TDT4160 Datamaskiner grunnkurs

Oppsummering

Vegard Aas, 2005

1. Introduksjon

1.1 Structured computer organization

Datamaskin
En maskin (elektronisk, mekanisk, biologisk etc.) som tar inn en form for data, prosesserer denne og gir ut behandlede data.

1.1.1 Maskinspråk
Primitive instruksjoner
Hva en datamaskin egentlig kan: legge sammen to tall, se om et tall er 0 eller ikke, kopiere data fra et sted til et annet.

Program
En sekvens av instruksjoner som beskriver hvordan en bestemt oppgave skal utføres

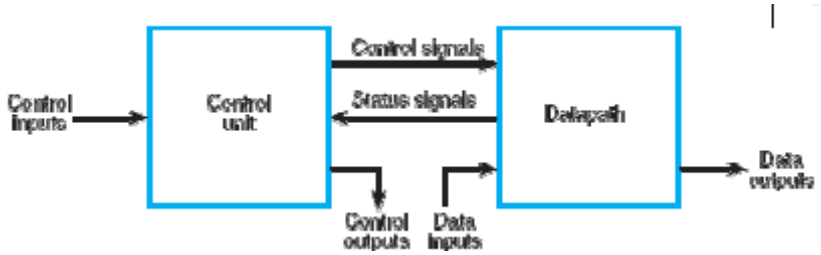
Maskinsipråk
Utgjøres av en datamaskins primitive instruksjoner og er måten mennesker kan kommunisere med datamaskinen på.

1.1.2 Strukturert datamaskinorganisering
- Hierarki av abstraksjoner ved hjelp av *oversetting* (hele programmet oversettes til lavnivåspråk) eller *tolking* (en og en instruksjon oversettes hver gang programmet skal utføres)
- L0, maskinspråk: Det datamaskinen kan
- L1 Nærmere hvordan mennesker tenker, alle L1-instruksjoner kan uttrykkes vha L0
  Forstås lettere som en virtuell maskin som benytter dette språket

N nivåer

Hvorfor ikke lage en maskin der L4 er maskinspråk?
*K*ompleks og dyr maskinvare, vanskelig å rette feil i maskinvare og blir da ikke støtte for flere språk.

Nivåer i dagens maskiner
Nivå 0: Digitale kretser: Fundamentale komponenter (AND, OR, vipper), sammensatte
Nivå 1: Mikroarkitektur: Styreenhet (mikroprogram eller maskinvare), utførende enhet (registere+ALU) komponenter
Nivå 2: Instruksjonssettarkitektur (ISA): Maskinkode, Første nivå tilgjengelig for brukere, grensesnitt mellom maskin- og programvare
Nivå 3: Operativsystem: Hybridnivå språk fra niv2, støtte for flere programmer, virtuelt minne, filsystem
Nivå 4: Assemby (Dark): Symbolsk språk ADD R1, R2, R3
Nivå 5: Høynivå programspråk (Java, C etc.): naturlig språk, kompilert til maskinkode



1.1.3 Maskin- og programvare
- Opprinnelig to nivåer: ISA og digitale kretser: Dyr og komplisert maskinvare for å implementere ISA-Instruksjoner
- 1951: Introduksjon av mikroprogrammering: et mikroprogram for hver ISA-instruksjon, maskinvare erstattet av programvare, enklere og billigere
- Vha. mikroprogram kan instruksjonssett utvides uten ny maskinvare, fra 1970 større instr. sett
- Gav kompliserte og trege mikroarkitekturer, spesialiserte instruksjoner som ble lite brukt, nå: enkelt

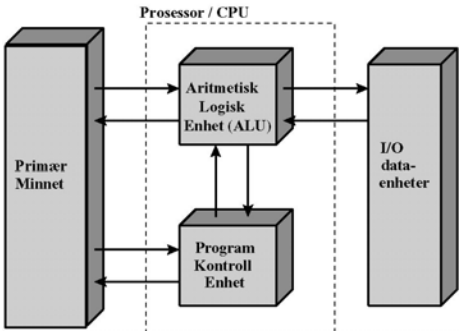
1.2 Historikk: Generasjoner

1.2.0: Mekaniske datamaskiner (1642-1945)
Mekanisk regnemaskin, Blaise Pascal 1642
Differensieringsmaskin, Charles Babbage, addisjon og subtraksjon
Analytisk maskin, programmerbar, lager, I/O, fungerte aldri

1.2.1: Radiorør (1945-1955)
2. verdenskrig katalysator for utvikling
Kodeknekking: Colossus (England) 1943
Tabeller for artilleri: Electronic Numerical Integrator And Computer (USA) 1946, bryterprogrammert

Von Neumann-arkitektur
ENIAC og EDVAC ble programmert av brytere, von Neumann fant ut at programmer kan lagres binært akkurat slik som data. Kan da lese instruksjoner fra lager og utføre disse.
Fire deler:

Primærminnet:	Inneholder data og instruksjoner
Kontrollenhet:	Tolker instruksjoner og sørger for at de blir utført
ALU:	Aritmetisk og logisk enhet som utfører beregningene
I/O:	Enheter for inn- og utdata



1.2.2: Transistorer 1955-1965)
Transistor oppfunnet 1948, gjør datamaskiner mer pålitelige og tar mindre plass.
Enkeltbusser ble tatt i bruk for å koble komponentene sammen vha. parallelle kabler

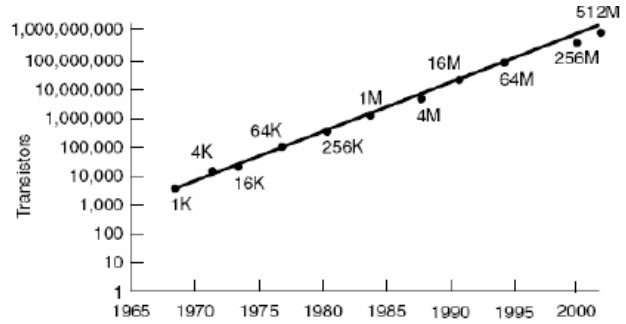
1.2.3: Integrerte kretser (1965-1980)
Integrerte kretser på silisiumbrikke tillot mange transistorer på en brikke

1.2.4: VLSI: Very Large Scale Integration (1980-)
Flere titusener av transistorier på en brikke, muliggjør personlige datamaskiner
GUI ble innført, RISC (reduced instruction set computer) tok over for CISC (Complex)

1.2.5: Integrerte datamaskiner
PDAer, klokker, kredittkort
Spesialtilpasset maskin- og programvare

1.3 Moores lov

”Antall transistorer på en krets dobles hver 18. måned”



1.4 Datamaskinsystemer

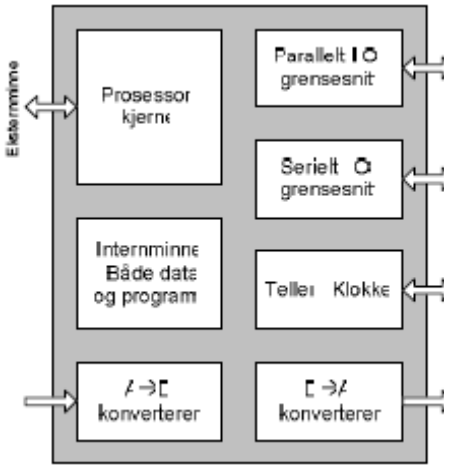
- *Engangsdatamaskiner* Bursdagskort, RFID etc.
- *Mikrokontroller* Embedded systems; kommunikasjon, klokker, biler
- *Spillmaskin* Optimalisert for underholdning og spill ift grafikk, ytelse og lyd
- *PC* Desktop eller laptop
- *Server*
- *Workstation* Spesialtilpasset arbeidsmaskin
- *Stormaskin*
- *Superdatamaskin*

1.5 Datamaskin/prosessor-familier

Intel Pentium
Fullt bakoverkompatibel til 8086 som var første 16-bitrs CPU
Pentium 4 nyeste og siste (?) x86 prosessor, kan ikke lage ny ISA
Problemer med varmeutvikling, og har derfor flere CPU på en chip i stedet for høyere klokkefrekvens
CISC: Complex Instruction Set Computer

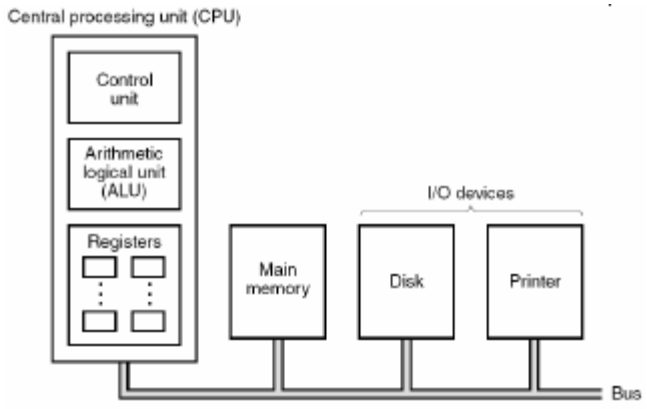
Chip	Date	MHz	Transistors	Memory	Notes
4004	4/1971	0.108	2300	640	First microprocessor on a chip
8008	4/1972	0.108	3500	16 KB	First 8-bit microprocessor
8080	4/1974	2	6000	64 KB	First general-purpose CPU on a chip
8086	6/1978	5–10	29,000	1 MB	First 16-bit CPU on a chip
8088	6/1979	5–8	29,000	1 MB	Used in IBM PC
80286	2/1982	8–12	134,000	16 MB	Memory protection present
80386	10/1985	16–33	275,000	4 GB	First 32-bit CPU
80486	4/1989	25–100	1.2M	4 GB	Built-in 8-KB cache memory
Pentium	3/1993	60–233	3.1M	4 GB	Two pipelines; later models had MMX
Pentium Pro	3/1995	150–200	5.5M	4 GB	Two levels of cache built in
Pentium II	5/1997	233–450	7.5M	4 GB	Pentium Pro plus MMX instructions
Pentium III	2/1999	650–1400	9.5M	4 GB	SSE Instructions for 3D graphics
Pentium 4	11/2000	1300–3800	42M	4 GB	Hyperthreading; more SSE instructions

UltraSparc III
SPARC: Scalable Prosessor ARChitecture
RISC: Reduced Instruction Set Computer
Arkitektur laget av Sun Microsystems
64 bits prosessor: 64 bits registere og adresser til minnet, ALU utfører beregninger på 64b



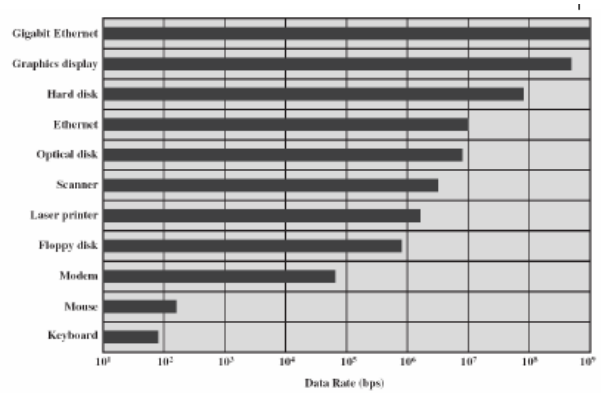
8051
Mikrokontroller
Programvare typisk fast og lagret i ROM
Treg men billig

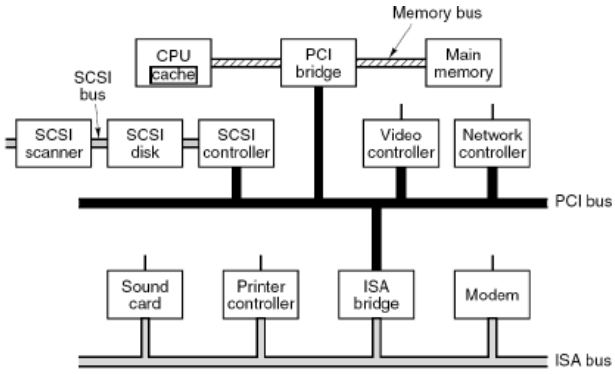
2. Organisering av datamaskinsystemer



2.1 Busser

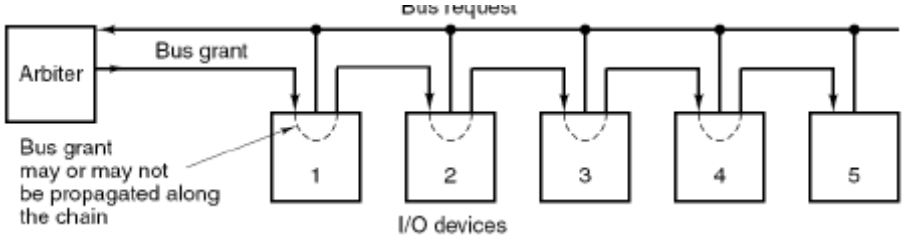
- Vanligste sammenkoblingsstruktur
- Metalliske ledere som kobler sammen to eller flere enheter
- Består av adresselinjer, datalinjer og kontrollinjer
- Kun en enhet kan sende om gangen (master/slave), arbitrering bestemmer busmaster
- Interne eller eksterne
- Opprinnelig en buss, nå hierarki på grunn av ulik hastighet, prioritet og avstand fra CPU



