UNIVERSITETET
I OSLO

# Dagens tema

- Dagens tema er hentet fra kapittel 4.3 og 4.4
- Mer om pipelining
  - Ytelse
  - Hasarder
- Pipelining i Pentium-arkitekturen
- Mikrokode
  - Hard-wired
  - Mikroprogrammert
- RISC og CISC
- Fordeler og ulemper

2008

# Repetisjon - Pipelining

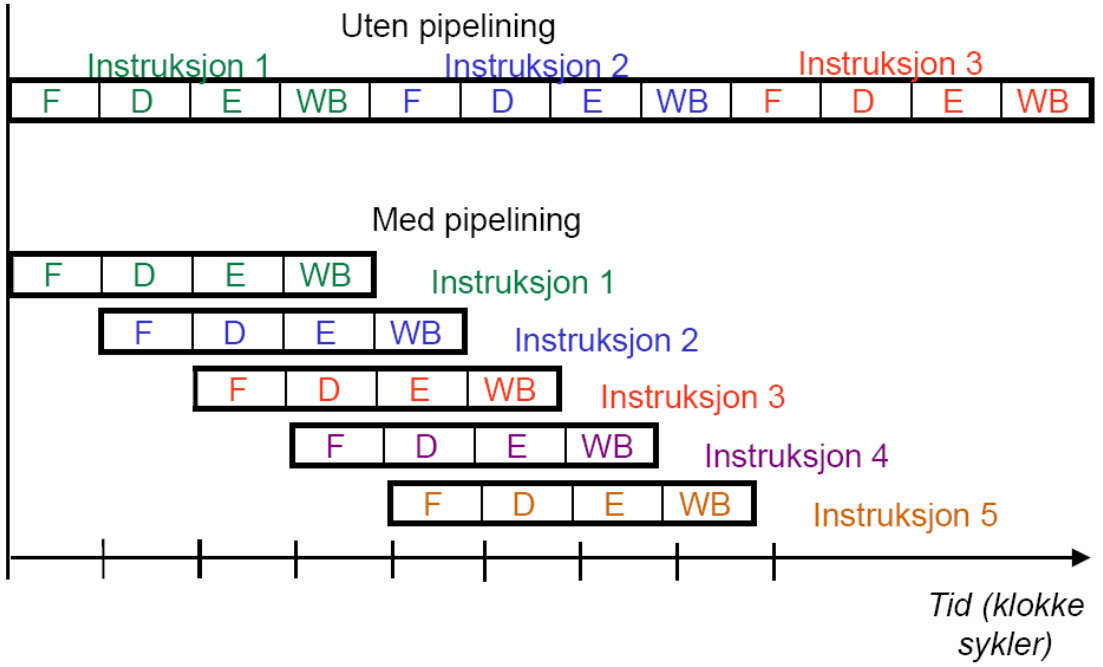
- Pipelining er en teknikk for å øke hastigheten til en CPU
- Utførelsen av de fleste instruksjoner er ganske komplekse og trenger mange klokkesykler for å bli eksekvert ferdig
- Idèen bak pipelining er å starte eksekvering av en ny instruksjon hver klokkesykel
- Kan sammenlignes med samlebåndsproduksjon:
  - Isteden for å vente til forrige instruksjon er ferdig eksekvert, setter man i gang neste instruksjon så fort som første steg av forrige instruksjon er ferdig.
- Med pipelining øker man antallet instruksjoner som blir ferdig eksekvert per tidsenhet, **men**: hver instruksjon tar fortsatt like lang tid
- Forutsetningen for at pipelining er at hver enhet som utfører en del av en instruksjon arbeider uavhengig av de andre enhetene i pipelinen
- Gitt en prosessor hvor hver instruksjon består av fire steg:

  FETCH (Hent instruksjon)
  DECODE (Dekod instruksjonen)
  EXECUTE (Utfør instruksjonen)
  WRITE BACK (Skriv resultatet til minne)

# Repetisjon - Pipelining

- Antar at det finnes 4 uavhengige hardware-enheter som kan utføre hver av disse stegene uten å benytte de andre delene:
  - For eksempel skal FETCH ikke trenge å **bruke** EXECUTE-enheten for å hente en instruksjon eller variabel (men EXECUTE er avhengig av **input** fra DECODE-steget, og DECODE-steget er avhengig av input fra FETCH-steget osv.)
- Alle enhetene kunne jobbe i parallell, med hver sine steg fra **ulike** instruksjoner, uten å gå i "beina på hverandre".

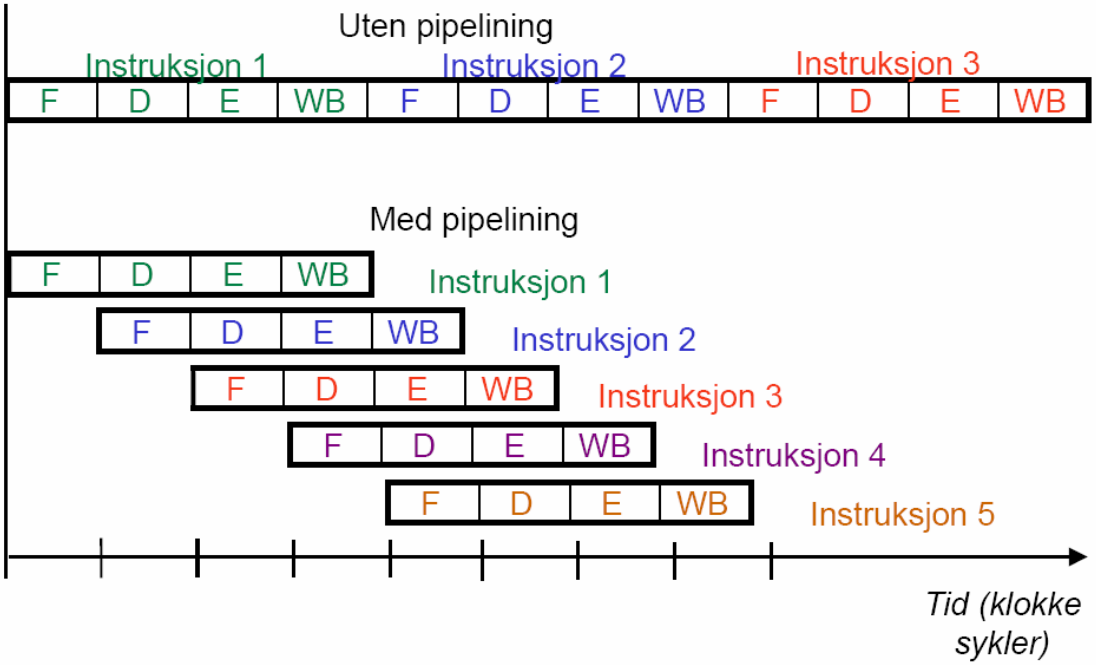
- Uten pipelining kan FETCH-steget til instruksjon 2 først kan starte etter WRITEBACK steget til instruksjon 1
- Med pipelining vil FETCH-steget til instruksjon 2 starte rett etter at FETCH-steget til instruksjon 1 er ferdig




2008

# Repetisjon - Pipelining

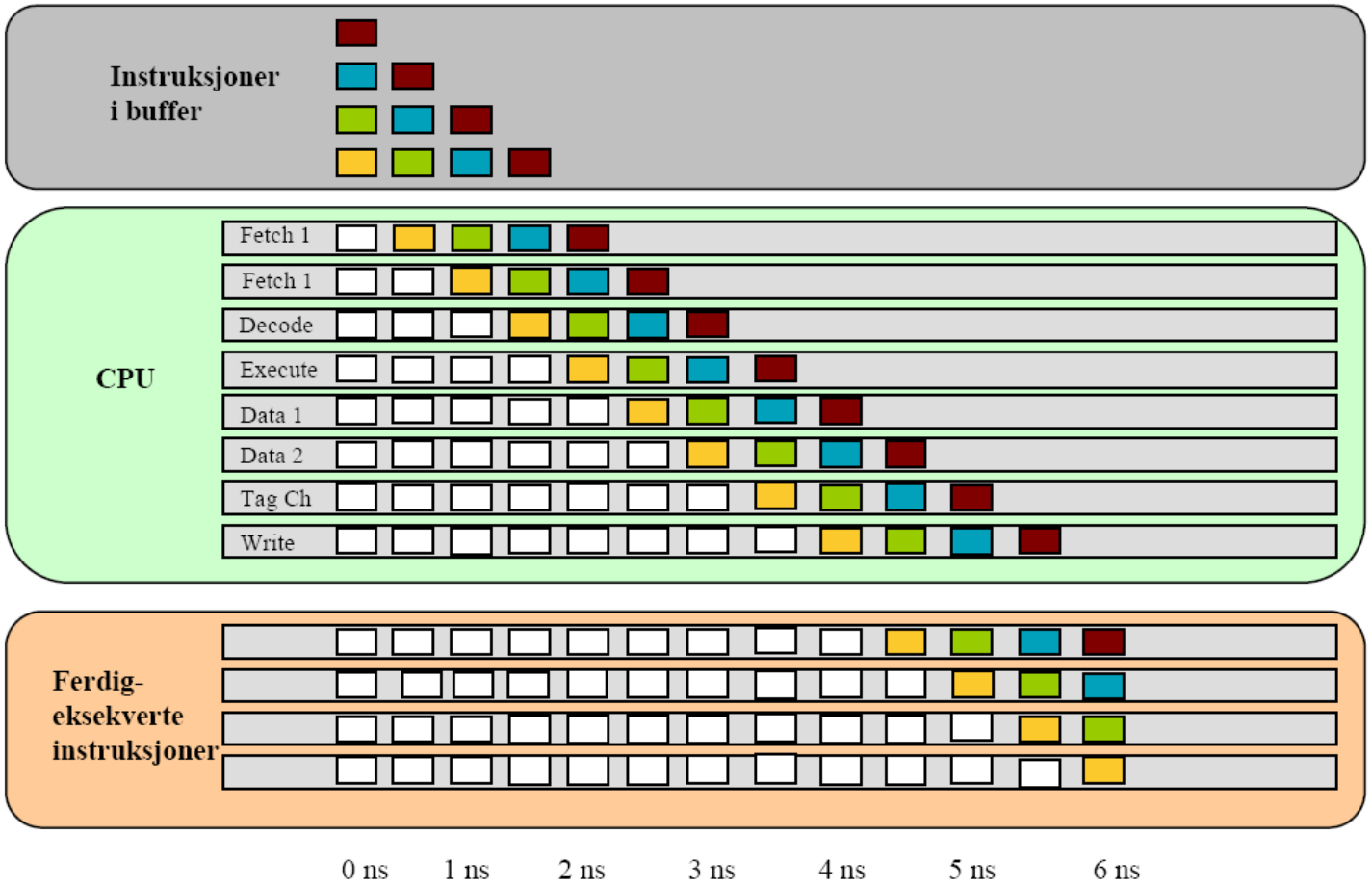
- Uten pipelining er det kun ett steg fra én instruksjon som prosesseres av gangen i prosessoren.
- Med pipelining er det opptil 4 steg fra forskjellige instruksjoner som prosesseres i parallell.
  I klokkesykel 4 utføres WRITE-BACK fra instruksjon 2, EXECUTE fra instruksjon 3, DECODE fra instruksjon 4 og FETCH fra instruksjon nummer 5




