IN1020 — "wanna-be" lærebok

Omid og Yngvar

Høsten 2026

sist oppdatert: 25. oktober 2026

Innhold

Om denne boka 1

1 Hvordan håndtere kompleksitet? Abstraksjonsnivåers betydning 7

2 Digital representasjon 13

3 Instruksjoner 55

4 Arkitektur og ALU 79

5 Porter og logiske funksjoner 105

6 Transistor, strøm og spenning 143

7 Minnehierarki og pipeline 173

8 Moderne maskiner og hele stabelen 201

I Fasit for selv-test 235

F1 Abstraksjonsnivåer 237

F2 Digital representasjon 239

F3 Instruksjoner 245

F4 Arkitektur og ALU 249

F5 Porter og logiske funksjoner 251

F6 Transistor, strøm og spenning 255

F7 Minnehierarki og pipeline **259**

F8 Moderne maskiner og hele stabelen **265**

II Appendiks 273

A Datahistorie **275**

B Forenkling med Karnaugh-diagram **277**

C Kretser som husker **281**

D Inni transistoren **287**

E Synkronisering **303**

F Hvordan cachen finner linja **313**

G Hvorfor telefonen har en egen KI-brikke **317**

Om denne boka

Du sitter sannsynligvis i ditt første semester. Du har kanskje skrevet noen linjer Python. Nå skal du lære hvordan en datamaskin virker, hele veien ned til strømmen i silisiumet og opp igjen. Det er et langt stykke, og denne boka finnes for å gjøre veien kortere.

Hvorfor denne boka finnes

En forelesning går fort. Du rekker å forstå noe mens du sitter der, og så er du på vei hjem og merker at halvparten er borte. Lysarkene hjelper lite, for de er stikkord skrevet av en som allerede kan stoffet.

Pensumboka er grundig. Men den er på engelsk, og den er lagt opp i et helt annet tempo enn det du holder. Nettet har tusen forklaringer av hvordan negative tall lagres, og de bruker tusen forskjellige ord på det.

Vi ville ha én tekst som du kan lese i rekkefølge, på norsk, med de samme ordene fra første til siste kapittel. En tekst der ordet «register» betyr det samme hele veien, og der analogien med postkassene kommer tilbake når den trengs igjen. Det er den teksten du holder i hendene.

Om tittelen

Boka kaller vi en «wanna-be» lærebok, og det ordet står der med vilje.

En ekte lærebok er fagfellevurdert, komplett og gjennomarbeidet over mange år. Denne er skrevet av to personer ved siden av å undervise emnet, i de timene som ble til overs. Noen kapitler har vært gjennom mange runder og tåler et kritisk blikk. Andre er ferske, og et par er fortsatt bare et skjelett.

Så kall den heller et arbeidsdokument. Det dekker stoffet du skal kunne, og det er skrevet av noen som vet nokså presist hvor studenter pleier å stå fast.

Den ene ideen i hele boka

En datamaskin gjør ingenting avansert. Den skiller mellom to tilstander, og den gjør det milliarder av ganger i sekundet. Alt annet — tekst, bilder, lyd, programmer, kunstig intelligens — er lag på lag med avtaler om hva de to tilstandene skal bety.

Kapittel og planet er samme liste

Emnet har en øvingsdel som heter Universet. Der er stoffet delt opp i planeter, og hver planet har noder du løser deg gjennom.

Kapittelnummeret i denne boka er planetnummeret i Universet. Står du fast på planet 2, er det kapittel 2 du skal slå opp i, og du skal aldri måtte lete deg fram.

Kap.	Planet	Spørsmålet kapitlet svarer på
1	Abstraksjon og lagdeling	Hvordan kan noen forstå en maskin med milliarder av deler?
2	Digital representasjon	Hvordan blir tekst, bilder og lyd til to tilstander?
3	Instruksjoner	Hvordan blir en linje Python til noe en maskin adlyder?
4	Arkitektur og ALU	Hva er det inni maskinen som faktisk utfører ordren?
5	Porter og logiske funksjoner	Hvordan kan en krets regne?
6	Transistor, strøm og spenning	Hva er et bit når vi tar på det?
7	Minnehierarki og pipeline	Hvorfor er maskinen rask, og når er den plutselig ikke det?
8	Moderne maskiner og hele stabelen	Hvordan henger alt dette sammen, og hvor går veien videre?

Rekkefølgen er en V. Vi starter på toppen, hos deg og programmet ditt. Derfra går vi nedover: først hvordan alt blir til bit, så instruksjoner, arkitektur, porter og til slutt transistoren. På bunnen er vi nede i fysikken. Så går vi opp igjen gjennom minne og hastighet, til hele stabelen står samlet i siste kapittel.

Bakerst ligger to appendikser, ett om datahistorie og ett om synkronisering. Begge er god lesning og dårlig eksamensstoff.

Et kapittel kan strekke seg over flere forelesninger, og en forelesning kan ta deler av to kapitler. Vi har latt temaene bestemme hvor skillene går, framfor å la timeplanen gjøre det.

Hva som er hva

Boka er hovedteksten. Forelesningen er destillatet: den tar de vanskeligste bitene og bruker tid på dem. Planetene i Universet er stedet der du finner ut om du faktisk kan stoffet. De tre tar det samme fra hver sin kant, og det er meningen.

Slik ser et kapittel ut

Kapitlene som er ferdige, har samme form. Tenk på en strikkeoppskrift. Forkortelsene betyr det samme fra første til siste rad, og da slipper du å lære oppsettet på nytt for hvert plagg.

Først kommer en kort åpning med den bærende ideen. Så kommer **læringsmål**, en punktliste over hva du skal kunne når du er ferdig. Deretter følger selve stoffet, delt i seksjoner.

Seksjonsrekkefølgen inne i kapitlet følger rekkefølgen på submodulene på planeten, altså de små temabolkene nodene er sortert i. Det er den samme navigasjonsgrunnen som ligger bak nummereringen.

Mot slutten kommer fire faste deler. **Oppsummering** samler trådene. **Videre lesning** peker videre for deg som vil mer, og er ikke pensum. **Refleksjon** er spørsmål uten fasit, ment å tenkes på i dusjen. **Selv-test** er spørsmål med fasit, og fasitene ligger samlet i en egen del mot slutten av boka, sammen med stikkordregisteret.

Fagord forklares der de dukker opp første gang, og de står i registeret bakerst.

Slik kan du bruke boka

Les kapitlet før forelesningen hvis du rekker det, for da har du sett begrepene én gang allerede, og forelesningen kan brukes på det du ikke fikk tak i. Rekker du det ikke, les det etterpå. Men les det på et tidspunkt der du kan stoppe opp underveis.

Regn ut eksemplene selv mens du leser. Et regnestykke du leser, glir forbi, mens det samme regnestykket med penn i hånda setter seg. Dette gjelder særlig konverteringene mellom tallsystemer i kapittel 2 på side 13 og sporingen av programmer i lekemaskinen LMC i kapittel 3 på side 55.

Ta selv-testen før du ser på fasiten. Ser du fasiten først, kjenner du svaret igjen og tror du kunne det hele tiden.

Bruk kryssreferansene. Når det står at noe ble innført i et tidligere kapittel, er det fordi du trenger det akkurat der.

Boksene underveis

I de fleste kapitlene er teksten brutt opp av fargede bokser. Fargen og overskriften sier hva slags boks det er, og hvor mye oppmerksomhet den fortjener.

Tankevekker

Oransje boks, merket Tankevekker. Den bærende ideen i den delen av kapitlet du er inne i, skåret ned til noen få setninger. Leser du bare disse boksene, har du i det minste fått med deg hovedlinja.

Note-to-myself

Gul boks, merket Note-to-myself. En analogi, en sidebemerkning, eller et varsel om at vi kommer tilbake til dette senere. Nyttig uten å være tungt.

Tenk om...

Blå boks, merket Tenk om. Et hva-hvis, der vi tar bort en forutsetning og ser hva som skjer. Prøv å svare selv før du leser videre, for det er derfor boksen står der.

Fun Fact

Grønn boks, merket Fun Fact. Historie og kuriosa. Aldri eksamensstoff, men ofte det du husker lengst.

Advarsel :)

Rød boks, merket Advarsel. En misforståelse som rammer mange studenter, hvert eneste år. Hvis du leser én boks ekstra nøye, la det være denne typen.

Under arbeid

Grå boks, merket Under arbeid. Her står det ingenting ennå. Boksen er synlig med vilje, og den blir stående helt til den kan slettes.

Læringsmål og Blooms taksonomi

Læringsmålene i hvert kapittel starter med et verb, i de fleste kapitlene satt i halvfet. «Forklare», «konvertere», «vurdere». Verbet er valgt med omhu, og det sier noe om hvor mye som kreves av deg.

Verbene kommer fra Blooms taksonomi. Benjamin Bloom og kollegene hans la fram rammeverket i 1956, og Lorin Anderson og David Krathwohl ga ut en revidert utgave i 2001. Tanken er at kunnskap har nivåer. Å gjengi en definisjon er noe annet enn å bruke den på et problem du aldri har sett før.

Tabellen under viser nivåene i den reviderte utgaven, med typiske verb på hvert nivå. De fleste av dem møter du igjen i læringsmålene i denne boka.

Nr.	Nivå	Hva det krever av deg	Verb du møter
1	Huske	Hente fram noe slik det står	kjenne til, nevne, definere
2	Forstå	Si det samme med dine egne ord	forklare, gjøre rede for, beskrive
3	Anvende	Bruke kunnskapen på et nytt tilfelle	regne ut, konvertere, oversette, spore, bruke
4	Analysere	Se delene, og hvordan de henger sammen	dele, skille, finne feilen
5	Vurdere	Veie to løsninger mot hverandre og begrunne valget	vurdere, forutsi konsekvensen, velge
6	Skape	Sette sammen noe som ikke fantes	designe, konstruere, bygge

Nivåene er ordnet etter hvor mye de krever, men de glir over i hverandre. Du beveger deg opp og ned mellom dem mens du jobber. Hovedlinja holder likevel: du kommer sjelden langt på nivå 3 uten å ha vært innom nivå 2.

Tabellen er også en målestokk du kan bruke på deg selv. Forskjellen ligner den mellom å kjenne igjen et bykart og å finne fram i byen uten å se på det. Kjenner du igjen ordet toerkomplement, altså måten negative tall lagres på, og vet omtrent hva det handler om, står du på nivå 1. Kan du forklare det til en medstudent uten å slå opp, er du på nivå 2. Kan du regne -19 om til åtte bit og tilbake igjen, er du på nivå 3. Og klarer du å finne feilen i en løkke som ikke stopper, jobber du på nivå 4.

De fleste læringsmålene i emnet ligger mellom nivå 2 og 4. Nodene på en planet følger samme tanke. Nivå 1 spør om du kjenner stoffet, nivå 3 om du kan bruke det på noe nytt.

Note-to-myself

Se på verbet når du leser en oppgavetekst. «Beskriv» varsler at du skal gjengi, mens «begrunn» varsler at du skal argumentere. Verbet forteller deg både hvor mye som kreves, og hvor lenge du bør skrive.

KI-bruk

Vi mener du skal vite hvordan denne boka er laget.

Stoffet er ikke nytt. I perioden fra 2008 til 2025 skrev vi fire kompendier til dette emnet og til emner som ligner. Ingen av dem ble noen gang ferdige. Hvert av dem stoppet et sted: kapitler som manglet, figurer som aldri ble tegnet, to versjoner av samme forklaring.

I tillegg ligger det åtte år med forelesninger bak teksten. Lysark, tavlenotater og spørsmål fra studenter, og etter hvert en ganske god oversikt over hvilke forklaringer som virker og hvilke som faller død til jorden.

Alt dette materialet er nå kvernet sammen, og til den jobben har vi brukt flere ulike KI-modeller. De har slått sammen overlappende tekster, funnet dublettene og strammet inn språket, og de har skrevet utkast til figurer, oppgaver og fasit.

Det modellene ikke har gjort, er å bestemme hva som er pensum eller å gå god for fagligheten. De valgene er våre. Hver påstand i boka står for vår regning, og feilene som står igjen er våre.

Vi skriver dette fordi vi ber deg være åpen om din egen bruk, og da må vi gå foran. Du bør også vite hva slags tekst du leser.

Og din egen bruk

Reglene for hva du har lov til å bruke i obligatoriske innleveringer og på eksamen står på emnesiden, og de går foran alt vi skriver her.

Men under reglene ligger et mer interessant spørsmål: hva tjener du på?

En språkmodell kan gi deg en riktig forklaring av toerkomplement på to sekunder. Da står den riktige forklaringen på skjermen, og veien derfra og inn i hodet ditt er like lang som før. Det er omtrent som med baking: du kan lese hundre oppskrifter på surdeigsbrød og likevel mislykkes grundig med det første brødet, fordi ferdigheten sitter i hendene og ikke i oppskriften.

Å kjenne igjen et svar føles nesten som å kunne det. Forskjellen viser seg på eksamen, og da er det sent å oppdage den.

Noen måter å bruke en modell på som faktisk hjelper: be den stille deg spørsmål i stedet for å svare på dine. Skriv din egen forklaring først, og be den finne hullene i den. Når en forklaring ikke traff, be om en ny vinkling framfor å lese den samme teksten en gang til.

Og en advarsel vi har dekning for, etter arbeidet med denne boka. Modellene tar feil på akkurat dette stoffet. De regner feil i binært, de bommer på bitbredder, og de tar kantene i toerkomplement på frihånd. Vi har rettet slike feil mange ganger underveis, og modellen tar feil med nøyaktig samme selvsikre stemme som når den har rett.

Det er en god grunn til å kunne stoffet selv.

Boka er under arbeid

Du vil møte grå bokser i flere av kapitlene. De betyr at stoffet ennå ikke er skrevet ut, og de bakerste kapitlene har flest av dem.

Vi har valgt å gi deg boka slik den står, framfor å vente til den er komplett. En halvferdig tekst du kan lese nå, er mer verdt enn en ferdig tekst du får etter eksamen.

Boksene står der for vår skyld også. De irriterer oss hver gang vi åpner fila, og det er hele poenget med dem.

Om det er noe som helst du ønsker å dele med oss, ris/ros eller bare brilijere med å ha funnet feil :), kan du veldig gjerne sende oss en e-post.

Kapittel 1

Hvordan håndtere kompleksitet? Abstraksjonsnivåers betydning

Innhold

1.1 Læringsmål 7
1.2 Hva er abstraksjonsnivåer? 8
1.3 Abstraksjon i datamaskiner og maskinvare 8
1.4 Eksempler på abstraksjon i praksis 9
1.5 Konklusjon 9
1.6 Videre lesning 10
1.7 Refleksjon 10
1.8 Selv-test 11

I dagens digitale verden er vi omgitt av komplekse systemer, fra smarttelefoner til store datanettverk. For å kunne forstå, designe og arbeide med disse systemene trenger vi verktøy for å håndtere deres iboende kompleksitet. Abstraksjonsnivåer er et slikt essensielt verktøy. De tillater oss å forenkle komplekse systemer ved å fokusere på relevante detaljer på hvert nivå, organisere kunnskap og systemer i håndterbare lag, muliggjøre samarbeid mellom eksperter på ulike nivåer, og fremme innovasjon ved å tillate forbedringer på ett nivå uten å påvirke andre.

Abstraksjon er spesielt viktig for de som arbeider med design, implementasjon og vedlikehold av datamaskinsystemer og programvare. Abstraksjonsnivåene i maskinvare og programvare gjør det mulig å bygge opp komplekse systemer ved å jobbe med høyere nivå konsepter, selv når de underliggende detaljene blir stadig mer komplekse.

1.1 Læringsmål

Etter å ha lest dette kapittelet skal du:

- Kjenne til de forskjellige abstraksjonsnivåene og kunne beskrive dem, inkludert maskinvare, minnehierarki, maskinkode og programmeringsspråk.
- Forstå den spesifikke rollen og funksjonen til hvert abstraksjonsnivå og hva de bidrar med i det generelle datamaskinsystemet.

- Forstå hvordan abstraksjonsnivåene påvirker design og implementasjon av programvare og maskinvarekomponenter.
- Kunne vurdere hvordan sikkerheten på hvert abstraksjonsnivå påvirker den overordnede sikkerheten til et datamaskinsystem, og forstå hvordan man sikrer systemet mot trusler på forskjellige nivåer.

1.2 Hva er abstraksjonsnivåer?

Abstraksjonsnivåer er lagdelte representasjoner av et system, hvor hvert nivå skjuler kompleksiteten til nivåene under det. En abstraksjon er en forenklet representasjon av noe komplekst, som fokuserer på relevante detaljer og skjuler unødvendig kompleksitet. Abstraksjonsnivåene danner et hierarki hvor hvert nivå bygger på funksjonaliteten til nivået under, og tilbyr et grensesnitt som spesifiserer hvordan andre nivåer kan interagere med det, uten å eksponere implementasjonsdetaljene. Denne informasjonsskjulingen og modulariteten gjør det enklere å endre eller oppdatere deler av systemet uten å påvirke hele systemet.

1.3 Abstraksjon i datamaskiner og maskinvare

I datamaskinarkitektur finner vi abstraksjonsnivåer fra de fysiske komponentene på bunnen, som transistorer og logiske porter, opp gjennom mikrokontrollere, maskinvareabstraksjonslag og operativsystemer, til applikasjonsprogrammer på toppen. En bruker kan kjøre et program uten å forstå hvordan CPU-en fungerer på transistornivå.

Tilsvarende kan en programmerer skrive kode i et høynivåspråk som Python, uten å måtte tenke på hvordan instruksjonene oversettes til maskinkode og utføres av CPU-en. Høynivåspråk abstraherer bort detaljer som minnehåndtering og instruksjonskoding.

Innen maskinvare finner vi abstraksjonsnivåer som:

- Nettverksabstraksjon: Nettverkskort og protokoller som Ethernet og Wi-Fi abstraherer bort detaljene ved dataoverføring, slik at nettverksprogramvare kan kommunisere uten å forholde seg til de fysiske forbindelsene.
- Lagringsabstraksjoner og minnehierarki: Fra fysiske harddisker til logiske volumer, partisjoner og filsystemer. SSD-er og HDD-er skjuler detaljene om hvordan data faktisk lagres.
- Maskinvareabstraksjonslag (HAL): Lar operativsystemer kommunisere med maskinvare gjennom et standardisert grensesnitt, uavhengig av de spesifikke maskinvarekomponentene.
- Mikroarkitektur: Beskriver hvordan instruksjoner fra programvare behandles og utføres av en prosessor, bygget opp av komponenter som ALU, registre, cache og kontrollenheter.
- Halvlederkomponenter: Transistorer kombineres til logiske porter, som igjen danner byggesteinene for mer komplekse kretser som adderere og minneenheter.

Andre eksempler på abstraksjon i dataverdenen inkluderer:

- Skytjenester: Abstraherer bort den fysiske maskinvaren og tilbyr virtuelle ressurser som lagring og prosessering.

- API-er: Fungerer som abstraksjonslag mellom ulike programvarekomponenter og lar dem kommunisere uten å måtte kjenne til hverandres interne detaljer.
- Nettlesere: Abstraherer bort de komplekse nettverksteknologiene og protokollene involvert i å hente og vise nettsider.
- Operativsystemer: Gir et abstraksjonslag mellom brukeren og maskinvaren, og håndterer lavnivåoppgaver som prosess- og minnehåndtering.

1.4 Eksempler på abstraksjon i praksis

La oss se nærmere på noen konkrete eksempler på hvordan abstraksjon anvendes i ulike deler av dataverdenen.

Nettverkskommunikasjon og TCP/IP-modellen - Et godt eksempel på abstraksjonsnivåer er TCP/IP-modellen for nettverkskommunikasjon. Denne modellen består av flere lag, hver med sin spesifikke funksjon:

- Applikasjonslaget: Der nettverksapplikasjoner og høynivåprotokoller som HTTP og FTP opererer.
- Transportlaget: Ansvarlig for ende-til-ende kommunikasjon og feilhåndtering, med protokoller som TCP og UDP.
- Nettverkslaget: Håndterer adressering og ruting av datapakker, ofte ved hjelp av IP-protokollen.
- Datalinklaget: Overfører data mellom noder på samme lokalnett.
- Det fysiske laget: Spesifiserer de fysiske og elektriske egenskapene til nettverkstilkoblingen.

Hvert lag tilbyr tjenester til laget over, og benytter tjenestene til laget under. En webutvikler kan for eksempel lage en nettside uten å bekymre seg for detaljene på de lavere lagene, som rutingtabeller eller signalstyrke.

Minnehierarki og caching - Datamaskiner bruker et hierarki av minnetyper med ulike hastigheter og kapasiteter, fra CPU-registre og cache til hovedminne (RAM) og harddisker. Disse danner et abstraksjonshierarki som optimaliserer ytelse og kostnader.

CPU-cacher er et godt eksempel på hvordan abstraksjon kan forbedre ytelse. Ved å lagre ofte brukte data i raskt cache-minne nær prosessoren, kan man redusere ventetiden ved minnetilgang betraktelig. Programmereren trenger ikke å håndtere dette eksplisitt - cachekontrolleren sørger for å holde dataene synkronisert med hovedminnet på en måte som er transparent for programvaren.

Virtualisering og skytjenester - Virtualisering er en kraftig abstraksjon som lar én fysisk datamaskin opptre som flere virtuelle maskiner. Hver virtuell maskin kan kjøre sitt eget operativsystem og applikasjoner, uavhengig av de andre. Hypervisoren, som styrer virtualiseringen, abstraherer bort detaljene ved den fysiske maskinvaren og fordeler ressursene mellom de virtuelle maskinene.

Skytjenester bygger på virtualisering for å tilby abstrakte databehandlingsressurser over internett. Brukere kan provisjonere virtuelle maskiner, lagring og andre tjenester uten å måtte forholde seg til den underliggende fysiske infrastrukturen. Dette tillater stor fleksibilitet og skalerbarhet.

1.5 Konklusjon

Abstraksjonsnivåer er et fundamentalt konsept i informatikk som hjelper oss å håndtere kompleksitet og muliggjør utvikling av avanserte datamaskinsystemer. Ved å dekomponere komplekse systemer i lag

med veldefinerte grensesnitt, kan vi fokusere på ett nivå av gangen, gjenbruke komponenter, og tillate innovasjon og optimalisering på hvert nivå uavhengig av de andre.

Abstraksjon er nøkkelen til å bygge komplekse systemer som er forståelige, vedlikeholdbare og skalerbare. Enten det er snakk om høynivåprogrammering, nettverksdesign eller virtualisering, er evnen til å tenke på og arbeide med ulike abstraksjonsnivåer en essensiell ferdighet for enhver som arbeider med informatikk og teknologi.

Det er viktig å huske at abstraksjon er et kompromiss. Høyere abstraksjonsnivåer forenkler design og resonnering, men kan skjule detaljer som er viktige for ytelse eller ressursbruk. En dyktig informatiker må kunne navigere disse avveiningene og velge det riktige abstraksjonsnivået for oppgaven.

Etter hvert som datamaskinsystemer fortsetter å utvikle seg og bli mer komplekse, vil forståelsen av abstraksjonsnivåer forbli en sentral del av informatikkfaget. Enten du er en programvareutvikler, systemarkitekt eller forsker, vil evnen til å tenke abstrakt og bygge og bruke gode abstraksjoner være avgjørende for å lykkes i en stadig mer digitalisert verden.

1.6 Videre lesning

For mer avanserte studenter som ønsker å utforske temaet videre:

Abstraksjonslekasje: Situasjoner hvor detaljer fra et lavere abstraksjonsnivå "lekker" gjennom til et høyere nivå og bryter abstraksjonsbarrieren. Dette kan føre til uventede problemer og kompleksitet.

Ytelse vs. abstraksjon: Det er ofte en avveining mellom høynivåabstraksjoner og ytelse. Høyere abstraksjonsnivåer gir ofte enklere kode, men kan medføre ytelseskostnader.

Domenespesifikke språk (DSL-er): Spesialiserte programmeringsspråk designet for spesifikke domener eller oppgaver. De tilbyr et høynivågrensesnitt skreddersydd for et bestemt problem.

Abstraksjon i distribuerte systemer: Hvordan kan vi designe abstraksjoner som effektivt skjuler kompleksiteten ved nettverksforsinkelser, partisjonering og konsistensproblemer?

Formelle metoder og abstraksjon: Matematiske rammeverk som typeteori og kategoriteori kan gi dypere innsikt i naturen av abstraksjon i informatikk.

Kvanteberegning og abstraksjon: Ettersom kvanteberegning blir mer relevant, må vi tenke på hvordan vi kan bygge abstraksjonslag over de unike egenskapene og utfordringene ved kvantesystemer.

1.7 Refleksjon

Tenk over disse spørsmålene:

- Hvordan kan abstraksjonsnivåer gjøre teknologi mer tilgjengelig for ikke-tekniske brukere?
- Kan du identifisere ulike abstraksjonsnivåer i en hverdagslig gjenstand, som en bil eller en kaffemaskin? Hvordan forenkler disse abstraksjonene bruken av gjenstanden?
- Hvorfor tror du det finnes så mange ulike programmeringsspråk, og hvordan relaterer dette til abstraksjonsnivåer?
- Tenk på et komplekst system du kjenner til. Hvordan kan abstraksjonsnivåer hjelpe med å organisere og forstå et slikt system?

- Kan det noen ganger være nyttig å "bryte" abstraksjonslagene? Hvilke potensielle ulemper eller utfordringer kan dette medføre?

1.8 Selv-test

Test din forståelse med disse spørsmålene:

1. Forklar med dine egne ord hva et abstraksjonsnivå er og gi et eksempel.
2. Ranger følgende i rekkefølge fra høyeste til laveste abstraksjonsnivå: Operativsystem, applikasjonsprogrammer, logiske porter, assemblykode, høynivåprogrammeringsspråk.
3. Hvordan bidrar abstraksjonsnivåer til å håndtere kompleksitet i store programvaresystemer?
4. Gi et eksempel på hvordan abstraksjonsnivåer kan fremme innovasjon i teknologiutvikling.
5. Forklar forskjellen mellom å arbeide på et høyt versus et lavt abstraksjonsnivå i programmering. Når kan det være fordelaktig å jobbe på hvert av disse nivåene?
6. Ranger følgende komponenter i et operativsystem fra høyeste til laveste abstraksjonsnivå: Filsystem, drivere, brukergrensesnitt, prosesshåndtering, systemkall.

Kapittel 2

Digital representasjon

Innhold

2.1 Læringsmål . . . 14
2.2 Bit – byggesteinen . . . 14
2.3 Posisjonssystemer . . . 15
2.4 Konvertering mellom baser . . . 17
2.5 Heksadesimal og oktal . . . 21
2.6 Byte, verdiområde og designvalg . . . 24
2.7 Negative tall . . . 27
2.8 Overflyt . . . 31
2.9 Tekst og tegn . . . 34
2.10 Flyttall og presisjon . . . 39
2.11 Bilde . . . 44
2.12 Lyd . . . 47
2.13 Oppsummering . . . 50
2.14 Videre lesning . . . 51
2.15 Refleksjon . . . 51
2.16 Selv-test . . . 52

Datamaskinen din kan én ting: skille mellom to tilstander. Det er alt. Likevel ringer du bestemoren din med den, hører på musikk og ser bilder av fjell du aldri har vært på.

Stopp litt ved hvor rart det er. Alt dette – farger, lyd, tekst, tall – er gjort om til et spørsmål som bare har to svar. Et spørsmål maskinen stiller milliarder av ganger i sekundet.

