

Datamaskinarkitektur og binær representasjon (Foreløpig utgave)



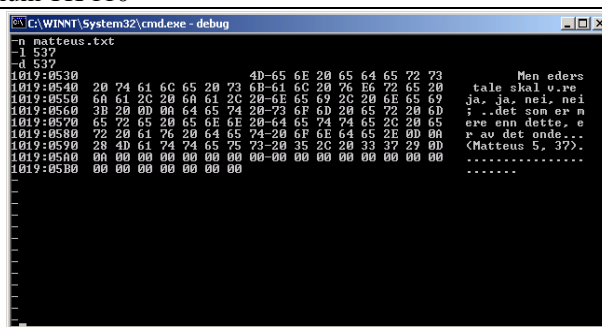
Innholdsfortegnelse

Innholdsfortegnelse	2
Forord	3
Innledning	4
Funksjonsorientert modell	5
Brukerprogramnivået	5
Kompilatornivået	5
Operativsystemnivået	6
Instruksjonsnivået	6
Mikroinstruksjonsnivået	6
Kretsnivået	7
Maskinvareorientert modell	7
Portnivået	7
Registernivået	9
Busser	10
Prosessornivået	11
Sentralenheten (CPU'en)	11
Representasjon av data i datamaskinen	13
ASCII-koder	14
Representasjon av heltall	15
BCD	16
Flyttall	17
Addisjon av binære tall	18
Negative tall	20
Konvertering - fra desimal til binær	21
Heksadesimale tall	22
Hvordan instruksjoner utføres	23
Instruksjonssettet	25
Assemblyprogrammering og assemblerens virkemåte	28
Assemblerens virkemåte	30
Første gjennomløp	31
Andre gjennomløp	32
Operativsystemnivået	33
Hensikten med operativsystemet	33
Historikk	34
Batch-systemer	34
Spooling	34
Multiprogrammerte batch-systemer	35
Tidsdelingssystemer/multitasking og interaktive systemer	35
Personlige datamaskiner og arbeidsstasjoner	36
Parallele systemer	36
Distribuerte systemer	37
Sanntidssystemer	37

Forord

*Men eders tale skal være ja, ja, nei, nei;
det som er mere enn dette, er av det
onde.*

(Matteus 5, 37)



Figur 1: Minnedump av sitatet til venstre

Dette kompendiet gir en kortfattet oversikt over hvordan programvare

(software) og maskinvare (hardware) samvirker. For å få en dypere forståelse av datamaskinen i arbeid må vi lære mer om prinsippene bak virkemåten til en CPU. Det er CPU'en som er "hjernen" i systemet. Moderne CPU'er er ekstremt komplisert teknologi så det ville ikke være formålstjenlig å sette oss inn i alle detaljer og varianter. Samtidig er det slik at skal man ha en sjanse til å forstå hvordan maskinen virker, og hvorfor den nå og da ikke virker, så må man kjenne de viktigste prinsippene bak hvordan den er konstruert.

Vi starter med å ta en ny titt på *Von Neumann-modellen*. Denne modellen er bar en grunnleggende oppskrift på en datamaskin. Deretter innfører vi en lagdeling av programvare og maskinvare. Med utgangspunkt i den oversikten lagdelingen gir oss skal vi:

- 1) først få en rask oversikt over funksjonene til de ulike nivåene (ovenfra og ned).
- 2) Deretter begynner vi fra bunnen av med de basale maskinvarekomponentene og arbeider oss oppover i lagene igjen.

Vi vil altså først dykke raskt ned til bunnen av maskinen for å få oversikt. Deretter beveger vi oss opp igjen i et mer bedaglig tempo for å få innsikt.

Et viktig tema er *representasjon* og *koding* av data. Etter å ha lest gjennom denne delen av kompendiet *to* ganger bør du ha forstått hvordan data og instruksjoner kan representeres med bits, som igjen kan representeres med høye og lave spenninger, som kan prosesseres i logikkretser og lagres i RAM og på harddisk. Dette er fundamentet for elektronisk databehandling og moderne informasjonsteknologi.

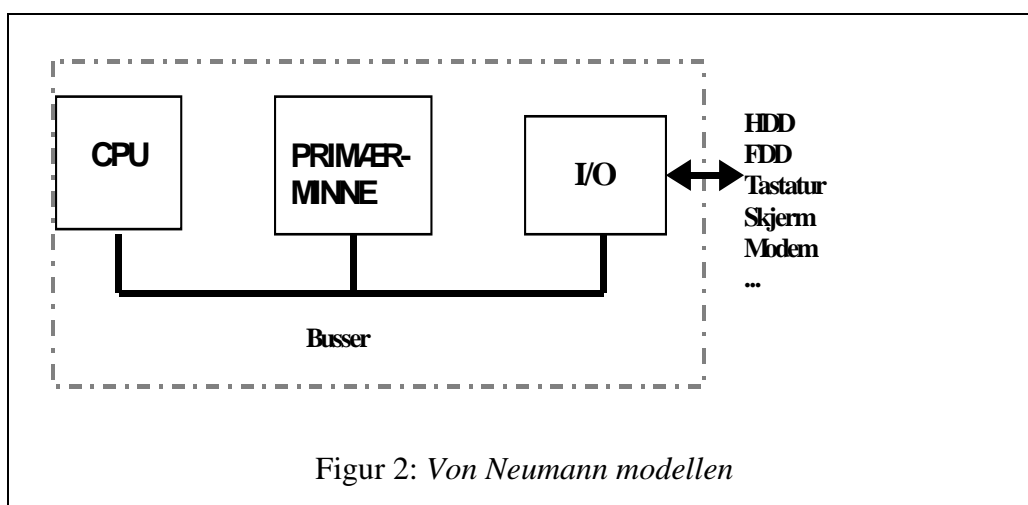
Mange synes at dette stoffet er vanskelig. En del av problemet er at vi er nødt til å benytte innarbeidet fagspråk og sjargong. Det er noe av det du må lære for å kunne arbeide som dataproff. Et annet problem er at det kan være vanskelig å se relevansen av logikkporter, registre, bussbredde og andre maskinvaredetaljer for praktisk bruk av datamaskinen.

Første versjon av av kompendiet var forfattet av Lasse Berntzen og Torunn Gjester. Denne versjonen er noe oppdatert og omskrevet av Bjørn O. Listog. Kommentarer og frustrasjoner kan sendes bjorn.listog@nith.no.

Innledning

Figuren under er en skjematisk framstilling av et system for dataprosessering. Det er en grov forenkling av hvilke komponenter som faktisk er til stede i en moderne PC. Denne prinsippskissen kalles Von Neumann modellen. Poenget er å få oversikt over hovedfunksjonene som må ivaretas av systemet. Her grupperes dette i tre ulike hovedblokker: CPU, primærminne og I/O-enhet. I/O-enheten styrer kommunikasjonen med ulike typer periferiutstyr. CPU'en henter og dekode instruksjoner og operer på data. Både instruksjonene til CPU'en og data må lagres i primærminnet (RAM, ROM). Instruksjonene utføres sekvensielt, dvs i den rekkefølgen de er lagret i minnet. Primærminnet og I/O-enhetene er forbundet med CPU'en ved hjelp av busser som gjør det mulig for CPU'en å adressere bestemte minneadresser og bestemte periferienheter gjennom I/O-enheten.

Det har vært designet mange alternativer til Von Neumann modellen med sin strengt



Figur 2: Von Neumann modellen

sekvensielle programutførelse, men de aller fleste datasystemer i våre dager følger framdeles denne femti år gamle oppskriften.

Datamaskinen kan beskrives på flere ulike måter. Von Neumann modellen er kun en grovskisse. En enkel og oversiktlig tilnærming er å dele opp i nivåer.

Først deler vi opp i maskinvare (hardware) og programvare (software). Løslig defineres disse som: *"Maskinvare er alt som kan ødelegges med en loddebolt. Programvare er alt som kræsjer."* Mer presist så består maskinvaren av ting man kan ta på: integrerte elektroniske kretser, kabler, strømforsyninger, minne, printere, osv. Programvare består på sin side av *algoritmen* (de detaljerte instruksjonene for hvordan en oppgave skal løses), datastrukturene algoritmen manipulerer, og den elektroniske *representasjonen* (gjengivelsen) av algoritmen: selve det kjørbare programmet som er lagret som høye og lave spenninger.

Det er greit å være klar over at dette skillet ikke er fullt så enkelt som det høres ut. Alle algoritmer *kan* nemlig "hardkodes", dvs bygges som maskinvare. Og det meste av maskinvaren *kan* programmeres. Vi sier derfor at maskinvare og programvare er logisk ekvivalente. Når man designer en ny datamaskin må man derfor velge hvilken funksjonalitet som skal tilbys av maskinvaren og hva som skal programmeres. Avgjørende faktorer vil være pris, hastighet, fleksibilitet, pålitelighet osv.

Von Neumann modellen gir oss altså en første oversikt over hvordan maskinvaren er bygd opp. Ingen vettuge mennesker kjøper en PC for maskinvarens skyld. Det er muligheten for å kjøre brukerprogrammer (applikasjoner) som interesserer oss. Så hvordan skal vi skaffe oss oversikt?

Det er nyttig å lage seg en lagdelt modell av systemet. Modellene kan ta utgangspunkt i *hvordan maskinen fungerer* eller *hvordan den er bygget opp*.

Funksjonsorientert modell

En vanlig bruker arbeider oftest ut fra en forestilling om tre nivåer: brukerprogram-, operativsystem- og maskinvare-nivå. Vi vil foreta en litt mer finmasket inndeling for å få fram hvordan en programmerer kan se på systemet.

I denne varianten fokuserer vi på programvare og konsentrerer oss om hvordan instruksjoner til maskinvaren utføres. Et eksempel på denne type lagdeling er gitt i figuren under.

Lag 5	Brukerprogramnivå
Lag 4	Kompilatornivå
Lag 3	Operativsystemnivå
Lag 2	Instruksjonsnivå
(Lag 1	Mikroinstruksjonsnivå)
Lag 0	Digitalt krets nivå

Tabell 1: Funksjonsorientert lagdeling

De tre øverste lagene er definitivt programvare, de to nederste er mest maskinvare, mens instruksjonsnivået "limer" sammen maskinvare og programvare. Operativsystem- og kompilatornivåene inneholder ulike typer systemprogramvare.

Brukerprogramnivået

Brukerprogramnivået skulle allerede være kjent. Her opplever vi datamaskinen som en problemløser som leser inn kommandoer og data fra fil eller tastatur, utfører kommandoer, og skriver resultatet på skjerm, skriver eller fil. På dette nivået er vi ikke opptatt av hva som egentlig skjer inne i maskinen. Spill, tegneprogram, tekstbehandlingsprogram, vevlesere o.l. er på dette nivået. Disse programmene er avhengige av operativsystemnivået for å kunne kjøres.

Kompilatornivået

En kompilator (f.eks. javac.exe) gir oss mulighet til å angi hva datamaskinen skal gjøre i et språk som er relativt lett å forstå for mennesker. Slike språk kalles for høynivåspråk (HLL – High Level Language). En kompilator oversetter fra et høynivåspråk til maskininstruksjoner.. I høynivåspråk må vi deklareere hvilke konstanter og variabler som benyttes, og vi må spesifisere hva slags operasjoner som skal gjøres på variablene. Et høynivåspråk gjør at vi kan skrive programmer uten å ha detaljkunnskaper om hva som skjer inne i maskinen.

Vi skiller dette ut som et eget lag i modellen vår fordi det er denne typen programvare som gjør oss i stand til å lage såvel brukerprogrammer som operativsystemer.

På samme nivå som kompilatorer vil det også være naturlig å plassere noen andre typer programvare som utfører systemrettede oppgaver. Kommandoskall og annen

programvare (f.eks Explorer/Utforsker) som tar i mot og videreformidler kommandoer til operativsystemet.

Operativsystemnivået

Operativsystemet starter når maskinen startes opp. Det er en samling programmer som bl.a. gir mulighet til å starte andre programmer, og som gir mulighet til å utføre forskjellige systemkommandoer.

Men operativsystemet er langt mer enn dette. Blant annet inneholder operativsystemet en mengde små programmer (rutiner) som f.eks. skriver et tegn på skjermen, leser et tegn fra tastaturet, henter data fra disk, skriver data på disk o.s.v. Når brukerprogrammer skal foreta slike I/O-operasjoner så må det foregå gjennom kall til disse rutinene i operativsystemet.

