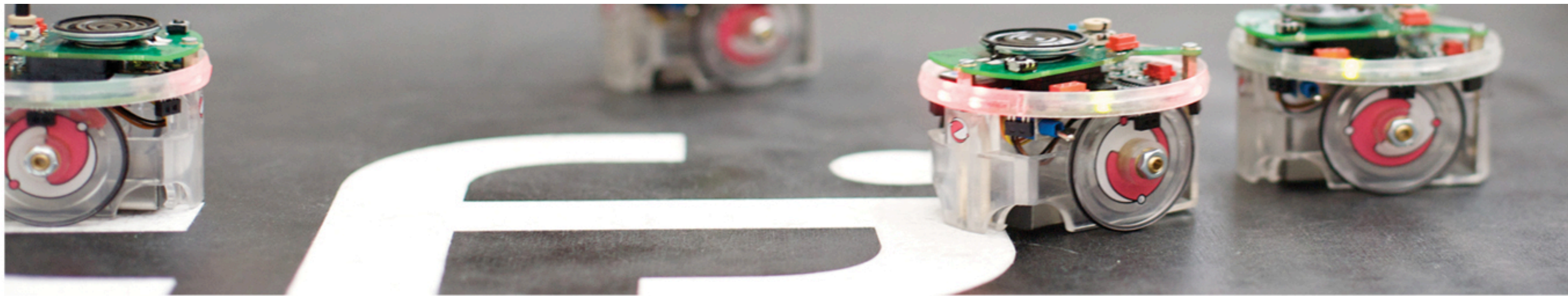


UiO : Institutt for informatikk
Det matematisk-naturvitenskapelige fakultet

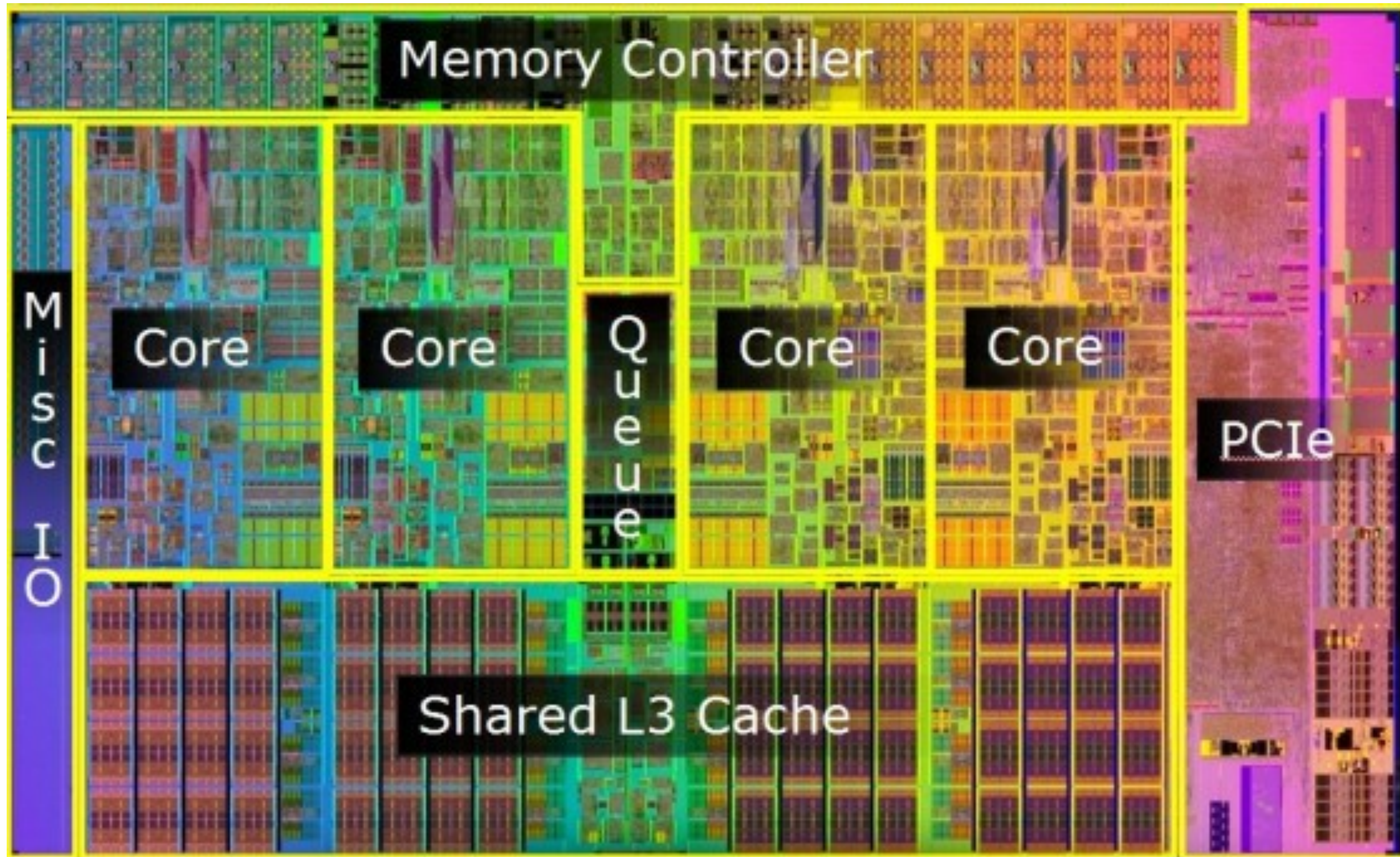
IN1020

Datamaskinarkitektur

Hovedpunkter

- Von Neumann Arkitektur
- BUS
- Pipeline
- Hazarder

Intel Core i7



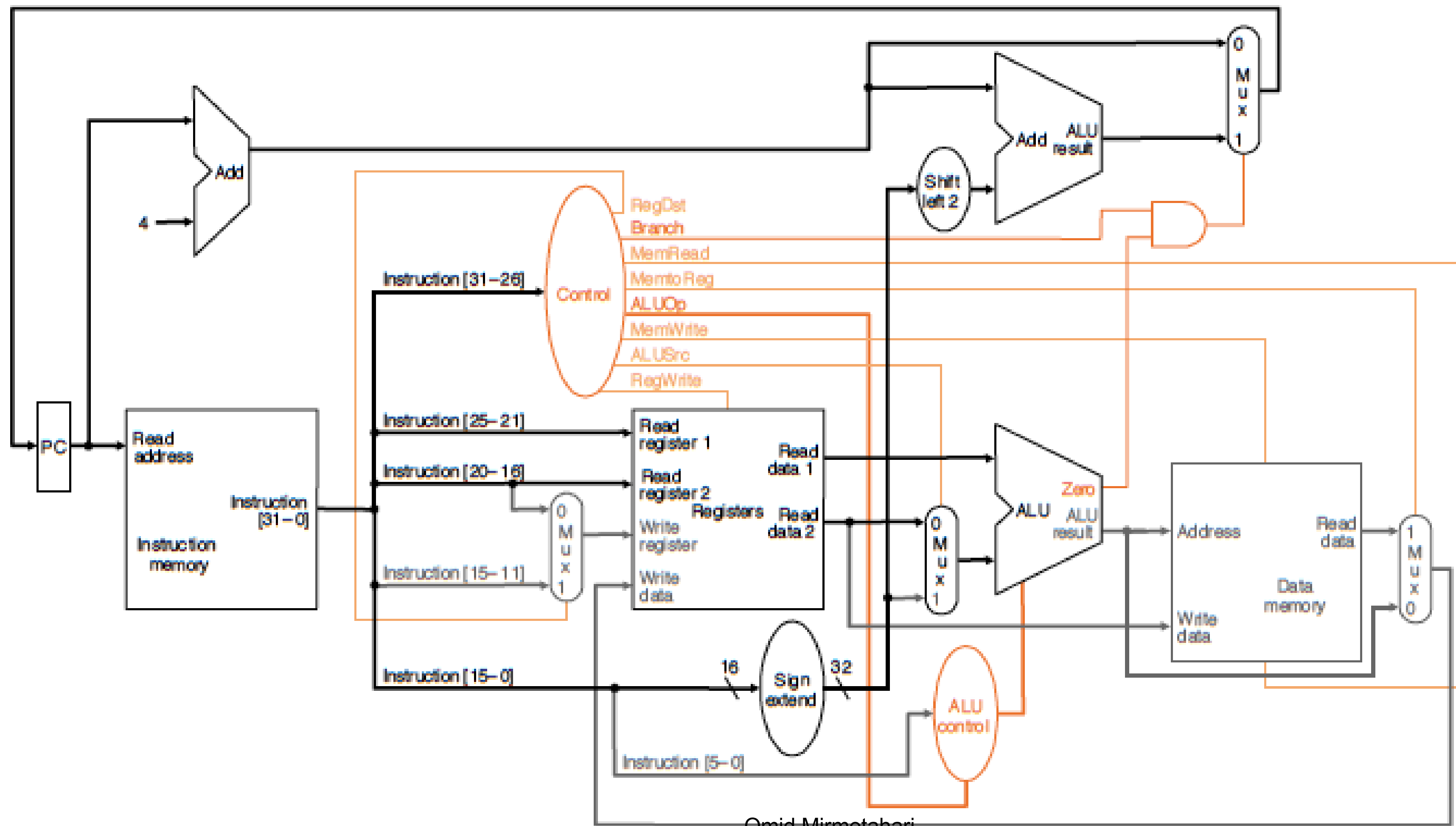
Von Neumann Arkitektur

John von Neumann publiserte i 1945 en model for datamaskin arkitektur som brukes fortsatt den dag i dag. Hovedbidraget hans og unikheten i denne modellen er å bruke et enkelt minneelement som både skal brukes for program og data.

Data- og instruksjonsbus

- Bus er kommunikasjonskanalen mellom registere, funksjonelle enheter (ALU), minne og I/O enheter.
- Bus kan deles mellom flere enheter, men kun en som kan sende av gangen.
- I en von Neumann arkitektur er det en bus mellom minne og CPU som skal både overføre instruksjon og data, dette vil da være flaskehalsen.
- Internt i en CPU er den en eller flere bus(er) som overfører data mellom interne registere.

