Kapittel 7

Minnehierarki og pipeline

Innhold

7.1 Læringsmål 175
7.2 Hvorfor det må være et hierarki 175
7.3 Lokalitet 180
7.4 Cachen 183
7.5 Pipelinen 187
7.6 Når pipelinen stopper 192
7.7 Oppsummering 195
7.8 Videre lesning 197
7.9 Refleksjon 198
7.10 Selv-test 198

Du har skrevet en løkke som går gjennom mange tall.

Her er to av dem. Begge legger sammen nøyaktig de samme tallene — 64 millioner av dem, satt opp i et rutenett med 8000 rader og 8000 kolonner. Den første går radvis gjennom rutenettet. Den andre går kolonnevis.

```
# radvis
for i in range(8000):
    summer.append(rutenett[i, :].sum())

# kolonnevis
for j in range(8000):
    summer.append(rutenett[:, j].sum())
```

To ting om skrivemåten, hvis den er ny for deg. Uttrykket `rutenett[i, :]` betyr «hele rad nummer `i`», og kolonet står for «alt i den retningen» — så `rutenett[:, j]` er hele kolonne nummer `j`. Og `.sum()` legger sammen tallene den får. Dette er ikke vanlig Python, men et tillegg som heter NumPy, og som brukes overalt der man regner på mange tall samtidig. Du trenger ikke kunne det for å lese videre.

Samme maskin. Samme tall. Samme antall addisjoner.

Den radvise bruker 13 millisekunder. Den kolonnevise bruker 441.

Den ene er over tretti ganger tregere enn den andre.

Stopp litt ved hvor urimelig det er. Du har ingenting å forklare det med ennå. Begge løkkene gjør like mange addisjoner. Begge leser hvert tall nøyaktig én gang. Antall operasjoner er likt ned til siste addisjon. Likevel bruker den ene over tretti ganger så lang tid.

Forskjellen ligger ikke i hva maskinen regner ut. Den ligger i hvilken *rekkefølge* maskinen henter tallene.

Tankevekker

Dette er ikke en kuriositet. Det er en av de største faktorene du som programmerer kan påvirke uten å endre algoritmen i det hele tatt.
To programmer med samme algoritme, samme antall operasjoner og samme språk kan skille en faktor tretti — og hele forskjellen sitter i hvordan de beveger seg gjennom minnet.

Vi kommer dessuten hit med en gjeld.

Kapittel 6 endte med et ubehagelig tall. Du regnet ut at en addisjon kan ta rundt 960 pikosekunder. Så kom påstanden: å *hente* et tall fra minnet tar omtrent hundre ganger lengre tid enn å regne med det.

Det er et merkelig regnskap. Maskinen bruker mesteparten av tiden sin på å vente på data, og bare en liten flik på å gjøre noe med dem. Hvis det stemmer, burde en prosessor nesten aldri få gjort noe.

Likevel får den gjort svært mye. Dette kapitlet handler om hvordan.

Svaret er to triks. Det første er å legge det du bruker ofte så nær som mulig, slik at turen blir kort. Det andre er å begynne på neste oppgave før den forrige er ferdig. Begge triksene er gamle, begge er enkle å forstå, og begge er grunnen til at maskinen foran deg er brukbar.

Du trenger ingenting nytt for å lese dette kapitlet. Vi kommer til å regne en del, men all regningen er gange, dele og veie et gjennomsnitt, og hvert svar står utskrevet rett ved siden av. Vi innfører ingen ny skrivemåte og ingen formler med symboler i — det vi skal telle, teller vi i ord. Det eneste du bør ha med deg, er klokkeperioden fra kapittel 6 og hent–utfør-sløyfa fra kapittel 3.

Til gjengjeld skal jeg si fra om én ting: dette kapitlet har to halvdeler som ser ut til å handle om helt forskjellige ting. Den første handler om minne, den andre om instruksjoner. De møtes på siste side, og da viser de seg å være samme idé. Hvis andre halvdel føles som et nytt kapittel, er det fordi den nesten er det — hold ut til oppsummeringen.

Hva du sitter igjen med

Innen siste side skal du kunne svare på tre spørsmål.
Hvorfor er den ene løkka over tretti ganger tregere enn den andre, når de gjør akkurat like mye arbeid?
Hvorfor har maskinen din et halvt dusin forskjellige slags minne i stedet for ett stort og raskt?
Og hvorfor går ikke en prosessor med fire pipelinetrinn fire ganger så fort som en uten?
Det siste spørsmålet er stilt i forelesningen også, og svaret er mer interessant enn et ja eller nei.

Dette er forresten vendepunktet i boka.

Kapittel 1 tegnet opp lagene, og vi har gått nedover siden: tall, instruksjoner, arkitektur, porter, transistor. I forrige kapittel sto vi på bunnen. Fra nå av går vi oppover igjen, og vi tar med oss alt vi plukket opp på veien ned.

7.1 Læringsmål

Etter å ha arbeidet med dette kapitlet skal du kunne:

1. **Forklare hvorfor minnet må være delt i flere lag.** Raskt, stort og billig er tre ønsker, og du får bare to av dem samtidig.
2. **Rangere lagene etter fart, pris og størrelse, og sette tall på dem.** Fra registeret til harddisken er det en faktor på flere millioner.
3. **Regne ut effektiv aksesstid.** Og se det overraskende: at de få bommene koster mer enn alle treffene til sammen.
4. **Skille tidslokalitet fra stedslokalitet, og peke dem ut i kode du selv har skrevet.** Dette er grunnen til at hele hierarkiet virker i det hele tatt.
5. **Forklare hva en cachelinje er, og hvorfor maskinen henter mer enn du ba om.** Her ligger svaret på de trettito gangene.
6. **Regne treffrate for en gitt rekke med tilganger.** Og si hva som skjer med den hvis linjestørrelsen endres.
7. **Gjøre rede for de to skrivestrategiene.** Og hva man vinner og taper på hver av dem.
8. **Navngi de fire trinnene i pipelinen, og telle sykluser for et gitt antall instruksjoner.** Både med og uten pipeline.
9. **Skille latens fra gjennomstrømning.** Dette er det punktet flest tar feil av, og det er verdt å bruke tid på.
10. **Kjenne igjen de tre hasardtypene, og finne hasarden i en instruksjonssekvens.** Og si hva den koster i tapte sykluser.

7.2 Hvorfor det må være et hierarki

7.2.1 Regnestykket fra forrige kapittel, gjort ferdig

Kapittel 6 ga deg verktøyet til å regne ut hvor lang tid en krets bruker. Finn den lengste veien gjennom kretsen, tell portene, gang med portforsinkelsen. En port tar rundt 5 pikosekunder, en fulladder rundt 15, og og en mentekjede på 64 bit kommer opp i rundt 960 pikosekunder hvis mentene må vandre hele veien. Med trikset fra kapittel 6, der mentene regnes ut på forskudd, kommer addisjonen godt innenfor én klokkeperiode.

Så ble spørsmålet stående: hvor lang tid tar det å hente tallene som skal legges sammen?

La oss regne med en prosessor på 3 gigahertz. Det er en vanlig fart, og det er omtrent der klokka stoppet i kapittel 6. Tre milliarder sykluser i sekundet betyr at én syklus varer en tredjedels nanosekund, altså rundt 0,3 nanosekunder.

Nå kan vi måle alt i sykluser i stedet for i nanosekunder. Det er mye lettere å kjenne igjen.

To ord dukker opp i tabellen som du ikke har møtt ennå. En **cache** er et lite og raskt minne som holder en kopi av det maskinen bruker oftest, slik at turen til det store og trege minnet kan droppes. Uttales «kæsj», og kommer av det franske ordet for gjemmested. Det finnes flere av dem, og de nummereres L1, L2 og L3 etter hvor nær kjernen de ligger — L1 er nærmest og minst. Vi bygger dem ordentlig i avsnitt 7.4; her trenger du bare å vite at de sitter mellom prosessoren og RAM.

Og enhetene: en *nanosekund* er en milliarddels sekund, en *mikrosekund* er en milliondels, og en *millisekund* er en tusendels. De forkortes ns, μs og ms, og det er tusen ganger mellom hver.

Hvor verdien ligger	Ventetid	Antall sykluser
I et register	under 0,3 ns	under 1
I L1-cachen	ca. 1,3 ns	ca. 4
I L2-cachen	ca. 4 ns	ca. 12
I L3-cachen	ca. 13 ns	ca. 40
I RAM	ca. 80 ns	ca. 240
På en SSD	ca. 50 μs	ca. 150 000
På en harddisk	ca. 8 ms	ca. 24 000 000

Tabell 7.1 Hvor lenge prosessoren venter, målt i klokkesykluser ved 3 GHz. Tallene varierer mellom maskiner, men størrelsesordenene holder.

Les den nederste raden en gang til.

Hvis verdien ligger på en harddisk, står prosessoren og venter i rundt 24 millioner klokkesykluser. I den tiden kunne den ha utført titalls millioner addisjoner. Den gjør ingen av dem. Den venter.

Selv RAM er ille. 240 sykluser er 240 tapte muligheter til å regne ut noe.

Tallene oversatt til noe du kjenner

Sykluser er abstrakt. La oss si at én klokkesyklus var ett sekund, slik at du kunne følge med.
Da ville det å hente en verdi fra et register tatt under ett sekund. Fra L1-cachen: fire sekunder.
Fra L2: tolv sekunder. Fra L3: førti sekunder.
Fra RAM: fire minutter.
Fra en SSD: nesten to døgn.
Fra en harddisk: ni måneder.
En prosessor som leser fra harddisken, venter altså — i sin egen tidsregning — omtrent like lenge som det tar å gjennomføre et skoleår.

7.2.2 Avstand koster tid

Hvorfor er det slik? Hvorfor kan ikke alt minne være like raskt som et register?

To av grunnene kommer fra fysikken i forrige kapittel.