- **Uten** pipelining avsluttes en instruksjon hver fjerde klokkesykel.
- **Med** pipelining er instruksjon 1 ferdig etter 3. klokkesykel, instruksjon 2 ferdig etter 4. klokkesykelosv. Mao.: En instruksjon er ferdig hver klokkesykel.
- Merk: fire-stegs pipelinen her er brukt som et eksempel. Dagens prosessorer har pipeliner med adskillig fler steg

2008

# 8-trinns pipeline (MIPS 5000)




2008

# Begrensninger ved pipelining

- Forutsetinger:
  - At instruksjonene kan deles opp i sekvensielle steg som løses etter hverandre
  - At de ulike stegene utføres på egne hardware-enheter som er uavhengige av hverandre

- Ideelt sett gir en *k*-trinns pipeline en faktor *k* i hastighetsøkning sammenlignet med en arkitektur uten pipelining
  - I praksis er dette ikke mulig grunnet ***latency*** og ***hasarder***

# Generell ytelse

- *CPUtid*, tiden en prosessor bruker på å utføre *n* instruksjoner, er gitt ved:

$$CPUtid = CPI \cdot I \cdot t, \text{ der}$$

*CPI* er gjennsomsnittlig antall klokkesykler per instruksjon,

*I* er antall instruksjoner og

*t* er klokkeperioden (lengden på en klokkesykel),
$t = 1/f$ hvor *f* er klokkefrekvensen

- Ytelsen, *P*, er definert som den inverse til *CPUtid*:

$$P = \frac{1}{CPUtid} = \frac{1}{CPI \cdot I \cdot t} = \frac{f}{CPI \cdot I}$$

# Latency

- Hvis en prosessor utfører *n* instruksjoner fullstendig sekvensielt på en *k*-trinns pipeline (uten å starte en ny instruksjon hver klokkesykel), trengs

$$T_S = nkt$$

- Benyttes derimot "ekte" pipelining, behøves

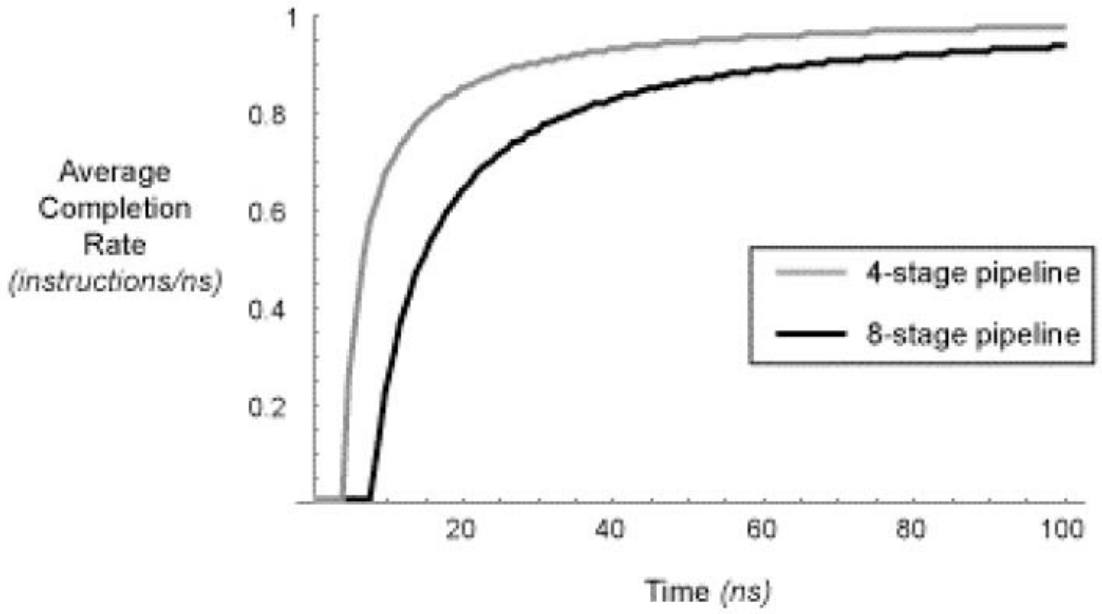
$$T_p = kt + (n-1)t$$

der *kt* er tiden det tar for pipelinen å fylles opp med den første instruksjonen, og (*n*-1)*t* tiden for å eksekvere de *n*-1 resterende

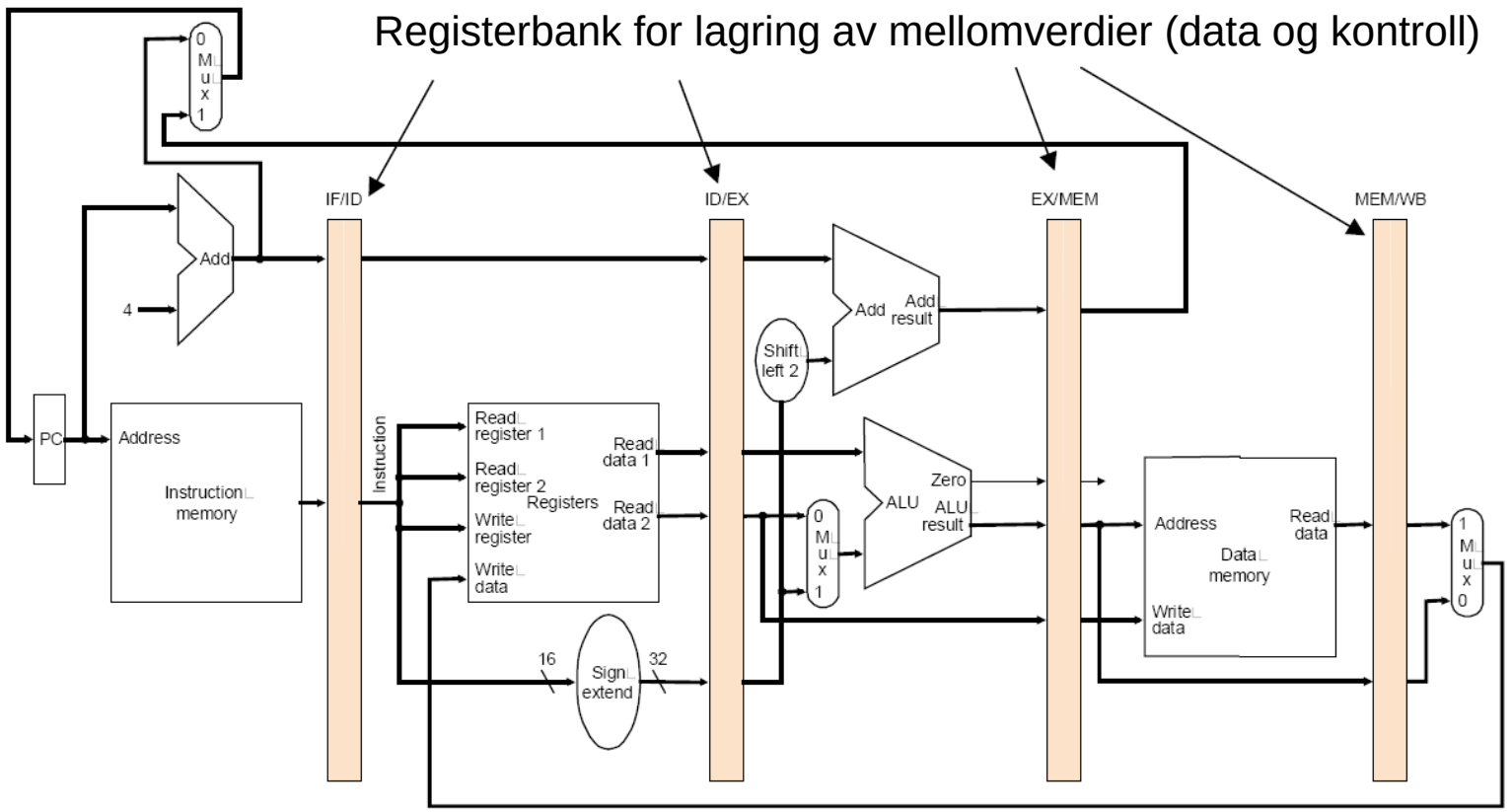
- Forbedringen blir da en faktor S gitt av

$$S = \frac{T_S}{T_p} = \frac{nkt}{kt + (n-1)t} = \frac{nk}{k + n - 1}$$

# Latency vs. antall eksekverte instruksjoner



# Eksempel på pipelinet arkitektur (5-trinns): MIPS4000



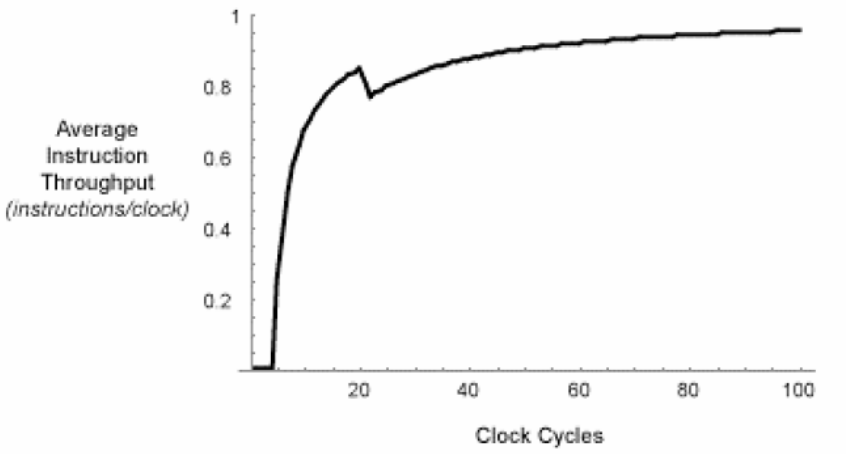
# Utfordringer med pipelining (1)