Dette kapitlet handler om hvordan. Men først vil jeg gi deg den ene ideen som binder alt sammen. Får du tak i den, henger resten på plass:

Tankevekker

Et bitmønster betyr ingenting i seg selv. Det er tolkningen – avtalen om hva mønsteret skal bety – som gir det mening. Og hver avtale har en grense. Når du treffer den grensen, får du helt konkrete feil i helt virkelige systemer.

Første setning er temaet for første halvdel av kapitlet. Andre setning er temaet for andre halvdel.

Dette er for øvrig den samme tanken som i kapittel 1, sett fra en ny kant. En representasjon er et grensesnitt. Å velge representasjon er å velge abstraksjon. Og som alle abstraksjoner skjuler den noe, og som alle abstraksjoner koster den noe.

2.1 Læringsmål

Når du er ferdig med dette kapitlet, skal du kunne:

- **forklare** hvorfor det samme bitmønsteret kan bety helt ulike ting, og hva som avgjør hvilken tolkning som gjelder
- **konvertere** tall mellom binær, oktal, heksadesimal og desimal, begge veier – og si hvorfor metodene virker
- **regne ut** hvor mange verdier et gitt antall bit rommer, og velge et fornuftig antall bit til en oppgave
- **regne** med negative tall på toerkomplementform, og forklare hvorfor addisjon virker uendret
- **forutsi** når overflyt skjer, hva svaret blir, og hva konsekvensen er
- **forklare** forskjellen på et kodepunkt og en koding, og telle byte i en UTF-8-streng
- **forklare** hvorfor 0,1 ikke finnes eksakt som flyttall, og hva det betyr for kode du skriver
- **vurdere** hva en representasjon av bilde eller lyd koster i plass, og hva komprimering gir og tar

2.2 Bit – byggesteinen

Ordet *bit* kommer fra engelsk *binary digit*, altså binært siffer. Et bit er rett og slett noe som kan være i én av to tilstander.

Vi skriver de to tilstandene som 0 og 1. Men navnene er tilfeldige. Vi kunne like gjerne sagt usant og sant, av og på, lavt og høyt, rødt lys og grønt lys.

Inne i maskinen din er et bit som regel en transistor som enten slipper strøm gjennom eller ikke. Hvordan den er bygget, kommer vi til i kapittel 6. Foreløpig holder det å vite at den er bitteliten, og at det er veldig mange av dem. En vanlig mobiltelefon har rundt hundre milliarder.

Hvorfor akkurat to?

Et spørsmål som ofte kommer: hvorfor nøyer vi oss med to tilstander? Hadde det ikke vært mer effektivt med ti?

Svaret er at to tilstander er *robust*. Tenk deg at du skal rope et tall over en støyete bakgård. Skal du rope et helt siffer fra 0 til 9, må den andre høre nøyaktig hva du sier. Skal du bare rope «ja» eller «nei», kommer beskjeden fram selv om det bråker.

Slik er det inni maskinen også. En krets som skal skille mellom ti spenningsnivåer, må måle presist. En krets som bare skal avgjøre om det er en spenningen (Volt) eller ikke, tåler støy, varme og at komponentene eldes. Vi kommer tilbake til den avveiningen når vi ser på spenningsnivåer og støymargin.

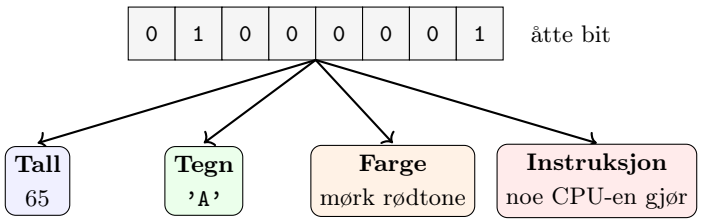
Dette er ikke en naturlov. Det forskes på maskiner som ikke er binære – en kvantedatamaskin har *kubiter*, som kan være i flere tilstander samtidig. Men alle maskiner du kommer til å bruke i dette kurset, og nesten alle som finnes i verden i dag, er binære.

Det samme mønsteret, fire betydninger

Her er åtte bit:

```
01000001
```

Hvilket tall er dette? Det spørsmålet har ikke ett svar. Det er faktisk ikke engang riktig stilt. Se på figur 2.1.



Bitene er de samme. Det eneste som skiller tolkningene, er hva vi har blitt enige om at de skal bety.

Figur 2.1 Ett bitmønster, fire tolkninger.

Som et positivt heltall er `01000001` tallet 65. I tegnkodingen ASCII er det bokstaven `A`. Som én av tre fargekomponenter i et bilde er det en mørk rødtone. Og ligger byten et sted maskinen er i ferd med å hente instruksjoner fra, er det starten på noe CPU-en vil forsøke å utføre.

Ingen av tolkningene er mer riktig enn de andre. Bitene selv sier ingenting om hvilken som gjelder.

Så hvor står det? Det står i datatypen du skrev i koden din. I formatet filen ble lagret som. I protokollen to maskiner ble enige om før de begynte å snakke sammen. Aldri i bitene.

Dette kommer vi tilbake til i hver eneste seksjon.

Note-to-myself

At et bitmønster kan være både data og instruksjon, er ikke en kuriositet. Det er hele ideen bak maskinen med lagret program, og grunnen til at datamaskinen din kan gjøre noe annet i morgen enn den gjorde i dag. Vi tar tråden opp igjen i kapittel 3.

2.3 Posisjonssystemer

Nå skal vi bli enige om hvordan tall skrives. Det høres opplagt ut. Det er det ikke. De neste sidene handler om ett prinsipp, ikke om fire tallsystemer.

Start med det tallsystemet du allerede kan. Hva betyr egentlig 527?

Det betyr fem grupper à hundre, pluss to grupper à ti, pluss sju enere:

$$527_{10} = 5 \times 10^2 + 2 \times 10^1 + 7 \times 10^0$$

Hvert siffer har en **vekt**. Vekten avhenger av *hvor* sifferet står. Den er basen opphøyd i posisjonen, der posisjonene telles fra høyre og starter på null. Sifferet lengst til høyre står i posisjon 0 og har vekten $10^0 = 1$. Naboen til venstre står i posisjon 1 og har vekten $10^1 = 10$. Og så videre.

Hvorfor virker det slik? Det ligger i måten vi teller på. Du teller 0, 1, 2 og oppover til 9. Da har du brukt opp alle sifrene du har. Så nullstiller du og øker naboen med én. Tierplassen er altså ikke annet enn en opptelling av hvor mange ganger du har vært helt gjennom sifferforrådet.

Det er ikke noe magisk med tallet ti her. Ti er valgt fordi vi har ti fingre. Det er hele grunnen. Prinsippet virker like godt med to siffer, eller åtte, eller seksten.

Note-to-myself

Hvorfor er 10^0 lik 1? Dette spørsmålet kommer alltid, og det er et godt spørsmål. Se på rekka baklengs: $10^3 = 1000$, $10^2 = 100$, $10^1 = 10$. For hvert steg deler du på 10. Fortsetter du ett steg til, får du $10^0 = 1$. Det passer med mønsteret, og det er derfor matematikken har bestemt seg for at det skal være slik. Det samme gjelder alle baser: $2^0 = 1$, $16^0 = 1$, $7^0 = 1$. Du trenger ikke bevise det. Du trenger bare å stole på at enerplassen alltid er enerplassen, uansett hvilket tallsystem du er i.

2.3.1 Telling i binær

I base 2 har vi bare to siffer, 0 og 1. Sifferforrådet er altså tomt allerede etter ett steg. Ellers er alt likt:

0	0
1	1
10	2
11	3
100	4
101	5
110	6
111	7
1000	8
1001	9
1010	10

Etter `1` er vi tomme. Da setter vi sifferet til 0 og øker naboen: `10`. Etter `11` er begge tomme, så begge nullstilles og vi tar i bruk en ny posisjon: `100`.

Legg merke til hvor fort tallene blir lange. Det er prisen for å ha få siffer. Vi kommer tilbake til hva vi gjør med det i seksjon 2.5.

2.3.2 Vekt i en hvilken som helst base

Vektene i base 2 er potenser av to: $1, 2, 4, 8, 16, 32, 64, 128$ og videre, fra høyre mot venstre. Generelt gjelder at i base B har sifferet i posisjon k vekten B^k.

Her er en måte å tenke på binære tall som hjelper mange. Se for deg en lommebok med sedler, én av hver valør: 1, 2, 4, 8, 16, 32, 64 og 128 kroner. For hver seddel har du bare to valg – ta den med, eller la den ligge. Et binært tall er ganske enkelt en beskjed om hvilke sedler du tok med.

siffer	1	1	0	0	1
posisjon k	4	3	2	1	0
vekt B^k	2^4	2^3	2^2	2^1	2^0
utregnet	16	8	4	2	1
siffer × vekt	16	8	0	0	1

$$16 + 8 + 0 + 0 + 1 = 25_{10}$$

Figur 2.2 Vektene i det binære tallet 11001_2. Under hvert siffer står posisjonen, vekten og produktet. Summen av produktene er tallets verdi.

Én ting til følger av dette, og vi får bruk for den om litt. Vekten til et siffer er alltid nøyaktig én større enn alt sifrene til høyre kan uttrykke til sammen. Det største tresifrede desimaltallet er 999, og vekten til fjerde siffer er 1000. Det er ingen tilfeldighet.

2.3.3 Notasjon

Skriver du 1001 på en tavle, er det umulig å vite hva du mener. Det kan være det binære tallet, som tilsvarer 9. Det kan være det desimale tallet ett tusen og én.

Vi løser det ved å skrive basen som et lite tall nede til høyre:

- 1001_2 er binært, altså 9_{10}
- 1001_{10} er tusen og én

En liten, men nyttig detalj: *basen selv skrives alltid desimalt.* Toeren i 1001_2 er et vanlig toer. Ellers hadde vi fått et uendelig problem, der vi måtte angi basen til basen til basen.

I programkode brukes andre skrivemåter, og dem kommer vi til i seksjon 2.5: `0b1001` for binær, `0x1001` for heksadesimal, `0o1001` for oktal.

2.4 Konvertering mellom baser

Nå kommer oppskriftene. Det er to retninger, og to metoder i hver retning. Den ene metoden er intuitiv, den andre er mekanisk. Du må kunne begge retninger. Hvilken av de to metodene du velger i hver retning, bestemmer du selv.

2.4.1 Fra en hvilken som helst base til desimal: vektmetoden

Dette har du allerede sett i figur 2.2. Gang hvert siffer med vekten sin, og legg sammen.

Eksempel 1: binær til desimal. Konverter 11001_2.

$$\begin{aligned} 11001_2 &= 1 \times 2^4 + 1 \times 2^3 + 0 \times 2^2 + 0 \times 2^1 + 1 \times 2^0 \\ &= 16 + 8 + 0 + 0 + 1 \\ &= 25_{10} \end{aligned}$$

Eksempel 2: oktal til desimal. Konverter 624_8. Vektene i base 8 er $1, 8, 64, 512$ og videre.

$$\begin{aligned} 624_8 &= 6 \times 8^2 + 2 \times 8^1 + 4 \times 8^0 \\ &= 6 \times 64 + 2 \times 8 + 4 \\ &= 384 + 16 + 4 \\ &= 404_{10} \end{aligned}$$

Eksempel 3: heksadesimal til desimal. Konverter $2A4F_{16}$. Her er A verdien 10 og F verdien 15 – mer om det i neste seksjon. Vektene er $1, 16, 256, 4096$.

$$\begin{aligned} 2A4F_{16} &= 2 \times 16^3 + 10 \times 16^2 + 4 \times 16^1 + 15 \times 16^0 \\ &= 8192 + 2560 + 64 + 15 \\ &= 10\,831_{10} \end{aligned}$$

Eksempel 4: en base som ikke er en toerpotens. Metoden bryr seg ikke om hvilken base du kommer fra. Konverter 142305_6. Vektene i base 6 er $1, 6, 36, 216, 1296, 7776$.

$$\begin{aligned} 142305_6 &= 1 \times 7776 + 4 \times 1296 + 2 \times 216 + 3 \times 36 + 0 \times 6 + 5 \times 1 \\ &= 7776 + 5184 + 432 + 108 + 0 + 5 \\ &= 13\,505_{10} \end{aligned}$$

Det eneste du bytter ut når du skifter base, er basen i vektene. Alt annet står.

Note-to-myself

Ta et overslag først. Det tar fem sekunder og redder deg fra å levere et svar som er dobbelt så stort som det skulle vært.
Det øverste bitet i 110010100_2 har vekten 256. Neste bit er også satt, og legger til 128. Svaret må altså ligge litt over 384. Regner du nøye, får du 404. Gjør dette hver gang.

2.4.2 Fra en hvilken som helst base til desimal: gjentatt multiplikasjon

Den andre metoden går fra venstre mot høyre, og du trenger ingen potenstabell. Start med sifferet lengst til venstre. Gang med basen, legg til neste siffer, gjenta til sifrene er brukt opp.

Eksempel 5. Konverter 13754_8.

1	start med første siffer
$1 \times 8 + 3 = 11$	
$11 \times 8 + 7 = 95$	
$95 \times 8 + 5 = 765$	
$765 \times 8 + 4 = 6124$	$\Rightarrow\ 13754_8 = 6124_{10}$

Hvorfor virker dette? Tell hvor mange ganger hvert siffer blir ganget med 8. Det første blir ganget fire ganger, altså med 8^4. Det andre blir ganget tre ganger, altså 8^3. Det siste blir aldri ganget, og får dermed vekten $8^0 = 1$.

Hvert siffer får altså nøyaktig samme vekt som vektmetoden ville gitt det. Regnestykket er bare delt opp annerledes. Dette er for øvrig metoden maskinen bruker når den skal gjøre tekststrengen `"13754"` om til et tall.

2.4.3 Fra desimal til en hvilken som helst base: største vekt som får plass

Nå den andre veien. Skriv opp vektene i basen du skal til. Finn den største som ikke er større enn tallet ditt, og se hvor mange ganger den får plass. Trekk fra. Gjenta med neste vekt.

Dette er akkurat som å veksle et beløp i sedler: du tar de største først.

Eksempel 6. Konverter 3193_{10} til binær. Vektene vi har å velge mellom er

$$\ldots,\ 4096,\ 2048,\ 1024,\ 512,\ 256,\ 128,\ 64,\ 32,\ 16,\ 8,\ 4,\ 2,\ 1.$$

$4096 > 3193$	for stor, ingen 4096-plass
$3193 - 2048 = 1145$	1 i 2048-plassen
$1145 - 1024 = 121$	1 i 1024-plassen
$121 < 512, 256, 128$	0 i alle tre
$121 - 64 = 57$	1 i 64-plassen
$57 - 32 = 25$	1 i 32-plassen
$25 - 16 = 9$	1 i 16-plassen
$9 - 8 = 1$	1 i 8-plassen
$1 < 4, 2$	0 i begge
$1 - 1 = 0$	1 i 1-plassen

Leser vi av plassene fra 2048 og nedover, får vi $3193_{10} = 110001111001_2$.

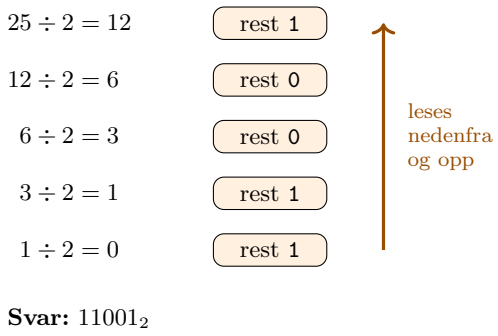
I baser større enn 2 må du i tillegg finne ut *hvor mange ganger* vekten får plass, ikke bare om den får plass. Tar vi samme tall til heksadesimal, med vektene $4096, 256, 16, 1$: 256 får plass 12 ganger, siden $12 \times 256 = 3072$, og resten er 121. Så får 16 plass 7 ganger, og 9 blir stående. Siden 12 skrives C i heksadesimal, er $3193_{10} = \mathrm{C79}_{16}$.

2.4.4 Fra desimal til en hvilken som helst base: divisjonsmetoden

Dette er metoden du kommer til å bruke mest. Den er lett å gjøre helt mekanisk, uten å tenke.

Del tallet på basen. Skriv ned resten. Del kvotienten på basen igjen. Fortsett til kvotienten blir 0. Les restene nedenfra og opp.

Eksempel 7. Konverter 25_{10} til binær. Regnestykket står i figur 2.3, og svaret blir 11001_2 – det samme som vektmetoden ga oss i figur 2.2 på side 17.



Svar: 11001_2

Figur 2.3 Divisjonsmetoden for 25_{10} til base 2. Restene leses nedenfra og opp.

Advarsel :)

Nedenfra og opp. Dette er den vanligste feilen på denne oppgavetypen, og den koster poeng hver eneste gang.
Regnestykket over gir restene 1, 0, 0, 1, 1 i den rekkefølgen du skrev dem. Svaret er *ikke* 10011_2. Det er 11001_2.
Grunnen kommer under: den første resten du finner, er enerplassen. Den hører hjemme lengst til høyre i svaret.

Eksempel 8: en større base. Konverter 8151_{10} til heksadesimal.

$$\begin{aligned} 8151 \div 16 &= 509 \quad \text{rest } 7 \\ 509 \div 16 &= 31 \quad \text{rest } 13 = \text{D} \\ 31 \div 16 &= 1 \quad \text{rest } 15 = \text{F} \\ 1 \div 16 &= 0 \quad \text{rest } 1 \end{aligned}$$

Nedenfra og opp: $8151_{10} = 1\text{FD}7_{16}$.

Eksempel 9: en base uten noe med to å gjøre. Konverter 6124_{10} til base 5.

$$
\begin{aligned}
6124 \div 5 &= 1224 \quad \text{rest } 4 \\
1224 \div 5 &= 244 \quad \text{rest } 4 \\
244 \div 5 &= 48 \quad \text{rest } 4 \\
48 \div 5 &= 9 \quad \text{rest } 3 \\
9 \div 5 &= 1 \quad \text{rest } 4 \\
1 \div 5 &= 0 \quad \text{rest } 1
\end{aligned}
$$

Nedenfra og opp: $6124_{10} = 143444_5$. Sjekk gjerne etter med vektmetoden.

2.4.5 Hvorfor divisjonsmetoden virker

Når du deler et tall på basen B, må resten ligge mellom 0 og $B-1$. Det er nøyaktig det samme området som ett siffer i base B dekker. Restene *kan* altså ikke bli noe annet enn gyldige siffer. Det er ingen tilfeldighet – det er hele trikset.

Den første divisjonen spør: hvor mange hele grupper à B får plass i tallet? Kvotienten er antall grupper. Resten er hvor mange enkeltenheter som blir til overs når gruppene er trukket fra. Og de enkeltenhetene *er* enerplassen.

Etter første divisjon er tallet du sitter igjen med allerede delt på B én gang. Neste divisjon spør derfor egentlig: hvor mange grupper à B^2 får plass? Resten denne gangen er hvor mange B-grupper som ble til overs. Det er nest siste siffer.

Slik fortsetter det. For hver divisjon blir gruppene B ganger større. Når kvotienten når null, er det ingenting igjen å gruppere. Og siden det høyre sifferet ble funnet først, må restene leses nedenfra og opp.

Fun Fact

Divisjonsmetoden bruker *modulo*, operasjonen som gir resten av en heltallsdivisjon. I C, Java og Python skrives den `%`. Uttrykket `25 % 2` gir 1.
Skal du skrive et program som konverterer et tall til binær – og det skal du – trenger du bare to operasjoner: `%` og heltallsdivisjon. Det er hele algoritmen.

2.5 Heksadesimal og oktal

Binære tall blir fort uleselige. Én million er 11110100001001000000_2. Det er tjue siffer, og du klarer ikke å sjekke om jeg har skrevet dem riktig uten å telle dem.

Vi mennesker har en løsning på dette fra før. Tenk på et mobilnummer. Ingen skriver 90123456. Vi skriver 90 12 34 56, fordi hjernen takler fire små biter bedre enn åtte siffer på rad. Kontonummer og fødselsnummer grupperes på samme måte.

Vi skal gjøre nøyaktig det samme med bit. Maskinen er og blir binær, og det kan vi ikke gjøre noe med. Men vi kan finne en kortskrift.

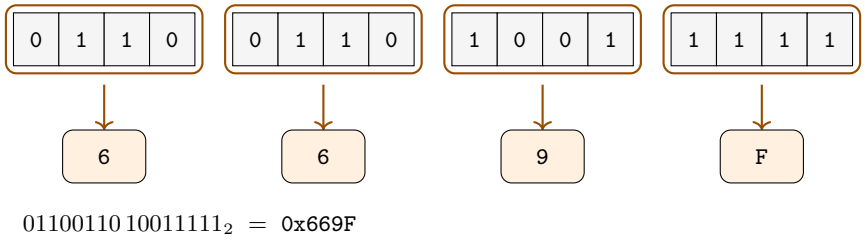
2.5.1 Heksadesimal

Heksadesimal er base 16. Det gir seksten siffer. Siden vi bare har ti sifferttegn, låner vi de seks første bokstavene:

```
0 1 2 3 4 5 6 7 8 9 A B C D E F
```

A er 10, B er 11, C er 12, D er 13, E er 14 og F er 15.

Hele poenget ligger i ett lite regnestykke: $16 = 2^4$. Når du har telt fra 0 til F, har du brukt opp nøyaktig alle kombinasjoner av fire bit. Derfor svarer **ett heksadesimalt siffer til nøyaktig fire bit**. Alltid, uten unntak, uten avrunding.



Figur 2.4 Grupper bitene i firere fra høyre, og slå opp hver gruppe. Ingen regning gjennom desimal.

Oppslagstabellen er hele metoden:

Binær	Heks	Desimal	Binær	Heks	Desimal
0000	0	0	1000	8	8
0001	1	1	1001	9	9
0010	2	2	1010	A	10
0011	3	3	1011	B	11
0100	4	4	1100	C	12
0101	5	5	1101	D	13
0110	6	6	1110	E	14
0111	7	7	1111	F	15

Tabell 2.1 De seksten firebitsmønstrene. Denne tabellen er verdt å kunne utenat.

2.5.2 Hva om det ikke går opp i fire?

Et spørsmål som kommer hver gang, og som du garantert møter i første øving. Hva gjør du med 1011010_2? Det er sju bit. Grupperer du i firere, sitter du igjen med en gruppe på tre.

Svaret er enkelt: **grupper alltid fra høyre, og fyll på med nuller til venstre.**

```
 1011010      sju bit
 101 1010     gruppert fra høyre -- øverste gruppe mangler ett bit
0101 1010     fylt på med en null foran
  5    A      slått opp i tabellen
```

Så $1011010_2 = 5\mathrm{A}_{16}$.

Å legge til nuller foran endrer ingenting. Akkurat som 0042 er det samme tallet som 42, er `0101` det samme som `101`. Nuller helt til venstre er gratis.

I praksis slipper du som regel unna problemet. Inne i en maskin kommer ting nesten alltid i hele byte, altså åtte bit, og åtte går pent opp i to firergrupper. Men regelen er verdt å ha i bakhodet.

2.5.3 Hvorfor skriver vi 0x?

Heksadesimale tall skrives med prefikset `0x` i de fleste programmeringsspråk: `0x669F`.

Grunnen er praktisk og litt gammeldags. Da programmeringsspråkene ble laget, skrev man kode i en editor som bare hadde ett skriftsnitt og én tegnstørrelse. Du kunne rett og slett ikke skrive små, senkede tall. Så man fant på et prefiks i stedet, og det har hengt med siden.

De to skrivemåtene betyr det samme. `0x669F` og $669F_{16}$ er samme tall.

Advarsel :)

Ikke gå veien om desimal. Dette er den klassiske tidstyven på eksamen.
Oppgaven er: her er HTML-fargen `0x6699FF`, skriv rød-, grønn- og blåkomponenten binært. Den er ment å ta to minutter. Mange bruker førti.
Feilen er å regne heks til desimal, og så desimal til binær, tre ganger. Det er seks unødvendige regnestykker, hvert med sin egen sjanse for slurv.
Slå opp i stedet. `6` er `0110`. `9` er `1001`. Ferdig:

rød	`0x66`	→	`0110 0110`
grønn	`0x99`	→	`1001 1001`
blå	`0xFF`	→	`1111 1111`

Å konvertere mellom binær og heksadesimal via desimal er som å oversette fra bokmål til nynorsk via japansk. Det går an. Men hvorfor?

2.5.4 Heksadesimal i det fri

Farger på nettsider er heksadesimale. En farge er seks heksadesimale siffer, altså 24 bit, fordelt på tre komponenter à én byte:

	`#6699FF`	rød 102, grønn 153, blå 255	lyseblå
	`#FF0000`	rød 255, grønn 0, blå 0	full rød
	`#000000`	alt av	svart
	`#FFFFFF`	alt på	hvit
	`#87CEFA`	rød 135, grønn 206, blå 250	himmelblå

Heksadesimal dukker også opp i minneadresser, i feilmeldinger, i MAC-adresser, i IPv6-adresser og i tegnkodetabeller. Det siste møter vi i seksjon 2.9.

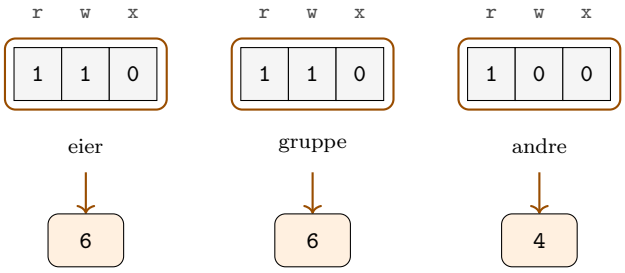
2.5.5 Oktal

Oktal er base 8, med sifrene 0 til 7. Samme idé som heksadesimal, men nå er $8 = 2^3$: **ett oktalt siffer er nøyaktig tre bit**.

Oktal var mer utbredt før. Men det har overlevd på ett sted alle informatikkstudenter møter: filrettigheter i UNIX og Linux.

En fil i Linux har ni rettighetsbit. Tre gjelder eieren av filen, tre gjelder gruppen filen tilhører, og tre gjelder alle andre. Innenfor hver trio betyr bitene lese, skrive og utføre, i den rekkefølgen.

Ni bit deler seg pent i tre grupper à tre. Og tre bit er nøyaktig ett oktalt siffer. Derfor skrives rettigheter som tre oktale siffer.



`chmod 664 notat.txt` – eier kan lese og skrive, gruppa kan lese og skrive, andre kan bare lese

Figur 2.5 Ni rettighetsbit gruppert i tre oktale siffer. 110 er 6, 100 er 4.

Skriver du `chmod 755 skript.sh`, sier du altså `111 101 101`. Eieren kan lese, skrive og utføre. Gruppa og alle andre kan lese og utføre, men ikke skrive. Og når `ls -l` viser deg `-rwxr-xr-x`, er det den samme informasjonen skrevet som bokstaver i stedet for som tall.

En liten felle til slutt: oktal og heksadesimal er ikke beslektet med hverandre. 8 er ingen potens av 16. Skal du fra oktal til heksadesimal, må du innom binær. Men det er ingen omvei, for begge stegene er rene oppslag.

2.6 Byte, verdiområde og designvalg

Ett bit rommer to verdier. To bit rommer fire: `00`, `01`, `10` og `11`. Tre bit rommer åtte. Hvert nytt bit dobler antall kombinasjoner, så

$$n \text{ bit rommer } 2^n \text{ ulike verdier, altså tallene fra 0 til } 2^n - 1.$$

Tre bit er ikke mye. Du får ikke engang lagret alderen din. Derfor grupperer vi bitene.

