UNIVERSITETET
I OSLO

Dagens temaer

- 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

Mer om pipeling

- Repetisjon:
  - 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.
  - Én instruksjon eksekveres ferdig per klokkesykel
- Forutsetinger:
  - En instruksjon kan deles opp i sekvensielle steg som løses etterhverandre
  - De ulike stegene utføres på egne hardware-enheter som ikke er avhengige 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

- Tiden en prosessor *Ct* bruker på å utføre *n* instruksjoner er gitt av

  $Ct = CPI * I * t$, der

  CPI er gjennsomsnittlig antall klokkesykler per instruksjon,

  I er antall instruksjoner og

  t er klokkeperioden (lengden på en klokkesykel), dvs 1/f

- Ytelsen *P* er definert som den inverse til *Ct*

$$P = \frac{1}{Ct} = \frac{1}{CPI * I * t} = \frac{f}{CPI * I}$$

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 $(n\text{-}1)t$ tiden for å eksevere de $n\text{-}1$ resterende.

- Forbedringen blir da en faktor S gitt av

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

UNIVERSITETET
I OSLO

Eksempel på pipelinet arkitektur (5-trinns): MIPS 4000

Registerbank for lagring av mellomverdier (data og kontroll)

Utfordringer med pipelining (1)

- Pipelining krever at alle stegene hver tar maks en klokkesykel
  - Spesielt utfordrende for instruksjoner med minneaksess
- Løsning 1: Velger klokkesykel lik den lengste
  - 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)

- Tilfeller hvor man ikke kan starte en ny instruksjon hver klokkesykel
  - Kalles også for stalling
- Hovedårsaken til stalling er *hasarder* av ulike typer:
  - Ressurshasarder
  - Datahasarder
  - Kontrollhasarder

