Kap. 6 Pipelining og superskalare teknikker i prosessorer

- **Pipeline**
  - synkrone/asynkrone
  - lineær / generalisert
  - instruksjoner / aritmetikk / lageraksess
  - statisk (en-funksjons) /dynamisk (flerfunksjons)
- **Prefetching**
- **Branch-handling**
- **Forwarding**
- **Scoreboarding**
- **"Hazarder"**

Lineær pipeline

- **For instruksjoner, aritmetikk og lageraksesser**
- **Fig 6.1**
- **Reservasjonstabell**
- **Klokke sykel = *t* = T(langsomste trinn) + T(latch)**
- **Frekvens = *f* = *1/t* = maksimal ytelse**
- **Hastighetsøkning**
  - $S(k) = T(1) / T(k) = nk / [k + (n-1)]$ (Eq. 6.5)
  - Fig. 6.2 a)
- **Tre nivåer av pipelining**
  - vanlig
    - » oppdeling av beregning i trinn
    - » $2 < k < 10$ (15 i sjeldne tilfelle)
  - mikropipelining
    - » pipelining innenfor trinn
  - makropipelining
    - » "kjede av prosessorer"
      - MISD (?)

Lineær pipeline, ytelse

- **Beregning av optimalt # trinn**
  - ytelse/kost forhold (Eng. Performance Cost Ratio (PCR))
  - (Larson 1973)
    - » t = beregningstid på vanlig prosessor
    - » på en "tilsvarende" k-trinns pipeline vil vi ha en klokkeperiode på $p = t/k + d$, hvor d er latch-forsinkelse
    - » pipeline har da maksimal ytelse på $f = 1/p = 1/(t/k + d)$
    - » pipeline-kostnad kan estimeres som $c + kh$ der c er kostnaden for alle trinnene og h er latch-kostnad
    - » PCR = $f/(c + kh) = [(t/k+d)(c + kh)]^{-1}$ (Eq. 6.6)
    - » Fig. 6.2 b
- **Effektivitet**
  - $E(k) = S(k)/k = n/[k + (n-1)]$ (Eq. 6.8)
- **Pipeline ytelse (Eng. throughput)**
  - $H(k) = n/[k+(n-1)]t = nf/[k+(n-1)]$
    - » Hvor t er varighet av en klokkeperiode

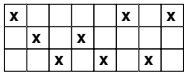
Ikke-lineær pipeline

- **En lineær pipeline er statisk, dvs. kan bare utføre en funksjon av gangen**
- **Dynamisk pipeline**
  - kan omkonfigureres hurtig mellom ulike funksjoner
    - » mer økonomisk utnyttelse av maskinvare
    - » mer kompleks styring
  - "feedforward og feedback" forbindelser
    - » i tillegg til de vanlige "strømlinje"-forbindelsene
  - kalles også ikke-lineær pipeline
- **Reservasjonstabell sentralt verktøy**
- **Fig. 6.3**

Pipeline forsinkelse (Eng. latency)

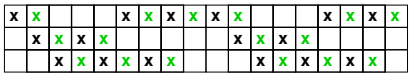
- **Tid mellom to oppstartinger av en beregning i pipeline**
- **Må velge verdier som ikke gir kollisjon**
  - forbudte forsinkelser
- **Fig. 6.4**
- **Forsinkelses-sekvens**
  - sekvens av pipeline forsinkelser som ikke gir kollisjon
- **Forsinkelses-sykel**
- **Fig. 6.5**

Eksempel, funksjon X

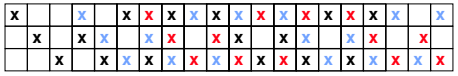


Funksjonen X

Sykel (1,8)



Sykel (3)



Kollisjonsfri tidsplanlegging (Eng. scheduling)