Den første er avstand. Et signal beveger seg ikke øyeblikkelig gjennom en ledning. I kapittel 6 så du hvorfor: ledningen har kapasitans, og kapasitans må fylles. Jo lenger ledningen er, jo mer er det å fylle, og jo lenger tid tar det. Et register ligger noen mikrometer fra regneenheten. En L3-cache ligger på den andre siden av brikka. En RAM-brikke ligger noen centimeter unna, på den andre siden av en kontakt og et stykke kretskort.

Den andre grunnen er at *stort* i seg selv betyr *tregt*. Et minne med mange adresser trenger en større dekoder for å finne fram til riktig celle — du bygde en slik i kapittel 5 — og en større dekoder har flere porter på den kritiske veien. Det trenger også lengre ledninger internt, rett og slett fordi brikka er større.

Hvor mye av de 80 nanosekundene som er selve turen

Avstandsargumentet er sterkest inne på brikka, der det forklarer hele spriket mellom register, L1 og L3.
For RAM holder det ikke hele veien. Signalet bruker under ett nanosekund på turen fram og tilbake til en minnebrikke noen centimeter unna. De øvrige åtti går med inne i selve brikka og i køen av forespørsler foran deg — DRAM-celler er trege å lese, og det er mange som vil fram samtidig.
Konklusjonen står: lenger unna er tregere. Men turen i seg selv er sjelden hele forklaringen.

Her ligger kjernen i hele kapitlet. Et minne som er stort, er langt unna i begge betydninger: dataene ligger fysisk mer spredt, og det tar flere porter å finne fram.

Et stort og raskt minne er derfor ikke bare dyrt. Det er delvis selvmotsigende.

Hvorfor et register er så mye raskere enn en cache

Et register har ingen adresse å slå opp. Instruksjonen peker rett på det, og ledningen går direkte dit.
En cache må lete. Den må sjekke om verdien i det hele tatt er der, og den må finne hvilken av tusenvis av plasser den ligger på. Den letingen tar tid, og den er grunnen til at selv den raskeste cachen er flere ganger tregere enn et register.
Hvordan letingen faktisk foregår, står i appendiks F. Det er ikke pensum, men det er ikke vanskelig.

7.2.3 Raskt, stort og billig — velg to

Så kommer penger, som er en helt annen slags grunn, men like reell.

De to lagene er bygget av ulik elektronikk. Cachene bruker **SRAM** — *statisk* RAM — der hvert bit holdes av seks transistorer som står og holder på verdien så lenge det er strøm. RAM bruker **DRAM** — *dynamisk* RAM — der hvert bit er én transistor og en liten kondensator, altså et av de badekarene fra kapittel 6. Karet lekker, så innholdet må friskes opp tusenvis av ganger i sekundet. Det er det «dynamisk» viser til, og det er grunnen til at DRAM er tregere.

Til gjengjeld er DRAM-cellen bitte liten. Seks transistorer mot én er bare begynnelsen: SRAM-cellen trenger også ledninger mellom de seks, og plass rundt dem, mens kondensatoren i en DRAM-celle kan graves ned i dybden. Regnet i areal per bit blir forskjellen omtrent ti til tjue ganger.

Prisen er grunnen til at ingen lager en maskin med 32 gigabyte SRAM. Brikka ville vært hundre ganger større enn en vanlig prosessorbrikke, og kostet mer enn resten av maskinen mange ganger om.

Så vi sitter med tre ønsker som ikke kan oppfylles samtidig:

- Vi vil at minnet skal være **raskt**, så prosessoren slipper å vente.
- Vi vil at det skal være **stort**, så hele programmet får plass.
- Vi vil at det skal være **billig**, så maskinen er til å betale.

Velg to. Du får ikke den tredje.

7.2.4 Løsningen: ha flere av dem

Her kommer trikset, og det er verdt å finne på selv før vi gir det et navn.

Tenk deg at du skriver en oppgave. Du har tre steder å ha papirer.

Rett foran deg på pulten ligger arket du skriver på og de to notatene du ser på hele tiden. Det er lite plass der — kanskje fem ark — men du når alt uten å strekke deg.

Litt lenger unna, i sekken eller på hylla, ligger pensumboka og notatene fra forrige uke. Det tar noen sekunder å finne fram noe derfra.

Og så finnes biblioteket. Der er det millioner av sider, men du må gå dit.

Ingen ville foreslått å løse dette ved å slepe hele biblioteket inn på pulten. Du ville ikke funnet noe. Pulten er rask *fordi* den er liten.

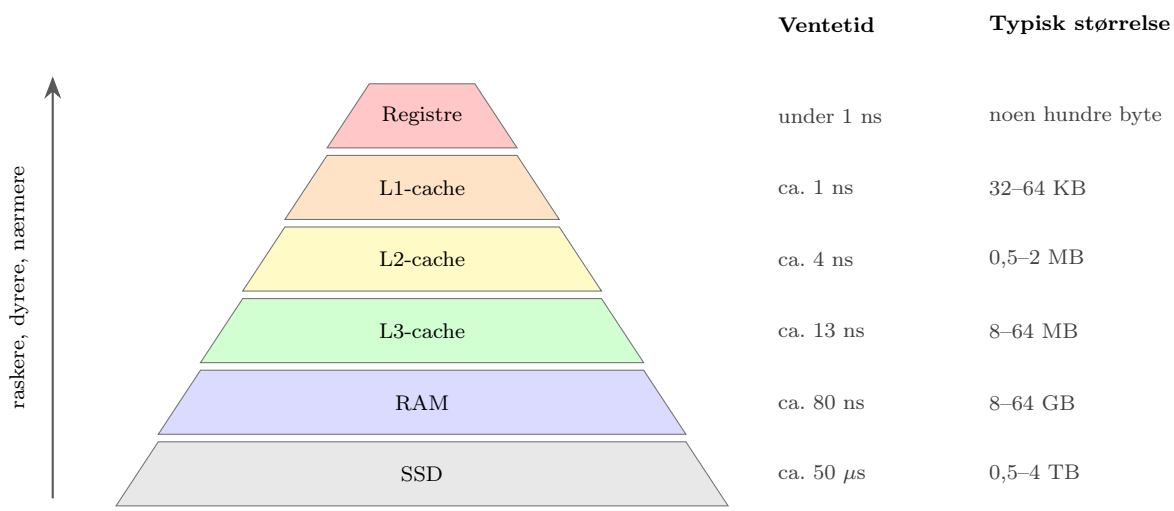
I stedet gjør du noe annet, helt uten å tenke over det: du flytter ting oppover etter hvert som du trenger dem. Boka du henter fra biblioteket, blir liggende på hylla. Siden du leser akkurat nå, ligger på pulten. Når du er ferdig med den, havner den på hylla igjen, og noe annet tar plassen.

Det er hele ideen. Du har ikke ett lagringssted. Du har flere, i lag, og du flytter ting oppover når du bruker dem.

En datamaskin gjør nøyaktig det samme. Lagene heter noe annet, og de er bygget av forskjellige slags elektronikk, men prinsippet er det du nettopp fant på.

7.2.5 Pyramiden

Nå kan vi sette navn på lagene. Figur 7.1 er den samme figuren du møter i forelesningen, med avrundede tall fra tabell 7.1 på side 176.



Figur 7.1 Minnehierarkiet. Hvert lag nedover er tregere, billigere per byte og mye større. Tallene er avrundet fra tabell 7.1 og varierer mellom maskiner. Harddisken ligger under SSD-en igjen, utenfor figuren.

Tre ting er verdt å merke seg med en gang.

Hvert lag holder en kopi av noe i laget under. Cachen inneholder deler av det som står i RAM. RAM inneholder deler av det som ligger på disken. Ingenting flyttes bort fra laget under — det kopieres oppover.

Skillet går mellom RAM og disk, ikke der du kanskje tror. Alt fra registeret til RAM kalles **primærminne**. Det er flyktig: slår du av strømmen, er innholdet borte. SSD-en og harddisken er

sekundærminne, og de husker uten strøm. Det er derfor et dokument du ikke har lagret, forsvinner når maskinen mister strømmen — det lå bare i primærminnet.

Registrene er de samme du møtte i kapittel 4. Akkumulatoren, programtelleren, instruksjonsregisteret. De er toppen av dette hierarkiet, og de er de eneste lagene instruksjonene dine nevner ved navn.

Et lag vi hopper over

Det finnes ett lag til, mellom RAM og disken, som vi ikke tar her: **virtuelt minne**. Det er mekanismen som lar hvert program tro at det har hele minnet for seg selv, og som flytter data ut på disken når RAM-en blir full.
Virtuelt minne hører like mye til operativsystemet som til maskinvaren, og det tas i kapittel 8. Vil du lese om det nå, står det grundig i Englander kapittel 19.7.
Du mister ingenting ved å vente. Alt i dette kapitlet gjelder uansett.

7.2.6 Hva bommene koster

Nå til det tallet som overrasker folk.

Først to ord til. Finner prosessoren det den leter etter i cachen, kaller vi det et **treff**. Finner den det ikke, og må hente det fra et lavere lag, kaller vi det en **bom**. På engelsk *hit* og *miss*, og de engelske ordene brukes like ofte som de norske. Andelen tilganger som er treff, heter **treffraten**.

Si at prosessoren treffer i cachen 90 prosent av gangene. Det høres bra ut. Ni av ti ganger går turen kjapt.

La oss regne, med runde tall for å gjøre det lett: et treff tar 1 nanosekund, og en bom betyr en tur til RAM som tar 100. Det er tabellens 1,3 og 80 nanosekunder, avrundet — svaret blir ikke annerledes av det, bare lettere å regne i hodet.

Steg 1. Ni av ti tilganger tar 1 nanosekund:

$$0{,}9 \times 1 \text{ ns} = 0{,}9 \text{ ns}$$

Steg 2. Én av ti tar 100 nanosekunder:

$$0{,}1 \times 100 \text{ ns} = 10 \text{ ns}$$

Steg 3. Til sammen:

$$0{,}9 + 10 = 10{,}9 \text{ ns}$$

Se på de to leddene. Treffene — ni av ti tilganger — bidrar med 0,9 nanosekunder. Bommene — én av ti — bidrar med 10.