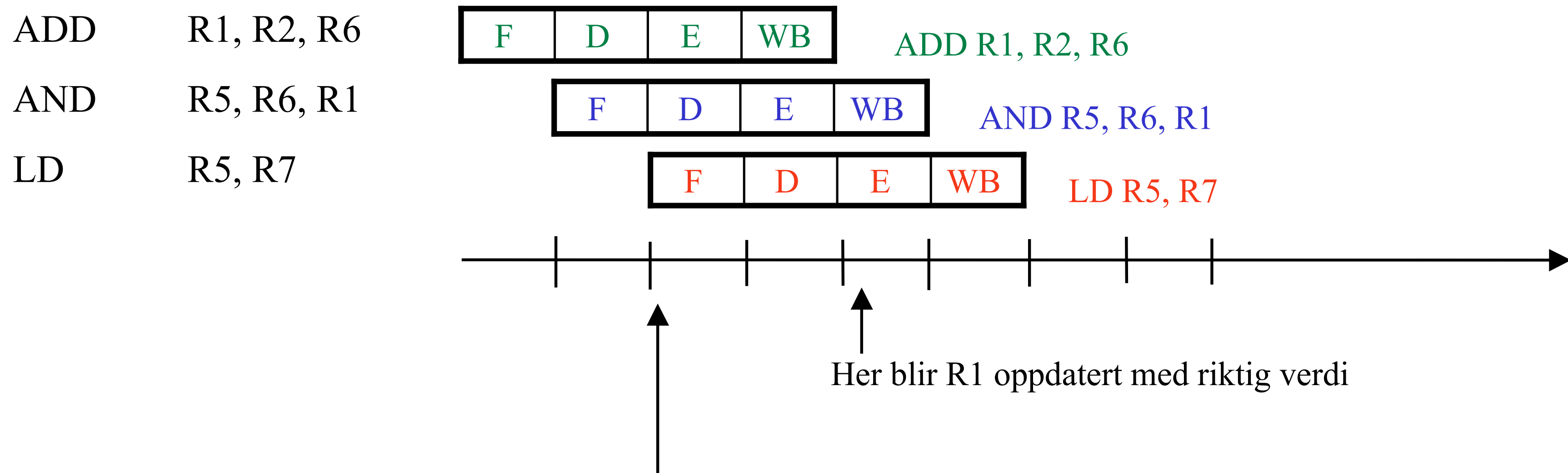
Ressurshasarder

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

Datahasarder

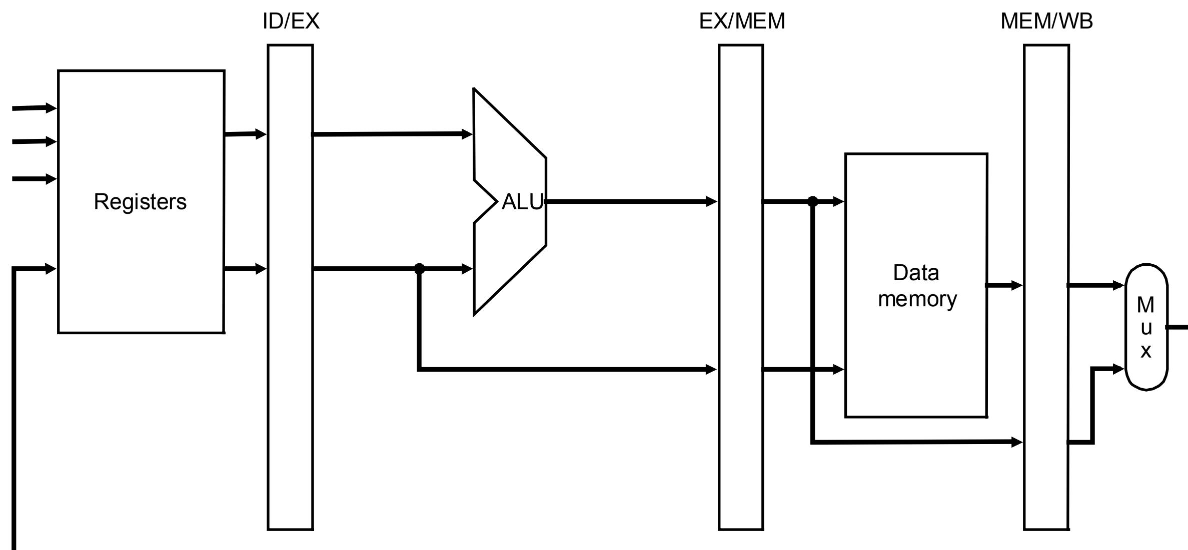
- Problem: Benytter en operand/resultat som beregnes av foregående instruksjon

ADD	R1, R2, R6
AND	R5, R6, R1
LD	R5, R7

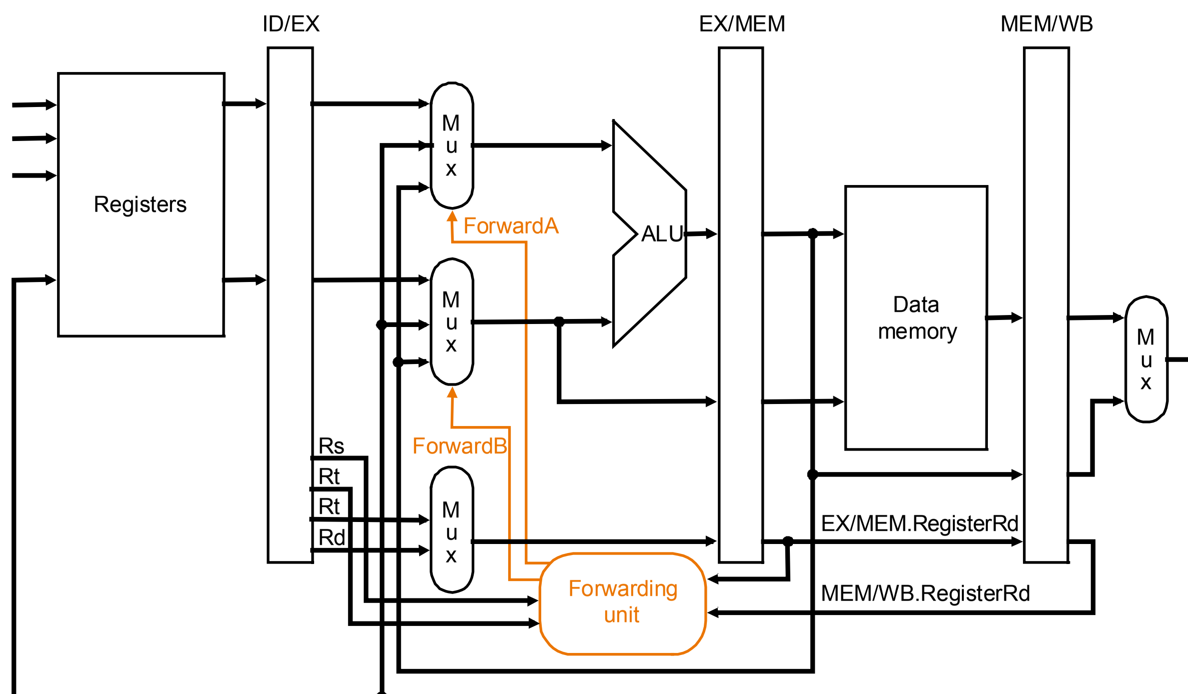


Løsning 1: Forwarding:

