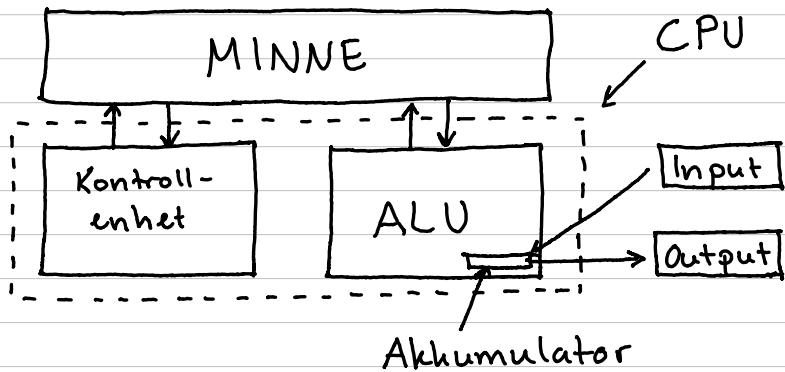


Datamaskiner  
&  
Digitalteknikk

TDT4160  
2020

# Datamaskinenes Historie,

- Ada Lovelace: Programmeringsspråkets mor  
1800-tallet
- Alan Turing: Definerer hva datamaskiner kan  
(alt!)
- 1948: Første transistor, erstatter vakuumrør, kan  
kontrolleres med spenning
- John von Neumann
  - + Sent 1940 / tidlig 1950
  - + foreslo hvordan arkitekturen til en datamaskin  
burde se ut
  - + von Neumann-arkitektur:



- + Ingen forskjell på data og program! Alt ligger i minnet, og kan byttes ut for å gjøre andre ting.
- 1958: Integrerte kretser (logiske porter)
  - ↳ Flere transistorer på én brikke
- + Starten på Moores lov: Hvert år dobles ant.  
transistorer på en brikke
- 1960: "Minicomputers" med display, PDP-1

- 1971: Intel 4004 Mikroprosessor - Verdens første!
  - ↳ Hva kan den?

Det er en generell datamaskin, så if.  
Turing kan den beregne alle beregnebare funksjoner.
- Herfra er endringen i hastighet/ytelse.
- 2006: Sliter med energiforbruk og kompleksitet
  - Begynner å forenkle prosessorene igjen, men bruker nå flere parallelt (multicore prosessor)
- 2014 → nå
  - + GPU + GPU-clusters
  - + AI
  - + 2D Tile Pattern Stacking av Silisium-brikker for å fordele varmen

## Datamaskinarkitektur

1. Low-level electronic design
  - ↳ enkeltkomponenter som transistorer, de minste
2. Kretsdesign / Digitalt logisk nivå
  - ↳ Settes sammen kretser med funksjonalitet
  - ↳ ALU, register osv.
3. Datamaskinarkitektur / ISA-nivå
  - ↳ Settes sammen kretsdesign til mikroarkitektur
  - ↳ Definerer alle instruksjoner prosessoren kan kjøre
4. System software / operativsystem
5. Applikasjoner

# System

- Hvordan ting er satt sammen (CPU, minne, I/O, buss)

## CPU

- CPU-en ligger inni prosessoren
- Alt utenfor CPU-en har en adresse, og vi må peke på det ← Datamaskina er et stort minnekart!
- CPU-en forholder seg til resten av systemet via disse addressene via busser
- Parallelitet: viser at ytelsen er kjempebra, men vanskelig å programmere.
  - ↳ 1. Instruksjonsnivåparallelitet (ILP)
    - + Flere instruksjoner utføres samtidig av én prosessor
    - Alle moderne maskiner har dette!
    - Pipelining: Alle de forskjellige delene av CPU-en kan jobbe samtidig med samme/ulike instruksjoner
    - + Kan også deles opp mer → øker klokkefrekvensen
  - 2. Prosessornivåparallelitet (PLP)
    - + Flere prosessorer, ulike møter å bygge de sammen på
      - ↳ △ SIMD og MIMD

## • Flynn's Taxonomy

+ SISD: Single instruction, single dataset

+ MISD: Multiple — " — "

↳ Hver prosessor har tilgang til sin egen bit av et program (eller eget), men opererer på samme data

+ SIMD: Prosessorene kjører samme instruksjon, men på ulike datasett. Typisk for GPU-er.