- Pipelining krever at alle stegene hver tar maksimalt en klokkesykel
  - Spesielt utfordrende for instruksjoner med minneaksess
- Løsning 1: Velger klokkesykel lik det som kreves for det "lengste" steget
  - Lange instruksjoner vil bruke mye lenger tid enn nødvendig
- Løsning 2: Klokkesykel med variable lengde
  - For komplisert!
- Løsning 3: Raskere minne
  - Cache bedrer ytelsen betraktelig
- Løsning 4: Bytte om rekkefølgen på instruksjoner
  - Gjøres automatisk av kompilatoren

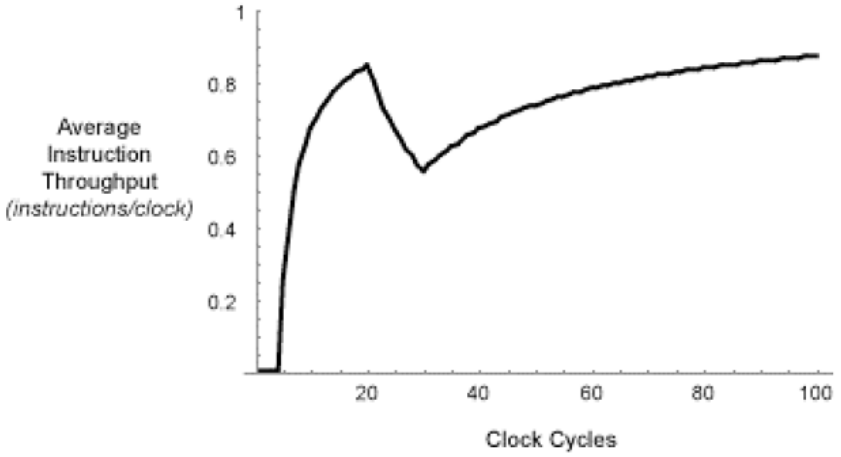
# Utfordringer med pipelining (2)

- Det forekommer tilfeller hvor man ikke kan starte en ny instruksjon hver klokkesykel og pipelinen må stoppe opp i en eller flere klokkesykler
  - Kalles også for stalling
- Hovedårsaken til stalling er ***hasarder*** av ulike typer:
  - Ressurshasarder
  - Datahasarder
  - Kontrollhasarder

# Konsekvenser på hasarder på ytelse




2-syklus stall




10-syklus stall

4-trinns pipeline

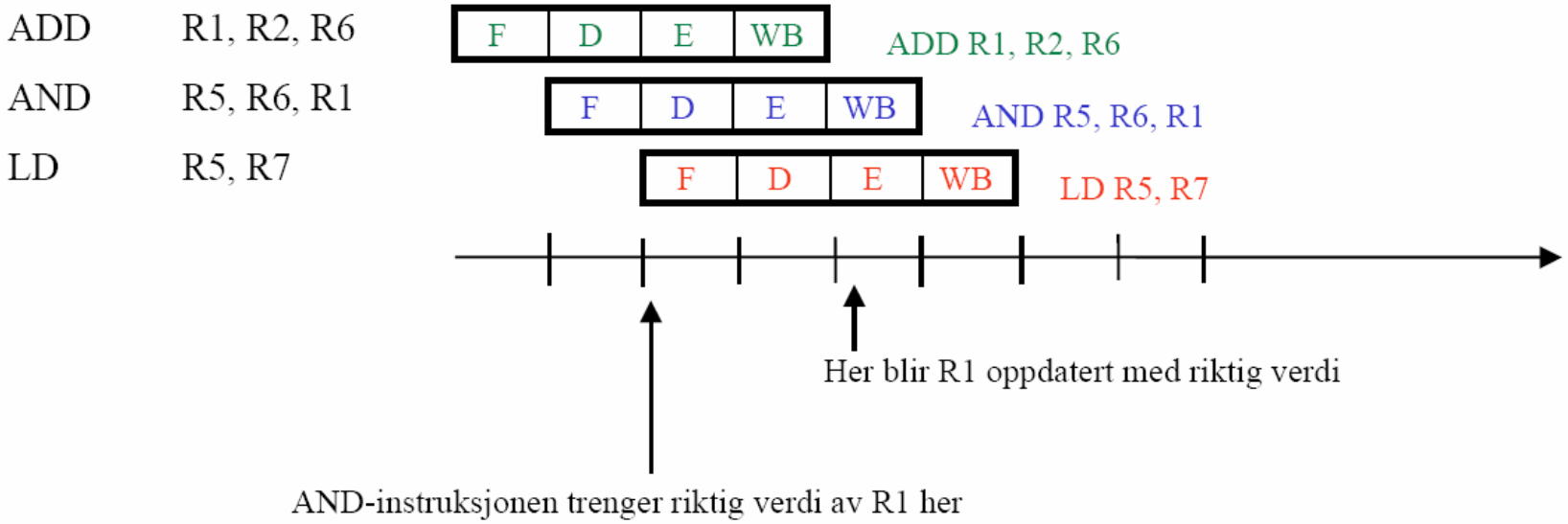
2008

# Ressurshasarder

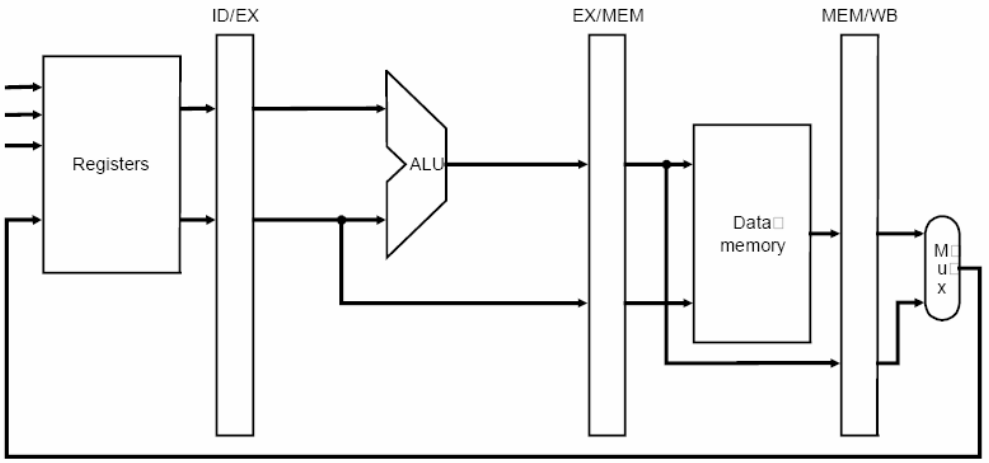
- En datamaskin består av mange delte ressurser:
  - RAM, ALU, Cache, registre
- Ressurshasard: Mer enn én instruksjon ønsker adgang til samme delte ressurs samtidig og pipelinen må stoppe inntil konflikten er opphørt.
- Vanligste løsning på problemet er å bruke ekstra hardware:
  - Egen ALU til adresseberegninger
  - Separat cache for instruksjoner og data
  - Flere instruksjonsregistre
  - Mange generelle registre
- Bytte om rekkefølgen på instruksjoner

# Datahasarder

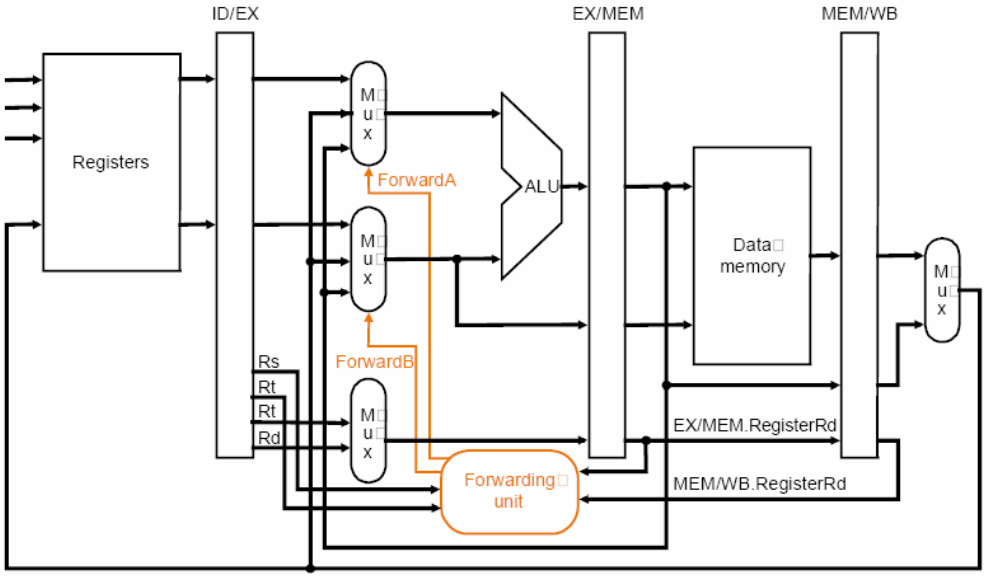
- Problem: Benytter en operand/resultat som beregnes av foregående instruksjon