- 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



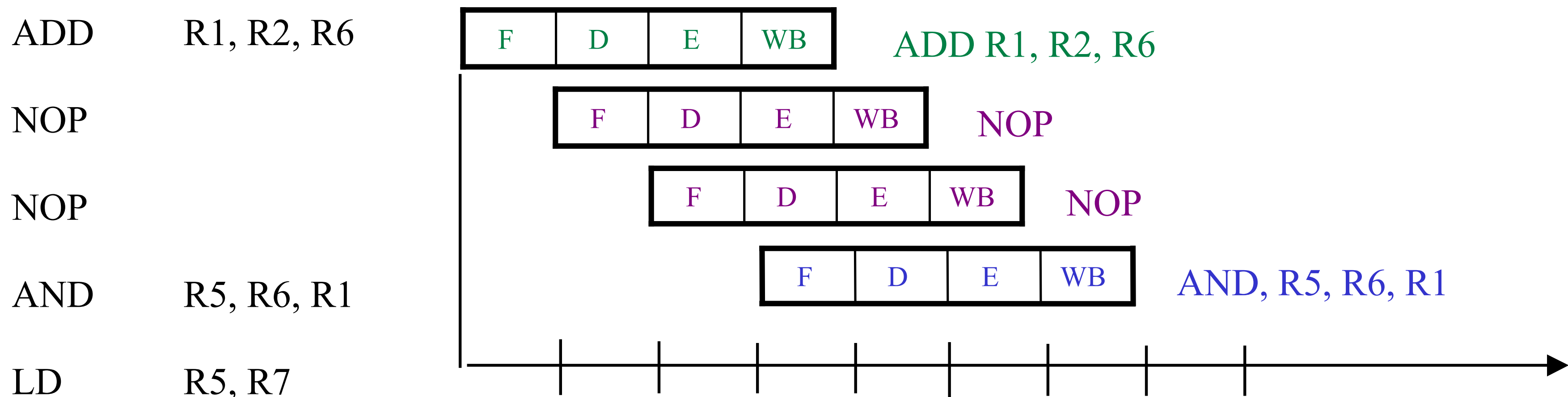
b. With forwarding

- 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'

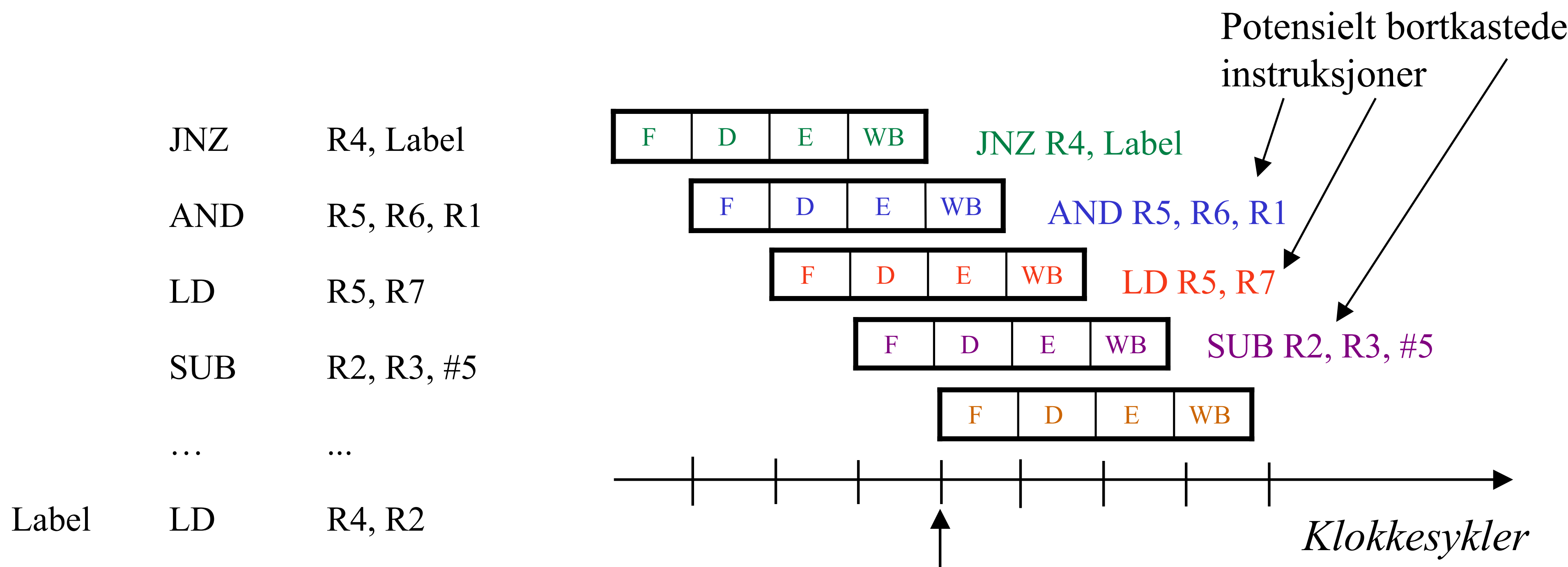


- Vil alltid fungere, men sløser bort prosessortid
- Benyttes i kombinasjon med forwarding, og ombytte av instruksjoner

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



- 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

$$CPI_{av} = 1 + b * p_b * p_t$$

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

- Absolutte hopp er ikke problem
  - Men må detekteres ved compile-time og behandles deretter