+ MIMD: Hver prosessor har egne program og data. Kan kjøre helt uavhengige av hverandre om ønskelig

# Minne

- Memory/processor-gap: Ytelsen for prosessorer vokser raskere enn ytelsen til minne...
- Main memory
  - + Alt minne som er tilgjengelig (RAM + Harddisk)
  - + Minne vi kan lese av og skrive til
- Hierarki
  1. Register: Like raskt som prosessoren. Nanosekunder
    - + Data prosessoren manipulerer aka register ALU-en jobber på
    - + Ligger inne i CPU-kjernen
    - + Kontrolllogikken har direkte tilgang
  2. Hurtigbuffer/Cache: Det prosessoren skrives til og leser fra. Nanosekunder.
    - + Ofte L1 + L2
    - + Ligger mellom CPU-en og hovedminnet
    - + Gennomsnittlig aksessstid:

$$\text{Cache aksessstid} \rightarrow C + \underbrace{(1 - h)}_{\substack{\text{Sannsynlighetene} \\ \text{for at vi ikke finner} \\ \text{det vi leter etter i cache}}} \cdot m \leftarrow \text{main memory aksessstid}$$

3. Hovedminne: RAM, GB-store.  $\mu\text{s}$  - ms.

1-3. er flyktig, altså går tapt når strømmen kuttes.

4. Magnetisk disk: SSD, TB-store. ms

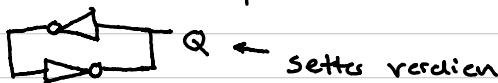
5. Tape/opdisk disk/online: Oftest utenfor maskina. m/t

- Processor/memory-gap minskes ved at instruksjoner kopieres gradvis over fra treigest til raskest minne. Byttes ut underveis mens programmet kjører. Som regel vil man da finne det man trenger i det raskeste minnet.

Access pattern: 90% → 9% → 1% (raskest aksessers oftest)

# Ram

- Sekvensielt minne tar variert tid å få tak i basert på hvor det fysiske er lagret, mens med RAM kan vi hente ut hva som helst med like kort tid uansett
- Statisk RAM (SRAM)
  - ↳ Raskt, stor minnecelle, bruker mye energi, lett å interface
  - ↳ Kan sees på som to NOR-poter i en sirkel
  - ↳ Typisk brukt i register og cache



- Dynamisk RAM (DRAM)
  - ↳ Ikke like raskt, lite areal og energi forbruk, må ha oppfriskning
  - ↳ Mer komplisert grensesnitt (DRAM-kontroller)
  - ↳ Lades enten minnecella opp til 1 eller ut til 0 for å skrive til den
  - ↳ Verdien leses fra kondensatoren

Kapitel

3

Digital Logistik  
Nivå

# Digitalt Logisk Nivå

## • Analoge vs. digitale verdier

↳ Analog signaler er kontinuerlige i tid og verdi. For å oversette dem til digitale, må vi sample dem (hente ut en verdi per en viss tidsenhet)

↳ De analoge signalene settes til 0 og 1 basert på terskelverdi

## • Logiske porter

↳ Dobbelt så mange mulige outputs som faste inputs (1/0)

△ XOR: Gir 1 om inngangene er ulike, 0 om de er like

## • Three-state-buffer

↳ "Bryter" som girer at vi kan koble to punkt på en ledning

↳ Tre pins: inn/ut/kontroll

↳ To tilstander: "High impedance"/tilkobla

↳ Går ikke strøm gjennom, gir ingen verdi

↳ Brukes for å koble av/på register fra bussen

## • Timing i logiske porter

↳ Hvor lang tid det tar å endre en verdi

↳ Bestemmes av lengste sti målt i porter  
(må være mindre enn klokkeperioden)

## • Sekvensielle kretser

↳ Kretser med minne (tid spiller en rolle)