- Løsning 1: Forwarding:
  - Legge til ekstra hardware slik at resultatet blir tilgjengelig bakover i pipelinen, dvs. for instruksjoner som kommer etter
- Vil ikke alltid fungere




a. No forwarding



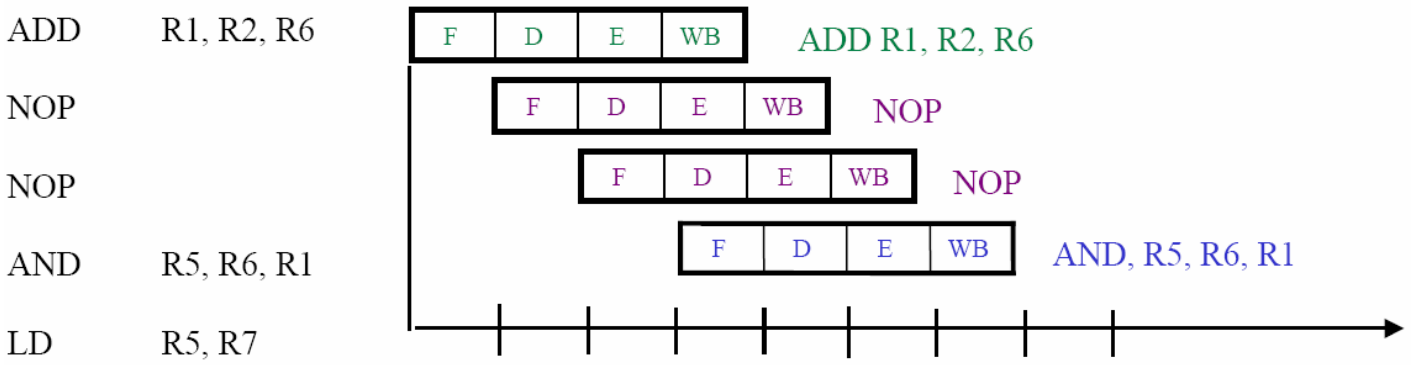

2008

- Løsning 2: Bytte om rekkefølgen på instruksjoner

```
ADD   R1, R2, R6          ADD   R1, R2, R6
AND   R5, R6, R1    →     LD    R5, R7
LD    R5, R7              SUB   R8, R3, R4
SUB   R8, R3, R4          AND   R5, R6, R1
```

- Gjøres av kompilatoren, men
  - Er komplisert og én flytting kan skape nye avhengigheter
  - Vil ikke alltid fungere

- Løsning 3: Sette inn ’tomme’ instruksjoner: NOP
  - Alternativt kan man stoppe prosessoren hvis problemet er ’Load’

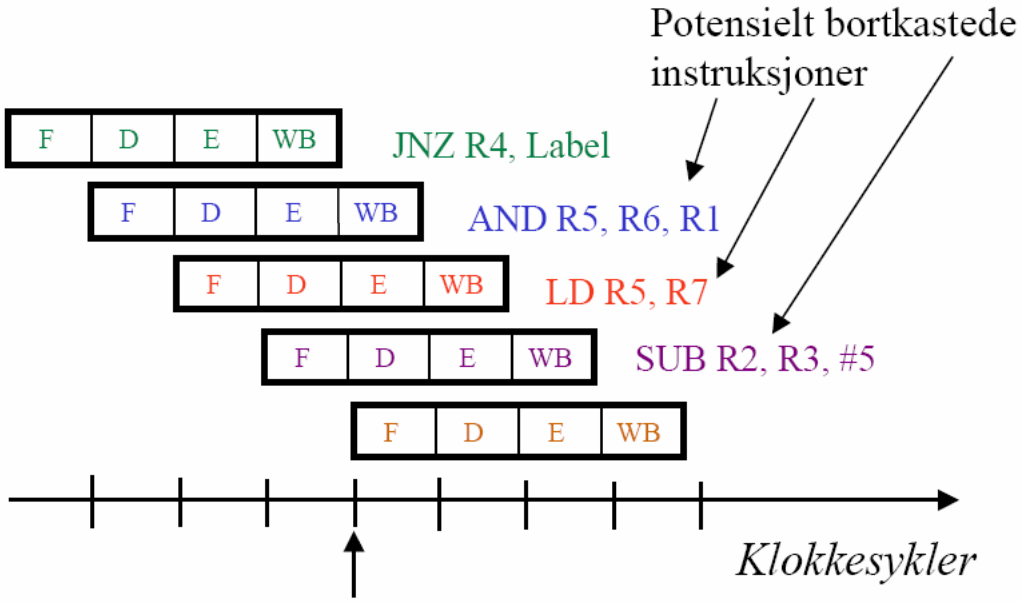



- Vil alltid fungere, men sløser bort prosessortid
- Benyttes i kombinasjon med forwarding, og ombytte av instruksjoner

# Kontrollhasarder

- Problem: Må kjenne resultatet fra foregående instruksjon ved betingede hopp

```
        JNZ     R4, Label
        AND     R5, R6, R1
        LD      R5, R7
        SUB     R2, R3, #5
        …       …
Label   LD      R4, R2
```




Vet først her om programmet skal hoppe eller ikke!

2008

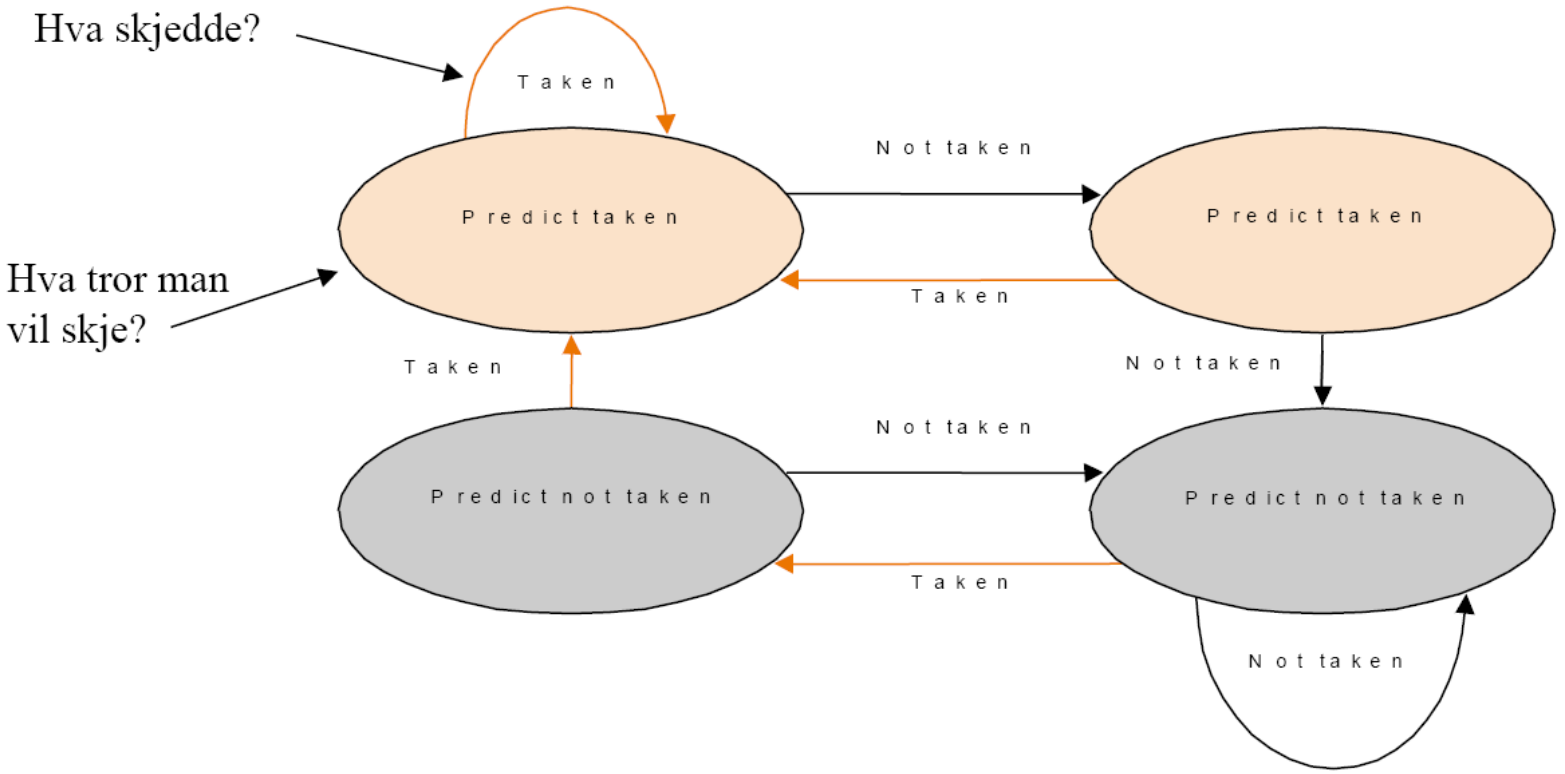
- Betingede hopp kan redusere eksekveringshastigheten betydelig hvis man regner med at det aldri skal hoppes (dvs. ikke gjør noe)
  - Gjennomsnittlig antall klokkesykler per instruksjon er gitt av

