhet benyttes ofte dersom det finnes mange komplekse fler-sykel-instruksjoner. I stedet for å lage en stor og kompleks tilstandsmaskin kan man lage en slags «prosessor-inni-prosessoren». Styreenheten inneholder et lite lager med mikroinstruksjoner, kalt styrelageret. Dette er egentlig en rekke med ferdige styreord. Når styreenheten begynner på en ny instruksjon, inneholder denne en adresse inn i styrelageret der den for hver sykel henter et styreord som den mater inn i den utførende enheten, helt til alle styreordene for denne instruksjonen er ferdig.

Fordelene med mikroprogrammert styreenhet er at man lett kan lage støtte for komplekse, fler-sykel-instruksjoner. Når man først har satt opp rammeverket for styreenheten vil implementering av nye instruksjoner rett og slett bestå i å endre på styrelageret. Tilsvarende vil det være enkelt å rette opp feil, da det bare vil bestå i endring av styrelageret. Etterhvert som kompleksiteten på instruksjonene økte ble dette en populær måte å implementere styreenheten på.

Fordelene med fastprogrammert styreenhet er at den for enkle instruksjoner kan bli både mindre, raskere og enklere å lage. I senere tid har dette blitt mer brukt igjen. Som dere vil lære senere, går man over til enklere instruksjoner – hvis man likevel deler instruksjonene inn i mindre mikroinstruksjoner, hvorfor ikke bare la instruksjonssettet bestå av slike enkle instruksjoner så slipper man dette ekstra laget. Dessuten har nye metoder innen prosessordesign minket problemene ved å lage og endre fastprogrammerte styreenheter.

4 - Instruksjoner

4.1 - Instruksjonssett

```
add $1, $2, $3  
jmp loop
```

En datamaskins oppførsel bestemmes av instruksjonene den utfører. En instruksjon er en konkret og ofte elementær oppgave som prosessoren kan utføre. Mange slike elementære instruksjoner settes sammen til et større program.

Samlingen med instruksjoner en gitt prosessor kan utføre kalles prosessorens instruksjonssett. Hvilke instruksjoner som bør være med i en prosessors instruksjonssett har vært gjenstand for mye forskning, og problemet har fremdeles ingen entydig løsning.

Viktige egenskaper ved forskjellige instruksjonssett:

- Operasjoner: Hvilke og hvor komplekse operasjoner det finnes instruksjoner til.
- Datatyper: Hvilke datatyper prosessoren kan håndtere direkte
- Hvilket instruksjonsformat instruksjonene bruker, det vil si hvordan instruksjoner kodes i et instruksjonsord
- Hvor mange registre prosessoren støtter
- Hvordan prosessoren angir operandene; hvordan operanddata adresseres

4.2 - Instruksjonstyper

Alle instruksjoner kan grovt plasseres i én av disse gruppene:

- Dataoverføring: Instruksjoner som overfører data fra én spesifisert plass til en annen. Dette kan for eksempel være mellom to registre eller mellom to RAM-celler.
- Aritmetikk: Instruksjoner som gjør enkle aritmetiske operasjoner, for eksempel addisjon, subtraksjon, multiplikasjon, divisjon.
- Logisk: Instruksjoner som gjør logiske operasjoner. Disse arbeider ofte på et helt dataord av gangen og gjør den logiske operasjonen på alle bitene i dataordet. Eksempler: AND, OR, NOT, XOR, diverse skiftoperasjoner
- Konvertering: Konvertering mellom forskjellige datatyper, for eksempel fra desimaltall (BCD) til binær.
- I/O: Instruksjoner for kommunikasjon med I/O-enheter.
- Systemkontroll: Instruksjoner som krever spesielle systemprivilegier for å bli utført. Disse blir som oftest bare utført av operativsystemet; vanlige brukerprogrammer vil ikke ha mulighet til å utføre disse. Eksempler kan være endring av konfigurering av virtuelt lager, endring av kontrollregistre etc.
- Kontrollflyt: Instruksjoner som endrer flyten av instruksjoner gjennom prosessoren. Vanligvis utføres instruksjoner én etter én sekvensielt slik de er lagret i RAM. Kontrollflytinstruksjoner medfører hopp i programmet.

Dette er helt nødvendig for å kunne lage programløkker og for å lage betinget kode, det vil si kode som kun utføres dersom en betingelse er oppfylt.

De vanligste kontrollflytinstruksjoner er hopp (branch) og prosedyrekall:

- Hoppinstruksjoner kan være ubetinget eller betinget. Betingete hoppinstruksjoner utfører hoppet hvis og bare hvis en gitt betingelse er oppfylt. Dette brukes til å implementere blant annet if-setninger i høynivåspråk.
- Prosedyrekall krever egne hoppinstruksjoner. En prosedyre er et delprogram som kan kalles hvor som helst i hovedprogrammet. Etter at prosedyren har kjørt seg ferdig hopper prosessoren tilbake til stedet i hovedprogrammet der prosedyren ble kalt. Prosessoren må altså huske hvor kallet til prosedyren ble utført (returadressen) for å kunne hoppe tilbake dit etter at prosedyren har kjørt seg ferdig. Returadressen kan enten lagres på en stakk eller i et spesielt returregister.

4.3 – Representasjon

4.3.1 – Maskinspråk

Som alt annet i en datamaskin lagres instruksjoner som lange remser med tall. Dette kan for eksempel se slik ut:

```
EE000000  
EE050005  
A3000503  
B7000000  
A4111110
```

Det er nesten umulig for mennesker å programmere et slikt rent maskinspråk direkte. Derfor kom det tidlig oversettere fra tekstlige programbeskrivelser til maskinspråk.

4.3.2 – Assemblyspråk

For å gjøre det lettere for mennesker å håndtere maskinspråk ble assembleren oppfunnet. Hver instruksjon fikk et huskesymbol (engelsk: mnemonic); et lite navn. Når man skal programmere kan man skrive slike navn i stedet for å skrive inn instruksjonsnumrene direkte. Assembleren tar seg av å oversette fra den tekstlige beskrivelsen til tall. Dette er en direkte oversetting; hver linje tilsvarer én gitt instruksjon, og hver prosessortype har derfor sitt eget assemblyspråk.

Etter hvert har assemblerne fått flere utvidelser. Man kan skrive kommentarer etter hver instruksjon for å forklare hva den gjør. Man kan bruke merkelapper (labels) i stedet for å angi absolutte adresser (for eksempel ved hopp), noe som gjør at man slipper å holde styr på adressene selv. Man har også fått makromuligheter (gruppere flere instruksjoner sammen til en makroinstruksjon) og betinget assemblering (assembleren bruker bare deler av koden, avhengig av parametere).

Et eksempel på assemblerkode:

```
        load $0, 0           ; legg verdien 0 i register 0
        load $5, 5           ; legg verdien 5 i register 5
loop    ; merkelapp
        jge $0, $5, slutt    ; hopp til slutt hvis register 0 er
                               ; større eller lik register 5
        inc $0               ; inkremitter register 0
        jmp loop             ; hopp tilbake til merkelapp
slutt
```

Denne koden teller opp register 0 fra 0 til 5. En assembler vil oversette dette til maskinkode bestående av tall. Både assemblerkode og tilhørende maskinkode vil variere fra prosessor til prosessor.

Assemblerkoden ser kanskje kryptisk ut for et utrenet øye, men det burde ikke være tvil om at den er enklere å håndtere enn maskinkoden.

4.3.3 – Høynivåspråk

Assembliespråk er som sagt en direkte overgang fra en tekstlig beskrivelse til maskininstruksjoner. Også dette ble etter hvert for detaljert, og man begynte å utvikle høynivåspråk. Dette er programmeringsspråk der man ikke har en direkte sammenheng mellom høynivåkoden og den underliggende maskinkoden.

Høynivåkoden gir som oftest muligheter til å skrive inn komplekse uttrykk på en enkel måte uten å ta hensyn til hvilke muligheter prosessoren i bunnen gir. En kompilator oversetter høynivåkoden til maskinkode (ofte med assembliespråk som mellomledd).

Et C-program som gjør det samme som assemblerkoden over kan se slik ut:

```
i = 0;
while (i < 5) {
    i++;
}
```

4.4 – Antall operander

Instruksjoner tar ofte operander (argumenter). Som et eksempel, se på følgende operasjon:

```
D <- A + B
```

Her skal A legges sammen med B. Svaret skal legges i D. En instruksjon som implementerer denne operasjonen må altså vite hvilke tre dataverdier den skal jobbe på (A, B og D).

Instruksjoner kan klassifiseres etter hvor mange eksplisitte operander de tar.

4.4.1 - Ingen operander

Instruksjonen har ingen eksplisitte operander. Instruksjonen som gjør operasjonen vist over vil derfor se slik ut:

```
ADD
```

En prosessor med overvekt av slike 0-operand-instruksjoner kalles «stakkmaskin». Prosessoren trenger ikke mer informasjon om denne instruksjonen fordi den vet at den skal hente A og B fra en stakk, og svaret skal legges på toppen av stakken. For å utføre $D \leftarrow A + B$, kan man utføre følgende stakkmaskinprogram:

```
PUSH A    ; legg A på toppen av stakken
PUSH B    ; legg B på toppen av stakken
ADD       ; fjern de to øverste tallene på stakken, legg de
          ; sammen og legg svaret på toppen av stakken
```

Svaret vil nå ligge på toppen av stakken.

4.4.2 - En operand

Instruksjonen kan ha maksimalt én eksplisitt operand, og har dermed to implisitte.

```
ADD B
```

Prosesorer med overvekt av slike instruksjoner kalles ofte for en «akkumulatormaskin» og er en prosessorarkitektur som var mye brukt på 60 og 70-tallet. Prosessoren har et spesielt register som kalles en akkumulator som automatisk tar opp rollen til to av operandene. Instruksjonen over vil gjøre følgende: Akkumulator \leftarrow Akkumulator + B. For å utføre operasjonen $D \leftarrow A + B$, må man gjøre følgende:

```
LDA A     ; legg A i akkumulatoren
ADD B     ; Legg sammen akkumulator og B
```

Svaret vil nå ligge i akkumulatoren

4.4.3 - To operander

Instruksjonen har maksimalt to eksplisitte operander.

```
ADD A, B
```

Her vil destinasjonen være implisitt. Instruksjonen vil utføre følgende operasjon: $A \leftarrow A + B$. Svaret vil altså ligge der A-operanden lå.

4.4.4 - Tre operander

Alle operander er eksplisitte.

```
ADD D, A, B
```

Prosessoren trenger ikke ta noen implisitte avgjørelser, all informasjon er gitt av instruksjonen.

4.5 – Big/Little/Bi endian

De fleste datamaskiner operer med byte som minste adresserbare enhet. Prosessorer klarer likevel å behandle større dataenheter, for eksempel 16 bit, og må ha en standard for hvordan disse 16 bitene skal splittes opp og legges inn i et byteadressert lager.

Eksempel: Et 16-bits heksadesimalt tall 0xABCD skal legges i et byteadressert lager på adresse 10. Dette kan enten gjøres slik:

Adresse 10: 0xAB
Adresse 11: 0xCD

eller slik:

Adresse 10: 0xCD
Adresse 11: 0xAB

Den første metoden, der mest signifikante byte legges på laveste adresse, kalles big endian (brukt blant annet av prosessorene Motorola 680x0 og SPARC). Den andre metoden, der mest signifikante byte legges på høyeste adresse, kalles little endian (brukt blant annet av Intel x86 og Pentium).

Begge deler er like vanlige, noe som er et klassisk problem i systemer som benytter flere forskjellige typer prosessorer (for eksempel datautveksling mellom Intel-prosessorer og Motorola-prosessorer). På grunn av dette finnes det bi endianprosessorer, det vil si prosessorer som kan bruke begge metodene (for eksempel PowerPC).

4.6 – Adresseringsmodi

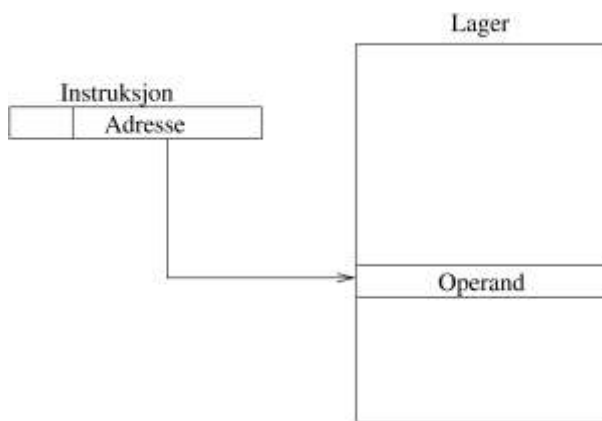
En instruksjon er ofte avhengig av operander for å kunne utføres. Et eksempel er en instruksjon som gjør dette: $D <= A + B$. En slik instruksjon må inneholde en referanse til hvor A og B skal hentes fra, og hvor resultatet D skal lagres. Måten instruksjonen angir hvor data skal hentes fra kalles en adresseringsmodus. Følgende adresseringsmoduser er vanlige:

4.6.1 – Immediate



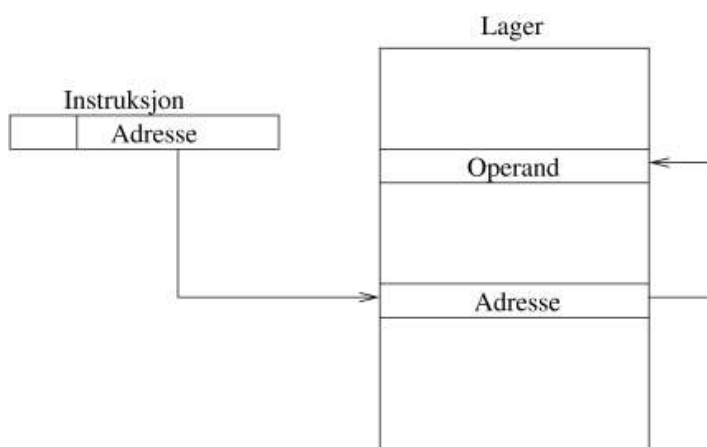
Operanden er innbakt i instruksjonen. Dersom operanden er kjent (en konstant) når programmet lages, kan verdien av denne legges inn i selve instruksjonen.

4.6.2 - Direkte



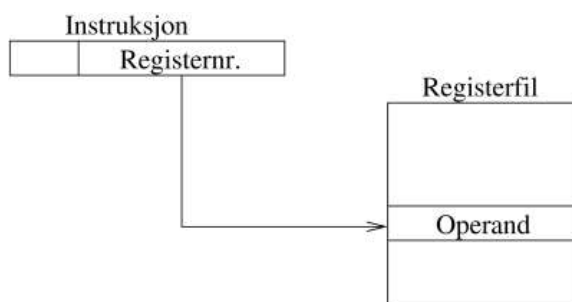
Instruksjonen angir adressen til operand i RAM.

4.6.3 - Indirekte



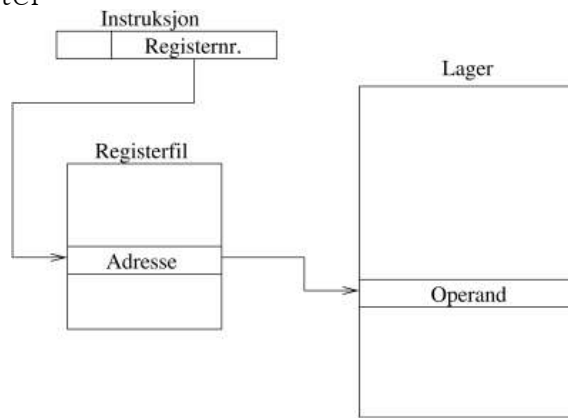
Instruksjonen angir adresse til RAM-celle som igjen inneholder adressen til operand

4.6.4 - Register



Instruksjon har nummer på register som inneholder operand.

