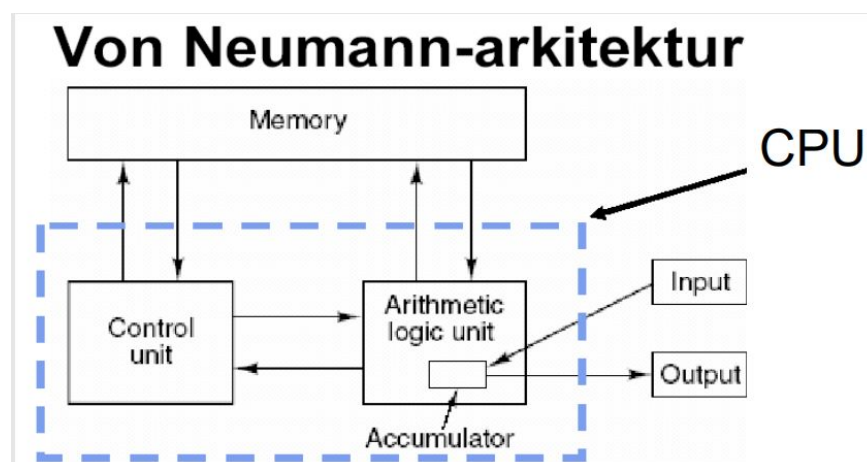
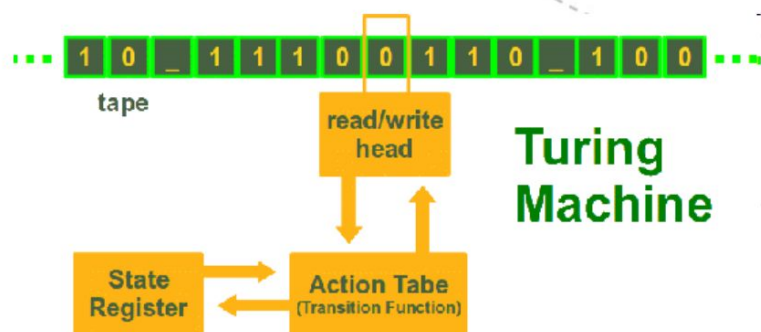


## Kapittel 1 og 2 - Historie og Computer Systems

- **Historie**
  - 1948 - Første transistor
  - 1951 - John von Neumann
  - 1971 - Verdens første mikroprosessor - Intel 4004
  - EDSAC og IAS- verdens første stored-program computer, brukte von Neumann arkitektur
- **von Neumann arkitektur**
  - Består av fire deler
    - Primærminne (Memory)
      - Inneholder data og instruksjoner (program)
      - Både minnelokasjoner med data og instruksjoner kan leses fra og skrives til, i hvilken som helst rekkefølge
      - Hurtigbuffer/Cache
    - ALU (Aritmetisk Logisk Enhet/Unit) - Utfører beregningene (aritmetiske og logiske)
    - Kontrollenhet (Controll Unit) - Tolker instruksjonene og sørger for at de blir utført. Velge alternative aksjonsveier basert på resultatet av tidligere utførte operasjoner
    - I/O - Inn og ut enheter, som kan feks lagre data over lengre tid, slik som eksterntminnet
  - ALU og Controll Unit utgjør CPU (Central Processing Unit) - dette er prosessoren
  - Arkitekturen er grunnpilaren for nesten alle datamaskiner siden
  - Programmer ble lagret på samme måte som data, og kunne endres av datamaskinen selv
  - Ligger til grunnlag for "Stored Program" konseptet
    - Programmer kan bli representert i en form som passer å bli lagret sammen med data i minnet
    - Datamaskinen kan da få sine instruksjoner ved å lese de fra minnet
  - Arkitektur

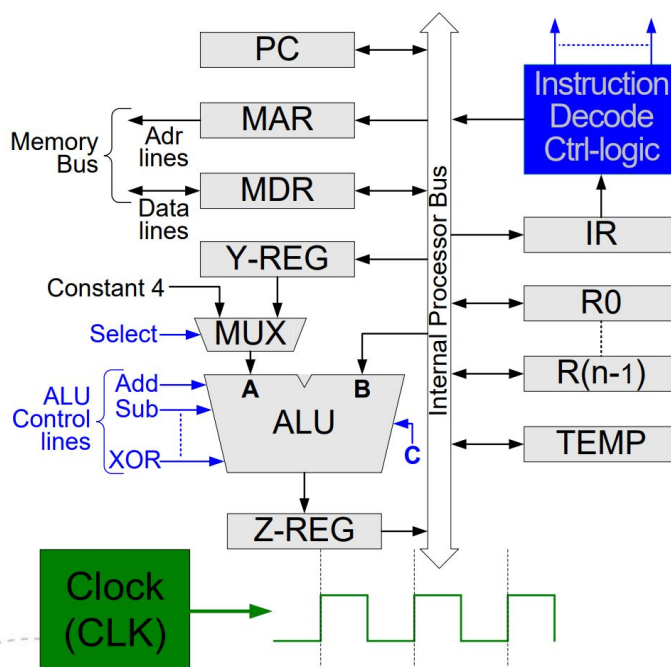


- **Turing Machine**

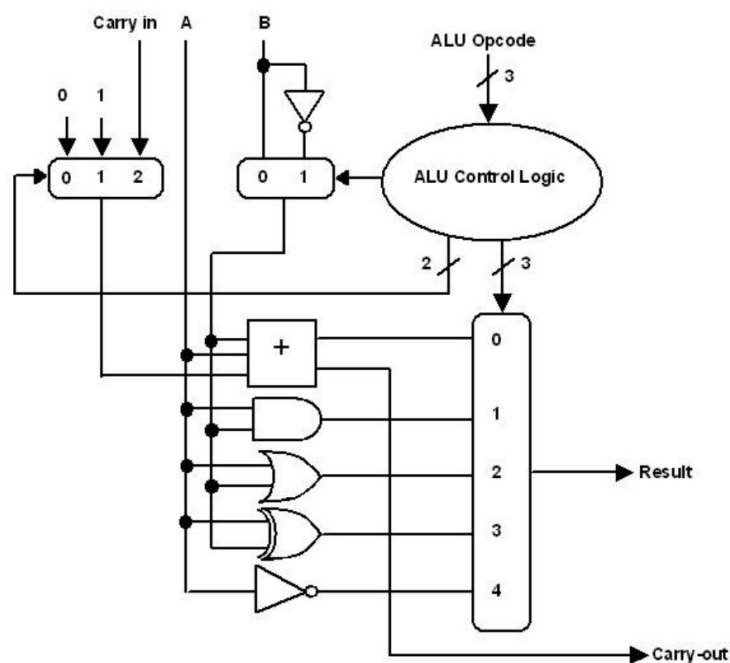


- 
- **CPU**

- Hjernen i en datamaskin
- Kjører program
  - Henter instruksjoner
  - Finner ut hva de gjør
  - Utfører
- Control Unit
  - Her ligger mikroinstruksjonsminne (microprogram control store)
  - Henter (fetch)
  - Dekoder (Decode)
  - Utfører (Execute)
- ALU
  - Register
    - Programteller (PC - Program Counter) - adresse til neste instruksjon
    - Instruksjonsregister (IR) - Instruksjon som utføres nå
    - Generelle heltallsregistre, flyttalsregistre (General Purpose Registers)



- Inni ALU:



- **Program**

- En rekke enkle instruksjoner
- Hver instruksjon utfører en bit av en oppgave
- Instruksjoner og data ligger i minnet
  - Programminnet og dataminnet
- Eksempel: Legg sammen tallene X og Y som ligger i minnet
  - Programmet:
    - Henter X, henter Y, summerer X og Y
- MEN instruksjoner og data ligger i forskjellige deler av minnet
  - I programminne ligger det pekere til dataminnet

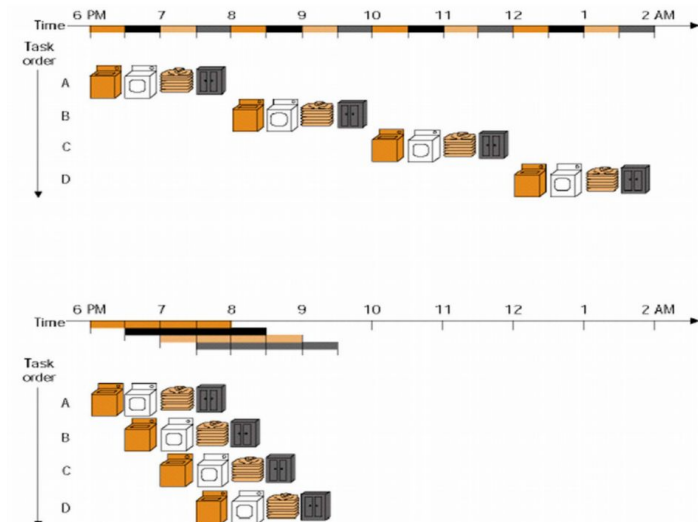
- **Moore's lov**

- Dobling av antall transistorer (på et område) hver 18 mnd (boka sier 18 mnd, andre sier hvert andre år). Dette tilsvarer en økning på 60 % flere transistorer hvert år