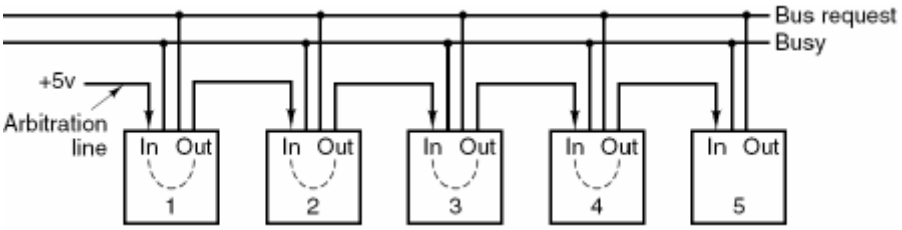


2.1.1 Arbitrering

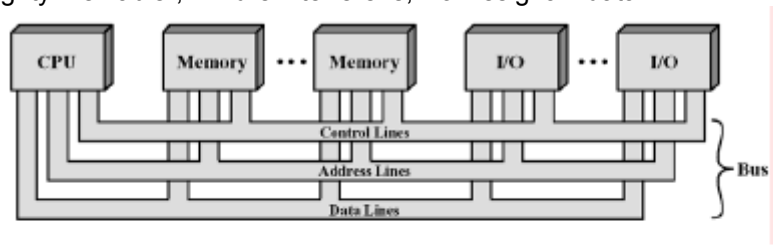
Sentralisert arbitrering
En egen arbitreringsenhet velger master, aktiverer grant ved request (dersom bus er ledig)
Når enhet mottar grant: bruker buss eller sender videre (jo nærmere arbitreringsenheten jo høyere pri)



Desentralisert arbitrering
Enhetene velger master i fellesskap
Ønsker ikke bruke buss: arbitrering inn = arbitrering ut
Ønsker å bruke buss: deaktiver arbitrering ut
Kan ta buss hvis: buss ledig, aktiv arbitrering inn
Enhet til venstre har høyest prioritet



2.1.2 Parallell vs seriell overføring
Parallell kan overføre flere bit om gangen og har tradisjonelt blitt brukt for raske enheter
Problemer: tykke kabler, høy kostnad, interferens, synkroniseringsproblem
Seriell mer vanlig: tynne kabler, mindre interferens, klokkesignal i data

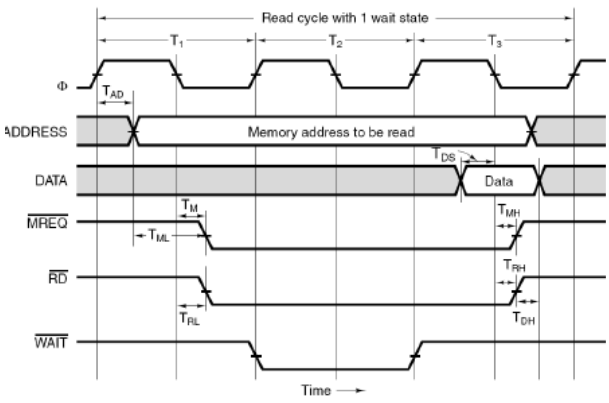


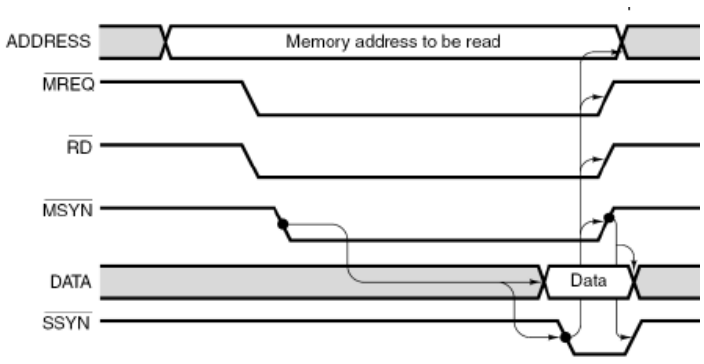
- Antall adresselinjer: Hvor mange lokasjoner kan være mål/kilde (jmf: antall siffer i postnummer)
  - Eks: Størrelse på hovedlager for buss CPU⇔Hovedlager
- Antall datalinjer: Hvor mange bit kan overføres i parallell
  - Øker ytelse, men bare til et visst punkt

2.1.3 Synkron vs asynkron

Synkron
Inkluderer klokkesignal
Alle hendelser koordinert med klokke
+ Lite logikk
+ Ingen overhead
- Hastighet bestemmes av tregeste enhet
- Rask klokke krever korte busslinjer

Asynkron
Overføring ikke knyttet til klokke
En hendelse starter når forrige er ferdig
+ Kan betjene både raske og trege enheter
+ Tåler lange busslinjer
-Protokoll nødvendig for å kommunisere
- Kompleks og dyr

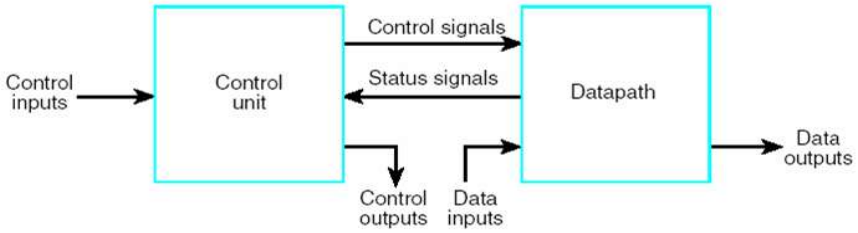




Programstyrt I/O	Prosessor kjører program med I/O instruksjoner Må sjekke I/O-enhet ”ofte nok” til at data ikke mistes, aktiv venting (busy loop), tar mye tid Brukes bare dersom man har dedikert prosessor
Avbruddsinitiert	I/O-enhet sier fra når data er klare, prosessor stopper og behandler avbrudd
DMA	Mye dataoverføring gir prosessor lite tid til annet Direct Memory Access, egen kontroller som overfører direkte til/fra hovedlager Prosessor håndterer kun oppstart av overføring, men kan da ikke aksessere hl

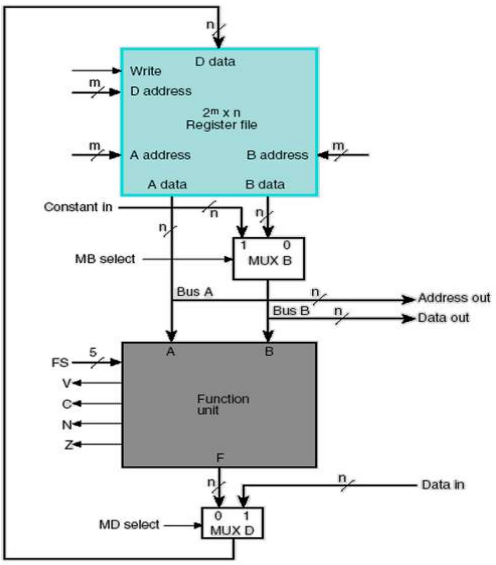
2.2 Prosessor

- Hjernen i datamaskinen, kompleks enhet som utfører algoritmer spesifisert ved lagret program
- *Styreenhet (control unit):* Henter instruksjoner fra hovedlageret og styrer utførende enhet
- *Utførende enhet:* Data flyter gjennom fra og til registre og blir behandlet av ALU



2.2.1Utførende enhet (datapath)

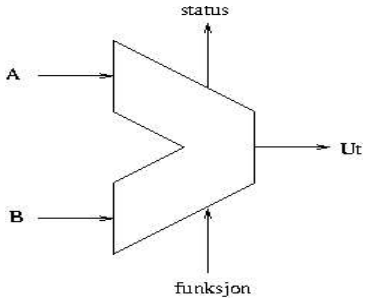
- ALU prosesserer data
- Registre (PC peker til neste instruksjon, IR inneholder instruksjon som utføres)
- Synkron enhet: Klokkesignalet veksler mellom verdiene 0 og 1, og en slik veksling kalles en klokkesyklus, og det som skjer innen denne går i parallell.



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): 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

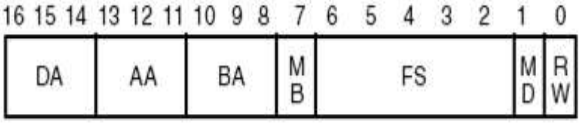
Null (Z): Forteller av svaret er 0

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 Dinngangen 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:

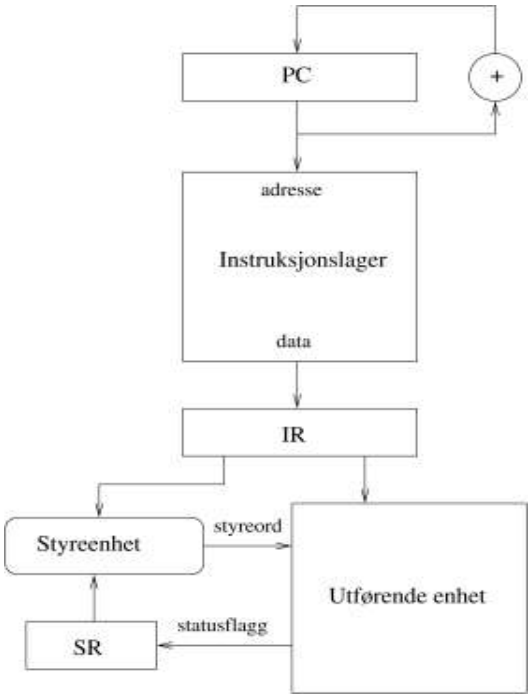
DA: Registeradresse til registerblokkens skriveport
AA: Registeradresse til registerblokkens leseport A
BA: Registeradresse til registerblokkens leseport B
MB: Kontrollsignal til MUX B (velger inngang)
FS: Velger funksjon som skal utføres av ALU
MD: Kontrollsignal til MUX D
WR: Angir om data skal skrives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



Mikrooperasjoner
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). Registre skrives på denne måten: Rn, 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.



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.

Instruksjonsutføring
1. Hente neste instruksjon fra hovedlager til instruksjonsregister vha. adresse i PC
2. Oppdatere programteller (PC)
3. Finne ut hvilken instruksjon som er hentet (dekoding)
4. Hvis instruksjon trenger data fra hovedlager, finn adresse
5. Hent data fra hovedlager hvis nødvendig
6. Utfør instruksjon
7. Gå til steg 1

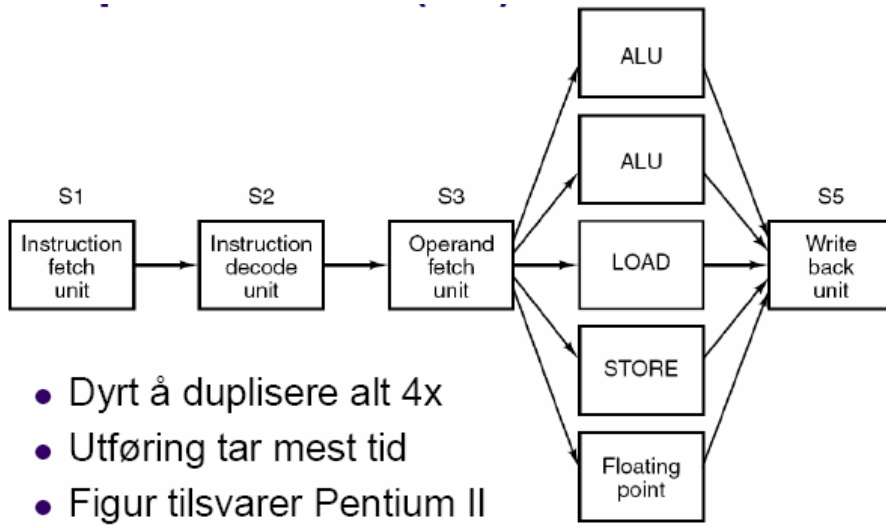
RISC: Reduced Instruction Set Computer
- Kompliserte mikroprogram var vanskelige å utnytte og ga treg utføring
- Utfører instruksjoner med maskinvare, ingen mikroprogram
- Fokus på hyppig oppstart av instruksjonutføring
- Enkel dekoding: fast instruksjonslengde, få format
- Kun få instruksjoner kan aksessere hovedlager: load/store
- Mange registre – reduserer bruk av hovedlager

Parallellitet
- Essensielt for å øke ytelse
1) Instruksjonnivåparallellitet: Flere instruksjoner utføres samtidig av 1 CPU → samlebånd
   Superskalaritet: duplisering av CPU-komponenter
2) Flere prosessorer

Instruksjonssamlebånd
- Instruksjoner deles opp og utføres av dedikert maskinvare
- Gitt 5 steg, hvert steg tar en klokkesyklus, instr. 1: 5 klokkesykler, deretter 1 instr. ferdig pr syklus
- Klokkesyklene kan gjøres mye kortere, bestemmes av tregeste steg.