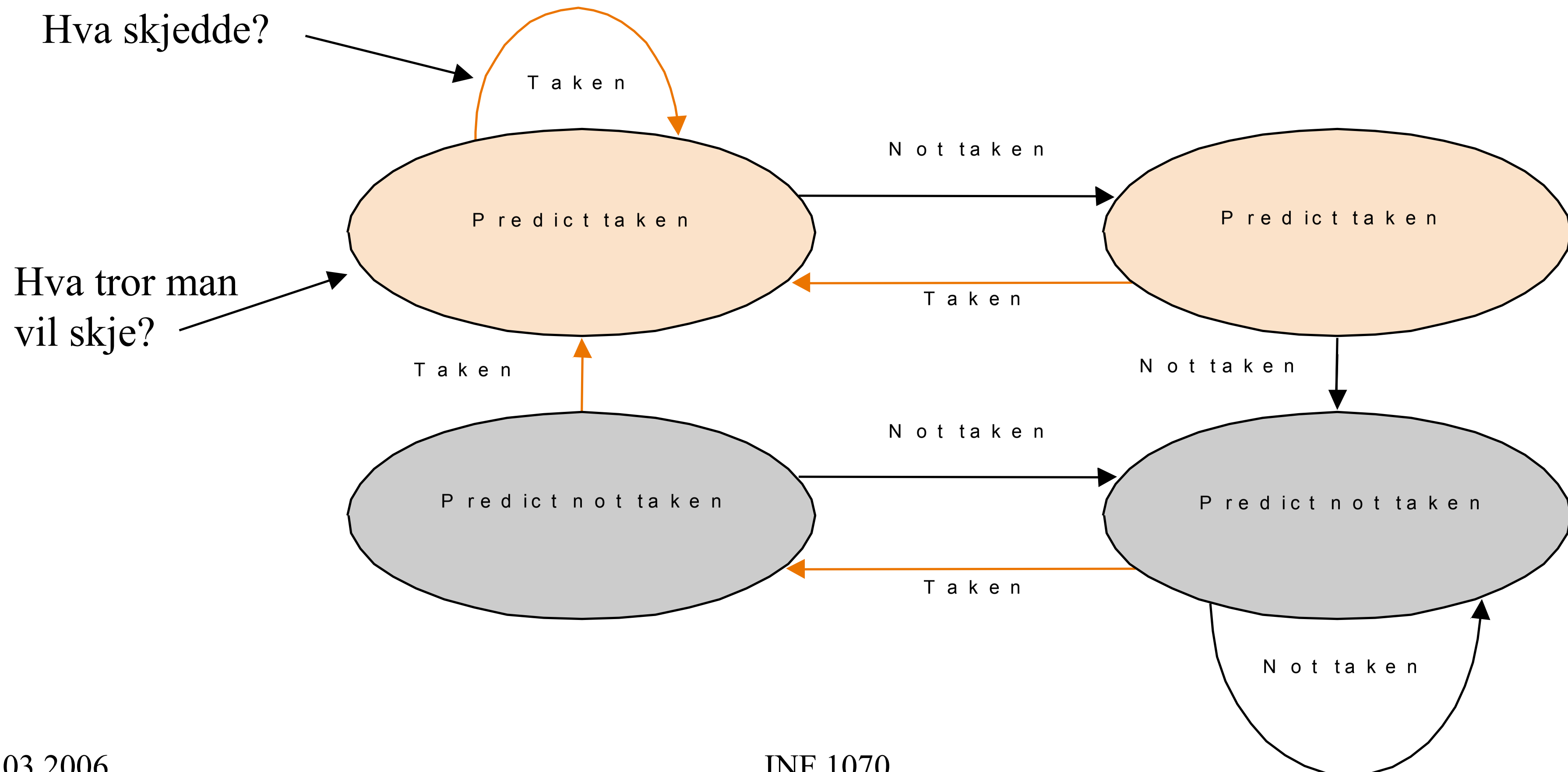
- Løsning 1: Hopp-prediksjon
  - Prøver å forutsi om en betinget-hopp instruksjon faktisk vil hoppe
  - Benytter denne kunnskapen til å starte eksekvering av korrekt instruksjonssekvens slik at flushing unngås

  - Statisk prediksjon: Regner med at en bestemt type betingede hopp alltid/aldri fører til hopp, mao statisk oppførsel
    - Enkel algoritme, men tar hensyn ikke 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