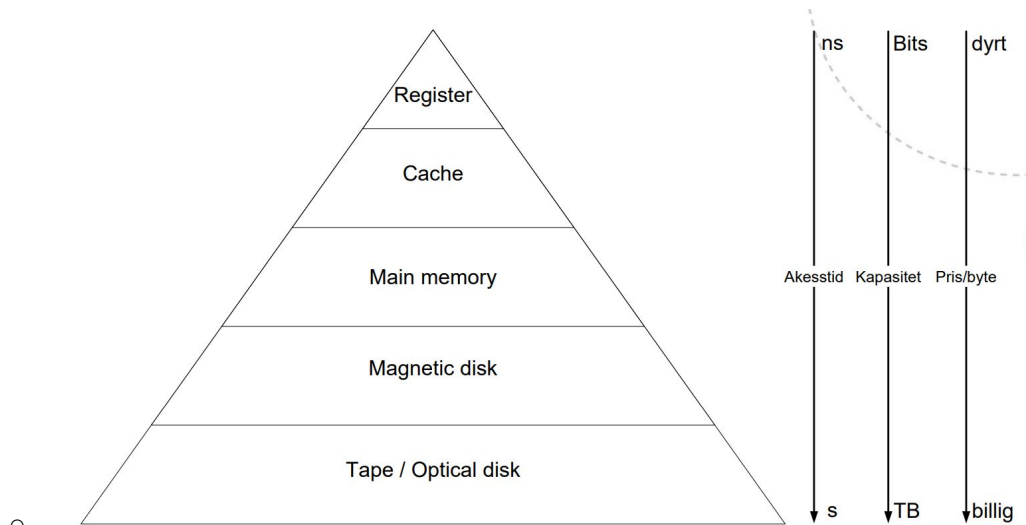
- **Chip Multiprocessors (CMP)**

- Varianter
  - Cache
    - Skal cache deles mellom kjerner?
    - På hvilket nivå skal cache deles?
  - Heterogene eller homogene kjerner?
    - Skal alle kjernene være like?
    - Hvis ikke: hvordan velger software hvilken kjerne?
  - Heterogen eller homogen ISA
    - Skal alle prosessorene ha samme instruksjonssett?
- Positivt
  - Bedre ytelse for én prosess → bedre ytelse for flere prosesser
  - Energisparing via deling av ressurser og redusert klokkehastighet

- Hver instruksjon tar lenger tid, men vi kan kjøre dobbelt så mange instruksjoner
  - Mange små enkle kjerner istedet for store
    - Redusere designkostnader (enkelt å verifisere)
  - Problematisk
    - Høyere minnebelastning
      - Kan ikke øke antall kjerner ubegrenset, siden vi har begrenset cache
    - Høyere I/O belastning
      - Trenger høyere båndbredde til minnesystemet for å holde alle kjernene opptatt
  - Dårlige
    - Cache coherency
      - Cacher må holdes i synk slik at alle kjernene ser samme data
    - Parallellprogrammering er vanskelig
- **Stream Multiprocessor (SM)**
  - Arbeid delt opp i “warps”
  - 32 tråder per warp
  - Hver tråd i en warp kjører samme instruksjon - men på forskjellig data
  - Hver SM har 4 warps - 128 kjerner per SM
- **Parallellitet**
  - Finnes to typer, ILP og PLP
  - ILP - Instruksjonsnivåparallellitet
    - Flere instruksjoner utføres samtidig (1 prosessor/kjerne)
    - Et mål på hvor mange instruksjoner som kan gjøres parallellt
    - Flere samlebånd - høyere ILP, flere instruksjoner behandles samtidig
    - Samlebånd/pipelining
      - Unødvendig at ALU er ledig mens CPU leser instruksjon
      - Jo flere pipelines jo bedre
      - Kan utføre flere instruksjoner samtidig
      - Tregeste trinnet angir hvor lang tid det tar
      - Dype samlebånd er problematisk ved programflyt operasjoner. Da må pipeline tømmes hvis en hopper. Kan føre til at et resultat av en instruksjon ikke er ferdig før det skal brukes av neste instruksjon, får da “stal” på pipelinen
      - Samlebånd med flere steg/dybde vil få høyere ILP
      - Arkitektur:
        - (1) Fetch
        - (2) Decode
        - (3) Execute
        - (4) Memory
        - (5) Writeback



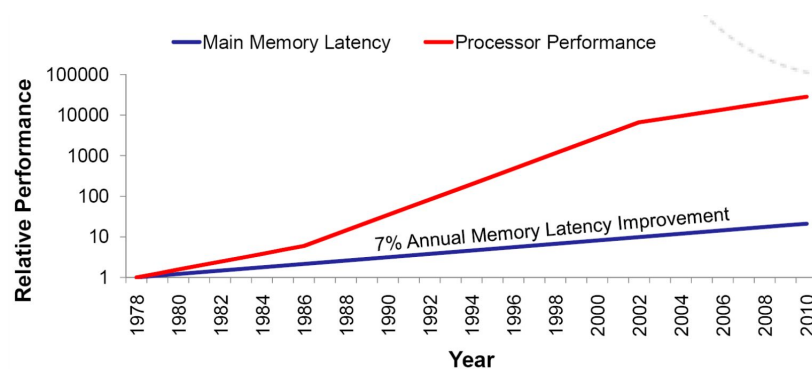
- ILP ikke definert på ISA nivå
  - PLP - Prosessornivå parallellitet (typar maskiner, e.g. SIMD, MIMD)
    - Flere prosessorer/kjerner som jobber i parallell
- **Minne hierarki**
  - Begrenset av
    - Ytelse, fysiske avgrensninger, økonomi og fleksibilitet
  - Hierarki



- Register
  - Register i prosessor (PC, IR)
  - Registerbank
  - Data som prosessoren manipulerer
  - All data må inn i register
  - Raskest
- Hurtigbuffer/cache
  - Mellom prosessoren og primærlageret, mye raskere
  - Lokalitet - Tid og rom
    - Tid - sannsynlig at data/instruksjon blir brukt igjen

- Rom - Sannsynlig at data/instruksjoner aksessert har nærliggende data/instruksjoner som kommer til å bli aksessert. Brukes til å få en effektiv avbildning av minne i feks cache
- Lokalitetsprinsippet
  - Når man ber om et ord fra hovedminnet, antar man at man kommer til å trenge det ordet flere ganger, og at man også kommer til å trenge noen av naboene
  - Derfor blir ordet sammen med noen av naboene kopiert til hurtigbufferet
  - Lokalitetsprinsippet betyr at for å bruke et ord  $k$  ganger trenger man ett tregt kall til hovedminnet og deretter  $k-1$  raske kall til hurtigbufferet.
  - Hva  $k$  kan man regne seg frem til et gjennomsnitt på hvor raskt prosessoren kan jobbe (se mean access time formel)
- Avbildningsfunksjon
  - Når prosessoren forespør en datablokk som ikke finnes i hurtigbufferet må blokken kopieres fra lageret til hurtigbufferet
  - Hurtigbufferet er mindre enn lageret, så man trenger en algoritme for å bestemme hvilken linje man skal kopiere blokken til
  - Direkte avbildning
    - 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
    - Hurtigbufferlinjenummer = lagerblokknr mod  $M$  (totalt antall linjer i hurtigbufferet)
    - Enkelt og ikke dyrt å implementere, men lite fleksibelt siden den har gitte plasser for alle lagerblokker
  - Assosiativ avbildning
    - Lar blokkene bli lastet inn på en hvilken som helst hurtigbuffer
    - Hvis den ikke finnes i bufferet fra før, legges den til og man oppdaterer hvor den ligger
    - Ulempe er at den trenger kompleks og dyr (mtp plass) logikk
  - Sett-assosiativ avbildning
    - Kompromiss av de overnevnte, inneholder egenskaper fra begge
    - Bufferet er delt inn i  $v$  sett med  $k$  linjer hver. Totalt antall linjer  $M$  blir da  $M = v \cdot k$
    - En gitt blokk vil avbildes direkte inn i et gitt sett, avbildningen inne i settet er derimot assosiativ