- Maksimal instruksjonshastighet er gitt av 1/(største forsinkelse i ns)
- Syklustid: $\tau = \max[\tau_i] + d = \tau_m + d \approx \tau_m$
- Tidsforbruk for utføring av *n* operasjoner i *k*-stegs samlebånd: $T_k = [k + (n-1)]\tau$
- Forbedring fra uten samlebånd: $\frac{T_1}{T_k} = \frac{nk\tau}{\tau[k+(n+1)]} = \frac{nk}{[k+(n+1)]} \approx k$
- Forutsetter at man vet adressene til instruksjonene som skal utføres, ikke mulig ved hopp
- Lagrer tabell med om de siste instr. innebar hopp og hva slags instr. som fører til hopp

Superskalaritet
- Bruker flere samlebånd, f.eks ved å hente to instr. om gangen og duplisere steg 2-5
- Gir opptil 10 instruksjoner under utføring samtidig
- Dyrt å duplisere alle enheter, utføring tar mest tid:



Problemer:
- Sanne dataavhengigheter (Read After Write)
- Utavhengigheter (Write After Write) : Registeromdøping: duplisering
- Antiavhengigheter (Write After Read): Registeromdøping: duplisering
- Ressurskoonflikter

Bruker scoreboard som oversikt med instruksjoner under utføring og bruk av registre for å vite når prosessoren skal vente

Prosessornivåparallellitet

- SISD – Single Instruction Single Data	Vanlige datamaskiner
- SIMD – Single Instruction Multiple Data	Samlebånd
- MISD – Multiple Instruction Single Data	Finnes ikke i praksis
- MIMD – Multiple Instruction Multiple Data	Multiprosessorer og multidatamaskiner

2.2.2 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 sykelen instruksjonen kom til instruksjonsregisteret (IR). Andre instruksjoner kan være mer komplekse, og bestå av flere mikroperasjoner. Er det tilfelle må styreenheten bruke flere sykler på å gjøre seg ferdig med instruksjonen, og sette opp forskjellige styreord for hver sykel.

Fast- 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. Etter hvert 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.

2.3 Lagerhierarki

Ønsker å ha et stort, raskt og billig lager. Oppnås ved å ha ulike nivå med forskjellig hastighet og størrelse; et lite og raskt lager (registre og hurtigbuffer) nær prosessoren som inneholder det meste den trenger. Når det trengs data fra hovedlager eller sekundærlager hentes dette inn i hurtigbufferet.

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.

Lokalitetsprinsippet
Hurtigbuffer er gjerne 0,1% av hovedlager, men har treffrate på 95%: kan forutse fremtidige lageradreser.

- 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

Hierarkiet viser hvor nær de forskjellige lagertypene er prosessoren. Øverst finnes registrene, som prosessoren kan benytte direkte i utregninger. Registre er raske, men har liten lagringskapasitet
Etter hvert som man går ned i hierarkiet:
- Minker kostnaden per bit
- Øker kapasiteten
- Øker aksesstiden
- Kommer man lenger og lenger bort fra prosessoren

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).

Lagertypenes kapasitet
- Hurtigbuffer nivå 1 være ca 8kB-32kB
- Hurtigbuffer nivå 2 ca 256kB-2MB
- Hovedlageret være mellom 128MB-4GB
- Sekundærlageret (harddisk) være mellom 30-160GB

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.

Lagertypenes ytelse

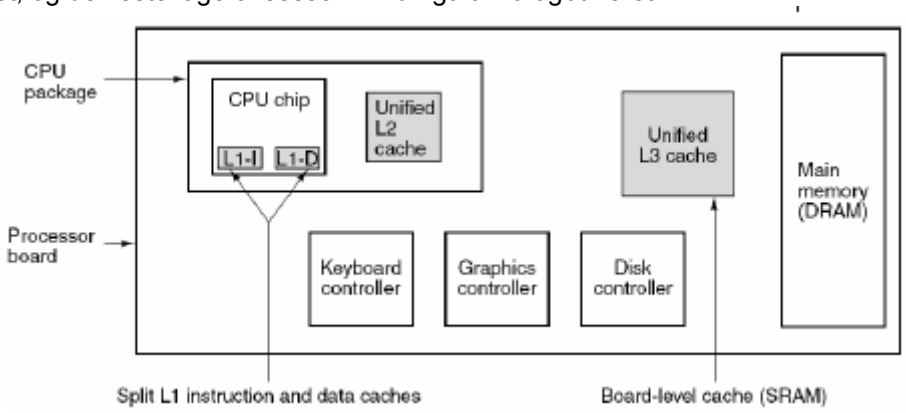
Aksesstid: For direkte aksess er dette tiden en leseoperasjon eller en skriveoperasjon tar, for andre typer aksess er det tiden det tar å sette les/skrive mekanismen på rett plass.

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/sykeltiden. For andre typer har man formelen
T(N) = Ta + (N/R)
der
T(N) er gjennomsnittlig tid for lesing eller skriving av N bit
Ta er gjennomsnittlig aksesstid, N er antall bit
R er overføringsraten i bit per sekund(bps).

2.3.1 Hurtigbuffer

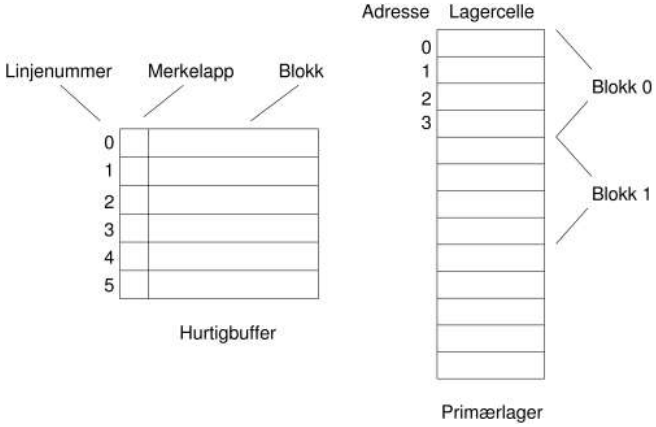
Hurtigbufferet (engelsk: cache) befinner seg mellom prosessoren og primærlageret. Når prosessoren skal hente data fra lageret gjør den oppslag i hurtigbufferet. Hurtigbufferet er mye mindre enn den totale lagerstørrelsen og inneholder derfor bare en liten delmengde av det som finnes i primærlageret.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



Gjennomsnittlig aksessid : cache aksesstid + bomrate*hl trefftid

Organisering

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).

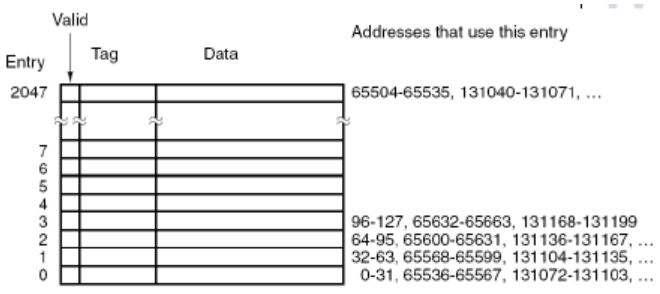


Prosessoren henter alltid data fra hurtigbufferet og aldri direkte fra primærlageret. Når prosessoren forespør en datablokk som ikke finnes i hurtigbufferet må blokken kopieres fra lageret til hurtigbufferet. Siden det er færre hurtigbufferlinjer enn blokker i lageret, trenger man en algoritme som forteller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

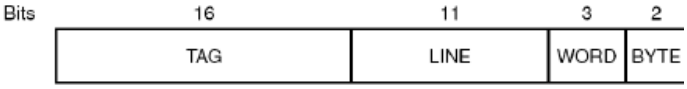
Direkte avbildning
Direkteavbildning avbilder hver blokk i primærlageret inn i en gitt hurtigbufferlinje. Hver gang denne blokken blir kopiert inn i hurtigbufferet vil den bli plassert på akkurat den samme hurtigbufferlinjen som sist.

Avbildningen skjer slik: hurtigbufferlinjenummer = 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.



- En gitt hovedlagerlinje kan bare ligge på en plass
- Flere hovedlagerlinjer kan ligge på samme plass
  - Tag ("Merkelapp") forteller hvilken av disse det er



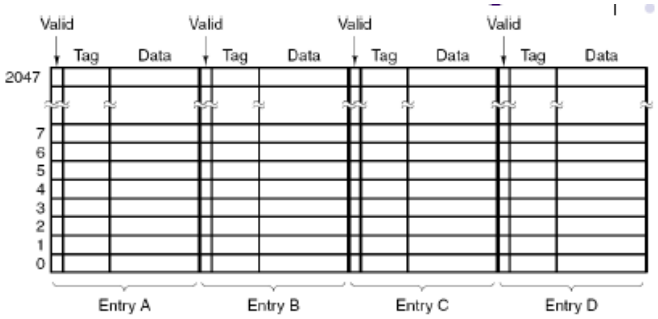
- Hovedlageradresse deles inn i 4 felt
- Eksempel:
  - Maks 4 GB hovedlager ➔ 32 bits adresse
  - 32 bits ord ➔ 2 bit BYTE-felt (byte innen ord)
  - 32 bytes linjer ➔ 3 bits WORD-felt (ord innen linje)
  - 2048 hurtigbufferlinjer ➔ 11 bits LINE-felt
  - Resten (32-2-3-11=16) brukes til TAG-felt
- Data som ligger på adresser med like LINE-felt, havner på samme hurtigbufferlinje

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.

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.



- Hver hurtigbufferlinje inneholder et sett med linjer
- Hurtigbufferlinjenr gitt av LINE-felt i adresse
- Men: Kan ligge på vilkårlig settposisjon

Organisering
Da man begynte å bruke hurtigbuffer hadde man bare et enkelt hurtigbuffer. I senere tid har det blitt vanlig med flere nivå med hurtigbuffere. 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.

Erstatningsalgortimer 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.
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.
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:
Gjennomskriving (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 (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.

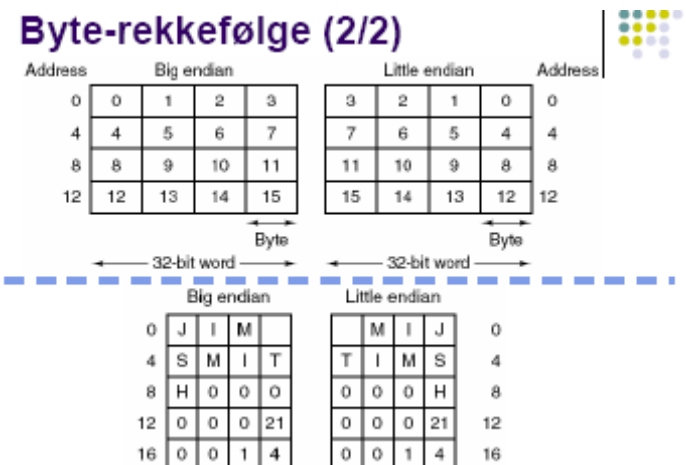
Fordelen med gjennomskriving er at man er garantert at hovedlageret alltid inneholder gyldige data (det vil si samme data som i hurtigbufferet). Ulempen er at man må gjøre oppslag i hovedlageret hver gang man skal skrive en verdi, noe som går tregt.

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.

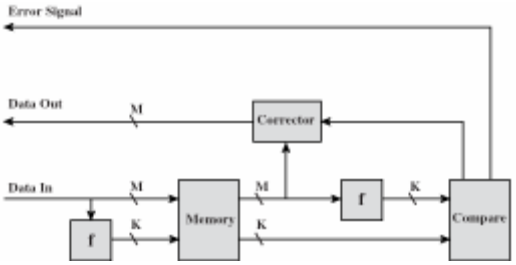
2.3.2 Hovedlager

Hovedlageradresser
- Hovedlager organisert i celler med unik adresse
- *k* bits kan adressere 2^k celler
- Celler normalt 1 byte = 8 bit
- Bytes gruppert i org, typisk 32/64 bits, instruksjoner arbeider på ord, registerstørrelse=ordstr
- 32 bits → Maks 4GB hovedlager

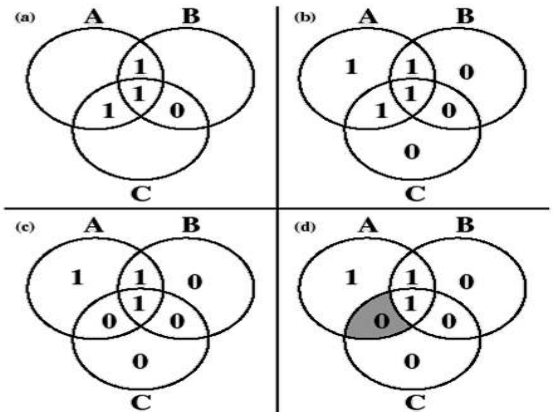
Byte-rekkefølge
- Byte kan nummereres fra venstre mot høyre (big endian) eller fra høyre mot venstre (little endian)



Feilkorrigerende kode
- Datafeil kan oppstå, for eksempel pga elektromagnetisk støy
- Feil kan oppdages via paritetsbit
- Feil kan oppdages og korrigeres via feilkorrigerende kode
- *M* bit dataord *K*-bit feilkorrigerende kode



Hammingkode
Hammingavstand *d* er antall bit som er ulik mellom to ord. For å oppdage *d* enkeltbitfeil trengs en kode med avstand *d+1*. For å rette *d* enkeltbitfeil trengs en kode med avstand *2d+1*. Paritetsbit velges slik at antall 1 bits i kodeordet er partall (eller oddetall). Løses med Venn-diagram
a) Vieer databit
b) Viser feilkorrigerende kode
c) Viser overført kode
d) Viser feildeteksjon

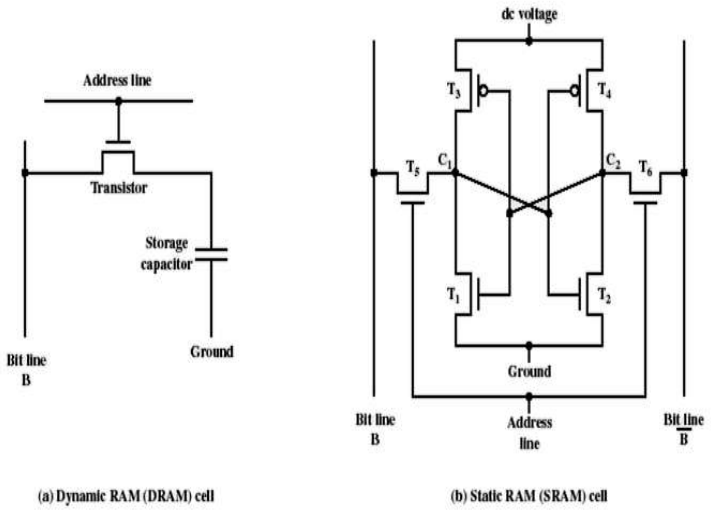


2.3.3 Minnemoduler

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.