Den ene bommen koster elleve ganger mer enn alle de ni treffene til sammen.

Dette tallet, 10,9 nanosekunder, heter **effektiv aksesstid**. Det er den gjennomsnittlige tiden en tilgang tar når du regner både treff og bom med.

La oss gjøre det en gang til, med en bedre cache. Nå treffer vi 99 prosent av gangene:

$$0{,}99 \times 1 + 0{,}01 \times 100 = 0{,}99 + 1 = 1{,}99 \text{ ns}$$

Treffraten gikk fra 90 til 99 prosent. Det høres ut som en liten forbedring — ni prosentpoeng. Men den effektive aksesstiden falt fra 10,9 til 1,99 nanosekunder. Minnetilgangene ble over fem ganger raskere, og siden maskinen bruker mesteparten av tiden sin på å vente på nettopp dem, merkes det godt på programmet.

Derfor jages de siste prosentene

Når noen bruker år på å flytte treffraten fra 97 til 98 prosent, er det ikke pirk.
Regn selv. Ved 97 prosent: $0{,}97 \times 1 + 0{,}03 \times 100 = 0{,}97 + 3 = 3{,}97$ ns. Ved 98 prosent: $0{,}98 \times 1 + 0{,}02 \times 100 = 0{,}98 + 2 = 2{,}98$ ns. Ett prosentpoeng, og en fjerdedel av tiden er borte.
Det er bommene som koster. Alltid. Og det er derfor neste seksjon — om hvorfor treffene i det hele tatt skjer — er den viktigste i kapitlet.

Regn på det selv

En maskin har to nivåer. L1 tar 1 ns, RAM tar 100 ns.
Et program treffer i L1 i 80 prosent av tilgangene. Hva er den effektive aksesstiden?
Og: hvor mye fortere går programmet hvis treffraten løftes til 95 prosent?
Svarene står i fasiten, men prøv først. Det tar et halvt minutt.

7.3 Lokalitet

7.3.1 Hvorfor hierarkiet virker i det hele tatt

Hele pyramiden hviler på én antakelse: at det du trenger, som regel allerede ligger høyt oppe.

Det er ikke opplagt at det skulle stemme. Hvis et program hoppet tilfeldig rundt i minnet, ville cachen vært ubrukelig. Den ville aldri hatt det du spurte om, og hver eneste tilgang ville endt i RAM. Da hadde vi betalt for cachen uten å få noe igjen.

Så hvorfor stemmer det likevel?

Svaret finner du i kode du har skrevet selv. Her er noe som ligner på alt du har laget i Python:

```
sum = 0
for tall in maalinger:
    sum = sum + tall
```

Tre linjer. Se på hva maskinen faktisk gjør med minnet mens dette kjører.

Den rører `sum` igjen og igjen. En gang per runde i løkka. Hvis listen har tusen tall, leses og skrives den samme variabelen tusen ganger.

Den går gjennom `maalinger` i rekkefølge. Først tall nummer null, så nummer én, så nummer to. Den hopper ikke. Den går fra element til nabo.

Og den kjører de samme tre linjene om igjen. Selve instruksjonene — ikke bare dataene — hentes fra samme sted tusen ganger.

Det er ikke tilfeldig at det ble sånn. Det er slik programmer ser ut. Løkker gjentar. Tellere telles opp. Lister gås gjennom. Funksjoner kalles om og om igjen.

7.3.2 To mønstre, og to navn

De to tingene vi nettopp observerte, har hvert sitt navn.

Brukte du noe nettopp, kommer du sannsynligvis til å bruke det igjen snart. Variabelen `sum` er et typeeksempel. Dette heter **tidslokalitet** — lokalitet i tid.

Brukte du noe, kommer du sannsynligvis til å bruke naboen. Gjennomgangen av `maalinger` er et typeeksempel. Dette heter **stedslokalitet** — lokalitet i rom.

Huskeregelen er to ord: *nylig* og *naboen*.

	Tidslokalitet	**Stedslokalitet**
Kort sagt	Det samme igjen	Naboen til det samme
I koden over	`sum` leses hver runde	`maalinger` gås gjennom i rekkefølge
Andre eksempler	En teller. Kroppen i en løkke. En funksjon som kalles ofte.	En tekststreng. Feltene i et objekt. Instruksjonene i en funksjon.
Cachen utnytter det ved å	beholde linja etter bruk	hente inn naboene samtidig

Tabell 7.2 De to formene for lokalitet. Nesten alle programmer har begge.

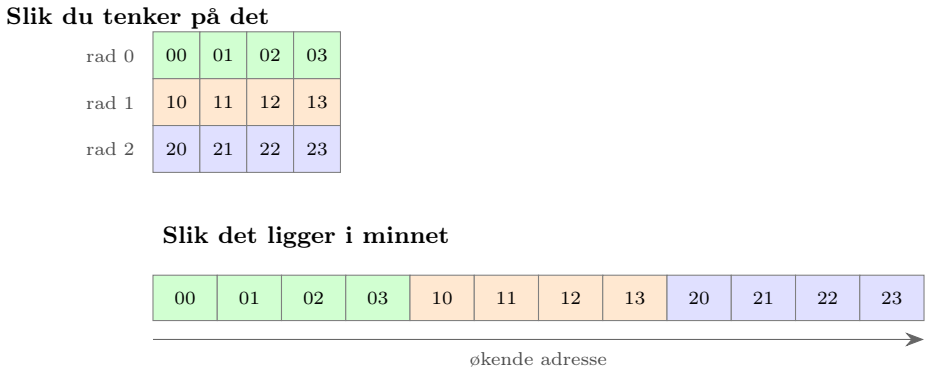
Det er verdt å merke seg at instruksjonene har lokalitet helt på egen hånd. En løkke på ti linjer som kjøres en million ganger, er ti linjer med ekstrem tidslokalitet. Det er derfor cachen virker like godt på kode som på data — og, som vi kommer til i avsnitt 7.4, derfor maskinen ofte har en egen cache for hver av dem.

7.3.3 Tilbake til de trettito gangene

Nå har vi det vi trenger for å forklare løkkene fra første side.

Et rutenett med tall ligger ikke i et rutenett inne i maskinen. Minnet er en lang rekke adresser — du så det i kapittel 3, der postkassene sto på rad. Rutenettet må brettes ut til én lang linje.

Måten det brettes på, er radvis: først hele rad 0, så hele rad 1, så hele rad 2. Figur 7.2 viser det.



Figur 7.2 Et rutenett brettes ut radvis i minnet. Naboene i en rad ligger ved siden av hverandre. Naboene i en kolonne gjør det ikke.

Dermed blir de to løkkene helt forskjellige reiser gjennom minnet.

Den radvise løkka går bortover den lange linja, ett steg av gangen. Hvert tall den trenger, ligger rett ved siden av det forrige. Perfekt stedslokalitet.

Den kolonnevise løkka tar ett tall, hopper over en hel rad, tar neste, hopper over en hel rad igjen. Hvert tall tar fire byte — det er en vanlig størrelse for et desimaltall, slik du så i kapittel 2. I et rutenett med 8000 kolonner blir hvert hopp dermed på 32 000 byte. Den lander aldri i nærheten av der den var.

Samme antall tall. Samme antall addisjoner. Helt ulik oppførsel i minnet.

7.3.4 Målingen, og det som er enda mer interessant

Du trenger ikke tro på dette. Det lar seg måle, og det tar noen sekunder.

Tabell 7.3 viser samme forsøk med tre forskjellige størrelser på rutenettet.

Rader	Størrelse	Radvis	Kolonnevis	Forhold
500	1 MB	0,33 ms	0,45 ms	1,3×
2000	16 MB	1,6 ms	3,8 ms	2,3×
8000	256 MB	13 ms	441 ms	33,6×

Tabell 7.3 Radvis mot kolonnevis summering, beste av fem kjøringer på en vanlig maskin. Legg merke til hva som skjer med forholdet når rutenettet vokser.

Her er det interessante, og det er ikke de trettitre gangene.

Ved 1 megabyte er det nesten ingen forskjell. Rutenettet får god plass i cachen. Når alt ligger der uansett, spiller rekkefølgen nesten ingen rolle — du henter fra samme hylle enten du gjør det ryddig eller rotete.

Ved 16 megabyte begynner det å svi, fordi rutenettet nå er på grensen av eller over det den største cachen i maskinen rommer. Ved 256 megabyte, der rutenettet er mange ganger for stort uansett, slår det ut for fullt.

Nøyaktig hvor knekkpunktet ligger, avhenger av hvor stor cachen i den enkelte maskinen er. Det er selve mønsteret som er poenget, ikke de eksakte tallene.

Effekten vokser altså med hvor dårlig dataene får plass. Det er den beste bekreftelsen man kan få på at det faktisk er cachen vi ser, og ikke noe annet.

Dette forsøket er lett å gjøre feil

Prøver du å gjenskape målingen med en naiv nøstet løkke, slik:

```
for i in range(5000):
    for j in range(5000):
        x = rutenett[i, j]
```

... så ser du nesten ingenting. Målt forhold: rundt 1,1 ganger.
Grunnen er verdt å tenke over. Python er et tolket språk. For hver eneste indeksering gjør tolkeren en mengde arbeid — slår opp navn, sjekker typer, lager et nytt objekt for tallet. Det arbeidet tar så mye lengre tid enn selve minnetilgangen at cachebommen drukner fullstendig.
Du måler altså tolkeren, ikke maskinvaren.
Det er i seg selv et poeng fra kapittel 1: et abstraksjonslag kan skjule laget under så godt at du ikke klarer å se det. Målingen i tabell 7.3 virker fordi den gjør få, store kall ned i et bibliotek skrevet i C, slik at det er maskinvaren som dominerer.

7.3.5 Dette er håndtaket du faktisk har

Det er verdt å stoppe her et øyeblikk.