Når f.eks. et C++ program kompiles så benytter det eksekverbare programmet disse rutinene. På den måten slipper en programmerer å vite alt om hvordan den kompliserte maskinvaren faktisk fungerer. Det holder å vite *hva* kommandoen gjør, *hvordan* overlater man til operativsystemet.

Vi skal komme nærmere inn på operativsystemnivået senere.

Instruksjonsnivået

Alle CPU'er har et sett med grunnleggende instruksjoner som maskinen kan utføre. Dette kalles også maskinkode. Disse instruksjonene hentes fra primærlageret og kommanderer maskinens CPU. Når et program kompiles oversettes programmet til slike instruksjoner.

Ulike mikroprosessorer har ulike instruksjonssett, ulikt antall instruksjoner og ulike måter å bygge dem opp på.

Vi kan gruppere instruksjonene i fem ulike typer:

Aritmetiske/Logiske instruksjoner er instruksjoner som f.eks. legger sammen og trekker fra hverandre tall og foretar logiske operasjoner på data. Disse utføres i CPU'ens ALU.

Dataflyttingsinstruksjoner henter og lagrer data i primærminnet eller flytter data mellom registrene inne i CPU'en.

Kontrolloverføringsinstruksjoner er typisk instruksjoner for å bytte program som kjøres på CPU'en, eller for å hoppe til et annet sted i det programmet som utføres (subrutinekall).

Teste-/Sammenligneinstruksjoner brukes f.eks. til å sjekke om variabler har samme verdi. Denne typen instruksjoner vil typisk brukes til å lage løkker eller til å avgjøre om en kontrolloverføringsinstruksjon skal utføres eller ikke.

I/O-instruksjoner vil sette i gang skrive-/leseoperasjoner til periferiutstyr. (Ikke alle mikroprosessorer har slike instruksjoner.)

I tillegg har de fleste prosessorer en egen instruksjon som ikke gjør noe som helst (NOP) og en instruksjon som avslutter et program.

Alle kjørbare programmer består av en sekvens maskininstruksjoner som utføres i den rekkefølgen de ligger lagret i primærminnet.

Mikroinstruksjonsnivået

Instruksjonsnivået er det laveste nivået vi vanligvis kan programmerere på. For hver instruksjon som utføres, skjer det en rekke aktiviteter inne i maskinen. Disse aktivitetene

styres ofte av såkalte mikroinstruksjoner. Vi kan si at hver enkelt instruksjon starter en sekvens med mikroinstruksjoner.

Det er ikke alle prosessorer som er mikroprogrammerte, det er mulig å utelate dette nivået og la instruksjonene utføres direkte av elektroniske kretser. Mikroprosessorer har dette nivået for konstruktørene å fikse feil og lage kompliserte instruksjoner.

Kretsnivået

Selve datamaskinen består av en rekke digitale kretser. Disse kretsene utfører de faktiske operasjonene i datamaskinen. Kretsene styres av elektriske signaler. Hver mikroinstruksjon inneholder et antall bits som gir disse styringssignalene. For å forstå hvordan dette foregår må vi se nærmere på maskinvarens byggeklosser.

Maskinvareorientert modell

Den funksjonsorienterte lagdelingen kan vi også kalle en *logisk* modell av systemet. Det betyr ikke så mye mer enn at dette er slik systemet ser ut for en programmerer som ser på systemet fra "toppen og nedover". Vi skal nå se på en annen modell som sier noe om hvordan systemet vil se ut fra "bunnen og opp", perspektivet til en person som skal bygge en datamaskin.

I denne modellen ser vi ikke på programvare og deler maskinvaren inn i tre nivåer. Når vi kan utelate programvaren er det nettopp fordi denne kun består av instruksjoner som er kodet som "enere og nuller", så det er det høyeste nivået vi trenger å bry oss om. De vanligste nivåene i en maskinvareorientert lagdeling er *portnivået*, *registernivået* og *prosessornivået*. Hovedforskjellen mellom nivåene er detaljeringsgraden, med portnivået som det mest detaljerte. En oversikt over disse nivåene er gitt i tabellen under.

Navn på nivået	Komponenter	IC tetthet	Informasjons-enheter	Tidsenheter
Portnivå (Gate)	Logiske kretser, Flip-Flops,...	SSI	Bits	10^{-10} - 10^{-8} Sek
Registernivå	Registre, sekvensielle og kombinatoriske kretser	MSI	Ord (Words)	10^{-9} - 10^{-6} Sek
Prosessornivå	CPU, Minne, busser, I/O	LSI/VLSI	Grupper av ord	10^{-6} - 10^3 Sek

Disse tre nivåene dekker omtrent det samme området som de tre nederste nivåene i den funksjonsorienterte modellen. Vi skal nå gå videre med den maskinvareorienterte modellen.

Teorien bak maskinvaren er komplisert og vil ikke bli vektlagt, det er kun meningen at du skal få en følelse av hvordan tingene "virkelig" foregår på fysisk nivå.

Portnivået

De fleste har hørt at datamaskiner egentlig bare jobber med "nuller og enere". Med dette menes at det benyttes kun to symboler til å *representere* ulike verdier og symboler. Ved å bruke kun to elektriske spenninger kan man ved hjelp av enkle byggeklosser få til større og mer kompliserte enheter.

Før vi fortsetter bør du bruke litt tid på å tenke gjennom hva ordet "representere" betyr. Forstavelen "re-" betyr "om igjen". Å "presentere" vil si "å vise frem". Ordet "telefon" som du leser nå, er en representasjon i språkssystemet (norsk skriftspråk) av et objekt jeg nettopp har observert. Når jeg skriver "telefonen ringer" representerer jeg et faktum fra min livsverden over i språkssystemet. Dette kan virke unødvendig filosofisk, men anvendeligheten til datamaskin skyldes nettopp muligheten til å re-presentere data i ulike formater og koder. Vi skal komme tilbake til dette i neste kapittel, men ha det i bakhodet!

Teorien som benyttes når vi skal anvende 0'ere og 1'ere kalles Boolsk algebra og ble egentlig utviklet for å analysere sannhetsverdiene til kompliserte påstander. Dersom du sier "Hvis det *ikke* regner, så går jeg hjem" kan vi sette opp en *sannhetstabell* for sammenhengen i utsagnet og bruke denne til å forta en beslutning:


"regner"	"gå hjem"
Sant	Usant
Usant	Sant

Tabell 2: Enkel sannhetstabell

Poenget med Boolsk algebra er at vi kun bryr oss om logikken, sannhetstabellen over vil være like korrekt for alle utsagn som har formen "hvis IKKE xxxxxxxx, så yyyyyy":


Ved hjelp av transistorer bygges de grunnleggende logiske kretsene. Disse kalles logiske porter. I Boolsk algebra er det vanlig å definere én som *Sann* (*True*) og null som *Usann* (*False*). I mikroprosessen er nullene og enerene egentlig bare ulike elektriske spenningsnivåer. Null kan være f.eks. være definert som mellom 0 og 1 Volt og én som mellom 3 og 5V.

En logisk ports egenskaper angis ved en sannhetstabell som beskriver output for de ulike




NOT

A	X
0	1
1	0




AND

A	B	X
0	0	0
0	1	0
1	0	0
1	1	1




OR

A	B	X
0	0	0
0	1	1
1	0	1
1	1	1




XOR

A	B	X
0	0	0
0	1	1
1	0	1
1	1	0



NAND

A	B	X
0	0	1
0	1	1
1	0	1
1	1	0



NOR

A	B	X
0	0	1
0	1	0
1	0	0
1	1	0

Figur 4: Symboler og sannhetstabeller for noen logikkporter.

mulige input-verdiene. Noen av de mest brukte portene er gjengitt i figuren under.

Sannhetstabellene leses ved å se etter hvilken verdi porten har ved utgangen (X) for de ulike inngangsverdiene (A og B).

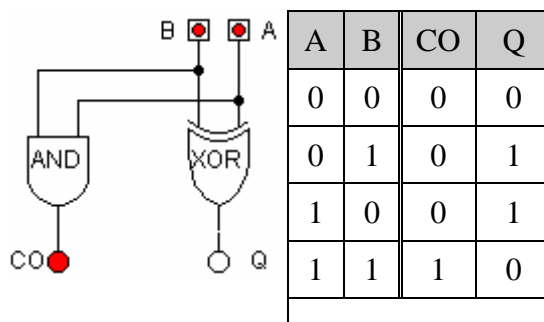
For eksempel ser vi at en NOT-port snur (inverterer) inngangsverdien. Det vil si at en 0 ved inngangen blir til en 1 ved utgangen. En AND-port må ha begge inngangene lik 1 for at utgangen skal bli 1, mens en OR-port kun krever at en av inngangene er lik 1 for at utgangen skal bli 1.

Figuren under viser hvordan slike porter kan brukes til å bygge en enkel logisk funksjon. A og B er input (f.eks. brytere), Q og CO er output (f.eks. lamper). For denne kretsen vil det gjelde at:

$$Q = A \text{ XOR } B$$

$$CO = A \text{ AND } B$$

Ut fra sannhetstabellene for de logiske portene XOR og AND (figuren over) kan vi da sette opp sannhetstabellen for denne kretsen.



Figur 5: Skjema og sannhetstabell for half adder

Ut fra denne kan vi lese at dersom enten A eller B er på, så lyser lampe Q. Dersom både A og B er på lyser lampe CO. Dermed har vi faktisk en krets som kan brukes til å legge sammen to enbits binærtall! Som vi skal se i delen om datarepresentasjon så fungerer binæraritmetikken slik at:

$$0+0=00$$

$$0+1=01$$

$$1+0=01$$

$$1+1=10$$

Den samme elektroniske kretsen kunne vært brukt til et helt annet formål. Si at du skal overvåke om dører i huset er åpne eller lukket. Du kan montere bryterne i dørkarmene. Dersom ingen lamper lyser så er begge dørene åpne. Dersom Q lyser så er en av dørene åpen. Dersom CO lyser så har du gjennomtrekk! På den måten kan du benytte de logiske portene til å signalisere om ulike kombinasjoner av tilstander har inntruffet. Og du kan bruke disse signalene som styresignaler for andre kretser.

Registernivået

En bit kan kun ha to verdier og dette er selvfølgelig alt for lite. Med to mulige verdier kan vi kun kode svært enkel informasjon, f.eks. om utgangsdøren er lukket (1) eller åpen (0).

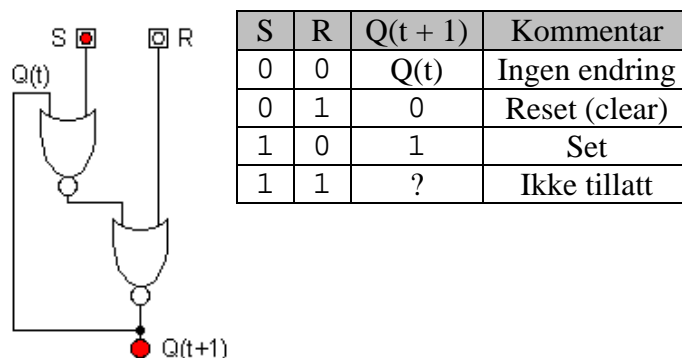
Vi samler derfor flere bit i en gruppe og behandler dem som en samlet enhet som vi kaller et *ord* (word). Ordstørrelsen kan variere fra 4 og helt opp til 128 bits. Ord bestående av 4 bits kalles en *nibble*, 8 bits kaller vi en *byte*, 16 bits kalles (forvirrende nok) et ord (word), 32 bits et dobbeltord (Dword) og 64 bits et kvadruppelord (Qword).

En krets som kan lagre et ord kaller vi et *register*. Registeret kan adresseres som en enhet og innholdet leses eller skrives parallelt (samtidig). Registre er den raskeste (og dyreste) formen for minne vi har i en datamaskin.

Registre finner vi inne i CPU'en der de brukes til å oppbevare adresser og data. Registre brukes også på kontrollerkortene til periferiutstyret for å mellomlagre data og instruksjoner.

De enkle logiske kretsene vi har sett på så langt var *responskretser*. Det vil si at de umiddelbart endrer utgangsverdi når inngangsverdien endres. De kan dermed ikke brukes til å lagre verdier. De husker ikke tidligere tilstander.