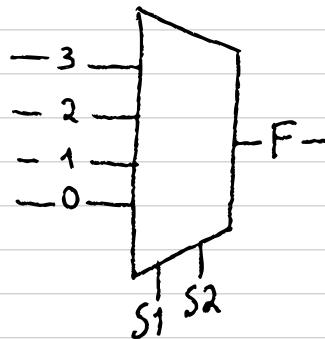
△ D-vippe, Flip-flop

↳ △ "Lagre det i inngang D på utgang Q dersom CK=1"

↳ Trenger like mange lagringselementer som bit som skal lagres, ettersom hver kun kan lagre én bit

↳ Kan brukes til å bygge opp register

# Komponenter



S1	S2	F
0	0	Inn 0
0	1	Inn 1
1	0	Inn 2
1	1	Inn 3

## Multiplexer

n antall innninger + et sett Select-innganger.

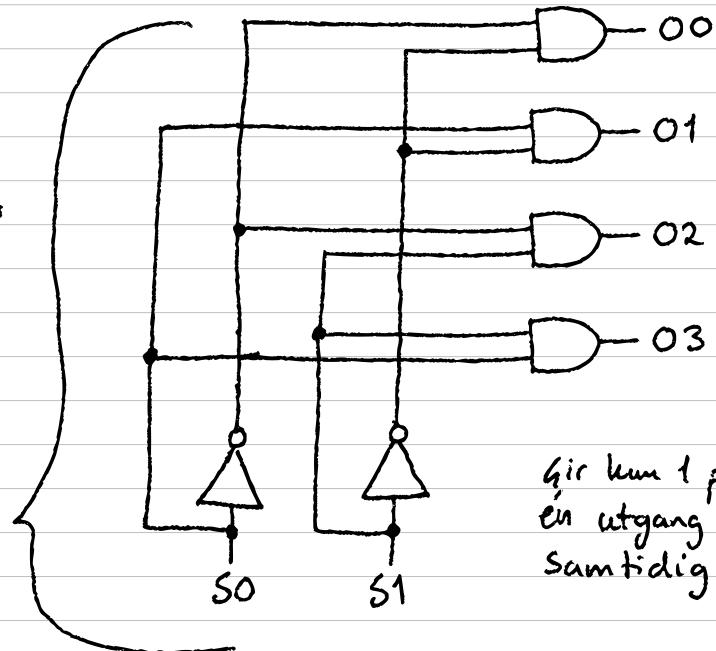
Med 2 S-inganger kan vi ha 4 muligheter til output (hver 1/0-kombo knyttes en spesifikk inngang til F)