Nesten alt annet i dette kapitlet er noe maskinen gjør for deg, uten å spørre. Du kan ikke bestemme hva som ligger i cachen. Du kan ikke slå av pipelinen. Du har ingen instruksjon som sier «legg dette nær».

Lokalitet er unntaket. Rekkefølgen du går gjennom data i, bestemmer du helt selv — og den rekkefølgen kan koste eller spare en faktor tretti, uten at du endrer algoritmen med et komma.

Det er derfor denne seksjonen er den du har mest praktisk nytte av. Ikke bare i dette faget.

7.4 Cachen

7.4.1 Maskinen henter mer enn du ba om

Si at du skal ned i kjelleren for å hente én skrutrekker.

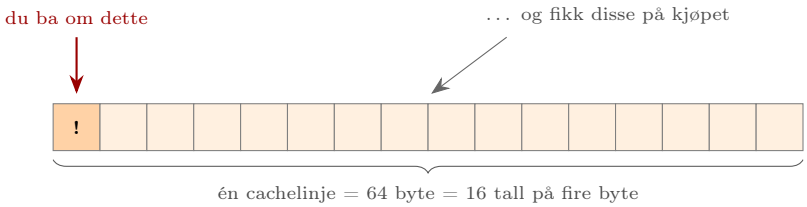
Du går ned, finner den, og går opp. Neste gang trenger du en tang. Ned igjen, opp igjen. Så en skrujern til. Ned, opp.

Etter tredje tur gjør du det alle gjør: du tar med hele kassa.

Poenget er at turen er dyr, ikke bærejobben. Når du først er nede, koster det nesten ingenting å ta med naboene til det du kom for.

En cache gjør nøyaktig dette. Ber prosessoren om fire byte fra RAM, kommer det ikke fire byte tilbake. Det kommer typisk 64.

Den klumpen heter en **cachelinje**, og den er den minste enheten cachen håndterer. 64 byte er den vanligste størrelsen i dag. Det er plass til seksten tall på fire byte, eller åtte på åtte byte.



Figur 7.3 Cachelinja. Prosessoren ba om fire byte, men hele linja på 64 byte hentes inn. Det er grunnen til at stedslokalitet lønner seg.

Og med det har vi svaret på løkkene.

Den radvise løkka ber om tall nummer 0. Cachen henter tall 0 til 15 i samme slengen. De neste femten spørsmålene besvares uten å gå til RAM. Én bom gir femten gratis treff.

Den kolonnevise løkka ber om tall nummer 0. Cachen henter tall 0 til 15 — men de femten naboene ligger i samme rad, og løkka skal til neste rad, 32 000 byte unna. Ingen av dem rekker å bli brukt før de for lengst er kastet ut igjen for å gi plass til andre linjer. Neste spørsmål er en ny bom. Og det neste. Hele veien.

Den ene løkka bommer én gang per seksten tall. Den andre bommer hver gang. Der er faktoren.

Hvorfor det målte forholdet ble enda større enn seksten

Regnestykket over peker mot en faktor på rundt seksten, og vi målte over tretti.
Forskjellen kommer av at maskinvaren gjør mer enn å hente én linje. Den legger merke til at du går jevnt framover, og begynner å hente de neste linjene *før* du spør om dem. Den radvise løkka får derfor en ekstra gevinst som den kolonnevise ikke får.
Størrelsesordenen er poenget, ikke det eksakte tallet. Og du skal ikke pugge noen av tallene — du skal kunne forklare hvorfor det ene er mye større enn det andre.

7.4.2 Treff og bom

Treff og bom innførte vi raskt i avsnitt 7.2: treff betyr at verdien lå i cachen, bom at den ikke gjorde det. Nå kan vi si hva en bom faktisk utløser.

Ved en bom skjer det tre ting, i denne rekkefølgen:

1. Forespørselen sendes videre til neste nivå. L1 spør L2, L2 spør L3, L3 spør RAM.
2. Hele cachelinja som verdien ligger i, hentes opp.
3. Linja legges i cachen, og noe annet må vike for å gi plass.

Prosessoren står og venter mens dette pågår. Det er derfor en bom koster så mye: ikke fordi selve letingen er dyr, men fordi turen nedover i hierarkiet er det.

Og treffraten — andelen tilganger som er treff — er nettopp den vi regnet med da vi fant effektiv aksesstid.

Regn en treffrate

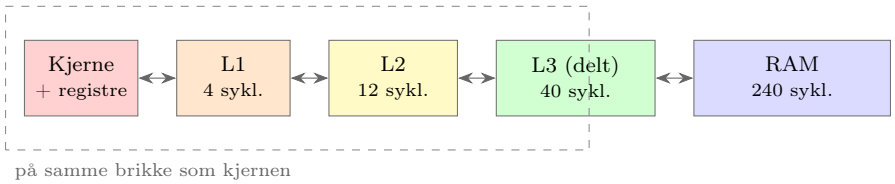
En cachelinje rommer fire tall. Et program leser en liste med tolv tall, i rekkefølge, én gang. Cachen er tom når programmet starter, og lista begynner akkurat der en linje begynner.
Hvor mange bommer blir det? Hvor mange treff? Hva er treffraten?
Og så det som er verdt å tenke på: hva skjer med svaret hvis cachelinja hadde rommet åtte tall i stedet?

7.4.3 Hvorfor det er flere nivåer

Du kan ha lurt på hvorfor det er tre cacher og ikke én.

Grunnen er at avveiningen fra avsnitt 7.2 ikke forsvinner bare fordi vi har bygget en cache. Den gjentar seg.

Vil du at cachen skal svare på fire sykluser, må den være liten — noen titalls kilobyte. Vil du at den skal romme et par megabyte, kan den ikke svare på fire sykluser. Begge er nyttige. Så vi lager begge.



Figur 7.4 Veien en forespørsel tar. Hvert nivå spør det neste bare hvis det bommer selv.

Fordelingen i en moderne flerkjerneprosessor er som regel slik:

- **L1** er privat for hver kjerne, og den er som regel *delt i to*: en for instruksjoner og en for data. Det er fordi kjernen trenger å hente en instruksjon og en dataverdi i samme syklus, og det er langt enklere å bygge to cacher som svarer på hver sin forespørsel enn én som svarer på begge. Vi kommer tilbake til nettopp dette i avsnitt 7.6.
- **L2** er også privat for hver kjerne, og større.
- **L3** deles som regel av alle kjernene, og er stor.

En påstand du kan møte, som ikke stemmer lenger

Eldre framstillinger sier at L1 ligger på prosessorbrikka mens L2 og L3 ligger utenfor den.
Det var riktig på 1990-tallet. I dag ligger alle cachenivåene på samme brikke som kjernene. Du kan se dem på et bilde av en moderne prosessor: den delte L3-cachen legger ofte beslag på et større areal enn kjernene til sammen.
Grunnen til at alt flyttet inn, er den samme som i avsnitt 7.2. Avstand koster tid, og en tur ut av brikka og inn igjen er dyr.

7.4.4 Hva skjer når du skriver

Alt vi har sagt til nå, handler om å lese. Skriving er vanskeligere, og det er verdt å se hvorfor.

Du endrer en variabel. Verdien ligger i cachen. Nå er det to kopier av den — én i cachen og én i RAM — og de er ikke like lenger.

Det finnes to strategier, og begge brukes.

Gjennomskriving (*write-through*) betyr at maskinen skriver begge steder med en gang. Enkelt, og de to kopiene er alltid like. Prisen er at hver eneste skriving må helt ned til RAM, selv om du skal endre den samme variabelen hundre ganger på rad. En skrivebuffer demper det, fordi prosessoren slipper å stå og vente på at turen fullføres, men turene blir like mange.

Tilbakeskriving (*write-back*) betyr at maskinen bare skriver i cachen, og merker linja som endret. Først når linja må vike for noe annet, skrives den ned til RAM. Endrer du samme variabel hundre ganger, blir det én tur i stedet for hundre.

Tilbakeskriving er raskere, og den er det vanlige valget i dag. Prisen er at det finnes et tidsrom der cachen og RAM er uenige om hva verdien er. Maskinen må holde orden på hvilke linjer som er endret, og den må sørge for at ingen andre leser den utdaterte kopien.

En analogi: tenk deg at du fører et regnskap. Gjennomskriving er å gå bort til den store permen og føre inn hver eneste post med en gang. Tilbakeskriving er å notere på en blokk ved siden av deg, og føre en linje inn i permen først når du trenger plassen på blokka til noe annet. Det andre er raskere. Det er også det som gjør at permen kan være feil akkurat nå.

En påstand du kan møte, som er upresis

Du kan støte på setningen «det tar lengre tid å skrive til minnet enn å lese fra det», ofte begrunnet med at skriving krever mer strøm.
For DRAM stemmer det dårlig. Lese- og skrivelatens er omtrent like, og fordi skrivinger kan legges i en buffer og utføres senere, kan de til og med *virke* raskere sett fra prosessoren.
Der påstanden stemmer, er for flash-minnet i en SSD. Der må en celle slettes før den kan skrives på nytt, og slettingen går blokkvis. Det gjør skriving reelt mye dyrere enn lesing.
Det som faktisk gjør skriving komplisert i en cache, er ikke tiden. Det er at det oppstår to kopier som kan være uenige.

7.4.5 Når flere kjerner har hver sin kopi

Og da er vi ved et problem vi bare skal navngi her.

Maskinen din har flere **kjerner**. En kjerne er en komplett prosessor — med egne registre, egen ALU og egen pipeline — og en moderne brikke har flere av dem ved siden av hverandre, slik at den kan kjøre flere programmer samtidig på ordentlig. Kapittel 8 tar dem ordentlig. Her trengs bare ett faktum: hver kjerne har sin egen L1-cache. Hva skjer hvis to kjerner har kopi av den samme cachelinja, og den ene endrer sin?

Da har maskinen to svar på samme spørsmål, og minst ett av dem er feil.