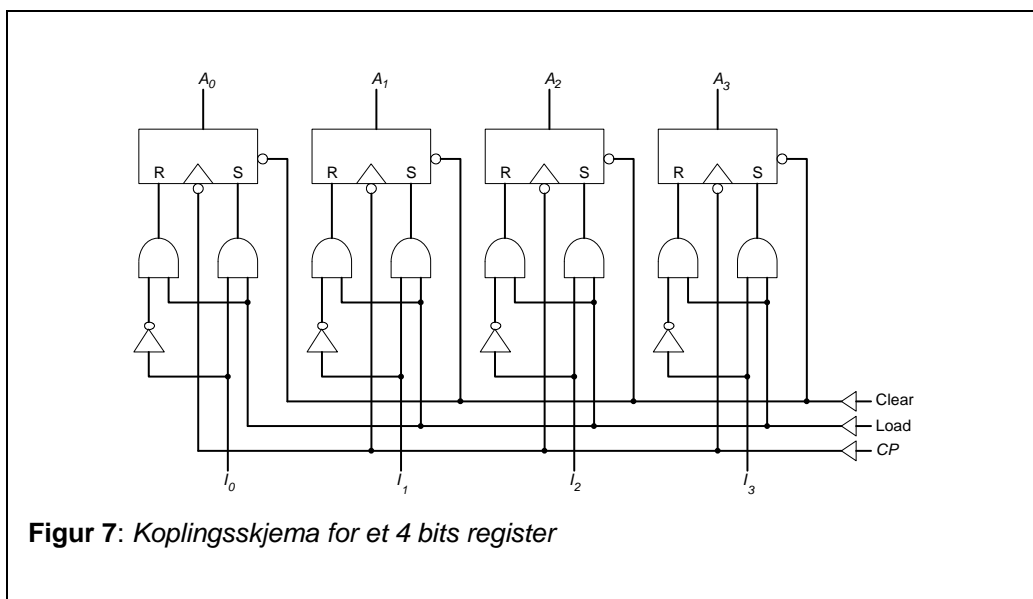
Flip-Flops er *sekvensielle kretser*, det vil si at inngangsverdien og en intern tilstandsverdi avgjør utgangsverdien.. Koplingsskjema og sannhetstabell for en enkel Flip-Flop er vist i figuren under.



Figur 6: Skjema og sannhetstabell for SR-Flip Flop

Flip-Flops kommer i både synkrone og asynkrone utgaver, men den synkrone er den mest brukte. I disse kan utgangsverdien kun endres når et klokkesignal går høyt.

Ved hjelp av synkrone Flip-Flops kan vi bygge registre. Figuren under viser koplingsskjemaet for et 4 bits register.



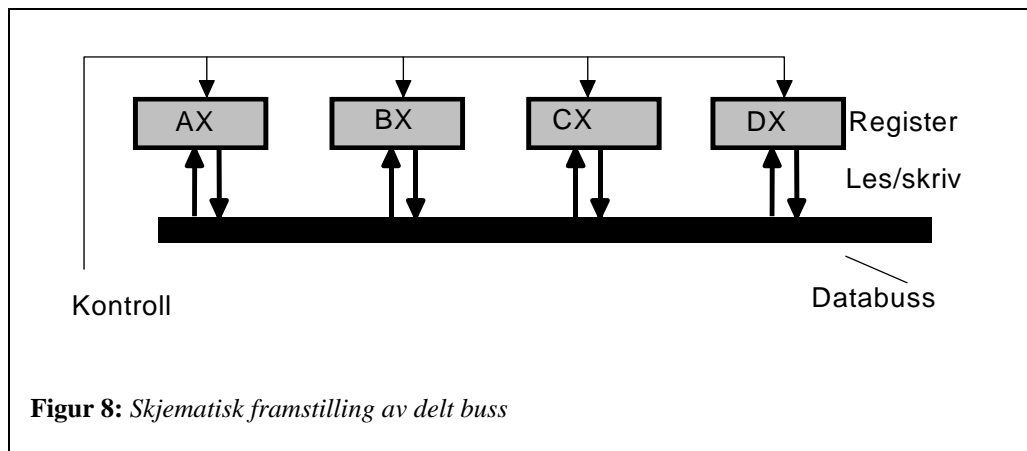
Figur 7: Koplingsskjema for et 4 bits register

Vi ser at registeret har fire utganger $A_0 - A_3$, fire innganger $I_0 - I_3$ og tre kontrollsignaler (Clear, Load og klokkepuls [CP]). De fire inngangsverdiene bestemmer hva som lagres i registeret når kontrollsignalet Load blir 1. Utgangene brukes selvsagt til å lese hva som er lagret i registeret. Klokkepulsens sørger for at inngangsverdiene $I_0 - I_3$ skrives til alle lagerplassene samtidig.

Busser

For at enhetene på registernivået skal kunne kommunisere benyttes *busser*. Busser er ikke annet enn en samling av ledninger. Typisk vil bredden på bussen være lik

dataordstørrelsen. Ved delte busser er flere registre knyttet til samme buss og det er egne kontrollsignaler på en egen ledning som bestemmer hvilken enhet det er som får lese/skrive fra/til bussen. En skjematisk framstilling av en delt buss er gitt i figuren under.



Prosessornivået

På prosessornivået bryr vi oss kun om hvordan prosessoren er bygget opp. Det er vanlig å tenke på CPU'en når vi snakker om prosessorer, men dette er langt fra den eneste prosessoren i maskinen. F.eks. er det ikke uvanlig at prosessoren som sitter på skjermkortet er raskere enn CPU'en. CPU'en er en spesiell mikroprosessor som bl.a. styrer de andre prosessorene i maskinen. Den vil være den mest kompliserte enheten i systemet.

Typisk vil både minnet og inn-/utenhetene ha egne kontrollerbrikker som utfører ønskede operasjoner. Når CPU'en skal hente innholdet fra en minnelokasjon (RAM) vil den først legge ut adressen på adressebussen, så signalisere til minnekontrolleren på kontrollbussen, og så vente på at svaret kommer på databussen.

Sentralenheten (CPU'en)

CPU'er kommer i mange varianter. Felles for dem alle er at de skal utføre generelle oppgaver. De er spesialdesignet for å utføre de instruksjonene som er lagret i internminnet.

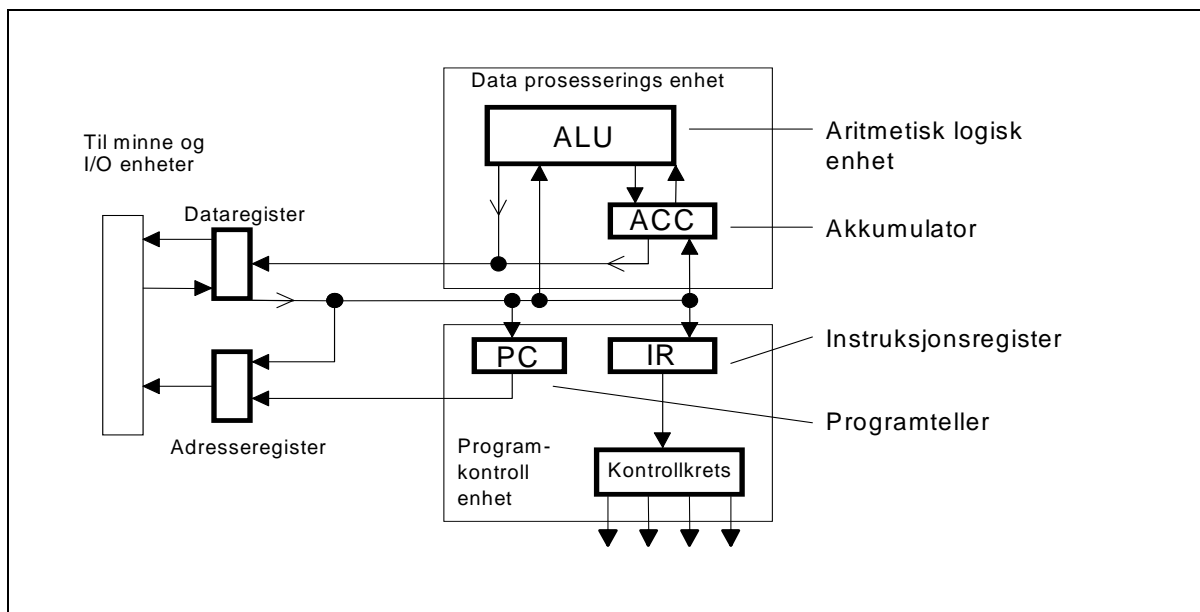
En enkel CPU har følgende hovedbestanddeler:

- *Kontrollenheten (CU)*
 - står for dekodning og utføring av instruksjonene
 - interne klokkesignaler/synkronisering
- *Aritmetisk Logisk Enhet (ALU)*
 - utfører instruksjoner
 - foretar matematiske operasjoner (+, -, *, /, ...)
 - foretar logiske operasjoner (AND, OR, NOT,...)
- *Registere*
 - Programtelleren (PC)
inneholder adressen til neste instruksjon som skal hentes fra primærminnet.
 - Instruksjonsregisteret (IR)
mellomlagrer bytekoden for instruksjonen mens denne dekodes og utføres.

- Akkumulatorregisteret (ACC)
innholder resultatet av en ALU-operasjon
- Tilstandsregisteret (Flags)
inneholder bits som settes til 1 eller 0 avhengig av resultatet av ALU-operasjoner. Blir resultat 0 settes et Z-flagg i registeret. Resulterer operasjonen i flere bits enn det er plass til i ACC-registeret settes et O-flagg (overflow). Annen tilstandsinformasjon kan også lagres på lignende vis..

I moderne CPU'er er det mange register som benyttes til ulike formål. For eksempel hadde Intel 8080 (1975) 10 registre, hvorav fire kunne brukes fritt av programmereren. En Pentiumprosessor har framdeles fire registre som kan brukes fritt, men de er ikke lenger 8 bits, men 32 bits (EAX, EBX, ECX, EDX). I tillegg har Pentiumprosessenor tyve registre som brukes i minnebehandlingen, 8 64 bits registre som benyttes til multimediaformål (MMX), 8 som benyttes til flyttallberegninger, registre som benyttes til å oppbevare nøkkelinformasjon om programmet under kjøring, m.fl.

Den eldste og enkleste typen CPU er den akkumulatorbaserte. Skissen under viser noen av de viktigste komponentene i en slik prosessor.



Figur 9: Akkumulatorbasert CPU (forenklet)

Her vil resultatet av alle operasjoner havne i akkumulatorregisteret. I dette kompendiet skal vi lage oss et instruksjonssett for en slik enkel CPU. Typiske instruksjoner vil være LOAD (hent data/instruksjon fra primærminnet eller Inn-enhet), STORE (send data til primærminne eller Ut-enhet), ADD (legg innholdet i ACC-registeret sammen med innholdet i dataregisteret...) osv.

CPUen har to hovedkomponenter. En kontrollenhet (CU) som henter og dekker instruksjonene og inneholder all styringslogikken. I denne finner vi PC-registeret som brukes som peker til plassen i primærminnet der instruksjonene hentes fra. Innholdet i dette registeret vil altså fortelle oss hvor langt i det aktive programmet vi har kommet. Instruksjonsregisteret (IR) brukes som mellomlager for den siste instruksjonen som har blitt hentet inn fra primærminnet. ALU'en har på sin side som hovedoppgave å utføre operasjoner på data som hentes fra minnet.

Denne prosessoren kan styre en maskin vi bruker til å skrive brev på! Hvordan det er mulig, er langt fra opplagt. For å forstå hvordan må vi se nærmere på hvordan alt kan kodes som tall. Vi må lenger inn i 0 og 1's enkle verden.

Representasjon av data i datamaskinen

Så langt har vi bare sett på hvordan vi kan lagre nuller og enere, samt hvordan disse verdiene kan manipuleres ved Boolsk algebra. Vi er ikke så opptatt av selve nullene og enerene som vi er av den informasjonen den bærer.

Datamaskinens primærlager (RAM) kan vi tenke på som et antall lokasjoner (plasser) som hver har plass til en byte. Hver lokasjon har en entydig adresse. Denne adressen benyttes av sentralenheten til å velge ut hvilken lokasjon det skal skrives til eller leses fra.

Lokasjonene i primærlageret kan inneholde både instruksjoner og data. Vi har så vidt sett på hvordan instruksjoner hentes fra primærlageret, for deretter å utføres av sentralenheten. Vi skal senere se på hvordan en instruksjon kan føre til at data leses fra eller skrives til primærlageret.

Hvordan er det mulig for deg å lese denne teksten? Det har å gjøre med minst to forhold:

- 1) Du og jeg har lært noen felles kodingsregler, nemlig norsk skriftspråk.
- 2) Siden du vet hvordan jeg koder, er du i stand til å dekode og (forhåpentligvis) forstå budskapet jeg har kodet.