$$CPI_{av} = 1 + b \cdot p_b \cdot p_t$$

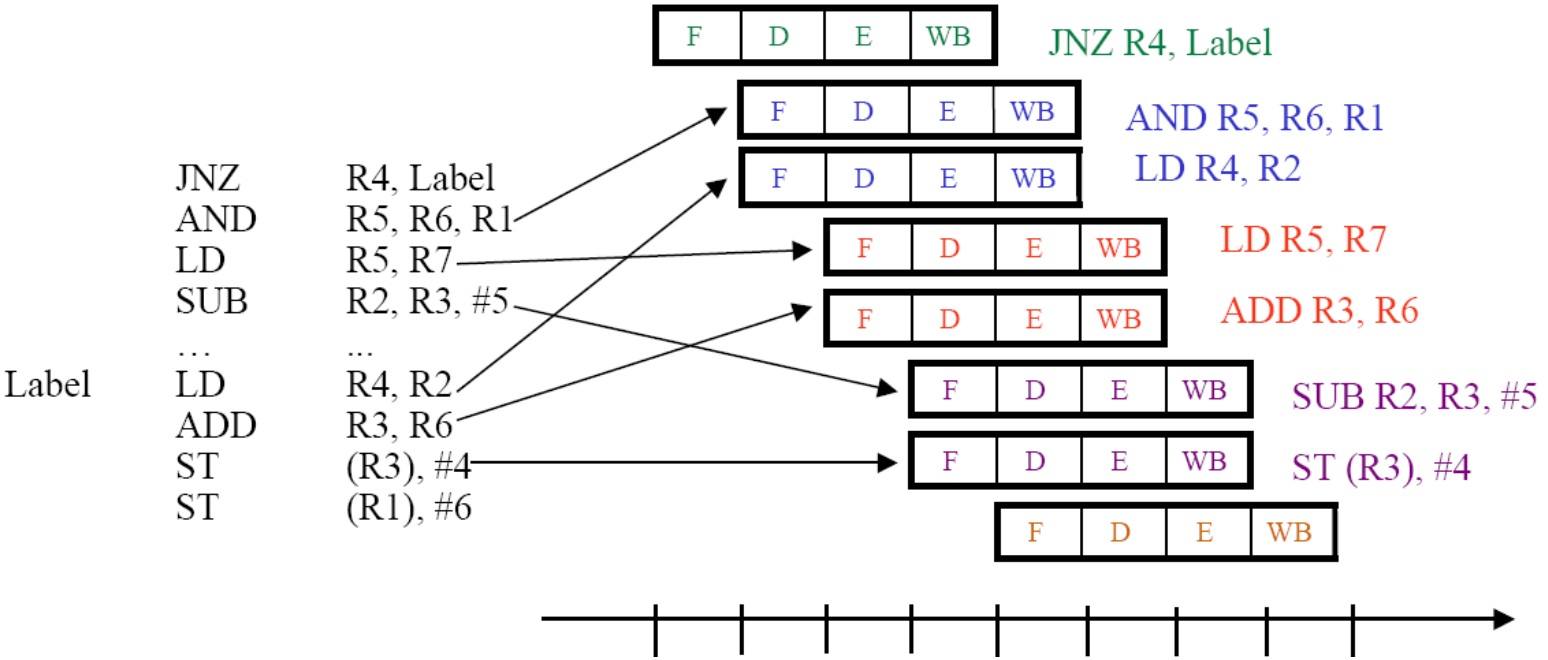
  - $p_b$ er sannsynligheten for at en instruksjon er et betinget hopp
  - $p_t$ er sannsynligheten for at man hopper ved betingede hopp
  - $b$ er straffen for å starte unødvendig eksekvering av instruksjoner (branch penalty)
- Absolutte hopp er ikke problem
  - Men må detekteres ved compile-time og behandles deretter

- Løsning 1: Hopp-prediksjon
  - Prøver å forutsi om en betinget-hopp instruksjon faktisk vil hoppe
  - Benytter denne kunnskapen til å starte eksekvering av korrekt instruksjonssekvens slik at flushing unngås
- Statisk prediksjon: Regner med at en bestemt type betingede hopp alltid/aldri fører til hopp, mao. statisk oppførsel
  - Enkel algoritme, men tar ikke hensyn til at programmer er dynamiske
  - Vil kunne konsekvent forutsi feil ved f.eks while/for-løkker
- Dynamisk prediskjon: Baserer seg på oppførelsen ved tidligere eksekvering av kodesekvensen
  - Krever noe mer hardware (må bla lagre tilstandsinfo)
  - Bedre treff-prosent enn statisk prediksjon

- Tilstandsdiagram for dynamisk hopp-prediksjon
  - For å få bedre prediksjon benyttes 2-bits tilstand, dvs. oppførselen må være lik for de 2 foregående gjennomløpene

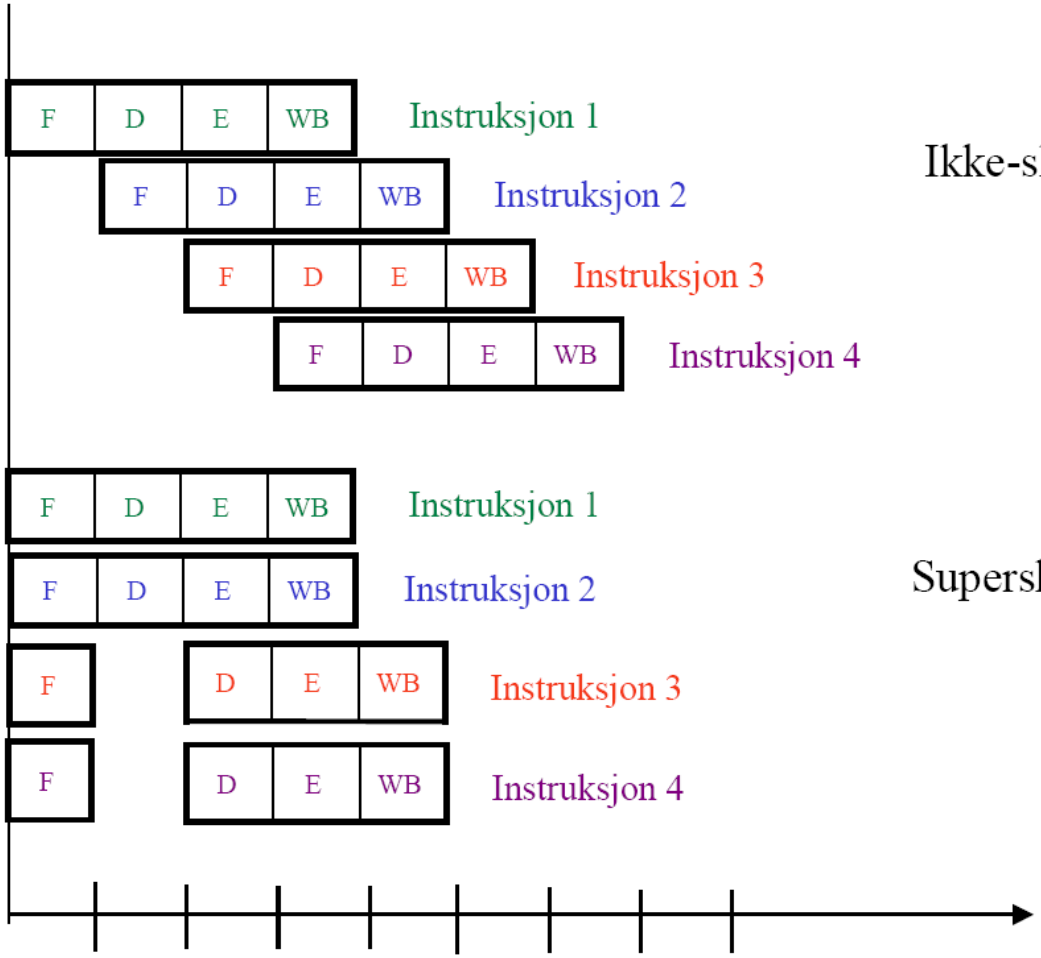


- Løsning 2: Parallelleksekvering av hoppinstruksjoner
  - I stedet for å forutsi, starter prosessoren eksekvering av begge kodesekvenser i parallell til riktig hopp kan bestemmes