- Vanlig med flere nivå av hurtigbuffer, gir høyere hastighet og treffrate
- Har to nivåer
  - Nivå 1 er lite og raskt
  - Nivå 2 er stort (gir høy treffrate) og “tregt” - likevel raskere enn primærminnet
  - Nå vanlig å separere data og instruksjoner i hvert sitt hurtigbuffer, da får man færre kollisjoner
  - Vanlig å ha nivå 1 separert og nivå 2 felles
- Erstattningsalgoritmer for hurtigbufferet
  - LRU (least recently used) - Bytter ut den blokken som det er lengst tid siden har blitt brukt
  - FIFO (First inn 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 random
- Aksesstid
  - Gjennomsnittslig
  - Mean access time formel  $c + (1 - h) * m$ 
    - $c$  = cache access time, tiden det tar programmet å hente info fra hurtigbuffer
    - $m$  = main memory access time, tiden det tar å hente info fra hovedminnet
    - $h$  = hit ratio, andelen av referanser som kan hentes fra hurtigbufferet i stedet for hovedminnet
  - Endring i aksesstid nedover i hierarkiet
    - Blir billigere, tregere og større jo lenger ned man kommer i hierarkiet
- Øker ytelsen - latency hiding
- Main memory
- Magnetic disk / Harddisk
  - Stor lagringskapasitet, ikke rask, ikke flyktig, mekanisk
- «Processor Memory Gap»
  - Det er et “gap” mellom utviklingen av prosessorytelse og minneforsinkelse/latency
  - Memory Wall - Prosessorytelsen øker mer enn minneytelsen
  - Latency hiding techniques
    - Teknikker for å skjule forsinkelse ved minneaksess
    - Programmet ser gjennomsnittlig lav aksesstid



- 
- **Superscalar processor**
  - Innfører en eller flere enheter i en prosessor med samleband, feks felles fetch og decode, flere ALUer
  - Håndterer flere instruksjoner i én klokkesykel
  - Innfører ILP i en CPU
- **SIMD processor (Single Instruction-stream Multiple Data-stream)**
  - Består av flere like (multi)prosessorer som gjør samme sekvens av instruksjoner på forskjellige sett av data
  - Typisk array prosessorer
  - GPU avhenger av dette
- **Array processor**
  - Mange identiske prosessorer som synkront gjør de samme operasjonene på datasett med lik struktur og ulikt innhold.
  - Lar datamaskinen utføre store mengder arbeid parallellt
- **Vector processor**
  - Ligner på array processor
  - Dataene som skal jobbes med puttes i en vektor og sendes samlet gjennom en prosessor spesialdesignet for å håndtere slike vektorer
- **Multiprocessor (MIMD - Multiple Instruction-stream Multiple Data-stream)**
  - Flere uavhengige prosessorer - deler samme minne/hovedlager
  - De må koordinere slik at de ikke er i veien for hverandre
  - Enkel arkitektur
    - Mange CPUer, én buss og alle koblet til delt minne
    - Finnes utvidelser av dette → multicomputer
  - CPUene er “tightly coupled”
  - Buss blir flaskehals etter hvert
    - Kan ha litt lokalt lager til hver prosessor
  - Homogen multiprocessor - Like kjerner
  - Heterogen multiprocessor - Ulike kjerner
- **Multicomputer (MIMD)**
  - System bestående av flere computers med sitt private minne, ikke delt minne.
  - Distribuert hovedlager
  - CPUene sender hverandre beskjeder
  - Maskiner med nesten 10.000 prosessorer har blitt laget
  - Lettere å lage en multiprocessor - men vanskeligere å programmere
  - CPUen er “loosely coupled”



- **Generell prosessor**


- Når vi har datamanipulasjon ved ALU, dataflytt og betinget hopp (conditional branch)

## Kapittel 3 - Digital logisk nivå

- **Logiske portar (NAND, NOR, XOR, AND etc)**

NOT   $F = \overline{A}$  (F = not A, F = ~A)

A	F
0	1
1	0

AND   $F = A * B$  (F = AB, F = A and B)


A	B	F
0	0	0
0	1	0
1	0	0
1	1	1

OR   $F = A + B$  (F = A or B)


A	B	F
0	0	0
0	1	1
1	0	1
1	1	1

XOR   $F = A \oplus B$  (F = A xor B)

A	B	F
0	0	0
0	1	1
1	0	1
1	1	0

NAND   $F = \overline{A * B}$  (F =  $\overline{AB}$ , F = A nand B)

A	B	F
0	0	1
0	1	1
1	0	1
1	1	0

NOR   $F = \overline{A + B}$  (F = A nor B)

A	B	F
0	0	1
0	1	0
1	0	0
1	1	0

- **Enkel logikk (alt kan lagast av NOR eller NAND)**

$F = \overline{A * B}$  (F =  $\overline{AB}$ , F = A nand B) ;  $F = \overline{A + B}$

○

$F = \overline{A + B}$  (F =  $\overline{A+B}$ , F = A nor B) ;  $F = \overline{A} * \overline{B}$

○

$$F = \overline{\overline{(A * A)} * \overline{(B * B)}}$$

$$F = \overline{\overline{(A)} * \overline{(B)}}$$

$$F = \overline{\overline{(A)}} + \overline{\overline{(B)}}$$

$$F = A + B$$

○

- **Flyktig minne**

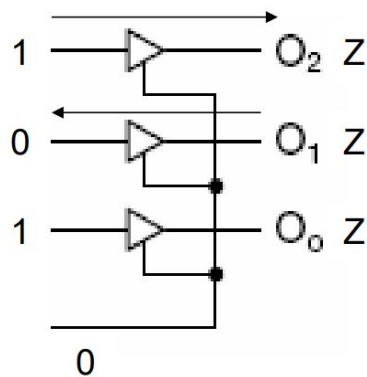
- Statisk RAM (SRAM)
  - Rask - “switch time” til transistor
  - Stor minnecelle (areal) - bruker feks 6 transistorer
  - Stort effektbruk - Mange transistorer, aktive komponenter lagrer bit
  - Må ikke oppfriskes - aktive komponenter lagrer bit
  - Enkelt grensesnitt - data kan i prinsippet leses rett fra cella
- Dynamisk RAM (DRAM)
  - Ikke så rask - lite strøm, passive komponenter, aksess må ikke påvirke oppfriskning
  - Liten minnecelle - bruker 1 transistor og 1 kondensator
  - Lite effektforbruk - svært små ladninger, lite lekkasje og “switch” strøm
  - Må ha oppfriskning - Kondensatoren må oppfriskes, logikk og tid
  - Mer kontrollert grensesnitt (DRAM kontroller) - Logikk for å kontrollere oppfriskning og aksestidspunkt
  - Gir ut data når den har det klart, prosessoren må vente på data, og bussen er opptatt helt til overføringen er ferdig
- Synkron DRAM
  - Ønsket høyere ytelse i primærminne.
  - Styrt av klokkesignalet på systembussen
  - Vil svare prosessoren etter et gitt antall sykler, i mellomtiden kan prosessoren fortsette med andre ting, og systembussen er ledig til annet bruk
  - Mest vanlig form for dynamisk ram i moderne pcer

- **Non-Volatile, ROM (ikke flyktig)**

- ROM - Read Only Memory
  - Lagrer program eller data som aldri skal endres
  - Fast innhold definert ved produksjon
  - Celle bruker lite areal, bit koblet til 1 (VCC) eller 0 (GND)
  - Cella i seg selv bruker ikke strøm, ekstremt lite effektforbruk
  - Stort sett kun for masseproduksjon (billig)
  - Ofte grensesnitt som statisk RAM, men uten skriving/write
- PROM - Programmable Read Only Memory
  - Som ROM, men kan programmeres minst en gang

- Innholdet kan defineres etter produksjon
- Mange typer
  - PROM - programmeres én gang
  - EPROM - (erase) kan slettes (ved å utsettes for UV lys)
  - EEPROM - (electrically) kan slettes elektrisk
  - Ofte lang programmeringstid
- Flash memory
  - EEPROM med rask programmering
  - Billig
  - Programmeres i blokker
  - Kan endres ca 1 million ganger, blir bedre og bedre
  - Solid state harddisk
- **Busser og arkitektur**
  - Kommunikasjonsvei som kobler sammen flere enheter
  - Kun én enhet kan sende noe over en buss på et gitt tidspunkt, men alle de andre enhetene som er koblet til bussen kan motta
  - Består typisk av flere kommunikasjonslinjer, hver med mulighet til å sende et binært signal
  - Kan kategoriseres i
    - Data Buss - Sender data
      - Bredden/antall linjer på databussen bestemmer hvor mye data som kan overføres for hver klokkesykel
      - Ikke nødvendig slik at jo flere datalinjer, jo raskere buss
      - Hvis man sender data over en linje serielt, kan man ofte overføre med høyere hastighet fordi man bl.a. slipper problemene med å få data på alle datalinjene synkronisert
    - Address Bus - Hvor skal info sendes
      - Feks hvis prosessoren skal lese fra en gitt minneadresse, legges denne adressen på adresselinjene slik at minnet vet hvilket ord som skal legges på datalinjene
      - Antall adresselinjer avgjør hvor stort adresseområde som kan nås via denne bussen
    - Control Bus - Hva skal sendes
      - Styrer hva bussen skal brukes til, feks hvem som skal skrive til den
  - Når man kobler mange enheter til en buss vil man generelt få en tregere buss
    - Dette skyldes at bussen blir fysisk lengre og tiden det tar for signalet å komme fra den ene siden av bussen til den andre øker
    - Må da senke frekvensen på bussen slik at man er sikker på at alle enhetene mottok siste signal på bussen før man begynner å sende et nytt
    - Hvis to signaler sendes samtidig vil signalene bli ødelagt
    - Konkurransen om å sende data over bussen vil også øke når vi kobler til flere enheter. Dermed vil enhetene bruke mer tid på å vente på å få kontroll over bussen