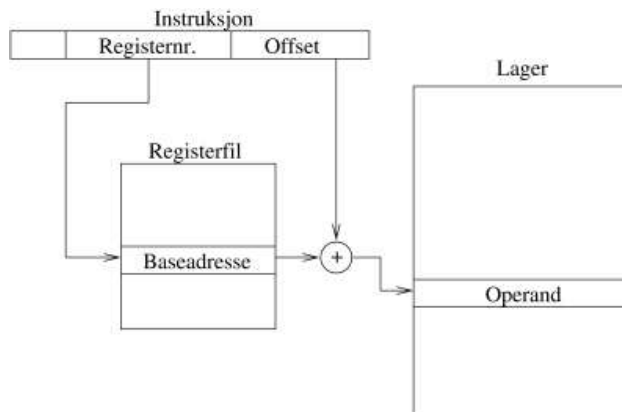
4.6.5 – Indirekte register



Instruksjon har nummer på register som inneholder adresse til operand i RAM

4.6.6 – Displacement

Instruksjon har nummer på register som inneholder baseadresse som adderes med en konstant angitt i instruksjon. Vi har altså en konstant (offset i figuren) som blir lagt til innholdet i et register.



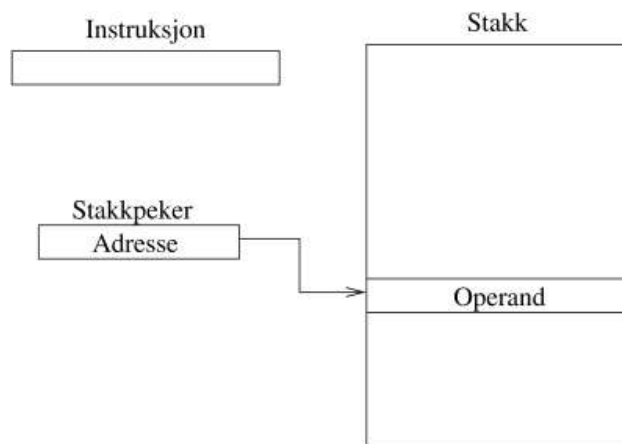
Dette er veldig nyttig, for eksempel til tabelloppslag der man har baseadressen til tabellen i et register og man skal ha det n-te elementet i tabellen.

Dersom konstanten benyttes som baseadresse og registeret inneholder «offset»-en, kalles denne metoden for indeksring.

Dette er egentlig det generelle tilfellet av relativ adressering som brukes til hoppinstruksjoner der man hopper et gitt antall bytes relativt til programtelleren. Da vil programtelleren være implisitt gitt, og «offset» gis direkte i instruksjonen.

4.6.7 – Stakk

Adressen er implisitt gitt av stakkpeker.



4.7 – Mer om stakk

En stakk er en sentral datastruktur for prosessorer; de fleste prosessorer benytter stakk på en eller annen måte.

En stakk kan sammenlignes med en bunke papirer der du bare har lov til å legge til eller hente ut papirer fra toppen av bunken. Dette blir altså en Last-In-First-Out (LIFO) kø: Det siste elementet som ble lagt til stakken er det elementet som hentes ut.

Stakken består av et lager og en stakkpeker (SP). Siden stakken endrer størrelse brukes stakkpekeren til å vise hvor i lageret toppen av stakken befinner seg for øyeblikket. Stakkpeker peker på neste ledige element i lageret.

De fundamentale stakkoperasjonene er PUSH og POP:

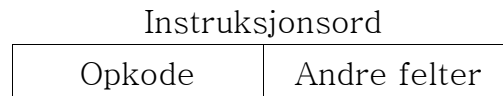
- PUSH legger til et dataelement til toppen av stakken og øker stakkpeker med én.
- POP henter ut data fra toppen av stakken og minker stakkpeker med én. '

4.8 – Instruksjonsformat

Instruksjoner består av en mengde bit, satt sammen til et instruksjonsord. Instruksjonsordet kan ha en fast lengde, eller det kan variere fra instruksjon til instruksjon. Lengden på instruksjonsordet har betydning for hvor mye informasjon man kan legge inn i instruksjonen. Ofte vil man likevel ha det så kort som mulig for å bruke minst mulig plass. En løsning kan være variabel lengde på instruksjonene, men det medfører en mer kompleks prosessor.

For at prosessoren skal kunne kjenne igjen og forstå instruksjoner må disse angis på et fast bestemt format. En instruksjon må inneholde informasjon om hva som skal utføres og på hvilke data den skal utføres på. Instruksjonsordet deles inn i felter, som hver er på en eller flere bit. Måten instruksjonsordet deles inn i felter på kalles instruksjonens instruksjonsformat.

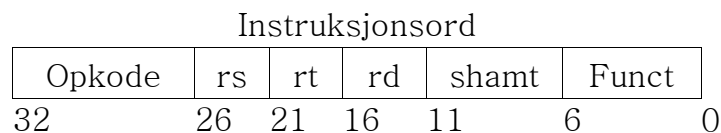
Hver instruksjon må inneholde et felt kalt opkode som forteller prosessoren hvilken type instruksjon det er snakk om. I tillegg må instruksjonen inneholde felter som beskriver tilleggsinformasjon som trengs under utføring av instruksjonen (blant annet operandene).



Opkoden forteller entydig hva denne instruksjonen skal gjøre og hvordan resten av instruksjonsordet skal tolkes. Det finnes ofte flere instruksjonsformater for en og samme prosessor, og opkoden brukes derfor til å finne ut hvilket format den aktuelle instruksjonen har. Instruksjonsformater for ALU-operasjoner har ofte felter for to kildeoperander og destinasjon, mens en kontrollflytinstruksjon (for eksempel hoppinstruksjon) har felt med referanse til neste instruksjon.

Eksempel

Som et eksempel vises en ADD instruksjon fra MIPS-prosessoren. Alle MIPS-instruksjoner er på 32-bit, og den har svært få forskjellige instruksjonsformater. Opkoden er alltid de 6 mest signifikante bitene. Instruksjonsformatet som benyttes til de fleste instruksjoner ser slik ut:



Vi skal sette opp instruksjonsordet til følgende instruksjon:

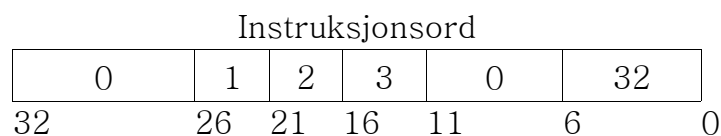
ADD \$3, \$1, \$2

Legg sammen register 1 og 2. Legg resultatet i register 3.

Feltene i instruksjonsordet over vil settes slik:

- Opkoden vil være 0. Dette spesifiserer hvilket instruksjonsformat, og at dette er en ALU-operasjon.
- rs er kilderegister én, og settes til 1
- rt er kilderegister to, og settes til 2
- rd er destinasjonsregister og settes til 3
- shamt brukes ikke i denne instruksjonen, settes til 0
- funct settes til 32, som betyr addisjon

Da ender vi opp med et instruksjonsord som ser slik ut:



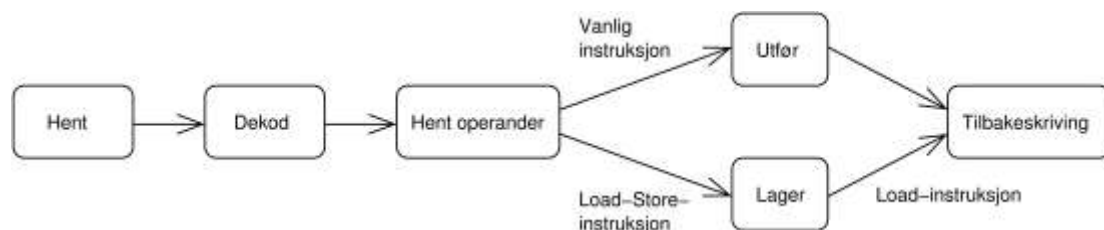
Som binær: 00000000 00100010 00011000 00100000
Som heksadesimal: 00221820

4.9 – Instruksjonssykel

Når prosessoren utfører en enkelt instruksjon går den gjennom flere stadier. Hvilke stadier som er nødvendige avhenger av prosessorarkitektur og instruksjon.

MIPS-prosessoren benyttes som et enkelt eksempel. Dette er en såkalt RISC-prosessor med en Load-Store-arkitektur. En LS-arkitektur medfører at alle operander må befinne seg i registerfilen. Det er altså bare mulig å gjøre operasjoner på registre, og aldri på data i datalageret. Eneste tilgang til datalageret er gjennom LOAD- og STORE-instruksjoner. LOAD leser data fra datalageret og legger i et register. STORE tar data fra et register og skriver til datalageret.

Med en slik arkitektur kan instruksjonssykel være slik:



1. Hent (engelsk: fetch): Hente instruksjon
2. Dekod (engelsk: decode): Dekode instruksjon
3. Hent operander (engelsk: operand fetch)
4. Utfør (engelsk: execute): Utføre instruksjon
5. Lager (engelsk: memory): Lese til / skrive fra datalager
6. Tilbakeskriving (engelsk: writeback): Skrive resultat av instruksjon til registerfil

Hver instruksjon må gjennom de fleste av disse stadiene hver gang de utføres. En vanlig instruksjon vil gå gjennom stadiene 1-2-3-4-6, en LOAD-instruksjon vil gå gjennom stadiene 1-2-3-5-6 og en STORE-instruksjon vil gå gjennom stadiene 1-2-3-5.

4.9.1 – Hent

Første stadium går ut på å hente instruksjonen fra instruksjonslageret.

4.9.2 – Dekod

Etter at instruksjonen er hentet ut må den dekodes. Dette vil si å se på instruksjonens opkode (og muligens noen andre felter) for å finne ut hva som skal utføres. På bakgrunn av dette settes alle styresignaler for resten av prosessoren opp. Dette gjøres av styreenheten.

4.9.3 – Hent operander

Når instruksjonen er dekodet vet styreenheten hvilke operander som trengs. Neste skritt er da å hente ut disse operandene fra registerfilen.

4.9.4 – Utfør

Når alle operander er klare har prosessoren all informasjon den trenger, og instruksjonen kan utføres. Dette gjøres av den funksjonelle enheten (skifter og ALU). For eksempel for en ADD-instruksjon er det her selve addisjonen utføres.

4.9.5 – Lager

Dersom instruksjonen er en Load/Store-instruksjon vil den gå inn i dette trinnet i stedet for «utfør»-trinnet. Her skrives eller leses datalageret.

4.9.6 – Tilbakeskriving

Til slutt må resultatet av instruksjonen lagres til riktig register i registerfilen. Dersom instruksjonen er en Store-instruksjon trengs ikke dette steget.

4.10 – Instruksjonssykel i maskinvare

Det finnes mange måter å realisere instruksjonssykel i maskinvare på, hver med sine fordeler og ulemper.

Prosessoren jobber i takt med et klokkesignal. Frekvensen av denne klokken benyttes ofte (feilaktig) som et mål på hvor rask en prosessor er. Dette blir feil fordi det er stor variasjon mellom prosessorarmodeller når det gjelder hvor mye de gjør per klokkesykel.

4.11 – En-sykelmaskin

Prosessoren kan lages som en en-sykelmaskin, det vil si at alle stadiene i instruksjonssykel utføres i samme klokkesykel. Da får man utført én instruksjon per sykel, det vil si en CPI (Cycle Per Instruction) på 1.0. Dette medfører som oftest svært lav klokkefrekvens fordi kritisk sti (lengste sti gjennom prosessoren m.h.p tidsforbruk) blir svært lang -- «Alt» må gjøres på en eneste sykel. Man må ha en klokke som går sakte.

4.12 – Fler-sykelmaskin

En annen arkitektur kan være en fler-sykelmaskin. Her vil man benytte flere klokkesyklar for å utføre en instruksjon. Man kan for eksempel bruke én sykel på hvert av stegene i instruksjonssykel slik at man får én sykel på å hente instruksjonen, én sykel på å dekode etc. Fordelen med dette er kort kritisk sti og dermed høy klokkefrekvens. Ulempen er at man bruker mange sykler per instruksjon (CPI > 1.0), og prosessoren får gjort mindre per klokkesykel.

4.13 – Samlebåndsmaskin

Det finnes en måte å få til høy klokkefrekvens på, samtidig som man i gjennomsnitt får utført (nesten) én instruksjon per klokkesykel. Denne teknikken kalles samlebånd og benyttes i alle prosessorer der høy ytelse er vesentlig. Samlebånd kan illustreres med et hverdagslig eksempel: Tenk deg at du skal vaske klær og har én vaskemaskin og én tørketrommel til rådighet. Vaskemaskinen er bare stor nok til å ta halvparten av klærne, så du må derfor kjøre to runder i vaskemaskinen. Den naive måten å ta vasken på vil være slik:

- Vaske første halvpart av klærne
- Tørketromle første halvpart av klærne
- Vaske andre halvpart av klærne
- Tørketromle andre halvpart av klærne

Dette er lite effektivt. Til enhver tid står enten vaskemaskinen eller tørketrommelen ubrukt. Da er det bedre å gjøre det på samlebåndsmetoden:

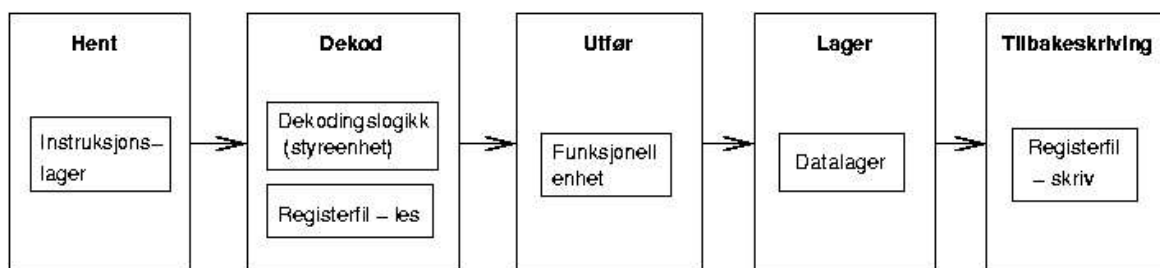
- Vaske første halvpart av klærne
- Tørketromle første halvpart av klærne, samtidig som du vasker andre halvpart
- Tørketromle andre halvpart av klærne

Nå har du spart én operasjon. Og dersom du har mange runder med klesvask, vil du spare mye mer. Samme gjelder for prosessorer. Den viktige observasjonen her er altså at de forskjellige stegene i instruksjonssyklusen benytter forskjellige enheter:

- Steg 1 (Hent) benytter instruksjonslageret
- Steg 2 (Dekod) benytter dekodingslogikken
- Steg 3 (Hent operander) leser fra registerfil
- Steg 4 (Utfør) benytter den funksjonelle enheten
- Steg 5 (Lager) benytter datalageret
- Steg 6 (Tilbakeskriving) skriver til registerfil. Skrivning til registerfil skjer uavhengig av lesing, og dette steget vil derfor ikke være avhengig av hva som skjer i steg 3.