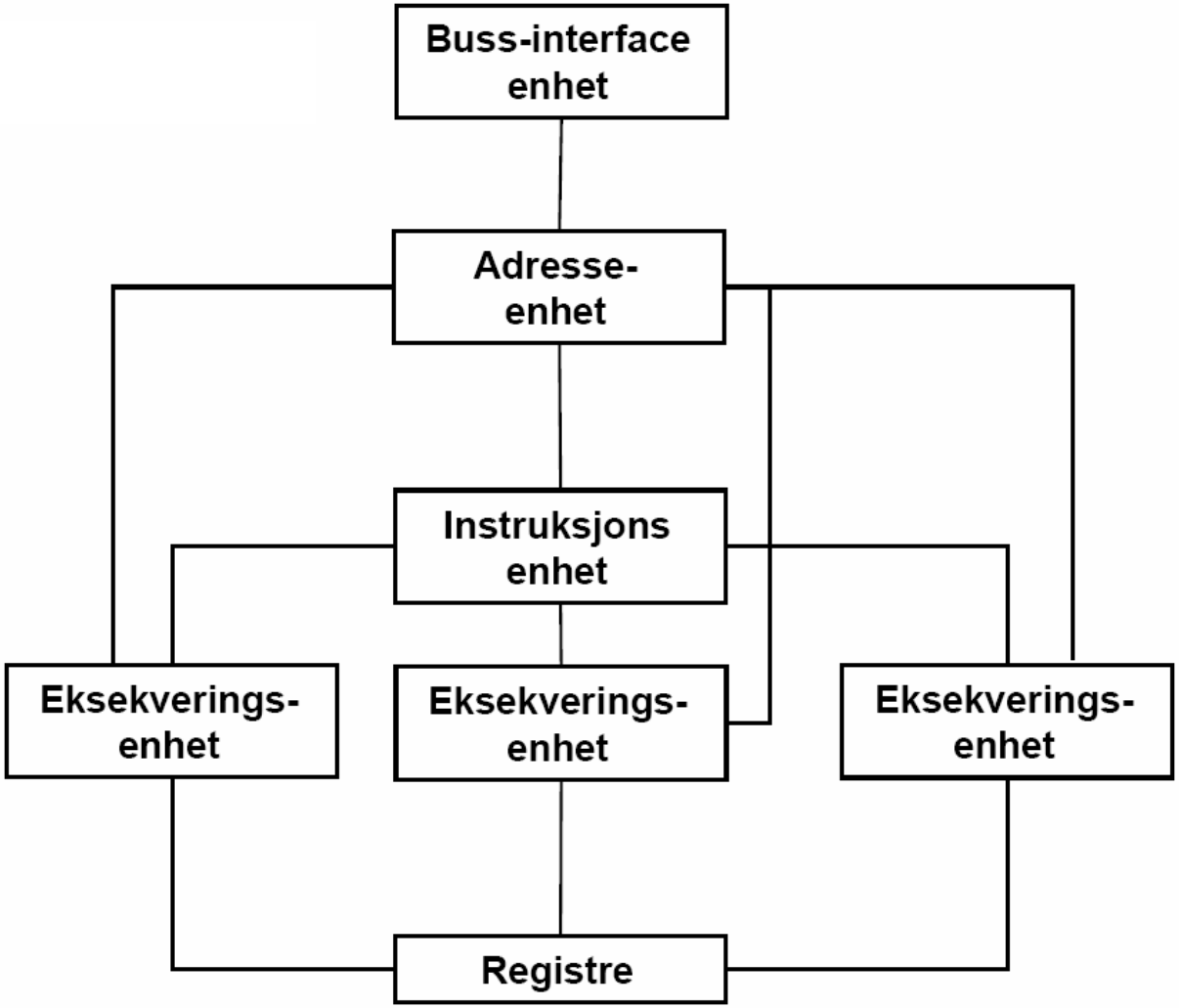


# Superskalare prosessorer

- En vanlig pipelinet prosessor er begrenset til å eksekvere ferdig en instruksjon per klokkesykel.
- Hastigheten kan økes ved å eksekvere flere instruksjoner i parallell:
  - Flere instruksjoner kan kjøres samtidig,
  - Enkelte steg kjøres samtidig, men har f.eks sekvensielle FETCH-steg.
- Superskalare prosessorer har flere hardware-enheter som gjør heltalls og flyttallsaritmetikk, load/store osv. i parallell
  - Operasjoner som tar mye lenger tid enn andre sperrer ikke for andre instruksjoner.
- Superskalare prosessorer er mer kompliserte og krever ekstra kontroll-logikk, flere registre, etc.
- F.eks. Pentium og G4 er superskalare prosessorer

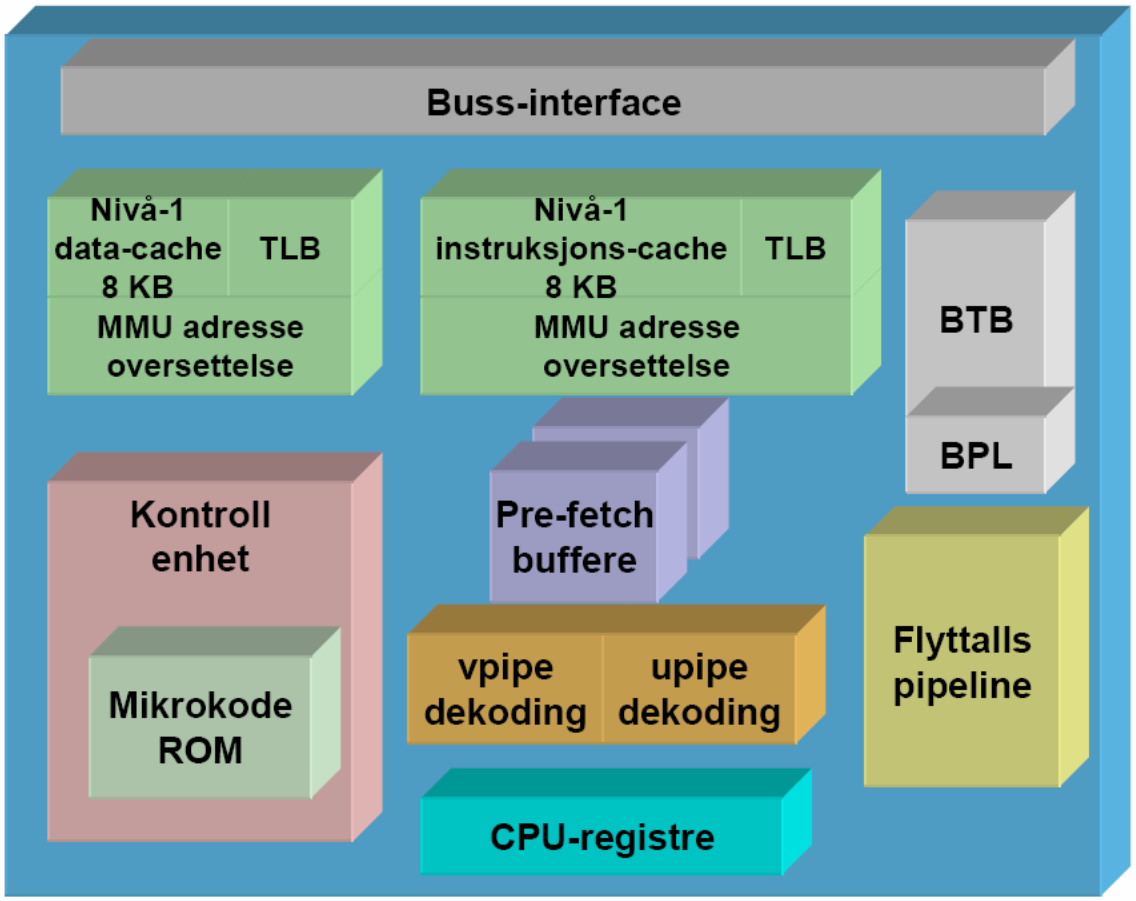

F D E WB
Instruksjon 1
F D E WB
Instruksjon 2
F D E WB
Instruksjon 3
F D E WB
Instruksjon 4
Ikke-skalar pipelining
F D E WB
Instruksjon 1
F D E WB
Instruksjon 2
F D E WB
Instruksjon 3
F D E WB
Instruksjon 4
Superskalar pipelining