- **Ønsker lavest mulig gjennomsnittlig forsinkelse uten kollisjoner**
- **Davidson (1971)**
- **Kollisjons-vektor**
  - m er største forbudte forsinkelse
  - $C = (C_m C_{m-1} \ldots C_2 C_1)$ er en m-bits kollisjons-vektor
  - F.eks. 1101 angir at forbudte forsinkelser er 1, 3 og 4
- **Tilstandsdiagram**
  - Fig. 6.6
  - Tilstands-diagrammet viser sykler
    - » Ønsker å finne sykel med minimal gjennomsnittlig forsinkelse (Eng. MAL)
  - Enkle sykler
    - » Sykel hvor ingen tilstand besøkes mer enn maks. en gang
  - "Grådige sykler"
    - » Enkel sykel hvor en fra hver tilstand velger den transisjon som har lavest forsinkelse
    - » MAL finnes blant disse
- **Optimalisering**
  - Kan oppnå bedre ytelse ved å legge inn forsinkelses-trinn i pipeline
  - Fig. 6.7 og Fig. 6.8

Teoretiske grenser for MAL-verdi i pipeline

- **Shar (1972)**
  - 1) en nedre grense for MAL-verdien er det maksimale antall kryss i en rad i reservasjons-tabellen
  - 2) MAL-verdien er mindre el. lik den gjennom-snittlige forsinkelsen av en vilkårlig "grådig sykel" i tilstandsdiagrammet.
  - 3) en øvre grense for gjennomsnittlig forsinkelse i en "grådig sykel" er $z + 1$, der z er antall 1'ere i den initielle kollisjons-vektoren
- **Konsekvenser:**
  - Den nedre grensen garanterer optimal verdi
    - » Ønskelig å redusere antall kryss i en rad (samtidig som riktig beregning beholdes)
    - » Ett kryss i hver rad er optimalt m.h.p. hastighet, jfr. "vanlig" lineær pipeliine

Mer om pipeline-ytelse

- **Pipeline-ytelse = gj.snittlig # oppstartinger (Eng. task initiations) pr. klokke-sykel**
  - oppstartingsrate = 1 /MAL
- **Pipeline-effektiviet**
  - Trinn-utnyttelse
  - Se f.eks. fig 6.5b), 8/9 for uendelig lange sekvenser
- **Kortere forsinkelses-sykel gir bedre ytelse og effektivitet**

Instruksjons pipeline

Kap. 6.3, side 280->

- **Instruksjons pipeline for skalare CISC og RISC prosessorer**
- **Fig. 6.9 a) Typisk oppbygging**
  - Henting
    - normalt fra hurtigbuffer
    - evt. "prefetch"
  - Dekoding
    - finner ut hva som skal gjøres
    - avklarer ressursbehov, som registre, busser, funksjonelle enheter
  - Oppstarting (Eng. Issue) ("Operand fetch")
    - reserverer ressurser
    - leser operander fra registre
  - Utføring
    - gjerne flere trinn
    - #(RISC) < #(CISC) < #(CRAY)
  - Tilbakeskriving
    - skriver resultat(er) i registre
- **Fig. 6.9 b-c)**
  - Viser hvordan endret rekkefølge av instruksjoner kan redusere ventetiden, dvs. gi bedre utnyttelse

Teknikker for ytelses-forbedring i instruksjons pipeline

- **Forhåndshenting av instruksjoner**
  - Eng. prefetching
  - Sekvensielt buffer
  - Hoppadresse buffer (Eng. target buffer)
  - Løkke buffer
    - CRAY-1 med "back-eight-test"
  - Se fig. 6.11
- **Flere funksjonelle enheter**
  - Flaskehals
    - trinn i pipeline med flest kryss i reservasjonstabell
  - dupliser trinnet, dvs. benytt parallellitet i "bredden"
    - kompliserer administrasjon i prosessoren
    - reservasjons-stasjoner (RS)
- **Intern foroversending (Eng. Internal (data) forwarding)**
  - Utbytting av lageraksesser med registeraksesser
    - Ønsker generelt å redusere lager-trafikk
  - "bypassing" el. "short-circuiting" av data