Eksempel



Oppsummering av én-sykel implementasjon

- Høy effektivitet oppnås gjennom:
  1. Alle instruksjoner bruker én klokkesykel
  2. Operasjoner utføres samtidig ved å bruke flere identiske komponenter
  3. Instruksjonsminne og dataminne aksesseres samtidig

(1) Alle instruksjoner bruker én klokkesykel

- Klokkeperioden må være like lang som den **lengste** forsinkelsen for enhver instruksjon gjennom data-path´en
  - Resultat: Raskere instruksjoner tvinges til å bruke mer tid enn nødvendig
  - Mulig løsning: Klokke med variabel periode
  - Vurdering: Vanskelig å implementere

(2) Bruke flere identiske komponenter

- Trenger flere like komponenter som egentlig gjør samme jobb
  - Resultat: Trenger mer plass på CPU´en og øker effektforbruket
  - Mulig løsning: Endre kontroll-logikk slik at samme komponent kan brukes til flere oppgaver.
  - Vurdering: Vanskelig, umulig eller lite effektivt hvis man krever at alle instruksjoner skal ta én klokkesykel.

(3) Inst. MEM og Data MEM aksesseres samtidig

- Må kunne aksessere to forskjellig lokasjoner i samme minne-enhet samtidig.
  - Resultat: Må enten ha 2 separate minne-enheter, eller kunne adressere flere ord samtidig i én minne-enhet
  - Mulig løsning: Operere med en minne-enhet for program, og en annen for data
  - Vurdering: Lite fleksibelt ved blant annet dynamisk data-allokering, og mer plasskrevende (doble adressebusser osv)

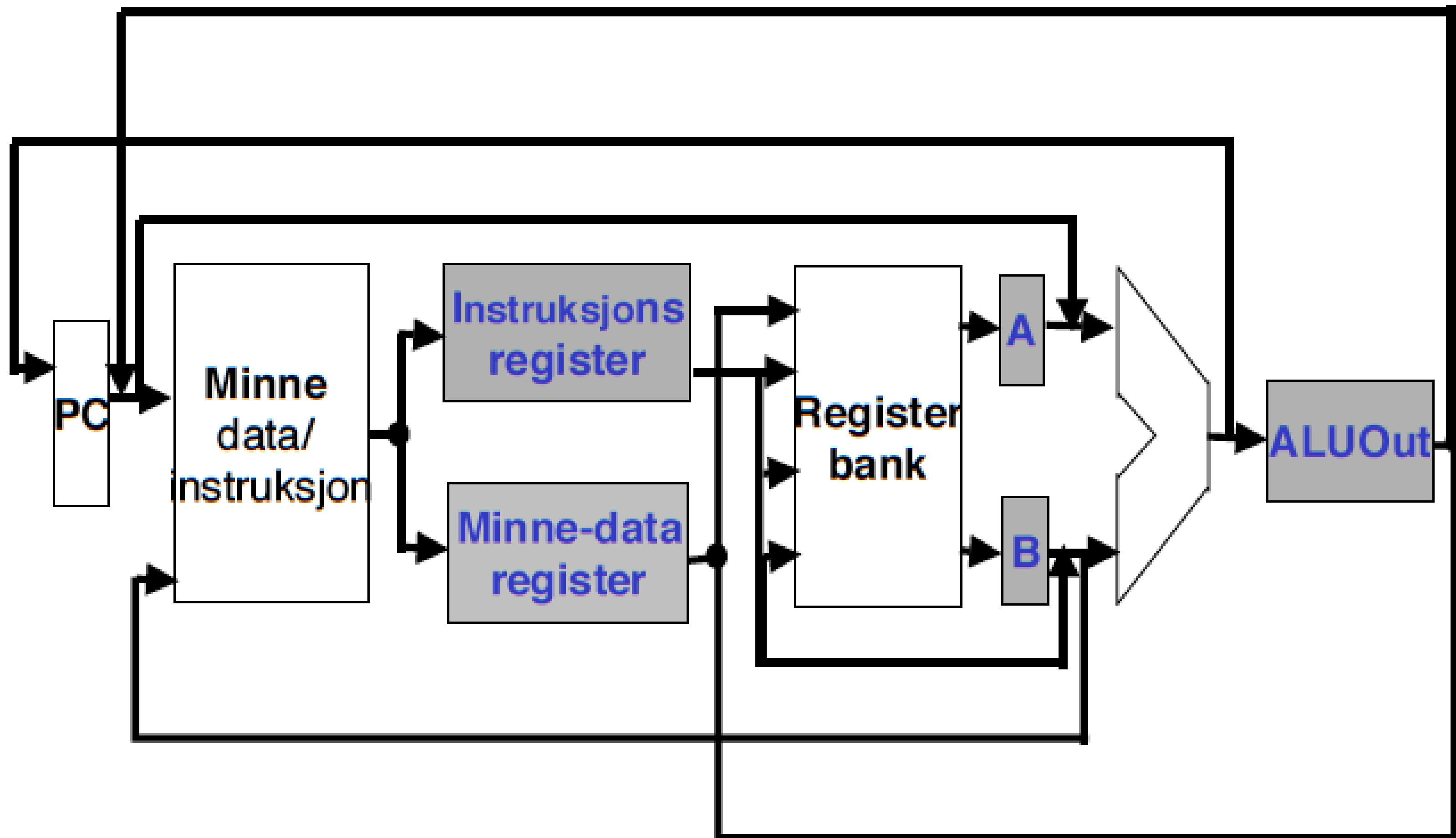
Forbedring av én-sykel designet

- Innfører to nye strategier for å øke hastighet:
    1. Multicycle
    2. Pipelining