2.6.1 Byten

En **byte** er åtte bit. Det gir $2^8 = 256$ verdier, altså 0 til 255.

Byten er den minste enheten de fleste maskiner faktisk henter fra minnet. Vil du ha tak i ett enkelt bit, må du hente hele byten og plukke bitet ut selv.

Trenger du mer, tar du flere byte:

Bredde	Bit	Antall verdier	Område (uten fortegn)
1 byte	8	256	0 – 255
2 byte	16	65.536	0 – 65.535
4 byte	32	4.294.967.296	0 – ca. 4,3 milliarder
8 byte	64	$1{,}8 \times 10^{19}$	0 – ca. $1{,}8 \times 10^{19}$

Tabell 2.2 Hvor langt du kommer med et gitt antall byte.

Denne tabellen danner mye av grunnlaget for resten av kapitlet. Legg merke til hva den egentlig sier: *antall bit er et valg, og valget har en konsekvens.* Velger du for få, får du feil svar. Velger du for mange, sløser du med plass og båndbredde. Det finnes ikke noe riktig svar uten at du vet hva tallet skal brukes til.

Fun Fact

Hvorfor heter det egentlig *byte*? Det er det ingen full enighet om. Den vanligste forklaringen er at det er en lek med ordet *bite*, altså en jafs – en bit er en bitte liten flis, en byte er en hel munnfull. En annen forklaring er at det er en sammentrekning av *by eight*.
At en byte er åtte bit, er dessuten en konvensjon og ikke en naturlov. Tidlige maskiner brukte byte på 6, 7 og 9 bit. Standarden satte seg først på 1960-tallet.
Derfor finnes ordet *oktett*, som du har møtt i nettverksdelen av emnet. En oktett er per definisjon åtte bit, verken mer eller mindre. Nettverksstandarder bruker det ordet nettopp fordi de ikke kan ta for gitt hva «byte» betyr på maskinen i den andre enden.

2.6.2 Hvor mange bit trenger jeg?

Dette spørsmålet dukker opp hver gang du lager en variabel. Det finnes en enkel tommelfingerregel.

Siden $2^{10} = 1024 \approx 1000$, rommer ti bit omtrent tusen verdier. Derfra kan du gange deg oppover:

- 10 bit gir omtrent tusen verdier
- 20 bit gir omtrent en million
- 30 bit gir omtrent en milliard
- 18 bit deler seg i $10 + 8$, altså omtrent $1000 \times 256 = 256.000$

Litt mer presist trengs det omtrent **3,3 bit per desimalsiffer**, fordi $2^{3{,}3} \approx 10$. Skal du lagre et sekssifret tall, trenger du rundt 20 bit.

Advarsel :)

MB eller MiB? Her er en av de mest utbredte kildene til forvirring i faget.
I resten av verden betyr prefikset M en million, altså 10^6. I databransjen har det lenge betydd $2^{20} = 1.048.576$, fordi den potensen ligger nær en million og passer med hvordan minne er bygget. De to tallene skiller seg med nesten fem prosent, og forskjellen vokser oppover: for T er avviket ti prosent.
Standarden IEC 80000-13 løser det med egne prefikser for toerpotensene:

1 kB = 10^3 byte	1 KiB = 2^{10} = 1.024 byte
1 MB = 10^6 byte	1 MiB = 2^{20} = 1.048.576 byte
1 GB = 10^9 byte	1 GiB = 2^{30} = 1.073.741.824 byte

Diskprodusenter oppgir kapasitet i GB. Operativsystemer viser ofte GiB. Det er derfor en «1 TB»-disk plutselig har 931 GiB når du kobler den til. Ingen har jukset – det er samme tall, oppgitt i to ulike enheter.
I denne boka betyr MB alltid 10^6 byte, med mindre noe annet står.

2.6.3 BCD – når desimal må være desimal

Før vi forlater heltallene, skal vi se på en representasjon som velger stikk motsatt av alt vi har gjort til nå.

BCD, *binary coded decimal*, koder hvert desimalsiffer for seg i fire bit. Tallet 68 blir ikke 1000100_2, men

`0110 1000` der `0110` er sifferet 6 og `1000` er sifferet 8.

Mønstrene `1010` til `1111` – altså A til F – brukes aldri. Seks av seksten mulige mønstre kastes bort i hver eneste rute. Tabellen under viser hvor mye vi mister etterhvert som antall bitplasser øker:

Bit	BCD	Rent binært
8	0 – 99	0 – 255
16	0 – 9.999	0 – 65.535
20	0 – 99.999	0 – 1.048.575
32	0 – 99.999.999	0 – 4.294.967.295
64	16 siffer	19 hele siffer

Tabell 2.3 BCD kaster bort plass, og det blir verre jo bredere tallet er.

Hvorfor i all verden bruke det, da?

Fordi noen desimaltall rett og slett ikke finnes i binær. Vi skal se i kapittel 2.10 at 0,1 ikke kan skrives eksakt med bit. En bank som regner i kroner og øre, kan ikke leve med at 0,10 kroner blir 0,100000001. BCD gir eksakt desimalpresisjon, og det er verdt både plassen og tiden når pengene skal stemme på øret. IBMs stormaskiner har fortsatt maskinvarestøtte for BCD, og kode i banker og forsikringsselskaper bruker det i dag.

Det finnes en bonusgrunn også. I ASCII og Unicode er de fire nederste bitene i et siffertegn nøyaktig BCD-sifferet. Tegnet `'7'` har koden `0011 0111`, og de fire siste bitene er `0111`, altså 7. Å gjøre innskrevet tekst om til BCD er derfor nesten gratis.

BCD er et godt eksempel på den bærende ideen. Representasjonen har valgt eksakthet framfor kompakthet. Derfor er den riktig noen steder, og feil de fleste andre.

2.7 Negative tall

Alt vi har gjort til nå gjelder positive heltall. Men programmer må kunne trekke fra. Kontoer går i minus, og termometre viser kuldegrader.

Problemet er at vi ikke har noe minustegn. Vi har bare bit. Skal vi klare å si «minus», må vi bygge det av nuller og enere som alle de andre tingene.

Det finnes flere måter. Vi tar den som vant først, og de som tapte etterpå – for det er lærerikt å se hvorfor de tapte.

2.7.1 Toerkomplement: øverste bit har negativ vekt

Toerkomplement er det praktisk talt alle maskiner bruker. Og hele definisjonen får plass i én setning:

Tankevekker

I toerkomplement er alt akkurat som før. Bortsett fra at det øverste bitet har *negativ* vekt.

Det er alt. Ingen ny regnemåte, ingen egen fortegnsregel. Vektene i en byte er fortsatt potenser av to, men den øverste har fått et minustegn foran seg:

posisjon	7	6	5	4	3	2	1	0
vekt	-128	64	32	16	8	4	2	1

Alt annet gjør du som før. Gang hvert bit med vekten sin, og legg sammen.

Husker du lommeboka fra seksjon om vekter? Nå har den fått ett nytt kort: et gjeldskort på 128 kroner. Tar du det med, starter du på minus 128 og kan jobbe deg oppover med de vanlige sedlene.

Eksempel 10. Hvilket tall er `10000000`?

$$1 \times (-128) + 0 + 0 + 0 + 0 + 0 + 0 + 0 = -128$$

Eksempel 11. Hvilket tall er `11111111`?

$$-128 + 64 + 32 + 16 + 8 + 4 + 2 + 1 = -128 + 127 = -1$$

Eksempel 12. Hvilket tall er `11111011`?

$$-128 + 64 + 32 + 16 + 8 + 0 + 2 + 1 = -128 + 123 = -5$$

Eksempel 13. Hvilket tall er `00000101`?

$$0 + 0 + 0 + 0 + 0 + 4 + 0 + 1 = 5$$

Legg merke til to ting.

Er øverste bit 0, bidrar det ingenting. Tallet er positivt, akkurat som før. De positive tallene ser helt like ut som de gjorde i forrige seksjon.

Er øverste bit 1, starter du på -128 og legger til. Da havner du et sted mellom -128 og -1.

Derfor kalles øverste bit ofte *fortegnsbitet*. Vær litt forsiktig med det navnet. Det er ikke et rent fortegn slik minustegnet er det. Det er et siffer med en vekt, som alle de andre. Vekten er bare negativ.

2.7.2 Å snu fortegnet

For å negere et toerkomplementtall gjør du to ting: **inverter alle bitene, og legg til 1**.

Eksempel 14. Finn -5 ut fra 5.

$$\begin{aligned} 5 &= \texttt{00000101} \\ \text{inverter} &\to \texttt{11111010} \\ \text{legg til 1} &\to \texttt{11111011} \end{aligned}$$

Og `11111011` er nettopp -5, slik vi regnet ut i eksempel 12.

Metoden virker begge veier. Inverterer du `11111011`, får du `00000100`. Legger du til 1, får du `00000101`, altså 5 igjen.

Hvorfor virker det? Å invertere alle bitene i et åttebitstall er det samme som å trekke tallet fra `11111111`, som er 255. Legger du så til 1, har du regnet ut $256 - x$. Og $256 = 2^8$ er nøyaktig det punktet der en byte «renner rundt». Hva det betyr, kommer med én gang.

2.7.3 Hvorfor addisjon virker uendret

Her ligger hele grunnen til at toerkomplement vant.

Prosessoren trenger ikke vite om tallene er positive eller negative. Den legger sammen bit for bit som om alt var vanlige positive tall, kaster bitet som ikke får plass, og svaret blir riktig.

Eksempel 15. Regn ut $5 + (-3)$ med åtte bit.

```
  00000101     (5)
+ 11111101     (-3)
----------
1 00000010     menten faller utenfor
  ========
  00000010     = 2
```

Svaret er 2, som det skal være. Menten lengst til venstre forsvinner bare. Den er ikke en feil, og du skal ikke ta vare på den.

Eksempel 16. Regn ut $-5 + 3$.

```
  11111011     (-5)
+ 00000011     (3)
----------
  11111110     = -2
```

Riktig igjen, og denne gangen uten mente i det hele tatt.

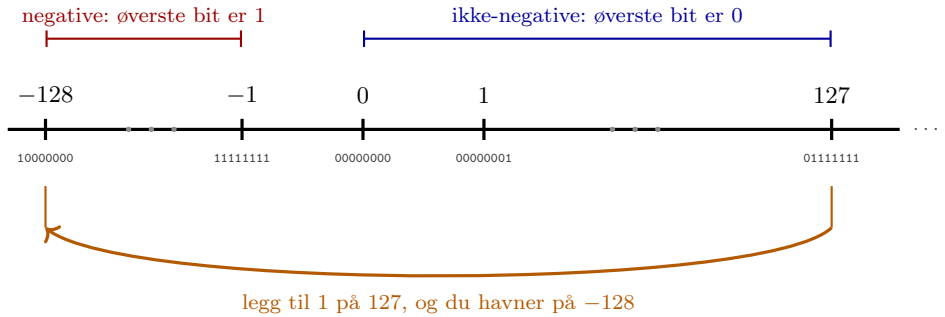
Tankevekker

Tenk på en klokke. Den er analogien som får dette til å klikke på plass for de fleste. Klokka er 3. Du vil gå fire timer tilbake. Du kan trekke fra 4 og få 11. Eller du kan legge til 8 og gå hele veien rundt skiva – $3 + 8 = 11$. Samme svar.
Det er fordi skiva har tolv plasser. Å trekke fra 4 og å legge til $12 - 4 = 8$ gir alltid det samme, så lenge du kaster hele runder.
En byte er en klokke med 256 plasser. Å trekke fra 5 og å legge til $256 - 5 = 251$ gir samme svar, så lenge du kaster hele runder. Og å kaste hele runder er nøyaktig det som skjer når det niende bitet faller utenfor.
Derfor trenger maskinen aldri en egen subtraksjonskrets. *Trekk fra* blir *negér og legg til*, og ALU-en slipper unna med én operasjon. Vi ser hvordan den bygges i kapittel 5.

2.7.4 Tallinja og verdiområdet

Med åtte bit i toerkomplement får vi tallene fra -128 til 127.

Den øvre grensen er `01111111`, altså $64 + 32 + 16 + 8 + 4 + 2 + 1 = 127$. Den nedre er `10000000`, altså -128. Til sammen 256 verdier – akkurat like mange som en byte alltid rommer. Vi har bare flyttet halvparten av dem ned under null.



Figur 2.6 Toerkomplement med åtte bit. Tallinja er egentlig en ring – teller du forbi 127, kommer du inn igjen nederst.

To ting følger av dette.

Det finnes ingen minus null. Negerer vi `00000000`, får vi `11111111` etter inverteringen. Legger vi til 1, blir det `1 00000000`. Menten faller utenfor, og vi står igjen med `00000000` – samme mønster vi startet med.

Null er altså sin egen negasjon, og det finnes bare én av den. Det høres ut som en detalj. Men det betyr at maskinen slipper å teste for to ulike nullmønstre etter hver eneste operasjon, og det er mye spart.

Området er ikke symmetrisk. Det finnes et -128, men ikke et $+128$. Det gir en pussighet som faktisk biter: negerer du -128, får du -128 tilbake. Inverter `10000000` og du får `01111111`. Legg til 1, og du er tilbake på `10000000`.

I Java betyr det at `Math.abs(Integer.MIN_VALUE)` gir et negativt tall. Det er ingen feil i Java. Det er en representasjon som har møtt grensen sin.

2.7.5 De som tapte

Toerkomplement er den beste av flere måter, og det er nyttig å vite hva den ble målt mot.

Fortegn og magnitude. Den mest opplagte ideen. Bruk øverste bit som et rent fortegn, og la de sju andre være tallets størrelse. Da er `00100101` lik $+37$ og `10100101` lik -37. Lett å lese for et menneske.

To ting ødelegger den. For det første finnes det to nuller, `00000000` og `10000000`, altså $+0$ og -0. Maskinen må teste for begge hver gang den lurer på om noe er null.

For det andre blir addisjonen et mareritt. Skal maskinen legge sammen 4 og -2, må den først sammenligne fortegn, så sammenligne størrelser, så bestemme om den skal legge til eller trekke fra, og til slutt finne fortegnet på svaret. Fire tilfeller, der toerkomplement har ett.

Fortegn og magnitude er ikke helt borte, forresten. Flyttall bruker det, og vi ser hvorfor i seksjon 2.10.

Enerkomplement. En smartere idé. Negér ved å invertere alle bitene, uten å legge til 1. Da er -37 lik `11011010`.

Addisjonen blir nesten grei. Men bare nesten: renner det ut en mente, må den legges til på nytt nederst. Og de to nullene er fortsatt der – `00000000` og `11111111` er begge null.

Toerkomplement er altså enerkomplement pluss én. Den ene ekstra addisjonen ved fortegnsskifte er hele prisen. Til gjengjeld forsvinner både minus null og den ekstra menten. Det er en veldig god handel, og den er grunnen til at maskinen på pulten din regner slik den gjør.

2.7.6 Hvordan vet du at det er toerkomplement?

Dette er et av de beste spørsmålene man kan stille i dette kapitlet.

Du får utdelt en byte:

```
10000000
```

Er det -128 eller 128?

Svaret er at du ikke kan vite det. Begge tolkningene er helt gyldige. Er byten en `signed char` i C, er den -128. Er den en `unsigned char`, er den 128. Bitene er de samme i begge tilfeller, og maskinen lagrer ikke noe sted hvilken som gjelder.

Opplysningen ligger i *typen*. Den står i kildekoden din. Den finnes i kompilatorens hode mens den oversetter, og den avgjør hvilke maskininstruksjoner kompilatoren velger. Men den finnes ikke i minnet. Når programmet kjører, er det bare bitene igjen.

Dette er den bærende ideen i sin skarpeste form. Det er også kilden til en hel familie virkelige feil. Bytter du en `signed` mot en `unsigned` i en lengdesjekk, kan et negativt tall plutselig bli en veldig stor positiv lengde. Og der har du et sikkerhetshull.

Note-to-myself

I Java har alle heltallstypene fortegn: `byte` (−128 til 127), `short`, `int` (rundt ±2,1 milliarder) og `long`.
C og C++ har i tillegg `unsigned`-varianter. De har dobbelt så stort positivt område, fordi de slipper å bruke halvparten av mønstrene på negative tall.
Valget er ditt, og du må ta det bevisst. Spør alltid: kan denne verdien bli negativ, og hvor stor kan den bli?

2.8 Overflyt

Nå har vi etablert første halvdel av den bærende ideen: et bitmønster betyr det vi har blitt enige om at det betyr. Resten av kapitlet handler om andre halvdel – at hver representasjon har en grense, og hva som skjer når du treffer den.

2.8.1 Kilometerteller en

Tenk på kilometertelleren i en gammel bil, den med mekaniske hjul. Den har seks siffer og viser opptil `999999`. Kjører du én kilometer til, ruller alle hjulene rundt samtidig, og telleren viser `000000`.

Bilen har ikke glemt de kilometerne. Telleren har bare ikke plass til å si det. Den har møtt grensen sin, og reagerer på den eneste måten den kan: den begynner på nytt.

Det samme skjer i en datamaskin. Det heter **overflyt**.

2.8.2 Overflyt i praksis

Her er et program som ser ganske uskyldig ut:

```
int v1 = 1000;
int v2 = 1000 * 1000;
int v3 = 1000000 * 1000000;

System.out.println(v1);
System.out.println(v2);
System.out.println(v3);
```

og her er hva det skriver ut:

```
1000
1000000
-727379968
```

Den siste linja er grundig feil. Feil størrelsesorden, feil antall siffer, feil fortegn.

Hva skjedde? En `int` i Java er fire byte, altså 32 bit, og rommer tall opp til 2 147 483 647. Det riktige svaret, 1 000 000 000 000, trenger 40 bit. De åtte øverste bitene har ingen steder å være. De blir kastet – ikke lagret, ikke rapportert, ikke mulig å få tilbake.

Og se hva som blir igjen. Av de 32 bitene som overlever, er det øverste satt. I toerkomplement har det øverste bitet vekten -2^{31}. Derfor blir svaret negativt. Det er ikke tilfeldig og ikke mystisk – det er nøyaktig regelen fra seksjon 2.7, anvendt på et bitmønster som havnet der ved et uhell.

Advarsel :)

Det verste med overflyt er ikke at svaret blir feil. Det er at ingen sier fra.
Programmet over kaster ingen feilmelding, logger ingenting og krasjer ikke. Det regner videre med $-727\,379\,968$ som om det var et helt vanlig tall. Alt som beregnes etterpå blir feil, og feilen kan dukke opp mange steg senere, et helt annet sted i programmet.
Derfor bør du venne deg til å tenke på verdiområdet *før* du velger datatype. Ikke etterpå.

2.8.3 Hva kan vi gjøre?

Det opplagte svaret er å bruke en større type. Bytter vi `int` mot `long`, som er åtte byte, får 10^{12} god plass, og programmet skriver ut riktig svar.

Men det er en utsettelse, ikke en løsning. En `long` har også en grense. Den ligger bare lenger ute.

2.8.4 Python, og hvorfor det ikke er gratis

Noen språk velger en helt annen vei. Python har heltall uten fast bredde. Trenger tallet flere bit, skaffer Python flere bit. Du kan gange sammen tall til minnet ditt tar slutt:

```
v = 1000000 * 1000000
for i in range(10):
    v = v * v
    print(v)
```

Dette gir riktig svar hele veien. Det er behagelig, og en av grunnene til at Python er et godt førstespråk.

Men det koster, og det koster på to måter.

Den ene er den opplagte. Hver eneste regneoperasjon må sjekke om resultatet fortsatt får plass, og eventuelt skaffe mer minne. Der C legger sammen to tall med én maskininstruksjon, gjør Python et titalls.

Den andre er mindre opplagt og verre. Når kompilatoren ikke vet hvor stort et tall kan bli, kan den heller ikke optimalisere. Den kan ikke legge tallet i et register, den kan ikke bruke den raske addisjonsinstruksjonen, den kan ikke slå sammen operasjoner. Usikkerheten om størrelse forplanter seg gjennom hele oversettelsen.

Til sammen er dette en stor del av grunnen til at Python er tregt. Det er et bevisst designvalg: språket har prioritert at det skal være behagelig å programmere i, framfor at det skal gå fort. C har valgt motsatt. Ingen av valgene er feil, men du bør vite hvilket du har gjort.

Advarsel :)

Og her er fella. Det er lett å lese avsnittet over og konkludere med at «i Python slipper jeg å tenke på overflyt». Det stemmer ikke.
Store deler av Pythons biblioteker er skrevet i C. Og C har ikke Pythons forhold til tallstørrelse – der er en 32-bits heltall en 32-bits heltall, punktum.
Regner du med vanlige Python-tall, er du trygg. Regner du i et bibliotek som NumPy, er du tilbake i C-verdenen med faste bitbredder og stille overflyt. Samme språk, samme fil, to helt ulike regler.
Sjekk hvilken datatype du faktisk regner med. Ikke hvilket språk du skriver i.

Prosessoren har for øvrig egne flagg for både mente og overflyt, som et program *kan* sjekke. Vi ser på dem i kapittel 4.

2.8.5 Ariane 5

Den 4. juni 1996 skjøt Den europeiske romfartsorganisasjonen opp Ariane 5 for første gang. Trettini sekunder senere var raketten og de fire satellittene om bord borte. Tapet anslås vanligvis til rundt 370 millioner dollar.

Årsaken var en konvertering mellom to representasjoner.

Rakettens treghetsnavigasjonssystem holdt rede på horisontal hastighet som et 64-bits flyttall. Denne verdien ble konvertert til et 16-bits fortegnstall, som rommer tall opp til 32 767. Under oppstigningen ble den horisontale hastigheten større enn det. Konverteringen feilet, og navigasjonsdatamaskinen slo seg av. Reservesystemet kjørte identisk programvare og hadde feilet av samme grunn noen hundredels sekund tidligere. Styringssystemet tolket feilmeldingen som flydata, svingte dysene helt ut, raketten begynte å brekke, og selvdestruksjonen utløste seg.

Det interessante er hvor koden kom fra. Den var arvet fra Ariane 4, der den hadde virket feilfritt i årevis. På Ariane 4 *kunne ikke* den horisontale hastigheten bli stor nok til å sprenge et 16-bits tall. Flybanen tillot det rett og slett ikke. Representasjonen var riktig valgt, gitt forutsetningene.

Ariane 5 var kraftigere og fløy annerledes. Forutsetningen holdt ikke lenger. Ingen hadde gjort en feil i koden. De hadde arvet en antakelse om verden, og verden hadde endret seg.

Tankevekker

Det er en bedre historie enn «noen valgte for få bit».
Et valg av representasjon er alltid et valg gitt en sammenheng. Koden bærer ikke sammenhengen med seg. Og den varsler deg ikke når sammenhengen forsvinner.
Neste gang du skriver `int` uten å tenke: hvilken antakelse om verden gjør du akkurat nå, og hvem kommer til å arve den?

Tenk om...

I 1999 var mange redde for at datamaskiner verden over skulle slutte å virke ved årsskiftet til 2000. Man kalte det Y2K-frykten, og det ble brukt enorme summer på å forberede seg.
Hva tror du det handlet om? Og hvilken representasjon var det som hadde møtt grensen sin?
Hint: minne var dyrt på 1960- og 70-tallet, og et årstall kan skrives med to siffer hvis du tar for gitt at de to første er 1 og 9.

2.9 Tekst og tegn

Nå forlater vi tallene. Eller: vi later som. For maskinen finnes det fortsatt bare tall. Alt vi skal gjøre i denne seksjonen, er å bli enige om hvilket tall som skal bety hvilken bokstav.

2.9.1 Den enkleste ideen, og problemet med den

Du skal lagre bokstaven Ø. Du har bare tall. Løsningen er opplagt: lag en tabell.

0	A
1	B
2	C
⋮	⋮
27	Ø
28	Å

Lagrer du tallet 27, mener du Ø. Ferdig. Og dette er faktisk slik det gjøres, den dag i dag. Det finnes ingen dypere magi her.

Problemet dukker opp i det øyeblikket to maskiner skal snakke sammen.

Maskin A sender tallet 27 og mener Ø. Maskin B tar imot 27, slår opp i *sin* tabell, og finner noe helt annet. Resultatet har et navn: **mojibake**, fra japansk, omtrent «tegnforvandling». Teksten er intakt som tall, og søppel som tekst.

Dette er ikke et historisk problem. Du har sett det selv. Hver gang du limer inn tekst fra én app til en annen og æøå blir til `Ã¦Ã¸Ã¥` eller til spørsmålstegn, har du sett to tabeller som ikke er enige.

Tankevekker

Legg merke til hva en tegnkoding egentlig er: en *avtale mellom to parter om hva et bitmønster betyr*.
Det er samme type avtale som en nettverksprotokoll, og slike har du allerede jobbet med i første halvdel av emnet. Forskjellen er bare at HTTP avtaler hva en melding betyr, mens UTF-8 avtaler hva en byte betyr.
Prinsippet er det samme. Det er derfor tegnkoding hører hjemme i et kapittel om representasjon – det er den bærende ideen i sin reneste form.

2.9.2 ASCII

Det første virkelig vellykkede forsøket på en felles tabell var **ASCII**, *American Standard Code for Information Interchange*, fra 1963. Den brukte **sju bit**, som gir $2^7 = 128$ plasser.

Sju bit, ikke åtte. Dette er verdt å få med seg, for ASCII omtales ofte upresist som en åttebitskode. Den åttende biten i byten var opprinnelig ledig. Den ble gjerne brukt til *paritet*: en enkel feilsjekk der biten settes slik at antallet enere i byten alltid blir et partall. Linjene var dårlige, og en byte som kom fram med ulikt antall enere var åpenbart forvansket underveis.

	0	1	2	3	4	5	6	7
0	NUL	DLE	␣	0	@	P	‘	p
1	SOH	DC1	!	1	A	Q	a	q
2	STX	DC2	"	2	B	R	b	r
3	ETX	DC3	#	3	C	S	c	s
4	EOT	DC4	$	4	D	T	d	t
5	ENQ	NAK	%	5	E	U	e	u
6	ACK	SYN	&	6	F	V	f	v
7	**BEL**	ETB	’	7	G	W	g	w
8	BS	CAN	(	8	H	X	h	x
9	HT	EM	)	9	I	Y	i	y
A	LF	SUB	*	:	J	Z	j	z
B	VT	ESC	+	;	K	[	k	{
C	FF	FS	,	<	L	\	l	|
D	CR	GS	-	=	M	]	m	}
E	SO	RS	.	>	N	^	n	~
F	SI	US	/	?	O	_	o	**DEL**

Tabell 2.4 ASCII-tabellen. Kolonnen gir øverste heksadesimale siffer, raden det nederste. Bokstaven `K` står i kolonne 4, rad B, og har altså koden `0x4B`.

Tabellen er ikke satt sammen tilfeldig. Fire trekk er verdt å se etter.

Sifrene ligger på `0x30` til `0x39`. De fire nederste bitene i et siffertegn er dermed nøyaktig sifferverdien. Tegnet `'7'` er `0011 0111`, og de fire siste bitene er `0111`, altså 7. Det er samme mønster som BCD i seksjon 2.6.3, og det er ingen tilfeldighet. Skal du gjøre et innskrevet siffer om til et tall, trekker du bare fra 48.

Sifrene kommer før bokstavene. Vil du sjekke om et tegn er et siffer, holder det å sjekke om koden ligger mellom `0x30` og `0x39`. Én sammenligning, ingen oppslagstabell.

Store og små bokstaver ligger nøyaktig 32 fra hverandre. `A` er `0x41` og `a` er `0x61`. Siden $32 = 2^5$, skiller de seg i ett eneste bit. Å veksle mellom stor og liten bokstav er derfor ikke et oppslag – det er å snu bit nummer 5.