- Løser dette ved å dele bussen i flere nivåer, altså flere busser som er koblet sammen med broer
- Multiplekset buss
  - En buss der noen av de fysiske linjene har flere roller
  - Feks 32 datalinjer som også kan fungere som 32 adresselinjer
    - En bussoverføring som tar flere sykler å gjennomføre vil kunne sette av 32 linjer i 1. sykel til adresse, for så å bruke de samme linjene i neste sykel til data
  - Gir færre antall linjer i bussen, og en enklere buss å håndtere
  - Motsatte av multiplekset buss er dedikerte busser
- Arbitrering
  - Busser har master og slave
  - Master bestemmer hvem som skal skrive til bussen
  - Kan også ha flere mastere, da får vi et problem med å styre hvem som skal ha kontrollen over bussen til enhver tid
  - Løses ved arbitrering
    - Enhetene forhandler om hvem som skal styre bussen
    - Deles inn i to grupper; sentral og distribuert
    - Sentral
      - En enhet/busskontroller som har kontroll over bussen
    - Distribuert
      - Ingen sentral kontroller, enhetene må bli enige seg i mellom om hvem som skal bruke bussen
      - “daisy chained”
  - Dersom arbitrering skjer uten å “bruke opp” sykler for dataoverføringen, kalles den “skjult”
- Busser er enten synkrone eller asynkrone
  - Synkron - Klokke som avgjør når og hvor lenge et signal skal ligge på bussen. Sykeltiden må tilpasses slik at den er lang nok for den tregeste enheten. Har en klokke som bestemmer hvor lenge data skal være tilgjengelig og når data skal leses og skrives
  - Asynkron/handshaking - Styrelinjer/control bus forteller at det ligger et gyldig signal på bussen, slik at den enheten som skal lese dette vet at signalet er klart for lesing. Tilsvarende vil det finnes styrelinjer som forteller at signalet er lest, og dermed kan fjernes. Data ligger tilgjengelig helt til slaven har indikert at den er ferdig (kvitteringssignal)
- I CPU ligger Internal Processor Bus
- Må ha en protokoll, så man vet hva som skal gjøres i bussen. Det som er koblet til bussen må vite hvordan de skal “snakke” sammen
- **Three state buffer**
  - Buffer som gjør at man kan koble fra busslinjer
  - Tre pinner - inn, ut og kontroll
  - To tilstander - Tilkoblet, høy impedans (three state)



- ...
- Forstå blokk digram av f.eks:
  - CPU
    - Kva er ein instruksjon
    - Korleis utføres instruksjonar
  - Logikk i CPU einingar (Register, ALU etc)
  - Adressedekoding og hva adresserom er
    - Minnekart
    - Hva ligger på adressene
    - Kan akseptere Register ved å lese/skrive til minneadresser
  - Sekvensielllogikk (Geiski kapitel)
  - FSM ligningar, tabell, statediagram
    - Flip-Flops kva typar (me brukar D-vippe)
    - Moore maskin (state drevet)
    - Mealy maskin (input drevet)
    - Forstå eit state diagram
    - Finne likningar for ein enkel krets
    - Sette opp likningar for ei enkel maskin

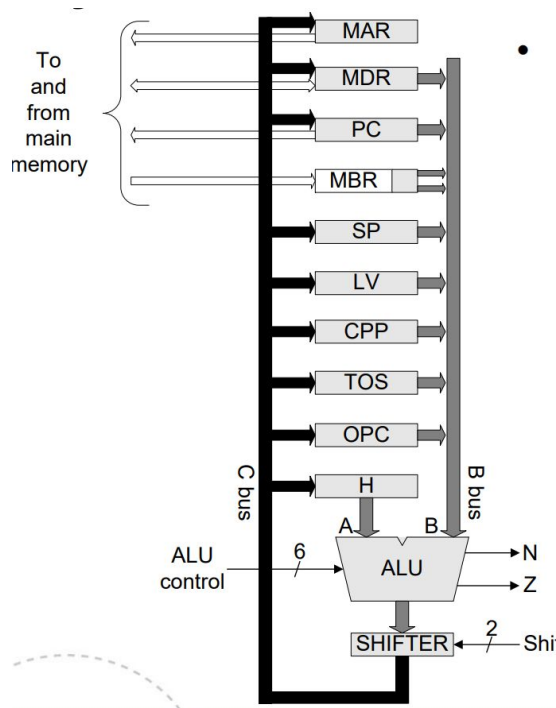
## Kapittel 4 - Mikroarkitekturnivå

- Dette nivået skal realisere ISA
  - Hvilke funksjonelle enheter er nødvendige (generelt)
  - Hvilke funksjonelle enheter er nødvendige for å oppfylle spesifikasjon
- Bruker **IJVM - Integer Java Virtual Machine** som et eksempel på mikroarkitektur i dette faget
- **Datapath**
  - ALU-funksjoner, + - AND OR NOT osv
  - Instruksjonssett (stack, top down)

Hex	Mnemonic	Meaning
0x60	IADD	Pop two words from stack; push their sum
0x64	ISUB	Pop two words from stack; push their difference
0x7E	IAND	Pop two words from stack; push Boolean AND
0x80	IOR	Pop two words from stack; push Boolean OR
0x84	IINC <i>varnum const</i>	Add a constant to a local variable

○

- Antall register (de 4 første kan sende til og fra hovedminne)
  - (Dataminne) MAR - Memory Address Register - kan ikke skrive til ALU. Data som skal skrives til minne (ved write-signal) eller data som er lest fra minne (ved read-signal).
  - (Dataminne) MDR - Memory Data Register - Peker til aktiv minneadresse (for MDR)
  - (Programminne) PC - Program Counter - Peker på instruksjon i programminne
  - (Programminne) MBR - Memory Buffer Register - OpCode for instruksjon som er hentet (fetch) fra programminne
    - Har to kontrollsignaler til B-bus, en for signed og en for unsigned operasjoner
  - SP - Stack Pointer
  - LV - Local variable
  - CPP - Constant Pool Pointer
  - TOS - Top of Stack
  - OPC - OpCode register
  - H - Holding register
  - Shifter - styrt shiftregister og ALU utverdi

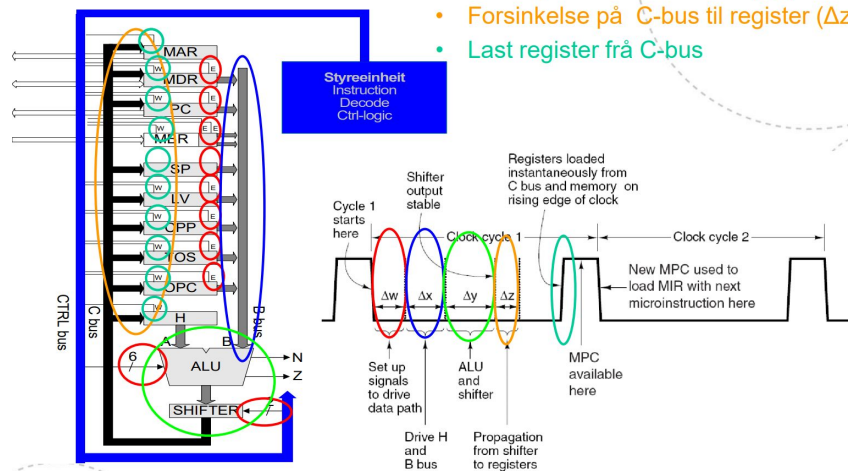


- Antall busser
  - Sender via B-bus til ALU
  - Kan sende lagrede verdier fra H-register via A-bus til ALU
  - Sender output av ALU-operasjonen ut på C-bus
  - En svart pil under et register indikerer et kontrollsignal som skriver/load til registeret fra C-busen
  - En hvit pil indikerer enable til B-bus
- Shifter
  - To kontrollinjer, SLL8 og SRA1

- SLL8 - Shift left logical 1 byte (8 bit)
  - Shifter innhold 8 plasser mot venstre
  - Fyller LSB (least significant bit) med 0
    - AA55AA55 før
    - 55AA5500 etter
- SRA1 - Shift right arithmetic 1
  - Shift 1 bit mot høyre
  - La MSB være uendret
    - 1100 1000 ---- 1000 1000 før
    - 1010 0100 ---- 0100 0100 etter
- Ytelse (parallelitet)
- Instruksjonstyper