Multicycle

- Multi-cycle bruker mer enn én klokkesykel per instruksjon hvis nødvendig
- De ulike trinnene i en instruksjon kan dele samme komponent
- Trenger ikke separat data- og instruksjonsminne
- Trenger ekstra registre for å mellomlagre data

Multi cycle



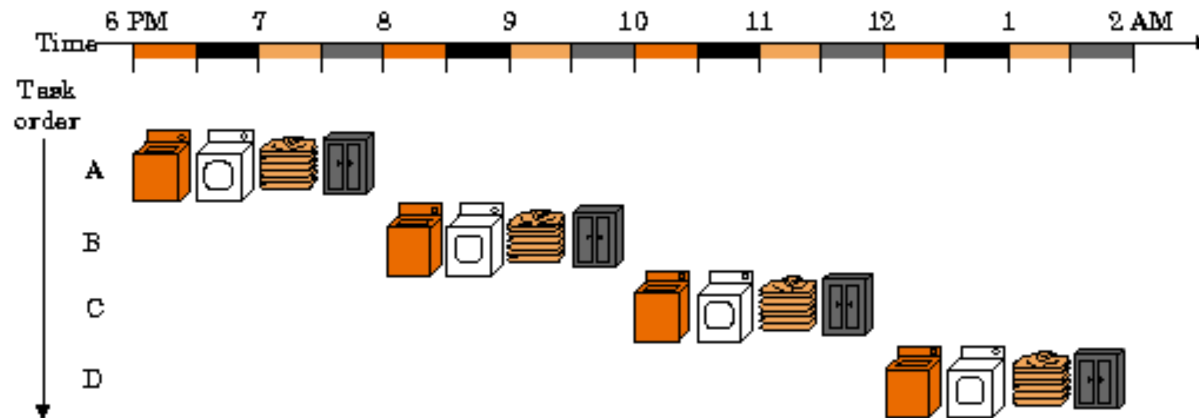
Pipelining

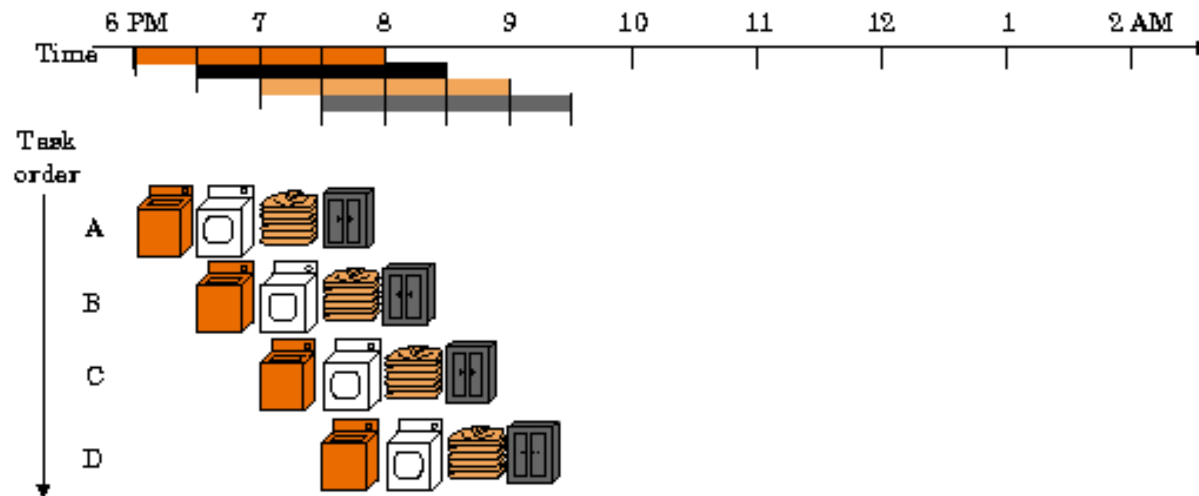
- Innfører samlebåndsprinsipp for eksekvering av instruksjoner
- Hver instruksjon må splittes opp i uavhengige deler (subinstruksjoner) som utføres etter hverandre
- Hver subinstruksjon kan utføres uavhengig av de andre subinstruksjonene
- Neste instruksjon settes igang før forrige instruksjon er helt ferdig.
- **NB!! Hver instruksjon tar like lang tid å utføre, men prosessoren utfører flere instruksjoner i et gitt tidsrom!**