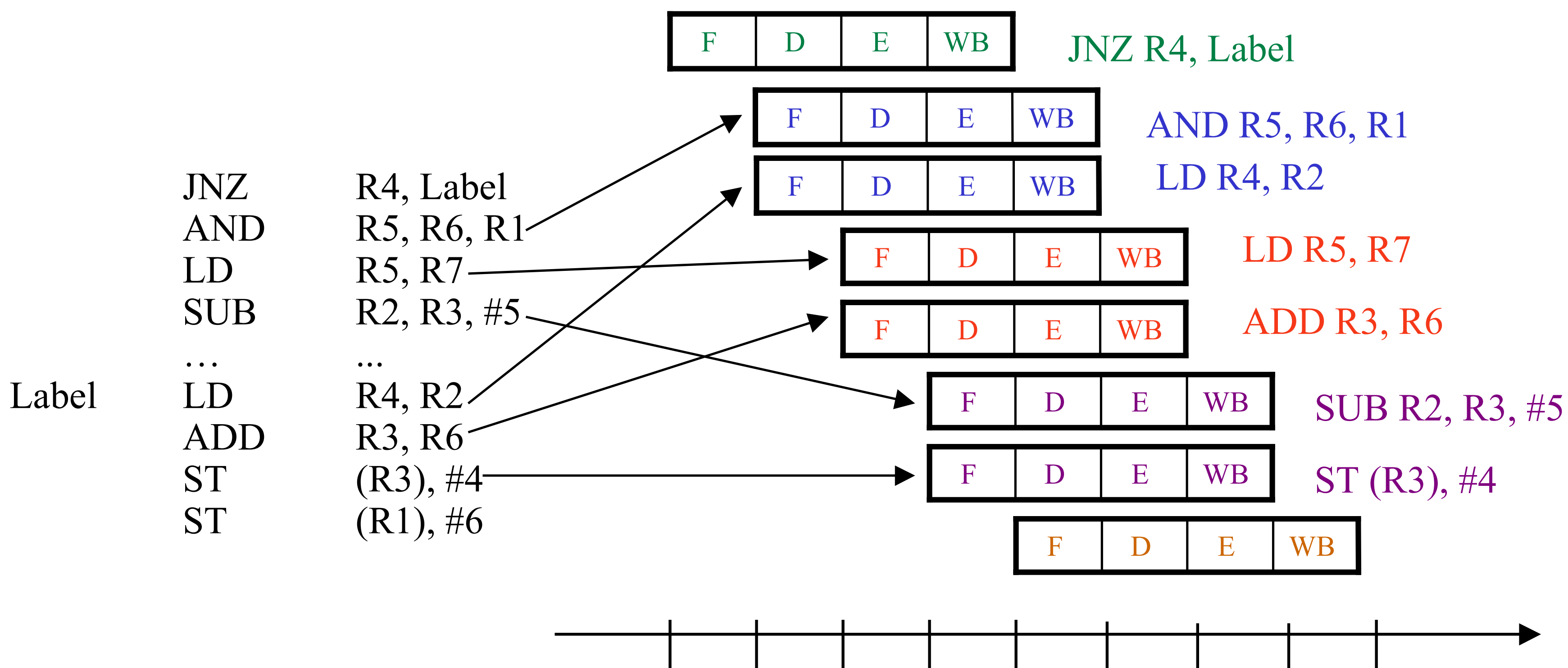
- Tilstandsdiagram for dynamisk hopp-prediksjon
  - For å få bedre prediskjon benyttes 2-bits tilstand, dvs oppførselen må være lik for de 2 foregående gjennomløpene

Hva skjedde?

Hva tror man vil skje?

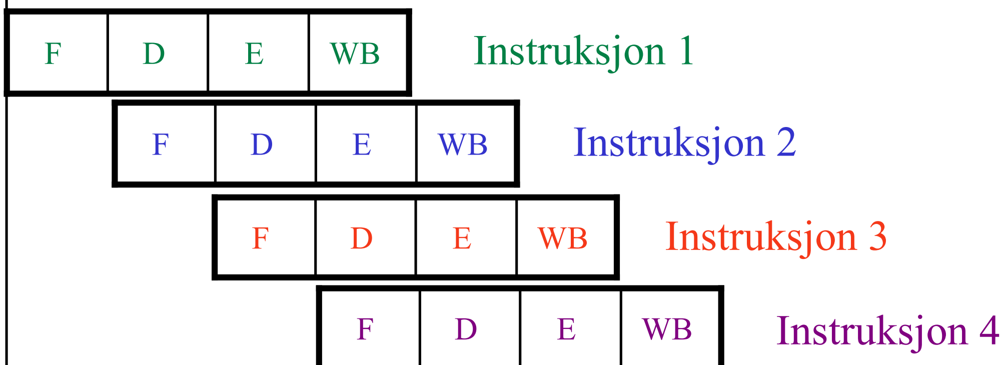


- Løsning 2: Parallelleksekvering av hoppinstruksjoner
  - Istedenfor å forutsi, starter prosessoren eksekvering av begge kodesekvenser i parallell til riktig hopp kan bestemmes