32

## Utføre instruksjon

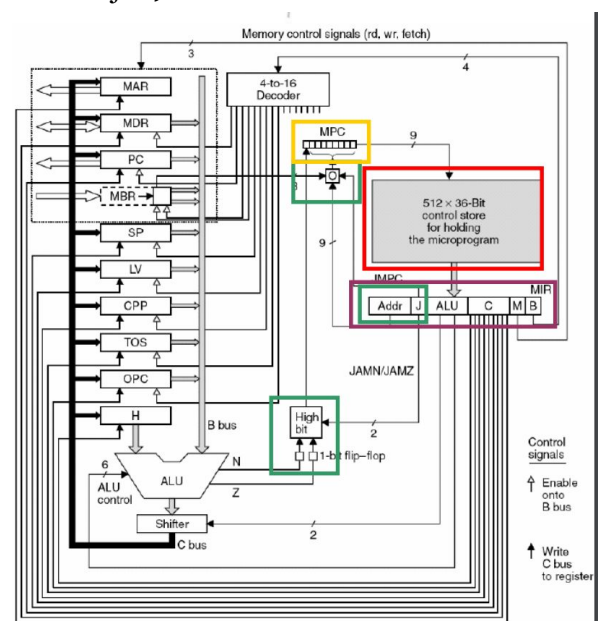
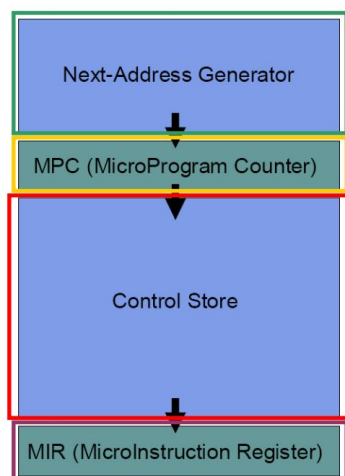


- Kontroll av datapath
  - Styreenhet, gitt av arkitektur som av RISC og CISC
  - CISC - Complex Instruction Set Computer
    - Avanserte instruksjoner - kan utføre mye
      - Bruke flere enheter
      - Nærmere høynivå språk
    - Fleksibelt instruksjonssett (hvis mikroprogram kan oppdateres/endres)
    - Variabel lengde på instruksjoner
    - Variabel tid på å utføre instruksjoner
    - Mikroprogram (ikke alltid)
    - Mange typer instruksjoner og adresseringsmåter
    - Komplisert kontrollenhet, ofte vanskelig å lage et godt samleband til
    - Ofte mange adresseringsmodi, med mulighet for operander i RAM
    - Få generelle regisre
  - RISC - Reduced Instruction Set Computer
    - Enkle instruksjoner

- Register til register instruksjoner
  - Bruke mange enkle instruksjoner på å utføre oppgaver
  - Én instruksjon per sykel (pluss evt samlebåndsoppholding)
- Fast instruksjonsett (Hardwired)
- Fast lengde på instruksjoner
- Fast tid på å utføre instruksjoner
- En maksinvare statemaskin som styreenhet (eller helst bare logikk)
- Kun én type adressering LOAD/STORE
- Rask
- Enkle adresseringsmodi
- Stor registerfil, LOAD/STORE medfører at man stort sett bare jobber på registre, mange genrelle registre gjør at man slipper å bruke sykler på å lese/skrive til RAM
- **Flagg i ALU**
  - Z (zero) og N (negative) flagg
  - Verdiene lagres i en 1-bit flipflop/vippe, på stigende klokkekurve
  - B-bus er ikke lenger aktiv når klokkekurven stiger og klokken=1, dette gjør at ALU outputene ikke lenger kan antas å være korrekt
  - Bruker derfor flaggene til å lagre ALU i flipfloppene, som gjør verdiene tilgjengelige og stabile uavhengig av hva som skjer i ALUen
- **Status register**
  - Register der bit gir status til enheter eller resultat av instruksjon
  - Bit i status register: flagg
  - IJVM status
    - JAMZ: Sjekker Z-flagget (zero) til ALU
    - JAMN: Sjekker N-flagget (negative) til ALU
      - JAMZ og JAMN kan påvirke MPC
    - JMPC: Ekstra hopping kommer senere
  - Nå mulig å detektere N og Z for å kunne bestemme hopp
    - Kan sjekke resultat fra beregning utført av ALU
    - Kan bruke resultatet til betinget hopp
- **Hopp**
  - Ubetinget hopp (Unconditional Branch eller jump)
    - Hopper alltid (goto adr xxxx)
  - Betinget hopp (Conditional Branch)
    - Hopper hvis Z-flagg=1 eller N-flagg=1, goto adr xxxx
    - MPC setter til hopp mikroinstruksjon
      - Mikroinstruksjon som utfører et hopp
      - Mikroinstruksjon med JMPC bit = 1
  - MPC: 9 bit
    - 8 fra addr i mikroinstruksjon
    - Hvis JAMZ eller JAMN = 0
      - MPC = Addr
      - Neste mikroins Addr



- Hvis JAMZ eller JAMN = 1
    - MPC = Adr or 100000000
    - Neste mikroins 1xxxxxxx
  - På denne måte kan MPC ha 9 bit istedet for 8
- **En-sykel maksin**
  - Én instruksjon hver klokkesykel
  - Enkle instruksjoner, Add, Load, Store, Copy
- **Fler-sykel maksin**
  - Flere instruksjoner per klokkesykel
  - Komplekse instruksjoner
- **Styreenhet**
  - Next-Address Generator
  - Program Counter - PC (inneholder instruksjon opCode (definert på ISA-nivå)
    - OpCode verdi angir peker til første mikroinstruksjon i instruksjon.
  - MPC - MicroProgram Counter (innholder adressen til neste mikroinstruksjon i control store)
  - MIR - Micro Instruction Register (, holder nåværende mikroinstruksjon, styreord, alle signal for å styre datapath, ikke tilgjengelig for programmereren, ingen måte å adressere direkte (Shift, ALU, B-buss og oppdatere MPC))
  - Control Store (Innholder mikroinstruksjoner (styreord, adresse til neste mikroinstruksjon og om det er en branch instruksjon)



- 
- Har nå en generell datamaskin
  - Styreenhet (Control Unit)
    - Mikroprogram MPC
    - Control inputs - MBR (OpCode)
    - Control output - Read/Write/Fetch og PC
    - Status signaler - N og Z flagg
    - Control signal - MIR
  - Utførende enhet (datapath)
    - Register

- ALU og Shifter
- Data input/output - MDR
- Status signaler - N-flagg og Z-flagg

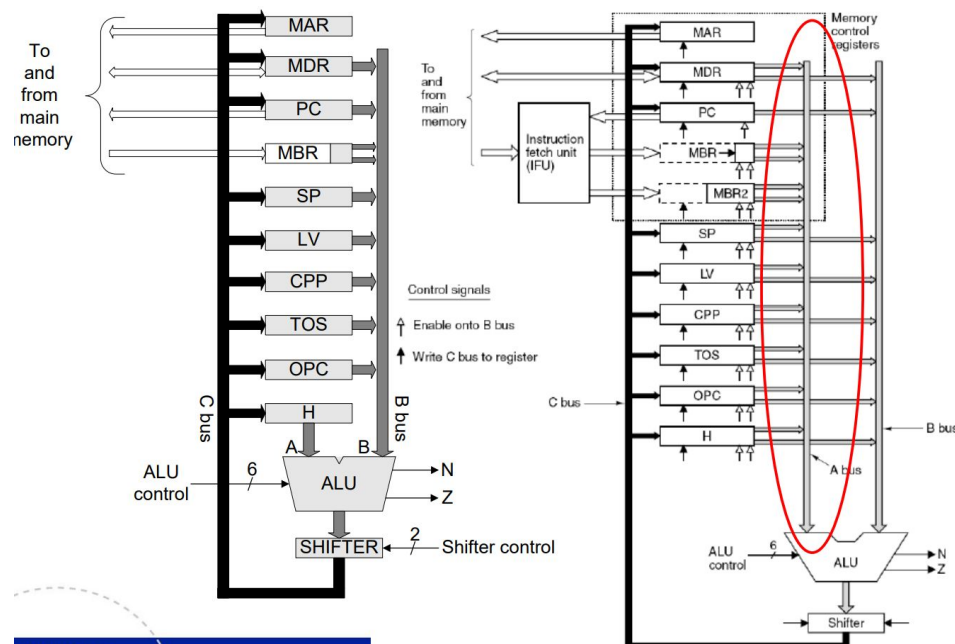
No har me gått gjennom eit eksempel på korleis ein prosessor kan vere bygdopp, har kontroll på korleis IJVM virkar. Vidare er det metodar for å endre ytelse parameter (her auke hastigheit på instruksjonsutføring/berekning throughput). Huks at ytelse er gitt ut frå ytelsemål. Ytelsemål kan, som for for IJVM, vere hastigheit men andre parameter er også mulig, f.eks. silisiumsareal eller effektforbruk.

Viktig å få med seg: Når mikroarkitekturen endrast så endrast ikkje ISA. Alle variantar av IJVM kan utføre dei same instruksjonane.