## Dekoder

Ingen inn ganger,  
kun select og ut

Select bestemmer  
hvilke utganger  
som skal være 1

4 AND-porter +  
2 NOT-porter



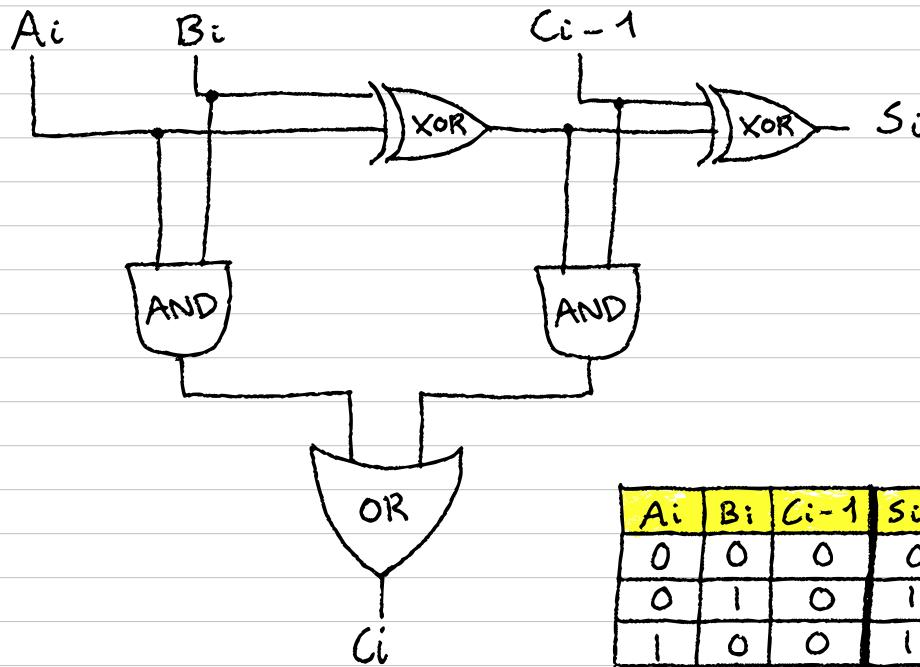
Gir kun 1 på  
en utgang  
Samtidig

S0	S1	00	01	02	03
0	0	1	0	0	0
0	1	0	0	1	0
1	0	0	1	0	0
1	1	0	0	0	1

# Komponenter

## Binær adder

Legger sammen. Kan få carry bit:  $1+1=0$  og  $C_i=1$   
Full (forenkla) adder:



$A_i$	$B_i$	$C_{i-1}$	$S_i$	$C_i$
0	0	0	0	0
0	1	0	1	0
1	0	0	1	0
1	1	0	0	1
0	0	1	1	0
0	1	1	0	1
1	0	1	0	1
1	1	1	1	1

## ALU

Kan utføre alle de logiske operasjonene + binær-addisjon.

Output bestemmes av en multipleser.

# Busser

- Deles på, og kan start sett kun brukes av to enheter samtidig
- Samling med fysiske ledninger. 1 bit/ledning
- Signaltyper:
  1. Adressebuss: overfører addresser/peker på adressa vi vil ha
  2. Databuss: Der dataen overføres
  3. Kontrollbuss: Overfører kontrollsignal
- Bussprotokollen: Hvordan blir signal lagt ut? Hvor lenge skal det være aktitt? Hva er min/max?
- Synkron buss
  - ↳ Alle operasjoner på bussen er synkronisert til en tilhørende klokke
  - ↳ Går et bestemt ant. klokkeperioder mellom hver gang det skytes noe på bussen
- Asynkron buss
  - ↳ Ingen klokke! Bruker handshakes isteden
  - ↳ MSYN (master sync) sender signal ved å f.eks. gai fra  $1 \rightarrow 0$  SSYN (slave sync) svart ved å gi øre det samme når det har blitt lagt ut data på bussen som er klar til å leses av
- Multiplesesbuss
  - ↳ Kan bytte mellom ulike bussignaler og destinasjoner. Sist gi øres ved bruke av kontrollporta ("latches")
  - ↳ Spares ant. pins (gjenbrukes), areal og kompleksitet
- Arbitrering: Hva skyter hvis to dinger vil bruke bussen samtidig?
  1. Sentralisert arbitrering: Prioriterer i nivåer. Ting sender requests for å bruke bussen. Nærmest på høyest nivå først.
  2. Desentralisert arbitrering: Ingen nivå. Nærmest arbitreresen først. Sier ifra at bussen er "busy".

Kapitel

4

Mikroarchitektur -  
Nivå

# Mikroarkitektur

- Realiserer ISA
  - ↳ Maskinvareimplementasjonen til et instruksjonssett
- Hva/da henger delene sammen? Hva er best sammensetning?
- En-sykkel-maskin: en instruksjon/klokkeperiode. Enkle instr.
- Fler-sykkel-maskin: flere klokkeperioder/syklus fordi mer "komplekse" instruksjoner. Mer relevant nå instr. inneholder minneadresser.