Problemet heter **cache-koherens**, og det løses i maskinvaren: cachene holder øye med hverandre, og når én endrer en linje, får de andre beskjed om at deres kopi ikke lenger gjelder.

Hvordan det foregår, og hva det koster, hører til kapittel 8, der flerkjerne tas ordentlig. Det du skal ta med deg herfra, er at problemet finnes og hvorfor det oppstår: fordi hver kjerne har sin egen kopi.

7.4.6 Cachen er usynlig — med ett unntak

Til slutt noe som overrasker mange.

Ingen instruksjon forteller cachen hva den skal gjøre. Programmet ber om en adresse og får en verdi tilbake, og det samme programmet gir det samme svaret enten maskinen har cache eller ikke. Det eneste som endrer seg, er hvor lang tid det tar.

Cachen er altså et abstraksjonslag av akkurat den typen kapittel 1 handlet om: noe som er skjult, som virker, og som du aldri trenger å tenke på for å få riktig svar.

Med ett forbehold

Ekte instruksjonssett har noen få instruksjoner som sier fra til cachen på forhånd — «denne adressen kommer jeg snart til å trenge, hent den nå». De kalles *prefetch*, og pensumboka nevner dem i avsnitt 8.3.
De bryter likevel ikke med det vi har sagt, for de er *hint*. De endrer ikke hva programmet regner ut, bare hvor fort det går. Fjerner du dem, gir programmet nøyaktig samme svar.

Helt til du trenger det. For det ene unntaket er det vi så i avsnitt 7.3: rekkefølgen du går gjennom data i. Den bestemmer du, og den er hele forskjellen mellom en bom og femten gratis treff.

Vil du se hvordan cachen faktisk leter? Appendiks F

Vi har sagt at cachen «finner» eller «ikke finner» en linje, uten å si hvordan. Det er en reell forenkling, og den dekker over noe som er både enkelt og elegant.
Appendiks F tar det:

- hvordan en adresse deles i tre deler, og hva hver del brukes til
- hva en direkte avbildet cache er, og hvorfor den kan bomme på data den faktisk har
- hva assosiativitet betyr, og hvorfor det hjelper
- hvilken linje som må vike når det ikke er plass

Det er **ikke pensum**. Men det er der du finner svaret hvis du har lurt på hvordan maskinen kan lete gjennom tusenvis av linjer på fire klokkesykluser.

7.5 Pipelinen

7.5.1 Tre som tar oppvasken

Nå skifter vi tema. Ta en pause på en halv side hvis du trenger det — resten av kapitlet handler om noe annet, og det er greit å begynne med blanke ark.

Her er situasjonen. Tre personer skal ta oppvasken for en hel kveld. Hver tallerken må gjennom tre steg: den skal vaskes, skylles og tørkes. Hvert steg tar ett minutt.

Første forsøk. Amina gjør alt alene. Hun vasker en tallerken, skyller den, tørker den, og legger den fra seg. Tre minutter per tallerken. Tretti tallerkener tar nitti minutter. Jonas og Henrik ser på.

Det er åpenbart dumt, og de ser det med en gang.

Andre forsøk. De stiller seg opp ved siden av hverandre. Amina vasker, Jonas skyller, Henrik tørker. Hver tallerken går fra hånd til hånd.

Følg med på hva som skjer det første minuttet. Amina vasker tallerken 1. Jonas og Henrik står uten noe å gjøre.

Andre minutt: Amina vasker tallerken 2, Jonas skyller tallerken 1. Henrik venter fortsatt.

Tredje minutt: Amina vasker 3, Jonas skyller 2, Henrik tørker 1. Nå jobber alle tre.

Fra og med tredje minutt blir det ferdig én tallerken i minuttet. Tretti tallerkener tar 32 minutter i stedet for 90.

Legg merke til to ting, for de er hele poenget.

Ingen tallerken går fortere gjennom. Den første tallerkenen er ferdig etter tre minutter — nøyaktig like lenge som da Amina gjorde alt selv. Ingen av de tre jobber raskere enn før.

Men tallerkenene kommer ut mye oftere. Fra ett hvert tredje minutt til ett hvert minutt.

Det er nøyaktig det en prosessor gjør, og nå kan vi gi det navnet sitt: en **pipeline**, eller et samlebånd.

7.5.2 De fire trinnene

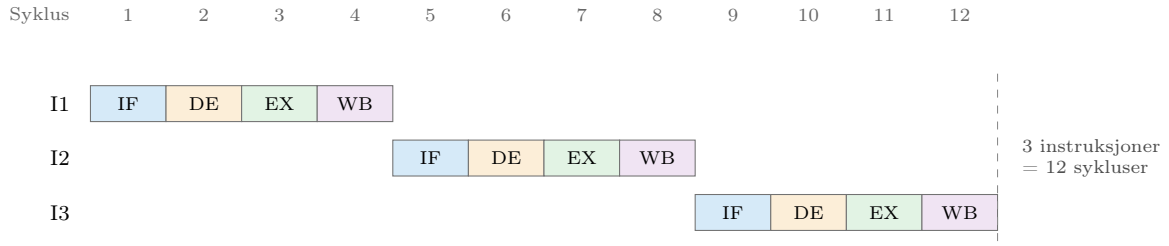
I kapittel 3 og 4 fulgte du en instruksjon gjennom maskinen. Den ble hentet, tydet, utført, og resultatet ble lagret.

Det er de fire stegene, og de er tallerkenens tre med ett til:

Kort	Navn	Hva som skjer
IF	Instruction Fetch	Instruksjonen hentes fra minnet, og programteller-en økes.
DE	Decode	Instruksjonen tydes. Maskinen finner ut hva den skal gjøre og hvilke verdier den trenger.
EX	Execute	Arbeidet gjøres. For en addisjon er det her ALU-en regner.
WB	Write Back	Resultatet skrives tilbake til et register eller til minnet.

Tabell 7.4 De fire pipelinetrinnene. Forkortelsene er engelske og beholdes slik på norsk. Merk at mange bøker skriver ID for dekodetrinnet der vi skriver DE.

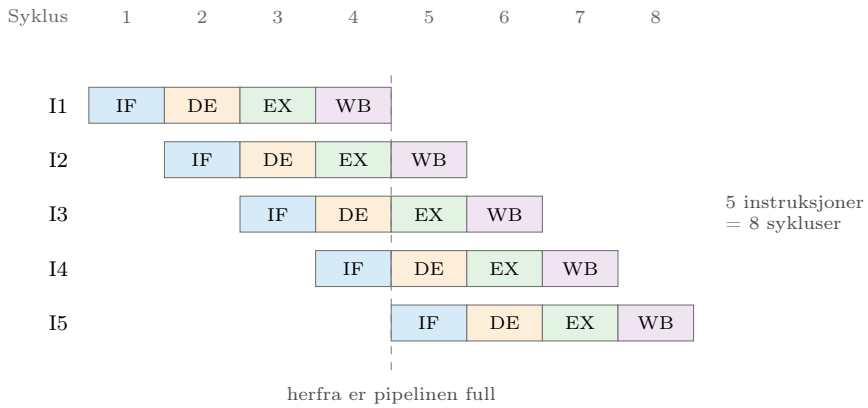
Uten pipeline gjør maskinen én instruksjon helt ferdig før den begynner på neste. Figur 7.5 viser det.



Figur 7.5 Uten pipeline. Hvert trinn har maskinvare som står ubrukt tre firedeler av tiden.

Se på hvor mye som står stille. Mens instruksjon 1 er i EX-trinnet, gjør hentemaskineriet ingenting. Og dekoderen gjør ingenting. Og tilbakeskrivingen gjør ingenting. Tre firedeler av maskinen venter, hele tiden.

Med pipeline starter neste instruksjon så snart trinnet foran er ledig.



Figur 7.6 Med pipeline. Fra og med syklus 4 er alle fire trinnene i arbeid samtidig, og det blir ferdig én instruksjon i hver syklus.

Les figur 7.6 loddrett i stedet for vannrett. I syklus 5 skjer alt dette samtidig: instruksjon 2 skrives tilbake, instruksjon 3 utføres, instruksjon 4 tydes, og instruksjon 5 hentes. Fire instruksjoner er inne i maskinen på én gang.

7.5.3 Regnestykket

Nå kan vi telle sykluser. Vi sammenligner med en maskin som er delt i de samme fire trinnene, men som gjør ett trinn av gangen og ikke begynner på en ny instruksjon før den forrige er helt ferdig. Klokka går like fort i begge.

Uten pipeline bruker hver instruksjon fire sykluser, og de kommer etter hverandre. Antall sykluser er altså fire for hver instruksjon.

Med pipeline må pipelinen først fylles. Det tar tre sykluser — fire trinn, minus det ene som er i arbeid med en gang. Deretter blir det ferdig én instruksjon i hver syklus. Antall sykluser er altså tre, pluss én for hver instruksjon.

Sjekk mot figur 7.6: fem instruksjoner gir tre pluss fem, altså åtte sykluser. Det stemmer.

Instruksjoner	Uten pipeline	Med pipeline	Gevinst
1	4	4	1,0×
4	16	7	2,3×
10	40	13	3,1×
100	400	103	3,9×
1000	4000	1003	3,99×

Tabell 7.5 Gevinsten ved fire pipelinetrinn. Den nærmer seg fire, men kommer aldri helt dit.

Og der har du svaret på spørsmålet forelesningen stiller: nei, fire trinn gir ikke fire ganger så høy hastighet.

Grunnen er de tre syklusene det tar å fylle pipelinen. De er en fast kostnad. Kjører du bare fire instruksjoner, er den kostnaden nesten like stor som arbeidet. Kjører du tusen, forsvinner den i mengden.

For én enkelt instruksjon er gevinsten null. Den bruker fire sykluser uansett.