Pipelining - Analogi

- Vasking av klær

    1. Vaske i vaskemaskin
    2. Trøke i tørketrommel
    3. Brette sammen
    4. Sette dem i skapet

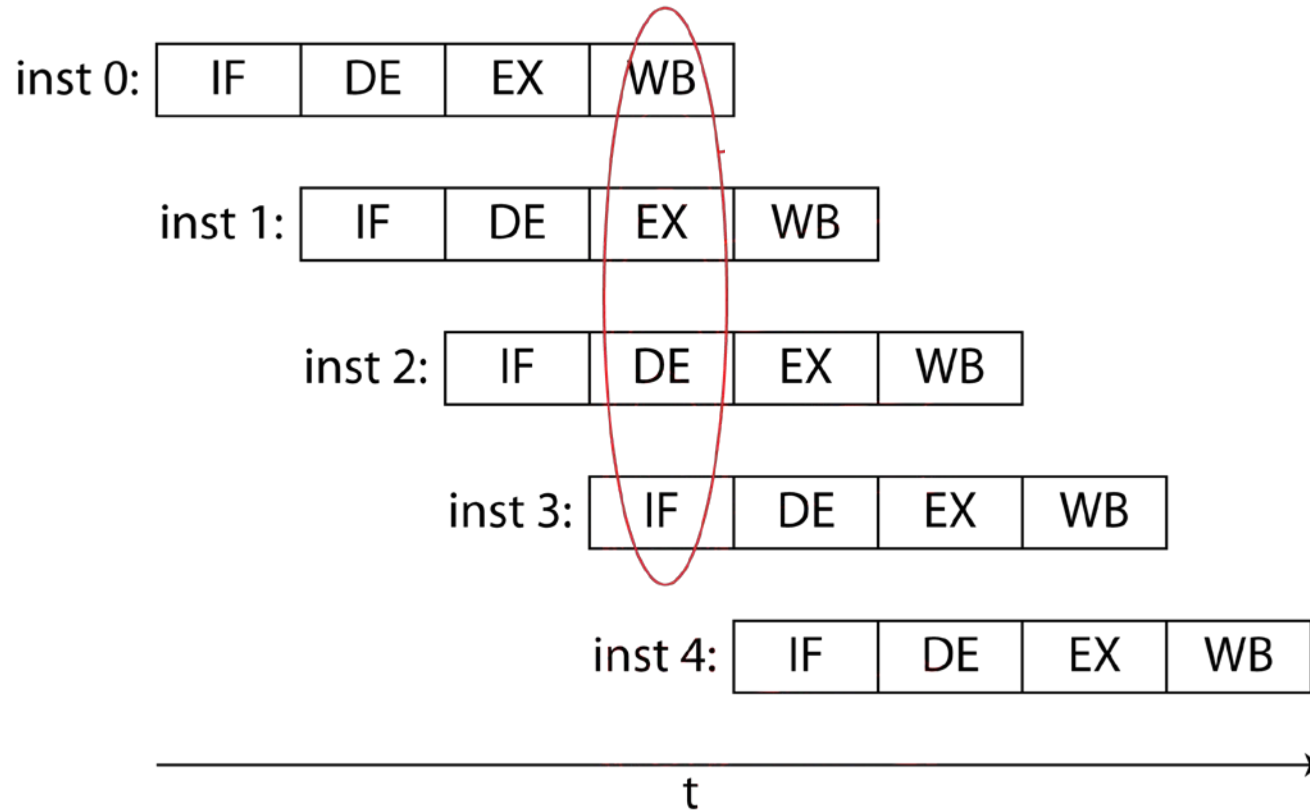

6 PM
7
8
9
10
11
12
1
2 AM
Time
Task order
A
B
C
D

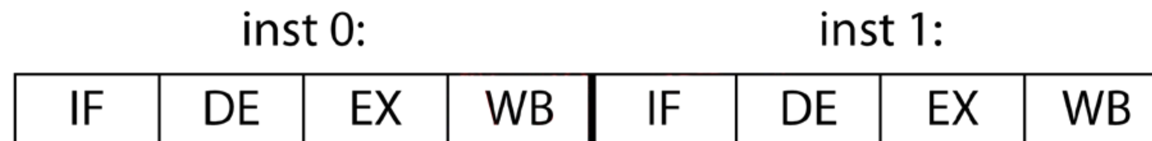

6 PM
7
8
9
10
11
12
1
2 AM
Time
Task order
A
B
C
D

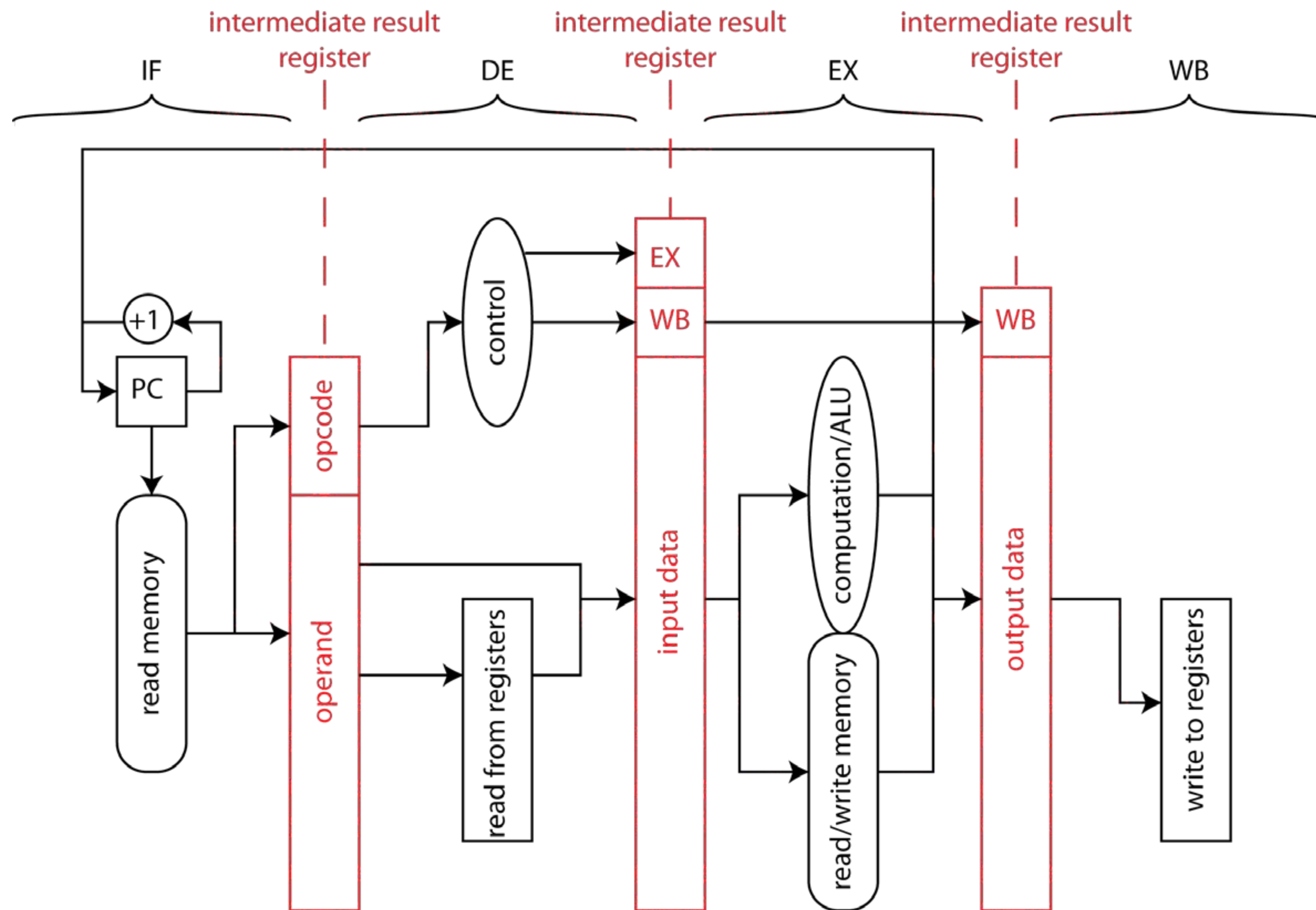
Pipelining instruksjonssett

- Eksempelvis kan vi ha følgende subinstruksjoner i en pipeline
  - IF: Instruction fetch (get the instruction)
  - DE: decode and load (from a register)
  - EX: Execute
  - WB: Write back (write the result to a register)

- **PS! Read og write fra register/MEM kan gjøres i hver sin halvdel av en sykel (1/2 klokkeperiode)**


with pipelining
inst 0:
IF
DE
EX
WB
inst 1:
IF
DE
EX
WB
inst 2:
IF
DE
EX
WB
inst 3:
IF
DE
EX
WB
inst 4:
IF
DE
EX
WB
t


without pipelining
inst 0:
inst 1:
IF
DE
EX
WB
IF
DE
EX
WB


IF
intermediate result register
DE
intermediate result register
EX
intermediate result register
WB
+1
PC
read memory
opcode
operand
control
read from registers
EX
WB
input data
computation/ALU
read/write memory
WB
output data
write to registers

Pipeline med flere trinn

- Denne 4-trinns pipeline er den korteste som finnes for en CPU.

- Moderne CPU bruker vesentlig flere trinn

- Pentium III har 16 trinn, Pentrium 4 har 31 trinn

Speed-up

- Det å ha en 4 trinns pipeline betyr ikke at man får 4 ganger raskere prosessering. Det går alltid noe tid bort til administrering av instruksjoner.

Effektiv speed-up

En ***k***-trinns pipeline som trenger én klokkesykel pr trinn med eksekverering av ***n*** instruksjoner vil ha en speed-up på:

$$\frac{kn}{k+n-1}$$

For veldig store programmer så vil vi nærme oss k, men dette skal vi se i neste foil hvorfor vi ikke klarer.