IJVM mikroarkitektur endringar for å auke ytelse (kva og korleis):

- **Innfører A-buss.**

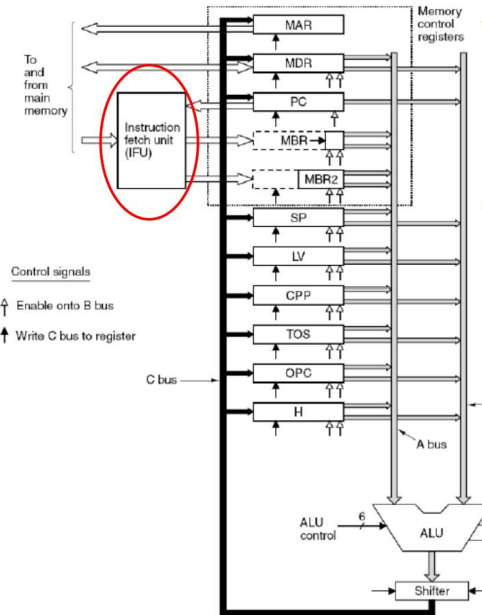
- Sparer klokkesykler
- Må ikke lengre ha ene operand i H-register
- Må utvide MIR med et felt for å styre A-buss (som for B-buss)
- Har nå to register tilgjengelig for ALU
- Må ha felt for A-buss i Control Unit, Control Store og MIR
- Kontrollsignal til register for 2 busser - 4 til 16 dekodere, så 4 bit for hver bus
- A-buss utvidning krever ekstra areal og logikk



- **Innfører Instruction Fetch Unit (IFU)**

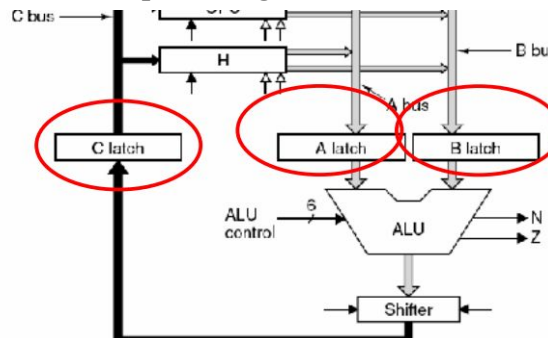
- Henter neste instruksjon automatisk/automagisk
- Oppdaterer PC uten at vi må bruke datapath (klokkesykler) (auto increment av PC)
- Har en kø av instruksjonar og operander (operander fra programminne), slipper å vente
- Slipper å vente på instruksjoner fra MIR og kontrollenhet har instruksjoner hele tiden
- Unntak - ved conditional branch (betinget hopp) må man kunne oppdatere PC uten å bruke auto increment

- I IJVM IFU er det valgt å stoppe instruksjon henting til en vet om branch-betingelser er oppfylt eller ikke (se også pipelining)
- IFU krever ekstra logikk og dermed ekstra areal og effektforbruk.



### ● Innfører pipelining

- Deler opp instruksjon i biter som kan utføres uavhengig av hvarandre
- Kan da korte klokkeperioden til det lengste (trege) pipeline trinnet
- Husk fra tidligere: Fetch, Decode, Execute (legg og til writeback)
  - Raskt - enklere trinn og kortere tid
- Brukar latch/register til å skille trinn i datapath
  - Legger inn latch på A, B og C buss

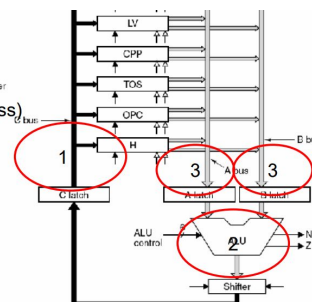


- IJVM datapath pipeline; 3 trinn (A/B-buss, ALU, C-buss)
  - I vanlig program utfører man en og en instruksjon, men med pipeline kan man gjøre flere samtidig

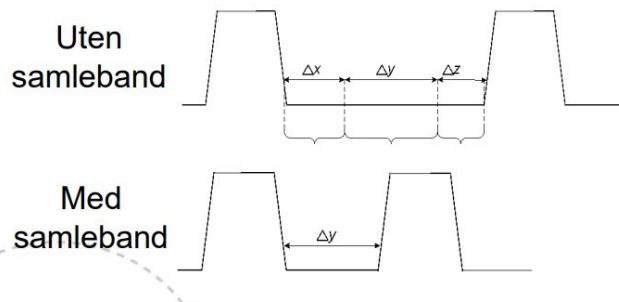
#### I samleband

- Inst 1 (A/B buss)
- Inst 1 (ALU) Inst 2 (A/B buss)
- Inst 1 (C buss), Inst 2 (ALU), Inst 3 (A/B buss)
- Inst 2 (C buss), Inst 3 (ALU)
- Inst 3 (C buss)

Alle 3 instruksjonar ferdig



- Utover datapath: Fetch, Decode ♣ Utvider til 5 og 7 trinn
  - 7 steg
    - Hent instruksjoner (IFU)
    - Finn type og lengde på instruksjon (Decoder)(nytt)
    - Finn mikroprogram og legg mikroinstruksjonene i dette mikroprogrammet i en kø (Queue) (nytt)
    - Hent operander (Operands)
    - Utfør ALU-operasjon (Execute=
    - Skriv til register (Writeback)
    - Hovedlageraksess (Memory)
- Kan minke klokkeperioden - aka tregeste trinn



- Pipeliner må handtere avhengigheter (RAW, WAW, WAR) sjå ISA
  - RAW - Read after Write - leser fra et register som en tidligere instruksjon skriver til
  - WAW - Write after Write - skriver til et register som en tidligere instruksjon skriver til
  - WAR - Write after Read - skriver til et register som en tidligere instruksjon leser til
- Pipeline må handtere branches («feil» instruksjonar i pipelinen ved hopp)
- **Flytkontroll**
  - Dynamisk endring av instruksjonsrekkefølge under utføring
  - Typer
    - (Betinget) hopp
    - Prosedyrekall
    - Ko-rutiner
    - Trap/avbrudd (interrupt)

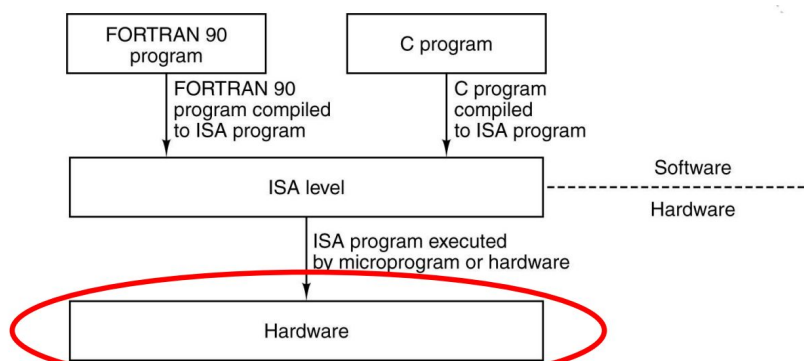
## Kapittel 5 ISA

ISA Opprinnelig det einaste nivået, definerar instruksjonar logisk og minnemodellar.

Instruksjonar:

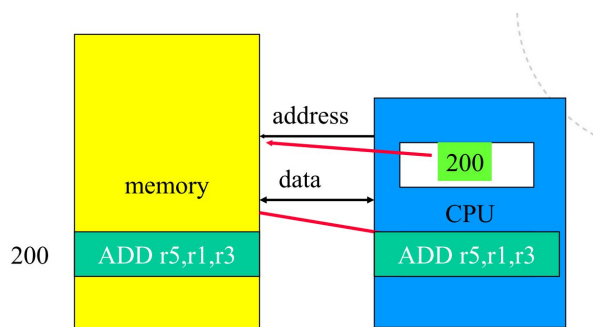
- **ISA - Instruksjonssettarkitektur**
  - Første nivå tilgjengelig for (ekspert-)brukere
  - Grense mellom maskinvare og programvare
  - Opprinnelig det eneste nivået, språket er maskinkode
  - Endrer man instruksjonssett, ender man ISA

- Grad av ILP er ikke definert på ISA nivå, ISA kan ha forskjellige typer mikroarkitekturer med forskjellig ILP

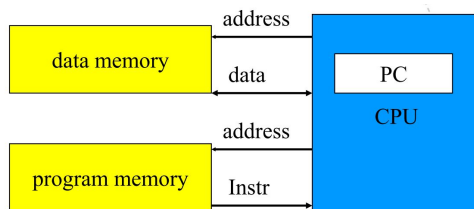


- **Adresserom**

- Minnemodell omfatter
  - Størrelse på adresser (antall bit) - normalt 32 bit som gir 4 G adresser
  - Adresserbar enhet - normalt 8 bit, men 1-60 bits har eksistert
  - Organisering i større enheter - ord på typisk 32 bit eller 64 bit, mange instruksjoner manipulerer ord om gangen
- Normalt har man kun ett adresserom - felles for data og instruksjoner
- Alternativ: Forskjellige adresserom
  - En instruksjon på adresse 8 og et dataelement på adresse 8 ligger ikke samme sted
  - 32 bits adresse gir dermed  $2 \cdot 2^{32}$  lokasjoner
  - Skriveoperasjoner kan ikke ødelegge instruksjoner
- NB! Ikke det samme som delt hurtigbuffer
  - Kan ha delt hurtigbuffer med ett adresserom
- Von Neumann arkitektur