I norsk skriftspråk benytter vi en notasjon som består av 29 små bokstaver, 29 store bokstaver, samt en del andre symboler slik som komma, punktum, tankestrek, utropstegn m.fl. Internt i datamaskinen bruker vi kun to symboler. Vi vil heretter notere disse som 1 og 0. Det finnes ingen andre måter å representere data på enn med 1 og 0.

Hva betyr så 0100 0001?

For å kunne besvare det spørsmålet må du igjen spørre deg selv:

- 1) Hvordan er det kodet?
- 2) Foreta dekodingen.

I datamaskinen finnes det flere tabeller som benyttes til å oversette/dekode, men det er programmereren som må holde styr på hvilken tabell som skal brukes i hvert tilfelle.

For eksempel så kan 0100 0001 være koden for:

- en bokstav (A)
 - et heksadesimalt tall (41)
 - et binærtall (0100 0001)
 - et billedpunkt på en (gammel) skjerm
 - en Intel assemblyinstruksjon (INC CX)
 - en del av en instruksjon
 - en del av en adresse i primærminnet
 - adressen til en periferienhet
- osv...

Hvor mange symboler vi har å kode informasjonen med avhenger av hvor mange bits vi bruker i kode-tabellen. Dersom vi bruker fire bits har vi 16 mulige symboler:

0000
0001
0010
0011
0100
0101

0110
0111
1000
1001
1010
1011
1100
1101
1110
1111

Dersom vi vil ha en kode-tabell for de seksten tallene fra 0 til 15, så er vi i mål. Dersom vi vil ha en måte å representere kodesettet til norsk skriftspråk innser vi at vi må bruke flere bits. Vi skal nå se nærmere på en av de mest utbredte kodetabellene ASCII.

ASCII-koder

Tegn lagres i primærlageret som bitmønstre (en ordnet rekke enkeltbits). Hvert tegn har sin egen unike kombinasjon av bits. I svært mange sammenhenger benyttes ASCII-koder. ASCII (American Standard for Information Interchange) er en standard som definerer hvilken bitkombinasjon som representerer hvert enkelt tegn.

I sin opprinnelige form definerer ASCII et 7-bits mønster for hvert enkelt tegn. Dette gir mulighet for 128 forskjellige tegn. Det finnes også utvidete, 8-bits utgaver som gir mulighet for 256 forskjellige tegn.

Tabellen under viser noen få tegn med tilhørende bitmønstre:

Tegn	Bitmønster
'A'	0100 0001
'B'	0100 0010
'C'	0100 0011
'a'	0110 0001
'b'	0110 0010
'0' (null)	0011 0000
'1'	0011 0001

Tabell 3: Noen ASCII-tegn og verdier

Legg merke til at 0 i tabellen over viser til symbolet 0 og ikke selve tallet! Dette er en kodetabell over tegn og symboler, som aldri må forveksles med det de skal symbolisere.

Når vi trykker på en tast på tastaturet er det et slikt bitmønster som blir laget. Dette gjelder også når vi skriver inn symbolene for tall. Dersom vi fra tastaturet skriver inn etthundreogtjuetre (taster 123) så vil det bli til tre bytes for tegnene: 00110001 00110010 00110011.

Vi skal senere se at ordentlige heltall representeres på en annen måte internt i maskinen, men det er viktig å legge merke til at tall leses inn og skrives ut som enkelttegn. Det vil i de fleste tilfeller være en prosedyre inne i maskinen som gjør om (konverterer) de enkelte tegnene til den form som benyttes til å representere heltall.

Det finnes også andre standarder for å representere tegn enn ASCII. En standard som ble benyttet på IBM's stormaskiner kaltes for EBCDIC. Som en følge av globaliseringen av datateknologien har det blitt et problem at man i ASCII valgte et symbolsett som i første omgang kun tillot 128 forskjellige tegn og kontrollkoder. I første omgang ble dette

utvidet til å benytte alle 8 bits noe som gav dobbelt så mange muligheter og man fikk dermed med tegn som "æ", "ø" og "å". ISO-8859-1 og Windows 1252 er begge tegnssett av denne typen, der også de vanligste tegnene i Vest-Europeiske alfabet er tatt inn. Skal du skrive kinesisk holder dette framdeles ikke.

Standarden på verdensbasis skal bli UNICODE (<http://www.unicode.org>). Siden alle verdens skriftspråk til sammen benytter mer enn 200000 ulike symboler forutsetter dette at vi bruker med enn 8 bit pr tegn. I Unicode skilles tegnene og deres tilordnede tall fullstendig fra hvordan det skal representeres. Den vanligste brukte delen av Unicode er Basic Multilingual Plane som tilbyr 65536 ulike koder. De 256 første er de samme som i ISO 8859-1, og de første 128 de samme som i ASCII. Unicode-koden for en bokstaven "Æ" (kode-koordinatene) skrives U+00C6, der 00C6 er et heksadesimalt tall (se under). Unicode sier derimot ikke noe om hvordan disse kode-punktene faktisk skal representeres binært. De vanligste tilnærmingene er UTF-16 og UTF-8. I UTF-16 vil Uni-koder gjengis direkte, altså 0x00C6 for "Æ". UTF-8 benytter binær-koder med variabel lengde. 7-bit ASCII-kodene vil alle starte med 0 og kodes med en byte. Koden for "Æ" vil i UTF-8 kreve to byte. Standarden sier da at 0xC6 = 1100 0110 skal legge inn i to byte i et bestemt mønster. Noen bits er forhåndsbestemt og ligger fast: **110y yyxx 10xx xxxx**, så legges de bitsene vi skal ha inn på x-plassene: **1100 0011 1000 0110** blir den binære kodingen.

Representasjon av heltall

Heltall representeres sjelden internt i maskinen som en samling av ASCII-tegn. Grunnen til dette er at det er lite praktisk å gjøre regneoperasjoner på slike tegn. Det blir litt som om vi også skulle ha brukt notasjonen til skriftspråket når vi regnet, og satt opp regnestykket:

Hundreogtjuetre
pluss Tusenogtrettifem

Dette er ikke så langt unna det de gamle romerne gjorde. Som du sikkert vet brukte de ulike bokstaver for tall av ulik størrelsesorden. Tallet som vi noterer som 2366 skrev romerne som MMCCCLXVI. Tanken bak er den samme som i vårt vanlige tallsystem:

2 stk 1000
+ 3 stk 100
+ 6 stk 10
+ 6 stk 1

De hadde også egne symboler for 5 (V), 50 (L) og 500 (D) i tillegg.

Forskjellen består i at vi benytter et posisjonstallsystem der de ulike sifrene vektes i forhold til hvor de er plassert. En annen måte å skrive 2366 på er:

$$2 \cdot 10^3 + 3 \cdot 10^2 + 6 \cdot 10^1 + 6 \cdot 10^0$$

Internt i datamaskinen benyttes et tallsystem tilsvarende det titallsystemet vi kjenner så godt, men igjen med den begrensningen at vi kun har 1 og 0. Dersom du forstår hvordan det vanlige titallsystemet fungerer så er det ikke vanskelig å forstå hvordan dette binærtallsystemet virker.

Vi utnytter at ethvert tall kan skrives som en sum av potenser av tallet 2. Dette skulle vises ved følgende enkle eksempler:

$5 = 4 + 1$	$= 2^2 + 2^0$
$13 = 8 + 4 + 1$	$= 2^3 + 2^2 + 2^0$
$192 = 128 + 64$	$= 2^7 + 2^6$
$12678 = 8192 + 4096 + 256 + 128 + 4 + 2$	$= 2^{13} + 2^{12} + 2^8 + 2^7 + 2^2 + 2^1$

Vi utnytter dette ved å la hver bit i en lagerlokasjon representere en potens av tallet 2. De bits som representerer potenser som inngår i summen, settes til verdien 1, mens de potenser som ikke inngår i summen, settes til verdien 0.

La oss nå ta utgangspunkt i at en lokasjon i minnet har plass til 8 bits. Tallene som er gitt i eksempelet ovenfor vil da bli representert på følgende måte:

vekt	128	64	32	16	8	4	2	1
2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0	
5 =	0	0	0	0	0	1	0	1
13 =	0	0	0	0	1	1	0	1
192 =	1	1	0	0	0	0	0	0
12678 =	kan ikke skrives direkte med 8 bits!							

Legg merke til en viktig begrensing her. 8 bits betyr at vi kun har 256 forskjellige symboler og dermed blir det største heltallet vi kan representere rett fram 255. Når vi i et programmeringsspråk deklarerer variablene vi skal bruke, så er det nettopp for at kompilatoren skal ha mulighet til å vite hvor mye plass i minnet den må reservere for hver enkelt variabel.

Vi bør også nevne at vi bruker heltall til to forskjellige formål: til å telle hvor mange det finnes av en gitt type objekt (f.eks. kroner) og til å angi posisjon i en sekvens (f.eks. tredje rikeste reder i Norge). Dette kalles hhv kardinal- og ordinaltall. I datasammenheng starter vi alltid ordialtallrekken på 0, slik at 0 vil være symbolet for "første", 1 for "andre", 10 for "tredje", 11 for "fjerde" osv.

BCD

Man behøver bare fire bits for å representere tallene 0-9, og derfor kan man lagre to desimaltall i hver byte. Denne lagringsformen heter "Packed Binary Coded Decimal". Den er ikke spesielt godt egnet til aritmetikk og krever konverteringsrutiner. De fleste prosessorers instruksjonssett har av historiske årsaker likevel støtte for å regne med BCD. I dette systemet ville 2366 bli 0010 0011 0110 0110. Det er lettere for et menneske som kjenner kodingsregelen "hver nibble er et tall mellom 0 og 9" å lese seg fram til at dette

bitmønsteret representerer 2366 enn å se at binærtallet 0000 1001 0011 1110 representerer det samme tallet.

Poenget å ta med seg er at det finnes mange måter å representere tall på. Man må foreta et valg av hvilken representasjonsform man vil bruke og så holde seg til det.

Flyttall

Så langt har vi kun sett på heltall. Hva så med tall som 0,356 eller $\frac{1}{2}$ eller de absurd store og små tallene som fysikken og økonomien er fulle av?

Det har blitt benyttet og foreslått ulike standarder, men i våre dager er alle maskiner utstyrt med egne flyttallsenheter (FPU'er) som er spesialdesignet for nettopp å håndtere slike størrelser. Metoden som benyttes kalles et flyttallsformat. Standarden som råder når det gjelder flyttall heter IEEE 754. Den angir regler for å kode reelle tall ved hjelp av 32, 64 eller 80 bits.

I et 32 bits flyttall benyttes den mest signifikante biten til å angi hvorvidt tallet er positivt eller negativt. Så følger 8 bits som angir en eksponent for base 2. Og 23 bits til å lagre måltallet (mantissa).

Et tall kodet i dette formatet vil se ut som følger:

0 1000 0110 1101 1100 1000 0000 0000 000

Dersom du vil avkode denne binærkoden trenger du noe ekstra informasjon. Eksponenten er kodet i det som kalles 127-offset notasjon. Det betyr at 1000 0110 er en koding av tallet $(128+4+2) - 127 = 7$

Mantissa er oppgitt i normalisert form med en implisitt ledende 1'er. Uten at vi skal gå så nøye inn på forklaringen betyr det at mantissa her koder:

$$1 + 1 * 2^{-1} + 1 * 2^{-2} + 0 + 1 * 2^{-4} + 1 * 2^{-5} + 1 * 2^{-6} + 0 + 0 + 1 * 2^{-9} + \dots + 0 =$$

$$= 1 + \frac{1}{2} + \frac{1}{4} + \frac{1}{16} + \frac{1}{32} + \frac{1}{64} + \frac{1}{256} = 1,861\ 328\ 125$$

Forutsatt at vi har regnet riktig, og vi ser at dette er et format som egner seg langt bedre for logiske maskiner enn for kreative mennesker, vil da det kodete tallet i desimalform være

$$1,861\ 328\ 125 * 2^7 = 110,25$$

Det er selvsagt *ingen* vettuge mennesker som setter seg ned og regner med flyttall for hånd! At det derimot er viktig å forstå hvordan dette formatet fungerer erfarte man ved ESAs oppskytingsbase for raketter i Fransk Guiana i juni 1996. Der hadde man unnlatt å følge en av spesifikasjonene i standarden for hvordan flyttall skulle behandles, med det resultat at raketten de skulle sende opp sprengte seg selv i luften.