-Statisk 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 hurtigbuffere.

-Dynamisk 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). Ulemper er lav hastighet og at datamaskinen må oppfriske innholdet hele tiden. Bruksområdet til DRAM er blant annet primærlager i PC-er.



(a) Dynamic RAM (DRAM) cell (b) Static RAM (SRAM) cell

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 sykler. 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 PCer.

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.

ROM Har data fastprogrammert inn i brikken som en del av fabrikkeringsprosessen.
Et problem med ROM er at den må fabrikkeres 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.

PROM Programmerbar ROM. 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.

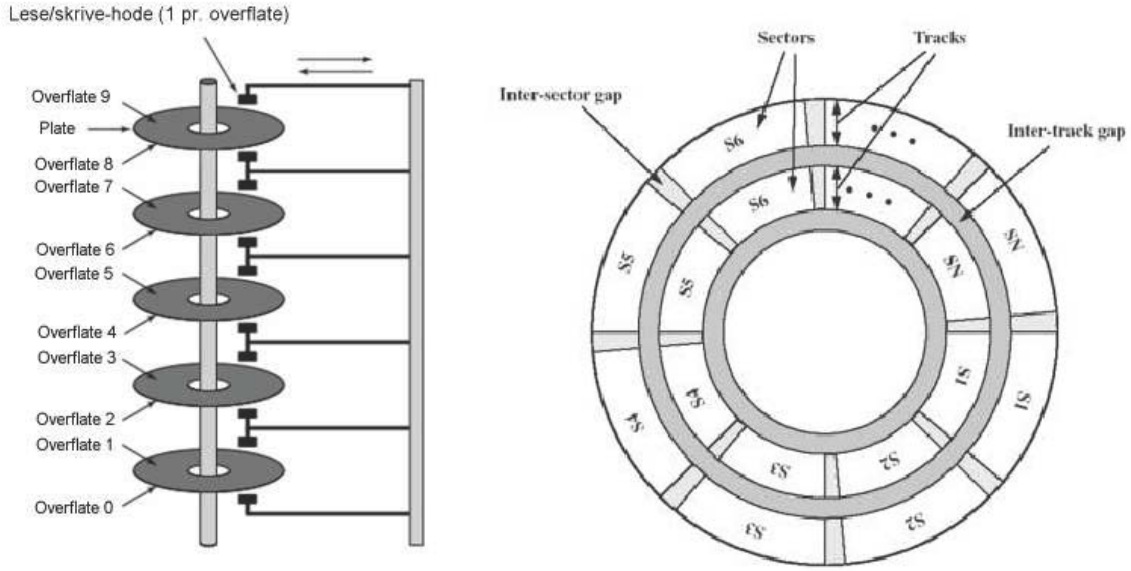
EPROM Erasable PROM kan slettes ved å bruke UV-lys, slik at man kan skrive nye data til den mer enn en gang. Ulempen er at det tar lang tid, og krever UV-belysning av brikken.

Belysning skjer gjennom et vindu på brikken, vist i figuren under.

EEPROM Electronical Erasable PROM kan slette innholdet 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.

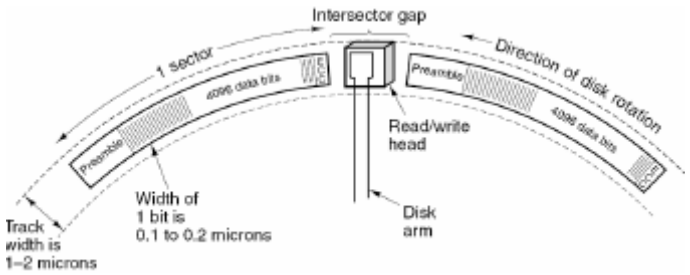
Flash Flash har relativt hurtig skriving og billig i forhold til EEPROM. Tåler like mange overskrivinger som EEPROM. Skriving skjer blokkvis, og er derfor egnet når man som oftest bytter ut hele innholdet hver gang man skriver.

2.3.4 Harddisk



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).

Spor: Sirkulær sekvens av bits skrevet når disken gjør en komplett rotasjon
Sektor: Hvert spor er delt opp i sektorer av fast størrelse, typisk 512 byte
Preamble: Synkroniseringsbit i starten av en sektor
ECC: Feilkorrigerende kode etterfølger data i hver 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.

Ytelse
Aksesstid = Søketid (flytte hodet til riktig spor) + rotasjonsforsinkelse + Overføringstid (tiden det tar å overføre dataene)

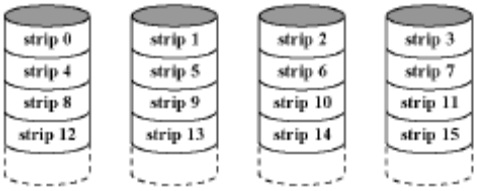
RAID
«Redundant Array of Independent Disks». Opprinnelig sto det for «Redundant Array of Inexpensive Disks», fordi RAID var en måte å benytte billige og upålitelige harddisker til kritiske oppgaver. Nå er alle disker billige og man syntes derfor at RAID trengte en ny betydning.

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 karakteristikkene:
- 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 disker slutter å fungere

RAID 0
Ingen redundand

- Ingen redundans – en stripe finnes bare på en disk
- Derfor: Ikke ekte RAID-nivå
- Fordeler: Høy ytelse, ingen ekstra kostnad
- Ulempe: Høy risiko for diskfeil
- Liten stripe-størrelse gir høy datarate
- Stor stripe-størrelse gir lav gjennomsnittlig responstid



RAID 1
Speiling

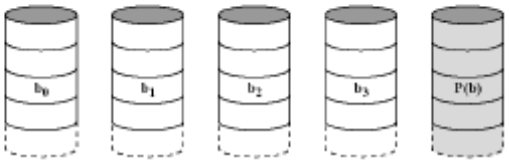
- Alle striper ligger på to forskellige disker
- Fordeler: Lav responstid, feilhåndtering enkelt
- Ulemper: Kostbart, mye skriving gir lav ytelse

RAID 2
Redundans m/ Hamming

- Hamming-kode brukes for å øke pålitelighet
- Små striper – f.eks. byte eller ord
- Parallell aksess – alle disker deltar i alle transaksjoner
- Fordeler: Høy pålitelighet, feilretting tar liten tid
- Ulemper: Kostbar, høyere pålitelighet enn nødvendig

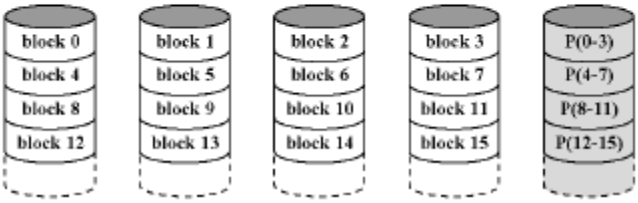
RAID 3
Bit-flettet
Paritet

- Paritetsbit brukes for å oppnå pålitelighet
- Paritet: $P(b) = b_0 \oplus b_1 \oplus b_2 \oplus b_3$
- Små striper – f.eks. byte eller ord
- Parallell aksess – alle disker deltar i alle transaksjoner
- Fordeler: Høy datarate, "passelig" pålitelig
- Ulempe: Ingen forespørsler i parallell



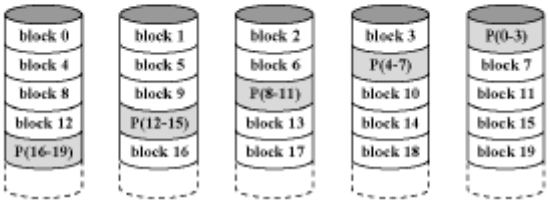
RAID 4
Blokknivå
Paritet

- Pålitelighet oppnås vha. paritetblokker, all paritet på en disk
- Relativt store striper (blokker, ikke byte)
- Uavhengige aksesser, disker opererer uavhengig
- Fordel: Forespørsler kan håndteres i parallell
- Ulempe: Alle skriveoperasjoner må oppdatere paritetsdisk



RAID 5
Blokknivå
Distribuert
paritet

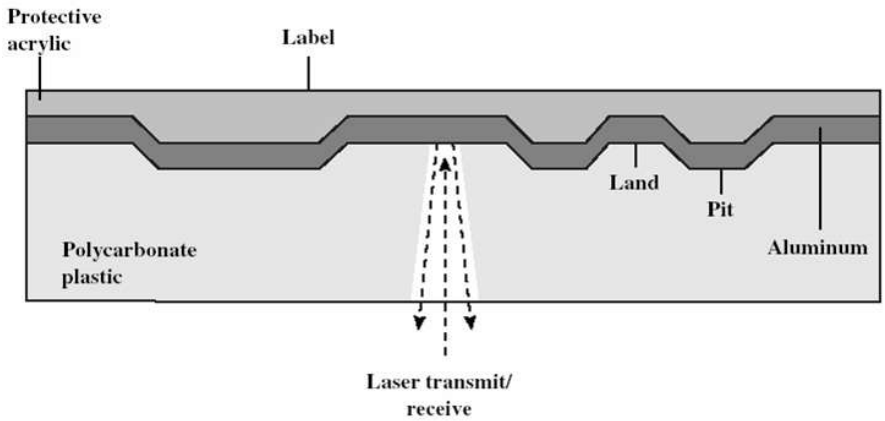
- Som RAID 4, men paritetsblokker er distribuert
- Fordel: Unngår flaskehals ved skriving



RAID
Sammen-
ligning

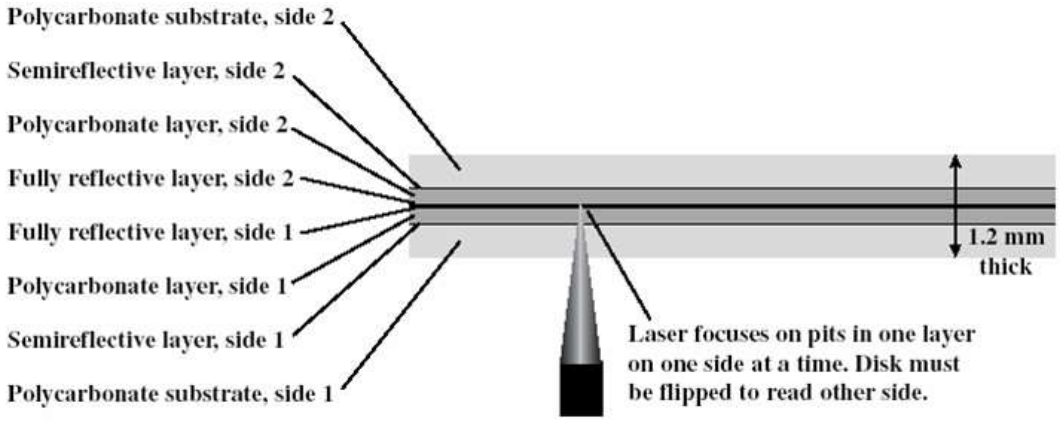
2.3.5 Optisk plate

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. Hastighet x150 KB/s
En CD-ROM-plate har en kapasitet på 700MB.

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.

2.3.6 Input/output (I/O)

Tastatur
Tastetrykk medfører at en krets blir lukket, genererer avbrudd for trykk og slipp

CRT-skjerm
Cathode Ray Tube – elektronkanon tegner linje for linje, fosforbelegg lyser

Flatskjermer
Liquid Crystal Display består av flytende krystaller som orienteres med et elektrisk felt

3 Digitalt logisk nivå

3.1 Tallsystemer

3.1.1 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 digiTS) 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).

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. $1024 = 2^{10}$.

3.1.2 Flyttallsrepresentasjon

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.

Fastpunkt
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.