- **von Neumann architecture computer**
- Harvard arkitektur
  - Separate busser til data og instruksjoner



**Harvard architecture**

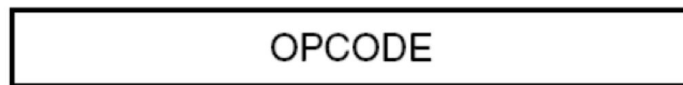
- **Type (instruction format (fig. 5.9 bok), antall operander, o adr, 1, adr, 2, adr, 3, adr osv**

- Hvilke instruksjoner som finnes og hvordan de fungerer er helt sentralt
- Viktigste typene

- Flytting av data
- Aritmetiske operasjoner
- Sammenligninger og hoppinstruksjoner
- Prosedyrekall
- Løkker
- I/O

- o-adresseinstruksjoner

- Ingen eksplisitte operander i instruksjonen
- Eks: ADD
- Bruker stakk for operander og svar
- Unntak Push og Pop



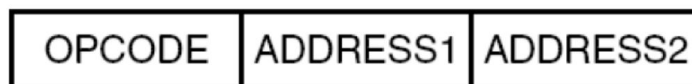
- 1-adresseinstruksjon

- Én eksplisitt operand per instruksjon
- Eks: Add A
- Andre operander er implisitt - alltid funnet i hardware register, kjent som "akkumulator"



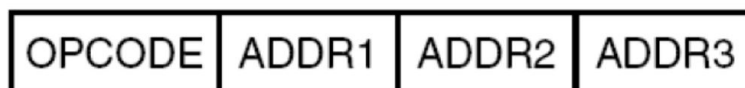
- 2-adresseinstruksjon

- To eksplisitte operander per instruksjon
- Eks: ADD B, A
- Resultatet overskriver en av operandene
- Operandene er kjent som kilde eller destinasjon
- Fungerer bra for instruksjoner som memory copy



- 3-adresseinstruksjon

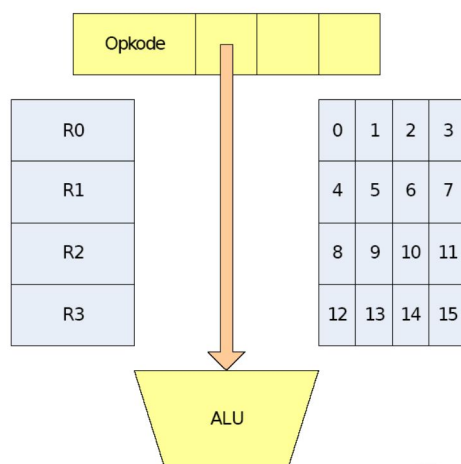
- Tre eksplisitte operander per instruksjon
- Eks: ADD D, B, A
- Spesifisert som kilde, destinasjon og resultat



- Operander som spesifiseres som kilde

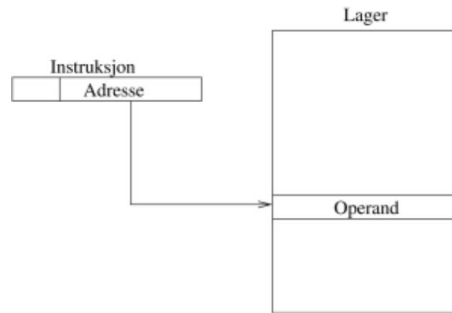
- Signed constant
- Unsigned constant

- Innhold i register
  - Verdi i en minnelokasjon
- Operander som spesifiseres som destinasjon
  - Single register
  - Par av sammenhengende register
  - Minnelokasjon
- von Neumann Bottleneck
  - Tiden det tar å utføre minne aksessering kan begrense den overordnede ytelsen
  - Unngår dette ved teknikker som feks å begrense de fleste operandene til registrene.
- Implisitt
- **Adresseringsmodi**
  - En instruksjon er ofte avhengig av operander for å kunne utføres
  - Måten instruksjonen angir hvor data skal hentes fra kalles en adresseringsmodus
    - Regel for tolkning av adressefelt
    - Mål: effektiv adresse → peker på operand
    - Spesifiseres av OpCode eller eget modus-felt i instruksjonen
    - Forskjellige operander i samme instruksjon kan ha forskjellig adresseringsmodus
- **Hvordan operander håndteres, f.eks:**
  - Immediate
    - Operander ligger direkte i instruksjonen
    - Dermed tilgjengelig uten videre
    - Størrelse på operand begrenset av feltlengde
    - Kan bare brukes til konstanter/integer - verdi bestemmes ved kompilering



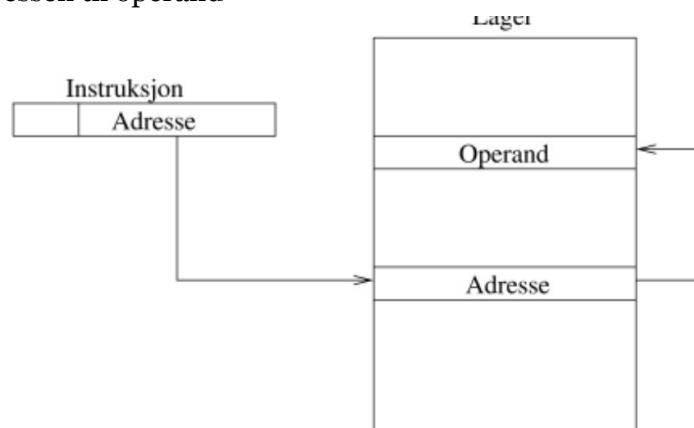
- 
- Direct
  - Instruksjonen angir adressen til operand i RAM
  - Vil alltid aksessere samme minnelokasjon, verdien kan endres, men lokasjonen kan ikke

- Kan bare brukes til å nå globale variabler som har kjent adresse ved kompilering



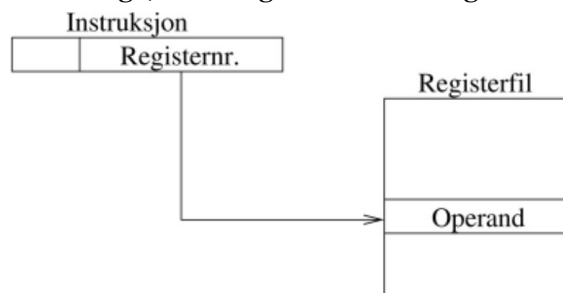
- Indirekte

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



- Register

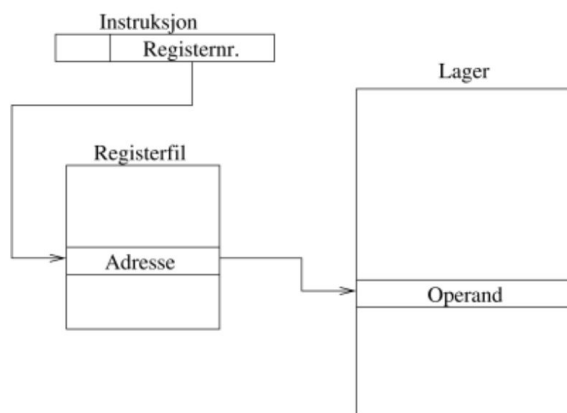
- Instruksjon har nummer på register som inneholder operand
- Samme som direct, men gir deg register i stedet for minnelokasjon
- Mest vanlige, fordi register er så viktige



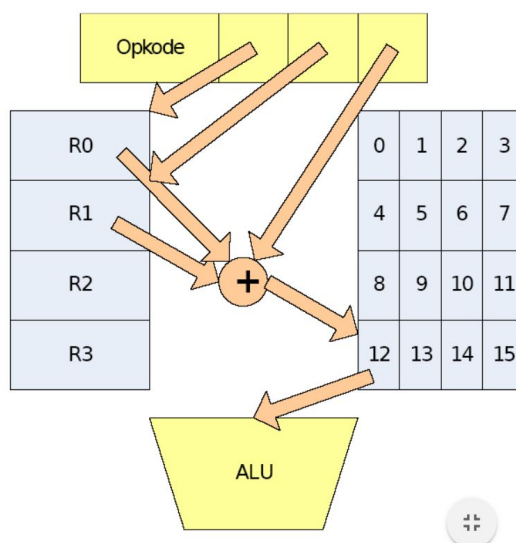
- Register Indirect

- Instruksjonsfelt inneholder registernummer
- Register inneholder hovedlageradresse - peker
- Registernummer krever færre bit enn hovedlageradresse
- Fleksibelt





- 
- Indeksert
  - Adresserer minne ved å gi et register pluss en konstant
- Baseindeksert
  - Instruksjon inneholder to registernummer
  - Hovedlageradresse er summen av innholdet i disse to registrene
  - Kan ha offset i tillegg
  - Eks: MOV R4, (R2 + R5)



- 
- Hva de forskjellige er:
  - LOAD/STORE = Register Indirect
  - ALU = Register
  - MOVC = Immediate - konstanten er en immediate operand, som blir henta fra programminne som en del av instruksjonen
  - CP = Register
  - BZ = Register Indirect
- **Instruksjonslengde**
  - Fast
    - Enklere dekoding
    - Fordel for samlebånd - spesielt superskalare
    - Men: mulighet for sløsing med plass
  - Variabel