32 bits flyttall i henhold til denne standarden kan representere tall mellom 1.18×10^{-38} og 3.40×10^{38} . Det burde holde til de fleste lønnsprogrammer, Microsoft innkludert.

Addisjon av binære tall

Det er ikke så vanskelig å addere binære tall. Egentlig benytter vi akkurat den samme teknikk som vi benytter når vi adderer desimaltall. La oss se på hvordan vi vanligvis utfører regnestykket $236+468$ i desimaltallsystemet:

	Antal 1 100	antall 10	antall 1
	2	3	6
+	4	6	8
	6	9	14

Først kan vi legge sammen antall enere, deretter 10'ere, deretter 100'ere osv. Vi ser at vi har mer enn 9 enere, så vi "veksler ti enere i en tier". Da har vi ti tiere som vi veksler inn i en hundrer. Slik er logikken bak at vi ender opp med 704. De fleste av oss har lært å gjøre dette ved å benytte et "mente-tegn" hver gang vi får en sum større enn 9 i en kolonne:

mente	1	1	
	2	3	6
+	4	6	8
	7	0	4

Vi benytter samme teknikk når vi skal legge sammen binære tall, men da kan vi ikke tillate at summen i en kolonne blir større enn 1. Eksempelet under illustrerer framgangsmåten:

$$\begin{array}{r} 0001 \ 0011 \\ + \ 0001 \ 0011 \\ \hline = \end{array}$$

Vi starter med tallene i kolonnen helt til høyre. Her ser vi at vi har to 1'ere. Det blir en toer og skal dermed noteres i posisjonen til venstre for enerkolonnen. Det vil si at vi må sette mentemerke over den neste kolonnen mot venstre.

$$\begin{array}{r} 1 \\ 0001 \ 0011 \\ + \ 0001 \ 0011 \\ \hline = 0 \end{array}$$

Vi går nå videre til neste kolonne mot venstre og legger sammen elementene i denne. Her skal vi legge sammen tre 1'ere. I det binære tallsystem er $1+1+1=11$.

$$\begin{array}{r} 1 \\ 0001 \ 0011 \\ + \ 0001 \ 0011 \\ \hline = 10 \end{array}$$

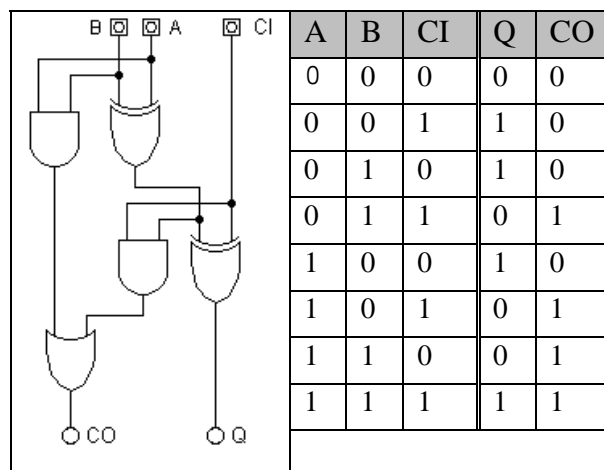
Neste kolonne er greiere ettersom vi her kun må ta med resultatet fra forrige kolonne. Vi skal altså legge sammen to 0'er og en 1'er, noe som gir resultatet 1.

$$\begin{array}{r} 1 \\ 0001 \ 0011 \\ + \ 0001 \ 0011 \\ \hline = 110 \end{array}$$

Og slik kan vi fortsette med resten av kolonnene til vi er ferdig med regnestykket:

$$\begin{array}{r} 11 \\ 0001 \ 0011 \\ + \ 0001 \ 0011 \\ \hline = \ 0010 \ 0110 \end{array}$$

La oss nå knytte denne framstillingen litt opp mot maskinvaren. Figur 10 viser hvordan vi kan videreutvikle den enkle tobits addisjonskretsen vi så på tidligere til en ny krets som adderer binære tall og tar hensyn til mente. A og B er her to bits som skal adderes, CI er mente fra "forrige kolonne", Q er summen og CO er mente fra denne addisjonen.



Figur 10: Addisjonskrets (full adder) med sannhetstabell

For å lage en krets som kan legge sammen byte-store tall vil vi sette åtte slike kretser ved siden av hverandre og kople CO fra mindre signifikante bits i byten til CI på det mer signifikante.

Negative tall

Vi har foreløpig ikke brydd oss særlig om negative tall. Igjen handler dette om å finne en praktisk måte å kode informasjonen på. En mulighet er å sette av en bit i ordet, f.eks det som befinner seg helt til venstre, til å indikere om tallet er negativt eller positivt. Dette så vi at ble gjort i flyttallstandarden IEEE 754.

Ett av problemene med å bruke en slik framgangsmåte er at vi vil få to ulike representasjoner av tallet 0, nemlig +0 og -0. I matematikken er 0 pr def et positivt tall. For å løse dette og visse andre problemer benyttes vanligvis en spesiell koding som kalles toer-komplement.

Før vi kan forklare hva toer-komplement er, skal vi se på noe som kalles for ener-komplement. I dette tilfelle er et negativt tall det samme som komplementet av det tilsvarende positive tall. Komplement betyr at alle 0'er omgjøres til 1'ere, mens alle 1'ere omgjøres til 0'er. Dette kalles også for å "flippe" bits'ene.

Tallet 13 representeres som vi tidligere har sett på følgende måte:

0 0 0 0 1 1 0 1

Tallet -13 vil da bli representert på følgende måte:

1 1 1 1 0 0 1 0

Vi ser at mest signifikante bit (MSB) angir hvilket fortegn tallet har. Men selv om vi benytter ener-komplement, unngår vi ikke problemet med +/- 0. Vi får to ulike koder for tallet 0:

0000 0000 og enerkomplementet til dette, nemlig: 1111 1111.

Toer-komplement tar utgangspunkt i ener-komplement, bortsett fra at vi legger til en 1'er etter at vi har funnet fram til komplementet. Algoritmen for å finne toerkomplementet til en binærtall er altså:

- 1) flipp alle bits'ene
- 2) legg til 1

La oss se på hvordan firebits siffere vil representeres ved hjelp av toer-komplement:

7	=	0	1	1	1
6	=	0	1	1	0
5	=	0	1	0	1
4	=	0	1	0	0
3	=	0	0	1	1
2	=	0	0	1	0
1	=	0	0	0	1
0	=	0	0	0	0
-1	=	1	1	1	1
-2	=	1	1	1	0
-3	=	1	1	0	1
-4	=	1	1	0	0
-5	=	1	0	1	1
-6	=	1	0	1	0
-7	=	1	0	0	1
-8	=	1	0	0	0

Vi ser at ved å legge til 1 så unngår vi problemet med +/- 0. Når vi skal omgjøre et negativt tall til det tilsvarende positive tall går vi fram på akkurat same vis: vi tar enerkomplement og legger til én. Bruk litt tid på denne tabellen, ser du noen klare mønstre?

Dersom vi benytter toerkomplement så er det lett å se om tallet er positivt eller negativt. Dersom mest signifikante bit er 1, så er det negativt. Dersom mest signifikante bit er 0, så er det positivt.

Når vi regner med toerkomplement så ser vi at det kan oppstå et problem. La oss forsøke å foreta regnestykket $(-3) + 7$ ved hjelp av firebits toerkomplement:

$$\begin{array}{r}
 1 \ 1 \ 0 \ 1 \\
 + \ 0 \ 1 \ 1 \ 1 \\
 \hline
 1 \ 0 \ 1 \ 0 \ 0
 \end{array}$$

10100? Det er jo også binærtall-representasjonen for desimaltallet 20!

Toerkomplement-representasjonen forutsetter at vi holder oss strengt til antall bits vi har sagt at vi skal bruke til å kode tallene med (presisjonen). Vi har sagt fire bits presisjon. Og vi ser at de fire minst signifikante bits i svaret er 0100, som jo tilsvarer desimaltallet 4. Hva så med det ekstra 1-tallet? Det kaller vi et spillsiffer og blåser i!

Konvertering - fra desimal til binær

Ofte har vi behov for å konvertere et tall i desimalrepresentasjon til et tall i binær-representasjon. Når du tenker på hvordan posisjonstallsystemet er bygd opp er det ikke vanskelig å finne ut hvordan vi gjør dette. Husk at binærtallet 1001 konverteres til desimaltallsystemet ved å finne ut at dette betyr en åtter, ingen firer, ingen toer, en ener: $8+0+0+1 = 9$. Framgangsmåten (algoritmen) for å gjøre det motsatte er enkel. Man tar desimaltallet og deler dette på to helt til heltallskvotienten blir null. Ved hver divisjon sparer vi på resten etter divisjonen (null eller en). Binærtallet får vi fra resten av hver divisjon, hvor den første resten er det minst signifikante bit (hvor mange enere), og den

siste resten gir den mest signifikante bit. Eksemplet viser konvertering fra desimaltallet 41.

Tall som divideres	Rest fra divisjonen
41	
20	1
10	0
5	0
2	1
1	0
0	1

Ved å lese sifrene fra rest-kolonnen *nedenfra og opp*, ser vi at resultatet blir

101001 ($=32+0+8+0+0+1=41$)

Denne samme algoritmen kan brukes til å konvertere til et hvilket som helst tallsystem, det være seg heksadesimalt (grunntall 16), oktalt (grunntall 8), eller sprut galt (grunntall 4711).

Heksadesimale tall

Binære tall er kompliserte å håndtere, ikke minst fordi det fort blir så mange 1'ere og 0'er å holde styr på.

Å bruke desimaltall i stedet for binære tall er heller ingen god løsning, fordi det er vanskelig å se hvilke verdier de enkelte bits har. Man må i tilfelle regne om desimaltallene til binærtall hver gang man har behov for å vite verdien på hvert enkelt bit.

For å bedre lesbarheten bruker man det heksadesimale tallsystem.

Det heksadesimale tallsystem inneholder ikke bare tallene 0 til 9, men fortsetter med bokstavene A til F. Det som er fint med dette tallsystemet er at vi kan representere alle kombinasjoner av fire bit med et enkelt tegn. Dette gjør at det er langt enklere å se hvilke verdier enkeltbits har til enhver tid.

Tabellen under viser hvilke heksadesimale siffer som finnes:

Heksadesimal	Binært	Heksadesimal	Binært
0	0000	8	1000
1	0001	9	1001
2	0010	A	1010
3	0011	B	1011
4	0100	C	1100
5	0101	D	1101
6	0110	E	1110
7	0111	F	1111

Tabell 4: Heksadesimale siffer

Dette betyr f.eks. at tallene i de eksemplene vi har benyttet tidligere har følgende heksadesimale representasjon:

$$\begin{aligned} 5_{10} &= 0005_{16} \\ 13_{10} &= 000D_{16} \\ 192_{10} &= 0060_{16} \\ 12678_{10} &= 3186_{16} \end{aligned}$$

Mange typer systemprogramvare (f.eks. debugere) forutsetter at du kan lese heksadesimale siffer og har dette som standardrepresentasjon av binærtallene.

Å konvertere mellom binærtall og heksadesimale tall er enkelt. La oss si at vi vil finne ut hvilket heksadesimalt tall som tilsvarer 1000100111011000. Vi gjøre dette ved å gruppere binærtallet fra venstre i bolker på fire siffer, og så oversetter vi disse bolkene direkte.

1000	1001	1101	1000
8	9	D	8

Dette fungerer fordi grunntallet i det heksadesimale tallsystemet $16=2^4$ dermed vil de fire minst signifikante sifrene gi oss antall enere, de fire neste vil gi oss antall 16, de fire neste antall 256, de fire neste antall 4096 osv. Hvert heksadesimalt siffer tilsvarer altså fire bits.

Det er flere notasjoner som benyttes for å angi at man har med et heksadesimalt tall å gjøre. Matematikere skriver et lite 16-tall nederst på tallet, slik vi har gjort i eksempelet over. I datasammenheng er en metode å skrive en liten h etter sifferet. For eksempel: 41h, 2F3Ah. En annen mye brukt notasjon er å skrive 0x foran. For eksempel: 0x41, 0x2F3A. Noen bruker sogar begge deler!