Dette kan man utnytte. I stedet for å ta hver instruksjon for seg og dermed la de fleste enheter være ubrukt mesteparten av tiden, kan man la alle enheter jobbe hele tiden. Man splitter prosessoren inn i samlebåndssteg som tilsvarer stegene i instruksjonssyklusen. En instruksjon vil først komme inn i samlebåndssteg 1. Neste sykel går den videre til samlebåndssteg 2 etc., helt til den er ferdig med alle samlebåndsstegene. Siden stegene er uavhengige av hverandre kan man etter at første instruksjon har nådd steg 2, ta inn en ny instruksjon i steg 1. På denne måten vil man kunne fylle opp samlebåndet slik at hvert samlebåndssteg inneholder en instruksjon. Prosessoren jobber altså med mange forskjellige instruksjoner på én gang, på hvert sitt sted i instruksjonssyklusen.

Som eksempel brukes samlebåndet til MIPS-prosessoren. Denne har 5 samlebåndssteg. Disse tilsvarer stegene i instruksjonssyklusen, bortsett fra at «Decode» og «Operand Fetch» er slått sammen til ett samlebåndssteg.



Instruksjoner flyter fra venstre til høyre i figuren over: Først når de «Hent», deretter «Dekod», «Utfør», «Lager» og «Tilbakeskriving». Alle instruksjoner går gjennom alle samlebåndssteg. Selv om en instruksjon ikke benytter seg av ressursene i ett av stegene vil den likevel gå gjennom dette steget. Samlebåndets natur gjør at det ikke er noe poeng i at en instruksjon hopper over et steg -- neste instruksjon vil uansett ikke bli ferdig tidligere.

Eksempel

Tenk deg et program på 8 instruksjoner. Disse vil flyte gjennom samlebåndet på denne måten:

Instruksjonsflyt gjennom samlebåndet												
	1	2	3	4	5	6	7	8	9	10	11	12
instr1	IF	ID	EX	MEM	WB							
instr2		IF	ID	EX	MEM	WB						
instr3			IF	ID	EX	MEM	WB					
instr4				IF	ID	EX	MEM	WB				
instr5					IF	ID	EX	MEM	WB			
instr6						IF	ID	EX	MEM	WB		
instr7							IF	ID	EX	MEM	WB	
instr8								IF	ID	EX	MEM	WB

Figuren viser klokkesyklus (tid) horisontalt, og instruksjoner vertikalt. IF = Hent, ID = Dekod, EX = Utfør, MEM = Lager, WB = Tilbakeskriving.

På sykel 1 vil første instruksjon komme inn i «hent»-steget. På sykel 2 vil første instruksjon fortsette til «dekod»-steget, samtidig som at andre instruksjon kommer inn i «hent»-steget. Slik fortsetter det. På sykel 6 er første instruksjon ferdig. Og nå ser man fordelingen med samlebånd: Heretter blir det ferdigstilt en instruksjon for hver eneste sykel. Og alle samlebåndsstegene er i bruk samtidig.

Disse 8 instruksjonene bruker 12 sykler på å gjøre seg ferdig. Dette kan regnes ut slik (der # betyr «antall»):

$$\#sykler = \#samlebåndssteg + \#instruksjoner - 1$$

Siden antall samlebåndssteg er konstant, ser vi at et samlebånd kan gi en CPI (cycles per instruction) tilnærmet lik 1.0 når antall instruksjoner er stort. Altså nesten like bra som en én-sykel-maskin, men med den høye klokkefrekvensen til en fler-sykel-maskin.

4.14 – Samlebånd og kontrollflyt

Kontrollflytinstruksjoner er vesentlige i alle reelle programmer. Slike instruksjoner medfører hopp til et annet sted i programkoden. Men disse representerer et problem for samlebåndet. Som du sikkert forsto av eksempelet i rommet

«Instruksjonssykel i maskinvare» fungerer bare samlebåndet optimalt så lenge alle samlebåndsstegene er i bruk. I begynnelsen av programmet går det noen sykler før man får fylt opp samlebåndet med instruksjoner.

En betinget hoppinstruksjon medfører at prosessoren enten fortsetter på samme sted, eller at den begynner å hente instruksjoner fra et annet sted i programlageret. Problemet er at prosessoren ikke vet om hoppet skal utføres før hoppinstruksjonen har nådd «utfør»-steget. Og selv for ubetingete hoppinstruksjoner har ofte ikke prosessoren adressen til neste instruksjon klar før hoppinstruksjonen er i «utfør».

Dersom prosessoren finner ut at hoppet ikke skal foretas fortsetter den som om ingen ting har skjedd.

Men dersom prosessoren finner ut at hoppet skal tas, må flere ting skje. Dette finner den ut først når hoppinstruksjonen er i «utfør»-steget, og instruksjonene i «hent» og «dekod» vil derfor ikke være gyldige (de er hentet fra feil plass i instruksjonslageret). Disse må «flushes», det vil si slettes. Deretter må prosessoren begynne å hente nye instruksjoner fra rett plass. Slettingen medfører tomme samlebåndssteg, og dermed dårlig utnyttelse av samlebåndet. Prosessoren har «tapt» flere sykler på å utføre hoppet.

Samlebånd med kontrollflytinstruksjon												
	1	2	3	4	5	6	7	8	9	10	11	12
instr1	IF	ID	EX	MEM	WB							
instr2		IF	ID	EX	MEM	WB						
instr3 (hopp til 8)			IF	ID	EX	MEM	WB					
instr4				IF	ID							
instr5					IF							
instr8						IF	ID	EX	MEM	WB		
instr9							IF	ID	EX	MEM	WB	
instr10								IF	ID	EX	MEM	WB

Tabellen over viser samlebåndet for en programsekvens der den tredje instruksjonen er et betinget hopp til instruksjon 8. Prosessoren vet ikke at den skal hoppe før hoppinstruksjonen er i «utfør»-steget. Instruksjon 4 og 5 vil derfor feilaktig komme inn i samlebåndet. I sykel 5 oppdager prosessoren at den skal hoppe, og sletter derfor alt som ligger i «hent»- og «dekod»-stegene (instruksjonene 4 og 5). I neste sykel vil instruksjon 8 komme inn i samlebåndet. Vi ser her at prosessoren har «tapt» to sykler, den har brukt to sykler på å hente inn instruksjonene 4 og 5 uten at disse skal brukes.

Det finnes flere strategier for å forbedre ytelsen i forbindelse med hopp. De fleste moderne prosessorer benytter en eller annen form for forgreningsprediksjon (engelsk: branch prediction). «Hent»-steget inneholder logikk, en

forgreningsprediksjonsenhet (engelsk: branch prediction unit), som prøver å forutsi om hoppet blir tatt eller ikke, og henter instruksjoner fra stedet som den antar er rett. Dersom forutsigelsen stemmer (blir sjekket når hoppinstruksjon er i «utfør») har prosessoren spart seg for problemene i forbindelse med hopp. Dersom forutsigelsen ikke stemmer, må den slette instruksjoner akkurat som avsnittet over. Kunsten blir dermed å lage en forgreningsprediksjonsenhet som er god til å gjette utfallet av en hoppinstruksjon.

Forgreningsprediksjon kan gjøres på flere måter:

- Forutsi «aldri tatt»: Dette er det samme som en prosessor uten forgreningsprediksjon. Alle hopp som blir tatt medfører sletting.
- Forutsi «alltid tatt»: Alle hopp som IKKE blir tatt medfører sletting
- Forutsi basert på opkode: Opkoden bestemmer om hoppet forutsies tatt eller ikke. Man setter altså statisk opp hvilke instruksjoner som forutsies tatt eller ikke tatt når prosessoren konstrueres.
- Forutsi basert på hopphistorie: En hoppinstruksjon er ofte del av en løkke, og utføres gang etter gang – ofte med samme resultat (hopp eller ikke hopp). Det er sannsynlig at dersom instruksjonen medførte hopp forrige gang vil den også medføre hopp neste gang. Derfor kan en tabell benyttes til å lagre utfallet av hoppinstruksjoner. Denne kan benyttes som basis neste gang hoppinstruksjonen skal forutsies.

De tre første metodene er statiske; en hoppinstruksjon vil forutsies likt uansett hva som har skjedd tidligere. Den fjerde metoden er dynamisk; en instruksjon vil forutsies til å hoppe slik den har vist seg å hoppe tidligere i det samme programmet. Det er denne metoden som gir best resultat.

4.14.1 – Dataavhengigheter

Det er ikke bare hoppinstruksjoner som kan lage problemer i et samlebånd. Gitt følgende program:

```
ADD $1, $2, $3 ; legg sammen registre 2 og 3, svaret legges i
                ; register 1
ADD $4, $1, $2 ; legg sammen registre 1 og 2, svaret legges i
                ; register 4
```

Dette er et eksempel på en dataavhengighet (engelsk: data hazard). Problemet er at den andre instruksjonen er avhengig av resultatet fra den første.

Samlebånd med kontrollflytinstruksjon												
	1	2	3	4	5	6	7	8	9	10	11	12
instr1	IF	ID	EX	MEM	WB							
instr2 (avhengig av resultat fra instr1)		IF	ID	ID	ID	ID	EX	MEM	WB			
instr3			IF	IF	IF	IF	ID	EX	MEM	WB		
instr4							IF	ID	EX	MEM	WB	
instr5								IF	ID	EX	MEM	WB

Siden operander hentes ut fra registerfilen i «dekod»-trinnet, betyr det at den andre instruksjonen må vente i «dekod»-trinnet helt til den første har forlatt «tilbakeskriving»-trinnet (man sier at instruksjonen oppholdes (stalles) i «dekod»-trinnet). Det blir altså et opphold på flere sykler hver gang en slik dataavhengighet oppstår. Dette er vist i tabellen over der instruksjon 2 må oppholdes i 3 sykler før den kan fortsette ut av «dekod»-trinnet.

Med et samlebånd er det mange situasjoner der slike avhengigheter lager problemer for instruksjonsutføringen. En egen avhengighetsenhet (engelsk: hazard unit) tar seg av å oppholde samlebåndssteg når dette er nødvendig på grunn av en avhengighet. Å garantere korrekt utføring av instruksjoner i et samlebånd, samtidig som man skal unngå for mye opphold av samlebåndssteg, er en av de vanskelige problemstillingene i forbindelse med konstruksjon av samlebåndsprosessorer.

4.14.2 – RISC og CISC

Opprinnelig ble datamaskiner programmert ved hjelp av maskinkode eller assemblyspråk. Instruksjonssettet til prosessoren ble laget for å være lett å programmere direkte. Etterhvert som høynivåspråk ble vanlig, ble instruksjonssettene utstyrt med instruksjoner som gjør mange av høynivåkonstruksjonene i maskinvare. Dette for å lette utviklingen av kompilatorer. Tendensen var stadig større instruksjonssett med stadig mer komplekse instruksjoner.

Etterhvert begynte en gruppe mennesker å se på mønstre i høynivå-generert maskinkode. Resultatene av disse studiene resulterte i en filosofi om å gjøre maskinvare som støtter høynivåspråk enklere, ikke mer kompleks.

Noen observasjoner som ble gjort:

- Folk programmerer ikke assembly lengere; ingen vits å lage et instruksjonssett som passer for hånd-koding.
- Komplekse instruksjoner blir sjeldent brukt av kompilatorer fordi det er vanskelig for kompilatoren å generere kode som passer til en kompleks, spesialisert instruksjon. Som oftest er det små enkle instruksjoner som dominerer. Det er derfor ingen vits å bruke dyr silisium på å implementere komplekse instruksjoner.
- Enkle instruksjoner er det enkelt å lage maskinvare for. Spesiellages instruksjonssettet slik at det er lett å få til samlebånd, vil det bli lettere å lage en rask prosessor.
- En fant ut at en liten andel av de ulike instruksjonstypene utgjorde en meget stor andel av de instruksjoner som blir utført av prosessoren.

Dette medførte en ny prosessordesign-filosofi: RISC (Reduced Instruction Set Computer). Til sammenligning ble de tidligere prosessorarkitekturer kalt CISC-arkitekturer (Complex Instruction Set Computer). Merk at det ikke finnes noen klar definisjon på det ene eller det andre.

Vanlige egenskaper ved RISC:

- Få og enkle instruksjoner -- enkle instruksjoner medfører enkel maskinvare
- Én instruksjon per sykel (pluss evt. samlebandsoppholding)
- Load/Store-arkitektur; man kan ikke gjøre aritmetikk på operander i RAM. RAM får man tilgang til kun gjennom egne LOAD og STORE-instruksjoner.
- Enkle adresseringsmodi
- Enkle og få instruksjonsformater. Instruksjonsordet har ofte samme lengde for alle instruksjoner.
- Stor registerfil. En Load/Store-arkitektur medfører at man stort sett bare jobber på registre. Mange generelle registre gjør at man slipper å bruke sykler på å lese/skrive til RAM.

Vanlige egenskaper ved CISC:

- Mange og komplekse instruksjoner. Medfører komplisert kontrollenhet, og er ofte vanskelig å lage et godt samleband til
- Kan bruke mange sykler for å utføre én instruksjon, for eksempel en kompleks løkke-instruksjon
- Ofte mange adresseringsmodi, med mulighet for operander i RAM
- Kompliserte og lange instruksjonsformater
- Få generelle registre

En RISC-prosessor vil være enklere å konstruere enn en CISC-prosessor. Den enkle arkitekturen er lettere å implementere og samlebandet blir enklere. Enklere arkitektur betyr mindre og raskere krets. Siden mindre plass blir brukt til logikk, får man plass til desto større registerfil og cache.

RISC-prosessorer har vist seg å ha mange gode egenskaper. Det er likevel få rene RISC-prosessorer i dag. Det finnes også få rene CISC-prosessorer. Nye prosessorer har en tendens til å ligne mest på RISC-prosessorer, men men noen CISC-lignende konstruksjoner i tillegg.

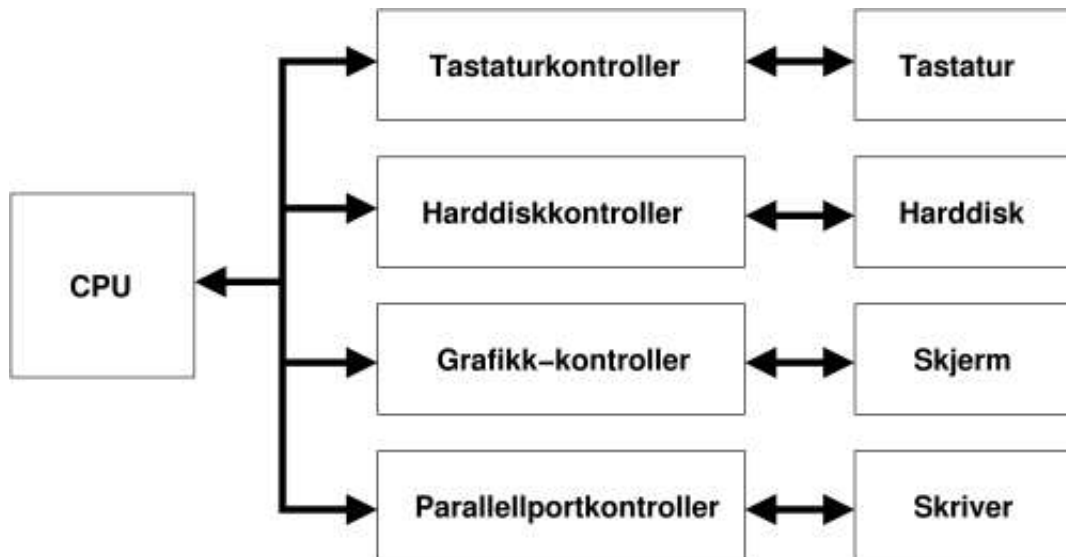
5 - I/O (input/output)

En prosessor som jobber for seg selv uten kontakt med omverdenen er det ikke mye nytte i. Derfor kobler man til I/O-enheter (I/O er forkortelse for Input/Output), som er et slags grensesnitt mot den virkelige verden.