## IJVM (Integrated Java VM)

- Fler-sykkel stack-maskin
- Instruksjonstyper:
  1. Dataflytt: kopier fra hit til dit
  2. Dataomanipulasjon: ALU-operasjoner
  3. Betinga hopp: Velges programmet basert på visse betingelser
- Instruksjonssett (stack)
  - ↳ Variablene har ikke en hardkodet adresse, ligge i en stack med peker til bunn og aktuelt element
  - ↳ Popper ett og ett element inn i prosessorkjernen
    - + Kan gjøres av instr. Som ADD osv.
- Datapath
  - ↳ Registrerne MAR, MDR, MBR og PC er ansvarlige for å få data til/fra hovedminnet vårt
  - ↳ H-registeret (Holding) er alene ansvarlig for 1 av 2 ALU-innganger. Det andre har alltså kontroll over MAR tilgang til
  - ↳ ALU-control: 6 kontrolllinjer spesifisert 16 mulige ALU-funksj.
  - ↳ Shifter-control: SHIFTER ligger under ALU-en, 2 kontrollsignaler
    - 1. SLL8 : SLS 1 Byte
    - 2. SRA1 : Shift Right Arithmetic, 1 bit mot høyre
      - ! La MSD være uendra for å beholde +/- tall

- Minnelengde
  - + Dataminnet er 32-bit
  - + Programminnet er 8-bit (opcode er koda med 8 bit)
- Minnepath
  1. MAR-registeret peker på minnet
  2. Innholdet hentes og legges i MDR-registeret
  3. PC peker på neste instruksjon (ligger i programminnet)
  4. Denne leses av til MBR-registeret (8 bit start)

## Control Unit

- Mikroprogrammet: Hver instruksjon er bygd opp av flere mikro-instruksjoner.
  - ↳ En mikroinstruksjon kan f.eks. være "Send data gjennom ALU-en 1 gang"
- MIR-registeret: Kontrollerer datapathen pr. klokkefrelvens
  - ↳ 3 adresser som kontrollerer minnet:
    1. Fetch: Aktiveret → Hent opcode med adresse gitt i PC og legg i MBR
    2. Read: Leser fra adressa som ligger i MAR og legg det i MDR
    3. Write: Skriv det som ligger i MDR til adressa som ligger i MAR
  - ↳ Innholdet også: Addr. til neste mikroinstruks. + hvordan den velges, ALU+SHIFTER-funksj., B-Bus source, hvilke register C-Bus skal skriva til
- CIRL-store: Minnebrukke (ROM: read only)
  - ↳ Holdes mikroinstruksjonene (512 · 36-bit register) som peker på av 9-bit MPC-peker og sendes til MIR

## Programflykt

+ Sti velges vha. flagg

↳ Z-flagg: Sjekker om null

↳ N-flagg: Sjekker om negativt

+ Et hopp gjøres ved å endre MPC-en

↳ Gjøres via JAMZ / JAMN / JAMPC (s. 256)

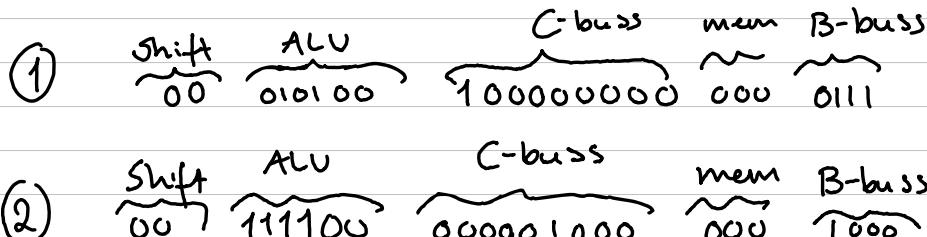
• JAMZ sjekker Z-flagget til ALU, JAMN N-flagget

I MBR ligger →      • JAMPC laster MBR → MPC vis 0, eurs next address  
hovedinstr. + 9-bit i MPC, kun 8-bits addresser

↳ Siste bit blir satt fra Z eller N dersom vi skal sjekke de (JAMZ eller JAMN = 1). Dersom denne bit-en = 1 er vi i øvreste addresseområde i CTRL-store (256 høyeste), altså bit 9 = 1

## MIR-registrer eksempel:

$$SP = TOS + OPC$$



# Forbedret Ytelse

- Hovedsakelig hastighet: Hvor lang tid tar det å utføre en instruksjon? Ser på i/s

- Hvordan øker vi hastigheten?