Intern foroversending (Eng. Internal data forwarding)

- ***"forwarding"***
  - "Definisjon" i [Hwa93] (og [HB84]): Utbytting av lageraksesser med registeraksesser
    - Ønsker generelt å redusere lager-trafikk
    - programvare-teknikk (kompilator)
  - Fig. 6.13: store-load, load-load, og store-store - forwarding
- ***"internal forwarding"***
  - Definisjon i [HP90] (for en funksjonell enhet (ALU)): ALU resultat sendes alltid tilbake til ALU inngangsregistre. Maskinvare sjekker om ny operand er forrige resultat, og i så fall henter en ikke inn ny operand fra register eller lager. Også kalt "**bypassing**" el. "**short-circuiting**" av data
    - maskinvare-teknikk
    - Eks. IBM 360/91, Tomasulo, (flere funksjonelle enheter)

"Hazarder"

- **Problemer forårsaket av avhengigheter mellom instruksjoner i en instruksjons-pipeline**
  - Lesing og skriving av lokasjon som er felles for ulike instruksjoner som er under utføring i en pipeline kan gi problemer hvis rekkefølgen av aksesser til felles lokasjon som programmereren forventer brytes
  - Dette kan oppstå p.g.a. overlappende utførelse og fordi lesing/skriving skjer i ulike trinn
- **"Control hazards"**
  - også kalt "branch hazards"
  - problemer som oppstår ved betingete hopp
- **"Data hazards"**
  - p.g.a. data avhengigheter
  - Read-After-Write (RAW) hazard
    - tilsv. flytavhengighet (Def. [Hwa93] s. 52)
  - Write-After-Write (WAW) hazard
    - tilsv. output-avhengighet
  - Write-After-Read (WAR) hazard
    - tilsv. antiavhengighet

RAW, WAW og WAR-hazarder

- **Read-After-Write (RAW) hazard**
  - Fig 6.15(a): Hvis instruksjon I skriver en verdi som instruksjon J leser, får vi feil hvis J leser verdien før I har skrevet den ut. Dette kan lett skje siden operand-henting ofte utføres i ett tidligere trinn enn resultat-skriving. F.eks.:

```
I:(R1 := R2+R3): IF DE OF EX WB
J:(R4 := R1+R5):    IF DE OF EX WB
```

Utføres i gal rekkefølge

  - **Write-After-Write (WAW)** hazard
    - tilsv. output-avhengighet
    - Fig 6.15(b): Hvis en pipeline skriver i flere trinn får vi feil hvis I skriver til samme lokasjon etter J
  - **Write-After-Read (WAR)** hazard
    - tilsv. antiavhengighet
    - Fig 6.15(c): Instruksjon J skriver til en lokasjon før instruksjon I rekker å lese denne
    - Kan bare skje hvis pipeline har et skrive-segment (trinn) som ligger foran et lese-segment. Kan f.eks. skje ved "autoincrement"-adressering.[HP90].

Håndtering av hazarder

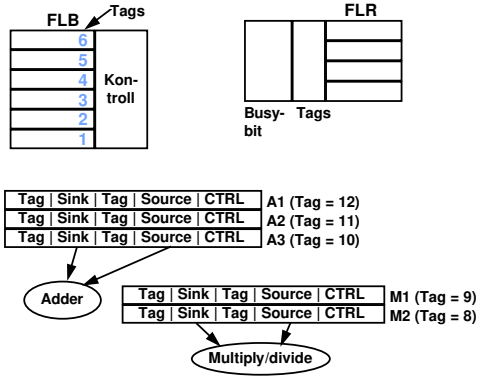
- **Hazard deteksjon**
  - **sentralisert**, under instruksjonshenting
    - sammenlikner ressursbruk for nye instruksjoner med de som er under behandling
  - **distribuert** i alle trinn
    - mer fleksibelt, effektivt
    - mer kostbart
- **Hazard fjerning**
  - stopp opp behandling av instr. *J, J+1,* ...osv. inntil instr. *i* har kommet så langt at problemet er fjernet
    - det enkleste
  - stopp opp behandling av instr. *J*, men fortsett med *J+1* osv.
    - må da sjekke muligheter for nye problemer (hazarder) p.g.a. at *J* er utsatt
    - betraktelig mere komplisert
  - intern foroversending, register merking, og reservasjons-stasjoner
    - avhengigheter fører til at instruksjoner "koples sammen" (kjedes) i maskinvare