Eksempler på I/O-enheter:

- Tastatur
- Skriver
- Skjerm
- Harddisk

Slike I/O-enheter kan styres direkte av prosessoren. Det gjøres nesten aldri fordi det kan være både komplisert og prosessorintensivt å styre slike enheter. Derfor benyttes i stedet en I/O-kontroller (også kalt I/O-modul) som tar seg av styring av I/O-enheten. Denne sitter mellom prosessoren og I/O-enheten, og gir prosessoren et enkelt grensesnitt mot I/O-enheten.



En I/O-kontroller har flere oppgaver:

- Kommunikasjon med I/O-enhet
- Kommunikasjon med CPU
- Mellomlagre data slik at CPU kan lese og skrive data når det passer, og ikke når I/O-enheten krever det
- Feildeteksjon og rapportering av I/O-enhetens tilstand

5.1 - Kommunikasjon mellom CPU og I/O-kontrollere

Som du lærte i I/O-rommet, kommuniserer ikke prosessoren direkte med en I/O-enhet. I stedet kommuniserer den med en I/O-kontroller. Det finnes flere måter prosessoren kan kommunisere med I/O-kontrolleren på:

5.1.1 – Programmert I/O (polling)

Det er den enkleste og mest ressurskrevende formen for I/O-operasjoner. Hvis man bruker programmert I/O vil dette foregå på følgende måte: Når en prosessor ønsker å skrive eller lese data fra en I/O-enhet, vil den sende en kommando til I/O-kontrolleren. I/O-kontrolleren vil så kontakte I/O-enheten på vegne av prosessoren og utføre oppgaven. Dersom data leses fra I/O-enheten vil I/O-kontrolleren legge dem i en intern buffer. Prosessoren må hele tiden sjekke status på I/O-kontrolleren for å vite om den har data klare fra I/O-enheten i bufferet sitt. Dette fører til at mye prosessorkraft går med til å sjekke status.

5.1.2 – Avbruddsdrevet I/O

Problemet med programmert I/O er at prosessoren er nødt til hele tiden å sjekke status til I/O-kontrolleren. Et alternativ er avbruddsdrevet I/O. Ved avbruddsdrevet I/O sender prosessoren en forespørsel til I/O-kontrolleren og fortsetter med andre beregninger. Når I/O-kontrolleren er klar til å utveksle data med prosessoren vil den sende et avbruddssignal. Prosessoren vet da at data er i bufferet og overfører disse fra bufferet til primærlageret, før I/O-kontrolleren behandler resten av I/O forespørselen. Dette fører til at vi får en minimal kostnad for prosessoren å sjekke om data er klare.

En analogi til dette er hvis man ser på prosessoren som eieren av et hus, dataene som skal overføres som gjester som skal inn i huset. Hvis man da skal illustrere programmert I/O vil det være som om huset ikke hadde ringeklokke, og eieren måtte med jevne mellomrom gå ut på trappen for å sjekke om det er kommet gjester. Derimot hvis man bruker avbruddsdrevet I/O vil man ha ringeklokke på huset, slik at eieren slipper å gå ut å sjekke om det står noen på trappen hele tiden, han må bare passe på å lytte om ringeklokken ringer.

5.1.3 – DMA (direct memory access)

Avbruddsdrevet I/O er som regel mer effektivt enn programmert I/O, men det har sine ulemper: Prosessoren må selv overføre data fra I/O-kontrolleren til lageret. Selv om prosessoren kan gjøre andre ting mens I/O-kontrolleren jobber, må den fremdeles ta seg av selve dataoverføringen.

Løsningen på dette er å sette en annen og mindre prosessor til å ta seg av I/O-overføringer. Denne teknikken kalles DMA (Direct Memory Access) – DMA-kontrolleren har direkte tilgang til lageret og data trenger ikke gå via prosessoren.

Ved bruk av DMA blir data overført på følgende måte: CPU-en sender en forespørsel til DMA-kontrolleren om å gjøre en overføring av data fra en I/O-enhet til primærlageret. CPU-en blir så frigitt til å gjøre andre beregninger mens den venter på signal fra DMA-kontrolleren. Når I/O-overføringen er ferdig, sender DMA-kontrolleren et avbrudd til prosessoren som forteller at data som er forespurt ligger klar i lageret.

Når DMA-kontrolleren skal overføre data er den nødt til bruke systembussen. DMA-kontrolleren må vente på at denne er ledig, eller den må tvinge prosessoren til å gi den fra seg i en kort periode. Den siste muligheten er det som oftest blir brukt og kalles «sykelstjeling» (engelsk: cycle stealing)

5.2 – Busser

En buss er en kommunikasjonsvei som kobler sammen to eller flere enheter. Den er altså et kommunikasjonsmedium. Kun en enhet kan sende noe over en buss på et gitt tidspunkt, men alle de andre enhetene som er koblet til bussen kan motta. En buss består typisk av flere kommunikasjonslinjer, hver med mulighet til å sende et binært signal. En vanlig PC har koblet til seg ulike busser for eksempel systembussen, IDE, Firewire, USB, SCSI, PCI, AGP og ISA.

Systembussen i en datamaskin består typisk av femti til hundrevis av separate linjer, hver har sin egen funksjon. Det finnes mange forskjellige bussarkitekturer, men i hovedtrekk kan vi gruppere linjene i data-, adresse- og styre-linjer.

På datalinjene overføres data som skal sendes. Bredden (antallet linjer) på databussen bestemmer hvor mye data som kan overføres for hver klokkesykel. Det er ikke nødvendigvis slik at jo flere datalinjer jo raskere er bussen. Hvis man sender data over en linje (serielt) kan man ofte overføre med høyere hastighet fordi man blant annet slipper problemene med å få data på alle datalinjene synkronisert.

Adresselinjene forteller hvor kilden eller destinasjonen til dataene er. Hvis for eksempel prosessoren skal lese fra en gitt minneadresse legges denne adressen på adresselinjene slik at minnet vet hvilket ord som skal legges på datalinjene. Antall adresselinjer avgjør selvsagt hvor stort adresseområde som kan nås via denne bussen.

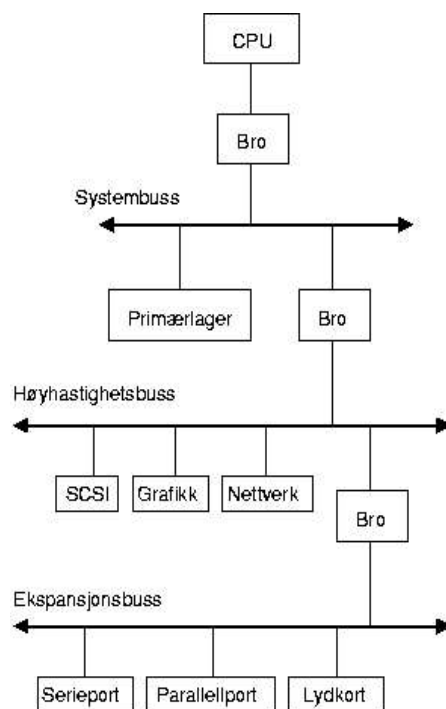
Styrelinjene brukes til å styre hva bussen skal brukes til, for eksempel hvem som skal skrive til den.

5.2.1 – Bussarkitektur

Når man kobler mange enheter til en buss vil man generelt få en tregere buss. Dette skyldes i hovedsak to ting:

- Bussen blir fysisk lengre, dermed vil tiden det tar for det elektriske signalet å komme fra den ene siden av bussen til den andre øke. Dermed er man nødt til å senke frekvensen på bussen slik at man er sikker på at alle enhetene mottok siste signal på bussen, før man begynner å sende et nytt. Hvis to signaler blir sendt samtidig vil signalene bli ødelagt.
- Når man kobler flere enheter til bussen vil konkurransen om å få sende over bussen øke. Dermed vil sannsynligvis enhetene bruke mer tid på å vente på å få kontroll over bussen.

Løsningen på dette problemet er å dele opp bussen i flere nivåer, det vil si at vi i prinsippet sitter igjen med flere busser som er koblet sammen med broer (enheter som kobler sammen forskjellige busser). I PC-er har man et hierarki av busser med ulik hastighet. Dette er vist i figuren under. Prosessoren er koblet til systembussen, der primærlageret befinner seg. Systembussen kobles sammen med en høyhastighets ekspansjonsbuss gjennom en bro. Her kobles enheter som trenger rask overføring av data. Nederst er en lavhastighets ekspansjonsbuss for trege enheter.



5.2.2 – Multiplekset buss

De forskjellige roller til linjene i bussen (adresse, data og styring) ble beskrevet over. Man kan ha en buss der noen av de fysiske linjene har flere roller. Man kan for eksempel ha 32 datalinjer som også kan fungere som 32 adresselinjer. Dette kalles multiplekset buss. En bussoverføring som tar flere sykler å gjennomføre vil kunne sette av 32 linjer i 1. sykel til adresse, for så å bruke de samme linjene i neste sykel til data. På denne måten vil man få færre antall linjer i bussen, og dermed en enklere buss å håndtere fysisk. Linjer som ikke er multiplekset kalles dedikerte.

5.2.3 – Arbitrering

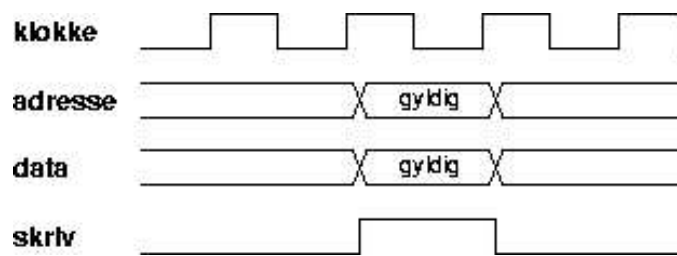
En buss kan ha en spesiell enhet som alltid bestemmer hvem som skal skrive til bussen. Vi har en herre (engelsk: master) og flere slaver (engelsk: slave).

Noen busser kan ha flere herrer. Da får vi et problem med å styre hvem som skal ha kontrollen over bussen til enhver tid.

Dette løses vha arbitrering, som er navnet på teknikker der enhetene forhandler om hvem som skal få styre bussen. De forskjellige arbitreringsteknikkene kan grovt sett deles inn i to grupper: sentral og distribuert. Ved sentral arbitrering er det en enhet, gjerne en busskontroller, som har kontroll over bussen og den bestemmer hvilke enheter som har lov til å bruke bussen. Ved distribuert arbitrering finnes det ingen sentral kontroller, enheten må bli enige seg i mellom hvem som skal bruke bussen. Dersom arbitreringen skjer uten å «bruke opp» sykler for dataoverføringen, kalles arbitreringen for «skjult».

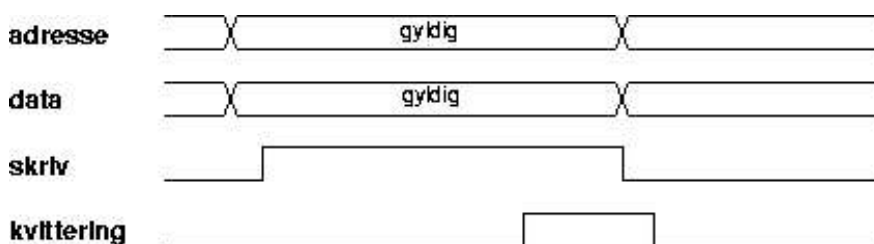
5.2.4 – Timing (tidsstyring)

Timingen refererer til hvordan hendelser er koordinert over bussen. Busser er enten synkrone eller asynkrone.



Ved synkron timing er det en klokke som avgjør når og hvor lenge et signal skal ligge på bussen. Denne klokken må være felles for alle enhetene som ligger på bussen, og klokkesignalet er typisk en del av styrelinjene. Sykeltiden må tilpasses slik at den er lang nok selv for den tregeste enheten.

Dette er vist i figuren over. Vi har et klokkesignal, en adressebuss, en databuss og et signal som forteller at bussens herre ønsker å skrive til bussen. Når skriveoperasjonen skal starte, settes adresse- og databuss opp med riktige verdier, og skrivesignalet settes høy. Dette indikerer at alle enheter som skal lese bussen må lese adresse- og databuss på starten av neste klokkesykel.



Ved asynkron timing brukes styrelinjer til å fortelle at det ligger et gyldig signal på bussen, slik at den enheten som skal lese dette vet at signalet er klart til lesing. Tilsvarende vil det finnes styrelinjer som forteller at signalet er lest, og dermed kan fjernes igjen.

Dette er vist i figuren over. Her er det ikke noe klokkesignal som forteller når data skal skrives og leses. I stedet finnes et kvitteringssignal. Herren setter opp adressebuss og databuss som før, og setter deretter skrivesignalet. Nå vet slaven at den kan lese data fra bussen. Etter at slaven har lest data, setter slaven opp kvitteringssignalet for å fortelle at herren kan fjerne data fra bussen og fortsette med andre ting.

Forskjellen her er altså at vi i en synkron buss har et klokkesignal som dikterer hvor lenge data skal være tilgjengelige og når data skal skrives/leses. I en asynkron buss vil data ligge tilgjengelige helt til slaven har indikert at den er ferdig (gjøres med kvitteringssignalet).

5.2.5 – Eksterne busser

Eksterne busser brukes til å koble til I/O-enheter av forskjellig slag. Dette kan være mus, tastatur, lagringsmedier etc.

Vi deler eksterne busser inn i to hovedkategorier: parallelle og serielle. I parallelle

busser er det flere linjer som kobler sammen enhetene, og flere bit blir overført samtidig. I serielle busser brukes kun en linje for å transportere data slik at kun en og en bit kan overføres til samme tid. Eksempel på parallelle busser er SCSI, mens FireWire, USB, FibreChannel og Serial ATA alle er serielle.

SCSI (Small Computer System Interface) var først vanlig på Macintosh på 80-tallet, men brukes nå også i PC-er og arbeidsstasjoner. Brukes som regel som grensesnitt mot CD-ROM, harddisker, scannere osv. SCSI støtter 8-16 enheter per SCSI-kontroller og hastigheter opptil 320 MB/s.

FireWire (IEEE 1394) bruker som sagt seriell overføring noe som fører til billigere kabler og kontakter, mindre kontakter, enkel implementasjon samt at man unngår synkroniseringsproblem. FireWire støtter hastigheter opptil 400 Mbps og 63 enheter. Dessuten støttes både «Hot plugging» (mulighet for å koble til enheter mens strømmen er på) og «Plug and Play» (system som gjør at brukeren ikke trenger gjøre noe aktivt for å få enheten til å virke).

USB (Universal Serial Bus) brukes til enheter med lav eller medium datarate (12Mb/s = 1,5MB/s). Opptil 127 enheter kan tilkobles samtidig og USB støtter både «Hot Plugging» og «Plug and Play». Brukes gjerne til enheter som ikke bruker mye båndbredde slik som: mus, tastatur, modem, printer osv. Etterhvert har denne standarden blitt oppdatert til versjon 2.0 som øker båndbredden til 480Mbps og er bakoverkompatibel med tidligere versjoner.

FibreChannel er tenkt som en erstatning til SCSI. FibreChannel støtter deling av lagringsenheter mellom tjenere (SAN = Storage Area Network), 100-200MB/s i hver retning (seriell), og kan bruke kabellengder på opp til 10km (med optisk fiber).

Serial ATA er tenkt som den serielle erstatningen til ATA (for harddisker, CD-ROM osv). Kabelen bruker kun 7 ledere og overføringsrater er 150MB/s. Kommende versjoner vil ha overføringsrater på 300 og 600 MB/s.