Komplikasjoner ved pipelining

- Til enhver tid kan det være subinstruksjoner fra opptil 4 instruksjoner i en pipeline
- Noen ganger er ikke alle subinstruksjonene gyldige
- Neste instruksjon kan ikke eksekveres rett etter hvis hopp-betingelsen slår til
- En slik situasjon kalles HAZARD. Vi har tre typer:
  1. Resource hazard
  2. Data hazard
  3. Control Hazard

Resource Hazard

- Kan oppstå hvis to subinstruksjoner i pipelinen ønsker å aksessere samme ressurs (eks minne)
  - Løsning 1: Designe slik at de ikke oppstår (!)
  - Løsning 2: Stoppe (STALL) pipelinen lenge nok til resurssen kan aksesseres sekvensielt. (lese inne en NOP?)
  - Løsning 3: Bruke lokale register som er organisert i en REGISTER FILE
  - Løsning 4: Bruke Harvard arkitekturen som har 2 separate minner for data og instruksjon.

Resource Hazard

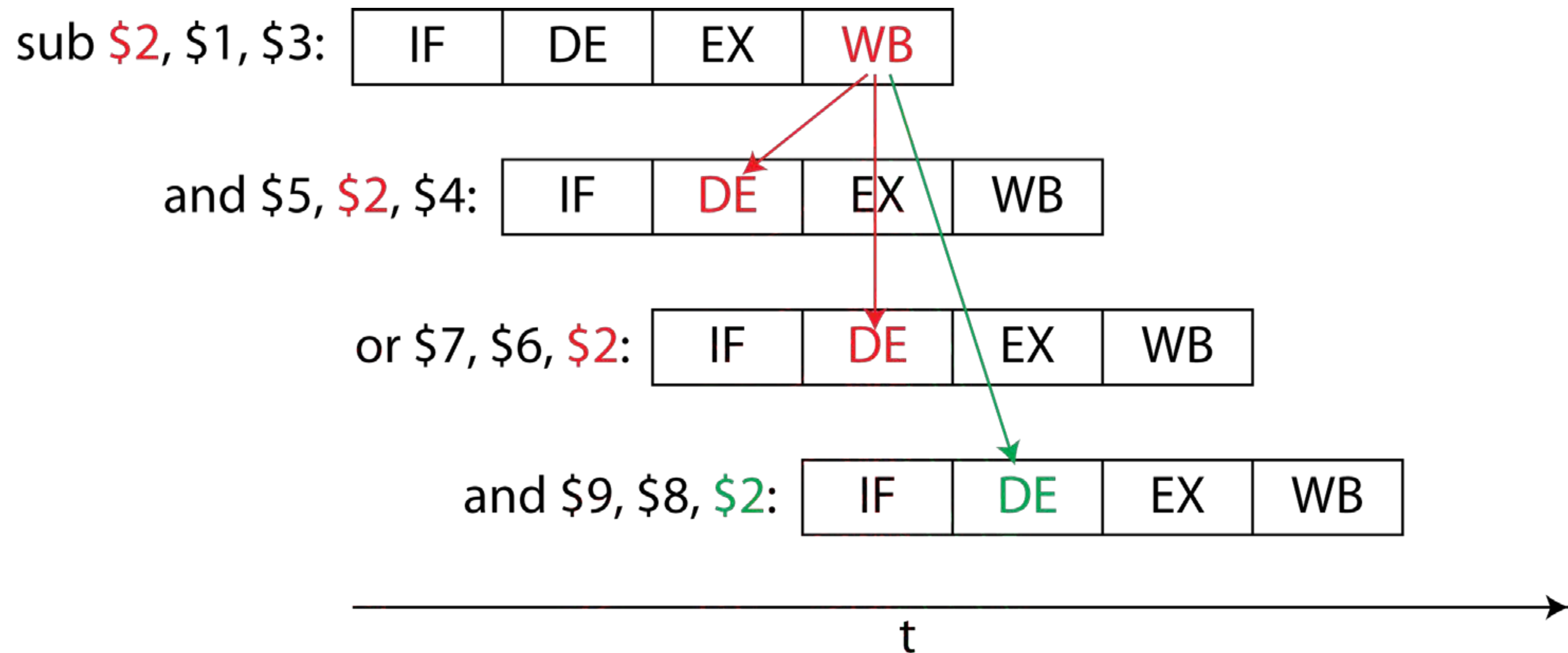
- Andre resources kan også gi harzards, litt avhengig av hvordan CPU arkitekturen er bygget opp
  - Minne
  - Cache
  - Register files
  - Busser
  - ALU
  - osv

Data Hazard

- Data hazard oppstår fordi to forskjellige instruksjoner trenger å aksessere samme data samtidig.

- Trenger resultat fra forrige instruksjon før denne har produsert gyldig svar.

Data Hazard



Data Hazard

– Løsning 1: Detektere avhengigheten i IF-stadiet av pipelinen for første instruksjon og stoppe (STALL) neste instruksjons EX-stadie til WB-stadiet for første instruksjon er ferdig.