Instruksjons planlegging i en prosessor (Eng. instruction scheduling)

- **Statisk planlegging**
  - optimaliserende kompilator
    - bytter om rekkefølge på instruksjoner for å redusere ventetid på operander
    - billig (enkelt) og fleksibelt -> vanlig
- **Dynamisk planlegging**
  - register-merking
    - IBM 360/91, Tomasulo, 1967
    - distribuert logikk/styring
    - instruksjoner venter på operander i reservasjons-stasjoner (RS), som "snuser på bussen" og snapper opp operander straks de er produsert
  - "scoreboarding"
    - CDC 6600
    - sentralisert logikk/styring (scoreboard) som vet "hvem som venter på hva"
    - Fig. 6.17
  - Begge teknikkene er en slags maskinvare realisering av data-drevne beregninger (instruksjoner venter til operander er klare)
- **Statisk og dynamisk kan kombineres**
- ***Etablerte teknikker fra "gamle" stor-maskin-prosessorer som nå begynner å bli vanlig i nyere RISC og superskalare prosessorer***

Register-merking, Tomasulo's metode, eksempel, (1/3)

FLB = Floating Point Buffers, FLR = Float.p. Registers



Dynamisk intern foroversending v.hj.a. register-merking. Endrer mottaker-adresse dynamisk i maskinvare.

Eksempel:

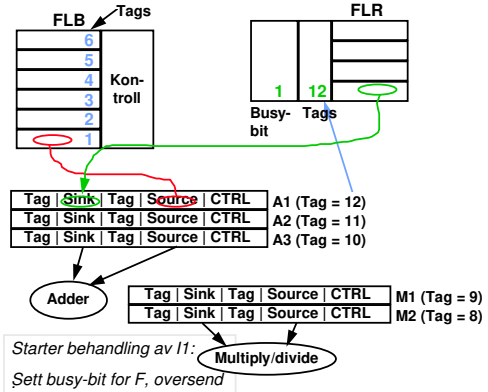
I1: F := F + B1

I2: F := F * B2

B1 = FLB nr. 1, F = et FLR register

Register-merking, Tomasulo's metode, eksempel, (2/3)

FLB = Floating Point Buffers, FLR = Float.p. Registers



Starter behandling av I1:

Sett busy-bit for F, oversend innholdet av F og B1 til reservasjons-stasjon A1. Send merkelapp = 12 til F som betyr at F "er i A1".

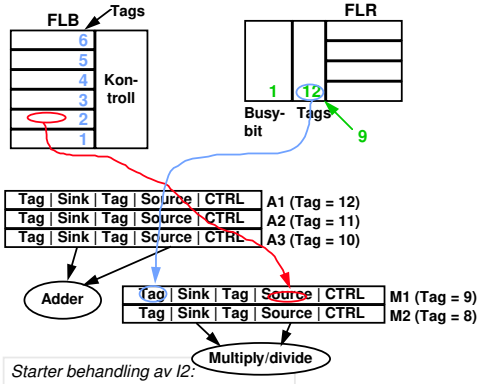
Eksempel:

I1: F := F + B1

I2: F := F * B2

Register-merking, Tomasulo's metode, eksempel, (3/3)

FLB = Floating Point Buffers, FLR = Float.p. Registers



Starter behandling av I2:

Oppdager at F er opptatt og sender F's tag til reservasjonsstasjon M1 for å si at F skal komme fra resultatet av A1. Endrer deretter F's tag til 9 (betyr "skal til" M1). Når I1 er ferdig ser den sentrale kontrollen av F's tag at verdien kan sendes direkte til M1

Eksempel:

I1: F := F + B1

I2: F := F * B2

Effekten av program-forgreninger (Eng. "branching")

- **Effekten av forgreninger på ytelsen**
  - simuleringer av en gitt arkitektur med sporings-data (Eng. trace-data)
  - analytiske modeller
- **Sentrale begreper**
  - utført hopp (Eng. branch taken)
  - hopp-adresse (Eng. branch target)
  - hopp-forsinkelse (Eng. delay slot)
    - » antall pipeline-sykler fra utført hopp til instruksjonen lagret på hopp-adresse utføres
  - Fig. 6.18
  - Effektiv pipeline-ytelse H_{eff}

n = # instruksjoner utført

k = # trinn i pipeline

τ = trinn - forsinkelse

b = hopp - forsinkelse

p = sannsynlighet for betinget hoppinstrukjon

q = sannsynlighet for at hopp utføres

$$T_{eff} = k\tau + (n-1)\tau + pqnb\ \tau$$

$$H_{eff} = \frac{n}{T_{eff}} = \frac{nf}{k+n-1+pqnb} \qquad (6.12)$$

Mer om effektiv pipeline-ytelse

k, τ, n, p, q, b som tidligere

$$H_{eff} = \frac{n}{T_{eff}} = \frac{nf}{k+n-1+pqnb} \qquad (6.12)$$

Antar: $(b = k-1) \wedge (n \to \infty)$

$$H^*_{eff} = \frac{f}{pq(k-1)+1} \qquad (6.13)$$

H^*_{eff} er strengeste øvre grense for pipeline ytelse

$$D = \frac{f - H^*_{eff}}{f} = 1 - \frac{1}{pq(k-1)+1} = \frac{pq(k-1)}{pq(k-1)+1} \qquad (6.14)$$

D er faktor for ytelsesdegradering

Forgrenings-prediksjon

- **Hvis vi kan forutsi resultatet av en betinget hopp-instruksjon kan dette utnyttes til å forbedre ytelsen**
  - hent instruksjoner fra den "kontroll-flyt-tråden" som har størst sannsynlighet
- **Statisk forgrenings-strategi**
  - benytter typen av hopp-instruksjon som indikasjon på hvilket utfall som er mest sannsynlig
    - » bestemt ved konstruksjon av prosessoren
- **Dynamisk forgrenings-strategi**
  - benytter en begrenset mengde historie-informasjon for hver betinget hopp-instruksjon for å kunne forutsi (gjette) "hvilken vei den hopper"
    - » krever ekstra maskinvare og plass
    - » "Branch Target Buffer (BTB)
    - » Fig. 6.19

Utsatt hopp (Eng. delayed branch)

- **Ved hopp til en instruksjons-sekvens som ikke "allerede er på vei igjennom instruksjons-pipeline" vil påbegynte instruksjoner måtte kastes og vi får et hull i instruksjons-strømmen som ikke blir utnyttet**
  - forgrenings-straff (Eng. branch penalty)
  - slike instruksjoner må kunne stoppes/reverseres hvis de ikke skal utføres
    - » en enkel teknikk er å stoppe videre behandling av instruksjoner i pipeline inntil utfallet av betinget hopp-instruksjon er kjent
    - » evt. legg inn NOP
- **Fornuftig bruk av instruksjoner som slippes rett inn etter en hopp-instruksjon kan gi bedret ytelse**
  - det er ønskelig at slike instruksjoner kan utføres ferdig
    - » instruksjoner som er uavhengige av hoppet kan legges inn mellom hopp-instruksjonen og hopp-adressen
    - » Eks. 6.8, fig. 6.21
  - dette kalles utsatt hopp (Eng.delayed branch)
    - » MIPS R4000, Motorola 88110