La oss til slutt ta opp igjen problemstillingen vi startet med. Hva **betyr** ordet 89D8h? Jo, det kan være:

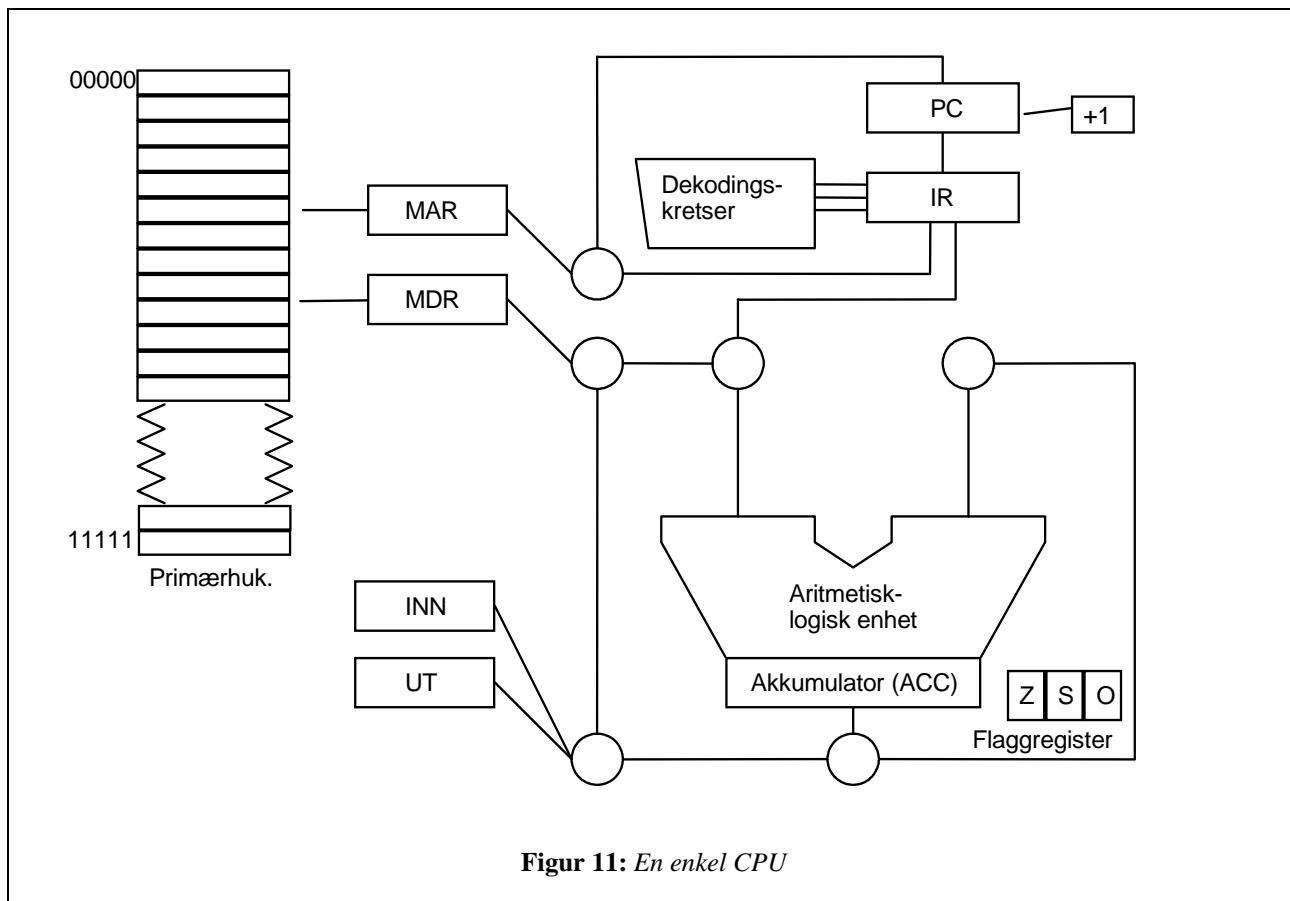
- heksadesimaltallrepresentasjonen av desimaltallet 35288
- heksadesimal representasjon av binærtallet 1000100111011000
- toerkomplementrepresentasjon av det negative tallet: -30248
- maskinkoderepresentasjon for Intel assemblyinstruksjonen MOV AX, BX
- heksadesimal spesifisering av fargen på en pixel på en skjerm
- osv

Heksadesimal representasjon er altså ikke til noe annet enn å skrive binærtall på en lettlest form.

Vi har nå sett hvordan ulike former for data og instruksjoner kan kodes som binærtall. Disse vil inne i maskinen igjen representeres med høye og lave spenninger. Vi har nå et bedre grunnlag for å se på hvordan en CPU arbeider.

Hvordan instruksjoner utføres

Vi skal nå se nærmere på hvordan en CPU utfører instruksjoner som ligger lagret i primærlageret. Figuren under viser oppbygningen av en enkel load/store akkumulatortorbasert sentralenhet med tilhørende primærlager og inn-/utenhet:



Sentralenheten inneholder registre som benyttes til å lagre informasjon som trengs for å utføre (eksekvere) programmer. Listen under beskriver de registrene som benyttes i vår eksempelmaskin.

PC (Programteller). Dette registeret vil til enhver tid inneholde adressen til den lokasjon i primærlageret som neste instruksjon skal hentes fra.

IR (Instruksjonsregister). Dette registeret oppbevarer instruksjoner etter at de har blitt hentet fra primærlageret og før de sendes over i dekodekretsene.

MAR (adresseregister). Når vi ønsker å lese noe fra, eller legge noe inn i primærlageret må vi først plassere adressen i MAR.

MDR (dataregister). Når vi gjør en leseoperasjon fra primærlageret, plasseres resultatet her. Skal vi skrive til primærlageret, må vi først plassere det vi skal skrive i MDR

ACC (Accumulator). Dette registret brukes først og framst i forbindelse med regneoperasjoner. Det er også mulig å lese innholdet av en lagerlokasjon inn i akkumulatoren med instruksjonen LOAD, og lagre innholdet av akkumulatoren med instruksjon STORE.

I tillegg inneholder sentralenheten en ALU (Aritmetisk Logisk Enhet). Denne enheten utfører selve regneoperasjonene i datamaskinen. Avhengig av resultatet av en regneoperasjon, settes ulike flagg (bits) i flaggregisteret:

Z-flagget (Zero-flag) blir satt til 1 hvis resultatet av en regneoperasjon blir null.

S-flagget (Sign-flag) blir satt til 1 hvis resultatet av en regneoperasjon blir negativt.

O-flagget (Overflow-flag) blir satt til 1 hvis resultatet av en regneoperasjon blir for stort (overflyt).

Flaggene brukes til betingede hoppinstruksjoner. Hoppinstruksjonene medfører at innholdet i programtelleren endres slik at den peker til en annen minnelokasjon. Vi har dermed mulighet til lage IF-THEN og DO-WHILE løkker.

Instruksjonssettet

Som tidligere nevnt har en datamaskin et grunnleggende sett av instruksjoner. Tabellen under viser instruksjonssettet for datamaskinen i forrige figur:

Mnemonic	Opkode	Beskrivelse	Funksjon
STOP	0000	Stopp maskinen	
LOAD	0001	Les inn i akkumulator	$(ACC) \leftarrow (MDR)$
STORE	0010	Lagre akkumulator	$(MDR) \leftarrow (ACC)$
ADD	0011	Adder til akkumulator	$(ACC) \leftarrow (ACC) + (MDR)$
SUB	0100	Subtraher fra akkumulator	$(ACC) \leftarrow (ACC) - (MDR)$
BRANCH	0101	Hopp uansett	$(PC) \leftarrow IR<4:0>$
BRZERO	0110	Hopp hvis resultat = 0	$(PC) \leftarrow IR<4:0>$ eller $(PC) \leftarrow (PC) + 1$
BRNEG	0111	Hopp hvis resultat < 0	$(PC) \leftarrow IR<4:0>$ eller $(PC) \leftarrow (PC) + 1$
BROVFL	1000	Hopp hvis overflyt	$(PC) \leftarrow IR<4:0>$ eller $(PC) \leftarrow (PC) + 1$
IN	1001	Les inn fra tastatur	$(ACC) \leftarrow INN$
OUT	1010	Skriv til skjerm	$UT \leftarrow (ACC)$

Tabell 5: Instruksjonssett for CPU'en

Mnemonic er assemblykoden for instruksjonen. Opkoden er det bitmøsteret som angir hva kontrollenheten skal gjøre. Under funksjon lister vi opp "mikrokoden": hvilke operasjoner en bestemt opkode skal utløse i kontrollenheten. Den notasjonen vi benytter leses på følgende måte:

$(ACC) \leftarrow (MDR)$ Kopier innholdet i dataregisteret inn i akkumulatorregisteret

$(PC) \leftarrow IR<4:0>$ Kopier innholdet fra de fem minst signifikante plassene i instruksjonsregisteret over i programtellerregisteret

\leftarrow betyr altså "kopier over i"

() brukes for å angi at det er innholdet i hele registeret vi operer på

<X:Y> angir hvilke bits i et register som skal benyttes, <4:0> angir f.eks. de fem minst signifikante bit'ene.

Maskinarkitekturen vår er litt spesiell:

- opkodene består av 4 bits
- primærlageret består av 16 bits minneceller, som adresseres med 5 bits.
- instruksjoner består av 16 bits ord, der 7 bits er overflødige.

Vi skal nå se litt nærmere på hva som skjer når instruksjoner utføres av en datamaskin. La oss ta utgangspunkt i at eksempelmaskinen har følgende innhold i de syv første minnelokasjonene.

Minneadresse binært/(desimalt)	Maskinkode (opkode+adresse) binært/(desimalt)	Asembly- instruksjon
00000 (0)	0001 0000 0000 0100 (1)(4)	LOAD 4
00001 (1)	0011 0000 0000 0101 (3)(5)	ADD 5
00010 (2)	0010 0000 0000 0110 (2)(6)	STORE 6
00011 (3)	0011 0000 0000 0000 (0)(-)	STOP
00100 (4)	0000 0000 0000 0001 (0)(1)	
00101 (5)	0000 0000 0000 0010 (-)(2)	
00110 (6)	0000 0000 0000 0000 (-)(0)	

Tabell 6: Innholdet i de syv første minnelokasjonene

En datamaskin utfører instruksjoner som hentes fra primærlageret. Programtelleren holder orden på hvor neste instruksjon skal hentes fra. Når maskinen startes opp er innholdet i programtellerregisteret lik null, slik at den første instruksjonen som blir utført blir hentet fra minnelokasjon 0.

I vår eksempelmaskin består en instruksjon av to deler. En del sier hva som skal gjøres (opkode), mens den andre delen sier hvor eventuelle data skal hentes fra eller plasseres. Denne delen kalles adressedelen av instruksjonen.

Adressedelen består av de 5 minst signifikante bits, noe som gir mulighet til å adressere 32 forskjellige lagerlokasjoner. Opkodedelen består av de 4 mest signifikante bits, som gir mulighet for 16 ulike instruksjoner.

Følgende skjer hver gang en instruksjon skal utføres:

- Programtelleren (PC) plasseres i adresseregistret (MAR), og peker derved på den instruksjonen som skal utføres.
- Programtelleren (PC) økes med 1, slik at den peker på neste instruksjon. Samtidig leses den instruksjonen som skal utføres inn i dataregistret (MDR).
- Fra dataregistret (MDR) overføres instruksjonen til instruksjonsregistret (IR), hvor instruksjonen blir mellomlagret på vei til dekodingskretsene.

Hva som skjer nå, vil være avhengig av hva slags instruksjon som har havnet i instruksjonsregistret. Innholdet i lageret vist ovenfor er i realiteten et lite program som legger sammen verdien av to tall som er lagret i lokasjon 4 og 5 og plasserer resultatet i lagerlokasjon 6.

Under gis en mer detaljert beskrivelse av hva som skjer etter at hver enkelt av instruksjonene i maskinens instruksjonssett har havnet i instruksjonsregistret (IR)

STOP Opkode=0 (0000)

Instruksjonen dekodes, og maskinen stopper.

LOAD Opkode=1 (0001)

Instruksjonen dekodes, og adressedelen kopieres over i MAR. Innholdet av den lokasjon som MAR peker på, kopieres inn i MDR. Herfra kopieres innholdet videre til akkumulatoren.

STORE Opkode=2 (0010)

Instruksjonen dekodes og adressedelen kopieres over i MAR. Samtidig kopieres innholdet av akkumulatoren over i MDR. Det som nå ligger i MDR blir plassert i den lagerlokasjon som MAR peker på.