De 32 første plassene er ikke tegn. De er *kontrolltegn*, altså kommandoer til utstyret i den andre enden. `LF` skifter linje, `CR` flytter skrivehodet til venstre marg, `HT` er tabulator, `ESC` innleder en styresekvens. Det finnes koder for «start tekst», «slutt tekst», «mottatt» og «ikke mottatt». ASCII ble laget for teleprintere, ikke for skjermer.

Fun Fact

To av kontrolltegnene fortjener en egen historie.
BEL, kode `0x07`, fikk en fysisk bjelle på mottakermaskinen til å ringe, slik at operatøren kom bort og så hva som foregikk. Koden finnes fortsatt: skriver du `printf("\a")` i C, ber du terminalen om å pipe.
DEL, kode `0x7F`, ligger helt alene nederst i høyre hjørne, langt fra de andre kontrolltegnene. Hvorfor? Fordi `0x7F` er sju enere – alle bit satt.
Tekst ble overført på hullbånd, der hvert tegn var en kolonne av hull og ikke-hull. Slo du feil, kunne du ikke sette hullene tilbake. Men du kunne slå ut *alle* hullene i kolonnen, og dermed gjøre tegnet om til sju enere. Derfor måtte «slett dette tegnet» ha nettopp den koden.
Plasseringen i tabellen er altså en fossil etter en maskin som ikke finnes lenger.

2.9.3 Når 128 tegn ikke holder

ASCII ble laget i USA, for engelsk. Det finnes ingen Æ, Ø eller Å. Og ingen ä, é, ñ eller ß.

Den første norske løsningen var å ofre noe annet. Vi tok tegnene `[`, `\` og `]` og erstattet dem med Æ, Ø og Å. Krøllparentesene `{`, `|` og `}` ble til æ, ø og å. Nå kunne vi skrive norsk. Vi kunne bare ikke skrive C-kode lenger, siden krøllparentesene var borte.

Og en maskin uten den norske tabellen leste tallene med sin egen. Sender du «Gjøvik» fra en norsk maskin til en amerikansk, kommer det fram som

```
Gj|vik
```

Neste steg var **ISO 8859-1**, også kalt Latin-1. Ta i bruk den åttende biten, og få 256 plasser i stedet for 128. De første 128 er identiske med ASCII. De neste 128 rommer de vesteuropeiske tegnene, inkludert våre tre.

Det var en klar forbedring, og den holdt i mange år. Men den løste ikke problemet – den flyttet det. Nå fantes det femten ulike ISO 8859-varianter: én for latinsk, én for kyrillisk, én for gresk, én for hebraisk. Alle uforenlige med hverandre. Og for kinesisk, japansk og koreansk, med titusenvis av tegn, var 256 plasser aldri i nærheten av nok.

2.9.4 Unicode: kodepunkt er ikke koding

Unicode er svaret, og det består i mer enn å utvide tabellen. Unicode gjør noe som er lett å overse: den **deler ett spørsmål i to**.

1. **Hvilket tall er dette tegnet?** Svaret kalles tegnets *kodepunkt*. Det er et rent tall, uavhengig av bit og byte. Bokstaven ø har kodepunktet 248, som skrives `U+00F8` – altså `U+` etterfulgt av kodepunktet i heksadesimal.
2. **Hvordan blir det tallet til byte?** Svaret kalles en *koding*, på engelsk *encoding*. Og det finnes flere, for det samme kodepunktet.

Dette skillet er hovedpoenget i seksjonen. ASCII og Latin-1 hadde ikke noe slikt skille. Der *var* tegnets nummer det samme som byten i minnet, og derfor kunne tabellen aldri bli større enn én byte tillot.

Unicode har plass til litt over en million kodepunkter, fra `U+0000` til `U+10FFFF`. Over 150 000 av dem er tatt i bruk. Der finnes alle levende skriftsystemer, en god del utdødde, matematiske symboler og emojier. Nye tegn kommer til når verden trenger dem – eurosymbolet måtte for eksempel finne seg en plass da valutaen ble innført.

De første 128 kodepunktene er identiske med ASCII, og de første 256 med Latin-1. Det var et bevisst valg. Nå skal vi se hva det kjøpte oss.

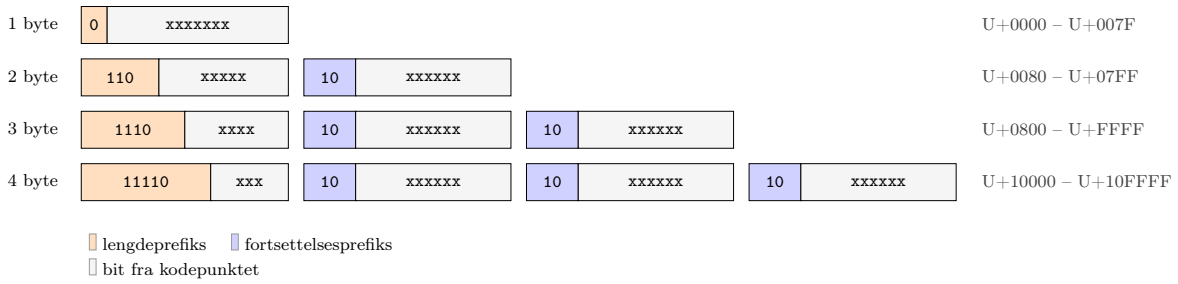
2.9.5 UTF-8

Den mest brukte kodingen heter **UTF-8**, og den er en av de smarteste konstruksjonene i faget.

Utgangspunktet er et dilemma. Skal vi bruke fire byte per tegn, så det er plass til alle kodepunkter? Da blir en helt vanlig engelsk tekstfil fire ganger så stor som nødvendig, siden nesten alle tegnene i den ligger under 128. Skal vi bruke én byte? Da får vi ikke plass til noe.

UTF-8 bruker et *variabelt* antall byte – én, to, tre eller fire. Og den løser det underliggende problemet med noen få **prefiksbit** i starten av hver byte.

Tenk på et togsett. Lokomotivet har et skilt som sier hvor mange vogner som følger etter. Hver vogn har et skilt som sier «jeg er en vogn, ikke et lokomotiv». Ser du et lokomotiv, vet du hvor langt toget er. Ser du en vogn, vet du at du står midt i et tog.



Figur 2.7 UTF-8. Første byte sier hvor mange byte tegnet består av, og hver etterfølgende byte begynner med 10.

Reglene er:

- Begynner byten med `0`, er tegnet én byte langt, og de sju siste bitene er kodepunktet. **Dette er nøyaktig ASCII.**
- Begynner byten med `110`, følger det én byte til. Med `1110` følger to. Med `11110` følger tre. Antall enere før den første nullen er antall byte.
- Begynner byten med `10`, er den en *fortsettelse* av et tegn som startet tidligere.

To konsekvenser følger, og begge er verdt å tenke gjennom.

En ren ASCII-fil er allerede en gyldig UTF-8-fil. Ikke én eneste endring trengs. All verdens engelskspråklige tekst og all eksisterende programkode ble UTF-8 den dagen standarden kom. Derfor vant UTF-8, og i dag brukes den av rundt 99 prosent av alle nettsider.

Kodingen er selvsynkroniserende. Dette er det virkelig smarte, og det er lettest å se på nettverket.

Du mottar en byte-strøm, og en pakke forsvinner underveis. Nå står du midt inne i noe, og vet ikke hvor. Med UTF-8 ser du det med én gang: begynner byten med `10`, er du inni et tegn som startet før pakketapet. Da hopper du framover til du finner en byte som ikke gjør det, og der starter neste hele tegn.

Du kan altså plukke opp tråden igjen, hvor som helst, uten å lese filen fra starten. Uten prefiksbitene ville én tapt byte ødelagt alt som kom etter.

2.9.6 Ett tegn, kodet bit for bit

La oss gjøre det konkret med ø.

Steg 1: finn kodepunktet. Bokstaven ø har kodepunktet `U+00F8`, altså 248_{10}.

Steg 2: hvor mange byte? 248 er større enn 127 og mindre enn 2048. Vi trenger toformen.

Steg 3: skriv kodepunktet binært. $248_{10} = 11111000_2$. Toformen har plass til 11 bit, så vi fyller på med nuller foran: `000 11111000`.

Steg 4: fordel bitene. Toformen er `110xxxxx 10xxxxxx`, altså fem bit i første byte og seks i andre. Vi deler `00011111000` i `00011` og `111000`:

```
første byte:  110 00011   =  11000011  =  0xC3
andre byte:    10 111000  =  10111000  =  0xB8
```

Bokstaven ø er altså `0xC3 0xB8` i UTF-8.

Åpner du en norsk tekstfil med et program som tror den er Latin-1, blir de to bytene tolket som hvert sitt tegn. `0xC3` er `Ã` og `0xB8` er `¸`. Der har du `Ã¸`-mønsteret du har sett før.

Vil du prøve en til: den smilende emojien har kodepunktet `U+1F600`. Kodepunktet trenger 17 bit, og det er mer enn de 16 treformen har plass til. Altså må vi opp i fireformen, og resultatet blir `0xF0 0x9F 0x98 0x80`.[1]

2.9.7 De andre kodingene

UTF-32 bruker fire byte for alle tegn, alltid. Det er sløsende, men enkelt: tegn nummer n ligger på byte $4n$, og du kan hoppe rett dit.

UTF-16 bruker to byte for de vanligste tegnene og fire for resten. Windows og Java bruker den internt. De fire-byte tilfellene løses med *surrogatpar*, altså to spesielle 16-bitsverdier i området `U+D800`–`U+DFFF` som sammen peker ut ett kodepunkt. Det er grunnen til at `String.length()` i Java kan gi 2 for en emoji som åpenbart er ett tegn.

Det viktige er at dette er *tre kodinger av de samme kodepunktene*. Bokstaven ø er `U+00F8` i alle tre. Bare bytene er ulike. Det er skillet mellom kodepunkt og koding som gjør den setningen mulig å si.

2.9.8 Tekst eller tall?

Her er et spørsmål som er lettere å stille enn å svare på. Du skriver 38,5 i en fil. Hvordan lagres kommaet?

Svaret avhenger av hva du holder på med, og det er nettopp poenget.

Skriver du en setning – «jeg følte meg varm, og temperaturen var 38,5» – er alt sammen tekst. Sifferet 3 er et tegn med et kodepunkt. Sifferet 8 er et tegn. Kommaet er et tegn. Sifferet 5 er et tegn. Fire tegn, fire kodepunkter, fire byte i UTF-8. Maskinen kan ikke regne med dette. Den vet ikke at det er et tall.

Lagrer du derimot verdien 38,5 for å regne med den, bruker du et flyttall, slik vi skal se i seksjon 2.10. Og da lagres kommaet *ikke i det hele tatt*. Det ligger i formatet, ikke i dataene.

Samme tall på papiret. To helt ulike ting i minnet, med helt ulike regler. Dette er den bærende ideen, i en form du møter hver gang du leser inn data fra en fil.

2.9.9 Tegn er ikke byte

Om du tar med deg én praktisk ting fra denne seksjonen, la det være denne.

Ordet «Gjøvik» har seks tegn. I UTF-8 er det sju byte, fordi ø trenger to. «Åsgårdstrand» har tolv tegn og fjorten byte. En emoji er ett tegn, fire byte i UTF-8, og to kodeenheter i UTF-16.

Det betyr at fire ting som virker like, ikke er det:

[1] Den opprinnelige definisjonen av UTF-8 tillot opptil seks byte. RFC 3629 begrenset den til fire i 2003, for å matche Unicodes øvre grense på `U+10FFFF`. Femte og sjette form er ugyldige i dag, og en dekoder skal avvise dem.

- antall tegn en bruker ser på skjermen
- antall kodepunkter i strengen
- antall kodeenheter i den kodingen språket ditt bruker internt
- antall byte strengen tar i en fil eller på nettverket

I Python ser forskjellen slik ut:

```
>>> s = "Gjøvik"
>>> len(s)
6
>>> len(s.encode("utf-8"))
7
```

Feilen som følger av å blande disse sammen, er svært vanlig. Du setter av 20 byte til et navn som kan være 20 tegn. Så kommer det inn et navn med ææå, og du kutter midt i et tegn. Det som kommer ut er en ugyldig byte-sekvens som mottakeren ikke kan tolke i det hele tatt.

Note-to-myself

Regelen er enkel. Lagre og send tekst som UTF-8. Si eksplisitt fra at det er UTF-8. Tell i tegn når du snakker med mennesker, og i byte når du snakker med filer og nettverk. Og ikke kutt en byte-sekvens uten å vite hvor tegngrensene går.

2.10 Flyttall og presisjon

Heltall er fine, men de holder ikke. Noen størrelser er for store, noen er for små, og noen har desimaler.

Avstanden til Andromedagalaksen er omtrent $2{,}4 \times 10^{19}$ kilometer. Et hydrogenatom er omtrent en tidels nanometer bredt. Og π har uendelig mange desimaler, hvorav vi som regel trenger noen få.

Alle tre har noe til felles: vi trenger ikke alle sifrene. Vi trenger noen få signifikante siffer, pluss en opplysning om hvor stort tallet er.

Det gjør du allerede selv. Du sier ikke at Norge har 5 594 340 innbyggere. Du sier 5,6 millioner. Du har beholdt to siffer og kastet resten, fordi resten uansett er utdatert i morgen.

2.10.1 Ideen

Et **flyttall** lagres som to deler:

$$\text{verdi} = \pm\,\text{mantisse} \times \text{base}^{\text{eksponent}}$$

Mantissen bærer presisjonen. Eksponenten bærer størrelsesordenen. I en datamaskin er basen 2 og ikke 10, men ideen er den samme.

Hvorfor heter det flyttall?

Navnet er faktisk beskrivende, og det er verdt å stoppe ved.

Skriver du $2{,}4 \times 10^{19}$, sier eksponenten: flytt kommaet nitten plasser til høyre. Skriver du $9{,}6 \times 10^{-8}$, sier den: flytt kommaet åtte plasser til venstre.

Kommaet står altså ikke fast. Det *flyter*, og eksponenten forteller hvor det skal ende opp. Derfor heter det flyttall, og på engelsk *floating point*.

Motsatsen finnes også: *fastkommatall*, der kommaet alltid står på samme sted. Det brukes fortsatt i enkelte systemer, for eksempel når man regner i hele øre.

Hva slags handel er dette?

Dette er viktig å forstå, for det er lett å tro at flyttall er gratis.

Har du 32 bit, har du 2^{32} mulige bitmønstre. Verken flere eller færre enn du hadde med et heltall. Du kan ikke trylle fram nye verdier.

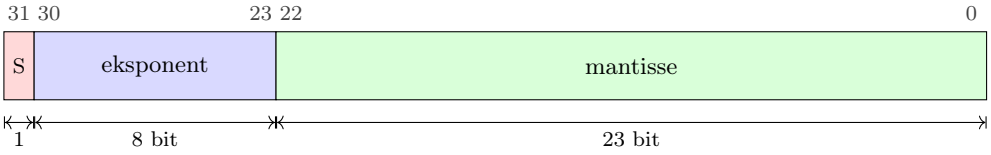
Det du gjør, er å *fordele* de samme mønstrene annerledes. I stedet for fire milliarder tett pakkede heltall får du fire milliarder verdier spredt utover et enormt område. Tett samlet nær null, gissent langt ute.

Tenk på et kart. Zoomer du ut, får du hele landet på skjermen, men du ser ikke lenger hvilken gate du står i. Kartet er like stort uansett – det er detaljene som forsvinner. Flyttall gjør nøyaktig det samme med tall.

2.10.2 IEEE 754

Nesten alle maskiner følger standarden **IEEE 754** fra 1985. Den definerer blant annet et 32-bits format, som heter `float` i C og Java, og et 64-bits som heter `double`.

Det 32-bits formatet deler bitene slik:



S – fortegn: 0 for positivt, 1 for negativt. Rent fortegn, ikke toerkomplement.

eksponent – lagret med forskyvning 127: står det 129, er eksponenten $129 - 127 = 2$.

mantisse – sifrene *etter* det ledende ettallet, som ikke lagres.

Figur 2.8 IEEE 754, enkel presisjon. 32 bit fordelt på 1, 8 og 23.

To detaljer i figuren trenger en forklaring.

Forskyvningen. Eksponenten må kunne være negativ, siden vi skal kunne lage små tall. I stedet for toerkomplement bruker IEEE 754 *forskyvning*: man lagrer den virkelige eksponenten pluss 127. Står det 127, betyr det 2^0. Står det 128, betyr det 2^1. Står det 126, betyr det 2^{-1}.

Hvorfor akkurat sånn? Fordi to positive flyttall da kan sammenlignes som om de var vanlige heltall, bit for bit, uten å pakke ut noe. Det gjør maskinvaren enklere. For negative tall må fortegnsbitet håndteres

for seg, siden IEEE 754 bruker rent fortegn og ikke toerkomplement.

Det skjulte bitet. Et normalisert binært tall har alltid formen 1,noe $\times 2^n$. Det ledende sifferet *kan ikke* være noe annet enn 1, for det finnes ikke andre siffer i base 2. Da er det ingen grunn til å lagre det. Standarden tar det for gitt, og de 23 lagrede bitene gir dermed 24 bit reell presisjon. Det svarer til omtrent sju desimalsiffer.

2.10.3 Et tall bygget opp bit for bit

La oss kode 5,75 som en `float`.

Steg 1: skriv tallet binært. Heltallsdelen og brøkdelen tas hver for seg. $5 = 101_2$. Brøkdelen 0,75 er $\frac{1}{2} + \frac{1}{4}$, altså $0{,}11_2$. Til sammen:

$$5{,}75_{10} = 101{,}11_2$$

Steg 2: normaliser. Flytt binærpunktet til det står rett etter det første ettallet, og tell hvor mange plasser du flyttet:

$$101{,}11_2 = 1{,}0111_2 \times 2^2$$

Steg 3: fyll ut feltene.

- Fortegn: tallet er positivt, så $S = 0$.
- Eksponent: den virkelige eksponenten er 2, så vi lagrer $2 + 127 = 129 = 10000001_2$.
- Mantisse: sifrene etter det ledende ettallet er `0111`, resten fylles med nuller.

```
0  10000001  01110000000000000000000
S  eksponent           mantisse
```

Grupperer vi de 32 bitene i firere og slår opp i tabell 2.1 på side 22, får vi `0x40B80000`. Skriver du ut bitmønsteret til `5.75f` i et hvilket som helst språk, er det dette du får.

Den andre veien. Hvilket tall er `1 10000100 10000111100000000000000`?

Fortegnsbitet er 1, så tallet er negativt. Den lagrede eksponenten er $10000100_2 = 132$, så den virkelige er $132 - 127 = 5$. Vi setter det skjulte ettallet foran mantissen og flytter binærpunktet fem plasser til høyre:

$$1{,}100001111_2 \times 2^5 = 110000{,}1111_2$$

Heltallsdelen 110000_2 er 48. Brøkdelen $0{,}1111_2$ er $\frac{1}{2} + \frac{1}{4} + \frac{1}{8} + \frac{1}{16} = 0{,}9375$. Svaret er $-48{,}9375$.

2.10.4 De to formatene og spesialverdiene

	Bit	Fortegn	Eksp.	Mantisse	Presisjon
`float`	32	1	8 (forsk. 127)	23	ca. 7 desimalsiffer
`double`	64	1	11 (forsk. 1023)	52	ca. 16 desimalsiffer

Tabell 2.5 De to formatene du kommer til å møte.

Standarden setter av to eksponentverdier til spesialformål. Det er derfor du av og til ser rare ting i utskrifter:

Eksponentfelt	Mantisse	Betydning
bare nuller	bare nuller	± 0 (det finnes både $+0$ og -0)
bare nuller	noe annet	denormalisert tall, svært nær null
bare enere	bare nuller	$\pm\infty$
bare enere	noe annet	`NaN` – *not a number*

Tabell 2.6 Reserverte bitmønstre i IEEE 754.

Deler du 1 på 0 i flyttallsaritmetikk, får du ingen feilmelding. Du får `Infinity`. Deler du 0 på 0, får du `NaN`.

Og `NaN` har en urovekkende egenskap: den er ikke lik seg selv. Uttrykket `x != x` er sant hvis og bare hvis `x` er `NaN`. Det er faktisk den vanligste måten å teste for `NaN` på.

2.10.5 Hvorfor 0,1 ikke finnes

Nå kommer avsnittet som forklarer en hel familie av merkelige feil du kommer til å møte som programmerer.

Prøv å skrive 0,1 binært. Metoden for brøkdeler er gjentatt multiplikasjon: gang med 2, noter heltallsdelen som neste bit, fortsett med resten.

$$
\begin{aligned}
0{,}1 \times 2 &= 0{,}2 && \to 0 \\
0{,}2 \times 2 &= 0{,}4 && \to 0 \\
0{,}4 \times 2 &= 0{,}8 && \to 0 \\
0{,}8 \times 2 &= 1{,}6 && \to 1 \quad \text{(fortsett med 0,6)} \\
0{,}6 \times 2 &= 1{,}2 && \to 1 \quad \text{(fortsett med 0,2)} \\
0{,}2 \times 2 &= 0{,}4 && \to 0 \\
0{,}4 \times 2 &= 0{,}8 && \to 0 \\
0{,}8 \times 2 &= 1{,}6 && \to 1 \\
&\vdots
\end{aligned}
$$

Vi er tilbake på 0,2 etter fem steg. Derfra gjentar alt seg, for alltid. Resultatet er

$$0{,}1_{10} = 0{,}0\overline{0011}_2 = 0{,}00011001100110011\ldots_2$$

Dette tar aldri slutt. Uansett hvor mange bit maskinen har, må den kutte et sted. Det den lagrer er ikke 0,1, men nærmeste tall den klarer å lage. For en `double` er det

$$0{,}1000000000000000055511151231257827\ldots$$

Dette er nøyaktig samme fenomen som at $\frac{1}{3}$ ikke kan skrives ferdig i desimal. $0{,}3333\ldots$ tar heller aldri slutt, og ingen synes det er maskinens feil.

Regelen er den samme i alle baser: *en brøk har et endelig antall siffer i base B bare hvis nevneren ikke har andre primfaktorer enn dem B har.* Nevneren i $\frac{1}{10}$ er 2×5. Femtallet går ikke opp i 2. Derfor er det umulig.

Legg merke til hvem som egentlig er «feil» her. Det er ikke maskinen. Det er vår vane med å skrive tall i base 10.

Advarsel :)

Konsekvensen ser du med én gang du prøver:

```
>>> 0.1 + 0.2
0.30000000000000004

>>> 0.1 + 0.2 == 0.3
False
```

Dette er ikke en feil i Python. Du får det samme i Java, C, JavaScript og alt annet som bruker IEEE 754 – altså praktisk talt alt.

2.10.6 Feil som hoper seg opp

Én avrundingsfeil er liten. Problemet er at de legger seg oppå hverandre.

```
sum = 0.0
for i in range(10):
    sum = sum + 0.1
print(sum)          # 0.9999999999999999
print(sum == 1.0)   # False
```

Ti addisjoner, og vi bommer allerede. Gjør du det ti tusen ganger, blir avviket større.

Verre blir det når du legger sammen tall av svært ulik størrelse. Det minste tallet skyves helt ut av mantissen til det største, og forsvinner sporløst.

For en `float` skjer det allerede ved $2^{24} = 16\,777\,216$. Fra og med det tallet er avstanden mellom to nabo-flyttall større enn 1, så

`16777216f + 1 == 16777216f` er *sant*.

For en `double` ligger grensen på $2^{53} \approx 9 \times 10^{15}$. Det er derfor JavaScript, som bare har `double`, har en konstant som heter `Number.MAX_SAFE_INTEGER` med verdien $2^{53} - 1$. Over den grensen kan språket ikke lenger telle nøyaktig.

Note-to-myself

Sammenlign aldri flyttall med ==.
Sjekk i stedet om de er nær nok hverandre:

```
abs(a - b) < 1e-9
```

Hvor stor toleransen bør være, avhenger av hva du regner på. Det er en vurdering du må gjøre bevisst hver gang.
Og skal du regne med penger: ikke bruk flyttall. Bruk heltall og tell i øre, eller bruk en egen desimaltype. Det var nettopp dette BCD i seksjon 2.6.3 ble laget for.

2.10.7 Avveiningen, én gang til

Hvorfor akkurat 8 eksponentbit og 23 mantissebit? Fordi noen måtte velge.

Flytter du ett bit fra mantissen til eksponenten, dobler du rekkevidden og halverer presisjonen. Flytter du andre veien, blir tallene mer presise og området mindre. Antall representerbare verdier er det samme uansett – det er 2^{32}, og det endrer seg ikke.

Dette er ikke en akademisk øvelse, og valget er ikke tatt en gang for alle. Maskinvare for kunstig intelligens gjør den samme avveiningen i dag, og svarer annerledes enn IEEE gjorde i 1985. Formatet `bfloat16` bruker 16 bit fordelt på 1 fortegnsbit, 8 eksponentbit og 7 mantissebit. Samme rekkevidde som en `float`, men langt grovere presisjon. Det viser seg at nevrale nett tåler upresise tall godt, og halvert bitbredde gir dobbelt så mye data gjennom minnebussen.

Et flyttall er altså en *tilnærming med en kjent og dokumentert feil*, ikke et eksakt tall. Vet du det, er flyttall et av de mest nyttige verktøyene i faget. Glemmer du det, får du feil som er svært vanskelige å finne.

2.11 Bilde

Tall og tekst er diskrete. De består av avgrensede enheter vi kan telle: siffer, bokstaver.

Bilder og lyd er ikke det. Virkeligheten er kontinuerlig. Mellom to farger finnes det uendelig mange mellomtoner, og mellom to øyeblikk finnes det uendelig mange øyeblikk til.

Derfor må vi nå gjøre noe vi ikke har måttet gjøre før: **kaste bort informasjon med vilje**, på en måte vi kan leve med.

2.11.1 Piksler og farge

En skjerm består av millioner av små lyspunkter, **piksler** – fra engelsk *picture element*. Hvert piksel er satt sammen av tre lyskilder: én rød, én grønn og én blå. Ved å variere styrken på de tre kan pikselet vise nesten hvilken som helst farge. Det kalles **RGB**-modellen.

Tankevekker

Her stusser mange, og med god grunn. På skolen lærte du at grunnfargene er rød, gul og blå. Og blander du all maling du har, blir det grumsete brunt, ikke hvitt.
Forskjellen er at maling *trekker fra* lys. Et rødt malingslag ser rødt ut fordi det svelger alt annet og kaster tilbake det røde. Legger du på flere lag, svelges mer, og til slutt er nesten alt borte.
En skjerm *legger til* lys. Den starter svart og skrur på lamper. Skrur du på alle tre for fullt, får du hvitt – akkurat som hvitt lys gjennom et prisme deler seg i alle regnbuens farger.
Derfor er skjermens grunnfarger rød, grønn og blå. Trykkerier, som jobber med blekk og ikke lys, bruker CMYK i stedet: cyan, magenta, gult og svart.

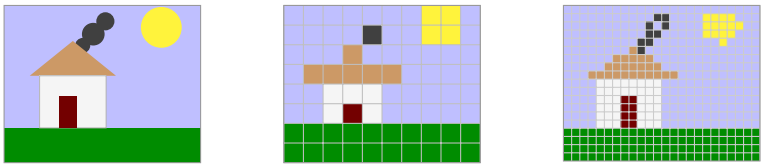
Den vanligste representasjonen gir én byte til hver komponent:

- hver komponent har verdi fra 0 (helt av) til 255 (full styrke)
- ett piksel tar tre byte, altså 24 bit
- det gir $256^3 = 16\,777\,216$ mulige farger

Dette kalles 24 bits *fargedybde*. Det er langt flere farger enn øyet kan skille, og derfor holder det. Noen formater bruker en fjerde byte til gjennomsiktighet, og profesjonell bildebehandling bruker gjerne 16 bit per komponent.