Flyttall
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 tiereksponent: $1{,}49 \cdot 10^{14}$. 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 IEEEs 32-bits flyttallsstandard. Vi har tre felt:
fortegn
signifikand
eksponent + bias

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 + bias i stedet for å lagre E direkte. Dersom eksponenten er på 8 bit vil man typisk ha en bias på 127. Dette gjør 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) = 10000001(2)
Signifikanden blir 0001
Vi ender da opp med dette: 0 10000001 00010000000000000000000

Her er noen andre eksempler på hvordan vi lagrer noen gitte binære flyttall: $1.1010001 \cdot 2^{10100} = 0$ 10010011 10100010000000000000000 $-1.1010001 \cdot 2^{10100} = 1$ 10010011 10100010000000000000000 $1.1010001 \cdot 2^{-10100} = 0$ 01101011 10100010000000000000000 $-1.1010001 \cdot 2^{-10100} = 1$ 01101011 10100010000000000000000
Eksempler binærform av flyttall

3.1.3 Omregning fra ett tallsystem til et annet

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

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

$4d20(16) = (0 \cdot 16^{0}) + (2 \cdot 16^{1}) + (13 \cdot 16^{2}) + (4 \cdot 16^{3}) = 19744$

3.5 Prosessor-eksempler

3.5.1 Pentium 4
- Laget av Intel, første utgave i 2000
- 42 mill transistorer
- 32 bits prosessor, men overfører 64 bits mellom CPU og hovedlager
- Hyperthreading: to sett med registre, kan bytte fort mellom to programmer
- Hurtigbufferhierarki: 1: 8-16KB, 2: 256KB-2MB, 3: 2MB (noen modeller)
- Samlebånd i minnebuss
- Bruker Snoopy-protokoll, distr. Løsning for ugyldiggjøring ved skriving og oppdatering ved skriving

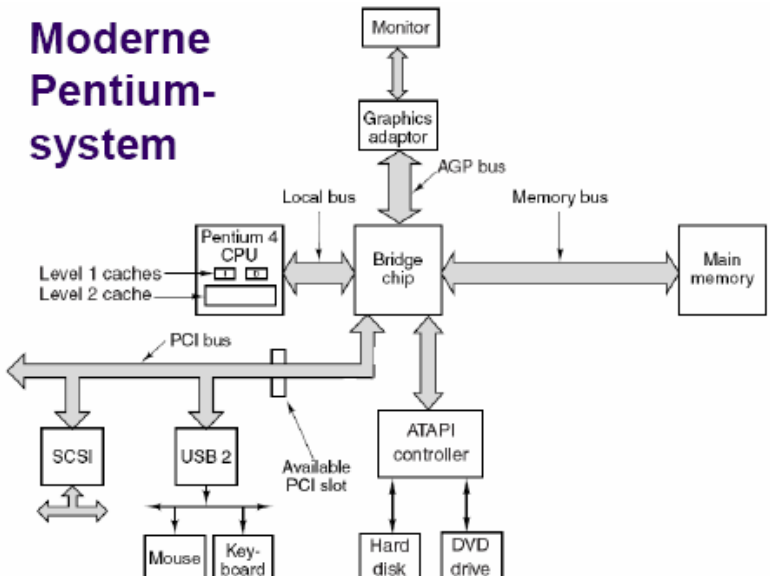
3.5.2 UltraSPARC III
- 64 bits prosessor, vekt på multiprosessering
- Hurtigbuffer: 1: 32KB instruksjoner 64KB data, 2: Kun kontroller på prosessorbrikke, eksternt 1-16MB
- Aksess av hovedlager: Ultra Port Architecture kontroller som grensesnitt

3.5.3 8051
- Mikrokontroller for innebygde system
- Kun 40 pinner
- 128-256 B RAM
- 2-54 KB ROM
- Ingen hurtigbuffer
- Ekstremt lavt strømforbruk

3.6 Busseksempler

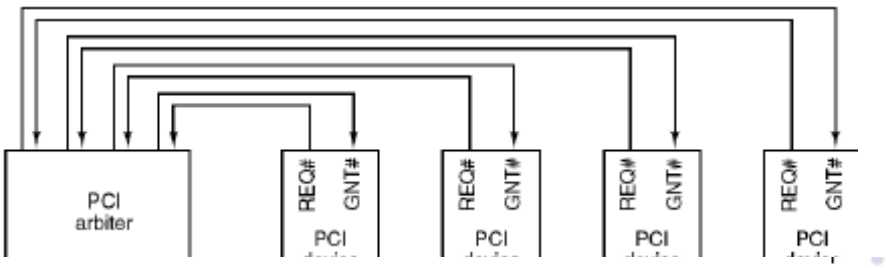
3.6.1 Industry Standard Architecture (ISA)
- PC/AT ble for tregt, kunne ikke utvide med flere kontaktpunkter
- IBM Microchannel bskyttet av patenter
- ISA: PC/AT på høyere klokkefrekvens; bakoverkompatibel
- Maks 16,7 MB/s

3.6.2 Accelerated Graphics Port (AGP)
- Statig høyere oppløsning og hastighet på output krever høyere båndbredde enn PCI klarer
- Ny buss for grafikk: 264 MB/s – 3,1 GB/s

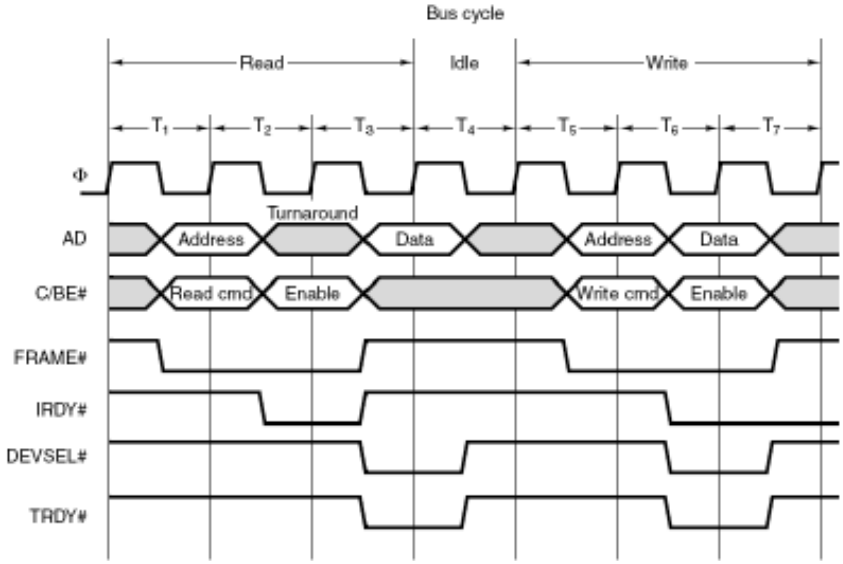


3.6.3 Peripheral Component Interface (PCI)
- Laget av Intel, men åpen for alle: prosessoruavhengig
- 33 eller 66 MHz
- 32 eller 64 data- og adresselinjer (multiplexet)
- 133 til 528 MB/s
- Støtter opptil 16 tilkoblingspunkter
- Synkron buss, sentralisert arbitrering
- Autokonfigurasjon (”plug-and-play”)

Arbitrering
- Sentralisert, del av ”bridge chip”
- Separate linjer til hver enhet
- Skjer samtidig med bussoverføring (skjult arbitrering)
- Algoritme for valg av enhet ikke del av PCI-standard



Busstransaksjoner
1. Bli bussmaster (arbitrering)
2. Start transaksjon:
    1. Aktiver FRAME#-linja
    2. Legg startadresse på AD-linjer
    3. Kommando på C/BE-linjene
        - I/O Read, I/O Write
        - Memory Read/Write
3. Gjennomfør transaksjon
4. Deaktiver FRAME# før siste overføring



PCI-express
- PCI ikke rask nok, tar stor plass
- Seriell
- Punkt-til-punkt, ikke buss
- 250 MB/s pr serielle forbindelse (lane), kan ha flere i parallell
- Pakkebasert med full adresse
- Klokkesignal kodet inn i data (10 bit pr byte for å sikre nok 0/1-1/0 transisjoner)
- Lag-basert protokoll (fysisk, link, transaksjon, programvarelag)

3.6.4 Universal Serial Bus (USB)
- PCI for dyrt og stort for eksterne enheter
- 1,5, 12 og 480 MBps
- Enkel kabel, kun 4 ledere, leverer også strøm
- Hub-topologi hvor hver enhet kan gi flere tilkoblingspunkter og gi strøm, opp til 127 enheter
- Plug-and-play
- Reboot ikke nødvendig

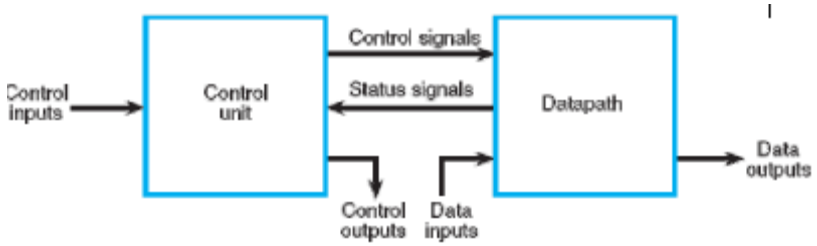
3.6.5 Andre busser

ATA/IDE/ATAPI	Enkel parallell buss for harddisk og CD-ROM, maks 133 MBps
Serial-ATA	Seriell versjon av ATA, 150-300 MBps
SCSI	Avansert parallell buss, mest brukt i servere, maks 320 MBps
FireWire	Seriell buss mest brukt for videoprodukter, 400-800 MBps

4. Mikroarkitekturnivået

Nivået over digitalt logisk nivå, og skal implementere ISA-nivået over.
Eksempel på ISA-instruksjoner:
DUP Kopier ord på toppen av stakken og kopier til stakk
IADD Pop to ord fra stakken og push summen tilbake på stakken

Organisering



Styreenhet
Styrer den utførende enheten
Enten via mikroprogram eller maskinvare

Utførende enhet (datapath)
Samling av registre
ALU: Aritmetrisk logisk enhet

Mikroprogram
- Sekvens av mikroinstruksjoner som til sammen implementerer en ISA-instruksjon
- En mikroinstruksjon styrer styreenhet og utførende enhet i én klokkesyklus
- Mikroprogrammene lagres i eget styrelager, kun synlig for mikroarkitektur
- Hver klokkesyklus: hente ny mikroinstruksjon fra styrelager, bruk denne til å styre utførende enhet, bruk denne til å bestemme adresse til neste mikroinstruksjon.

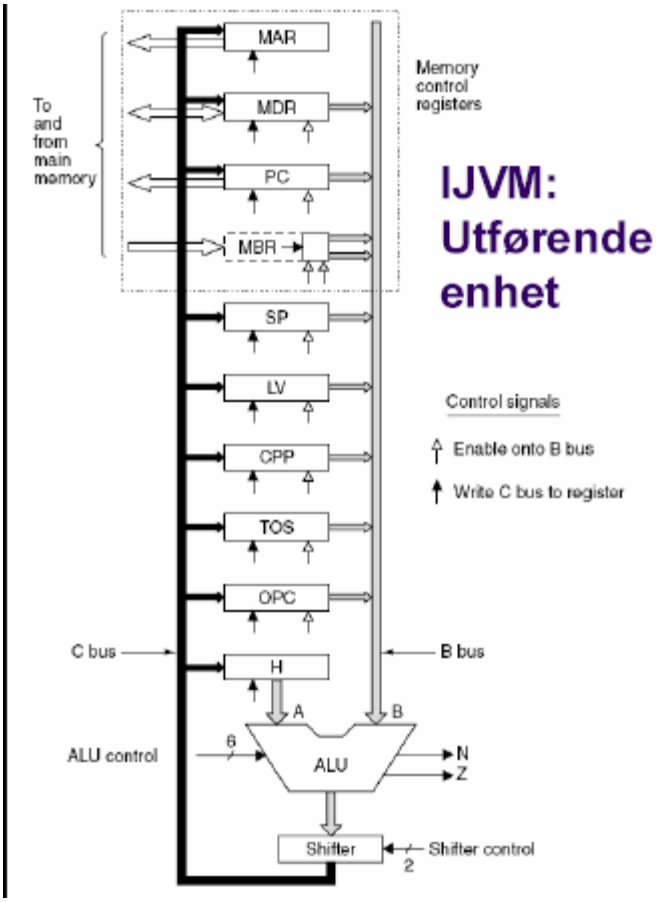
4.1 Eksempel på mikroarkitektur: IJVM

- Mikroarkitektur som støtter Integer Java Virtual Machine instruksjoner direkte

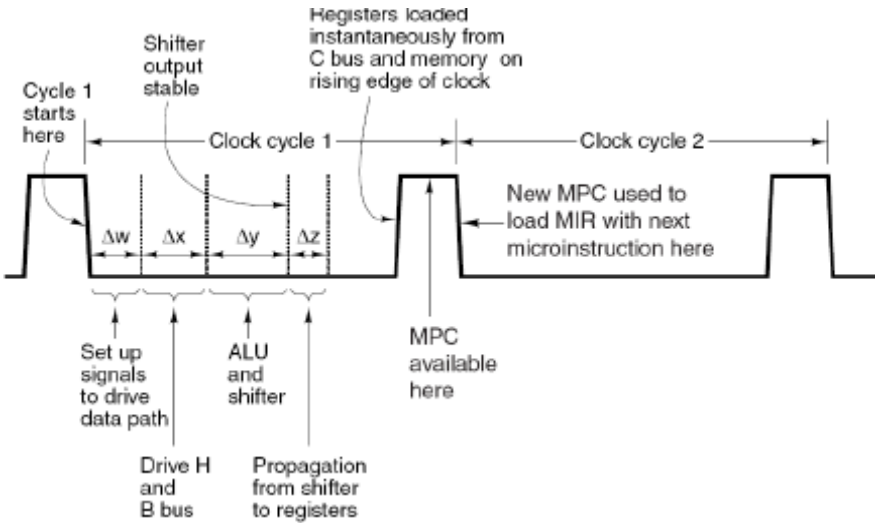
4.1.1 Utførende enhet

Tilstander Variable
PC Programmteller:
adresse til neste instr
opcode Instr. som skal utføres
32 bits registre
ENA Enable input A
ENB Enable input B
INVA Invert input A
INC Inkrementere res med 1
H Holding register
SLL8 Shift Left Logical, skifter innh
1 bit til venstre+8lsb 0
SRA1 Shift Right Artimetic, skifter innh
1 bit til høyre, msb uendret