- + Legges til A-buss: færre mikroinstr. men må ha A-felt i MIR  
+ 4 ekstra bit pr. instr. i CTRL-store.

Krever også litt større fysisk areal for bus

- + Legges til IFU (instruction fetch unit)

↳ Redusere ant. klokke pulsers som trøys for å hente i

↳ Styres oppdateringa av PC slite at vi slipper i hele tiden brukte busser og ALU på det

↳ Holdes en klokke av instruksjoner og operander

- Øker areaal (adder, shift-registrer for kro, logikk for styring (FSM/mikrooperasjoner) og dekoding MBR2 = operande, MBR1 = OpCode)
    - Hentet av en prefetcher

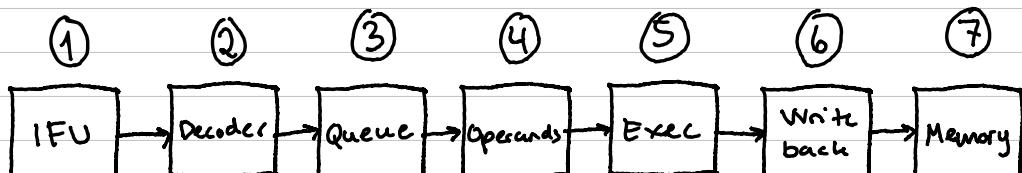
- ## + Pipelining

↳ Deler opp dataparten med latches på bussene for å kunne øke klokkefrekvensen

↳ Problem: RAW dependence, trenger at noe annet  
sljes først... må vente (stalling)

↳ Tar flere rundes i fullfør en tank, men rundene tar kortere tid! Klokkefellesskap reduseres ned til den freigeste operasjonen vår!

## ↳ Seven-Stage Pipeline MIC4:



## + Superskalaritet

- ↳ Henter 2 instr. av gangen fra samme k6 av instruksjoner (kjører to ulike pipelines)
  - ↳ Optil 10 instr. under utføring
  - ↳ Dobles areal og energiforbruk...
    - + Kan isteden kun duplisere mest tidskrevende Steg! (men krever ekstra logikk)

↳ Problem: Pipelines går egentlig i parallell! Vi har

- + Dependencies: intr. Kan være avhengig av hverandre

four 11

**HAZARDS!** Kan gi feil svar. Ikke alle dependencies må føre til Hazards, men kan.

- \ | + Kan legge til logikk for å flagse det

- True dependencies: én instr. leser det en annen skrev
  - Name dependencies: to instr. bruker samme register
  - Control dependencies: intr. avhengig av resultat av br

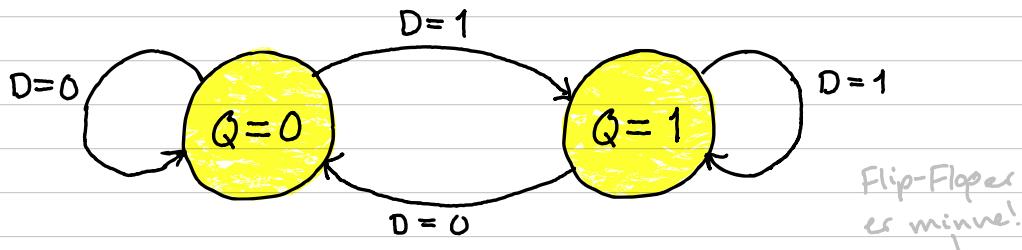
- Data hazards: Resultat av true dependencies. ÅRAWS, KAWRS
  - Control hazards: - " (branches). Begynt på feil instr.
  - Structural hazards: Pipelinen støtter ikke kombinasjonen av instr. fysisk. Å Kan ikke lyse to READs samtidig elns.

# Sekvensiell Logikk

- Tilbakekobling: Output avhenger av input og output av førre tilstand (tilstandsmaskin)
  - ↳ Has en vanlig output-utgang + en utgang til et minne som lagrer output og kan mate det tilbake igjen som input v/ neste klokkeslag
  - ↳ Pass på at timingen er sånn at den lengste stien for tilbakekobling < 1 klokkeperiode!