6 - Lagerhierarki

Lageret i datamaskinen er den delen av maskinen der data (og program) ligger lagret. Det finnes flere typer lager i en datamaskin. De viktigste karakteristikkene til de forskjellige typene er kostnad, kapasitet og aksesstid. Det optimale lager har minimal kostnad, minimal aksesstid og maksimal kapasitet, men det er umulig å oppnå.

Det ble observert at selv om et program trenger et stort lager vil prosessoren stort sett bare jobbe på et lite subsett av gangen. Dette subsettet vil gradvis skiftes ut over tid men innen et lite tidsintervall er det et relativt lite sett av data som er «aktivt».

Dette er essensen i lokalitetsprinsippet. Vi har lokalitet i lagerreferanser. Det finnes to typer lokalitet:

- Temporær lokalitet; hvis prosessoren har gjort en aksess til en spesiell lagercelle er det sannsynlig at det blir gjort en aksess til samme lagercelle i nær fremtid
- Romlig lokalitet; hvis prosessoren har gjort en aksess til en spesiell lagercelle er det sannsynlig at det blir gjort en aksess til en nærliggende lagercelle i nær fremtid

Denne egenskapen kan utnyttes. Vi kan lage et stort lager som ikke er særlig raskt. I tillegg lager vi et lite og raskt lager som "bufrer" (det vil si inneholder en kopi av) det aktive datasettet som prosessoren jobber på for øyeblikket. Dersom vi klarer å ordne det slik at prosessoren nesten alltid finner data i det lille lageret vil vi få en enorm hastighetsøkning, samtidig som at prosessoren har mulighet til å benytte hele det store lageret. Dette kalles hurtigbufring og er en sentral del arkitekturen i en moderne PC.

En moderne PC har ikke bare to nivåer med lager slik som forklart over. Lageret er ordnet i et hierarki med flere forskjellige lagernivåer som hver bufrer data fra nivået under.

Lagerhierarkiet ser vanligvis slik ut:

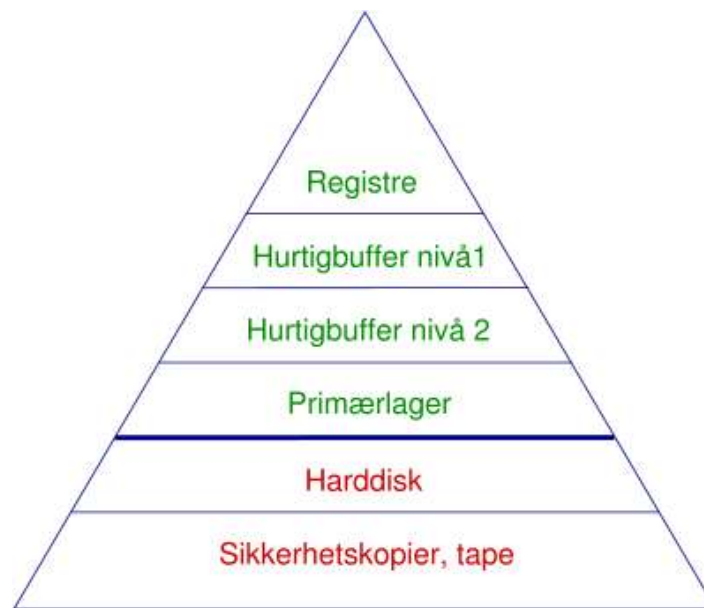
- Registre
- Hurtigbuffer, nivå 1
- Hurtigbuffer, nivå 2
- Primærlager
- Harddisk
- Sikkerhetskopier på kassett

Hierarkiet viser hvor nær de forskjellige lagertypene er prosessoren. Øverst finnes registrene, som prosessoren kan benytte direkte i utregninger. Registre er raske, men få i antall.

Etter hvert som man går ned i hierarkiet:

- Minker kostnaden per bit
- Øker kapasiteten
- Øker aksesstiden
- Kommer man lenger og lenger bort fra prosessoren

For å illustrere forskjellen i størrelse, tegnes ofte hierarkiet som en trekant, som vist i denne figuren:



Poenget med et lagerhierarki er å kunne ha et stort lager, samtidig som at hastigheten er høy og prisen lav. Man oppdaget at selv om man hadde behov for et stort lager, ville prosessoren bare jobbe med et lite sett data av gangen. Ved å ha et lite og raskt lager nær prosessoren, vil denne det meste av tiden klare seg med det. Når prosessoren en sjelden gang må ha tilgang til resten av lageret, bytter den ut innholdet i det lille raske lageret med nye data fra det større, tregere lageret. På denne måten vil prosessoren stort sett få hastigheten til det lille, raske lageret, men likevel størrelsen til det store, trege lageret. Dette fungerer bra så lenge prosessoren slipper å bytte ut data fra det lille lageret så ofte noe man oppnår med smarte utskiftningsalgoritmer.

Slik utskiftning av data har vi mellom alle lagene i lagerhierarkiet, og vi får i praksis et lager like raskt som om det består av bare registre, men med størrelsen (og kostnaden) til et lager av samme type som primærlageret. Merk at alle data ligger lagret nederst i hierarkiet. Oppover i hierarkiet ligger kopier av data som finnes lenger nede.

Under primærlageret finnes det to nivåer til: harddisk og magnetbånd. Alle nivåer over dette vil skifte ut data automatisk, uten at prosessoren trenger å gjøre noe selv for å få dette til. Støtte i operativsystemet kan benyttes til å øke hierarkiet enda mer, slik at en harddisk også blir del av lageret (kalles virtuelt minne).

6.0.1 – Lagertypenes kapasitet

Alle lager har en viss kapasitet, som måles i antall byte eller ord. I en vanlig PC vil størrelsen på...

- ...hurtigbuffer nivå 1 være ca 8kB–32kB
- ...hurtigbuffer nivå 2 ca 256kB–2MB
- ...primærlageret være mellom 128MB–1GB
- ...eksternlageret (harddisk) være mellom 30–160GB

6.0.2 – Lagertypenes aksessmetoder

En annen forskjell mellom lagertypene er aksessmetoder.

- Sekvensiell aksess: Starter på begynnelsen og leser gjennom lagerenheten i en lineær sekvens. Aksesstid avhenger av hvor data befinner seg. Brukes for eksempel i magnetbånd.
- Blokk–direkte aksess: Hver blokk (som er en samling med data) har en unik adresse. Man aksesserer data ved å hoppe direkte til riktig blokk og leter sekvensielt gjennom den etter ønsket data. Aksesstid avhenger av hvor data befinner seg. Brukes for eksempel i harddisker
- Direkte aksess: Hver datacelle kan adresseres direkte, det vil si man kan hoppe direkte til de data man ønsker. Brukes for eksempel i RAM
- Assosiativ aksess: Man identifiserer lagringssted basert på deler av innholdet i stedet for adresse. Brukes for eksempel i hurtigbuffer.

6.0.3 – Lagertypenes ytelse

De ulike lagertypene har ulik ytelse. Ytelsen blir målt med tre størrelser:

- Aksesstid: For direkte aksess er dette tiden en leseoperasjon eller en skriveoperasjon tar, for andre typer aksess er det tiden det tar å sette les/skrive mekanismen på rett plass.
- Sykeltid: Et konsept som er mest brukt ved RAM. Sykeltiden er aksesstid pluss tiden man trenger før neste aksess kan begynne
- Overføringsrate: Vil si hvor mye data man kan overføre per tidsenhet. For direkte aksess er overføringsraten lik $1/\text{sykeltiden}$. For andre typer har man formelen
- $T(N) = T_a + (N/R)$

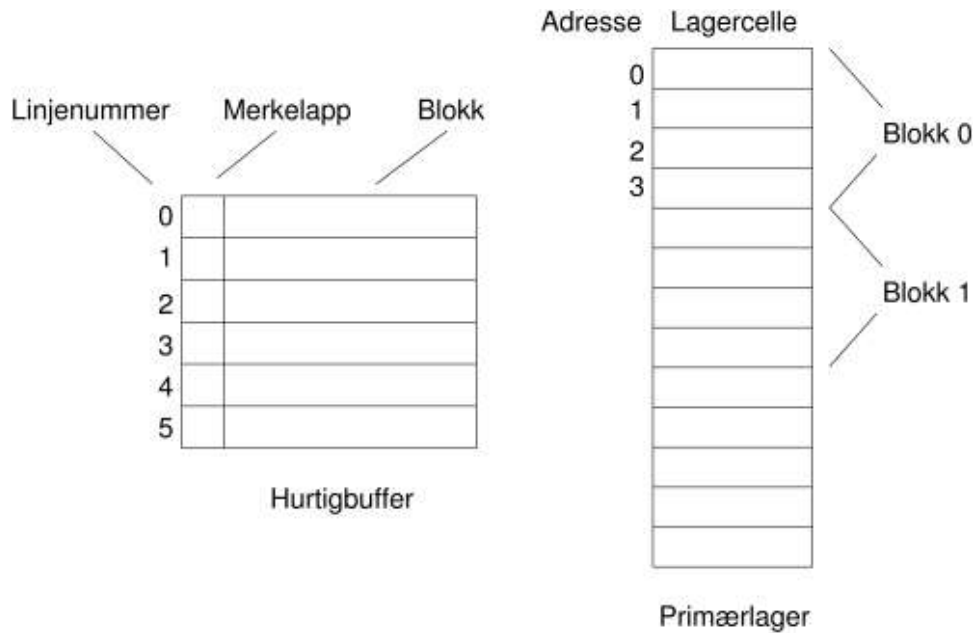
der

- $T(N)$ er gjennomsnittlig tid for lesing eller skriving av N bit
- T_a er gjennomsnittlig aksesstid, N er antall bit
- R er overføringsraten i bit per sekund(bps).

7 - Hurtigbuffer

Hurtigbufferet (engelsk: cache) befinner seg mellom prosessoren og primærlageret. Når prosessoren skal hente data fra lageret gjør den oppslag i hurtigbufferet. Hurtigbufferet er mye mindre enn den totale lagerstørrelsen og inneholder derfor bare en liten delmengde av det som finnes i primærlageret. Dersom hurtigbufferet ikke inneholder de aktuelle data sørger hurtigbufferet for å kopiere disse fra primærlageret og inn i hurtigbufferet.

Som du lærte i et tidligere rom, er hensikten med dette å øke hastigheten på lagertilgang. Hastigheten øker fordi hurtigbufferet er mye raskere enn primærlageret, og de fleste lageraksesser vil kun gå til hurtigbufferet.



Figuren over viser generelt hvordan primærlageret og hurtigbufferet er organisert.

Primærlageret er delt inn i lagerceller på ett ord hver. Hver lagercelle har en unik adresse. Lagercellene grupperes inn i blokker (for eksempel 8 celler i hver blokk).

Hurtigbufferet deles inn i linjer. En linje har plass til en blokk med data. Den inneholder også informasjon om hvilken blokk den inneholder (merkelappen).

Prosessoren henter alltid data fra hurtigbufferet og aldri direkte fra primærlageret. Når prosessoren forespør en datablokk som ikke finnes i hurtigbufferet må blokken kopieres fra lageret til hurtigbufferet. Siden det er færre hurtigbufferlinjer enn blokker i lageret, trenger man en algoritme som forteller hvilken hurtigbufferlinje man skal kopiere blokken til. Dette kalles avbildningsfunksjon.

Valg av avbildningsfunksjon bestemmer hvordan hurtigbufferet er organisert. Det finnes tre vanlige typer: Direkteavbildning, assosiativ avbildning og sett-assosiativ avbildning.

7.1 – Plasseringsmetoder

7.1.1 – Direkteavbildning

Direkteavbildning avbilder hver blokk i primærlageret inn i en gitt hurtigbufferlinje. Hver gang denne blokken blir kopiert inn i hurtigbufferet vil den bli plassert på akkurat den samme hurtigbufferlinjen som sist.

Avbildningen skjer slik:

$$\text{hurtigbufferlinjenummer} = \text{lagerblokknummer} \% M$$

hvor % er modulo-operatoren og M er totalt antall linjer i hurtigbufferet. Merkelappen til hurtigbufferlinjen settes til blokknummeret for å unikt identifisere hvilken blokk som er lagret.

Direkte avbildning er enkelt og ikke dyrt å implementere, men lite fleksibelt siden den har gitte plasser for alle lagerblokker. Hvis prosessoren hele tiden bytter mellom to minneblokker og de har samme hurtigbufferlinje, må man hele tiden bytte ut innholdet av denne linjen, selv om resten av hurtigbufferlinjene er tomme.

7.1.2 – Assosiativ avbildning

Assosiativ avbildning tar høyde for ulempene med direkteavbildning ved å tillate å la alle blokker bli lastet inn på en hvilken som helst hurtigbufferlinje.

Når en blokk skal buffres sjekkes først om den ligger i hurtigbufferet fra før ved å sjekke alle merkelappene. Gjør den ikke det vil den bli lagt i en hvilken som helst hurtigbufferlinje (helst en som er ledig eller lite brukt). Deretter oppdateres linjens merkelapp med riktig blokknummer.

Ulempen med assosiativ avbildning er at den trenger kompleks og dyr (det vil si plasskrevende) logikk.

7.1.3 – Sett-assosiativ avbildning

Sett-assosiativ avbildning er et kompromiss av de overnevnte, og inneholder egenskaper ved både direkteavbildning og assosiativ avbildning.

I dette tilfellet er hurtigbufferet delt inn i v sett som hver består av k linjer. Totalt antall linjer i hurtigbufferet (M) blir da:

$$M = v \cdot k$$

En gitt blokk vil avbildes direkte inn i et gitt sett. Blokknummeret vil altså entydig spesifisere hvilket sett som skal benyttes i hurtigbufferet. Avbildning inn i en gitt settlinje er derimot assosiativ, en gitt blokk kan legges i en hvilken som helst linje innen det gitte settet.

7.2 – Organisering

Da man begynte å bruke hurtigbuffer hadde man bare et enkelt hurtigbuffer. I senere tid har det blitt vanlig med flere nivå med hurtigbuffer. Man har flere nivå fordi man ønsker å ha høy hastighet og høy treffrate. Begge disse faktorene er

avhengig av størrelsen på hurtigbufferet og det er ikke så lett å få begge kravene oppfylt i et og samme hurtigbuffer. Løsningen på problemet er blitt at man har to nivåer. Nivå 1 som er lite og raskt og nivå 2 som er stort (høy treffrate) og «tregt», men likevel raskere en primærlageret.

Før var det vanlig å lagre instruksjoner og data i samme hurtigbuffer. Fordelen med felles hurtigbuffer for instruksjoner og data er at man kan få høyere treffrate og det er enkelt. Nå er det mer vanlig å separere instruksjoner og data inn i hvert sitt hurtigbuffer. Fordelen med separate hurtigbuffer er at man får færre kollisjoner. For å få fordeler fra begge organiseringsmåtene er det vanlig å ha nivå 1 separert og nivå 2 felles.

7.3 – Erstatningsalgoritmer for hurtigbufferet

Når en ny blokk blir satt inn i hurtigbufferet må en av de eksisterende ut. For direkteavbildning er blokken som må ut gitt på forhånd, men for assosiativ avbildning trenger man en erstatningsalgoritme. Fire vanlige algoritmer er:

- LRU (least recently used): Bytter ut den blokken som det er lengst tid siden har blitt brukt.
- FIFO (first-in-first-out): Bytt ut den blokken som har vært i hurtigbufferet lengst, uavhengig av hvor ofte den har blitt brukt.
- LFU (least frequently used): Bytt ut den blokken som har blitt brukt minst.
- Random: Velg en tilfeldig linje som skal byttes ut.