Latens og gjennomstrømning: det viktigste skillet i kapitlet

Dette er punktet flest tar feil av, så det er verdt å si det tydelig.
Latens er hvor lang tid *én* instruksjon bruker fra start til slutt. Med pipeline: fire sykluser. Uten pipeline: fire sykluser. **Pipelinen gjorde ingen instruksjon raskere.** Ikke én.
Gjennomstrømning er hvor mange instruksjoner som blir ferdige per syklus. Uten pipeline: 0,25. Med pipeline: nesten 1. Her er hele gevinsten.
Tilbake til oppvasken: hver tallerken brukte tre minutter både før og etter. Det som endret seg, var hvor ofte en ferdig tallerken kom ut.
Hvis noen spør om pipelining gjør prosessoren raskere, er det riktige svaret et spørsmål: raskere for hvem? For én instruksjon, nei. For programmet som helhet, nesten fire ganger.

7.5.4 Alle trinn må ta like lang tid

Det er ett krav vi har sneket inn uten å si det.

Gå tilbake til oppvasken. Hva skjer hvis skyllingen tar ett minutt, men vaskingen tar tre?

Da hoper det seg opp. Jonas står ledig i to av tre minutter og venter på Amina. Tallerkenene kommer ut hvert tredje minutt, ikke hvert minutt. Hele gevinsten er borte, og det hjelper ikke at skyllingen er lynrask.

Samlebåndet går i takt med det tregeste trinnet. Alltid.

I en prosessor er dette klokka fra kapittel 6. Klokkeperioden må være lang nok til at *alle* trinnene rekker å bli ferdige. Det tregeste trinnet bestemmer, og de raske står og venter på det.

Det samme, sett fra kapittel 6

Dette er verdt å lese sakte, for det knytter de to siste kapitlene sammen. Merk at vi nå bytter sammenligningsgrunnlag: her stiller vi pipelinen opp mot en maskin som gjør *hele* instruksjonen i én eneste, lang klokkeperiode.
Kapittel 6 sa: klokkeperioden må romme den kritiske veien gjennom kretsen. Jo lengre vei, jo tregere klokke.
I en slik maskin er den kritiske veien hele instruksjonen — hent, tyd, regn og lagre, i én omgang. Det er en lang vei, og den tvinger fram en lang klokkeperiode.
Deler du arbeidet i fire trinn og setter et lagringssted mellom hvert, blir den kritiske veien bare *ett trinn* lang. Rundt en firedel. Da kan klokka gå rundt fire ganger så fort.
Og nå det viktige: dette er ikke en gevinst som kommer *i tillegg* til firedoblingen i tabell 7.5 på forrige side. Det er den samme firedoblingen sett fra en annen kant. Enten teller du fire korte sykluser der du før hadde én lang, eller du teller én ferdig instruksjon per syklus der du før brukte fire. Begge veier ender på rundt fire ganger, og du kan ikke gange dem sammen.
Pipelining handler altså ikke bare om å gjøre flere ting samtidig. Det er også måten man forkortet den kritiske veien på — og dermed måten man fikk lov til å skru opp klokkefrekvensen i det hele tatt.

Lagringsstedet mellom to trinn kalles et **pipelineregister**. Det er ikke et register du kan nevne i en instruksjon, slik akkumulatoren i kapittel 4 er. Det er en rad med vipper som holder på resultatet fra ett trinn til neste trinn er klar for det, og programmet vet ikke at det finnes.

Pipelineregistrene er ikke gratis. De koster transistorer, de koster plass, og de koster litt tid i hver eneste syklus. Det er en av grunnene til at man ikke bare deler opp i hundre trinn.

7.5.5 Hvorfor ikke hundre trinn?

Hvis fire trinn er bra, burde vel førti være bedre?

Det ble faktisk prøvd. Rundt årtusenskiftet bygde Intel en prosessorfamilie med rundt tjue trinn, og i siste utgave over tretti. Målet var høy klokkefrekvens, og den fikk de. Men ytelsen fulgte ikke med, og linja ble lagt ned.

Tre ting setter en grense.

Registrene mellom trinnene tar tid selv. Hvert register koster litt i hver syklus. Deler du arbeidet i nok biter, blir det til slutt mer registerarbeid enn nyttig arbeid i hver syklus.

Arbeidet lar seg ikke dele vilkårlig. En addisjon er en addisjon. Du kan dele den i to halvdeler, men ikke i hundre like store biter som hver tar like lang tid.

Og det som er verst: jo dypere pipeline, jo mer koster det når noe går galt. Neste avsnitt handler om det. Jo flere instruksjoner som er inne i maskinen samtidig, jo flere må kastes når det viser seg at de var feil.

Moderne prosessorer ligger som regel på mellom ti og tjue trinn. Det er et kompromiss, ikke et riktig svar.

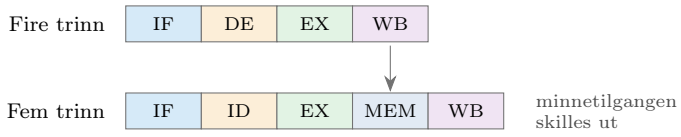
«Fire er den korteste pipelinen som finnes»

Den setningen dukker opp i en del framstillinger, og den stemmer ikke.
Små prosessorer bruker gjerne tre trinn — for eksempel kjernene som sitter i en mikrobølgeovn eller et armbåndsur — og det finnes maskiner med to.
Fire er den korteste modellen *vi* bruker, fordi den er den minste som viser poenget tydelig. Det er noe annet enn den korteste som finnes.

7.5.6 Modellen med fem trinn

Åpner du en annen lærebok, er det ikke sikkert du finner fire trinn. Den vanligste modellen utenfor dette kurset har fem.

Forskjellen er at minnetilgangen får sitt eget trinn:



Figur 7.7 Fire trinn mot fem. Samme idé, bare delt opp litt finere.

Grunnen til å skille ut MEM-trinnet er hele første halvdel av dette kapitlet. Å lese fra eller skrive til minnet er en helt annen slags jobb enn å regne, og den tar helt annen tid. Da er det ryddig å gi den sitt eget trinn.

Denne boka blir værende på fire, fordi det er modellen kurset bruker og fordi den viser det samme. Men nå vet du hvorfor andre tegner fem, og du blir ikke forvirret når du ser det.

Og begge er forenklinger. En ekte prosessor i maskinen din har langt flere trinn enn begge modellene.

7.6 Når pipelinen stopper

7.6.1 Tre ting som kan gå galt

Hele regnestykket i tabell 7.5 på side 189 hviler på en antakelse: at det kommer en jevn strøm av instruksjoner, og at hver av dem kan gå rett gjennom uten å vente.

Den antakelsen brytes hele tiden.

Gå tilbake til oppvasken en siste gang. Hva kan stanse den?

Henrik kan ikke tørke en tallerken Jonas ikke har skylt ferdig. Hvis noen trenger et resultat som ikke er klart ennå, må de vente.

Noen kan rope «stopp, nå tar vi glass i stedet». Da er alt som allerede står i køen av tallerkener, feil. Det må ut, og køen må fylles på nytt.

Og Amina og Jonas kan trenge den samme vasken samtidig. To som skal bruke det samme utstyret, og det finnes bare ett.

Alle tre finnes i en prosessor, og de har et felles navn: **hasard**. En hasard er en situasjon der pipelinen ikke kan gå videre som planlagt.

De tre typene heter datahasard, kontrollhasard og strukturhasard, og de er nøyaktig de tre over.

Avhengighet og hasard er to forskjellige ting

Ordene brukes ofte om hverandre, også i lærebøker, men skillet er verdt å holde på.
En **avhengighet** er en egenskap ved programmet ditt. Instruksjon 3 trenger resultatet fra instruksjon 2. Det er sant uansett hvilken maskin du kjører på.
En **hasard** er en følge av maskinvaren. Avhengigheten blir en hasard bare hvis maskinen ikke rekker å levere verdien i tide.
Skillet har en praktisk konsekvens: en smartere maskin kan fjerne hasarden uten at avhengigheten forsvinner. Kapittel 8 viser et triks som gjør nettopp det for de fleste datahasardene.

7.6.2 Datahasard: å vente på et svar

Dette er den vanligste, og den er lett å se i kode du har skrevet.

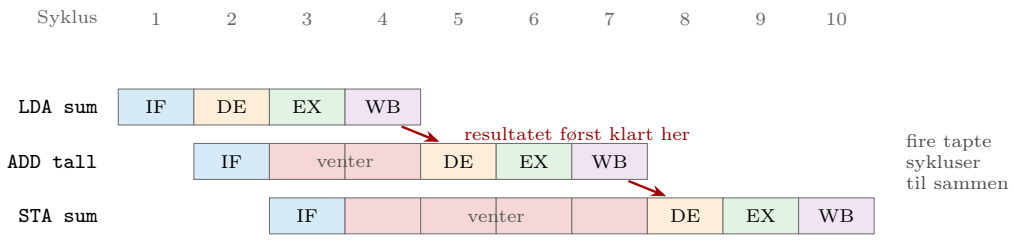
Her er tre instruksjoner i stil med dem fra kapittel 3. De legger et tall til en sum:

```
I1:  LDA sum      # hent sum inn i akkumulatoren
I2:  ADD tall     # legg til
I3:  STA sum      # lagre tilbake
```

Instruksjon 2 skal legge noe til verdien instruksjon 1 nettopp hentet. Den kan ikke gjøre det før verdien finnes.

Se på tidslinja. Instruksjon 1 har verdien klar først i WB-trinnet, i syklus 4. Men instruksjon 2 trenger den allerede i syklus 3, når den skal tydes og hente operandene sine.

Verdien er ikke der ennå. Instruksjon 2 må vente.



Figur 7.8 Datahasard. Hver instruksjon venter på resultatet fra den forrige. De tomme rutene kalles bobler. Tre instruksjoner som skulle tatt seks sykluser, bruker ti.

De tomme rutene har et navn: **bobler**. Det er sykluser der et trinn i pipelinen ikke gjør noe nyttig, fordi det venter. At maskinen settes til å vente, kalles at den **stopper opp** — på engelsk *stall*.