## • Flip-Flop:

- ↳ Output Q endres via D dersom  $CLK = 1$



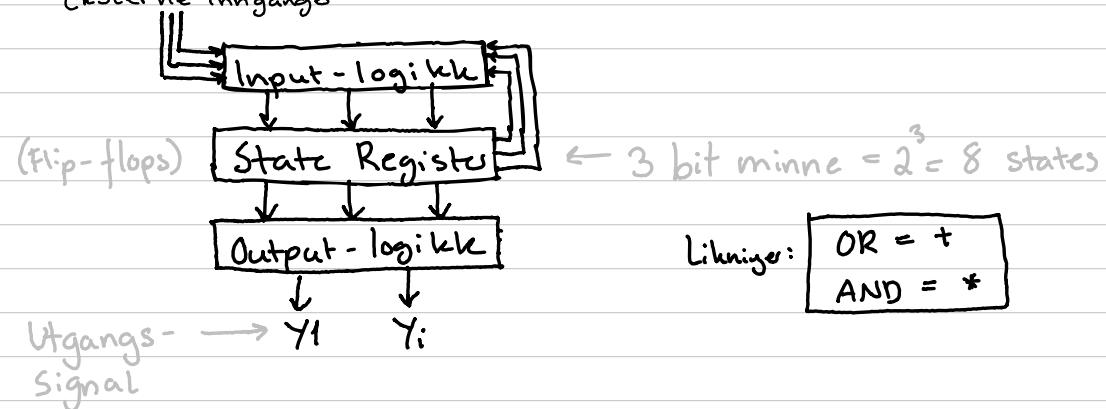
- ↳ Antall tilstander bestemmes av ant. bits i minnet  
Én flip-flop har én  $Q \Rightarrow 2^1 = 2$  tilstander.  
System med to flip-flopper?  $\Rightarrow 2^2 = 4$  tilstander.
- ↳ Det er vanlig å sette opp likninger for D med Q
  - $\Delta D = \bar{Q}$  ( $D = Q$  invertert)

# Finite State Machine

## Moore FSM

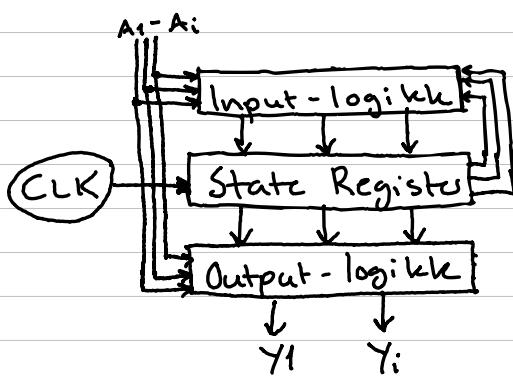
- State based: Endrer kun output når tilstandsregisteret endrer seg
- Alt skjer synkront med klokka

Eksterne innganger



## Mealy FSM

- Input based: Output kobla til input i tillegg til minnet
- Tilstandsregisteret endrer kun verdi om CLK er aktivt (1) men output kan fra dette oppdateres momentant!



Kapitel

5

ISA-nivå

# ISA

- Komplekse instr.  $\Rightarrow$  Store styringsenheter, mye logikk
- Enkle  $\Rightarrow$  Hver bruker bare én klokkesyklus, lite areal/varme  
 $\hookrightarrow$  men raslet! + Pipeline

## CISC (Complex Instruction Set Computer)

- $\hookrightarrow$   $\Delta$ instr.: ADD minneSvar, minneAdr X, minneAdr Y
- + Variabel tid på å utføre instr. med variante lengde
- + Fleksibel instruksjonssett
- + JVM er en CISC-maschine (mange mikroinst. per instr.)