7.4 – Skrivestrategi

Prosessoren skriver nye data til hurtigbufferet. Hvordan skal man sørge for å holde hovedlageret oppdatert med de nye dataene? Det finnes to strategier for å løse dette:

- Gjennomskrivning (engelsk: write through): Hver gang data endres i hurtigbufferet skrives de samme data tilbake til hovedlageret slik at hurtigbuffer og hovedlager til enhver tid er likt.
- Utsatt tilbakeskriving (engelsk: write back): Data kan endres i hurtigbufferet uten at hovedlageret oppdateres med de nye data. Hovedlageret oppdateres kun når en blokk i hurtigbufferet skal skiftes ut.

Fordelen med gjennomskrivning er at man er garantert at hovedlageret alltid inneholder gyldige data (det vil si samme data som i hurtigbufferet). Ulempen er at man må gjøre oppslag i hovedlageret hver gang man skal skrive en verdi, noe som går tregt.

Fordelen med utsatt tilbakeskriving er at man slipper å gjøre oppslag i hovedlageret hver gang man skriver en verdi. Ulempen er at hovedlageret og hurtigbuffer kan inneholde forskjellige verdier. Utskiftning av blokker i hurtigbufferet medfører derfor sjekk på om blokken som skal skiftes ut er forandret, og isåfall skrives denne tilbake til primærlageret før den slettes. Dette skaper også problemer i datamaskiner der andre enheter har tilgang til primærlageret (DMA) for eksempel fordi disse kan endre primærlageret uten å

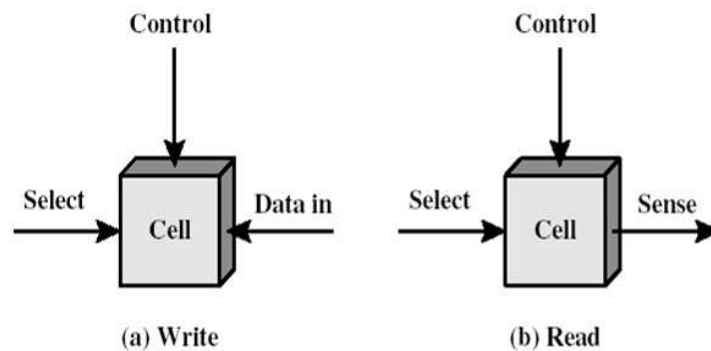
endre tilsvarende i hurtigbufferet.

8 - Primærlageret

Primærlageret, eller hovedlageret, er det største direkte adresserbare lageret i datamaskinen. Over dette nivået i lagerhierarkiet finner vi hurtigbufferet. Under dette nivået i lagerhierarkiet finner vi eksternt lageret, som bare kan benyttes i lagerhierarkiet dersom man har operativsystemstøtte for dette.

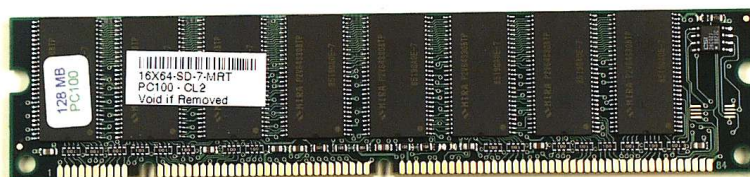
Basiselementet til et lager er lagerceller. Typiske egenskaper ved lagerceller er:

- De har to stabile tilstander, som kan bli brukt til å representere 0 og 1 (en bit).
- Man kan skrive til en celle minst en gang for å sette tilstanden.
- Det er mulig å lese cellene for å finne tilstanden.



Primærlageret består hovedsaklig av RAM (random access memory). I noen systemer er det også naturlig å definere ROM (read only memory) som en del av primærlageret. Både RAM og ROM er «random aksess», som vil si at man har direkte tilgang til hele lageret. Du vil få lære mer om RAM og ROM senere.

Figuren under viser et eksempel på et DIMM-kort (Dual In-line Memory Module) som er en vanlig minnebrikke i PC-er. Denne finner du på hovedkortet i nærheten av prosessoren.



8.1 - RAM

RAM (Random Access Memory) er en lagerteknologi som det er mulig å både lese fra og skrive til elektronisk. En annen egenskap ved RAM er at den er flyktig, det vil si at strømmen må være på for at innholdet skal bli bevart. De to hovedtypene RAM er statisk RAM og dynamisk RAM.

8.1.1 – Statisk RAM

Statisk RAM består av matriser av vipper, der hver vippe (består av 4–6 transistorer) utgjør en celle (bit). SRAM-celler vil bevare sin verdi så lenge strømmen er på.

Fordelen med SRAM er at det er raskt. Ulempen er at det er dyrt (det vil si plasskrevende).

Brukes blant annet i hurtigbuffer.

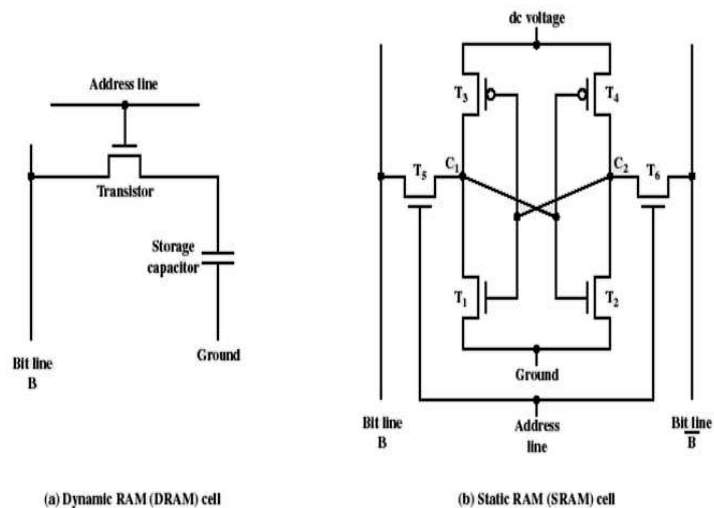
8.1.2 – Dynamisk RAM

Dynamisk RAM (DRAM) er laget av celler som består av en transistor og en kondensator. Cellene lagrer data som ladning på kondensatoren. Er ladningen over en viss verdi vil det bli tolket som 1, ellers 0. Kondensatorer lades ut over tid, noe som gjør at DRAM krever periodisk oppfriskning av kondensatorladningen for ikke å miste data.

Fordelen til DRAM er at det er billig (det vil si tar liten plass). Ulempen er lav hastighet og at datamaskinen må oppfriske innholdet hele tiden.

Bruksområdet til DRAM er blant annet primærlager i PC-er.

Figuren til høyre viser hvordan DRAM (venstre) og SRAM (høyre) er bygget opp, og er tatt med for å illustrere forskjellen i størrelse på celler i de to lagertypene. Kilde: W. Stallings, Computer Organization & Architecture, 6th. edition, p. 141.



8.1.3 – Synkron DRAM

DRAM har blitt videreutviklet til synkron DRAM (SDRAM), fordi man ønsker en høyere ytelse i primærminne.

Vanlig DRAM er asynkron, og gir ut data så snart den har de klare. Prosessoren må vente til data er klare, og bussen er opptatt helt til overføringen er ferdig.

SDRAM er synkron, noe som vil si at den er styrt av klokkesignalet på systembussen. Når prosessoren ber om data, vil SDRAM svare etter et gitt antall syklene. I mellomtiden kan prosessoren fortsette med andre ting, og systembussen er ledig til annet bruk.

Dette gjør SDRAM til den mest vanlige formen for dynamisk RAM i moderne PC-er.

8.2 – ROM

ROM (Read only memory) kan leses på samme måte som RAM. Men man kan ikke skrive nye data inn til den. Man trenger ikke strøm for å holde på verdiene i ROM. Det finnes flere typer ROM; ROM, PROM, EPROM, EEPROM og Flash.

8.2.1 – ROM

ROM har data fastprogrammert inn i brikken som en del av fabrikkingsprosessen.

Et problem med ROM er at den må fabrikeres med riktig innhold, noe som gjør det dyrt å lage noen få ROM-brikker. Det lønner seg ikke før man skal lage tusenvis.

Man bruker ROM til oppstartsrutiner og biblioteksfunksjoner i enheter som lages i stort antall.

8.2.2 – Programmerbar ROM (PROM)

Innhold kan skrives elektronisk første gang, det vil si at innholdet ikke er fastprogrammert ved produksjon. Dette gjør den mer velegnet enn vanlig ROM når man trenger et lite antall brikker.

8.2.3 – Slettbar PROM (Erasable PROM – EPROM)

EPROM kan slettes ved å bruke UV-lys, slik at man kan skrive nye data til den mer enn en gang. Ulempen er at det tar lang tid, og krever UV-belysning av brikken. Belysning skjer gjennom et vindu på brikken, vist i figuren under.

8.2.4 – Elektronisk slettbar PROM (Electronically Erasable PROM – EEPROM)

I en EEPROM kan innholdet slettes elektronisk. På samme måte som for EPROM kan man dermed skrive nytt innhold flere ganger, men man slipper å belyse brikken mellom hver gang. Sletting skjer byte-vis. Sletting/skriving er tregt og EEPROM-er er dyre. På grunn av treg hastighet og at EEPROM-er ødelegges av å skrives til for mange ganger (de fleste tåler 10 000 overskrivinger), benyttes EEPROM til oppgaver der man sjelden trenger å endre innholdet.

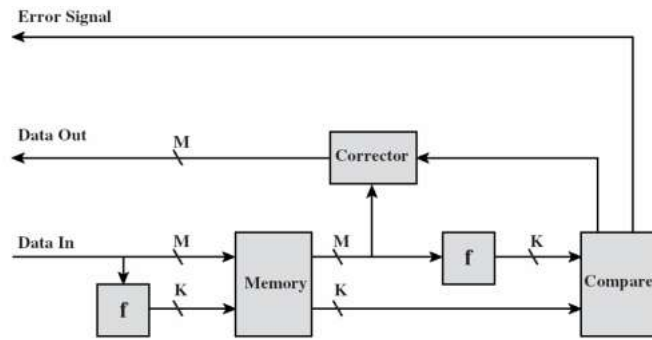
8.2.5 – Flash-minne

Flash har relativt hurtig skriving og billig i forhold til EEPROM. Tåler like mange overskrivinger som EEPROM. Skriving skjer blokkvis, og er derfor egnet når man som oftest bytter ut hele innholdet hver gang man skriver.

8.3 – Feilkorrigerende i primærlageret

I primærlageret kan det oppstå feil. Disse feilene kan være enten «harde» eller «myke». Harde feil er permanente fysiske feil slik at lagerceller blir upålitelige. Disse feilene kan oppstå ved produksjon, slitasje eller ytre skade. Myke feil er tilfeldige, ikkeødeleggende hendelser som forandrer innholdet til en eller flere lagerceller, uten å ødelegge lageret. Disse feilene kan oppstå ved strømforsyningsproblemer eller pga. alfapartikler (for eksempel kosmisk stråling i en satelitt).

Feil vil det alltid være, men det er mulig å bruke feilkorrigerende kode for å finne og/eller rette de opp.



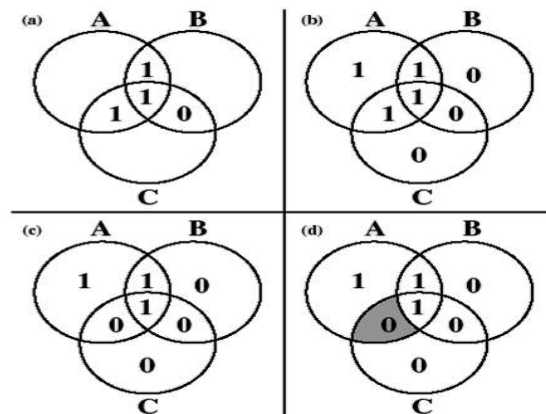
Figuren over viser hvordan en generell feilkorrigerende kode fungerer (Kilde: W. Stallings, Computer Organization & Architecture, 6th edition, p. 149).

Når et dataord skrives til lageret utføres en funksjon f på dataordet, med en kode som resultat. Både dataord (M) og kode (K) blir lagret. Når ordet blir lest ut bruker man koden til å sjekke om det har oppstått en feil og muligens rette den. I sammenlikningsprosessen ender man opp med et av disse resultatene:

- Ingen feil er detektert, send data ut
- En feil er detektert og det er mulig å rette den, send korrigert data ut
- En feil er detektert, men kan ikke rettes, send feilsignal

8.3.1 – Hammingkode

Den enkleste feilkorrigerende koden er Hammingkoding. Figuren under viser hvordan Hammingkoding fungerer for et 4-bits ord.



Figur a) viser plassering av de fire databitene.

Figur b) viser de innsatte paritetsbitene. Hver sirkel skal ha et like antall bits som er 1. Det medfører at dersom det er et odde antall enere fra før, setter man inn en ekstra ener i sirkelen. Er det allerede et like antall enere, setter man inn en bit med verdien null.

Figur c) viser dataordet etter at det har oppstått en feil

Figur d) viser at feilen er detektert. Ved å se på paritetsbiten i sirkel A ser man at det har oppstått en feil; det er ikke lengere et like antall enere i denne sirkelen.

Det samme gjelder sirkel C, men sirkel B er riktig. Siden den skraverte ruten er den eneste som både A og C deler og ikke B, må det være dette databitet som er feil.

Under er det en tabell som viser kostnad ved bruk av feilkorrigerende kode.

Enkeltfeil-korrigering			Enkeltfeil-korrigering + dobbeltefeil-deteksjon	
# databit	# sjekkbit	% ekstra	# sjekkbit	% ekstra
8	4	50,00	5	62,50
16	5	31,25	6	37,50
32	6	18,75	7	21,88
64	7	10,94	8	12,50
128	8	6,25	9	7,03
256	9	3,52	10	3,91

9 - Sekundærlageret

Sekundærlageret (også kalt eksternt lageret) er nivået lengst borte fra prosessoren i lagerhierarkiet. Typiske egenskaper ved sekundærlageret er:

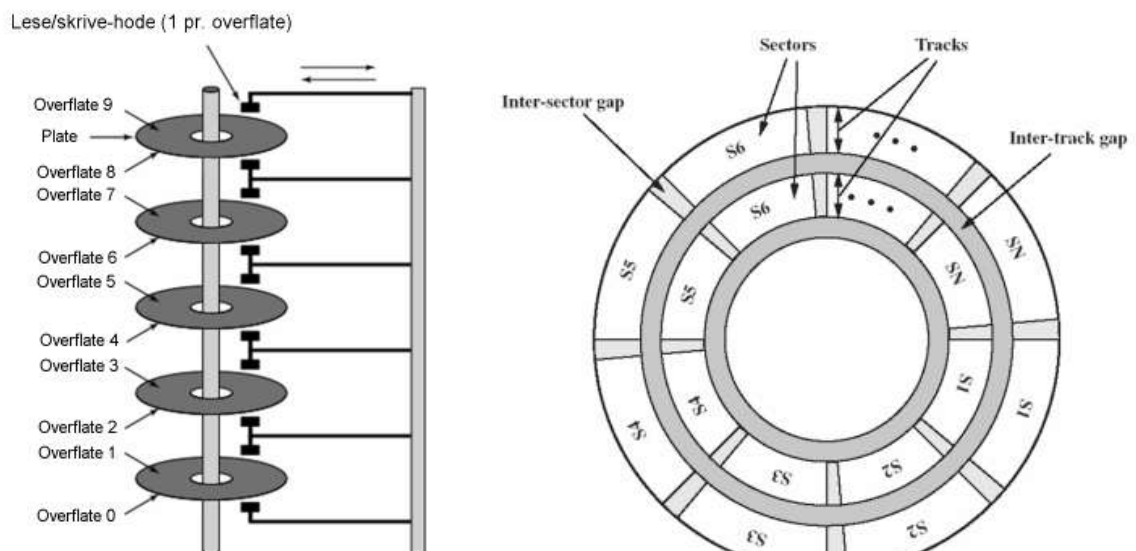
- Trenger operativsystemstøtte. Det er programvare som sørger for å kopiere data mellom sekundærlager og internlager.
- Data blir bevart selv om man skur av strømmen.
- Har stor lagringskapasitet.
- Har flyttbar informasjon, det vil si at man kan fysisk flytte lagringsenheten fra en maskin til en annen og informasjonen vil være bevart.
- Er tregt i forhold til resten av datamaskinen.
- Er relativt billig.
- Blir brukt til langtidslagring.