ADD Opkode=3 (0011)

Instruksjonen dekodes og adressedelen kopieres over i MAR. Innholdet av den lokasjon som MAR peker på, kopieres inn i MDR. Innholdet av MDR benyttes nå som input til den venstre inngangen til den aritmetisk-logiske enheten (ALU). Samtidig blir det som allerede ligger i akkumulatoren benyttet som input til den høyre inngangen til ALU'en. Det som befinner seg på de to inngangene blir addert, og resultatet plasseres i akkumulatoren. Flaggene blir satt avhengig av resultatet av addisjonen.

SUB Opkode=4 (0100)

Instruksjonen dekodes og adressedelen kopieres over i MAR. Innholdet av den lokasjon som MAR peker på, kopieres inn i MDR. Innholdet av MDR benyttes nå som input til den venstre inngangen til den aritmetisk-logiske enheten (ALU). Samtidig blir det som allerede ligger i akkumulatoren benyttet som input til den høyre inngangen til ALU'en. Det som befinner seg på den venstre inngangen blir nå subtrahert fra det som befinner seg på den høyre inngangen, og resultatet plasseres i akkumulatoren. Flaggene blir satt avhengig av resultatet av subtraksjonen. (I realiteten vil dette foregå ved bruk av toerkomplement: først inverteres alle bits i ordet som skal trekkes fra, så legges til 1 og deretter benyttes addisjonskretsen i ALU'en på vanlig vis).

BRANCH Opkode=5 (0101)

Instruksjonen dekodes, og adressedelen kopieres over i programtelleren (PC). Neste instruksjon vil altså hentes fra adressen angitt i instruksjonens adressedel.

BRZERO Opkode=6 (0110)

Instruksjonen dekodes. Hvis Z-flagget er satt (1), kopieres adressedelen over i programtelleren (PC). Dersom Z-flagget er 0 inkrementeres innholdet i PC-registeret på vanlig måte.

BRNEG Opkode=7 (0111)

Instruksjonen dekodes. Hvis S-flagget er satt, kopieres adressedelen over i programtelleren (PC). Dersom S-flagget er 0 inkrementeres innholdet i PC-registeret på vanlig måte.

BROVFL Opkode=8 (1000)

Instruksjonen dekodes. Hvis O-flagget er satt, kopieres adressedelen over i programtelleren (PC). Dersom O-flagget er 0 inkrementeres innholdet i PC-registeret på vanlig måte.

IN Opkode=9 (1001)

Innenheten avleses, og det som leses plasseres i akkumulatoren.

OUT Opkode=10 (1010)

Innholdet av akkumulatoren skrives til utenheten.

Assemblyprogrammering og assemblerens virkemåte

Når vi kompilerer et program skrevet i et høynivåspråk, f.eks. C++ eller Java, blir hver setning oversatt til en sekvens med enkeltinstruksjoner. Ved programmering i maskinspråk (assembly) blir hver setning oversatt direkte til en av instruksjonene i CPU'ens instruksjonssett. Vi kan ta et lite eksempel fra programmeringsspråket Java. Dette kompiles vanligvis ikke til maskininstruksjoner for en bestemt prosessor, men oversettes til byte-koder som er instruksjonssettet for Java Virtual Machine (JVM). JVM kjøres vanligvis opp mot et operativsystem, men Sun har også designet mikroprosessorer som kan kjøre bytekode direkte.

Java	Mnemonic	Bytekode (heksadesimalt)
$i = j + k$	ILOAD j	0x15 0x02
	ILOAD k	0x15 0x03
	IADD	0x60
	ISTORE i	0x36 0x01

Dette eksempelet viser hvordan en linje Javakode skrevet i en editor blir oversatt til fire linjer bytekode som må utføres sekvensielt. Vi har her ulike måter å representere den samme operasjonen på. Javakoden vil vi lese: "variabel i er lik variabel j pluss variabel k". Noe forenklet vil vi lese bytekoden "hent verdi fra minnelokasjon 2, hent verdi minnelokasjon 3, legg sammen verdiene og legg summen i minnelokasjon 1". Dette er også et eksempel på det vi kaller det *semantiske gapet* mellom høynivåspråk og maskinkode/bytekode. Programmereren forstår hva kodelinjen " $i=j+k$ " skal gjøre, men hvordan denne faktisk utføres på maskinen har hun sjelden noen ide om.

I datamaskinens barndom måtte programmereren selv legge instruksjonene direkte inn i primærlageret ved hjelp av brytere. Siden utviklet man særskilte programmer, såkalte assemblere, som oversetter symbolske navn for instruksjonene til de tilsvarende bitmønstrene.

Dagens assemblere vil i tillegg til å oversette de symbolske navnene (mnemonics) til instruksjonene til maskinkoder også holde rede på minneadresser. Dermed kan vi også benytte symbolske navn for disse. Følgende eksempel viser et enkelt assemblyprogram for vår simple datamaskin. Programmet leser inn to tall, multipliserer disse og skriver ut resultatet.

```

START:    IN          ; Leser et tall fra tastaturet
          STORE      A    : Lagrer tallet i variabel A
          IN          ; Leser et tall fra tastaturet
          STORE      B    : Lagrer tallet i variabel B

LOEKKE:   LOAD       A    ; Henter A inn i (ACC)
          SUB        EN   ; A=A-1
          STORE      A    ; Lagrer A
          BRNEG      FERDIG; dersom A<0 gå til subrutine FERDIG
          LOAD       B    ; Les variabel B inn i (ACC)
          ADD        SUM  ; adder variabel B til variabel SUM
          STORE      SUM  ; lagre sum
          BRANCH     LOEKKE; gå til label LOEKKE og fortsett

FERDIG:   LOAD       SUM  ; henter variabel SUM inn i (ACC)
          OUT         ; skriver ut SUM til skjerm
          STOP        ; avslutter programmet

EN:       DW         0001; initialiser variabel EN til 1
SUM:      DW         0000; initialiser variabel SUM til 0
A:        DW         0000; initialiser variabel A til 0
B:        DW         0000; initialiser variabel B til 0

```

Linjer som skal oversettes må ha et bestemt format (syntaks), akkurat som setninger må ha en bestemt form i et høynivåspråk. Eksempelet på syntaks i figuren under er for assemblyprogrammering for Intel-prosessorer.

[label:] opkode [operand] [;kommentar]

Syntaks for programsetning i et lavnivåspråk.

En label (merkelapp) definerer adressen til en instruksjon eller til en lokasjon som benyttes til lagring av data. I eksemplet ovenfor, vil START: være det symbolske navn på programmets startadresse, mens A, B, SUM og EN er symbolske navn på lokasjoner som benyttes til lagring av data.

I eksekverbar kode er den eneste formen for navngiving som finnes minne- og register-adresser.

Assemblerens virkemåte

De fleste assemblere går gjennom kildekoden to ganger. I første gjennomløp registreres alle labler. Samtidig teller assembleren opp antall instruksjoner og finner dermed ut hvor mange minnelokasjoner som vil måtte benyttes til lagring av data og instruksjoner.

Alle labler plasseres fortløpende i en såkalt symboltabell. I denne tabellen er det plass til selve labelen, samt adressen til den instruksjon eller lagerlokasjon som labelen peker ut.

Adressen finnes ved å sette en egen teller til 0 før kildeteksten gjennomløpes. For hver instruksjon som leses, økes denne teller med 1.

I andre gjennomløp oversettes hver instruksjonene til det korresponderende bitmønseret (opkoden). Hvis en label nå påtreffes i høyre kolonne, vil assembleren slå opp i symboltabellen og bytte ut labelen med den tilsvarende adresse hentet fra tabellen.

La oss nå gå gjennom eksempelet vist ovenfor på nytt og se hvordan en assembler vil behandle denne kildekoden.

Første gjennomløp

SUM plasseres i symboltabellen sammen med verdien til teller som nå er 16, teller økes

Linje lest: START: IN

Dette fører til at START blir plassert i symboltabellen. Telleren er 0, og denne verdien plasseres i tabellen sammen med START. Så økes telleren.

Linje lest: STORE A Ingen label, teller økes til 2

Linje lest: IN Ingen label, teller økes til 3

Linje lest: STORE B Ingen label, teller økes til 4

Linje lest: LOEKKE:LOAD A

LOEKKE plasseres i symboltabellen sammen med verdien til telleren som nå er 4. Telleren økes til 5.

Linje lest: SUB EN Ingen label, teller økes til 6

Linje lest: STORE A Ingen label, teller økes til 7

Linje lest: BRNEG FERDIG Ingen label, teller økes til 8

Linje lest: LOAD B Ingen label, teller økes til 9

Linje lest: ADD SUM Ingen label, teller økes til 10

Linje lest: STORE SUM Ingen label, teller økes til 11

Linje lest: BRANCH LOEKKE Ingen label, teller økes til 12

Linje lest: FERDIG:LOAD SUM

Ferdig plasseres i symboltabellen sammen med verdien til teller som nå er 12. Telleren økes til 13.

Linje lest: OUT Ingen label, teller økes til 14

Linje lest: STOP Ingen label, teller økes til 15

Linje lest: EN: DW 0001

EN plasseres i symboltabellen sammen med verdien til teller som nå er 15, teller økes til 16.

til 17.

Linje lest: SUM: DW 0000

Linje lest: A: DW 0000

A plasseres i symboltabellen sammen med verdien til teller som nå er 17, teller økes til 18.

Linje lest: B: DW 0000

B plasseres i symboltabellen sammen med verdien til teller som nå er 18, teller økes til 19.

Første gjennomløp er nå avsluttet, og symboltabellen ser ut som følger:

Symbol	Verdi
START	0
LOEKKE	4
FERDIG	12
EN	15
SUM	16
A	17
B	18

Tabell 7: Symboltabell

Andre gjennomløp

Assembleren går nå gjennom kildekoden på nytt linje for linje og oversetter hver enkelt instruksjon til tilsvarende bitmønster. Når f.eks. linjen

```
BRNEG  FERDIG
```

leses, finner assembleren først fram til opkoden i tabellen over instruksjoner. I dette tilfelle er opkoden 0111. Så henter den fram adressen til labelen FERDIG fra symboltabellen (adressen er 12 desimalt, 01100 binært) og setter dette sammen til maskinkoden: 0111 0000 0000 1100.

Vi har nå fått fram en eksekverbar instruksjon bestående av en instruksjonskode (opkode) og en adressedel.

Disse ferdige instruksjonene lagres i primærlageret eller på fil, og kan siden utføres av maskinen.

Virkelig assmeblykoding av programmer er ikke så mye mer komplisert enn det vi har sett i dette eksempelet. I våre dager brukes slik lavnivåprogrammering først og framst til å skrive drivere og annen systemprogramvare. Applikasjoner skrives nå nærmest utelukkende i høynivåspråk.

Operativsystemnivået

Vi har nå sett på hvordan logiske porter og registre kan settes sammen til en CPU. Og vi har sett på hvordan denne CPU'en kan programmeres ved hjelp av et instruksjonssett. Vi kan da bevege oss et nivå til oppover i den funksjonsorienterte modellen og se på operativsystemet.

Operativsystemer er et sentralt tema i kursene TK200 og TK300. Her vil vi bare gi en første smakebit på stoffet.

Hensikten med operativsystemet

Brukerne av maskinen vil ha sine oppgaver utført. De vil kjøre applikasjonene sine og de vil ha en maskin som er effektiv, sikker og enkel å bruke. Ideelt sett tar operativsystemet seg av "det kompliserte" og gjør alle maskinvaredetaljene usynlige for brukeren. Dette kan sammenlignes med at en bilist ønsker å kjøre bil og er oftest ikke interessert i hvordan forgasseren, kardangen, bremselysene eller girboksen fungerer. Men på samme måte som disse komponentene er vesentlig for kvaliteten på bilen, vil operativsystemet være avgjørende for kvaliteten på tjenestene datamaskinen tilbyr.

Maskinen fungerer bare godt i den grad hukommelse, CPU og eksterne enheter som tastatur, skjerm og harddisk samvirker smertefritt. Når vi lager en applikasjon slik som en tekstbehandler, må det programmeres at programmet skal lastes inn i hukommelsen, at det skal lese fra tastatur, skrive til skjerm, og at hver instruksjon skal sendes til prosessoren. Programmering på instruksjonsnivået er komplisert og avhengig av utstyret maskinen består av.

Operativsystemet er programvare på dette nivået og tilbyr programmereren et forenklet grensesnitt mot maskinvaren. Vi kan si at applikasjonen og maskinvaren kommuniserer med hverandre gjennom operativsystemet. At operativsystemet er der for å skjule hvor komplisert maskinvaren i realiteten er.