2008



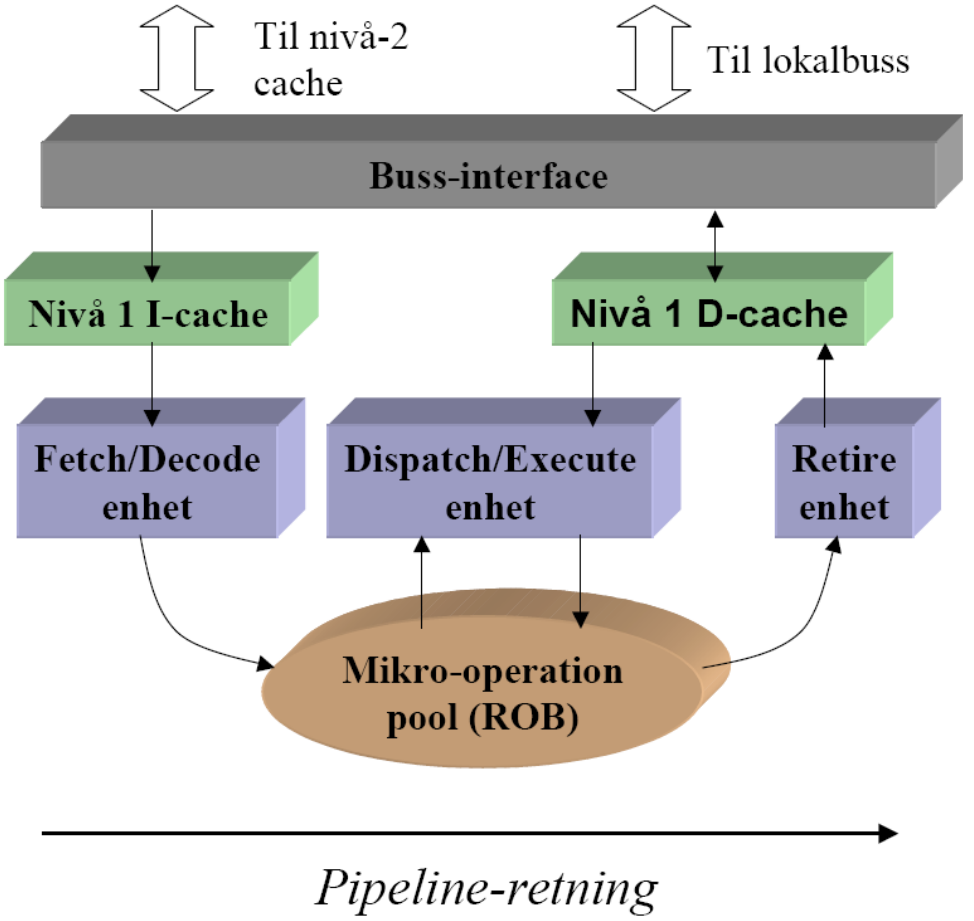

Parallelliteten gjør superskalare prosessorer mye vanskeligere å designe

# Blokkdiagram for Pentium III/4




2008

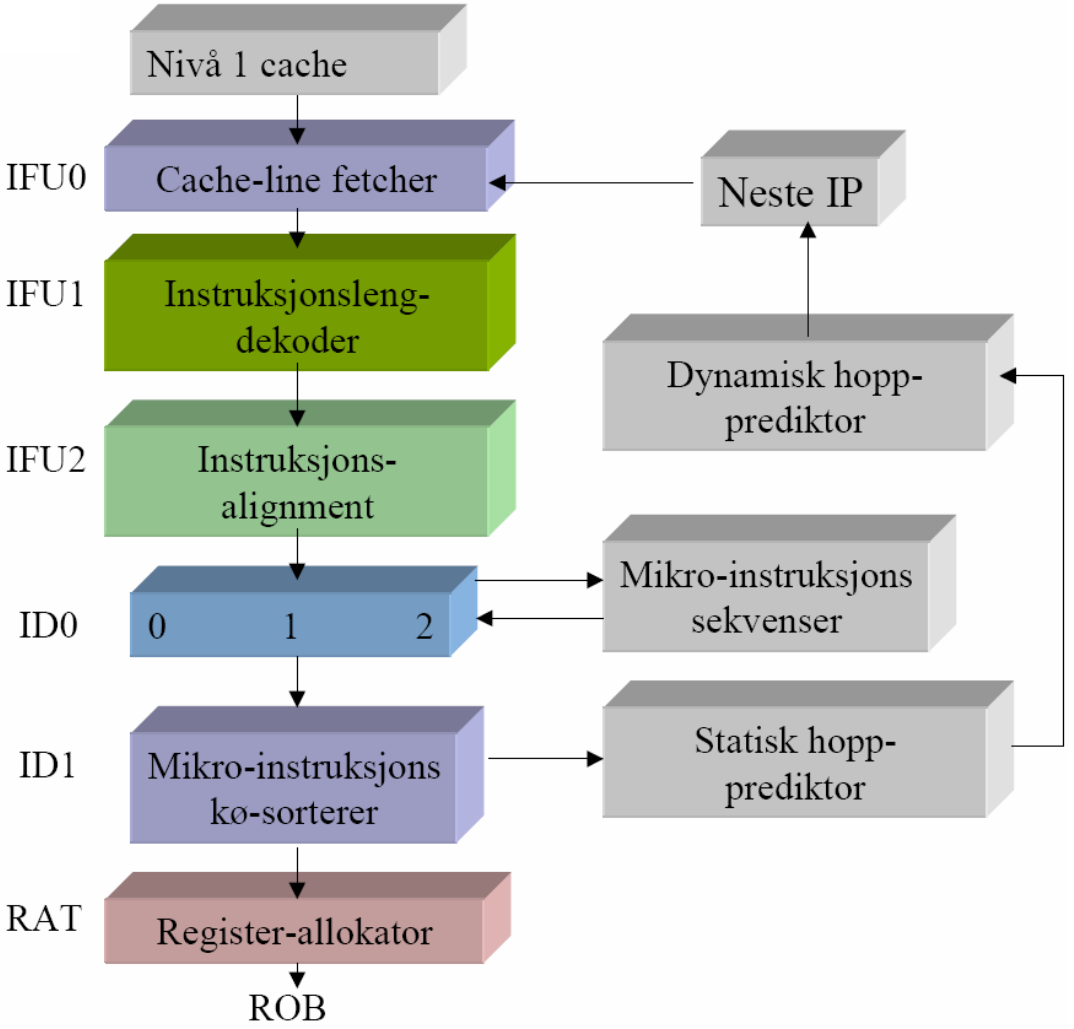
- Fetch/Decode, Dispatch/Execute og Retire utgjør tilsammen en 3 trinns høynivå pipeline.
- ROB inneholder informasjon om delvis eksekverte instruksjoner som av en eller annen grunn venter på å blir ferdige.




2008

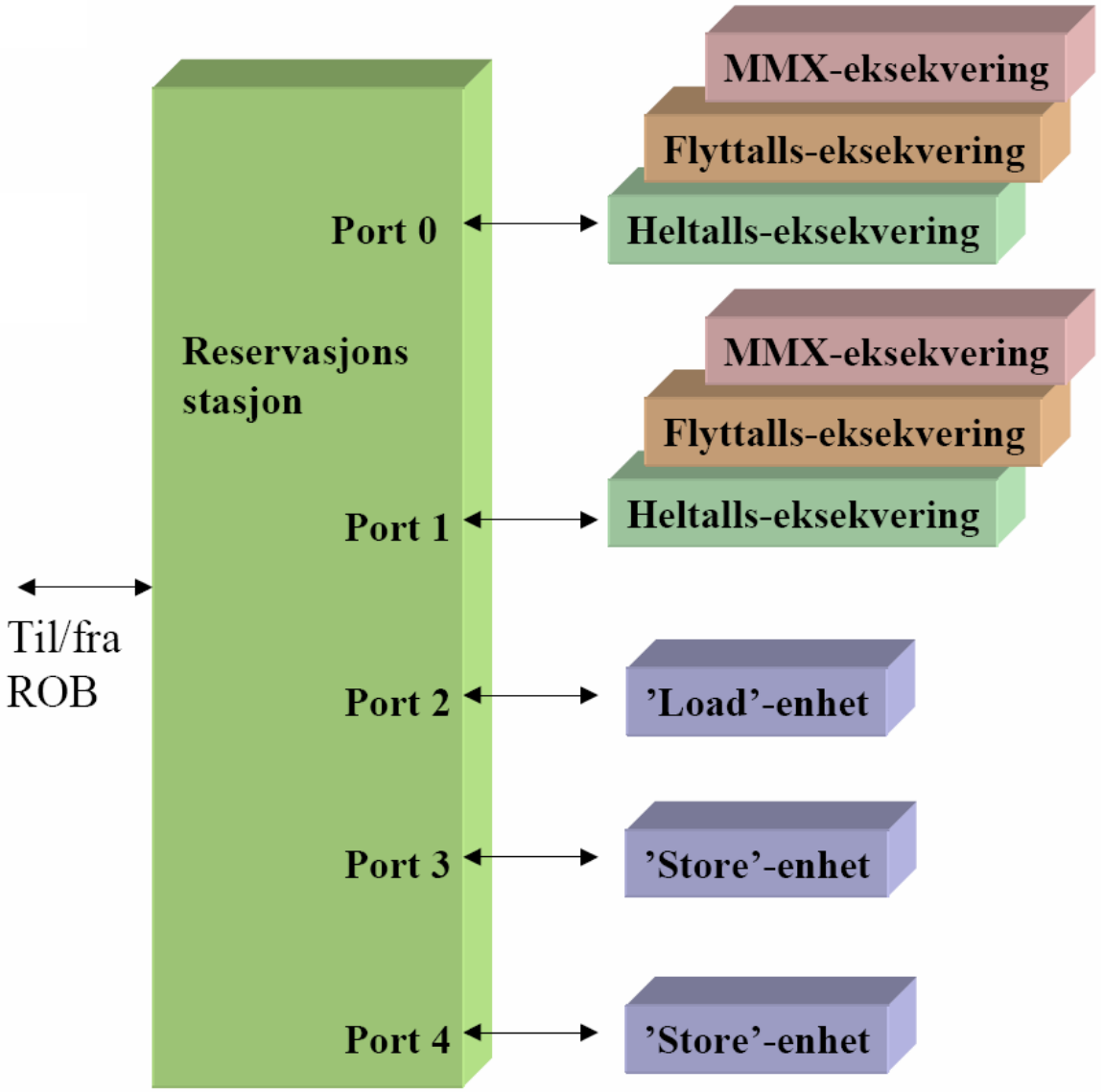
- Fetch/Decode henter instruksjoner og splitter dem opp i mikro-instruksjoner som lagres i ROB (klare til å legges i Execute).
- Dispatch/Execute eksekverer mikro-instruksjoner som er lagret i ROB.
- Retire-enheten fullfører eksekveringen av hver mikro-instruksjon og oppdaterer registre i henhold til hvilken instruksjon det er snakk om.
- Instruksjoner plasseres i ROB i riktig rekkefølge, men kan stokkes om før eksekvering, og sendes til Retire-enheten i riktig rekkefølge.
- Fetch/Decode-enheten er internt organsiert som en 7-trinns skalar pipeline
- Dispatch/Execute og Retire-enheten inneholder en 5-trinns pipeline til sammen
- Dispatch/Execute-enheten består av to enheter (kalt U og V, eller MMX og Heltall) som kan eksekvere instruksjoner i parallell.
- En tredje pipelinet eksekveringsenhet brukes til flyttall
- Totalt gir dette Pentium III en 12-trinns pipeline (siste P4 (2006) hadde 31-trinns pipeline)

# Fetch/Decode-enheten



- IFU0: Instruksjoner hentes fra instruksjons-cachen
- IFU1: Analyserer byte-strømmen for å finne starten på neste instruksjon.
- IFU2: Sorterer instruksjonen (av variabel lengde) slik at den lett kan dekodes.
- ID0: Start på dekoding:
  - Oversettelse til mikro-instruksjoner (3 i parallel)
  - Hver mikroinstruksjon inneholder opcode, to source-registre og ett destinasjonsregister
- ID1: Mikro-instruksjonene legges i en kø
  - Betingede hopp detekteres.
  - Dynamisk prediksjon bruker 4 bit
- RAT: Tilording til registre (nødvendig fordi gammel x86 kode kan kreve spesielle registre)

# Dispatch/Execute -enheten




2008

- Reservasjonsstasjonen kan innholde inntil 20 mikroinstruksjoner som venter på å bli utført.
- Inntil 5 mikroinstruksjoner kan eksekveres samtidig
- Mikroinstruksjonene kan eksekveres i en annen rekkefølge enn den som er gitt av hovedinstruksjonen.
- En kompleks algoritme holder styr på hvilke mikroinstruksjoner som skal eksekveres til hvilken tid og på hvilken port.
- Hasarder og registeravhengigheter løses i reservasjonsstasjonen.
- Når en mikroinstruksjon er ferdig eksekvert, sendes den tilbake til reservasjonsstasjonen og så tilbake til ROB for tilslutt å sendes til Retireenheten

- Retire-enheten sender reultatet av mikroinstruksjonen til rikig register, eller til andre eksekveringsenheter i Dispatch/Execute blokken.