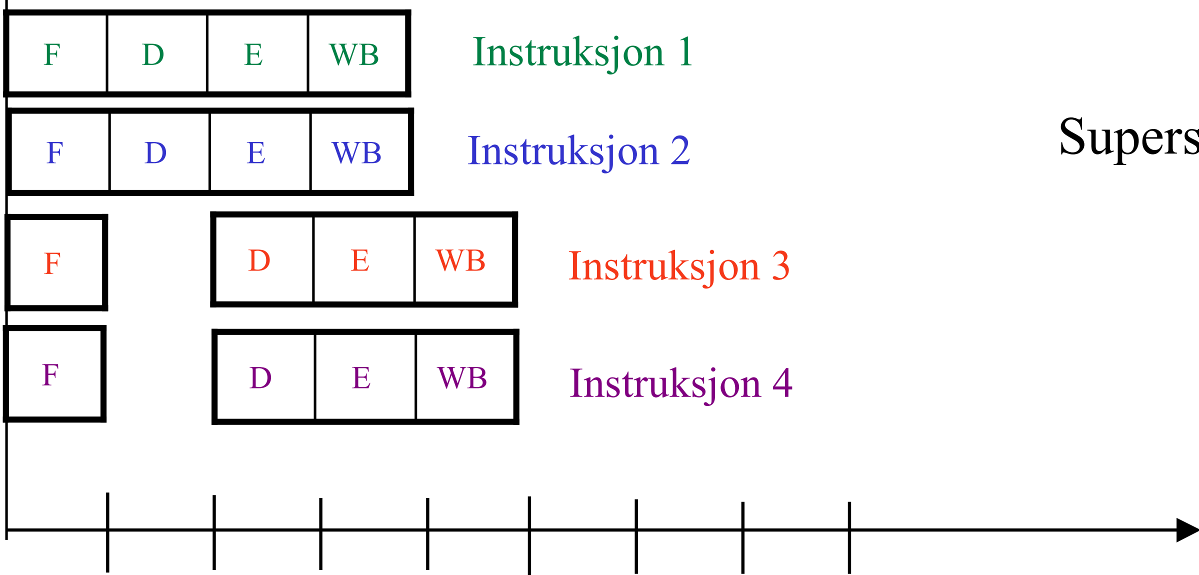


Superskalare prosessorer

- En vanlig pipelined 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 HW-enheter som gjør heltalls og flyttalls-aritmetikk, 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.
- Pentium og G4 er superskalare



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



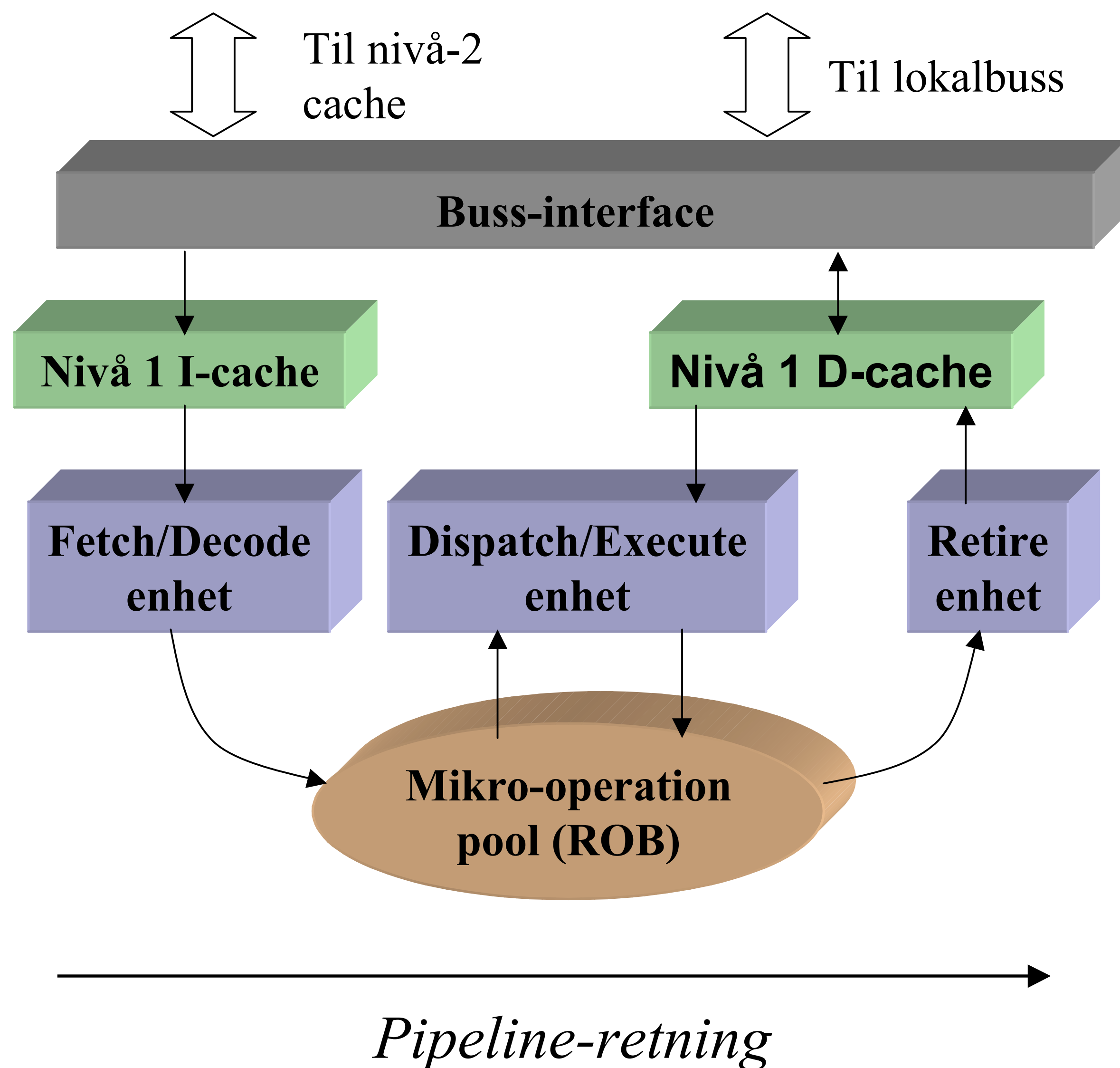
F
D
E
WB
Instruksjon 1
Instruksjon 2
Instruksjon 3
Instruksjon 4
Superskalar pipelining