2.11.2 Rutenettet

For å lagre et bilde legger vi et rutenett over det og lagrer én farge per rute. Det er hele ideen.



motivet | 10×8 ruter | 24×19 ruter

Figur 2.9 Samme motiv, samplet på to rutenett. I det grove rutenettet har taket mistet formen, døra er nesten borte, og hele røyken er redusert til én rute. Dobler du antall ruter i begge retninger, firedobles filen.

Se hva som skjer i det grove rutenettet. En rute som dekker både himmel og tak må velge én farge, og velger den som dekker mest. At ruta egentlig inneholdt to farger, er borte for alltid.

Det er dette som kalles sampling. Vi kommer tilbake til akkurat samme grep når vi tar for oss lyd.

2.11.3 Hva koster det?

Regnestykket er enkelt: *antall piksler ganger byte per piksel.*

Et bilde på 1000×1000 piksler med 24 bits farge tar

$$1000 \times 1000 \times 3 = 3\,000\,000 \text{ byte} = 3 \text{ MB}.$$

Husk at MB her betyr 10^6 byte, ikke 2^{20} – det er forskjellen fra seksjon 2.6. I MiB ville samme tall vært 2,86.

Et bilde fra et moderne mobilkamera har gjerne 12 millioner piksler, altså 36 MB rått. Ta hundre bilder, og du har brukt 3,6 GB. Det går ikke. Derfor komprimerer vi.

2.11.4 Fargetabell

Se på motivet i figur 2.9 en gang til. Hvor mange ulike farger er det egentlig i bildet?

Sju: mørkt gress, lys himmel, sol, tak, vegg, dør og røyk.

Men vi bruker 24 bit per piksel, altså plass nok til nesten 17 millioner farger. Det er sløsing på et nivå som roper på en løsning, og løsningen er en **fargetabell**, også kalt en palett.

Ideen er den samme som et fargekart hos malingsforhandleren. I stedet for å beskrive fargen på nytt hver gang, gir du den et nummer. Nummeret er kort. Selve beskrivelsen står på kartet, én gang.

Sju farger trenger tall fra 0 til 6, og det får plass i tre bit.

Regnestykket for et bilde med 348×257 ruter:

- rått: 89 436 piksler × 24 bit = 268 308 byte
- med palett: 89 436 piksler × 3 bit = 33 539 byte, pluss 21 byte til selve tabellen

Vi har spart sju åttedeler av plassen, og *ikke mistet en eneste piksel*. Bildet er bit for bit det samme når det pakkes ut igjen. Det kalles **tapsfri komprimering**.

Men legg merke til forutsetningen. Metoden virker fordi bildet hadde få farger. På et fotografi, der nesten hver piksel har sin egen nyanse, hjelper den ingenting.

2.11.5 Løpelengdekoding

Se på bildet en gang til. Hele den nederste fjerdedelen er gress – rute etter rute etter rute med nøyaktig samme verdi.

Tenk på hvordan en strikkeoppskrift er skrevet. Det står ikke «rett, rett, rett, rett, rett, rett». Det står «6 rette». Samme informasjon, en brøkdel av plassen.

Løpelengdekoding gjør nøyaktig dette med piksler. I stedet for `gress, gress, gress, gress, gress` lagrer vi `gress × 5`. For et bilde med store ensfargede flater er gevinsten dramatisk. Motivet vårt krymper til under tre tusen byte, altså omtrent en hundredel av rå-størrelsen.

Dette er også tapsfritt, og det er en av teknikkene formatene PNG og GIF bruker.

Men forutsetningen igjen: metoden virker fordi nabopiksler ofte er like. Gi den et bilde med tilfeldig støy i hver piksel, og den gjør filen *større*, fordi hver eneste piksel får en teller på kjøpet.

All komprimering bygger på en antakelse om hva slags data den får. Ingen komprimeringsmetode kan gjøre alle filer mindre.

Note-to-myself

Komprimering er ikke gratis, og prisen er ikke bare kompleksitet i koden.
Jo smartere du pakker, jo mer arbeid må til for å pakke ut igjen. En rå fil kan leses rett inn i minnet. En komprimert fil må oversettes underveis, og det koster regnekraft hver eneste gang den åpnes.
For bilder merkes det sjelden – maskinen er rask nok. For video som skal spilles av i sanntid, eller for data som skal leses tusenvis av ganger i sekundet, er det en helt reell avveining.

2.11.6 Komprimering med tap: JPEG

For fotografier holder ikke de tapsfrie metodene. Et fotografi har ingen store ensfargede flater, og ingen liten fargepalett. Ute i naturen er det ikke «her er det lyseblått og her er det grønt» – det er nyanser hele veien.

Løsningen er å slutte å kreve at bildet skal komme tilbake uendret, og i stedet kreve at det skal *se* uendret ut. Det er en helt annen avtale, og det er den **JPEG** inngår.

JPEG bygger på forskning om hva øyet er godt og dårlig til. Vi ser skarpt på lysstyrke og slurvete på farge. Vi legger dårlig merke til fine detaljer i områder med mye struktur. JPEG deler bildet i små blokker, beskriver hvor raskt fargene varierer innenfor hver blokk, og kaster de raskeste variasjonene. Dem ser vi likevel ikke.

Et fotografi tok 40,7 MB i rå rasterform. Lagret som JPEG på full kvalitet ble det 5,5 MB, uten synlig forskjell. Ved 25 prosent kvalitet var det nede i 0,6 MB, og selv da måtte man se etter – det første som

forsvant var de jevne fargeovergangene i himmelen, som ble til synlige trappetrinn.

Legg merke til at JPEG sparte 35 MB allerede på full kvalitet, altså *før* noe ble kastet. Formatet bruker tapsfrie teknikker i tillegg til de tapsbærende. Det er ikke enten eller.

Advarsel :)

Informasjon som er kastet, er kastet. Lagrer du et bilde som JPEG på 25 prosent kvalitet, kommer det aldri tilbake. Uansett hvilket program du åpner det i.
Verre: lagrer du et JPEG-bilde på nytt som JPEG, kastes det litt mer. Redigerer og beskjærer du mellom hver lagring, er bildet merkbart nedbrutt etter noen få runder. Det kalles generasjonstap, og det er derfor arkiver, medisinske bilder og bevismateriale lagres tapsfritt.
Bruk JPEG til fotografier du skal se på. Bruk PNG til strektegninger, skjermbilder og alt du skal redigere videre.

2.11.7 Vektorgrafikk – en helt annen idé

Alt vi har sett til nå er *rastergrafikk*: et rutenett av piksler. Det finnes et alternativ som bygger på en helt annen tanke.

Vektorgrafikk lagrer ikke bildet. Den lagrer *oppskriften på å tegne det*. Flytt til dette punktet. Trekk en linje dit. Tegn en sirkel med denne radien. Fyll flaten med denne fargen.

Fordelene følger direkte. En sirkel trenger tre tall – senter i x og y, pluss radius – uansett hvor stor den er på skjermen. Og siden oppskriften kjøres på nytt hver gang bildet vises, kan du zoome inn så mye du vil uten at det blir hakkete. Maskinen tegner bare sirkelen større.

Ulempen er like tydelig. Du kan ikke skrive en oppskrift på et fotografi av et fjell. Vektorgrafikk passer for logoer, ikoner, diagrammer, arkitekttegninger og skrifttyper. Ikke for noe et kamera har tatt.

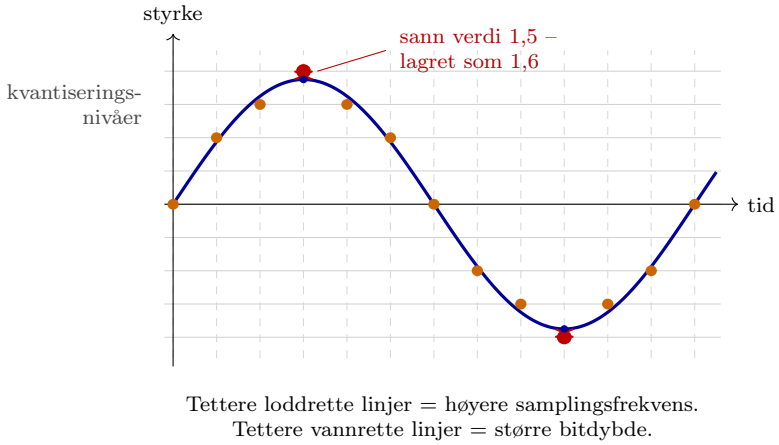
Alle figurene i denne boka er for øvrig vektorgrafikk. Det er derfor de fortsatt er skarpe når du zoomer inn i PDF-en.

2.12 Lyd

Lyd er trykkbølger i luft. En mikrofon gjør dem om til en varierende elektrisk spenning, og det er den spenningen vi skal lagre.

Signalet er kontinuerlig i to retninger samtidig. Det varierer i tid, og det varierer i styrke. For å få det inn i en datamaskin må vi gjøre begge deler diskrete. De to grepene har hvert sitt navn:

- **Sampling** deler opp *tidsaksen*. Vi måler signalet med jevne mellomrom, og glemmer alt som skjer imellom.
- **Kvantisering** deler opp *styrkeaksen*. Vi har bare et endelig antall nivåer å velge mellom, og hver måling rundes til nærmeste.



Figur 2.10 Sampling og kvantisering. De oransje punktene er det maskinen lagrer. Der kurven treffer mellom to nivåer, går differansen tapt – se de to røde punktene.

2.12.1 Det er samme grep som på bildet

Stopp ved denne figuren et øyeblikk, og sammenlign den med pikselrutenettet i figur 2.9 på side 45. Det er nøyaktig samme operasjon.

I bildet la vi et rutenett over noe kontinuerlig og lagret én verdi per rute. I lyden legger vi et rutenett over noe kontinuerlig og lagrer én verdi per rute. Forskjellen er bare at bildets rutenett går i to romlige retninger, mens lydens går i tid og styrke.

Konsekvensen er den samme begge steder: *det som lå mellom rutene, finnes ikke lenger*. Et finere rutenett gir et bedre resultat og en større fil. Samme handel, to ulike signaler.

2.12.2 Hvor tett må vi sample?

Svaret er ikke «så tett som mulig», og det er ikke en smakssak heller. Det finnes et presist matematisk svar, og det kalles **samplingsteoremet** eller Nyquist-kriteriet:

Tankevekker

For å kunne gjenskape et signal fullstendig må du sample med minst *dobbelt* så høy frekvens som den høyeste frekvensen i signalet.

Det er et sterkt resultat: over den grensen mister du *ingenting*. Sampler du raskere enn det dobbelte, kan det opprinnelige signalet regnes eksakt tilbake.

Et ungt menneske hører opp til omtrent 20 000 Hz. Dobbelt så mye er 40 000, og CD-standarden valgte 44 100 Hz – litt over, for å ha margin til filtrene.

Sampler du for sakte, får du et *feil* signal. Høye frekvenser folder seg ned og dukker opp som lave toner som aldri var der. Det er samme fenomen som gjør at bilhjul kan se ut til å snurre baklengs på film: kameraet tar 24–25 bilder i sekundet, og hjulet går for fort.

2.12.3 CD-en, regnet ut

CD-standarden er verdt å regne gjennom, fordi den viser hvordan tallene henger sammen.

- samplingsfrekvens: 44 100 sampler per sekund
- bitdybde: 16 bit per sample, altså $2^{16} = 65\,536$ nivåer
- kanaler: 2, altså stereo

Datarate:

$$44\,100 \times 16 \times 2 = 1\,411\,200 \text{ bit/s} = 176\,400 \text{ byte/s}$$

En CD rommer 74 minutter, altså 4440 sekunder:

$$4440 \times 176\,400 = 783\,216\,000 \text{ byte} \approx 783 \text{ MB}$$

Note-to-myself

Tallet 783 MB er *ren lyddata*. Feilkorreksjonen er ikke med.
En CD skal tåle riper og fettete fingre, så den lagrer også betydelige mengder feilkorreksjonsdata. De kommer *på toppen* av de 783 megabytene, på det fysiske laget, sammen med rammestruktur og synkroniseringsmønstre. Platen inneholder altså fysisk atskillig mer enn 783 MB.
Det er også derfor en data-CD bare rommer rundt 680 MB der en lyd-CD rommer 783. Den trenger strengere feilkorreksjon enn lyd gjør. En tapt bit i en musikkfil gir et knepp du knapt hører. En tapt bit i et program kan gjøre det ubrukelig, og prisen for den forskjellen er plass.

2.12.4 Komprimering av lyd

Som med bilder finnes begge slag.

Tapsfritt, som FLAC, utnytter mønstre i signalet. To eksempler:

Venstre og høyre kanal i en stereoinnspilling er som regel nesten like. I stedet for å lagre begge, lagrer man den ene pluss *differansen* til den andre. Differansen er et lite tall og trenger få bit.

Samme triks virker i tid. To nabomålinger ligger som regel nær hverandre, siden lyd er en jevn kurve. Så i stedet for hele verdien kan man lagre endringen fra forrige måling.

Tankevekker

Men differanser har en ubehagelig bieffekt. Lagrer du *bare* endringer, må du lese hele filen fra start for å vite hvor du er. Hopper du til midten av en sang, har du en haug med endringer og ingen anelse om hva de er endringer fra.
Løsningen er å legge inn faste punkter der hele verdien står, med jevne mellomrom. Da kan spilleren hoppe til nærmeste slike punkt og lese videre derfra.
Kjenner du igjen problemet? Det er nøyaktig det samme som UTF-8 løser med prefiksbitene i seksjon 2.9. En strøm av data der du må kunne plukke opp tråden hvor som helst, uten å lese alt som kom før.
To helt ulike domener, samme problem, samme type løsning. Det er verdt å merke seg – du kommer til å se dette mønsteret igjen.

Med tap, som MP3 og AAC, går mye lenger. De utnytter begrensninger i hørselen vår:

- vi hører ikke under omtrent 20 Hz eller over omtrent 20 000 Hz
- en kraftig lyd *maskerer* svakere lyder på nærliggende frekvenser – de er der, men vi hører dem ikke
- rett etter en kraftig lyd er øret mindre følsomt en liten stund

Dette kalles *psykoakustikk*. Legg merke til hva algoritmen egentlig gjør: den fjerner det som er uviktig for *et menneske*, ikke det som er uviktig for signalet. Representasjonen er tilpasset mottakeren.

Gevinsten er stor. Der en CD bruker 1411 kbit/s, gir MP3 akseptabel kvalitet på 128–192 kbit/s. Moderne formater som Opus gir CD-liknende kvalitet på rundt 96 kbit/s, altså under en tiendedel. Uten det hadde musikkstrømming ikke vært mulig.

Advarsel :)

Samme regel som for JPEG gjelder her. Skal du jobbe med lyd – ta opp, klippe, mikse – er MP3 et elendig arbeidsformat. Hver gang du lagrer på nytt, kastes mer, og du kommer aldri tilbake til originalen.
Jobb i et ukomprimert format som WAV, eller et tapsfritt som FLAC. Lag MP3 til slutt, når du er ferdig, og bare til folk som skal lytte.

2.13 Oppsummering

Vi har vært gjennom en lang rekke representasjoner. La oss stille samme spørsmål til alle: *hva koster den, og hvor går grensen?*

Representasjon	**Hva den koster**	**Hvor grensa går**
Heltall uten fortegn, n bit	n bit	$2^n - 1$; over det ruller tallet rundt
Toerkomplement, n bit	n bit, halve området går til negative tall	-2^{n-1} til $2^{n-1} - 1$; usymmetrisk, og -2^{n-1} kan ikke negeres
BCD	6 av 16 mønstre kastes; treg aritmetikk	eksakt i desimal, men bruker et helt siffer mer plass ved 20 bit
IEEE 754 `float`	32 bit fordelt 1/8/23	ca. 7 signifikante siffer; 0,1 finnes ikke; over 2^{24} kan den ikke telle
ASCII	7 bit per tegn	128 tegn; ingen æ, ø, å
Latin-1	8 bit per tegn	256 tegn; én region om gangen
UTF-8	1–4 byte per tegn	alle kodepunkter; men tegn er ikke byte, og du kan kutte midt i et tegn
RGB 24 bit	3 byte per piksel	16,7 millioner farger; filen blir stor
Fargetabell	indeks per piksel + tabell	bare så mange farger som tabellen har
Løpelengdekoding	tapsfri, avhenger av dataene	gjør støyete bilder *større*
JPEG	sterk komprimering med tap	informasjonen kommer aldri tilbake; generasjonstap
Sampling og kvantisering	samplingsfrekvens × bitdybde × kanaler	Nyquist setter nedre grense; under den folder frekvenser seg
MP3, AAC, Opus	en tiendedel av CD-raten	tilpasset menneskeøret, ikke signalet

Tabell 2.7 Alle representasjonene i kapitlet, med prisen og grensen.

Ser du på kolonnene, ser du hele kapitlet. Ingen av representasjonene er feil. Hver av dem er et svar på hva som er viktigst akkurat her: presisjon eller plass, rekkevidde eller nøyaktighet.

Og hver av dem har en grense. Ariane 5 traff grensen til et 16-bits heltall. En bank som regner renter i `float`, treffer grensen til IEEE 754. Et skjema som kutter et navn etter 20 byte, treffer grensen til

UTF-8. I alle tre tilfellene virket koden helt til den ikke gjorde det.

Tankevekker

Vil du ha den bærende ideen i sin mest håndfaste form, prøv dette.
Ta en bildefil og be lydspilleren din om å spille den av. Det går helt fint. Det blir neppe vakkert – det høres ut som en fax-maskin i dårlig humør – men maskinen nekter ikke. Den leser bitene og gjør dem om til lyd, slik du ba om.
Gjør det motsatte også. Åpne en musikkfil i et bildeprogram, og du får et bilde. Meningsløst, men et bilde.
Det er tross alt bare bit, alt sammen. Det eneste som skiller et bilde fra en sang, er hva du har bestemt deg for at bitene skal bety.

I neste kapittel tar vi tråden videre på det mest overraskende punktet i figur 2.1 på side 15: at det samme bitmønsteret også kan være en *instruksjon*. Det er ideen som gjør en datamaskin til noe annet enn en kalkulator.

2.14 Videre lesning

Stoffet i dette kapitlet dekkes også i pensumboka av Englander. Det er verdt å lese der, for framstillingen er en annen, og to vinklinger på samme stoff er ofte det som får det til å sitte.

Tallsystemer og konvertering: kapittel 3, særlig 3.2 om telling i ulike baser, 3.4 om konverteringsmetodene og 3.5 om baser som er beslektet. Avsnitt 3.6 om brøker går dypere enn vi har gjort her, og er nyttig hvis du vil forstå flyttall bedre.

Fortegnstall, BCD og flyttall: kapittel 5. Avsnitt 5.1 dekker BCD. Avsnitt 5.2 går gjennom fortegn-og-magnitude, enerkomplement og toerkomplement med en mer matematisk begrunnelse enn vår klokkeanalogi. Avsnitt 5.3 bygger opp flyttallsformatet steg for steg, med et pedagogisk desimalformat før IEEE 754.

Tegn, bilde, lyd og komprimering: kapittel 4, særlig 4.2 om tegnkoder, 4.3 om visuelle data og 4.4 om lyd. Avsnitt 4.7 om internt dataformat er en god parallell til diskusjonen vår om at typen ikke ligger i bitene.

Videre i denne boka: kapittel 3 tar opp igjen tanken om at et bitmønster kan være en instruksjon. Kapittel 5 viser hvordan en krets faktisk legger sammen to toerkomplementtall. Kapittel 6 går helt ned til det fysiske bitet.

Vil du videre på egen hånd: se på Huffman-koding, som bygger en optimal tapsfri kode ut fra hvor ofte hvert symbol forekommer. Og på Shannons kildekodingsteorem, som setter en teoretisk nedre grense for hvor mye en gitt datastrøm kan komprimeres. Det siste er et av de vakreste resultatene i informatikken, og det forklarer hvorfor ingen komprimeringsalgoritme kan gjøre alle filer mindre.

2.15 Refleksjon

Tenk over disse spørsmålene, gjerne sammen med noen andre:

- **Representasjon i hverdagen.** Velg en digital dings du bruker hver dag. Hvor har du merket grensene for representasjonene den bruker – i lydkvalitet, bildekvalitet, batteriprosent, tidsangivelser? Har du noen gang sett et program oppføre seg rart på en måte du nå har en forklaring på?

- **Penger i heltall.** Du skal skrive et system som håndterer transaksjoner i flere valutaer. Hvilke problemer får du hvis du lagrer beløp som flyttall? Og hvilke får du hvis du lagrer dem som heltall antall øre? Hva gjør du med en valuta som ikke har hundredeler, eller en renteberegning som gir en tredjedels øre?
- **Hvem sitt alfabet?** Før Unicode var det vanskelig eller umulig å skrive mange av verdens språk på en datamaskin. Hva slags konsekvenser kan det ha hatt for hvilke språk som ble brukt på nett, og for hvem som fikk delta? Unicode måtte også ta stilling til hvilke tegn som er «samme tegn» på tvers av kinesisk, japansk og koreansk. Hvorfor er slike avgjørelser politiske og ikke bare tekniske?
- **Når er tap greit?** Sammenlign tapsfri og tapsbærende komprimering for fire tilfeller: medisinske røntgenbilder, musikk på en strømmetjeneste, bilder i en rettssak, og et ferieminne på mobilen. Hva bør avgjøre valget i hvert tilfelle, og hvem bør bestemme?
- **Arvede antakelser.** Ariane 5 falt fordi kode som var riktig for én rakett ble brukt på en annen. Kan du komme på et tilfelle fra ditt eget liv eller fra nyhetene der en antakelse om verden lå skjult i et system og sluttet å stemme? Hvordan kunne man oppdaget det på forhånd?

2.16 Selv-test

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

1. Hva betyr det at et tallsystem er *posisjonelt*? Forklar med utgangspunkt i tallet 407_{10} hva vekten til hvert siffer er.
2. Konverter til desimal: (a) 10110_2, (b) $2\mathrm{F}_{16}$, (c) 351_8.
3. Konverter 200_{10} til (a) binær, (b) heksadesimal, (c) oktal. Bruk divisjonsmetoden i minst ett av tilfellene, og vis mellomregningen.
4. Skriv 0xB3 som binært tall uten å gå veien om desimal. Forklar i én setning hvorfor du kan slippe unna med det.
5. Skriv 1011010_2 som heksadesimalt tall. Hva må du gjøre med den øverste gruppa, og hvorfor er det lov?
6. En Linux-fil har rettighetene `750`. Skriv ut de ni rettighetsbitene, og forklar med ord hva eier, gruppe og andre har lov til.
7. Hvor mange bit trenger du som et minimum for å kunne representere alle tall fra 0 til og med 1000? Begrunn svaret.
8. Hvilket desimaltall er bitmønsteret `11101000` (a) hvis det leses som et heltall uten fortegn, og (b) hvis det leses som toerkomplement?
9. Skriv -37 som et åttebits toerkomplementtall. Vis begge stegene i negeringen.
10. Regn ut $-37 + 20$ i åttebits toerkomplement ved å legge sammen bitmønstrene. Kontroller svaret i desimal.
11. Hvorfor finnes det ingen -0 i toerkomplement, mens det finnes to nuller både i fortegn-og-magnitude og i enerkomplement? Hvilken praktisk fordel gir det?

12. En `short` i Java er 16 bit. Hva blir resultatet av `(short)(30000 + 10000)`, og hvorfor? Forklar hva som skjer med bitene.

13. En venn sier: «Python har ikke overflyt, så da slipper jeg å tenke på det.» Hva er riktig og hva er galt i den påstanden?

14. Hvor mange byte tar strengen `Bodø"` i (a) UTF-8 og (b) UTF-32? Hvor mange tegn består den av? Forklar hvorfor de tre tallene er forskjellige.

15. En byte-strøm inneholder `0xE2 0x82 0xAC`. Hvor mange tegn er dette i UTF-8, og hvordan ser du det uten å slå opp i noen tabell?

16. Du skriver temperaturen 38,5 inn i et program. Forklar hva som lagres hvis verdien havner i en tekststreng, og hva som lagres hvis den havner i en `double`. Hvor er kommaet i hvert av tilfellene?

17. Kod tallet $-6{,}5$ som en 32-bits IEEE 754 `float`. Vis alle stegene: binær form, normalisering, eksponent med forskyvning, og de tre feltene.

18. Hvorfor gir uttrykket `0.1 + 0.2 == 0.3` usant i praktisk talt alle programmeringsspråk? Og hvorfor gir `0.5 + 0.25 == 0.75` sant?

19. Et ukomprimert bilde er 1920×1080 piksler med 24 bits farge. (a) Hvor mange byte tar det? (b) Bildet inneholder bare 16 ulike farger – hvor mye sparer du på å bruke en fargetabell, og er det tapsfritt?

20. En lydopptaker sampler 48 000 ganger i sekundet med 24 bits dybde i stereo. Hvor mange byte blir ett minutt? Og hva er den høyeste frekvensen opptaket kan gjengi korrekt?

21. Både JPEG og MP3 kaster informasjon. Forklar hva de to har til felles i *hvorfor* de kan gjøre det, og gi ett eksempel på en situasjon der ingen av dem burde brukes.

22. Se tilbake på figur 2.1 på side 15. Du får utdelt en fil på disken, og får vite at den er 4 byte stor. Hva kan du si om innholdet, og hva kan du ikke si? Hvor må opplysningen komme fra?

Kapittel 3

Instruksjoner

Innhold

3.1 Læringsmål . . . 56
3.2 Fra språk til maskin . . . 56
3.3 Rommet med hundre postkasser . . . 57
3.4 Ordren som et tall . . . 59
3.5 Hent og utfør . . . 61
3.6 Instruksjonssettet og de første programmene . . . 64
3.7 Assembler: navn i stedet for tall . . . 66
3.8 Kontrollflyt . . . 67
3.9 Adresser, etiketter og data . . . 70
3.10 Oppsummering . . . 74
3.11 Videre lesning . . . 75
3.12 Refleksjon . . . 75
3.13 Selv-test . . . 76

Et program er en tekst du skriver. En prosessor er et stykke silisium. Mellom de to ligger dette kapitlet.

I 1930-årenes USA fantes det rom fulle av mennesker som regnet. De ble kalt *computers*, for det var det ordet betydde den gangen: en som regner. De fleste av dem var kvinner. Hver av dem hadde en mekanisk regnemaskin, en bunke papir og en liste med instruksjoner. Listen kunne si: ta tallet i rad 17, gang med 2,71828, rund av til fire desimaler, skriv svaret i rad 18.

Ingen av dem trengte å vite hva tabellen skulle brukes til. De fulgte listen. Det er nettopp derfor arbeidet lot seg overta av en maskin.

Tankevekker

Et program er tall i minnet. Maskinen kan ikke se forskjell på et tall den skal regne med og et tall den skal adlyde. Forskjellen ligger bare i hvordan tallet blir brukt.

I kapittel 2 så vi at et bitmønster ikke betyr noe uten en tolkning. Nå møter vi den viktigste tolkningen av dem alle. Maskinen leser et tall og gjør det tallet sier.