- Vanlig før
  - Lengde og oppbygging av instruksjoner
  - Koding av instruksjonar, f.eks. (tabell forelesing 17, fig 5.12 bok)
- **Instruksjonssett**
  - Komplekst/enkelt
    - Type instruksjoner
    - Lengde
    - AdrModi
  - Påvirker
    - Dekoding (CTRL-logikk)
    - Utføring (antall “states” for instruksjoner)
    - Bruk av datapath
  - Må implementeres i mikroarkitektur
- **Instruksjons typer**
  - Datatransport
  - Manipulering (Dyadic, monadic)
  - Sammenligning, betinga hopp
  - Prosedyrekall
  - Løkker
  - I/O (inkludert interrupt)
    - Busy-waiting / programmert I/O (polling)
      - CPU tester/leser status av I/O (loop)
      - CPU utfører loop til aktuell statusendring oppstår
      - Står og venter til den får svar, leser status til I/O enhet i loop
    - Interrupt
      - Enhet gir signal til CPU ved behov
      - CPU kan utføre andre oppgaver
      - CPU har innebygde mekanismer for å detektere og håndtere avbrudd
      - Interrupt vector
        - Peker
      - Sendes IRQ signal (interrupt request), da må man
        - Endre programflyt
        - Lagre status
        - Utføre IRQ koden
        - Gjenoppretter status
        - Tilbake til aktivt program

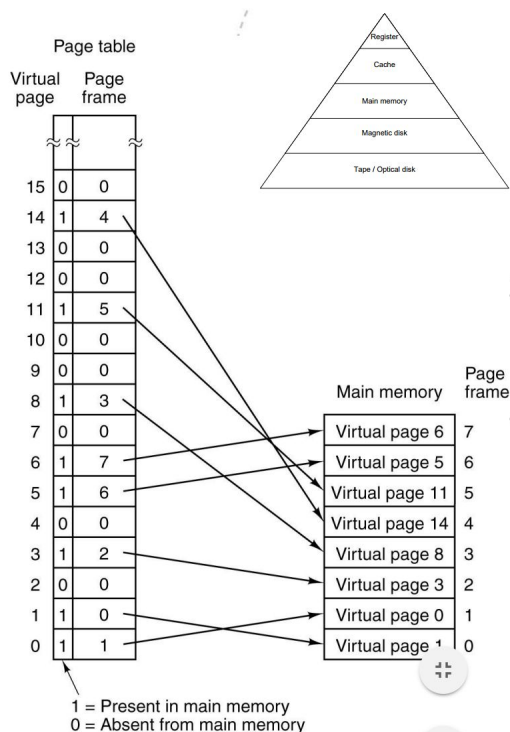
Forstå kva instruksjonar gjer ut frå spesifikasjon.

Flyttal: representasjon av nummer i datamaskiner. Korleis det kan gjerast (LES appendiks).

## Kapittel 6 - Virtuelt minne

- **Prosess vs program**

- Et program blir en eller flere prosesser når mn kjører det
- En prosess er instruksjoner, initialiserte data og ikke-initialiserte data i minnet
- En loader tar for seg programmet, laster det inn i minnet og initialiserer det som skal initialiseres
- **Preemption Control Flow**
  - OS skifter prosess ved interrupt
  - Interrupt gitt av timer (HW-timer)
  - Timer gir avbrudd etter gitt tid (n antall clk perioder)
  - Avbruddsrutine - skift prosess
  - OS hit HW-ressurser til annen prosess
  - Ved skifte
    - Må lagre data til prosess n
    - Last data til prosess m
- **Før virtuelt minne måtte programmereren passe på å ikke bruke for mye minne på egenhånd**
  - Programmet ble organisert som en samling overlays
    - Data og instruksjoner som passer i minnet
- Mål med virtuelt minne var derfor å unngå at programmereren må ta hensyn til at det bare er en begrenset mengde fysisk minne tilgjengelig
  - Hver prosess kan ha sitt eget adresserom
  - Passe på at forskjellige prosesser ikke aksesserer hverandres minne
  - Gjør størrelsen på adresserommet kun avhengig av antall bit i adressen
- **Paging**
  - Ide: Del opp minnet i blokker med fast størrelse kalt pages
    - Hver prosess har sitt virtuelle adresserom
    - Dette mappes til en eller flere pages som flyttes mellom disk og minne ved behov
    - Oversettelse mellom virtuel adresse og fysisk adresse nødvendig
  - Oversettelsen gjøres av Memory Management Unit (MMU)
  - Usynlig for software (transparent)
  - Implementering
    - Ide: bruk en tabell til å bytte ut de mest signifikante adressebitene
  - Minne vs disk
    - Noen av sidene er i minnet og andre er på disk
  - Vi bruker ett bit i sidetabellen for å fortelle om en side er i minnet eller på disk



- **Demand paging**

- Store adresserom gir store pagetabeller
  - Men: Få programmer bruker hele adresserommet
- Ide: Ikke allokere pages før de blir aksessert første gang
  - Kalles demand paging
- På et hvert tidspunkt finnes det et subset av alle pages som prosessen bruker
  - Kalles working set
  - En utvidet definisjon brukes også på hurtigbuffer

- **Page replacement policy**

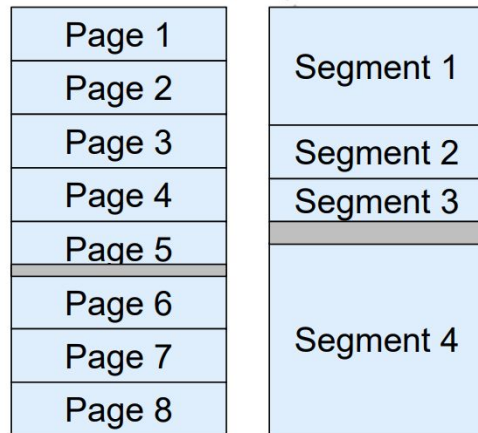
- Lokalitetsprinsippet gjelder også for pagereferanser
- Hvis minnet er fullt må vi bestemme oss for hvilken page som skal kastes ut
  - LRU, FIFO osv
- Samme avveining som for hurtigbuffer
- Page misses krever aksess til disk
  - Kostnaden ved feil avgjørelse er større enn i et hurtigbuffer

- **Segmentering**

- Ide: Separerer adresserom for forskjellige datastrukturer i et program
- Segmentering tilbyr flere uavhengige adresserom med forskjellig størrelse

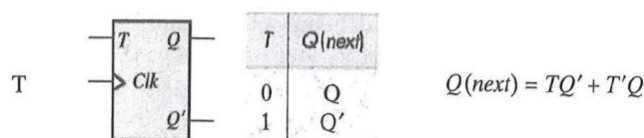
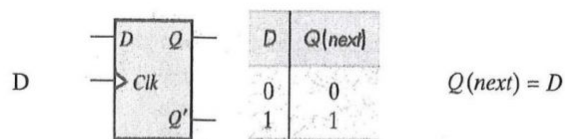
- **Fragmentasjon**

- Paging gir internal fragmentation
  - Vanskelig å bruke all plass internt i en page
  - Små pages vs store pages
- Segmentering gir external fragmentation
  - Vanskelig å få utnyttet all plass mellom segmenter

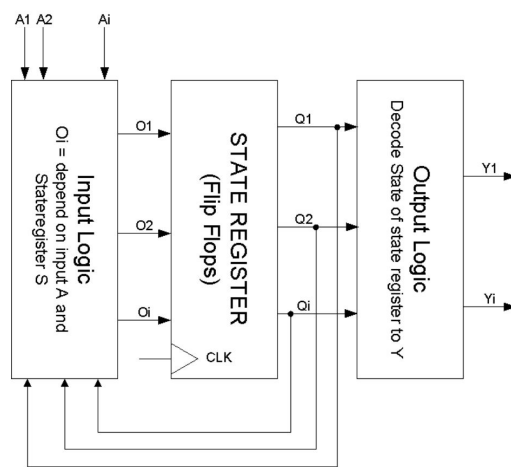


## Kapittel 6 (fra kretsbooka) - Sekvensiell logikk

- **Sekvensiell krets**
  - Tilbakekobling
  - Minne
  - $\text{Next} = F(\text{input}, \text{state})$
  - Tilstandsmaskin
- **Kontrollenheten i IJVM er en state machine**
  - MIR er minnedelen
  - MPC og CTRL er logiske krets delen
- **D-vippe**
  - Når  $D=0$  er  $Q=0$ , når  $D=1$  er  $Q=1$
- **T-vippe (Toggle)**
  - Når  $T=0$  er  $Q_{\text{nxt}} = Q$ , når  $T=1$  er  $Q_{\text{next}} = Q'$



- **Finite State Machine (FSM)**
  - To typer
  - State based Moore
    - Output kun avhengig av state



- Input based Mealy
  - Input påvirker output direkte