- Parallelliteten gjør superskalare prosessorer mye vanskeligere å designe

Blokkdiagram for Pentium III/4

- Fetch/Decode, Decode/Execute og Retire utgjør tilsammen en 3 trinns høynivå pipeline.

- ROB inneholder informasjon om delvis eksekverte instruksjoner som av en eller annen grunnen venter på å blir ferdige.



- 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 pipeline skalar pipeline
- Dispatch/Execute og Retire-enheten inneholder en 5 trinns pipeline tilsammen
- 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 (P4 har 20-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:
  - Ovesettelse 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)

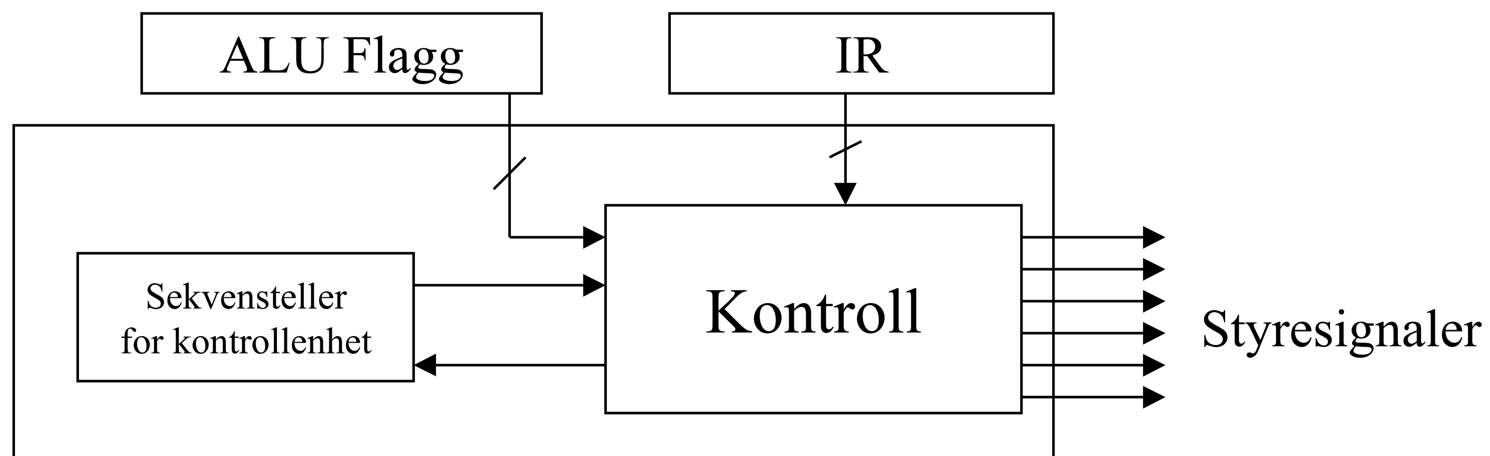
Dispatch/Execute enheten

- Reservasjonsstasjonen kan innholde opptil 20 mikroinstruksjoner som venter på å bli utført.
- Opptil 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 Retire-enheten