Konstruksjon av aritmetisk pipeline

Kap. 6.4, side 297->

- **Kap. 6.4.1. utgår, (antas kjent)**
  - fast punkt og flyt-tall
  - tallrepresentasjon og standarder
  - flyt-talls operasjoner
- **Shift og Add**
  - er tilstrekkelig for å realisere alle aktuelle aritmetiske operasjoner
    - » multiplikasjon kan utføres som en rekke addisjoner
    - » divisjon kan utføres som en rekke multiplikasjoner, såkalt konvergens divisjon, side 304. Detaljer er her ikke pensum.
- **To sentrale addisjonsmetoder**
  - CPA, Carry Propagate Adder
    - » vanlig addisjon med menteforplanting
    - » begrenset parallellitet
    - » "2-til-1 reduksjon"
  - CSA, Carry Save Adder
    - » bitvis addisjon av 3 tall til bit-sum og mente-vektor
    - » meget god parallellitet
    - » "3-til-2 reduksjon"
  - Fig. 6.22

Statisk aritmetisk pipeline

- **Generelt**
  - aritmetisk pipeline for **skalare** beregninger
    - » enhet for fast-punkt og heltalls aritmetikk
    - » enhet for flyt-talls aritmetikk
    - » i dag ofte begge to på samme brikke
  - aritmetisk pipeline for **vektor** beregninger
    - » ofte som opsjonelt påbygg
  - forskjell på disse to hovedtypene er i hovedsak:
    - » register-struktur
    - » kontroll-mekanismer
- **Eksempel**
  - Pipeline for multiplikasjon basert på tre av CSA-enheter
    - » Wallace-tre, Fig. 6.23
  - Pipeline flyttallsenhet i MC68040, Fig. 6.24
  - IBM 360/Model 91, flyttallsenhet, Fig. 6.25

Flerfunksjons aritmetisk pipeline

- **Husk at statisk aritmetisk pipeline betyr skreddersydd for en funksjon**
- **Flerfunksjons pipeline kan deles opp i**
  - **Statiske**, som kan utføre ulike funksjoner, men ikke samtidig
  - **Dynamiske**, flere funksjoner samtidig
- **Eks. Texas Instruments Advanced Scientific Computer (TI-ASC)**
  - bare en funksjon av gangen
  - Fig. 6.26-27.
- **De fleste superdatamaskiner benytter flere aritmetiske pipeline der hver er dedikert for en enkelt funksjon**
  - dette gir mye enklere kontroll-logikk
  - *Påstand fra Lasse (og andre):* ***Trend:*** *Maskinvare-mengden (# transistorer) får stadig mindre betydning som begrensende faktor ved konstruksjon av datamaskiner i forhold til problemet med å* ***mestre kompleksiteten*** *i konstruksjonen*
    - » verktøy spiller en sentral rolle

Superskalare prosessorer og superpipelining

Side 308->

- **To teknikker for å forbedre ytelse i (pipeline-baserte) prosessorer**
  1) Bredde-parallellitet, superskalar
  2) "Lengde parallellitet", super-pipelining
- **En superskalar prosessor av grad *m* kan starte opp *m* instruksjoner i en prosessor-sykel**
  - krever tilsvarende parallellitet på instruksjonsnivå (Eng. ILP)
  - Eks. Fig. 6.28(a)
    - » flere parallelle pipeline
    - » dupliserte el. ulike funksjonelle enheter innenfor samme trinn kan være delt mellom disse
    - » vanskeligere fordeling av instruksjoner
  - Program-eksempel med tilhørende avhengighetsgraf, Fig. 6.28(b)
  - Pipeline "holding" (Eng. stalling)
    - » Navn på det fenomen at instruksjoner må vente i pipeline p.g.a. data- og ressurs-avhengigheter
    - » Et større problem i superskalare prosessorer
    - » Eks., Fig. 6.29