Og det stopper ikke med instruksjon 2. Instruksjon 3 skal lagre det instruksjon 2 nettopp regnet ut, så den får nøyaktig samme problem — og den må i tillegg vente på at instruksjon 2 blir ferdig med å vente. Stansene legger seg oppå hverandre. Hele sekvensen bruker ti sykluser i stedet for seks.

To tapte sykluser per par høres lite ut. Men denne sekvensen — hent, regn, lagre — er noe av det vanligste som finnes i et program. Står den inne i en løkke som kjøres en million ganger, blir det fort millioner av tapte sykluser.

Finn hasarden

Her er fire instruksjoner:

```
I1:  LDA a
I2:  ADD b
I3:  STA c
I4:  LDA d
```

Hvilke par har en avhengighet mellom seg? Og er det noen av instruksjonene som kan gå gjennom uten å vente på noen av de andre?

Det finnes en god løsning på de fleste datahasardene, og den er enklere enn du tror. Den går ut på å sende resultatet direkte fra der det ble regnet ut til der det trengs, i stedet for å vente på at det tar omveien via et register. Vi tar den i kapittel 8.

7.6.3 Kontrollhasard: å ikke vite hvor man skal

Den andre typen kommer av en instruksjon du kjenner godt.

I kapittel 3 møtte du hoppene: `BRA`, `BRZ` og `BRP`. De er det som gjør `if` og `while` mulig. Uten dem hadde et program bare vært en rett strek nedover.

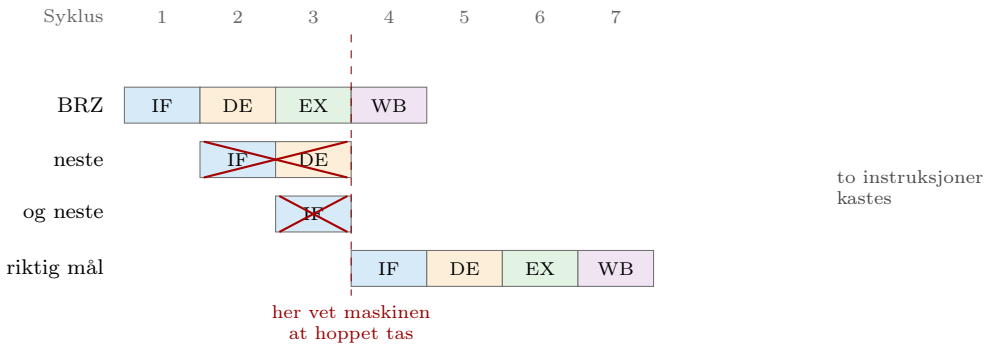
Men nå er de et problem.

Pipelinen henter instruksjoner i rekkefølge. Den henter den neste, og den neste, uten å spørre. Det må den, for den kan ikke stå og vente på å få vite hva som kommer.

Så kommer et betinget hopp. Maskinen vet ikke om hoppet skal tas før den har utført sammenligningen — altså i EX-trinnet. Og innen da har den allerede hentet inn to instruksjoner til.

Hvis hoppet ikke tas, går alt bra. De to er riktige, og de fortsetter.

Hvis hoppet tas, er begge feil. De må kastes ut av pipelinen, og maskinen må begynne på nytt fra riktig sted.



Figur 7.9 Kontrollhasard. Maskinen rekker å hente to instruksjoner før den vet at hoppet skal tas. Begge må kastes, og pipelinen fylles på nytt.

Dette kalles at pipelinen **tømmes**.

Og her ser du hvorfor dype pipelines er risikable. I vår modell kastes to instruksjoner. I en prosessor med tjue trinn kan det være femten eller flere. Hver gang. Ved hvert hopp som gikk den andre veien enn maskinen antok.

Tenk på hvor mange hopp det er i et program. Hver `if`. Hver runde i hver `while`-løkke. Hvert funksjonskall.

Derfor er en test inne i en løkke dyrere enn den ser ut

En `if` inne i en løkke som kjøres mange ganger, er et hopp som tas eller ikke tas mange ganger. Hvis utfallet veksler uregelmessig, taper maskinen noen sykluser hver eneste gang. Hvis det er det samme nesten hver gang, går det mye bedre — av grunner kapittel 8 forklarer.
Det er en av de få gangene der det kan lønne seg å skrive koden om, selv om begge versjonene gjør akkurat det samme.

7.6.4 Strukturhasard: to som vil ha det samme

Den tredje typen er den enkleste å forstå og den sjeldneste i praksis.

Den oppstår når to trinn trenger den samme maskinvaren i samme syklus.

Det klassiske tilfellet er minnet. Husk at IF-trinnet henter en instruksjon fra minnet — det er hele jobben dets. Men noen instruksjoner skal også hente eller lagre *data* i minnet, og i vår firetrinnsmodell skjer det inne i EX-trinnet, siden vi ikke har noe eget minnetrinn. (Det er nøyaktig derfor mange bøker skiller ut et MEM-trinn, slik avsnitt 7.5 viste.)

Se på figur 7.6 på side 189 igjen, syklus 5. Instruksjon 5 er i IF og skal hente en instruksjon. Samtidig er instruksjon 3 i EX, og hvis den er en `LDA`, skal den hente en dataverdi.

To forespørsler, samme minne, i samme syklus.

Løsningen er å ha to minner. Og det er nettopp derfor L1-cachen er delt i to — en for instruksjoner og en for data, slik vi nevnte i avsnitt 7.4.

Her møtes de to halvdelene av kapitlet

Stopp ved dette, for det er ikke en bagatell.
Da vi nevnte at L1 er delt i to, så det ut som en vilkårlig detalj i cachedesignet. Nå viser det seg å være et direkte svar på et problem i pipelinen — et problem vi ikke hadde møtt ennå da vi nevnte det.
Og det går dypere enn som så. En bom i cachen *er* en stans i pipelinen. Når IF-trinnet bommer i instruksjonscachen, står hele pipelinen stille mens verdien hentes — nøyaktig den samme boblen som i figur 7.8 på side 193, bare to hundre ganger lengre.
Oversatt til sekundene fra åpningen av kapitlet: en datahasard koster to sekunder. En bom som må helt til RAM koster fire minutter.

7.6.5 Hva maskinen egentlig venter på

La oss samle opp.

En prosessor har to måter å stå stille på.

Den kan vente på minnet. Det var første halvdel av kapitlet: en bom i cachen, og maskinen står i tolv, førti eller to hundre og førti sykluser mens verdien hentes.

Og den kan vente på seg selv. Det var andre halvdel: en hasard, og maskinen står i noen sykluser fordi en instruksjon ikke kan gå videre.

Begge er det samme sett fra utsiden. Klokka tikker, og ingenting nyttig skjer.

Det er derfor begge trikset i dette kapitlet finnes, og det er derfor hele kapittel 8 handler om å gjøre dem bedre.

7.7 Oppsummering

Kapitlet begynte med to løkker som gjør like mye arbeid, og der den ene er trettito ganger tregere. Nå vet du hvorfor: den ene går fra nabo til nabo gjennom minnet, og den andre hopper 32 000 byte hver gang.

7.7.1 Hva som egentlig var nytt

7.7.2 Den ene ideen, to ganger

Kapitlet har sett ut som to kapitler. Her er grunnen til at det likevel er ett.

Begge halvdelene gjør det samme: de nekter å vente.

Cachen venter ikke på RAM. Den holder en kopi nær, slik at svaret som regel allerede er der. Pipelinen venter ikke på at instruksjonen foran skal bli ferdig. Den setter i gang med den neste med en gang.

Begge virker fordi programmer har et *mønster*. Cachen virker fordi programmer har lokalitet: de bruker det samme om igjen, og de bruker naboer. Pipelinen virker fordi instruksjoner kommer i en jevn strøm, og fordi hver av dem kan deles i trinn som ikke tråkker på hverandre.

Og begge svikter på nøyaktig samme måte: når mønsteret brytes. Et oppslag langt unna gir en bom. Et hopp gir en tømt pipeline. I begge tilfellene står maskinen stille.

Begrep	Kort sagt
Minnehierarki	Flere lag minne, der hvert lag oppover er raskere, dyrere og mindre.
Primærminne	Registre, cache og RAM. Mister innholdet uten strøm.
Sekundærminne	SSD og harddisk. Husker uten strøm.
Effektiv aksesstid	Gjennomsnittstiden når treff og bom regnes sammen. Bommene dominerer.
Tidslokalitet	Brukte du noe nettopp, bruker du det trolig snart igjen.
Stedslokalitet	Brukte du noe, bruker du trolig naboen.
Cache	Lite, raskt minne som holder en kopi av det som brukes mest.
Cachelinje	Klumpen cachen håndterer. Typisk 64 byte. Derfor lønner naboskap seg.
Treff og bom	Om verdien var i cachen eller ikke. Bommen er den dyre.
Treffrate	Andelen tilganger som er treff.
Gjennomskriving	Skriv begge steder med en gang. Enkelt og tregt.
Tilbakeskriving	Skriv bare i cachen, og ned senere. Raskt, men kopiene kan være uenige.
Cache-koherens	Problemet med at hver kjerne har sin egen kopi.
Pipeline	Flere instruksjoner i maskinen samtidig, i hvert sitt trinn.
IF, DE, EX, WB	Hent, tyd, utfør, skriv tilbake.
Latens	Tiden én instruksjon bruker. Pipelining endrer den ikke.
Gjennomstrømning	Hvor mange som blir ferdige per syklus. Her ligger gevinsten.
Avhengighet	At en instruksjon trenger resultatet fra en tidligere. En egenskap ved programmet.
Hasard	En situasjon der pipelinen ikke kan gå videre som planlagt. En følge av maskinvaren.
Datahasard	En instruksjon venter på et resultat som ikke finnes ennå.
Kontrollhasard	Et hopp gjør de allerede hentede instruksjonene ugyldige.
Strukturhasard	To trinn vil ha den samme maskinvaren samtidig.
Boble	En syklus der et trinn står uten nyttig arbeid.
Å stoppe opp	At maskinen settes til å vente. På engelsk *stall*.
Å tømme pipelinen	Å kaste instruksjonene som allerede er hentet, fordi de viste seg å være feil.
Pipelineregister	Lagringsstedet mellom to trinn. Usynlig for programmet.