3.1 Læringsmål

Etter dette kapitlet skal du kunne

- **forklare** hva som skjer med en Python-linje på veien ned til maskinen
- **dele** en instruksjon i operasjonskode og adressedel
- **gjøre rede for** hent-utfør-syklusen og hva programtelleren peker på
- **spore** et LMC-program celle for celle og forutsi hva det skriver ut
- **skrive** små programmer med LMC sitt instruksjonssett
- **oversette** mellom maskinkode og assemblerkode
- **bruke** betingede hopp til å bygge løkker og valg
- **finne** feilen i en løkke som ikke terminerer
- **skille** en adresse fra verdien som ligger der
- **forklare** hvorfor et program kan endre seg selv, og hva det koster

3.2 Fra språk til maskin

Du har skrevet Python. Du har skrevet noe slikt som dette:

```
svar = a + b
```

Én linje. Men prosessoren i maskinen din kan ikke lese den linja. Den kan ikke lese bokstaver i det hele tatt.

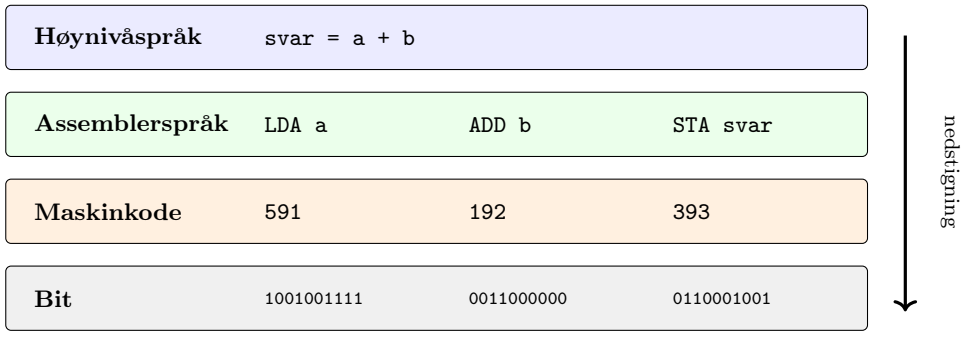
Så noe må skje på veien. Det som skjer, er en nedstigning gjennom flere språk.

3.2.1 Stigen

Øverst ligger *høynivåspråk* som Python, Java og C. De er laget for mennesker. Du skriver `a + b`, ikke hvor `a` ligger i minnet.

Nederst ligger *maskinkode*. Det er tall. Ingenting annet.

Mellom dem ligger *assemblerspråk*. Det er maskinkode med navn på. I stedet for tallet 5 skriver du `LDA`. Ett navn per instruksjon.



Figur 3.1 Den samme beregningen på fire nivåer. Hvert trinn ned fjerner et navn og legger til et tall. Nederste rad er de tre tallene skrevet binært.

3.2.2 Hvem gjør nedstigningen?

Det finnes to hovedmåter.

En *kompilator* oversetter hele programmet én gang, før det kjøres. Resultatet er en fil med maskinkode. C og Rust gjør det slik.

En *tolk* tar programmet linje for linje mens det kjører. Python gjør i hovedsak det. Kildekoden blir først oversatt til *bytekode*, en mellomform. Så er det et annet program, selve tolken, som leser bytekoden og gjør det den sier. Tolken er selv skrevet i C og finnes som maskinkode.

Det betyr at det aldri finnes maskinkode for *din* Python-linje. Det finnes maskinkode for tolken, og tolken adlyder linja di.

Note-to-myself

Forskjellen ligner den mellom å få en bok oversatt én gang for alle, og å ha en tolk ved siden av seg som oversetter setning for setning mens du leser. Den ferdige oversettelsen er raskere å lese. Tolken er raskere å komme i gang med.

Et *assemblerprogram* gjør den siste og enkleste oversettelsen. Det bytter `LDA` med 5 og et navn med en adresse. Ett navn inn, ett tall ut.

3.2.3 Hvorfor vi går helt ned

Det er ikke sikker du kommer til å skrive assemblerkode igjen, men nedstigningen er likevel verdt turen av flere grunner.

Den første er at abstraksjonene lekker. Når Python-programmet ditt er uforklarlig tregt, eller krasjer med en feilmelding om minne, ligger svaret under.

Den andre er at sikkerhetshull bor her nede. Et program som skriver til feil adresse, gjør nøyaktig det det fikk beskjed om. Skaden ligger i beskjeden, og beskjeden er et tall.

Den tredje er at hele resten av boka bygger på dette kapitlet. Kapittel 4 viser kretsene som utfører instruksjonene. Kapittel 5 viser portene de er bygd av.

Nederst på stigen ligger en liste med svært små ordrer. Hver ordre er et tall.

Fun Fact

Objektorientert programmering ble oppfunnet i Oslo. Ole-Johan Dahl og Kristen Nygaard utviklet Simula i 1960-årene. Klasser, objekter og arv kom derfra, og gikk videre til blant annet C++, Java og Python. Begge var professorer ved Universitetet i Oslo. Det er også derfor informatikk bygningene heter Ole Johan Dahls hus og Kristen Nygaards hus.

3.3 Rommet med hundre postkasser

Vi skal bruke en lekemaskin. Den heter *Little Man Computer*, forkortet LMC, og den ble laget av Stuart Madnick ved MIT. Simulatoren vi bruker er laget av Peter Higginson. Den er teoretisk korrekt og den er bare veldig liten. Hovedformålet er å få en forenklet inngang til hvordan prosessering i maskinvare og instruksjoner henger sammen.

Det som gjør LMC brukbar, er at den regner i titallssystemet. En ekte maskin gjør alt i binært, og da

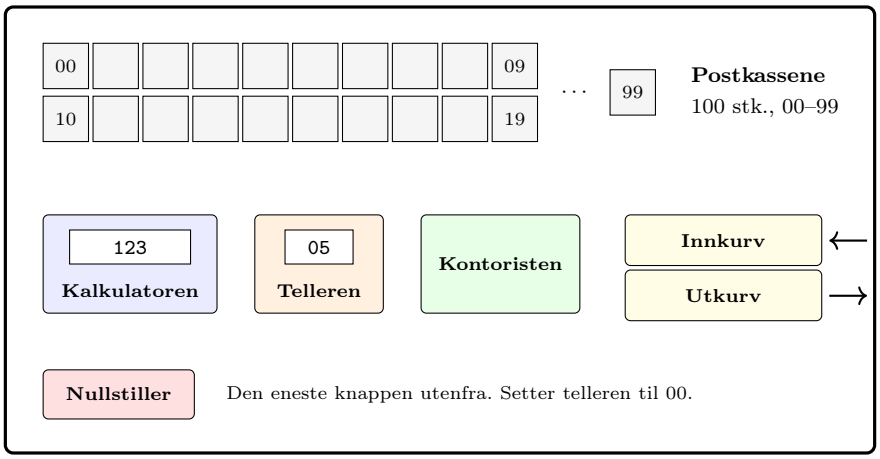
må du tenke på to ting samtidig: hvordan en instruksjon virker, og hvordan den er kodet i bit. LMC lar deg utsette det ene. Kodingen tar vi i kapittel 4.

Se for deg et lukket rom, som i figur 3.2. Langs veggen henger hundre postkasser. De er nummerert fra 00 til 99. Hver postkasse rommer én lapp, og på lappen står det et tresifret tall.

I rommet arbeider en kontorist. Kontoristen har fire ting til rådighet:

- **En kalkulator.** Den kan legge sammen og trekke fra. Displayet viser tre sifre.
- **En teller.** Den viser to sifre, og kontoristen kan klikke den ett hakk opp eller stille den direkte.
- **En innkurv.** Her legger noen utenfra lapper med tall.
- **En utkurv.** Her legger kontoristen lapper med tall.

Utenfor rommet sitter én knapp: en nullstiller som setter telleren til 00 og starter kontoristen på nytt. Bortsett fra den knappen og de to kurvene er rommet lukket.



Alt som går inn og ut av rommet, er tresifrede tall.

Figur 3.2 LMC-rommet. Kontoristen kan bare det som finnes i dette rommet.

3.3.1 Hvorfor akkurat hundre postkasser?

Fordi en adresse skal få plass i to sifre. To sifre gir tallene 00 til 99, altså hundre muligheter[1]. Flere postkasser ville krevd et tredje siffer i adressen, og da hadde det ikke vært plass til noe annet i en tresifret instruksjon.

Det er samme type regnestykke som i kapittel 2, der vi spurte hvor mange bit et tall trenger. Her er det sifre i stedet for bit, men avveiningen er den samme. Adressefeltet koster plass (bits) og antall plasser (bits) er alltid stor konkurranse om.

3.3.2 Innhold er ikke adresse!

Postkasse 42 er et sted. Tallet 42 er en verdi. De har ingenting med hverandre å gjøre.

[1] Merk at vi teller 0 med som det første tallet og da er 0-99 faktisk 100 tall :)

Tenk på en pakkeboks på et utleveringssted. Boksnummeret forteller hvor du skal gå. Det forteller ingenting om hva som ligger i boksen.

Postkasse 42 kan inneholde tallet `542`. Postkasse 542 finnes ikke. Og tallet 42 kan ligge i hvilken som helst av de hundre postkassene.

Advarsel :)

Når vi skriver `ADD 99`, er 99 en adresse. Instruksjonen betyr «legg til det som ligger i postkasse 99», ikke «legg til 99». LMC har ingen måte å si «legg til tallet 99» direkte. Skal du legge til 99, må tallet 99 først ligge i en postkasse.

3.3.3 Den regelen som bærer resten av kapitlet

Hver postkasse inneholder et tresifret tall. Ingenting ved postkassen forteller om tallet er noe å regne med eller en ordre å følge.

Kontoristen kan ikke se forskjell. Det finnes ingen forskjell å se.

Hva tallet er, avgjøres av én ting: om kontoristen kommer dit for å hente en ordre, eller for å hente en verdi. Det er hele forskjellen, og den ligger ikke i tallet.

Note-to-myself

Vi kommer tilbake til denne regelen fire ganger i dette kapitlet. Første gang som en advarsel om hvor data skal ligge. Andre gang som forklaringen på hvorfor et program uten stoppinstruksjon oppfører seg vilt. Tredje gang som et triks. Fjerde gang som et sikkerhetsproblem.

3.3.4 Simulatoren

LMC finnes som en simulator du kan kjøre i nettleseren, laget av Peter Higginson. Der ser du postkassene som et rutenett, og du kan taste inn tall direkte i cellene. Du kan kjøre programmet med `Run`, eller ett steg om gangen med `Step`.

Bruk `Step` i begynnelsen. Hele poenget med LMC er at du kan se hva som skjer.

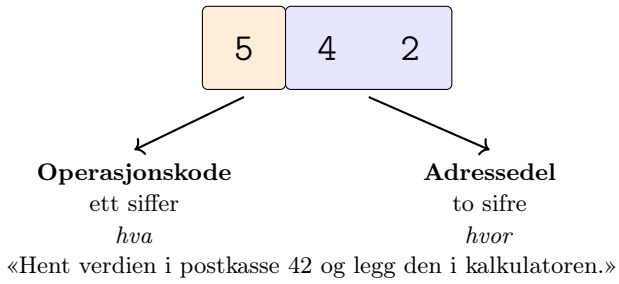
Advarsel :)

Husk `Reset` mellom to kjøringer. Glemmer du den, starter maskinen der forrige kjøring sluttet, og da stemmer ingenting. Dette er den vanligste kilden til forvirring de første timene med LMC.

3.4 Ordren som et tall

En instruksjon i LMC er et tresifret tall. Det er alt en instruksjon er.

Tallet leses i to deler, slik figur 3.3 på neste side viser. Det første sifferet sier *hva* som skal gjøres. De to siste sier *hvor*.



Figur 3.3 Instruksjonen 542, delt i de to feltene sine.

Operasjonskoden kalles ofte bare *opkode*, fra engelsk *opcode*. Hvilket tall som betyr hva, er et valg noen har tatt. Det følger ikke av noe. Den som designer maskinen, bestemmer.

Note-to-myself

I LMC har vi ett siffer for *opkode* som betyr at vi kan bare gjøre 10 ulike operasjoner (instruksjoner). Hva de forskjellige tallene betyr kommer vi til i kapittel 3.6 på side 64.

3.4.1 Tre eksempler

`542` er opkode 5 og adresse 42. Hent verdien i postkasse 42.

`199` er opkode 1 og adresse 99. Legg verdien i postkasse 99 til det som står i kalkulatoren.

`000` er opkode 0. Stopp.

Legg merke til at `000` også har en adressedel. Den er bare null, og den brukes ikke. Stoppinstruksjonen bryr seg ikke om adressen.

Note-to-myself

Fordi adressedelen ignoreres, stopper *alle* tall som begynner på null. `017` stopper maskinen like effektivt som `000`. Vi skriver alltid `000`, men det er sifferet lengst til venstre som gjør jobben.

3.4.2 Adressen som ikke er en adresse

Noen instruksjoner trenger ikke en postkasse i det hele tatt, f.eks. å lese et tall fra innkurven. Det finnes ingen adresse å oppgi. Kurven er kurven!

Da står adressedelen ledig, og den blir brukt til noe annet: å skille mellom flere instruksjoner som deler samme opkode.

- `901` leser et tall fra innkurven.
- `902` skriver tallet i kalkulatoren til utkurven.
- `922` skriver ut ett tegn.

Alle tre har opkode 9. Tallene 01, 02 og 22 forlenger opkoden, og ingen av dem viser til en postkasse.

Advarsel :)

`901` betyr ikke «gjør noe med postkasse 01». Ingen postkasse er innom. Hvis du leser `901` som en vanlig instruksjon med adresse, blir alt annet i dette kapitlet vanskeligere enn det trenger å være.

I en ekte maskin brukes det samme trikset, og der er det mindre vilkårlig enn det ser ut. Adressefeltet i en inn- og ut-instruksjon inneholder gjerne nummeret til en enhet. Skjermen har ett nummer, disken et annet. Adressen peker fortsatt på noe, bare ikke på minnet.

3.4.3 Hvor mange ordrer finnes det?

LMC bruker ni opkoder: 0, 1, 2, 3, 5, 6, 7, 8 og 9. Opkode 4 står ledig i denne utgaven av maskinen.

Instruksjonene er litt flere enn opkodene, fordi opkode 9 dekker tre stykker. Til sammen blir det elleve.

Til sammenligning har prosessoren i maskinen din flere tusen instruksjoner. Elleve er lite, men elleve er nok. Vi kommer til å regne, velge og gjenta med dem alle sammen.

Tankevekker

Kraften i en datamaskin kommer ikke av at operasjonene er avanserte. Den kommer av at svært enkle operasjoner utføres svært mange ganger i sekundet. Alt det kompliserte ligger i rekkefølgen.

3.4.4 Det samme, i bit

LMC teller i titallssystemet. En ekte maskin gjør ikke det.

I en ekte maskin er instruksjonen et bitmønster, og de to feltene er grupper av bit i stedet for sifre. Avveiningen blir den samme, men vesentlig skarpere. Hvor mange bit skal opkoden ha, og hvor mange skal adressen ha? Flere bit til opkoden gir flere instruksjoner. Flere bit til adressen gir mer minne å peke på.

Den regningen tar vi i kapittel 4. Her holder det å vite at delingen i to felt er den samme, uansett tallsystem.

3.5 Hent og utfør

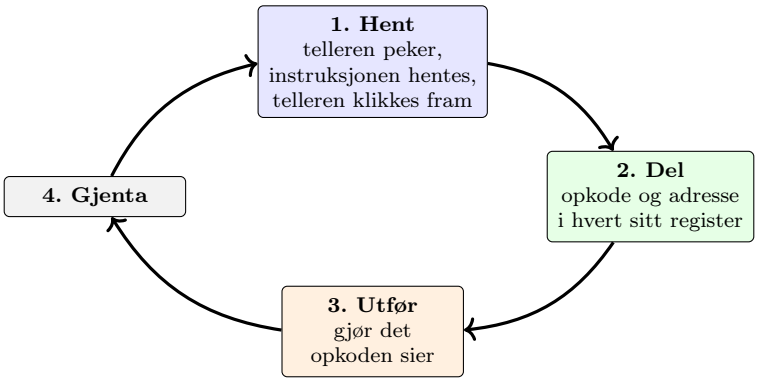
Nå vet vi hva en ordre er. Nå skal vi se hvordan den blir utført.

Kontoristen gjør det samme om og om igjen, i en løkke som aldri varierer. Den løkka heter *hent-utfør-syklusen*.

3.5.1 De fire stegene

1. Bruk verdien i telleren som adresse, og hent instruksjonen som ligger i den postkassen. Klikk samtidig telleren ett hakk fram.
2. Del instruksjonen i to. Opkoden går i instruksjonsregisteret, adressedelen i adresseregisteret.
3. Gjør det instruksjonsregisteret sier.
4. Begynn på nytt fra steg 1.

Det er hele maskinen, og figur 3.4 viser runden. Maskinen gjentar den til den treffer en stoppinstruksjon.



Figur 3.4 Hent-utfør-syklusen. Den stopper bare når opkoden er 0.

3.5.2 De fire registrene

Steg 2 nevnte to registre. Et *register* er en liten lagringsplass inne i prosessoren, langt raskere å nå enn en postkasse.

LMC har fire.

- **Akkumulatoren** er kalkulatoren. Her ligger tallet det regnes med, og hit kommer svaret.
- **Programtelleren** er telleren. Den sier hvilken instruksjon som skal hentes *neste* gang.
- **Instruksjonsregisteret** holder koden til instruksjonen som utføres nå.
- **Adresseregisteret** holder adressedelen av den samme instruksjonen.

De to siste finnes ikke som gjenstander i rommet. Ordren må huskes underveis, fra den hentes til den er utført, og det er den huskingen de to registrene står for.

En moderne prosessor har fra tjue til et par hundre registre som programmet kan se, og noen flere typer enn disse. Den deler dessuten opp arbeidet litt annerledes: der vi har ett register til koden og ett til adressen, har en ekte maskin egne registre for å snakke med minnet. Kapittel 4 tar dem.

3.5.3 Når klikkes telleren?

Dette er det punktet der flest går seg vill, så vi tar det sakte.

Telleren klikkes fram i steg 1, samtidig med hentingen. Ikke etterpå.

Grunnen er en definisjon som lønner seg å holde fast ved:

Tankevekker

Programtelleren peker aldri på instruksjonen som utføres nå. Den peker på den neste.

Tenk på en nummerert huskeliste. Du holder plassen med fingeren. Fornuftig rekkefølge er å flytte fingeren ned til neste linje *før* du gjør det linja sa. Ellers mister du plassen din i det øyeblikket oppgaven sender deg et annet sted.

Og det er akkurat det et hopp gjør. Et hopp sender deg et annet sted.

3.5.4 Derfor virker telleren etter hopp

Sett at instruksjonen i postkasse 07 sier «hopp til 01».

Kontoristen henter den fra 07 og klikker telleren til 08. Så utføres ordren, og ordren sier: sett telleren til 01. Telleren blir 01.

Klikket var ikke bortkastet. Det ble bare overskrevet.

Hadde telleren blitt klikket *etter* at ordren var utført, ville hoppet blitt ødelagt. Telleren hadde blitt satt til 01, og så klikket til 02. Hoppet hadde bommet med én.

Note-to-myself

Rekkefølgen på stegene inne i én instruksjon betyr like mye som rekkefølgen på instruksjonene i programmet. Det er lett å glemme, fordi den første rekkefølgen er usynlig.

3.5.5 Én runde, steg for steg

La oss følge én instruksjon helt gjennom. Utgangspunktet er dette:

- Telleren står på 25.
- Postkasse 25 inneholder `589`.
- Postkasse 89 inneholder `222`.
- Kalkulatoren viser `333`.

Steg	Hva som skjer	Etterpå
1	Les telleren, hent fra postkasse 25	Instruksjonen er `589`, telleren er 26
2	Del `589` i 5 og 89	Opkode 5, adresse 89
3	Opkode 5 betyr hent: les postkasse 89	Kalkulatoren viser `222`
4	Begynn på nytt	Neste instruksjon hentes fra 26

Tabell 3.1 Én runde i syklusen. Postkasse 89 er uendret etter steg 3, og innholdet i kalkulatoren er borte.

To ting å merke seg. Postkasse 89 er uendret, for kontoristen leste bare lappen og la den tilbake. Og tallet 333 er borte for godt, for kalkulatoren har bare plass til ett tall.

3.5.6 Den viktigste konsekvensen

Syklusen spør aldri om det den hentet er en instruksjon.

Steg 1 henter et tall. Steg 3 gjør det tallet sier. Det finnes ikke noe steg som sjekker om tallet var ment som en ordre.

Legger du data der kontoristen kommer til å hente sin neste ordre, blir dataene utført. Maskinen protesterer ikke. Dette er viktig å bemerke seg da det ofte er her både manipulering (hacking) kan skje og at maskinen utfører noe den egentlig ikke skulle ha utført.

3.6 Instruksjonssettet og de første programmene

Her er alle ordrene kontoristen forstår.

Kode	Navn	Hva den gjør
0xx	HLT	Stopper kjøringen
1xx	ADD	Legger verdien i postkasse xx til akkumulatoren
2xx	SUB	Trekker verdien i postkasse xx fra akkumulatoren
3xx	STA	Lagrer akkumulatoren i postkasse xx
4xx	–	Ikke i bruk
5xx	LDA	Henter verdien i postkasse xx til akkumulatoren
6xx	BRA	Hopper til adresse xx
7xx	BRZ	Hopper til xx hvis akkumulatoren er 0
8xx	BRP	Hopper til xx hvis akkumulatoren er 0 eller større
901	INP	Leser et tall fra innkurven til akkumulatoren
902	OUT	Skriver akkumulatoren til utkurven
922	OTC	Skriver akkumulatoren som ett ASCII-tegn
–	DAT	Setter av en postkasse til data

Tabell 3.2 Hele instruksjonssettet til LMC. De fire siste radene har ingen adressedel å fylle ut.

De korte navnene kalles *mnemonics*, av et gresk ord for hukommelse. Vi kommer tilbake til dem i seksjon 3.7. Foreløpig er de bare en lesehjelp.

3.6.1 Det første programmet

Vi skal legge sammen to tall. Brukeren legger to lapper i innkurven, og summen kommer ut.

```
Postkasse   Innhold
00          901       les første tall
01          399       lagre det i postkasse 99
02          901       les andre tall
03          199       legg til det som ligger i 99
04          902       skriv ut svaret
05          000       stopp
...
99          000       her ligger dataene
```

Hvorfor må det første tallet lagres i det hele tatt[2]? Fordi kalkulatoren bare har plass til ett tall. Leser vi det andre tallet rett inn, blir det første borte.

Det er hele grunnen til at STA finnes. Akkumulatoren er en arbeidsbenk, ikke et lager.

[2] Vi har i dette eksemplet valgt postkasse 99, men kunne i prinsippet valgt hvilke som helst annet. Mer om det i kapittel 3.6.2 på neste side

Postkasse	Instruksjon	Akkumulator	Postkasse 99	Hva skjer
00	901	12	000	Brukeren taster 12
01	399	12	012	12 kopieres til 99
02	901	5	012	Brukeren taster 5
03	199	17	012	$5 + 12 = 17$
04	902	17	012	17 havner i utkurven
05	000	17	012	Kontoristen tar pause

Tabell 3.3 Addisjonsprogrammet kjørt med tallene 12 og 5. Første kolonne er postkassen instruksjonen ble hentet fra. Kolonnene «Akkumulator» og «Postkasse 99» viser verdiene *etter* at instruksjonen er utført. Telleren står allerede ett hakk lenger fram hele veien.

Legg merke til rad 4. Postkasse 99 inneholder fortsatt 012 etter at summen er regnet ut. `ADD` leser postkassen, den tømmer den ikke.

3.6.2 Hvorfor postkasse 99?

Fordi den er langt unna programmet.

Programmet opptar postkassene 00 til 05. Enhver postkasse etter 05 hadde fungert like godt. Vi valgte 99 fordi det er tydelig utenfor. Kan du kanskje tenke deg hvordan man burde overveie bruken av postkassene? Hvordan vite hvor mange postkasser programmet trenger for sine instruksjoner og hvor langt unna du har til å lagre ting.

Legger vi derimot dataene i postkasse 03, skjer det noe stygt. Instruksjonen som lå der, blir borte. Og når kontoristen kommer til 03 for å hente sin neste ordre, ligger det et datatall der i stedet.

Det tallet blir utført. Hva maskinen da gjør, kan ingen forutsi uten å regne på det.

Advarsel :)

Å glemme `HLT` på slutten er den vanligste feilen i LMC. Uten stoppinstruksjon fortsetter kontoristen inn i dataene og utfører dem. Hva som da skjer, avhenger av hvilke tall dataene tilfeldigvis er. Vi ser et konkret eksempel på neste side.

3.6.3 Tre tall i stedet for to

Samme oppgave, men nå skal tre tall legges sammen.

```
00  901     les første tall
01  399     lagre i 99
02  901     les andre tall
03  199     legg til
04  399     lagre summen i 99 igjen
05  901     les tredje tall
06  199     legg til
07  902     skriv ut
08  000     stopp
99  000     data
```

Mønsteret er: les, legg til, lagre tilbake. Det er det samme mønsteret som en løkke bruker, og vi møter det igjen i seksjon 3.8.

3.6.4 Et program som skriver tekst

I kapittel 2 lærte vi at tegn er tall. Bokstaven `h` er 104 i ASCII, `e` er 101 og `i` er 105.

Instruksjonen `922` tar tallet i kalkulatoren og skriver det ut som tegn. Da kan vi skrive et ord.

```
00  507       hent tallet i postkasse 07
01  922       skriv det ut som tegn
02  508       hent tallet i postkasse 08
03  922       skriv det ut som tegn
04  509       hent tallet i postkasse 09
05  922       skriv det ut som tegn
06  000       stopp
07  104       'h'
08  101       'e'
09  105       'i'
```

Programmet skriver `hei`.

Her er dataene lagt rett etter programmet, i postkassene 07, 08 og 09. Det er trygt, for kontoristen stopper på 06 og kommer aldri dit.

Tenk om...

Hva hadde skjedd hvis vi hadde glemt `HLT`, og satt en fjerde utskrift i postkasse 06 i stedet? Kontoristen ville kommet til postkasse 07 og hentet `104` som sin neste ordre. Opkode 1 er `ADD`, adressen er 04. Maskinen hadde lagt innholdet i postkasse 04 til kalkulatoren, som er `509`, og fortsatt videre til postkasse 08. Tallet 104 er både bokstaven `h` og ordren «legg til det som ligger i 04». Ingenting i tallet avgjør hvilken av delene det er.

3.7 Assembler: navn i stedet for tall

Se på dette programmet en gang til:

```
901  399  901  199  902  000
```

Hva gjør det?

Du vet svaret, for du leste det for to sider siden. Men du ser det ikke. Du husker det.

Det er problemet med maskinkode. Den er ikke laget for mennesker.

3.7.1 Fire grunner til at tall er tungvint

1. Du må huske hvilket tall som betyr hva.
2. Du kan ikke se hva et program gjør ved å se på det.

3. Du kan ikke se hvilke postkasser som er data og hvilke som er kode.
4. Setter du inn én ny instruksjon, forskyves alt under, og alle adresser må rettes.

Den fjerde er verst, og vi kommer tilbake til den i seksjon 3.9.

3.7.2 Løsningen

Vi gir instruksjonene navn. Så skriver vi et program som bytter navnene tilbake til tall for oss.

Det programmet heter en *assembler*, og språket heter assemblerspråk. Programmet over ser slik ut:

```
INP          les et tall
STA 99       og legg det i postkasse 99
INP          les et tall til
ADD 99       og legg sammen
OUT          skriv ut svaret
HLT          stopp
```

Nå ser du hva programmet gjør.