Superskalare prosessorer. Betydningen av rekkefølge på oppstarting og avslutting av instruksjoner

- **Oppstarting av instruksjoner**
  - i program-rekkefølge
  - utenfor program-rekkefølge
- **Avslutting av instruksjoner**
  - i program-rekkefølge
  - utenfor program-rekkefølge
    - » må passe på utdata-avhengighet og antiavhengighet
- **I praksis har vi tre muligheter**
  - i rekkefølge **-->**i rekkefølge
    - » det enkleste å realisere
    - » ofte nødvendig i multiprosessorer
    - » sjelden brukt i vanlige prosessorer
  - i rekkefølge **-->** utenfor rekkefølge
  - utenfor rekkefølge **-->** utenfor rekkefølge
- **Motivasjonen er ytelses-forbedring**
  - vanlig både i skalare og superskalare prosessorer
  - beregning av optimal omstokking (permutasjon) er et NP-komplett problem
    - » men, håndterbart for et mindre # instruksjoner
- **Eks. Fig. 6.30**
- **Eks. Motorola 88110**

Superskalar ytelse

T(1,1) = tidsforbruk i vanlig skalar prosessor

T(m,1) = tidsforbruk i superskalar prosessor med parallellitetsgrad *m*

S(m,1) = hastighetsøkning

$$T(1,1) = k + N - 1 \qquad (6.26)$$

$$T(m,1) = k + \frac{N-m}{m} \qquad (6.27)$$

$$S(m,1) = \frac{T(1,1)}{T(m,1)} = \frac{N+k-1}{(N/m)+k-1} = \frac{m(N+k-1)}{N+m(k-1)} \qquad (6.28)$$

Superpipelining

- **I en superpipeline-prosessor av grad *n* er pipeline syklus tid *1/n* av basis-sykel**
  - Fig. 6.31 (a) , *n* = 3
    - » hvert trinn er delt opp i 3 under-trinn

$$T(1,n) = \frac{kn}{n} + \frac{1}{n}(N-1) \qquad (6.29)'$$

$$S(1,n) = \frac{T(1,1)}{T(1,n)} = \frac{k+N-1}{k+(N-1)/n} = \frac{n(k+N-1)}{nk+N-1} \qquad (6.30)$$

Superskalar prosessor med superpipelining

- **De to teknikkene kan kombineres**
  - superskalar og superpipeline prosessor av grad *(m,n)*
    - » *m* parallelle pipeline
    - » *n* ganger flere trinn i pipeline
  - Eks., DEC Alpha, MIPS R4000
  - Superskalar
    - » romlig (Eng. spatial) parallellitet
    - » egnet for CMOS, duplisering av HW er lett
  - Superpipelining
    - » temporal parallellitet
    - » krever økt trinn-inndeling av logikken og kortere klokkesykler, **vanskeligere**

$$T(m,n) = \frac{kn}{n} + \frac{N-m}{mn} \qquad (6.31)'$$

$$S(m,n) = \frac{T(1,1)}{T(m,n)} = \frac{k+N-1}{k+(N-m)/(mn)} = \frac{mn(k+N-1)}{mnk+N-m} \qquad (6.32)$$

Viktige parametre for pipeline-baserte prosessorer

- **Syklus-tid** **Begreper fra side 159.**
  - varighet av det lengste trinnet
    - » kalles basis-sykel for vanlig pipeline prosessor
- **Oppstartingsrate**
  - # instruksjoner startet opp pr. klokke-sykel
- **Oppstartingsforsinkelse**
  - nødvendig tidsforsinkelse mellom oppstarting av to instruksjoner (i samme pipeline)
- **ILP, parallellitet på instruksjonsnivå**
  - som er nødvendig for å kunne utnytte pipeline 100%
    - » antall instruksjoner som kan startes opp innenfor en basis-sykel
- **Tabell 6.1 side 310**