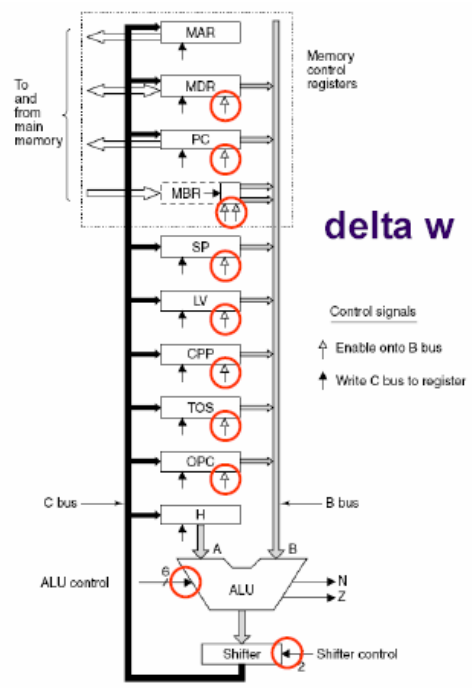
F_0	F_1	ENA	ENB	INVA	INC	Function
0	1	1	0	0	0	A
0	1	0	1	0	0	B
0	1	1	0	1	0	$\bar{A}$
1	0	1	1	0	0	$\bar{B}$
1	1	1	1	0	0	A + B
1	1	1	1	0	1	A + B + 1
1	1	1	0	0	1	A + 1
1	1	0	1	0	1	B + 1
1	1	1	1	1	1	B – A
1	1	0	1	1	0	B – 1
1	1	1	0	1	1	–A
0	0	1	1	0	0	A AND B
0	1	1	1	0	0	A OR B
0	1	0	0	0	0	0
1	1	0	0	0	1	1
1	1	0	0	1	0	–1

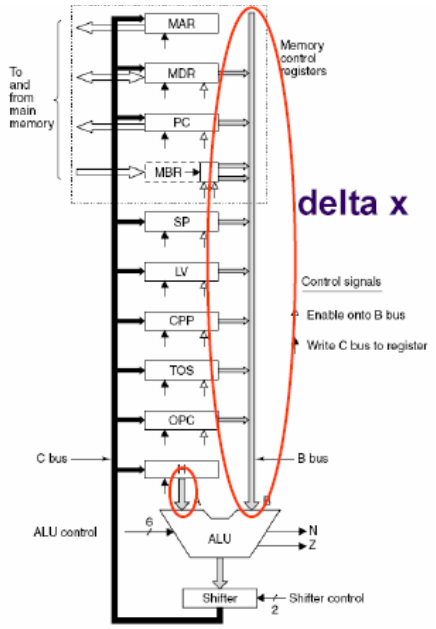


Timing



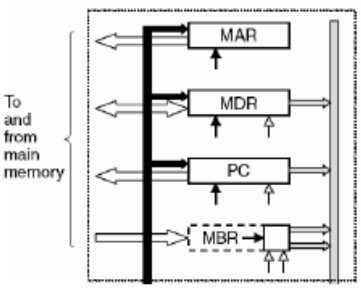
Δ w: fallende klokkeflanke: Kontrollsignaler settes opp
Δ x: Registrene lastes på B-bussen





Δ y: Shifter og ALU prosesserer data og blir stabile
Δ z: Resultater har nådd registrene langs C-bussen, og kan skrives på neste stigende klokkeflanke
Samtidig stoppes driving av B-bussen for å forberede neste sykel.

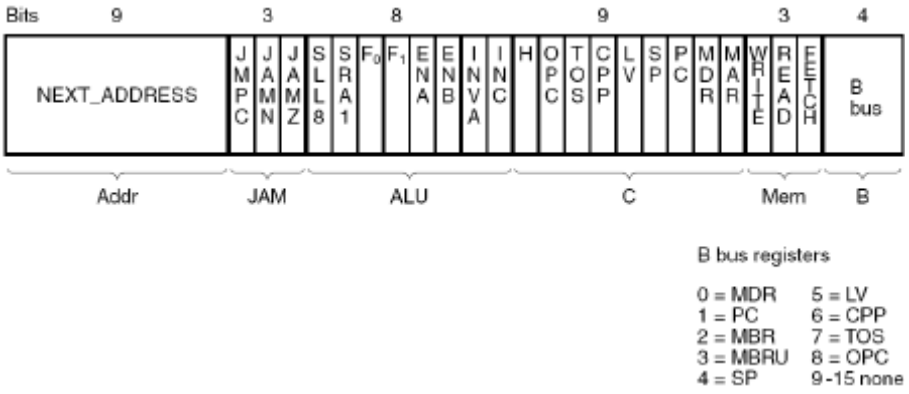
Minneoperasjoner



- 32-bit port
  - Memory Address Register
  - Memory Data Register
  - Brukes for data
  - MAR inneholder ordadr.
- 8-bit port
  - Program Counter
  - Memory Buffer Register
  - Brukes for instruksjoner
  - PC inneholder byteadr.
  - 8 bit utvides til 32
    - Enten med 0 eller sign extension

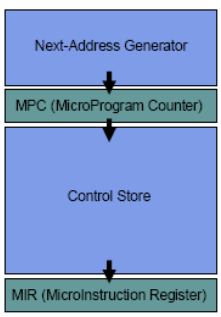
Lese fra hovedlager: Timing
Klokkesyklus *k*: MAR/PC oppdatert i slutten av klokkesyklus
Klokkesyklus k+1: Antar at minneaksess tar 1 syklus (100% treff i hurtigbuffer), MDR/MBR oppdatert
Klokkesyklus k+2: Data/instr. lest fra hovedlager er klar til bruk

Styring av utførende enhet
- Kilder til ALU
- ALU/skifter
- Mål for ALU
- Aksess av hovedlager
- Velge neste mikroinstr. til styreenhet



Addr + JAM:	Bestemmer neste mikroinstruksjon
Resten:	Styrer utførende enhet i én klokkesyklus
Addr:	Inneholder adressen til neste mikroinstruksjon
JAM	Avgjør hvordan neste mikroinstruksjon velges
ALU	ALU og skifter-funksjoner
C	Velger hvilke registre som skrives til fra C-bus
Mem	Minnefunksjoner
B	Velger kilde for B-bussen

Mikroprogrammert styreenhet



- Krets for å generere neste adresse
- Register for å holde adresse
- Lager for mikroprogram
- Register for å holde gjeldende mikroinstruksjon

Valg av neste mikroinstruksjon

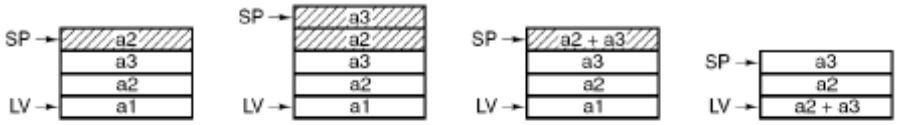
- Styrelager inneholder 512 mikroinstruksjoner → trenger 9 bits adrese
- Hva med forgreninger: JAMZ: ta hensyn til Z (siste svar=0)
  JAMN: Ta hensyn til N (siste svar negativt)
  JMPC: Ta hensyn til MBR

4.2 ISA: IJVM

4.2.1 Stakk

- Når en metode blir kalt settes det av plass på stakken til alle lokale variabler
- Register LV (local variables) pekter til der første (nederste) variabel ligger
- Register SP (stack pointer) peker til toppen av stakken
- Adressen til lokal variabel blir LV+offsettt, dermed ikke absolutt minneadresse
- LV og SP er ordadresser

Operandstakk



- Stakken brukes også til mellomlagring av operander
- Eksempel: a1 = a2 + a3
- Figur 1: Push a2 (LV + 1)
- Figur 2: Push a3 (LV + 2)
- Figur 3: Pop 2 verdier fra stakk, adder, push svar
- Figur 4: Pop svar og legg det i a1 (LV + 0)

Minnemodell

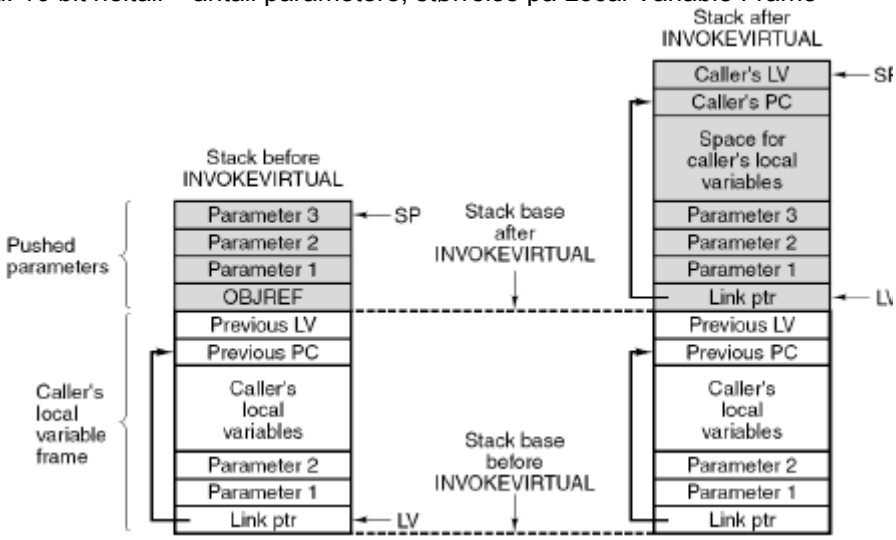
- 32 bits ordstørrelse = 32 bits adresser → 4GB hovedlager
- Ingen absolutte adresser, bare relative
- Relative adresser = perker + offsett
- Constant Pool (CPP): Kontanter fra Java-program, leses inn ved oppstart, endres ikke
- Local Variable Frame (LV): Setter av plass til lokale variable ved metodekall
- Operand Stack (SP) Midlertidig lagring av operander, SP peker til toppen av stakk
- Method Area (PC): Selve Java-programmet ligger het

Instruksjonssett

- Stakkmanipulasjon
- Aritmetiske instruksjoner
- Ubetinget og betinget hopp (PC+offset)

Hex	Mnemonic	Meaning
0x10	BIPUSH *byte*	Push byte onto stack
0x59	DUP	Copy top word on stack and push onto stack
0xA7	GOTO *offset*	Unconditional branch
0x60	IADD	Pop two words from stack; push their sum
0x7E	IAND	Pop two words from stack; push Boolean AND
0x99	IFEQ *offset*	Pop word from stack and branch if it is zero
0x9B	IFLT *offset*	Pop word from stack and branch if it is less than zero
0x9F	IF_ICMPEQ *offset*	Pop two words from stack; branch if equal
0x84	IINC *varnum const*	Add a constant to a local variable
0x15	ILOAD *varnum*	Push local variable onto stack
0xB6	INVOKEVIRTUAL *disp*	Invoke a method
0x80	IOR	Pop two words from stack; push Boolean OR
0xAC	IRETURN	Return from method with integer value
0x36	ISTORE *varnum*	Pop word from stack and store in local variable
0x64	ISUB	Pop two words from stack; push their difference
0x13	LDC_W *index*	Push constant from constant pool onto stack
0x00	NOP	Do nothing
0x57	POP	Delete word on top of stack
0x5F	SWAP	Swap the two top words on the stack
0xC4	WIDE	Prefix instruction; next instruction has a 16-bit index

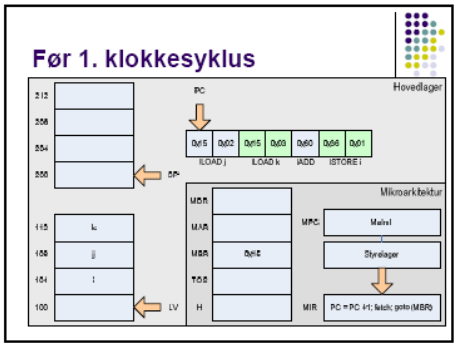
Metodekall
- Push objectReference (this) og parametere
- Utfør INVOKEVIRTUAL disp (disp = constanct pool offset til metodeadresse)
- Må ta vare på gammel Verdi for PC og LV
- Oppdaterer PC med ny verdi
- Method area: 16 bit heltall – antall parametere, størrelse på Local Variable Frame

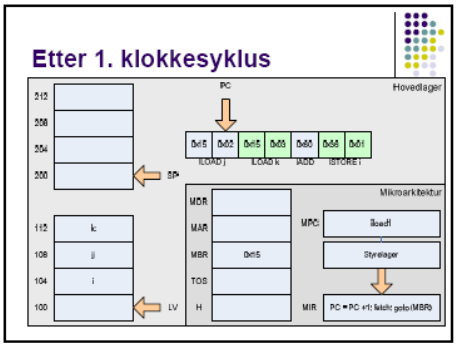


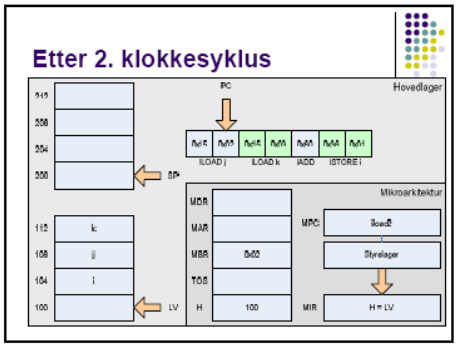
MAL – Micro Assembly Language
- Symbolsk språk for IJVM mikroarkitektur
- Hver linje tilsvarer det som skal skje i en klokkesyklus
- Eksempler: SP = SP+1; rd , MDR = SP
- Aksess av hovedlager: rd: les ord (MAR/MDR), wr (MAR/MDR), fetch (PC/MBR)