Operativsystemets oppgaver er:

- Å kjøre programmer, dvs. laste programmer inn i hukommelse, sende instruksjoner til prosessoren, motta input fra tastatur og skrive resultater til f.eks skjerm.
- Å tilby et brukergrensesnitt, dvs. å gi brukeren muligheter til konfigurere maskinen, sette i gang applikasjoner o.l.
- Å administrere devicer/enheter, dvs. å konfigurere og benytte drivere for periferienheter koplet til maskinen. Det kan være en mus, skriver, harddisk, skjerm, nettverkskort o.l..
- Å (ideelt sett) tilby ressursdeling på en rettferdig og effektiv måte Å sikre at ikke det ikke oppstår feil, eventuelt takle disse feilene på en god måte.
- Å tilby sikkerhet for hver enkelt bruker og sikre at programmer ikke ødelegger for hverandre.

Operativsystemet kan sees på som en regjering som skal fordele ressurser på en rettferdig og effektiv måte. Ressurser er alt som et program trenger for å kunne utføre sine oppgaver. Under denne fordelingen må det tas hensyn til de ulike applikasjonenes behov for prosessortid, minneplass og periferutstyr.

Historikk

Operativsystemer har blitt utviklet i takt med maskinvareutviklingen og programvarebehov. For å få et innblikk i hvilke oppgaver operativsystemet har utført og hvordan utviklingen har skjedd, skal vi se på det i historisk perspektiv. Det er nyttig å bruke litt tid på historien her fordi moderne operativsystemer er et resultatet av denne utviklingen. Den funksjonaliteten de tilbyr har blitt lagt til trinnvis ut fra opplevde behov og muligheter.

Batch-systemer

De første datamaskinene var store og dyre. Prosessortiden var kostbar og CPU-tid var den viktigste ressursen å utnytte på best mulig måte. Data og instruksjoner ble lest inn i maskinen fra hullkort. Deretter ble de lastet inn i hukommelsen og beregninger foretatt. Resultatet ble sendt til en utenhet som igjen produserte hullkort eller utskrifter. Operativsystemet foretok innlesing, utskrift, minnehåndtering og dataforsendelse mellom hukommelse og prosessor. Hver jobb ble lastet, utført og avsluttet før neste kunne begynne. Resultatet var at prosessoren stadig måtte vente mens neste jobb ble lest inn. Dette medførte mye dødtid.

I batch-systemer var jobbene satt sammen i bolker. Til hver jobb ble det sendt med nok instruksjoner til at jobben kunne fullføres. Man kunne f.eks bolke sammen alle FORTRAN-programmer å kompilere disse etter hverandre. Fordelen med dette var at FORTRAN-kompilatoren da bare behøvde å lastes inn en gang den dagen. System-softwaren kaltes *monitoren*. Denne leste og tolket instruksjonene knyttet til batch-jobbene.

Fordeler med batch-systemene var:

- Arbeid ble overført fra operatør til datamaskin
- Økt ytelse siden en ny jobb kan starte så snart en annen er ferdig

Ulemper var:

- Det var vanskelig å feilsøke (debugge) programmer
- Det var ingen beskyttelsesmekanismer slik at en feil ved et program kunne påvirke de andre (lese for mange kort f.eks).
- En jobb kunne lage feil i monitoren slik at andre jobber ble påvirket
- En jobb kunne komme i evig løkke

For å løse disse problemene måtte det lages beskyttelsesmekanismer for hukommelsen slik at monitor og brukerprogrammer ble adskilt og beskyttet fra hverandre. Problemet med at feil i et program påvirket et annet kunne løses ved at enkelte oppgaver bare kunne utføres av monitoren. For å forhindre evige løkker ble en timer lagt til systemet slik at man kunne legge inn maksimumstider for hver jobb og på den måten avbryte evige løkker.

Spooling

De mekaniske delene for innlesing og utskrift var mye tregere enn CPU'en så prosessoren ble dårlig utnyttet. Da magnetisk tape og annen lagringsteknologi kom på markedet, ble situasjonen noe bedre og det ble utviklet teknologi slik at en enhet leste data inn til en tape samtidig som prosessoren jobbet med en annen jobb. Resultatet av jobbene ble også lagret i kø på tape eller harddisk og ble så behandlet av ut-enhetene i deres eget tempo mens prosessoren kunne gjøre andre ting.

Dette ble kalt spooling, **Simoultaneous Peripheral Operation On-Line**. I stedet for at prosessoren måtte vente på at printeren ble ferdig med en utskriftsjobb kunne den starte på neste jobb.

En fordel med dette var at man fikk resultatet av en jobb uten å måtte vente på at hele bolken av jobber var ferdig prosessert. Ulempen var fortsatt var at siden databehandlingen ikke foregikk i sanntid, var det ikke mulig å kommunisere med et program som ble utført.

Multiprogrammerte batch-systemer

Siden jobbene nå ble lest inn og lagt i kø, kunne prosessoren velge mellom flere innleste jobber. Det ble mulig å laste flere jobber inn i hukommelsen. Dersom en jobb ventet på en utskriftsdel, kunne prosessoren jobbe videre med en annen jobb. Dette kaltes multiprogrammering og gjorde at prosessorutnyttelsen ble bedre. Hvordan jobb-utvelgelsen skulle foretas var en avgjørelse som måtte programmeres inn i operativsystemet.

Operativsystemet fikk nå mer å gjøre. Det måtte holde styr på:

- Antall jobber i køen
- Utvelge jobber for kjøring på prosessoren
- Hukommelseshåndtering
- Skifte mellom de ulike jobbene som behandles

En av de viktigste oppgavene til operativsystemet i denne og senere perioder var å holde rede på hvor mye prosessortid den enkelte brukeren tok opp. Det var dette vedkommende ble fakturert ut fra.

Tidsdelingssystemer/multitasking og interaktive systemer

Etter hvert ble det utviklet nye metoder å sende data til datamaskinen på. Man fikk tastatur istedenfor hullkort-lesing og skjerm for å presentere resultatet. Nå ble det mulig å sende kommandoer rett til maskinen. Det ble også mulig å sende data *under* programutføringen og man kunne dermed lage interaktive programmer. Det gjorde det lettere å debugge programmer og det ble mulig å oppgi verdier til programmene underveis.

Ulempen med tastaturet var at det var nødvendig å sitte foran maskinen som skulle kjøre programmet. For å bedre på dette ble det utviklet terminaler. Terminalene bestod av skjerm og tastatur koblet til den sentrale maskinen. Mange brukere kunne kople seg til en maskin med hver sin terminal samtidig. Programmene var interaktive under kjøringen. Det betydde også at et program ofte var passivt mens det ventet på input fra brukeren. For å få maksimal utnyttelse av prosessoren og rettferdig betjening av brukerne ble det utviklet tidsdelings-systemer. De ulike jobbene fikk avsatt en viss tid i prosessoren før de ble erstattet med neste jobb i køen. For brukerne vil det likevel se ut som de er alene på maskinen siden prosessorene arbeidet så pass raskt med hver enkelt jobb. Så hvis ikke det var for mange samtidige brukere som kjørte alt for tunge programmer var responstiden likevel god.

Operativsystemets nye oppgaver var:

- å holde rede på de ulike jobbene
- passe på tidsbruk på prosessoren
- holde rede på jobbkøen

- håndtere input fra brukerne under programutførelsen.

For å håndtere situasjoner som input fra brukeren under programutførelsen ble det innført spesielle signaler, *interrupts*. Når det kom et interrupt var det beskjed til prosessoren om at jobben trengte noe spesielt og de nødvendige rutiner ble igangsatt. Interruptene ble tildelt ulik prioritet, ofte slik at interaktive programmer ble favorisert.

For at brukerne skulle ha tilgang til data og programmer ble det nødvendig med lagring av disse på disken. Det krevde et filsystem. Et annet nytt problem var at responstiden for de aktive brukerne måtte være lavest mulig, og det hendte at primærminnet ble fullt slik at jobber måtte lastes ut av minnet til disken og tilbake. Dette kaltes *swapping*. Det av disken som benyttes som reserveminne kaltes virtuell hukommelse. Virtuell hukommelse gjorde det blant annet mulig å kjøre større programmer enn det som var mulig tidligere.

Personlige datamaskiner og arbeidsstasjoner

Dersom en datamaskin bare skal utføre en type oppgaver trengs det strengt tatt ikke noe operativsystem. En enkel lommekalkulator er en datamaskin god som noen, men all programvaren den kjører er lagret i ROM og et operativsystem ville vært sløsing.

De tidligste PC'ene hadde ikke operativsystem, eller de hadde et minimalt system kodet i ROM. Da diskettstasjoner og annet periferiutstyr ble tilgjengelig også for slike hobbymaskiner kom behovet for å forenkle bruken av slikt utstyr for programmerer og bruker.

Gary Kildall utviklet et tidlig operativsystem for PC'er som han kalte CP/M. I 1977 skrev han en versjon av dette systemet for produsenten IMSAI. I denne versjonen tok han alle de utstyrspesifikke rutinene ut av selve operativsystemet og la dem på en ROM-brikke. Denne delen kalte han BIOS (Basic Input Output System). På denne måten var det kun BIOS som måtte endres dersom man gikk over til en ny type disk. CP/M var det samme for alle mulige maskin-konfigurasjoner.

MS-DOS (MicroSoft Disk Operating System) var et annet PC-operativsystem. MS-DOS var ikke et multitasking-system. Dvs. at det ikke var mulig å utføre flere programmer "samtidig". Tanken var at det bare var en bruker av maskinen som kun kjørte et program om gangen og dermed var det ikke behov for å dele prosessoren mellom flere jobber. Det var heller ikke mulig å beskytte operativsystemet fra brukerprogrammene.

Minnehåndteringen var primitiv og operativsystemet kunne bare håndtere en viss minnestørrelse. Det var ingen innebygde beskyttelsesmekanismer i filsystemet så virusprogrammer og lignende kunne lett spre seg og gjøre store ødeleggelser.

Etter at PC-bruk eksploderte på åttitallet gjorde behovet for større funksjonalitet og brukervennlighet seg raskt gjeldende. Windows 3.1 var et grafisk brukergrensesnitt (GUI) mot maskinen som i tillegg tilbød bedre funksjonalitet. Fortsatt var MS-DOS operativsystemet på maskinen.

Windows95/98/NT/2000 er etterfølgende operativsystemer for PC. Vi skal se nærmere på disse etterhvert.

Før vi går over på eksempelsystemene skal vi kort nevne noen andre relevante typer systemer.

Parallelle systemer

For å få kraftigere maskiner er det mulig å bygge maskiner med flere prosessorer. Dersom disse prosessorene deler klokke, hukommelse m.m kalles det et *tett koblet*

system. Dersom de samarbeider, men ikke har delte ressurser kalles de et *løst koblet* system.

Det er flere grunner til å lage flerprossessor-systemer:

- Mer datakraft
- Billigere enn flere maskiner
- Bedre sikkerhet og feiltoleranse, går den ene prosessoren i stykker kan den andre overta.

Distribuerte systemer

I distribuerte systemer er det flere maskiner som samarbeider om en gitt oppgave. Hver maskin er selvstendig med egen hukommelse klokke etc. Dette er transparent for brukeren dvs. at brukeren ikke merker at det er flere maskiner som tar seg av databehandlingen.

Hensikten med et slikt system kan være:

- Ressursdeling. En maskin kan være server med programvare, en annen kan være filtjener etc.
- Økt prosesseringshastighet. Maskinen kan dele på arbeidsbyrden.
- Økt pålitelighet. Hvis en maskin er nede kan en annen benyttes.
- Kommunikasjon som epost og deling av dokumenter.

I løpet av 1970-tallet ble slike nettverksfunksjoner en ny viktig del av operativsystemer.

Sanntidssystemer

I en del systemer er responstid svært kritisk. Det kan for eksempel være for en maskin koblet til måleinstrumenter som måler temperatur i kjølevannsystemet på en atomreaktor. Hvis instrumentene når en gitt temperatur må responsen fra maskinen skje på en veldefinert og som regel svært kort tid. For å sikre dette kan man f.eks ikke ha bruk av virtuelt minne, vanlig multitasking e.l.

Vi skiller mellom harde og myke sanntidssystemer der harde sanntidssystemer har absolutte krav til responstid, mens myke er noe mer fleksible. Den modulen i operativsystemet som står for interaksjonen med brukeren vil typisk være et mykt sanntidssystem.

Moderne operativsystemer er myke sanntidssystemer i det at de gir brukeren umiddelbar respons, parallellt i den grad de støtter flere prosessorer, og distribuerte i den grad de støtter nettverksdrift.