3.7.3 Hva assembleren faktisk er

En assembler er en én-til-én-oversettelse. Hver linje du skriver blir nøyaktig én postkasse i minnet. Ingen linjer legges til, ingen fjernes.

Det skiller assembler fra en kompilator. En kompilator kan gjøre om én linje Python til hundre maskininstruksjoner, eller til null hvis linja var overflødig. En assembler gjør ingen slike valg.

Tankevekker

Maskinen kjenner ikke `LDA`. Den har aldri sett de bokstavene. `LDA` er noe vi skriver, og assembleren er programmet som fjerner det igjen før maskinen får se noe.

Dette er nøyaktig den typen abstraksjon kapittel 1 handlet om. Grensesnittet er navnene. Implementasjonen er tallene. Og her kan du se gjennom abstraksjonen, for begge sider av den får plass på én side.

Resten av kapitlet bruker assembler. Det gjør programmene lesbare. Husk likevel hva som faktisk ligger i postkassene. Det ligger tall der. Det ligger alltid bare tall der.

3.8 Kontrollflyt

Alt vi har skrevet så langt, kjører rett fram. Første instruksjon, andre instruksjon, og videre nedover til stopp.

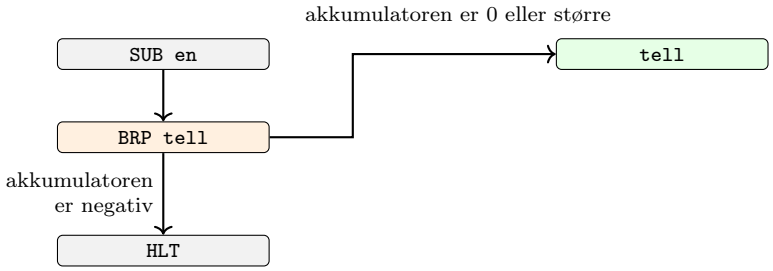
En maskin som bare kan det, er en kalkulator. Den kan gjøre én ting med tallene du gir den.

Det som gjør en datamaskin til noe annet, er at den kan velge. Og valget er bygd av én ting: muligheten til å sette programtelleren til hva som helst.

3.8.1 De tre hoppene

- `BRA xx` setter telleren til `xx`. Alltid.
- `BRZ xx` setter telleren til `xx` hvis akkumulatoren er null. Ellers skjer ingenting.
- `BRP xx` setter telleren til `xx` hvis akkumulatoren er null eller positiv. Ellers skjer ingenting.

Det er alt. Tre instruksjoner, og med dem kan du bygge enhver løkke og enhver betingelse som finnes i Python. Figur 3.5 viser hvordan et betinget hopp deler programmet i to veier.



Et betinget hopp er et veiskille. Maskinen tar den ene veien eller den andre, og valget avhenger *bare* av tallet i akkumulatoren.

Figur 3.5 Et betinget hopp. Instruksjonen som testes, står ikke i hoppet – den står over det.

Advarsel :)

Null er positivt i `BRP`. Instruksjonen hopper hvis akkumulatoren er null eller større, og faller bare gjennom hvis tallet er negativt. Dette er en klassisk felle, og den er årsaken til at nedtellingsprogrammet nedenfor stopper på -1 og ikke på 0.

3.8.2 Det hoppet ser på

`BRZ` og `BRP` ser på akkumulatoren. De ser ikke på instruksjonen over. De ser ikke på noen postkasse. De ser på ett tall, og det tallet ble satt av en tidligere instruksjon.

Det betyr at testen og det som testes, står to forskjellige steder i programmet. Mellom dem kan det stå hva som helst. Skriver du en `LDA` mellom dem, tester du på noe helt annet enn du trodde.

Note-to-myself

Vil du teste om et tall er 7, finnes det ingen instruksjon for det. Du må trekke fra 7 og teste mot null. Alle sammenligninger i LMC blir til en subtraksjon etterfulgt av et hopp. Slik er det i ekte maskiner også, og det er derfor de har et eget nullflagg.

3.8.3 En løkke som ikke stopper

Vi skal telle ned fra 10 og skrive ut hvert tall. Her er et forsøk.

```
00  LDA 05       hent 10
01  OUT          skriv ut
```

```
02  SUB 06       trekk fra 1
03  BRA 01       hopp tilbake
04  HLT
05  10           data: startverdien
06  1            data: tallet én
```

Kjør det, og maskinen skriver 10, 9, 8 og videre nedover. Den skriver også -1, -2, -3. Den kommer aldri til postkasse 04.

Feilen er `BRA`. Et ubetinget hopp hopper alltid.

Note-to-myself

I simulatoren stopper løkka likevel til slutt, men ikke fordi programmet er ferdig. Kalkulatoren har bare plass til tre sifre, så et sted under -999 går den tom og gir feilmelding. Det er en annen feil enn den vi leter etter.

3.8.4 Løkka som stopper

Bytt `BRA` med `BRP`:

```
00  LDA 05       hent 10
01  OUT          skriv ut
02  SUB 06       trekk fra 1
03  BRP 01       hopp tilbake hvis 0 eller større
04  HLT
05  10
06  1
```

Runde	Skrevet ut	Etter `SUB`	`BRP` gjør
1	10	9	hopper
2	9	8	hopper
⋮	⋮	⋮	⋮
10	1	0	hopper, for null er positivt
11	0	-1	hopper ikke

Tabell 3.4 Nedtellingen. Runde 11 skriver ut null og faller så gjennom til `HLT`.

Programmet skriver ut tallene fra 10 til 0, og stopper.

Tenk om...

Hva om vi heller ville stoppe *før* null ble skrevet ut? Da kunne vi lagt en `BRZ` et sted. Men hvor? Legger du den etter `SUB`, hopper programmet ut i det akkumulatoren når null, og null blir aldri skrevet ut. Legger du den før `OUT`, må du ha lastet verdien først. Rekkefølgen mellom test og utskrift avgjør hva som kommer med. Prøv begge i simulatoren.

3.8.5 Det samme i Python

Nedtellingen svarer til dette:

```
tall = 10
while tall >= 0:
    print(tall)
    tall = tall - 1
```

Fire linjer Python. Sju postkasser i LMC.

Forskjellen er ikke bare lengde. I Python står betingelsen øverst i løkka, og maskineriet som hopper er skjult. I LMC står hoppet nederst, og betingelsen er en subtraksjon lenger opp. Rekkefølgen snur.

Tankevekker

Byggeklossene i LMC er mye mindre enn dem du er vant til. Én `while`-linje er tre instruksjoner. Én `if` er to. Prosessoren i maskinen din bygger på nøyaktig samme vis, bare raskere og med flere klosser.

3.8.6 Et valg, ikke en løkke

Nå et program med `if`. Vi leser to tall og skriver ut forskjellen mellom dem, alltid som et positivt tall.

```
00  INP          les første tall
01  STA 10       lagre det
02  INP          les andre tall
03  STA 11       lagre det
04  SUB 10       andre minus første
05  BRP 08       hopp hvis svaret er positivt
06  LDA 10       ellers: snu regnestykket
07  SUB 11       første minus andre
08  OUT          skriv ut
09  HLT
10  0            data
11  0            data
```

Dette er en `if-else`. Linje 05 er testen. Linje 06 og 07 er `else`-grenen. Er svaret allerede positivt, hopper vi rett forbi dem.

Studer det til du kan følge begge veier. Dette mønsteret er alt en maskin trenger for å ta valg.

Note-to-myself

Dataene ligger i postkassene 10 og 11, rett etter `HLT`. Fjern `HLT` i postkasse 09, og kontoristen går rett inn i dataene. Postkasse 10 inneholder det første tallet brukeren tastet inn, så hva maskinen gjør, avgjøres av brukeren. Taster brukeren 7, blir tallet `007` lest som en stoppinstruksjon, og alt ser bra ut. Taster brukeren 512, blir det lest som `LDA 12`, og maskinen fortsetter inn i noe helt annet. Feilen viser seg altså bare for noen inndata, og det er den verste typen feil som finnes.

3.9 Adresser, etiketter og data

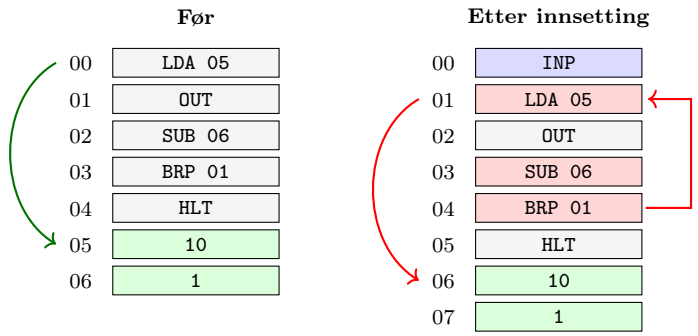
Programmene våre begynner å bli lange nok til å bli upraktiske. Og de er skjøre på en måte som er verdt å se på.

3.9.1 Problemet med tall som adresser

Se på nedtellingsprogrammet igjen. Startverdien ligger i postkasse 05.

Sett at du vil legge til en instruksjon på toppen, for eksempel en `INP` som lar brukeren velge hvor det skal telles fra. Alt under forskyves ett hakk. Dataene som lå i 05 og 06 ligger nå i 06 og 07.

Figur 3.6 viser hva som skjer. Instruksjonene sier fortsatt `LDA 05` og `SUB 06`. De peker feil. Og hoppet sier fortsatt `BRP 01`, som nå treffer `LDA` i stedet for `OUT`. Tre adresser er ødelagt, og ingenting sier fra.



Én ny instruksjon på toppen, og tre adresser peker feil.
Ingen feilmelding. Programmet gjør bare noe annet enn det skal.

Figur 3.6 Hvorfor talladresser ikke skalerer.

3.9.2 Etiketter

Løsningen er å gi postkassene navn.

En *etikett* er et navn på en adresse. Du skriver navnet i venstre kolonne for å døpe en postkasse. Du skriver det samme navnet i høyre kolonne for å peke på den.

Nedtellingen med etiketter:

```
        LDA start
tell    OUT
        SUB en
        BRP tell
        HLT
start   DAT 10
en      DAT 1
```

Nå kan du sette inn så mange instruksjoner du vil. Assembleren regner ut adressene på nytt hver gang.

Note-to-myself

Navnet betyr ingenting for maskinen. Du kan kalle postkassen `start`, `Amira` eller `x` uten at noe endrer seg. Etiketten forsvinner i det assembleren har gjort jobben sin, og det som ligger i minnet er et tall.

Er en etikett det samme som en variabel i Python? Nesten, men ikke helt. En variabel i Python er et navn på en verdi, og verdien kan flytte seg. En etikett er et navn på en *adresse*, og adressen ligger fast. Det er stedet som har navn, ikke innholdet.

3.9.3 DAT

`DAT` er ikke en instruksjon. Maskinen har ingen opkode for den.

`DAT` er en beskjed til assembleren: sett av en postkasse her, og legg eventuelt et starttall i den. Slike beskjeder kalles *pseudoinstruksjoner*.

Du kan skrive `DAT` uten tall. Da settes postkassen av uten at innholdet er bestemt. Det er sjelden lurt.

Advarsel :)

En `DAT` uten starttall inneholder det som lå der fra forrige kjøring. Programmet virker da kanskje første gang og feiler andre gang. Initialiser alltid dataene dine. Dette rådet gjelder i alle programmeringsspråk, men her ser du hvorfor.

3.9.4 Hva ligger egentlig i postkassen?

Tre spørsmål. Svar på hvert av dem før du leser videre.

```
Postkasse 00:   en      DAT 1
Postkasse 01:   peker   DAT en
Postkasse 02:   hent    LDA en
```

Hvilket *tall* står i hver av de tre postkassene?

I postkasse 00 står `001`. Det er verdien vi ba om.

I postkasse 01 står `000`. `en` er et navn på postkasse 00, og adressen til postkasse 00 er tallet 0.

I postkasse 02 står `500`. Opkoden for `LDA` er 5, og adressen til `en` er 00.

Tankevekker

En etikett er bare et tall. En instruksjon er bare et tall. Forskjellen mellom `peker DAT en` og `hent LDA en` er sifferet lengst til venstre, og ingenting annet.

Har du dette, har du hele resten av kapitlet.

3.9.5 Problemet med lister

Ta for deg at vi vil skrive ut en hel tekst. Teksten ligger i påfølgende postkasser.

Sett at teksten begynner i postkasse 12. Første bokstav hentes da med `LDA 12`, andre med `LDA 13`, tredje med `LDA 14` osv...

Instruksjonen må altså endre seg mellom hver runde i løkka. Men instruksjonene ligger fast i minnet.

Eller gjør de det?

Instruksjonen `LDA 12` er tallet `512`. Det ligger i en postkasse. Og et tall i en postkasse kan vi legge én til.

`512` pluss 1 er `513`, som er `LDA 13`. Legger vi til én på *instruksjonen*, peker den på neste postkasse.

```
Adr
00          LDA laster   hent tallet 512
```

```
01          STA hent     skriv det inn som instruksjon
02  hent    DAT 0        denne cellen overskrives hver runde
03          BRZ slutt    null betyr slutt på teksten
04          OTC          skriv ut tegnet
05          LDA hent     hent instruksjonen som et tall
06          ADD en       legg til én
07          STA hent     skriv den tilbake
08          BRA hent     kjør den på nytt
09  slutt   HLT
10  laster  DAT 512      dette er LDA 12, som et tall
11  en      DAT 1
12  tekst   DAT 104      'h'
13          DAT 101      'e'
14          DAT 105      'i'
15          DAT 0        null markerer slutten
```

Postkassen med etiketten `hent` inneholder en instruksjon som programmet selv skriver om mellom hver runde. Først `LDA 12`, så `LDA 13`, så `LDA 14`. Til slutt `LDA 15`, som henter nullen, og da hopper `BRZ` ut.

Legg merke til at `BRA hent` hopper til postkasse 02, ikke til 00. Runde to og utover starter altså midt i programmet, rett på den instruksjonen som nettopp ble skrevet om.

Og legg merke til én ting til: `laster DAT 512` er en adresse skrevet inn for hånd, i et kapittel som nettopp har forklart hvorfor det er en dårlig idé. Flytter du teksten, må du rette 512 selv. Slik er det med selvmodifiserende kode. Den bryter ut av det assembleren kan hjelpe deg med.

Advarsel :)

Det som legges til én, er *adressedelen* i instruksjonen. Ikke verdien som ligger på den adressen. Går du fra `512` til `513`, har du flyttet deg én postkasse bortover. Bokstaven du henter blir en annen, men ikke fordi bokstavtallet ble større. Dette er det spørsmålet studenter oftest stiller om selvmodifiserende kode.

3.9.6 Hva ekte maskiner gjør i stedet

En ekte prosessor har *indeksregistre*. Instruksjonen sier «hent fra adressen som står i registeret», og så teller du opp registeret i stedet for instruksjonen. Programmet ligger i ro.

Det kalles *indirekte adressering*, fordi adressen kommer et annet sted fra enn instruksjonen selv. LMC har det ikke. Kapittel 4 viser hvordan det er bygget.

3.9.7 Så er det lov, eller ikke?

Både og, og svaret er mer interessant enn det høres ut.

Moderne operativsystemer merker minnet ditt. En side kan være skrivbar, eller den kan være kjørbar. Som hovedregel ikke begge deler samtidig. Det gjør at et program ikke uforvarende, og ikke ondsinnet, kan skrive nye instruksjoner til seg selv.

Samtidig gjør noen programmer nettopp det, med vilje og med tillatelse. Nettleseren din oversetter JavaScript til maskinkode mens siden kjører, skriver koden i minnet og hopper til den. Det heter *just-in-time-kompilering*. Java og C# gjør det samme.

Og enda viktigere: for operativsystemet og for kompilatoren *er* programmet ditt data. Det må det være. Noen må lese det, flytte det og legge det i minnet før det kan bli et program.

Tankevekker

Grensen mellom kode og data er ikke noe maskinen finner. Den er noe vi håndhever, med vilje, fordi det ellers går galt. Den finnes i operativsystemet, ikke i silisiumet (maskinvaren).

3.9.8 Når det går galt

Et program som skriver til feil adresse, overskriver noe. Hva det overskriver, avhenger av hva som tilfeldigvis lå der.

Er det data, får du feil svar. Er det en instruksjon, kjører maskinen noe annet enn programmet ditt. Og er den feilaktige adressen noe en utenforstående kan påvirke, kan den utenforstående bestemme hva som blir kjørt.

Det er dette som ligger under en stor andel av sikkerhetshullene du leste om i første halvdel av emnet. Ikke et magisk angrep. Bare en maskin som utfører tallene den finner, slik den alltid har gjort.

Advarsel :)

Ikke bruk selvmodifiserende kode i vanlige programmer. Den er nesten umulig å feilsøke, den ødelegger for prosessorens forhåndshenting av instruksjoner, og den er blokkert på de fleste systemer. Her er den med fordi den avslører hva en maskin er, ikke fordi den er en god teknikk.

3.10 Oppsummering

Kapitlet startet med en Python-linje og endte med et program som redigerer seg selv. Veien mellom dem gikk gjennom én idé, gjentatt fire ganger.

Begrep	**Kort sagt**
Maskinkode	Programmet som tall. Det eneste maskinen faktisk har.
Assembler	Navn på tallene, og et program som fjerner navnene igjen.
Opkode	Første siffer. Sier hva som skal gjøres.
Adressedel	De to siste sifrene. Sier hvor, når det er aktuelt.
Akkumulator	Arbeidsbenken. Plass til ett tall om gangen.
Programteller	Adressen til den *neste* instruksjonen.
Hent-utfør	Hent, del, utfør, gjenta. Telleren klikkes under hentingen.
Betinget hopp	Setter telleren hvis akkumulatoren oppfyller en betingelse.
Etikett	Et navn på en adresse. Forsvinner når assembleren er ferdig.
`DAT`	Beskjed til assembleren om å sette av en postkasse.

Tabell 3.5 Begrepene fra dette kapitlet.

Den gjennomgående ideen kom igjen som en advarsel om hvor data skal ligge, som forklaringen på hva et program uten `HLT` gjør, som trikset bak selvmodifiserende kode, og til slutt som årsaken bak en hel klasse sikkerhetshull.

3.10.1 Videre til neste kapittel

Kontoristen var aldri der.

Vi har brukt hele kapitlet på en person i et rom, og det har vært nyttig, fordi du kan forestille deg selv i den rollen. Du kan flytte lapper mellom postkasser uten å vite hva regnestykket gjelder. Det er nettopp det en prosessor gjør.

Men det sitter ingen i maskinen din. Noe annet klikker telleren, deler instruksjonen i to og legger tall sammen. Det er kretser.

I kapittel 4 åpner vi rommet og ser hva som faktisk gjør jobben. Der blir også instruksjonen noe annet enn tre sifre. `542` blir et bitmønster, feltene blir grupper av bit, og hvor mange bit hvert felt skal ha blir et valg med konsekvenser.

Note-to-myself

Den samme ordren, i tre former. Som assembler: `LDA 42`. Som tall: `542`. Som bit, slik en ekte maskin lagrer den, er det bare et mønster av nuller og enere – og hvilket mønster, avhenger av hvordan maskinen er bygget. Det er neste kapittel.

3.11 Videre lesning

Little Man Computer. Englander kapittel 6 er den opprinnelige framstillingen, og den er kort og god. Merk at boka bruker `STO`, `IN` og `BR` der vi har brukt `STA`, `INP` og `BRA`.

Hent-utfør med registre. Englander 7.4 viser den samme syklusen med registeroverføringer skrevet ut, og med to registre vi ikke har hatt bruk for her. Les den når du har lest kapittel 4.

Hva slags instruksjoner finnes. Englander 7.5 går gjennom kategoriene i et ekte instruksjonssett: dataflytting, aritmetikk, logikk, skift, hopp og kall. LMC har de tre første i minimal utgave.

Simulatoren. Peter Higginsons LMC-simulator kjører i nettleseren. Skriv programmene i en tekstfil og lim dem inn, i stedet for å taste dem i nettleservinduet. Da mister du dem ikke.

På egen hånd. Finn ut hva prosessoren i din egen maskin heter, og søk opp instruksjonssettet. Du kommer til å kjenne igjen mer enn du tror.

3.12 Refleksjon

- **Om nedstigningen.** Python-linja `svar = a + b` ble til tre LMC-instruksjoner i figur 3.1 på side 56. Hva er det Python gjør for deg som LMC ikke gjør? Nevn minst tre ting, og avgjør for hver av dem om det koster tid under kjøring.
- **Om telleren.** Hvorfor definerer vi programtelleren som adressen til den *neste* instruksjonen, og ikke den som utføres nå? Hva hadde blitt vanskeligere med den andre definisjonen?
- **Om elleve instruksjoner.** LMC har elleve. Din egen prosessor har flere tusen. Hva bruker den de resterende til, når elleve er nok til å regne ut hva som helst?
- **Om grensen.** Vi sa at grensen mellom kode og data ikke finnes i maskinvaren, men håndheves av operativsystemet. Hva slags maskin ville du fått hvis grensen var bygget inn i maskinvaren, og hva ville du mistet?

- **Om abstraksjon.** Assembleren skjuler tallene. Etikettene skjuler adressene. Hva skjuler de *ikke*, som en Python-programmerer aldri trenger å tenke på?
- **Om kontoristen.** Modellen forutsetter et menneske som følger ordrer uten å forstå hva regnestykket gjelder. Hva sier det om hva forståelse er, at arbeidet lot seg overta av en maskin?

3.13 Selv-test

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

1. Hvor mange postkasser har LMC, og hvorfor akkurat så mange?
2. Del instruksjonen `347` i de to feltene sine, og si hva hvert felt betyr.
3. Hva er forskjellen på innholdet i en postkasse og adressen til den? Gi et eksempel der de to er lette å forveksle.
4. Nevn de fire stegene i hent-utfør-syklusen. I hvilket steg klikkes programtelleren fram?
5. Hva peker programtelleren på mens en instruksjon utføres?
6. Hvorfor er `901` en misvisende instruksjon å lese som «gjør noe med postkasse 01»?
7. Programmet `901 399 901 199 902 000` ligger fra postkasse 00. Brukeren taster inn 40 og deretter 2. Hva står i utkurven, og hva står i postkasse 99 når programmet er ferdig?
8. Skriv om det samme programmet til assembler med etiketter i stedet for talladressen 99.
9. Hvilken opkode er ikke i bruk i LMC?
10. Akkumulatoren inneholder 0. Hopper `BRP 20`? Hopper `BRZ 20`? Begrunn begge svarene.
11. Skriv et LMC-program som leser to tall og skriver ut det største av dem.
12. Skriv et LMC-program som leser et tall og skriver ut det dobbelte. LMC har ingen gangeinstruksjon.
13. Denne løkka skal telle ned fra 5 til 0, men den skriver bare ut ett tall. Finn feilen og rett den.

    ```
    00  LDA 05
    01  OUT
    02  SUB 06
    03  BRZ 01
    04  HLT
    05  5
    06  1
    ```

14. En medstudent sier: «`BRZ` sjekker om instruksjonen over ga null.» Hva er galt med den beskrivelsen, og hvordan ville du formulert den riktig?
15. Hva står det for et tall i postkassen etter linja `svar DAT 7`? Og etter linja `peker DAT svar`, hvis `svar` ligger i postkasse 31?
16. Hvorfor er `DAT` ikke en instruksjon?
17. Et program mangler `HLT`. Første postkasse etter siste instruksjon inneholder tallet 250. Hva gjør maskinen? Hvorfor er det umulig å svare uten å vite hva som ligger i postkasse 50?

18. Se tilbake på figur 3.6 på side 71. Forklar med egne ord hvorfor etiketter løser problemet figuren viser, og hvorfor assembleren må kjøres på nytt hver gang du endrer programmet.

19. I programmet med selvmodifiserende kode hopper `BRA hent` til postkasse 02 og ikke til 00. Hva hadde skjedd hvis det hoppet til 00 i stedet?

20. Skriv et LMC-program som leser inn tall til brukeren taster 0, og som så skriver ut summen av alle tallene.

21. Kapittel 2 sa at et bitmønster ikke betyr noe uten en tolkning. Dette kapitlet sa at et tall i minnet kan være både data og instruksjon. Er dette den samme påstanden? Forklar hva som er felles og hva som er nytt.

22. En maskin kjører et program som har skrevet til en adresse den ikke skulle ha skrevet til. Beskriv tre ulike ting som kan skje videre, avhengig av hva som lå på den adressen.

Kapittel 4

Arkitektur og ALU

Innhold

4.1 Læringsmål . . . 80
4.2 To ord for den samme maskinen . . . 80
4.3 Ordren som bitmønster . . . 81
4.4 Registrene . . . 84
4.5 Minnet sett fra prosessoren . . . 87
4.6 Bussene . . . 89
4.7 ALU-en . . . 91
4.8 Flaggene . . . 93
4.9 Kontrollenheten og datastien . . . 95
4.10 Modellen som binder det sammen . . . 99
4.11 Oppsummering . . . 101
4.12 Videre lesning . . . 102
4.13 Refleksjon . . . 103
4.14 Selv-test . . . 103

Når et fjell skal bores gjennom, graves det ofte fra begge sider samtidig.

To arbeidslag starter i hver sin dal. De borer mot hverandre, måned etter måned, uten å se hverandre. Alt hviler på at oppmålingen stemmer. Bommer de, møtes ikke gangene, og en av dem må svinge.

Lærdalstunnelen går 24,5 kilometer gjennom fjellet mellom to bygder på Vestlandet. Den ble gravd på denne måten. Da lagene møttes inne i fjellet, var avviket noen centimeter.

Vi har gravd siden kapittel 1.

Det første kapitlet sto på toppen og så nedover gjennom lagene. Kapittel 2 gjorde tekst, tall og bilder om til bit. Kapittel 3 gikk ned til ordrene maskinen faktisk utfører, og endte med en påstand: kontoristen var aldri der.

Fra den andre siden graver noen oppover. De begynner med strøm gjennom et stykke halvleder, lager en bryter av det, setter bryterne sammen til porter, og portene til kretser som kan regne. Den gangen tar vi i kapittel 5 og 6.

Dette kapitlet er stedet der de to gangene møtes.

Møtepunktet er én ting: ordren, skrevet i bit. Ovenfra er den siste ledd i en lang oversettelse som

begynte i Python. Nedenfra er den det første som ser ut som mening i stedet for spenning. Det er det samme mønsteret av nuller og enere, sett fra hver sin kant.

4.1 Læringsmål

Etter å ha arbeidet med dette kapitlet skal du kunne:

1. **Skille mellom hva en maskin kan gjøre og hvordan den gjør det.** Dette skillet har et navn, og det er grunnen til at programmer du skrev for ti år siden fortsatt kjører.
2. **Dele et instruksjonsord i felt, og regne på delingen.** Du skal kunne forklare hvorfor en bit mer til opkoden alltid koster en bit fra adressen, og hva det valget gjør med maskinen.
3. **Gjøre rede for registrene i en prosessor, også de to som snakker med minnet.** Du skal vite hva som ligger hvor mens et program kjører, og hvorfor registre finnes i det hele tatt.
4. **Forklare hva de tre bussene gjør, og regne på bredden deres.** Herfra kommer svaret på et spørsmål mange har lurt på: hvorfor stoppet gamle maskiner på fire gigabyte?
5. **Beskrive hva en ALU gjør, og lese av flaggene etter en operasjon.** Du skal kunne se forskjell på et regnestykke som gikk bra og ett som ga feil svar uten å si fra.
6. **Følge en ordre gjennom datastien, fra bitmønster til utført.** Dette er hent-utfør-syklusen fra forrige kapittel, nå med maskineriet synlig.