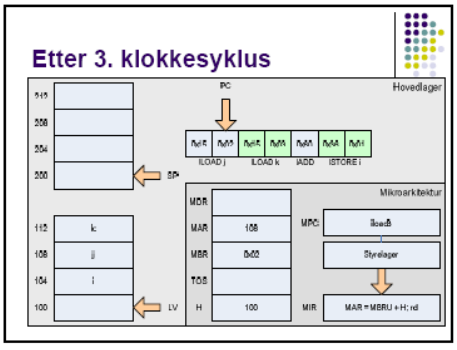
Eksempel: 1 = j + k

1 ILOAD j //i = j+k	0x15 0x02
2 ILOAD k	0x15 0x03
3 IADD	0x60
4 ISTORE i	0x36 0x01

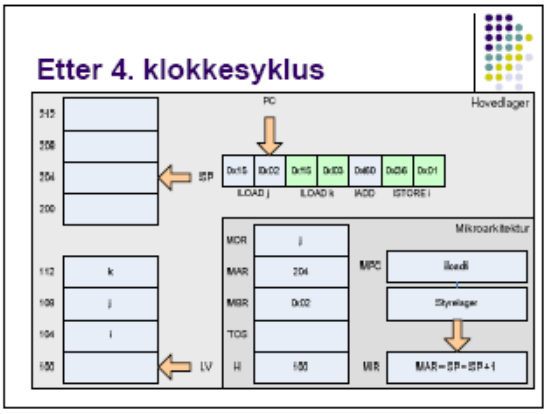




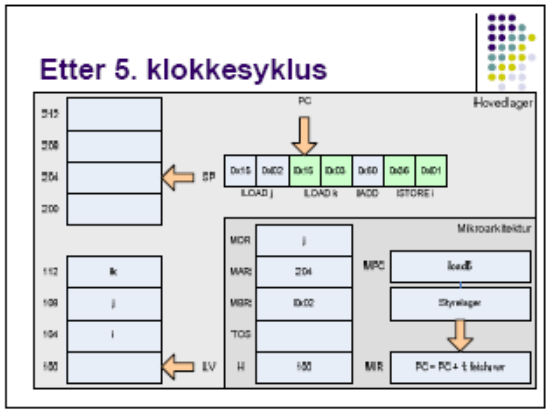




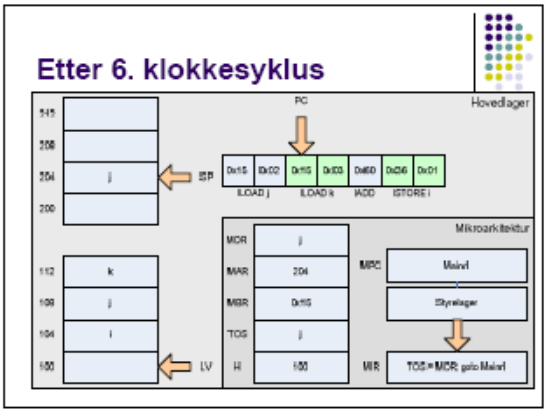
Etter 4. klokkesyklus



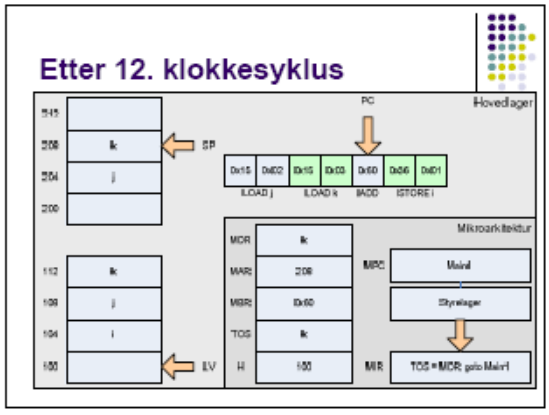
Etter 5. klokkesyklus



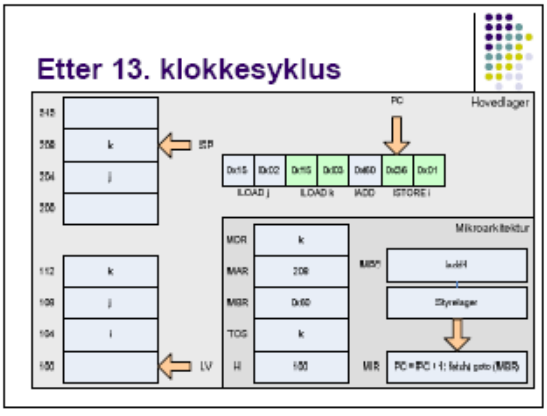
Etter 6. klokkesyklus



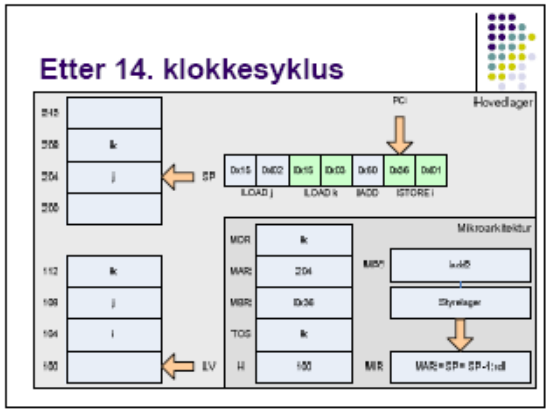
Etter 12. klokkesyklus



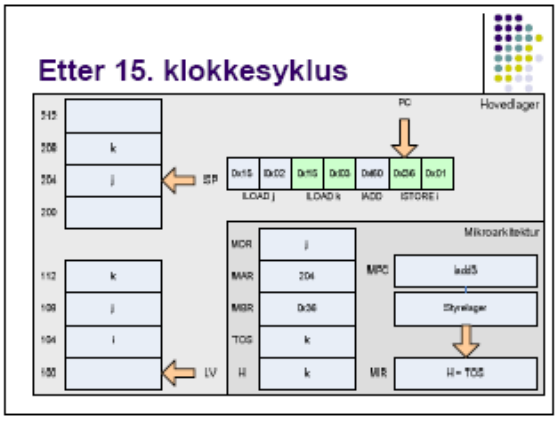
Etter 13. klokkesyklus



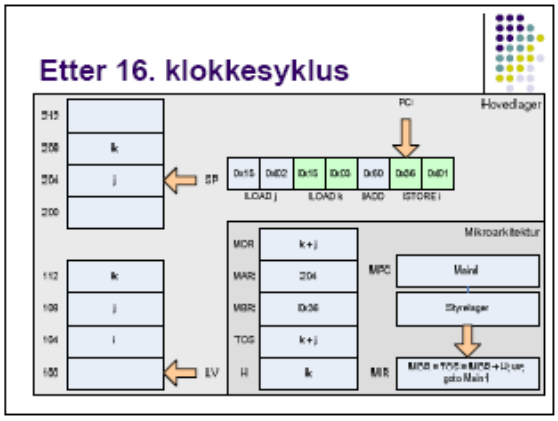
Etter 14. klokkesyklus



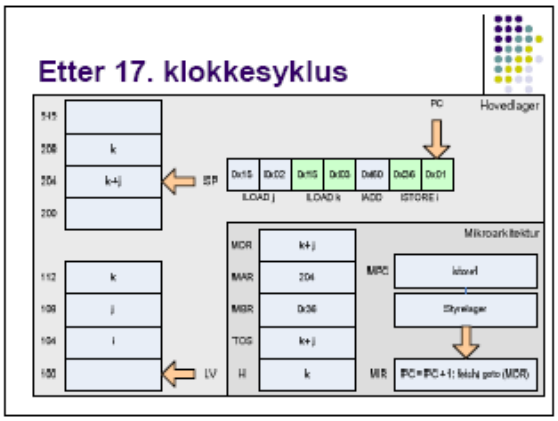
Etter 15. klokkesyklus



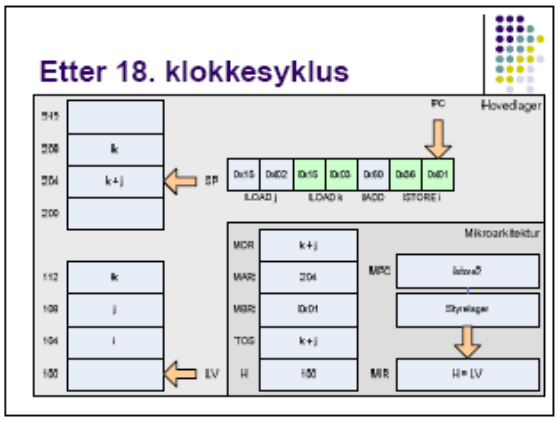
Etter 16. klokkesyklus



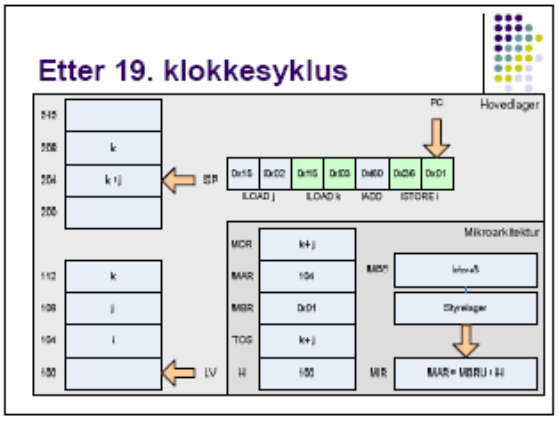
Etter 17. klokkesyklus



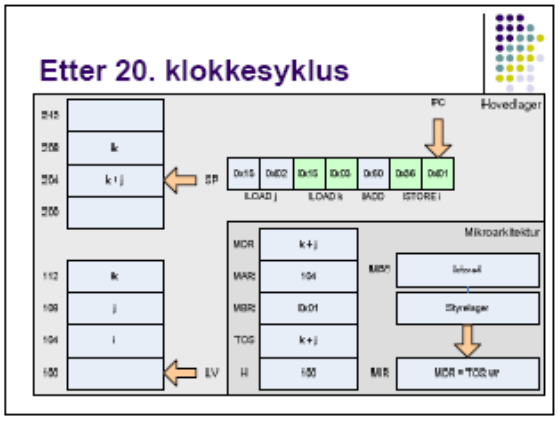
Etter 18. klokkesyklus



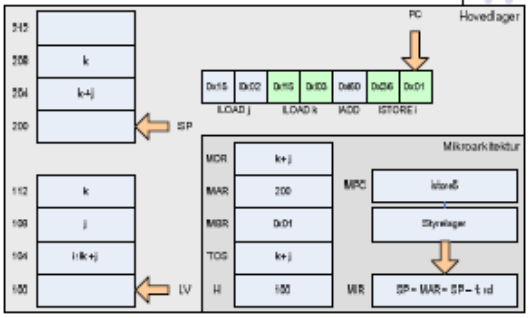
Etter 19. klokkesyklus



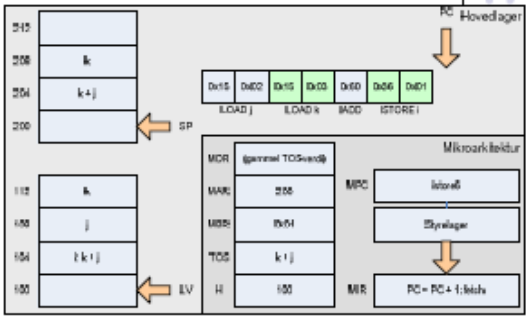
Etter 20. klokkesyklus



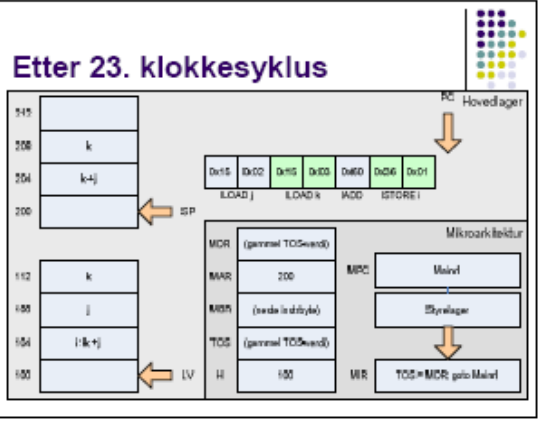
Etter 21. klokkesyklus



Etter 22. klokkesyklus

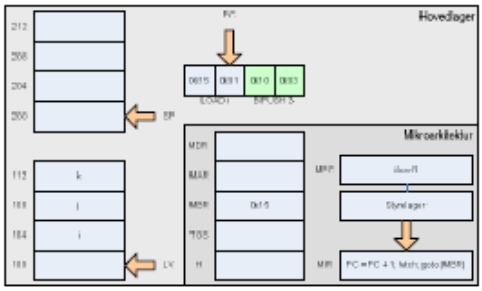


Etter 23. klokkesyklus



Dagens oppgave (1/2)

Gitt følgende situasjon etter en klokkesyklus.
Hva er innholdet i de forskjellige boksene
etter at ILOAD og BIPUSH er utført?