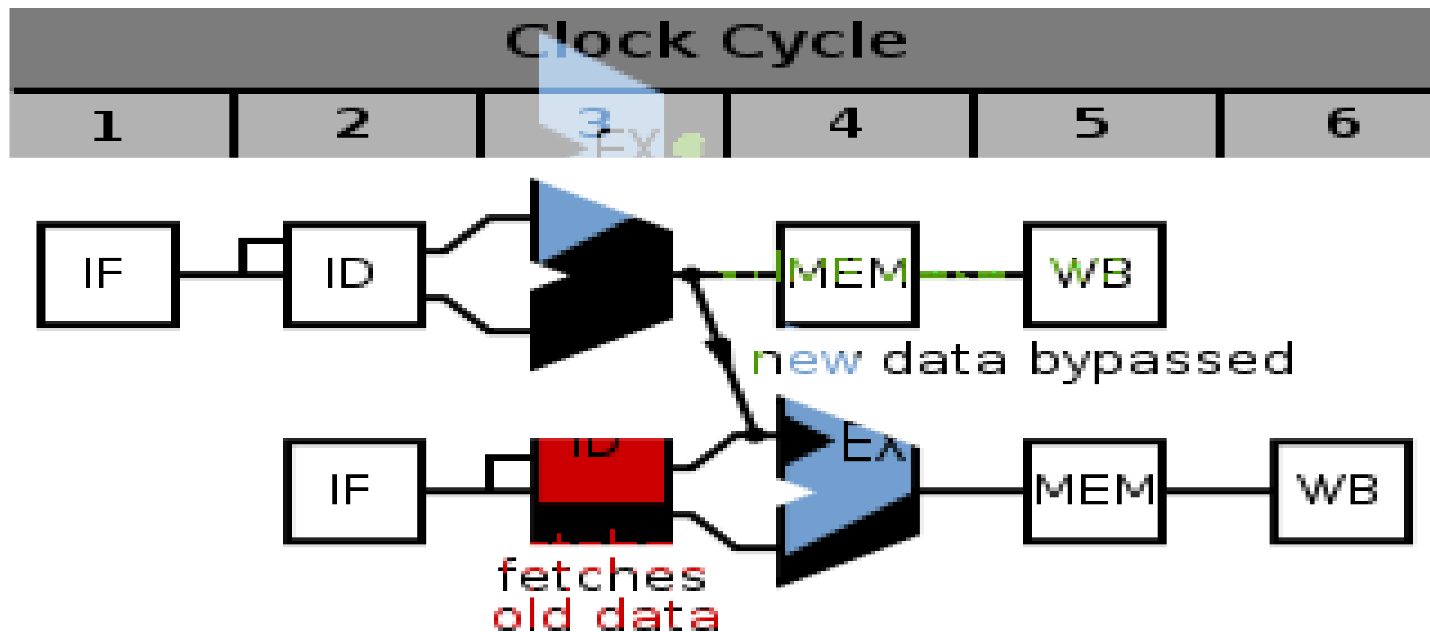
– Løsning 2: Snu om på rekkefølgen av instruksjonene slik at man ikke er avhengig av instruksjon rett før i pipelinen.

– Løsning 3: Ha en snarvei (FORWARDING) i pipelinen, i dette tilfelle en direkte datapath som aktiviseres ved en hazard.

Data Hazard

Løsning 3 er bedre fordi alt skjer i hardware og ikke i kompilatoren.

Forwarding benyttes også mellom andre enheter i datapathen, feks mellom utgangen av MEM og inngangen til ALU

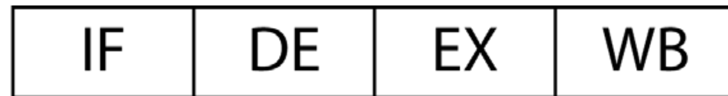


Control Hazard

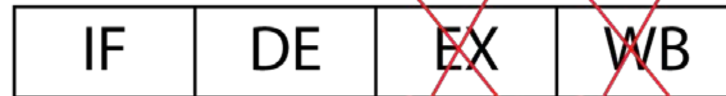
- I utgangspunktet har vi tenkt en pipeline som innhenter neste instruksjoner fortløpende. Men problemet oppstår når vi får en JUMP instruksjon. Da må vi tømme pipelinen for andre instruksjoner og lese inn den nye.

Control Hazard

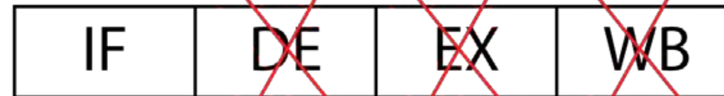
4: jz 100



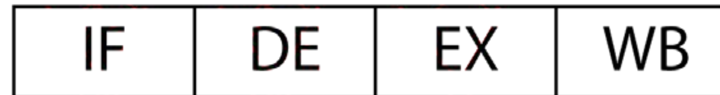
8: and $5, $2, $4



12: or $7, $6, $2



100: and $9, $8, $2



Control Hazard

- Løsning 1: Stoppe (STALL) ikke hent inn andre instruksjoner før det er helt klart at man ikke skal hoppe.

- Løsning 2: Prøv å forutsi om man skal hoppe, eksekver som vanlig ved ikke-hopp, stop pipelinen som løsning 1 ved hopp.

- Løsning 3: Dobling av hardware for deler av pipelinen. Dette for å kunne ha to parallelle pipelines som kan inneholde begge instruksjonsadressene. Når det er klart om instruksjonen er en hopp så “flushes” den andre.