Vi har ulike typer sekundærlager. Her er noen

- Magnetiske medier, platelager (disk):
 - Harddisk
 - Diskett
 - Zip disk
- Diverse typer magnetbånd (tape)
- Optisk Disk
 - CD (CD-ROM, CD-R, CD-RW)
 - DVD (DVD-ROM, DVD+R, DVD+RW, DVD-R, DVD-RW, DVD-RAM)

Fysisk sett ligger ikke sekundærlageret på hovedkortet men i egne avtakbare enheter som kobles til hovedkortet (eller til kontrollerkort tilkoblet hovedkortet).

9.1 - Magnetplate



Hvordan en harddisk ser ut inni

Diskutlegg

En harddisk består av en sylinder med plater. Hver plate består av flere spor (tracks). Det er like mye data per spor. Mellom hvert spor er det mellomrom (gaps) for å unngå feil ved innstilling av lese/skrive-hodet, og for å unngå interferens med det magnetiske feltet. Hvert spor er delt inn i sektorer som inneholder en gitt mengde data (ofte 512 bytes). I tillegg kan vi dele harddisken inn i logiske blokker som er den minste adresserbare enheten på harddisken sett fra operativsystemet. En blokk må inneholde et helt antall sektorer (ofte bare en enkelt sektor).

Mellom platene er det lese- og skrivehoder. Man kan ha et enkelt skrive/lese hode eller de kan være separerte. Ved lesing og skriving er hodet stasjonært og platene roterer.

Bit nær sentrum av den roterende platen passerer et gitt punkt saktere enn bit ytterst på platen. Platen roterer med en konstant vinkelhastighet (CAV) som gjør at sektorer ytterst er fysisk større enn sektorer innerst. Fordelen med CAV er at individuelle blokker kan bli direkte adressert ved spor og sektorer. Ulempen med CAV er at mengden data som kan bli lagret på de ytterste sporene er den samme mengden som det er mulig å lagre på de innerste, og tar dermed opp mer areal enn nødvendig.

For å øke tettheten bruker moderne harddisker «multiple sone recording», som vil si at overflaten blir delt inn i et visst antall soner. De ytterste sonene lagrer mer data per spor enn de innerste. Innen en sone er antall bit pr spor konstant.

Når man skal søke etter en gitt blokk finner man først det rette sporet, så venter man på at den rette sektoren skal komme forbi. Kort sagt:

$$\text{Aksesstid} = \text{Søketid (flytte hodet til riktig spor)} + \text{Rotasjonsforsinkelse} + \text{Overføringstid (tiden det tar å overføre dataene)}$$

Man er ikke alltid konsekvent vet definisjonen av aksesstid. Mange lærebøker definerer også denne uten å ta med overføringstid. Vi velger å ta med overføringstiden også da det virker ulogisk å ikke gjøre det.

- Aksesstid: Tid det tar å finne frem til data
- Søketid: Tid det tar å flytte lese/skrive-hodet over til riktig spor
- Rotasjonsforsinkelse: Tid det tar for platen å rotere slik at riktig sektor befinner seg under lese/skrive-hodet

Når man har funnet begynnelsen av sektoren man er ute etter kan man begynne å lese eller skrive.

9.1.1 – RAID

RAID står for «Redundant Array of Independent Disks». Opprinnelig sto det for «Redundant Array of Inexpensive Disks», fordi RAID var en måte å benytte billige og upålitelige harddisker til kritiske oppgaver. Nå er alle disker billige og man syntes derfor at RAID trengte en ny betydning.

RAID er å sette sammen flere uavhengige harddisker slik at de oppfattes som en

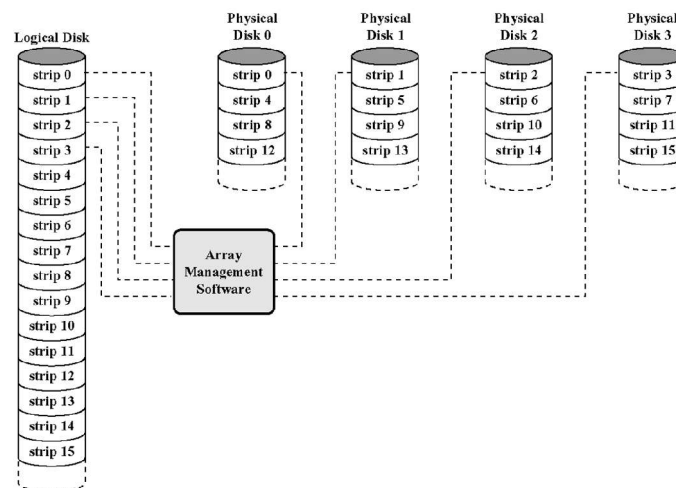
enkelt harddisk for operativsystemet. Det er flere grunner til at man ønsker dette. Man kan ønske å øke hastigheten på dataoverføring ved å spre data over flere harddisker, eller man kan ønske å lagre samme informasjon på flere harddisker for å kunne gjenopprette data hvis en harddisk går i stykker.

RAID har disse karakteristikene:

- Flere fysiske harddisker koblet sammen slik at de oppfører seg som en enkelt harddisk
- Data er spredt ut over alle de fysiske harddiskene
- Noe av diskkapasiteten (kan) benyttes til å lagre paritetsinformasjon slik at data kan gjenopprettes i tilfelle en eller flere diskler slutter å fungere

Noen RAID-systemer kan gjenopprette data også mens systemet er i drift. Dette gjør RAID mer attraktivt enn (bare) å benytte sikkerhetskopier.

Et eksempel på RAID er vist i figuren under. Det finnes fire fysiske harddisker, men operativsystemet ser bare en stor logisk enhet.



9.1.2 – RAID-nivå

Det finnes flere typer RAID med forskjellige egenskaper.

RAID-nivå 0

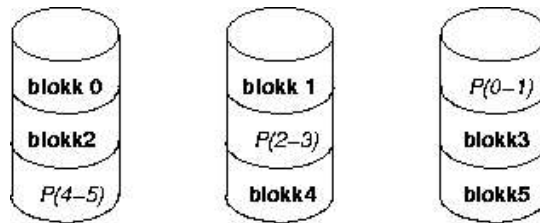
Dette er ikke et «ekte» RAID oppsett fordi man ikke har redundans (=en eller annen form for duplisering av data eller sjekksum). Man kan derfor ikke gjenopprette data dersom en av harddiskene går i stykker. I RAID-0 er data organisert akkurat som i tegningen over. Ved å spre data utover slik, kan overføringshastigheten øke betraktelig, fordi alle harddiskene kan lese ut data i parallell.

RAID-nivå 1-6

Benytter redundante diskler til å lagre data på slik at man kan gjenopprette data i tilfelle en av harddiskene skulle gå i stykker. Hvert RAID-nivå løser oppgaven på sin egen måte, og det finnes fordeler og ulemper med hvert nivå. Som eksempel forklarer vi RAID-5, som er det som benyttes ved IDI, NTNU.

RAID-nivå 5

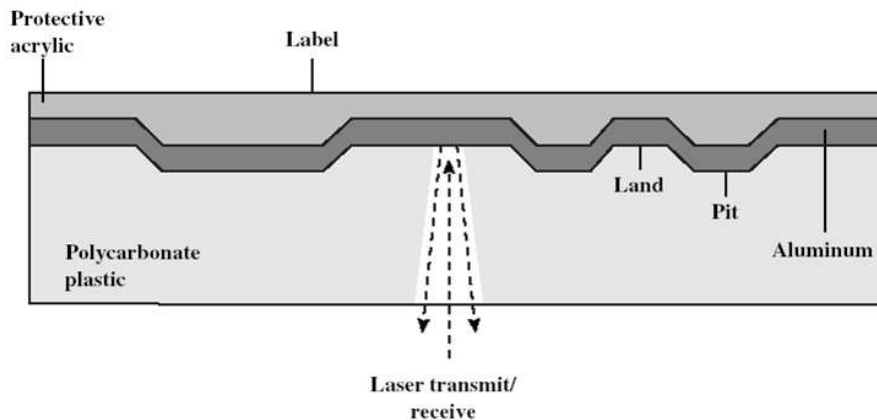
RAID-5 er et oppsett av harddisker laget for å tåle et stort antall I/O-forespørsler, samtidig som den klarer å gjenopprette data dersom en harddisk feiler.



Et RAID-5-oppsett er vist i figuren over. Datablokker er spredt utover de tre harddiskene. Harddiskene er uavhengige slik at man kan lese både fra blokk 0 og blokk 1 i parallell. I tillegg finnes det en paritetsblokk per harddisk som sørger for at man kan gjenopprette data dersom en av harddiskene går i stykker.

9.2 - Optisk plate

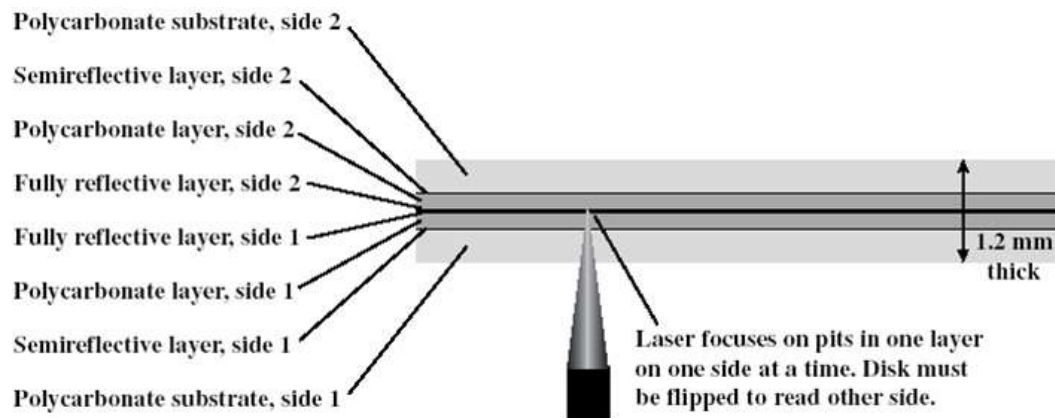
9.2.1 - CD-ROM



På en CD-ROM (compact disk read-only memory) er data lagret som «groper» (pits) i en aluminiumsplate og blir lest av med en laserstråle. Data blir lest med konstant lineær hastighet (CLV): platen roterer saktere ved aksess nær kanten av CD-en i forhold til nær sentrum slik at platearealet som passerer forbi lesehodet er lik både ved sentrum og ved kanten. I tillegg er data lagret i en spiral på platen, og ikke i flere separate spor slik som på en magnetplate.

En CD-ROM-plate har en kapasitet på 700MB.

9.2.2 – DVD



DVD (Digital Versatile Disk) ligner på CD-ROM både av utseende og oppførsel. Men den lagrer data tettere enn CD-ROM, og har mulighet til å lagre flere lag med data, på begge sider. DVD har kapasitet fra 4,7 GB (1 side, 1 lag) til 17 GB (2 sider, 2 lag). Aksesstiden er 150–250 ms.

10 - Innvevde (innebygde) systemer

Innvevde systemer (engelsk: Embedded Systems) er kort forklart datamaskiner som gjør en spesifikk oppgave og er del i et større system.

Ta for eksempel en vaskemaskin. Denne inneholder en liten datamaskin som sørger for å kjøre vaskeprogrammet når du trykker på startknappen. Denne datamaskinen er spesiallaget for denne ene oppgaven og kan ikke brukes til noe annet. Dette i motsetning til en generell datamaskin av typen du sitter foran nå.

Innvevde systemer finnes overalt og er faktisk den aller vanligste formen for datamaskiner, selv om man ikke alltid forbinder de med det. Her er en liste over typiske, hverdagslige ting som ofte inneholder prosessorer:

- Vaskemaskiner, tørketromler
- Klokker
- Minibanker
- Mobiltelefoner
- Biler (ABS-bremsesystem, innsprøytningssystem, etc.)
- Fjernsynsapparater
- Termostater

Innvevde systemer spesiallages for en spesiell oppgave. Design av slike systemer vil som oftest bestå i utvikling av både maskinvare og programvare. Dette gjør innvevde systemer til en veldig heterogen gruppe med datamaskiner.

10.1 – Designhensyn

Utvikling av innvevde systemer har ofte helt andre designkrav enn for andre typer datamaskiner.

10.1.1 – Kostnad

Mange av disse systemene er hverdagslige ting der kostnad spiller en vesentlig rolle. Totale produksjonskostnader bør derfor være så små som mulig.

10.1.2 – Strømforbruk

Portable enheter (for eksempel mobiltelefoner) må trekke strøm fra et batteri. Man ønsker maksimal batterilevetid, og strømforbruket til enheten må derfor være så liten som mulig.

10.1.3 – Sanntid

Mange innvevde systemer har sanntidskrav. Et sanntidssystem er et system som garantert svarer innen det har gått en viss tid. En vanlig PC har ikke slike garantier, og man vil kunne oppleve at maskinen «henger» noen sekunder fordi den jobber tungt. Et sanntidssystem garanterer at den oppfyller visse krav om responsivitet. For eksempel ønsker man ikke at stabilisatorsystemet på et jagerfly plutselig «henger» noen sekunder. Det kan være nok til å sende flyet i bakken.

10.1.4 – Levetid

Slike systemer er ofte i bruk i lang tid, og må derfor bygges for å vare. Systemet

må være driftssikkert. Det er ikke utenkelig med levetid på 20 år eller mer.

10.1.5 – Størrelse og vekt

Fysisk størrelse er viktig. for eksempel må en mobiltelefon være så liten og lett som mulig.

10.1.6 – Vanskelig miljø

Av og til skal systemet inn i et miljø som stiller spesielle krav til elektronikken. Dette kan være ekstrem varme, kulde, stråling og risting. Blant annet satellitter må ta hensyn til slike ting.

10.1.7 – Hastighet

Hastighet er ofte ikke viktig for et innvevd system. En vaskemaskin trenger ikke all verdens med datakraft for å kjøre vaskeprogrammet.

10.2 – Komponenter – Mikrokontroller

Den viktigste komponenten i et innvevd system er som i alle andre datamaskiner: prosessoren. Men det er ikke så vanlig å benytte en prosessor av den typen som sitter i en PC. I stedet bruker man en mikrokontroller. Dette er en liten mikrobrikke som inneholder prosessor men som i tillegg inneholder de andre viktigste komponentene en datamaskin trenger: I/O-moduler, RAM, ROM (og ofte Flash), timere etc. Man kan si at en mikrokontroller er en hel datamaskin på én brikke. Det finnes en lang rekke forskjellige mikrokontrollere på markedet, som ikke er like ensrettet som prosessormarkedet i PC-industrien der Intel-arkitekturen dominerer.