- 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

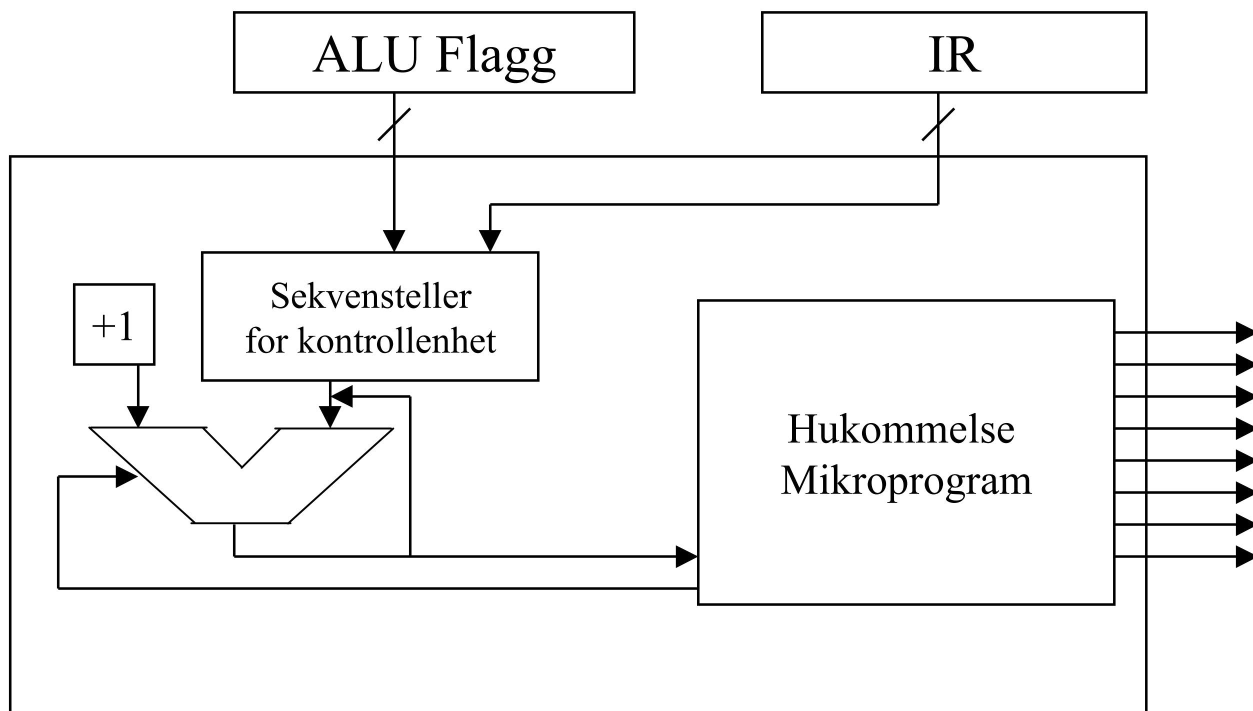
Design av kontrollenhet

- 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 fleksible fordi mikroprogrammet kan byttes undervies
  - Lettere å rette opp feil ved designet
  - Krever flere klokkesykler for å eksekvere en instruksjon
  - Krever stort CPU-areal

RISC og CISC (1)

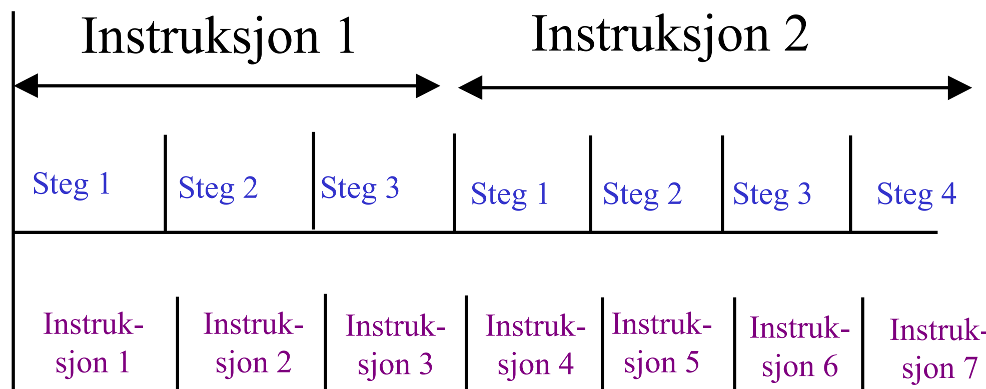
- To forskjellige filosofier for design og organisering av en CPU
- CISC-arkitektur har mange maskinspråk-instruksjoner som kompilatorer kan bruke ved oversettelse fra høynivå-språk til masinkspråk
- Assemblerprogramering blir enklere i en CISC-arkitektur fordi det finnes mange spesialiserte instruksjoner.
- En CISC-instruksjon kan bestå av et variabelt antall midre steg eller sub-instruksjoner, 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 mye færre maskinspråk-instruksjoner tilgjengelige for assemblerprogram 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 bruker (tester har vist at Load/Store og hopp-instruksjoner 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.

UNIVERSITETET
I OSLO

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 o 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