Tabell 7.6 Begrepene fra dette kapitlet.

Det er verdt å legge merke til en siste ting. Ingen av dem er synlige fra programmet. Ingen av dem endrer hva programmet regner ut. De er maskinvare som later som om ingenting har skjedd — maskinen oppfører seg nøyaktig som den enkle modellen i kapittel 3 sa den skulle.

Den eneste forskjellen er farten. Og det er den eneste måten du kan oppdage dem på.

7.7.3 Videre til neste kapittel

Vi har nå to problemer vi har navngitt og ikke løst.

Datahasardene koster to sykluser hver, og de er overalt. Kontrollhasardene koster like mye i vår modell, men mye mer i en dyp pipeline — og de er også overalt. Vi sa begge ganger at det finnes løsninger, og at de kommer senere.

De kommer i kapittel 8, sammen med det som skjer når man slutter å gjøre én ting av gangen i det hele tatt: flere regneenheter, flere kjerner, og maskinvare som er bygget for helt andre oppgaver enn den vi har sett på.

Det er også kapitlet der vi går helt opp igjen, og ser hele stabelen fra kodelinja til transistoren på én side.

7.8 Videre lesning

Englander, kapittel 8. Her er pensumboka tilbake etter to kapitler der den ikke dekket stoffet. Avsnitt 8.3 om cache er den klart beste delen — den bruker fire og en halv side på cache og har en god gjennomgang av linje, treff, bom og de to skrivestrategiene. Figur 8.9 er verdt å studere.

Avsnitt 8.2 dekker pipelining, men bare på halvannen side, og den navngir ikke hasardtypene slik vi har gjort. Den kaller dem «dependencies». Les den gjerne, men ikke forvent å finne de tre kategoriene der.

En advarsel om tallene i boka. Englander har noen år på baken, og flere av de konkrete tallene er utdaterte. Boka sier at cache gir «speed improvements of 50 percent and more» — i dag er forskjellen titalls ganger, ikke femti prosent. Den sier at SRAM er «two to three times as fast as DRAM», mens den reelle forskjellen i ventetid er nærmere femti ganger. Og den sier at flerkjerneprosessorer har privat L1 og delt L2, mens dagens brikker har privat L1 og L2 med delt L3. Prinsippene står seg. Tallene gjør det ikke.

Patterson & Hennessy, *Computer Organization and Design*. Den kanoniske framstillingen av begge halvdelene av dette kapitlet. Kapittel 5 tar minnehierarkiet grundig, med avbildning, assosiativitet og regneeksempler. Avsnitt 4.5 til 4.8 tar pipelining med fem trinn, hasardene hver for seg, og løsningene vi har utsatt til neste kapittel. Hvis du bare skal lese én ting etter dette kapitlet, les 4.5.

Ulrich Drepper, «What Every Programmer Should Know About Memory». Fritt tilgjengelig på nett, og den beste teksten som finnes om minnehierarkiet sett fra programmererens side. De første tretti sidene dekker alt i dette kapitlet med mer dybde og flere målinger. Resten er for de ivrige.

«Latency numbers every programmer should know». En kort liste som sirkulerer i ulike versjoner, og som setter opp ventetidene fra tabell 7.1 på side 176 ved siden av hverandre. Verdt å ha sett én gang, slik at størrelsesordenene sitter.

Videre i denne boka: appendiks F tar hvordan cachen faktisk finner en linje — adressedeling, direkte avbildning, assosiativitet og erstatningsstrategi. Det er ikke pensum, og det er den naturlige fortsettelsen hvis avsnitt 7.4 vakte lyst på mer.

7.9 Refleksjon

1. Minnehierarkiet virker bare fordi programmer har lokalitet. Finn et program eller en oppgave der lokaliteten er dårlig, og tenk over hva som ville skjedd om cachen ikke fantes i det hele tatt.
2. Cachen er usynlig for programmereren, men avgjør ytelsen. Er det en god eller dårlig abstraksjon? Sammenlign med andre skjulte mekanismer du kjenner.
3. Pipelining gjorde ingen enkelt instruksjon raskere, bare gjennomstrømningen bedre. Finn to eksempler utenfor datateknologien der man har valgt gjennomstrømning framfor hvor lang tid én enhet bruker — og ett der det motsatte valget ble tatt.
4. En dypere pipeline gir høyere klokkefrekvens, men taper mer hver gang et hopp går feil vei. Hvordan ville du avgjort hvor dypt det lønner seg å gå? Hvilke opplysninger ville du trengt?
5. Vi så at én prosent lavere treffrate kan koste en fjerdedel av tiden. Hvor ellers møter du systemer der de siste få prosentene koster mest?

7.10 Selv-test

Spørsmålene står i stigende vanskegrad, og fasiten finnes bakerst i boka.

1. Hvorfor kan vi ikke bare bygge ett stort og raskt minne?
2. Nevn de tre ønskene man må velge to av.
3. Ranger disse etter hastighet, raskest først: RAM, register, L2-cache, SSD, L1-cache.
4. Hva er forskjellen på primærminne og sekundærminne?
5. Hvorfor forsvinner et ulagret dokument når strømmen går?
6. Omtrent hvor mange klokkesykluser venter en prosessor på en verdi som ligger i RAM? Bruk tallet fra tabell 7.1.
7. Gi to grunner til at et stort minne er tregere enn et lite, uavhengig av pris.
8. Hvorfor er et register raskere enn selv den raskeste cachen?
9. Hva betyr det at hvert lag holder en kopi av noe i laget under?
10. Hva er effektiv aksesstid?
11. En cache treffer 90 prosent av gangene. Treff tar 1 ns, bom tar 100 ns. Hva er den effektive aksesstiden?
12. Samme maskin, men treffraten løftes til 98 prosent. Hva blir svaret nå, og hvor mye raskere er det?
13. Forklar med egne ord hvorfor bommene koster mer enn treffene, selv når de er mye sjeldnere.
14. Hva er tidslokalitet? Gi et eksempel fra kode du har skrevet.
15. Hva er stedslokalitet? Gi et eksempel.
16. Hvilken av de to utnytter cachelinja direkte?
17. Har instruksjonene i et program lokalitet, eller gjelder det bare data? Begrunn.
18. Hva er en cachelinje, og hvor stor er den typisk?
19. Hvor mange tall på fire byte får plass i en cachelinje på 64 byte?
20. Hva er et treff, og hva er en bom?

21. Hva skjer, steg for steg, når det blir en bom i L1?
22. En cachelinje rommer åtte tall. Et program leser 24 tall i rekkefølge, én gang. Cachen er tom ved start, og tallene begynner der en linje begynner. Hvor mange bommer blir det, og hva er treffraten?
23. Samme program, men nå rommer linja fire tall. Hva blir treffraten? Forklar hvorfor den endret seg.
24. Hvorfor har maskinen tre nivåer cache i stedet for ett?
25. Hvorfor er L1 ofte delt i to?
26. Hva er forskjellen på gjennomskriving og tilbakeskriving?
27. Hvilken av de to er raskest, og hva er prisen man betaler for den?
28. Hva er cache-koherens, og hvorfor oppstår problemet?
29. Finnes det en instruksjon som legger en verdi i cachen? Hva er den praktiske konsekvensen av svaret?
30. Hvorfor er den kolonnevise løkka fra første side så mye tregere enn den radvise? Svar i tre setninger.
31. Hvorfor ble forskjellen mellom de to løkkene mindre da rutenettet var på 1 megabyte?
32. Hva er en pipeline, forklart uten å nevne en prosessor?
33. Hva står IF, DE, EX og WB for, og hva skjer i hvert trinn?
34. En maskin uten pipeline kjører 50 instruksjoner. Hvor mange sykluser tar det?
35. Samme 50 instruksjoner, men nå med en pipeline på fire trinn. Hvor mange sykluser nå?
36. Hvorfor blir gevinsten aldri fire ganger, uansett hvor mange instruksjoner du kjører?
37. Hva er latens, og hva er gjennomstrømning? Hvilken av dem endrer pipelining?
38. En venn sier at pipelining gjør hver instruksjon raskere. Hva er galt, og hva er riktig?
39. Hvorfor må alle trinnene i en pipeline ta like lang tid?
40. Forklar hvordan pipelining henger sammen med den kritiske veien fra kapittel 6, og hvorfor det er den egentlige grunnen til at klokka kunne gå fortere.
41. Gi tre grunner til at man ikke bare deler pipelinen i hundre trinn.
42. Hvorfor skiller mange lærebøker ut et eget MEM-trinn?
43. Hva er en hasard?
44. Nevn de tre hasardtypene og gi ett eksempel på hver.
45. Hva er forskjellen på en avhengighet og en hasard?
46. Hva er en boble?
47. I sekvensen `LDA x`, `ADD y`, `STA x` — hvor er datahasardene, og hvor mange sykluser koster de til sammen i vår modell?
48. Hvorfor er et betinget hopp et problem for pipelinen, når et vanlig regnestykke ikke er det?
49. Hvorfor koster en kontrollhasard mer i en pipeline med tjue trinn enn i en med fire?
50. Hvordan løser maskinen strukturhasarden mellom instruksjonshenting og datahenting?
51. **Finn feilen.** En student skriver: «Cachen gjør RAM raskere.» Hva er galt med setningen, og hvordan ville du sagt det riktig?
52. **Finn feilen.** En annen skriver: «Pipelining gir fire ganger så høy klokkefrekvens, og i tillegg fire ganger så mange ferdige instruksjoner per syklus. Til sammen seksten ganger.» Hva er galt?

53. **Finn feilen.** En tredje skriver: «En bom i cachen betyr at dataene må hentes fra harddisken.» Hva er galt?

54. Begge halvdelene av dette kapitlet beskriver det samme trikset. Hvilket, og hvordan svikter det i begge tilfellene?