Fordelene med å benytte en mikrokontroller fremfor en vanlig prosessor:

- Mindre fysisk; én brikke erstatter mange
- Bruker mindre strøm
- Enkelt, man har det meste man trenger på én brikke noe som gjør det enkelt å bygge et system rundt
- Laget spesielt med tanke på enkel utvikling av innvevde systemer

En typisk mikrokontroller er svært mye mindre kraftig enn en vanlig PC. Prosessorene er ofte på 8-bit. Klokkefrekvenser er ofte fra 1MHz og oppover. RAM-størrelse kan være fra 1kB og oppover (ingen cache), og ROM er ofte den største delen av det adresserbare lageret.

Dette er nesten ufattelig små verdier for en som er vant til dagens PC-er, men er nok for de aller fleste dataprosesseringsoppgaver i innvevde systemer. Det er et morsomt faktum at selv i dag er det slike små 8-bits datamaskiner som fremdeles selger mest (på grunn av alle innvevde systemer), selv om de færreste kjenner til noe annet enn de kraftige 32-bits PC-ene som benyttes i dag.

I tillegg til mikrokontrolleren vil man ofte trenge en del komponenter for akkurat den spesielle oppgaven dette systemet skal utføre. Dette kan være sensorer, motorer, nettverkstilkobling etc.

10.3 – Utvikling

Utvikling av et innvevd system er en spennende oppgave fordi det involverer mange forskjellige ting. Man skal bygge maskinvare som er i stand til å løse et gitt problem, for deretter å programmere den samme maskinvaren. Viktige valg må tas om hva som skal gjøres av maskinvare og hva som skal gjøres av programvare i en mikrokontroller (HW/SW-kodesign).

Utvikling av programvare for et innvevd system er ikke som på en vanlig PC. Programvaren skal kjøre på et lite kretskort ofte uten skjerm, tastatur, operativsystem og andre nyttige ting. Man skriver derfor ikke programvaren på maskinen den skal kjøres på. I stedet skriver man programvaren på en vanlig PC og overfører programvaren til kortet før kjøring. Avlusning av programvare skjer ofte ved hjelp av en såkalt ICE (In Circuit Emulator) som erstatter mikrokontrolleren med en emulatkrets som gir brukeren mulighet til å styre og kontrollere alt vha. en PC.

11 - Oppsummering

11.1 – Primærkunnskaper

Vi bruker vanligvis titallsystemet når vi regner. Men det finnes mange andre tallsystemer i bruk. Et tallsystem defineres av dens base. Viktigst her er totallsystemet (binære tall) der man har base 2, det vil si man benytter bare tallene 0 og 1. Totallsystemet er grunnlaget for all regning i en digital datamaskin. I dag benytter vi som oftest toerskomplement til å representere negative binære tall.

Sentrale lagringsenheter i et digitalt system er bit, byte og dataord. Kilo i datasammenheng er som oftest er 1024 og ikke 1000.

Flyttall er «kommatal» med eksponent. Disse representeres ofte ved hjelp av IEEE's flyttallsstandard: fortegnsbitt, eksponent (med forskyvning) og signifikand.

Digitale systemer bygges opp av logiske porter. Noen elementære komponenter laget av porter er multipleksere, dekodere og adderere. Synkrone kretser benytter et klokkesignal til å synkronisere alle hendelser med. En viktig synkron komponent er en vippe.

Her er en liste over viktige stikkord:

- Tallsystemer
 - binære tallene
 - toers komplement
 - omregning mellom tallsystemer
 - bits og bytes, ord
 - flyttall
- Digitalteknikk
 - porter
 - multiplekser
 - adderer
 - dekodere
 - klokkesignal, synkrone kretser, vipper

11.2 – Prosessoren

Prosessoren slik den er i dag følger en modell beskrevet av von Neumann i 1945. Dette vil i hovedsak si at datalageret inneholder både instruksjoner og data. Instruksjoner mates inn i prosessoren som da utfører en operasjon på data.

Ofte deles prosessoren inn i en styreenhet og en utførende enhet. Den utførende enheten gjør selve dataprosesseringen, mens styreenheten dekoder instruksjoner og setter opp utførende enhet til å gjøre riktig operasjon.

Utførende enhet består av en registerblokk der midlertidige data lagres og en aritmetisk-logisk enhet (ALU) som utfører aritmetiske og logiske operasjoner. Den utførende enheten styres av et styreord som settes opp av styreenheten. Styreordet inneholder alle nødvendige styresignaler.

I tillegg til registerblokken finnes det noen spesialregistre. De viktigste er programtelleren (PC), instruksjonsregisteret (IR) og statusregisteret (SR). PC inneholder til enhver tid adressen til neste instruksjon som skal utføres. IR inneholder instruksjonen som er under utførelse. SR inneholder informasjon om tilstanden til prosessoren, informasjon som benyttes blant annet til betingete hopp.

Instruksjonsutføring deles inn i flere steg; det som kalles instruksjonssykel. Instruksjonssykel kan implementeres ved hjelp av en tilstandsmaskin eller ved hjelp av et samlebånd. Samlebånd lar flere instruksjoner kjøre samtidig med hverandre men i hver sitt steg i instruksjonssykel. Dette øker hastigheten. Problemer er hoppinstruksjoner og dataavhengigheter.

Her er en liste over viktige stikkord:

- von Neumann-arkitektur
- mikrooperasjoner
- prosessoren, introduksjon
- utførende enhet
 - ALU
 - registerblokk
 - styreord
- PC, SR, IR
- styreenhet
- instruksjonssykel
- samlebånd
 - dataavhengigheter
 - håndtering av forgreninger (hopp)
- RISC/CISC
- big/little endian

11.3 – Instruksjoner

Prosessoren er avhengig av å få tilført en jevn strøm av instruksjoner slik at den vet hva den skal gjøre. Det er prosessorarkitekturen som dikterer hvilke instruksjoner som finnes.

Instruksjonssettet består av alle instruksjoner en gitt prosessor kan utføre. Det finnes mange typer instruksjoner; dataoverføring, aritmetiske, logiske, konvertering, I/O, systemkontroll og kontrollflyt.

Hver instruksjon tar et antall operander, ofte fra ingen operander til tre operander. Stakkmaskininstruksjoner tar ofte ingen operander, stakken er implisitt operandkilden. Load/Store-maskininstruksjoner tar ofte tre operander; destinasjon, argument A og argument B.

Assemblyprogrammering er programmering på laveste nivå. Her spesifiseres direkte hvilke instruksjoner prosessoren skal utføre.

Hver operand angir hvor data til en instruksjon befinner seg. Dette kan angis på flere forskjellige måter og kalles operandens adresseringsmodus. Dere har lært disse: immediate, direkte, indirekte, register, indirekte register, displacement (og indeksering) og stakk.

Instruksjoner består av et gitt antall bit; et instruksjonsord. Instruksjonsordet består av en rekke felter som definerer instruksjonen. De viktigste er opkode som forteller hva denne instruksjonen skal gjøre (JUMP, ADD etc.) og operandfelte som angir hvor operandene finnes.

Her er en liste over viktige stikkord:

- instruksjonssett
- assembly
- antall operander
- load/store-maskin
- adresseringsmodi
- instruksjonsformat

11.4 - I/O

I/O er kommunikasjon mellom prosessoren og andre enheter (bortsett fra lageraksesser som ikke defineres som I/O). I/O gjøres enten ved hjelp av egne I/O-instruksjoner i prosessoren eller ved at I/O-enheter får en del av adresserommet (lageravbildet I/O).

Prosessoren kommuniserer nesten aldri direkte med ønsket I/O-enhet. I stedet kommuniserer den med en I/O-kontroller som sitter mellom prosessoren og I/O-enheten. I/O-kontrolleren gir prosessoren et enkelt grensesnitt mot I/O-enheten.

Kommunikasjon med I/O-kontroller kan enten være programmert (polling), avbruddsdrevet eller DMA (Direct Memory Access).

Busser er kommunikasjonsveier mellom enheter. De består ofte av datalinjer, adresselinjer og styrelinjer. Det finnes ofte mange forskjellige busser med forskjellig hastighet i en datamaskin. Bussarkitekturen i en datamaskin settes opp i et hierarki, akkurat som lageret som settes opp i et hierarki fordi forskjellige lagertyper har forskjellig hastighet.

Busser har ofte mange enheter tilknyttet seg. Bare en enhet kan skrive til bussen av gangen. For å styre hvilken enhet som skriver kan en av enhetene være «herre» som styrer alt som skjer, eller man kan benytte en form for arbitrering; en algoritme (ofte styrt av en arbitreringsenhet) som sørger for å velge ut hvem som får skrive til bussen.

Busser kan enten være synkrone (koordinert av en klokke) eller asynkrone (koordinert med et sett av «håndtrykk»-signaler).

Her er en liste over viktige stikkord:

- Lageravbildet I/O vs. egne I/O-instruksjoner
- I/O-kontroller
- programmert I/O
- avbruddsdrevet I/O
- DMA
- busser
 - datalinjer, adresselinjer, styrelinjer
 - busshierarki
 - bussarbitrering
 - synkron vs. asynkron buss

11.5 – Lagerhierarkiet

Alle datamaskiner trenger et lager til data. Konseptuelt sett finnes det bare ett stort lager (i tillegg til registrene). Fysisk er det derimot ordnet på en annen måte.

Et lager kan klassifiseres etter følgende kriterier: Størrelse (lagerkapasitet), hastighet og kostnad. Generelt kan man si at dersom lagerkapasiteten øker går hastigheten ned. utfordringen ligger derfor i å lage et stort og hurtig lager som ikke koster for mye.

En observasjon ble gjort: Lageraksesser følger noe som kalles «lokalitetsprinsippet». Det vil si at data som nylig er brukt sannsynligvis vil bli brukt igjen snart. Prosessoren jobber altså på et lite sett med data av gangen. Dette kan utnyttes ved å benytte et lagerhierarki. Et stort og tregt lager danner grunnlaget. Men de til enhver tid mest brukte data ligger også lagret i et mindre, hurtigere lager. Slik kan man lage et hierarki der vi har raskt og lite lager på toppen og tregt og stort lager på bunnen. Dette er vist i figuren på toppen. Resultatet blir et lager som er nesten like raskt som lageret øverst i hierarkiet men samtidig like stort som lageret nederst i hierarkiet.

Her er en liste over viktige stikkord:

- lagerhierarki
- hastighet vs. størrelse vs. pris
- lokalitetsprinsippet

11.6 – Hurtigbuffer

Hurtigbufferet er den delen av lagerhierarkiet som ligger mellom registrene på toppen og primærlageret. Hurtigbufferet er altså mindre og raskere enn primærlageret og brukes til å øke hastigheten på lageraksesser.

Det finnes ofte to hurtigbuffer i lagerhierarkiet; hurtigbuffer nivå 1 og nivå 2. Nivå 1 er det minste og raskeste og bufrer data fra nivå 2. Nivå 2 er litt større og tregere og bufrer data fra primærlageret.

Hurtigbufferet deles inn i linjer som hver inneholder en merkelapp som sier hvilken adresse som er bufret i denne linjen og et dataområde med data fra denne adressen. I tillegg finnes blant annet informasjon om data er gyldig eller ikke.

Når prosessoren gjør en lageraksess til en adresse som ikke er bufret vil hurtigbufferet hente data fra nivået under i lagerhierarkiet. Dette vil den plassere i hurtigbufferet, enten ved hjelp av direkte, assosiativ eller sett-assosiativ avbildning.

Brukes en form for assosiativ avbildning må det finnes en måte å bestemme hvilken linje i hurtigbufferen man skal skifte ut når nye data skal bufres. Dette gjøres med en erstatningsalgoritme. Denne kan være LFU (Least Frequently Used), FIFO (First In First Out), LRU (Least Recently Used) eller tilfeldig.

Hurtigbufferet inneholder en kopi av data fra lavere nivå i lagerhierarkiet. Derfor må data som oppdateres i hurtigbufferet også skrives tilbake til de lavere nivå for at de ikke skal gå tapt. Det finnes to tilbakeskrivingsstrategier: Gjennomskrivning (data skrives tilbake med en gang de oppdateres i hurtigbufferet) og utsatt tilbakeskriving (data skrives kun tilbake når hurtigbufferlinjen skal erstattes med en annen adresse).

Her er en liste over viktige stikkord:

- avbildningsfunksjon
 - direkteavbildning
 - assosiativ avbildning
 - sett-assosiativ avbildning
- erstatningsalgoritmer
 - LRU
 - FIFO
 - LFU
 - tilfeldig
- skrivestrategi
 - gjennomskrivning
 - utsatt tilbakeskriving

11.7 - Primærlageret

Primærlageret er den nederste delen av lagerhierarkiet som har automatisk (maskinvarestyrt) støtte. Det er primærlageret prosessoren «ser» når den gjør lageraksesser. Hurtigbufferet som prosessoren egentlig gjør en aksess til fungerer transparent for prosessoren slik at prosessoren ikke trenger å vite at det ikke er det egentlige primærlageret den gjør aksesser til.

Primærlageret består hovedsaklig av RAM. I noen systemer kan det også være naturlig å definere ROM som en del av primærlageret.

RAM er et flyktig lager som kan skrives til og leses fra. Det finnes hovedsaklig to typer: Statisk og dynamisk RAM. Statisk RAM tar størst plass (flesteplutt antall transistorer) men er raskest. Dynamisk RAM tar mindre plass, men benytter kondensatorer til å lagre verdier i. Den må derfor oppfriskes jenvlig for ikke å miste data.

ROM er et lager man bare kan lese fra (i normal operasjon). Det finnes mange typer: ROM, PROM, EPROM, EEPROM og Flash.

Her er en liste over viktige stikkord:

- RAM
 - Statisk RAM (SRAM)
 - Dynamisk RAM (DRAM)
- ROM
- adressring
- feilkorrigering

11.8 - Sekundærlageret

Sekundærlageret er I/O-enheter som tar seg av lagring av data. Det tilhører de nederste delene av lagerhierarkiet. Siden sekundærlageret er I/O-enheter vil det i motsetning til primærlageret (og nivåene over det) kreve programvarestøtte for å kunne brukes.

Sekundærlageret består ofte av magnetplater (harddisker og disketter). Harddisker kan settes opp i en konfigurasjon kalt RAID for å øke tilgjengelighet og hastighet.

Harddisker består av flere magnetplater. Hver plate deles inn i spor og sektorer. Hver plateoverflate har et lese/skrive-hode som kan beveges fra et spor til et annet.

$$\text{Aksesstid} = \text{Søketid (flytte hodet til riktig spor)} + \text{Rotasjonsforsinkelse} + \text{Overføringstid (tiden det tar å overføre dataene)}$$

Man er ikke alltid konsekvent vet definisjonen av aksesstid. Mange lærebøker definerer også denne uten å ta med overføringstid. Vi velger å ta med overføringstiden også da det virker ulogisk å ikke gjøre det.

Det finnes også en del former for optiske medier (CD og DVD).

Her er en liste over viktige stikkord:

- magnetplate
 - RAID
- optisk plate
- konstant vinkelhastighet (CAV) vs. konstant lineær hastighet (CLV)
- multiple zone recording (MZR)

11.9 - Innvevde systemer

Innvevde systemer er små datamaskiner spesialbygde for ett spesielt formål. Eksempler er styresystemene i vaskemaskiner og mobiltelefoner.

Innvevde systemer benytter ofte en mikrokontroller. Dette er en hel datamaskin på en enkelt brikke. Dette gjøres for å få systemet fysisk lite, strømeffektivt og billig. Ofte kan hastighet og datakraft nedprioriteres til fordel for pris, liten størrelse og vekt, lavt strømforbruk og andre faktorer viktige for det spesielle miljøet systemet skal benyttes i.