4.4 Optimalisering av mikroarkitektur

4.4.1 Hastighet vs kostnad

Redusere utføringstid:
1. Redusere antall klokkesykler (færre mikroinstruksjoner)
2. Gjøre hver klokkesyklus kortere (forenkling av mikroarkitektur)
3. Overlappe utføring av instruksjoner (samlebånd)

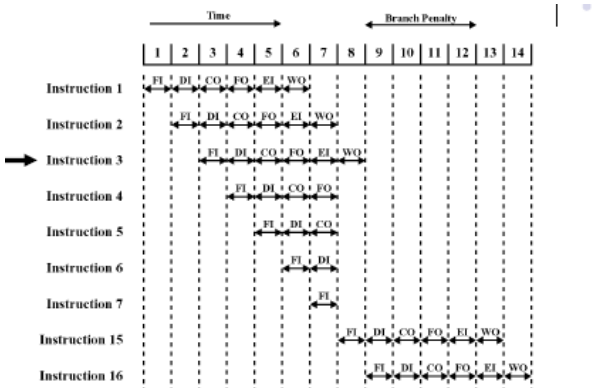
Frigjøring av ALU
Spesialiserte enheter for inkrementering av PC, henting av programkode, sammenslåing av operander

Instruction Fetch Unit
- Henter 4 byte om gangen
- Siste leste byte ligger i MBR1, siste to leste ligger i MBR2
- Oppdaterer PC hver gang MBR1/MBR2 leses
- Når det er få innleste bytes bufret leses 4 til
- IMAR nytt adresseregister inneholder adresse ti neste 4 byte-blokk
- PC inneholder adresse til neste byte
- Prefetch: Henter bytes før de trengs
- Trenger ikke PC = PC+1, skrives bare til av mikroinstr. ved hopp
- Trenger ikke sette sammen 2-bytes operander

→ Kortere mikroprogram, færre klokkesykler pr. program, bedre ytelse

Samlebånd
- Maksimal instruksjonshastighet er gitt av 1/(største forsinkelse i ns)
- Syklustid: $\tau = \max[\tau_i] + d = \tau_m + d \approx \tau_m$
- Tidsforbruk for utføring av *n* operasjoner i *k*-stegs samlebånd: $T_k = [k + (n-1)]\tau$
- Forbedring fra uten samlebånd: $\frac{T_1}{T_k} = \frac{nk\tau}{\tau[k+(n+1)]} = \frac{nk}{[k+(n+1)]} \approx k$
- Forutsetter at man vet adressene til instruksjonene som skal utføres, ikke mulig ved hopp
- Lagrer tabell med om de siste instr. innebar hopp og hva slags instr. som fører til hopp

Forgreningspredikering
- Statisk: alltid anta hopp
- Gjett basert på type forgreningsinstruksjon
- Gjett basert på hoppretning
- Hint fra kompilator, for eksempel ved løkker
- Historiebit
- Tabell av historiebits



4.6 Eksempler på mikroarkitektur

4.6.1 Pentium 4
- CISC pga. bakoverkompatibilitet
- RISC pga. samlebånd/superskalaritet (fast instruksjonsformat, tilnærmet like mye arbeid pr. instr.)
- Fokus: høy klokkefrekvens, 20-31 steg i samlebånd

Minnesystem
- L2: Felles for data og instruksjoner 256 KB – MB
- 8-vei sett assosiativ
- 128 bytes bufferlinjer
- Write-back
- Prefetch
- Nye data kan hentes annenhver syklus

Font end
- Instruksjoner hentes fra L2 og dekodes i rekkefølge
- Dekodes til mikrooperasjoner, ligner RISC-instruksjoner
- Trace cache = L1 hurtigbuffer for instruksjoner
- Forgreningspredikering
- Superskalaritet: spesialiserte enheter for heltall, flyttall, load/store

5. Instruksjonssettarkitektur-nivå (ISA-nivå)

- Nivå mellom maskin- og programvare

Bakoverkompativilitet
- Prosessorfamilier kan utføre samme ISA-instruksjoner
- Ny ISA lages sjelden kun utafra hva som er mest optimalt nå

Hva er en bra ISA?
God design kan gi opptil 25% ytelsesforskjell
To kunder:
1) Maskinvaredesignere (ISA-instruksjoner som kan implementeres effektivt)
2) Programvaredesignere (enkelt å generere enkel ISA-kode vha kompilator, regularitet og kompletthet)

Hva omfatter ISA?
”Alt som sees av de som skal programmere/kompilatoren”
- Minnemodell: Hvordan er hovedlageret organisert?
- Hvilke registere finnes
- Hva skal registerene brukes til
- Hvilke datatyper finnes
- Hvilke instruksjoner som fine og hva de gjør

Samlebånd, superskalaritet etc. er ikke med i definisjonen av ISA-nivå, men detaljer om dette er viktig mht optimalisering

Minnemodeller
- Størrelse på adresser (antall bit, normalt 32 bit som gir 4G adresser)
- Adresserbar enhet (normalt 8 bit, men 1-60 bits har eksistert)
- Organisering i større enheter (Ord typisk 4 eller 8 byte)

Alingment
Kan et 8 bytes ord ha en vilkårlig adresse?
Nei → ”Alignment”: kan kun ligge på adresse 0,8,16
Ja → ”Nonalignment”
Ofte enklere og raskere å hente ord som ligger på ”naturlige” adresser
P4 kan ikke oppgi 3 siste adressebit, men må støtte vilkårlige adresser pga. bakoverkompatibilitet

Adresserom
- Normalt har man kun ett adresserom – felles for data og instruksjoner
- Alternativ: forskjellig adresserom, instruksjon og dataelement kan ha samme adresse, skriveoperasjoner kan ikke ødelegge instruksjoner (read only i instruksjonsminnet)

LOAD/STORE og endret instruksjonsrekkefølge
Bytting av rekkefølge under utføring av ISA-instruksjoner kan være gunstig

Registere
- Raskeste lagringsressurs, på toppen av minnehierarkiet
- Noen registre er skjult fra ISA-nivået
- De fleste prosessorer har minst to modi på ISA-nivået: kjernemodus og brukermodus
- Operativsystem kjører i kjernemodus, programmer i brukermodus

Instruksjoner
- Hvilke instruksjoner finnes og hvordan de fungerer er sentralt i ISA-nivået

Pentium 4: IA-32
80386 og nyere Intel-prosessorer har hatt ISA-arkitektur IA-32

Tidligere prosessorer var 4, 8 og 16 bit
Hovedendringer har vært nye instruksjoner: MMX, SSE, SSE2
P4 kan likevel utføre programmer skrevet for tidligere prosessorer i samme familie
Real mode, Virtual 8086 mode, Protected mode
Har kjerne- og brukermodus (+2 til)
Adresserom delt i 16386 segmenter, hvert segment med adresser fra 0 til 2^{32} , i praksis brukes ett
Få/ingen helt generelle registre (spesialfunksjoner i 8-, 16- og 32 bits versjoner, vanskelig med komp)

RISC-prosessor
Opprinnelig 32 bits nå 64 bits
Ett adresserom: 0 – 2^{64} (byteadresserbart), men i praksis 0 - 2^{44}
Load/Store-arkitektur (Register-til-register)
Mange generelle registre
Registerbruk: 32 heltallsregister, 32 flyttallsregister, i utgangspunktet generelt, noen spesielle
Registervindu (deler opp alle registre i delvis overlappende vinduer, og hver subrutine får sitt eget vindu når instansiert. Parameter-overføring i overlappende del)

6. Datatyper

Programvarestøtte
Maskinvarestøtte for datatype
- ISA-instruksjoner som opererer på spesifikke format (eks. adderer to 32-bits 2s kompl. Heltall)
- Spesifisert hvordan data skal lagres binært

Heltall
- Som regel maskinvarestøtte for flere størrelser (eks. 8, 16, 32, 64)
- Med eller uten fortegn
- Negative tall håndteres som regel vha toskomplement
- BCD: Binary Coded Decimal (For desimalsystemet, 4 bit lager hvert siffer 1: 0001)

Flyttall
- $\pm F * B^{\pm E}$
- 3,14 = 314 * 10^{-2}, 5 000 000 000 000 = 5 * 10^{9}
- Flyttall kan ikke representeres nøyaktig binært
- Representerer fraksjon og eksponent
- Med 32 bit kan man representere $\pm 2^{-127}$ → 2^{128} under- og overflyt
- Overflyt: absoluttverdi til tall blir for stort til å kunne representeres; alvorlig feil, gir typisk exception
- Underflyt: tall for lite til å kunne bli representert, som regel ikke alvorlig, kan tilnærmes 0

Regning med flyttall
Krever egne instruksjoner
- Addisjon/subtraksjon: krever lik eksponent, adder/subtraher fraksjon, behold eksponent
- Multiplikasjon: multipliser fraksjon, adder eksponenter
- Divisjon: divider fraksjon, subtraher eksponenter
- Justerer tallet med den minste eksponenten for å unngå å miste mest signifikante siffer
- Normalisering: MSB i fraksjon er ulik 0, sikrer maksimal presisjon
- Excess (bias): Negativ eksponent lagres ved å legge til konstant før lagring. 8b: -127-128 → 0-255

IEEE standard 754 for binære flyttall
Single- eller double precision
Normaliserer til 1,xxx → Kan la være å lagre 1'eren
Spesielle bitmønster for å representere 0, uendelig, NaN (for eksempel 0/0)

Ikke-numeriske datatyper
- Karakterer – ASCII (7bit) eller Unicode (16bit)
- Boolske verdier – som regel brukes en hel byte
- Peker – adresser til lagerlokasjoner
- Multimedia – eks. fargeverdier

Instruksjonsformat
- Assembler: ADD R1, R2, R3
- Maskinkode: 01010111000
- Instruksjonsformatet forteller hvilke bits som tilsvarer hva
- Generelt: Opkode + adresser til operander (eksplisitt eller implisitt, registre eller hovedlager)

Eksplisitte operander
Gitt C = A + B
Hvor mange slak kunne oppgis eksplisitt?
Mangle eksplisitt → enklere kode
Få eksplisitt → enklere instruksjoner → raskere
3-0 adresseinstruksjoner

Fast instruksjonslengde: enklere dekoding, fordeling for samlebånd, sløsing med plassen (vanlig nå)

Dark-assembly

Load-store
I en load/store arkitektur er arbeidsflyten slik:
1. Laste inn data fra minnet til register
2. Utføre operasjoner på registrene
3. Skrive tilbake fra registrene til minnet

I en load-store maskin har man tilgjengelig:
— 32 registre ($0 til og med $31), MEN
— register 0 ($0) er alltid lik null
— register 1 ($1) er reservert for assembleren
— register 31 ($31) brukes til stakken
— Ett stort minne

Add
Syntaks
add REGISTER,REGISTER,{REGISTER|NUMMER|VARIABEL}
Semantikk
register register + {register|NUMMER|VARIABEL} Målregisteret får summen av andre og siste verdi.
Eksempel:
add $1, $2, $3 ; Summen av $2 og $3 legges i $1.

Load
Syntaks:
load REGISTER,{REGISTER,NUMMER|NUMMER|VARIABEL}
Semantikk:
register {mem[register+NUM]|NUM|VAR}
Målregisteret får enten verdien som er angitt som et nummer, innholdet i variabelen eller innholdet i minneposisjonen som man finner ut fra registeret + nummeret.
Eksempel:
load $4, $3+0 ; Legger innholdet i minneadresse $3 inn i register $4.

Store
Syntaks:
store REGISTER,{REGISTER|NUMMER|VARIABEL}
Semantikk:
{mem[register+NUMMER]|VARIABEL} register
Registeret lagres i variabelen eller i minneposisjonen som pekes ut av registeret + nummeret.
Eksempel:
store $2, $3+0 ; Legger register $2 inn i minneposisjon $3.

Betingede hopp
Det finnes mange situasjoner der man bare vil hoppe i gitte situasjoner, da benytter man betingede hopp. Disse er:
— jeq - Jump if EQual
— jne - Jump if Not Equal
— jge - Jump if Greater or Equal
— jgt - Jump if Greater
— jle - Jump if Less or Equal
— jlt - Jump if Less
Alle disse tar tre argumenter, stedet det eventuelt skal hoppes til, ett register og noe det skal sammenliknes med (register, variabel eller ett nummer).

IF-setninger
I Java og liknende sp har vi ofte behov fo kunne uttrykke betingelser i form av if-setninger. Ett eksempel:
Java

```
if (a<b) {
a = a + b;
} else {
a = a - b;
}
```

DARK

```
jge etikett2, $2, $3
add $2, $2, $3
jmp slutt
etikett2:
sub $2, $2, $3
slutt:
```

WHILE-løkker
Ett eksempel:
Java

```
int a = 0;
while(a < 10) {
// Noe nyttig
a = a + 1;
}
```

DARK

```
add $5, $0, $0
toppen:
jge slutt, $5, 10
; Noe nyttig
inc $5
jmp toppen
slutt:
...
```

FOR-løkker
Java

```
for(int i = 0; i < j; i ++) {
// ....
}
```

DARK

```
mov $5, 0 ; i = 0
test:
jge ferdig, $5, $6
; ....
inc $5
jmp test
ferdig:
```

Stakk
— I en load/store maskin hadde man registre, i en stakkmaskin har man en stakk.
— Denne stakken fungerer på samme måte som dere kjenner fra alg.dat og programmering.
— Alle operasjoner foregår på toppen av stakken.
— Aritimetikk foregår ved å ta vekk to øverste elementene på stakken og legge igjen resultatet. To elementer blir altså ett.

Stakkmanipulasjon
— dup - Kopierer det øverste elementet og legger det på toppen.
— swap - Bytter om rekkefølgen av de to øverste elementene.
— drop - Det øverste elementet tas vekk

Selve hoppet baserer seg så på testen:
— jfalse - Hopp hvis det ligger 0 på stakken
— jtrue - Hopp hvis det ligger 1 på stakken
— jmp - Hopp uansett
MERK! Ettersom man hopper på denne måten er parene eq / jtrue og ne / jfalse ekvivalente!