## RISC (Reduced Instruction Set Computer)

- $\hookrightarrow$   $\Delta$ instr.: LOAD R7, R1  
LOAD R8, R2  
ADD R0, R1, R2  
STR R7, R0

- + Register-til-register instruksjoner (enkle)
- + Fast instruksjonssett (Hardwired)
- + Fast lengde og fast tid på utførelse
- + Raskt
- + Kun LOAD/STORE som adressering
- + Dekodingslogikk enkel

## Instruksjoner

- $\hookrightarrow$  Instruksjoner er bygd opp av opCodes og operander, kallas "adresseinstruksjoner" etter hvor mange operander som brukes
- + 0-addresseinstruksjoner: Kun opcode, ingen operander  
 $\Delta$  POP, PUSH, NOP
- + 1-addresseinstruksjoner: Opcode + 1 adressefelt (operand)  
 $\Delta$  ADD

Addresseringsmodi: Hvordan oppgis vi en operand?

↳ Regel for tolking av addressefelt

+ Spesifiseres av opcode eller eget modus-felt i instr.

+ Forskjellige operander i samme instruksjon kan ha forskjellig addresseringsmodus

+ Immediate addressing: Operand er direkte verdi, hentes sammen med opcode. Må være bestemt ved compile time!

+ Register-indirekte addressing: Operand angis et register, som igjen peker til en minneadr.

+ Baseindexert addressing: Operand angis 2 register med hver sin adresse. Disse blir summert til én adresse i hovedminnet. Kan endres på i run-time. En av de kan også brukes som en "baseadresse"/"offset".  
Mai enten ha →  
addres tilgjengelig,  
eller bruke tilb.  
i en blokkesyklus

+ En instruksjon kan ha samme havn, men ulike opcode, og dermed støtte ulike addresseringsmodi (éin pr. modi)

Instruksjonskoding

↳ Må ha en måte å kunne kode for å støtte alle instr. typer

# OS/Minne

## Preemption Control Flow

- OS skifter prosess ved interrupt vha. timer i HW
  - ↳ Gir HW-ressurser til annen prosess
- Kan gi ulike prosesser ulik prioritet
- Når prosess skifts ut, må alle register lagres i minnet og fyldes på nytt av prosessen som tar over

Program på disk → Proses i minnet

## Virtuelt minne

- Virtuelt* ↗
- ↳ Unngår at programmereren må ta hensyn til begrenset mengde fysiske minne tilgjengelig
  - + Hver prosess kan ha eget adresserom ( $2^{64}$ -bits adres.som)
  - + Passte på at prosesser ikke addreserer hverandres minne
    - ↳ Delte opp i pages (blokker minne med fast str.)
  - + MMU (memory management unit) mapper mellom virtuelt og fysiske adresse
    - ↳ Usynlig for software
    - ↳ Bruker tabell til å bytte ut mest signifikante adressebits
    - ↳ Noen av sidene er i minnet og andre er på disk
      - Bruker kontrollbit i tabellen for å sjekke
  - + Demand\_paging: ikke alloker pages før de blir aksessert første gang
  - + Page replacement policy: Hvorvid minnet feilt? Hva skal ut? Muligheter:
    1. LRU (Least Frequently Used)
    2. FIFO

# Fysikk

Motstand  $\rightarrow R = U/I$   
 (spenning/strøm)

$$\Rightarrow P = U^2/R$$

Volt/amper  
 $(\Omega)$   
 ohm

$$P = U \cdot I \quad \leftarrow$$

Strøm  
 (ladning/tid)  
 Coulomb/s  
 $(A)$   
 amper

Effekt  
 (energi/tid)  
 Joule/s  
 $(W)$   
 watt

Spenning  
 (energi/ladning)  
 Joule/Coulomb  
 $(V)$   
 volt

- Strøm endres når spenning endres og motsatt, men motstand er konstant! Mai derfor bruke  $P = U^2/R$
- i dette tilfellet.

/ / / / )

# Datamaskinarkitektur

1. Low-level electronic design
  - ↳ enkeltkomponenter som transistorer, de minste
2. Kretsdesign / Digitalt logisk nivå
  - ↳ Setter sammen kretser med funksjonalitet
    - Δ ALU, register osv.
3. Datamaskinarkitektur / ISA-nivå
  - ↳ Setter sammen kretsdesign til mikroarkitektur
  - ↳ Definerer alle instruksjoner prosessoren kan kjøre
4. System software / operativsystem
5. Applikasjoner