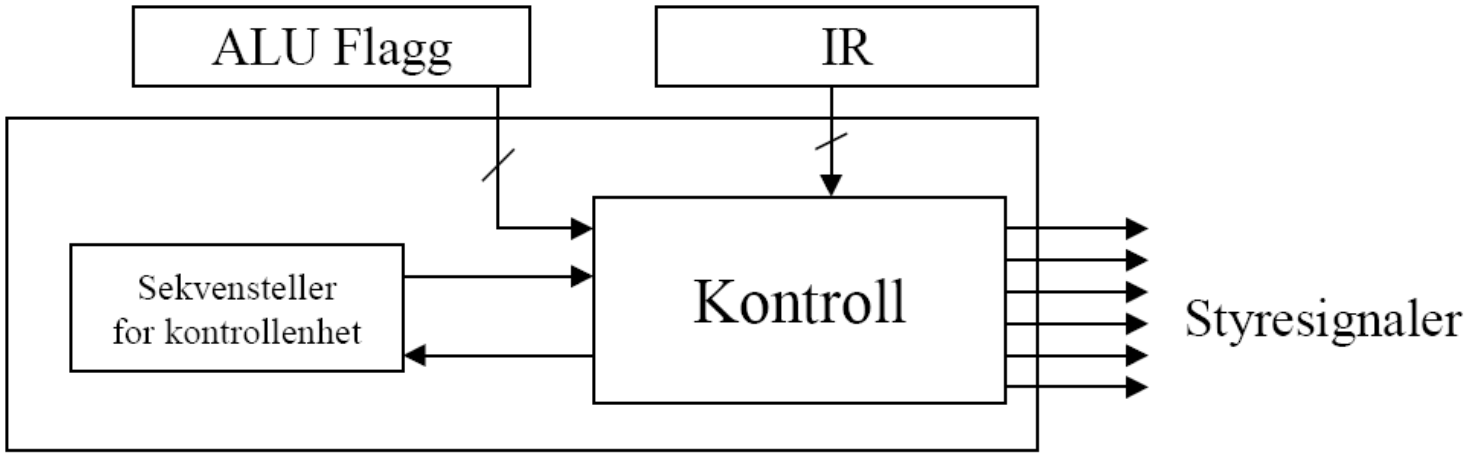
- Retire-enheten holder styr på instruksjoner som er satt i gang ved ’spekulativ’-eksekvering.

- Instruksjoner kommer ferdig utført fra Retire-enheten i samme rekkefølge som de ble sendt inn i Fetch/Decode-enheten

# Design av kontrollenhet

- Under instruksjonseksekvering må riktige kontrollsignaler settes avhengig av instruksjonen

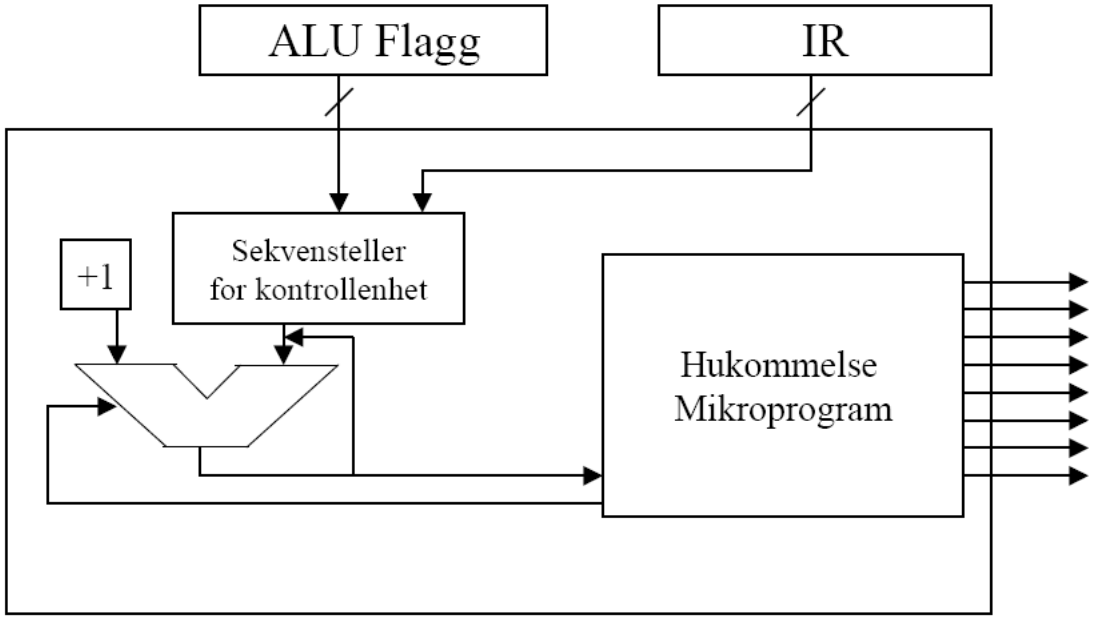
- Variant 1: Hardwired kontroll



- Egenskaper ved hardwired kontroll
  - Siden enheten er kombinatorisk (bortsett fra sekvenstelleren), er den meget rask.
  - Tar liten plass
  - Ikke mulig å endre etter implementasjon
  - Kan bli for stor og komplisert ved sammensatte og komplekse instruksjoner

# Mikroprogrammert kontroll

- Inneholder en liten CPU som tolker et styreprogram som styrer selve CPU'en




- Egenskaper ved mikroprogrammert kontroll
  - Mer fleksibel fordi mikroprogrammet kan byttes underveis
  - Lettere å rette opp feil ved designet
  - Krever flere klokkesykler for å eksekvere en instruksjon
  - Krever stort CPU-areal

# RISC og CISC (1)

RISC – Reduced Instruction Set Computer

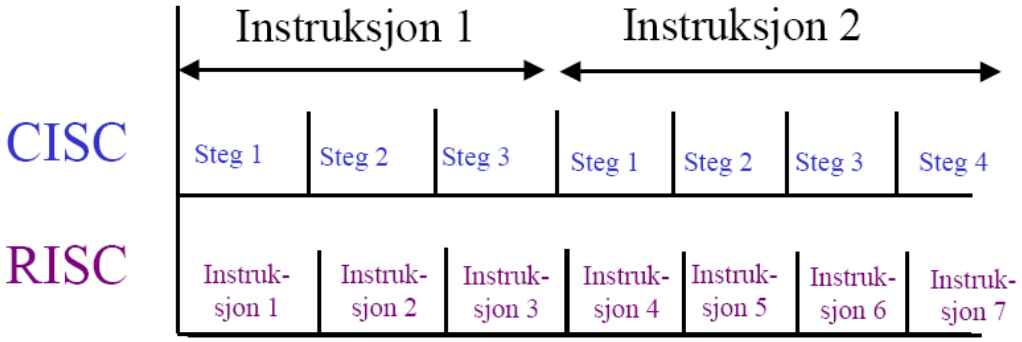
CISC – Complex Instruction Set Computer

- To forskjellige filosofier for design og organisering av en CPU
- CISC-arkitektur har mange maskinspråk-instruksjoner som kompilatorer kan bruke ved oversettelse fra høynivå-språk til maskinspråk
- Assemblerprogramering blir enklere i en CISC-arkitektur fordi det finnes mange spesialiserte instruksjoner.
- En CISC-instruksjon kan bestå av et variabelt antall midre steg eller subinstruksjoner, og hvert steg trenger en klokkesykel på å fullføres.
- I en moderne CISC-arkitektur (f.eks Pentium) varierer antall steg i instruksjonene fra et ti-talls til flere hundre

# RISC og CISC (2)

- RISC-arkitektur har langt færre maskinspråk-instruksjoner tilgjengelige for assembler-program og kompilatorer.
- Hver instruksjon i en RISC-maskin er optimalisert slik at den kun krever én klokkesykel på å eksekvere ferdig.
- Filosofien bak RISC er å optimalisere og tilby de mest brukte instruksjonene som høynivå-programmer anvender (tester har vist at Load/Store og hoppinstruksjoner står for mellom 70-80% av alle instruksjonene i et program)
- Instruksjoner som brukes sjelden implementeres som en sekvens av instruksjoner og blir ikke nødvendigvis implementert mest mulig effektivt.
- Kompilatorer og assemblerprogrammer blir større og mer kompliserte fordi hver høynivå-instruksjon splittes opp i mange flere maskinspråk-instruksjoner sammenlignet med CISC.

# RISC og CISC (3)




- I en RISC-arkitektur tar alle instruksjoner like lang tid.
- I en CISC-arkitektur kan to instruksjoner bruke ulik tid på å bli ferdige.
- Det er vanlig også i RISC arkitektur å dele opp instruksjoner i mindre steg. Ikke alle instruksjoner trenger alle stegene, men de blir allikevel ”tvunget” til å bruke dem (ikke gjøre noe) for at hver instruksjon skal ta like lang tid.

# Hva er best?

- Noen typer optimalsering lar seg lettere designe sammen med RISC-arkitektur, som f.eks pipelining.

- Visse anvendelser som f.eks mobiltelefoner ser ut til å egne seg bedre for RISC enn CISC pga. statisk kode med lite behov for spesial-instruksjoner

| RISC | CISC |
|---|---|
| Enkelt og begrenset instruksjonssett | Komplisert og rikholdig instruksjonssett |
| Register-orienterte instruksjoner med få instruksjoner for minneaksess | Alle instruksjoner er fleksible i adresseringsmekanismer for operander |
| Fast lengde og format på instruksjoner | Variabelt format og lengde på instruksjoner |
| Få adresseringsmodi | Mange adresseringsmodi |
| Stort antall interne registre | Lite antall interne registre |