4.2 To ord for den samme maskinen

Det finnes to helt ulike spørsmål du kan stille om en datamaskin.

Det første er hva den kan gjøre. Hvilke ordrer forstår den? Hvor mange registre har den? Hvordan er en ordre bygget opp? Dette er spørsmål du må ha svar på for å skrive et program.

Det andre er hvordan den gjør det. Hvor mange kretser ligger det bak en addisjon? Hvor mange klokkeslag tar en ordre? Ligger det en egen krets for multiplikasjon, eller løses det med gjentatt addisjon?

De to spørsmålene har hvert sitt navn.

- **Instruksjonssettarkitekturen**, forkortet **ISA**, er svaret på det første. Den er avtalen mellom programvaren og maskinvaren.
- **Mikroarkitekturen** er svaret på det andre. Den er den faktiske byggingen.

Alt kapittel 3 handlet om, var en ISA. De elleve ordrene, de hundre postkassene, akkumulatoren: det er en avtale, ikke en byggebeskrivelse. Vi sa aldri ett ord om hvordan en addisjon faktisk utføres.

4.2.1 Rattet og motoren

Tenk på en bil.

Rattet, pedalene og giret er grensesnittet. Lærer du å kjøre én bil, kan du kjøre nesten alle. Du vet at venstre pedal bremser, uten å vite om bremsene er skiver eller trommer.

Motoren er implementasjonen. To biler kan ha nøyaktig samme betjening og helt ulike motorer. Den ene er elektrisk, den andre går på bensin. For den som kjører, er forskjellen i hovedsak lyden.

ISA er rattet og pedalene. Mikroarkitekturen er motoren.

Tankevekker

Takket være dette skillet overlever programvare maskinvaren den ble skrevet for. Et program fra en gammel maskin kjører på en ny, fordi avtalen er den samme, selv om alt under er bygget om. Hadde programmene vært bundet til kretsene, måtte alt skrives på nytt hver gang en ny brikke kom.

Produsenter utnytter dette hele tiden. En ny prosessorgenerasjon er som regel en ny mikroarkitektur på en gammel ISA. Den er raskere og bruker mindre strøm, men den forstår de samme ordrene. Derfor merker du ingenting annet enn at maskinen er kvikkere.

Note-to-myself

Når noen sier at to maskiner er «kompatible», mener de at de har samme ISA. De sier ingenting om hva som er inni.

Resten av dette kapitlet er mikroarkitektur. Vi åpner panseret.

4.3 Ordren som bitmønster

Kapittel 3 endte med et løfte: `LDA 42` er `542`, og `542` er egentlig et bitmønster. Nå innfrir vi det.

4.3.1 Fra tre sifre til tolv bit

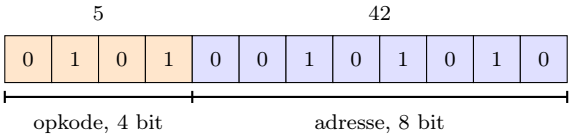
LMC teller i titallssystemet. En ordre er tre sifre: ett til opkoden, to til adressen.

En ekte maskin har ingen sifre. Den har bit. Spørsmålet blir derfor hvor mange bit hvert felt skal ha.

Start med opkoden. LMC bruker ni opkoder. Med 3 bit får vi $2^3 = 8$ ulike mønstre, og det er for lite. Med 4 bit får vi 16, og da er det plass til alle ni, med rom til overs.

Så adressen. Det er hundre postkasser, nummerert 0 til 99. Med 6 bit får vi 64, som er for lite. Med 7 bit får vi 128, som holder. Vi runder opp til 8 bit, så har vi en hel byte og plass til 256.

Til sammen blir ordren 12 bit lang.



Figur 4.1 Ordren `LDA 42` som tolv bit. Opkoden 5 blir 0101, adressen 42 blir 00101010.

Delingen i to felt er nøyaktig den samme som i kapittel 3. Det eneste som har endret seg, er tallsystemet.

Som assembler	Som tall	Som tolv bit
`LDA 42`	542	`0101 00101010`
`ADD 99`	199	`0001 01100011`
`STA 99`	399	`0011 01100011`
`BRZ 7`	707	`0111 00000111`
`HLT`	000	`0000 00000000`

Tabell 4.1 Den samme ordren i tre former. Mellomrommet mellom feltene står der bare for lesbarheten; maskinen ser tolv bit på rad.

Advarsel :)

Mellomrommet i den høyre kolonnen finnes ikke i maskinen. Det er like oppdiktet som mellomrommet i et telefonnummer. Maskinen vet hvor skillet går fordi feltbredden er bestemt én gang for alle, ikke fordi noe markerer det.

4.3.2 Hvor mange bit til hva?

Her kommer avveiningen som er hele poenget med seksjonen.

Ordet er tolv bit. Det tallet ligger fast. Da er også summen av de to feltene fast, og hver bit du gir til det ene, tar du fra det andre.

Opkode	Adresse	Antall opkoder	Antall postkasser	Duger til LMC?
3 bit	9 bit	8	512	Nei, vi trenger ni opkoder
4 bit	8 bit	16	256	Ja
5 bit	7 bit	32	128	Ja
6 bit	6 bit	64	64	Nei, vi trenger hundre postkasser

Tabell 4.2 Fire måter å dele tolv bit på. Bare de to i midten kan kjøre LMC.

Tabellen har riktige og gale svar, og det er verdt å stoppe ved. Øverste rad gir rikelig med minne og for få ordrer. Nederste rad gir rikelig med ordrer og for lite minne. En maskin må klare begge deler samtidig.

Tenk om...

... du skulle legge til tjue nye ordrer i LMC. Med 4 bit til opkoden er det ikke plass; seksten er taket. Du må enten utvide ordet, eller ta en bit fra adressen og halvere antall postkasser. Begge koster noe. Det er slike valg en arkitekt tar, og de står fast i tiår etterpå.

4.3.3 Å regne baklengs

I praksis går en arkitekt motsatt vei. Han vet hva maskinen skal klare, og regner seg fram til hvor bredt ordet må være.

Si at Tobias får i oppdrag å lage en maskin med 40 ulike ordrer og 1 megabyte minne, adressert byte for byte.

Han starter med opkoden. Fem bit gir 32, som er for lite til 40. Seks bit gir 64, som holder. **Opkoden må ha 6 bit.**

Så adressen. 1 megabyte er 2^{20} byte, og for å nå dem alle trengs 20 bit. **Adressefeltet må ha 20 bit.**

Til sammen 26 bit. Det er et ubekvemt tall, så han runder opp til 32, som er et helt antall byte. Da står seks bit ledige.

De seks blir sjelden liggende ubrukt. De kan gå til opkoden, slik at det blir plass til flere ordrer senere. De kan gå til adressen, slik at maskinen kan vokse. Eller de kan gå til et helt nytt felt, for eksempel et registernummer.

Tenk om...

... han hadde valgt å la ordet være 26 bit, akkurat så bredt som kravet tilsa.
Hver ordre ville tatt tre og en kvart byte i minnet. Noen ordrer ville begynt midt inne i en byte, og maskinen måtte regnet ut hvor hver enkelt starter. Det ville spart litt plass og kostet mye tid. Derfor velger nesten alle et helt antall byte, og lever med at noen bit står tomme.

4.3.4 Det samme regnestykket i en ekte maskin

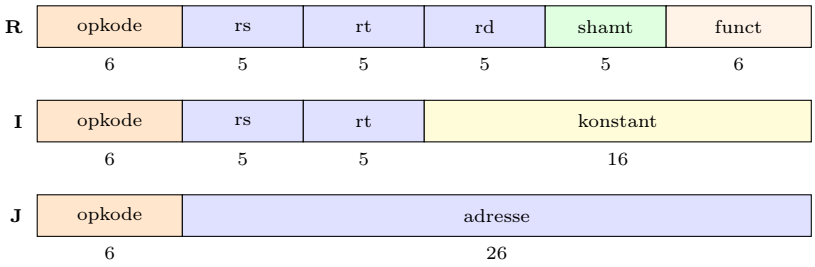
Tolv bit er et leketøy. Prinsippet er likevel uendret når ordene blir lange.

Pensumboka viser et enkelt instruksjonsord på 32 bit, delt i 8 bit opkode og 24 bit adresse. Det gir $2^8 = 256$ ulike ordrer og 2^{24} adresser, altså rundt seksten millioner.

Seksten millioner adresser høres mye ut. Det er 16 megabyte. En mobiltelefon i dag har flere hundre ganger så mye minne. Et enkelt 32-bits ord med ett adressefelt strekker rett og slett ikke til lenger, og det er en av grunnene til at ekte instruksjonsord ser mer innfløkte ut enn vårt.

4.3.5 Tre formater fra en ekte maskin

MIPS er en instruksjonssettarkitektur som brukes mye i undervisning, fordi den er ryddig. Alle ordrene er nøyaktig 32 bit. Det finnes tre formater, og alle tre starter med 6 bit opkode.



Figur 4.2 De tre instruksjonsformatene i MIPS. Alle er 32 bit, og alle starter med seks bit opkode. Tallene under feltene er feltbredden i bit.

Seks bit opkode gir $2^6 = 64$ ulike ordrer. Fem bit til et registernummer gir $2^5 = 32$ registre.

Legg merke til hva som skjer i R-formatet. Der er det ingen minneadresse. Tre felt peker på registre: to kilder og ett mål. Ordren `legg sammen innholdet i register 9 og register 12 og sett svaret i register 4` får plass på 32 bit med god margin. Et registernummer koster fem bit. En full minneadresse koster trettito.

Det er nettopp derfor moderne maskiner har mange registre. Det er billig å peke på dem.

Fun Fact

I LMC er akkumulatoren underforstått. Ordren `ADD 99` sier hvor det ene tallet ligger, men ikke hvor det andre ligger og ikke hvor svaret skal. Begge deler er akkumulatoren, alltid. Fagordet for en slik underforstått operand er *implisitt*. R-formatet over har tre *eksplisitte* operander, og derfor trenger det tre felt.

4.3.6 Fast eller variabel ordlengde

MIPS har ordrer som alltid er 32 bit. Det er ikke den eneste måten.

Noen arkitekturer lar ordren være akkurat så lang som den trenger. I x86, som sitter i de fleste bærbare maskiner, er de korteste ordrene én byte, og de lengste femten. En enkel ordre tar lite plass, en innfløkt ordre tar mer.

Det sparer minne. Til gjengjeld koster det noe annet: maskinen vet ikke hvor neste ordre begynner før den har funnet ut hvor lang denne er. Det er et reelt problem når maskinen prøver å jobbe med flere ordrer samtidig, noe den gjør hele tiden i kapittel 7.

Fast lengde	**Variabel lengde**
Alle ordrer like lange.	Hver ordre så lang den trenger.
Maskinen vet alltid hvor neste begynner.	Lengden må regnes ut først.
Litt sløsing med plass.	Sparer plass.
SPARC, ARM, RISC-V.	x86.
(Både ARM og RISC-V har en komprimert variant i tillegg.)	

Nesten alle nye arkitekturer velger fast lengde.

4.3.7 Hvorfor telleren plutselig øker med fire

Dette punktet forvirrer mange, så vi tar det sakte.

I LMC rommer én postkasse én hel ordre. Programtelleren øker derfor med én. Neste ordre ligger i neste postkasse, og alt stemmer.

En ekte maskin adresserer minnet i *byte*. Hver adresse peker på én byte, ikke på én ordre. En ordre på 32 bit er fire byte, og den opptar fire adresser på rad.

Ligger en ordre på adresse 1000, ligger neste på adresse 1004. Derfor øker programtelleren med fire, ikke med én.

Note-to-myself

Det er den samme regelen hele veien: telleren skal peke på neste ordre. I LMC er neste ordre ett hakk unna. I en 32-bits maskin er den fire hakk unna. Tallet endrer seg, regelen gjør det ikke.

4.4 Registrene

Kapittel 3 innførte registeret som «en liten lagringsplass inne i prosessoren». Det holdt så langt. Nå trenger vi en skarpere definisjon.

Et **register** er en fast lagringsplass inne i prosessoren, viet til én bestemt oppgave.

To ord i den setningen gjør hele jobben: *fast* og *bestemt*. De skiller registeret fra en postkasse på to måter.

4.4.1 Lommer, ikke garderobe

En garderobe har nummererte knagger. Hvilken som helst jakke kan henge på hvilken som helst knagg. Skal du finne jakka igjen, trenger du nummeret. Det er minnet.

Lommene dine er få, og hver av dem har sin ting. Nøklene i den ene, mobilen i den andre. Du slår ikke opp et nummer for å finne mobilen. Du stikker hånda ned der mobilen pleier å være. Det er registrene.

De to forskjellene, satt på ord:

- **Hvert register er kablet til sin rolle.** Alle postkasser er like, og en adresse er bare en adresse. Et register har en størrelse, en plassering og et sett operasjoner som følger av hva det skal brukes til.
- **Registre har ingen adresse.** De slås ikke opp. Kontrollenheten snakker direkte til dem.

Registre varierer fra én enkelt bit til 128 bit. De minste brukes til å holde ja eller nei, og dem kommer vi til i seksjon 4.8.

4.4.2 De fire du kjenner

Fra kapittel 3 har du fire:

- **Akkumulatoren** holder tallet det regnes med.
- **Programtelleren** holder adressen til neste ordre.
- **Instruksjonsregisteret** holder ordren som utføres nå.
- **Adresseregisteret** holder adressedelen av den ordren.

To av dem brukte vi litt enklere i forrige kapittel. Der lot vi opkoden gå i instruksjonsregisteret og adressedelen i adresseregisteret. I en ekte maskin blir hele ordren liggende i instruksjonsregisteret, og delingen skjer derfra. Begge deler forteller det samme; den siste er bare mer presis.

Adresseregisteret heter dessuten noe annet i en ekte maskin, og det gjør mer enn du tror. Dit kommer vi i neste seksjon.

Note-to-myself

Hvordan et register kan holde på en verdi så lenge strømmen står på, er kapittel 5. Det er det første av tre steder i dette kapitlet der vi stopper rett over porten og peker nedover.

Akkumulatoren har også fått selskap. En moderne prosessor har sjelden ett register til regning. RISC-V har 32, ARM i 64-bits utgave har 31, og x86-64 har 16. De kalles **generelle registre**, fordi programmet kan bruke dem til hva det vil, og samlingen kalles en **registerfil**.

Note-to-myself

Antallet registre og instruksjonsordet henger sammen, og her er koblingen. Trettito registre krever fem bit å peke på, og seksten krever fire. Flere registre gjør programmene raskere, men hvert ekstra registernummer i ordren spiser plass fra de andre feltene. Det er avveiningen fra tabell 4.2 på side 82 én gang til, i ny drakt.

4.4.3 Hvorfor registre finnes i det hele tatt

Et rimelig spørsmål: hvorfor ikke bare bruke minnet?

Svaret er tid. Å hente en verdi fra minnet krever at adressen legges ut, at dekoderen i minnet finner riktig rad, at radene får falle til ro, og at verdien sendes tilbake. Å flytte noe fra ett register til et annet hopper over alt dette.

Derfor er instruksjonssettene i moderne maskiner bygget rundt registre. I ARM og RISC-V er det bare to slags ordrer som rører minnet: den som henter en verdi inn i et register, og den som skriver en verdi ut igjen. Alt annet skjer mellom registre.

Tankevekker

LMC gjør det motsatte. Nesten hver eneste ordre der leser eller skriver en postkasse, fordi det bare finnes ett register å regne i. Modellen er enkel å forstå nettopp fordi den er ineffektiv.

4.4.4 Hva det koster, i tall

Påstanden over er lett å si. Her er den regnet ut.

Tenk deg en løkke som skal telle ned. Et sted ligger en teller, og hver runde skal den minke med én. I LMC blir det slik:

```
LDA teller
SUB en
STA teller
```

Tre ordrer. Tell nå minneoppslagene.

Hver ordre må først hentes fra minnet; det er ett oppslag. Deretter trenger hver av de tre et tall fra minnet eller skal skrive et tall dit; det er ett til. Tre ordrer à to oppslag blir **seks turer til minnet**, bare for å trekke fra én.

En maskin med flere registre lar telleren bli liggende i et register mellom rundene. Da trengs én ordre:

```
SUB  r5, r5, 1
```

Den ordren må hentes fra minnet. Det er **én tur**. Ingen av tallene ligger i minnet, så det blir ingen datatur.

Seks mot én, for akkurat den samme jobben.

Note-to-myself

En kompilator legger mye arbeid i ett spørsmål: hvilke variabler skal få bo i et register? Variabler programmet rører ofte, gir mest igjen, og særlig de som ligger inne i en løkke. Arbeidet har et eget navn i faget: registerallokering.
Dette gjelder språk som oversettes til maskinkode, som C, Rust og Java. Python kjører annerledes, og der skjer det samme først når en såkalt JIT-kompilator oversetter koden mens programmet er i gang.

4.5 Minnet sett fra prosessoren

Kapittel 3 lovet at en ekte maskin har egne registre for å snakke med minnet. Her er de.

- **Minneadresseregisteret**, forkortet **MAR**, holder adressen minnet skal slå opp.
- **Minnedataregisteret**, forkortet **MDR**, holder verdien som er på vei inn eller ut.

Forkortelsene kommer fra engelsk, *memory address register* og *memory data register*, og de brukes også på norsk.

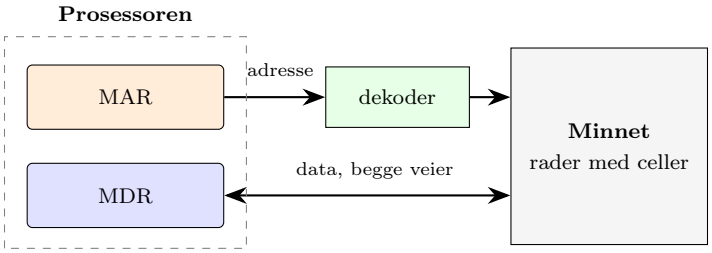
Adresseregisteret du møtte i LMC, er funksjonelt det samme som MAR. Forskjellen er at en ekte maskin bruker det to ganger i de ordrene som rører minnet, ikke én. Først til å hente selve ordren, så til å hente det ordren peker på. Det kommer vi tilbake til i seksjon 4.9.

4.5.1 Den ene går bare én vei

MAR er enveis. Adresser kopieres alltid *inn* i det, aldri ut.

Grunnen er enkel når du ser den. Prosessoren bestemmer selv hvilken adresse som skal brukes. Den har nettopp lagt adressen dit. Det finnes ingen grunn til å spørre minnet hvilken adresse man spurte om.

MDR går begge veier. Skal en verdi skrives, kommer den fra et register inn i MDR og derfra ut i minnet. Skal en verdi leses, kommer den fra minnet inn i MDR og derfra videre til et register.



Figur 4.3 De to registrene som utgjør grensesnittet mot minnet. Adressen går bare én vei, dataene går begge.

4.5.2 Lese eller skrive, og når

To ting mangler fortsatt. Minnet må vite hvilken vei det går, og det må vite når.

Retningen bestemmes av en egen ledning, **lese/skriv-linja**. Står den i den ene stillingen, sendes innholdet i cellen ut til MDR. Står den i den andre, skrives innholdet i MDR inn i cellen.

En annen ledning bestemmer tidspunktet: **aktiveringslinja**. Grunnen er verdt å merke seg. Når adressen i MAR skifter, bruker de elektriske signalene et lite øyeblikk på å stabilisere seg. Underveis kan flere rader stå halvveis påslått. Derfor venter aktiveringslinja til adressen har falt til ro.

En hel minneoperasjon blir da to ting. Først legger prosessoren adressen i MAR og setter lese/skriv-linja. Så slår den på aktiveringslinja et øyeblikk, og overføringen skjer.

Advarsel :)

Her ligger forklaringen på at en raskere klokke ikke alene gjør en maskin raskere. Signalene bruker den tiden de bruker. Klokker du fortere enn kretsene rekker over, leser du av verdier som ikke har falt til ro ennå, og da blir svarene feil i stedet for raske.

4.5.3 Hvorfor gamle maskiner stoppet på fire gigabyte

Nå kommer et regnestykke med stor rekkevidde.

MAR har en bestemt bredde. Er det k bit bredt, kan det inneholde 2^k ulike adresser. Flere adresser enn det finnes ingen måte å uttrykke på. Bredden på MAR setter dermed taket for hvor mye minne maskinen kan ha.

Bredde på MAR	Antall adresser	Minne, med én byte per adresse
16 bit	65 536	64 kilobyte
20 bit	1 048 576	1 megabyte
32 bit	4 294 967 296	4 gigabyte
48 bit	$2,8 \cdot 10^{14}$	256 terabyte
64 bit	$1,8 \cdot 10^{19}$	16 exabyte

Tabell 4.3 Hva bredden på minneadresseregisteret gir av adresserbart minne.

Note-to-myself

Her bruker vi de binære enhetene fra kapittel 2. 2^{32} byte er strengt tatt 4 GiB, altså 4,29 GB. I dagligtale heter grensen «fire gigabyte», og det er den formen du kommer til å møte.

Tredje rad er svaret på et spørsmål mange har stilt. En 32-bits maskin kunne ikke bruke mer enn fire gigabyte minne, uansett hvor mange brikker du plugget inn. Det var ingen vilkårlig grense satt av produsenten. Det var rett og slett slutten på tallene.

Nederste rad viser hvor voldsomt det ble da maskinene gikk til 64 bit. Seksten exabyte er mer minne enn noen kommer til å bygge. Grensen i dag ligger et helt annet sted: hvor mange brikker det er fysisk plass til, og hvor lang tid dekoderen bruker på å finne én rad blant milliarder.

Tenk om...

... du doblet bredden på MAR fra 32 til 64 bit. Du doblet ikke minnet. Du opphøyde det i andre. Hver bit du legger til, dobler antall adresser, og trettito ekstra bit ganger det med over fire milliarder. Det er den samme eksplosjonen du så i kapittel 2.

4.5.4 Når adressefeltet er smalere enn MAR

En siste detalj, som løser en floke du kanskje allerede har sett.

I LMC er adressefeltet i ordren to sifre, og det er nøyaktig like mange postkasser som finnes. De to tallene stemmer overens.

I en ekte maskin gjør de ikke det. Adressefeltet i et 32-bits ord kan være 24 bit, mens MAR er 64 bit bredt. Instruksjonen kan altså bare peke direkte på en liten del av minnet.

Løsningen er å la ordren peke på et *register* som inneholder adressen, i stedet for å bære adressen selv. Registeret er bredt nok, og et registernummer koster bare fem bit. De ulike måtene å angi hvor noe ligger, kalles **adresseringsmåter**. LMC har bare én, den enkleste: adressen står rett i ordren. Den heter direkte adressering.

Note-to-myself

To tall bestemmer altså hver sin ting. Bredden på MAR bestemmer hvor mye minne maskinen *kan ha*. Bredden på adressefeltet bestemmer hvor mye av det en enkelt ordre kan peke på *direkte*. Det første er en hard grense. Det andre lar seg omgå.

4.6 Bussene

Vi har snakket om at adresser og data flytter seg. Nå ser vi på hva de flytter seg gjennom.

En **buss** er et knippe ledninger som frakter signaler fra ett sted til et annet. Hver enkelt ledning kalles en **linje**.

Fun Fact

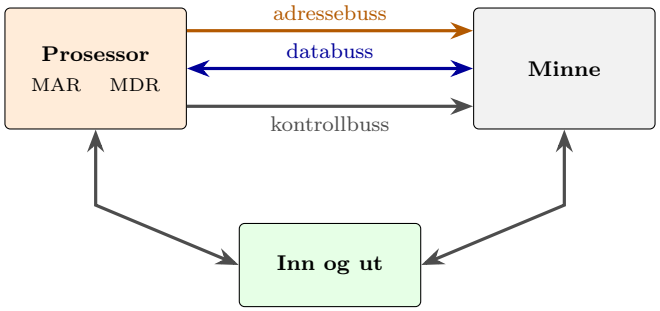
Ordet kommer fra latin *omnibus*, som betyr «for alle». Det er samme opphav som bussen du tar til byen. Poenget var opprinnelig i begge tilfeller at den er delt: mange kan bruke den, men bare én om gangen. Det er altså ikke en forkortelse, og det skrives med liten bokstav.

4.6.1 Tre busser, tre jobber

Tenk på LMC-rommet igjen. Mellom kontoristens pult og veggen med postkasser må tre forskjellige slags beskjeder fram.

- **Hvilken postkasse.** Det er **adressebussen**. Her snakker MAR.
- **Hva som ligger i den.** Det er **databussen**. Her snakker MDR.
- **Når, og hvilken vei.** Det er **kontrollbussen**. Her ligger lese/skriv-linja og aktiveringslinja fra forrige seksjon.

De tre jobber samtidig og uavhengig. Adressebussen bærer et tall som peker. Databussen bærer et tall som betyr noe. Kontrollbussen bærer ingen tall, bare beskjeder om rekkefølge og retning.



Figur 4.4 De tre bussene mellom prosessor og minne. Adressen går bare én vei, dataene begge.

Note-to-myself

Pensumboka tegner dette litt annerledes. Englander snakker om *én* buss med fire kategorier av linjer: data, adresse, kontroll og strøm. Innholdet er det samme, men ordene er ikke. Leser du der og lurer på hvor de tre bussene ble av, er det dette som er svaret.

4.6.2 Hvor mange ledninger

Bussbredde er bokstavelig talt antall ledninger.

En buss som skal bære 64 bit data om gangen, trenger 64 datalinjer. Skal den samtidig bære en 32-bits adresse, kommer 32 adresselinjer i tillegg. Det er 96 ledninger, og da har vi ikke talt med kontrollinjene ennå.

Bredere buss betyr mer data per overføring. Det er den samme tanken som med kjørefelt: en vei med fire felt tar mer trafikk enn en vei med to, ved samme fartsgrense.

4.6.3 Hvorfor brede busser likevel taper

Her kommer en vri som overrasker mange.

Hvis bredere er bedre, hvorfor har ikke alt tusen ledninger?

Fordi signalene ikke kommer fram samtidig. Ledningene er aldri helt like lange, og de ligger ikke helt likt. Ett signal bruker litt lengre tid enn et annet. Forskjellen kalles **skjevhet**, og den er bitteliten, men den er der.

Maskinen må vente til det tregeste signalet er framme før den kan lese av noe som helst. Jo flere ledninger, jo større sjanse for at én av dem henger etter. Skjevheten setter dermed et tak på hvor fort en bred buss kan klokkes.

Løsningen har vært å gå motsatt vei: færre ledninger, men mye høyere fart. En **seriell buss** sender bitene etter hverandre på ett ledningspar i stedet for ved siden av hverandre på mange. Da finnes det ingen skjevhet å vente på, og farten kan skrus langt opp.

Tankevekker

Dette er en avveining som dukker opp om og om igjen i faget. Gjøre mer om gangen, eller gjøre det fortere? Her vant farten. I kapittel 7 møter du det samme spørsmålet igjen, og da med motsatt svar.

De aller fleste bussene ut mot omverdenen er serielle i dag. PCI Express, som grafikkortet sitter på, er et knippe serielle forbindelser som kalles baner. USB er seriell, og navnet sier det: *universal serial bus*.

Minnebussen er det store unntaket. Avstanden dit er kort, og båndbredden betyr så mye at den fortsatt er bred og parallell. Det er altså nettopp de tre bussene i figur 4.4 på forrige side som ikke har gått over.

Inne i prosessoren er avstandene brøkdeler av en millimeter, skjevheten ubetydelig, og nesten alt parallelt.

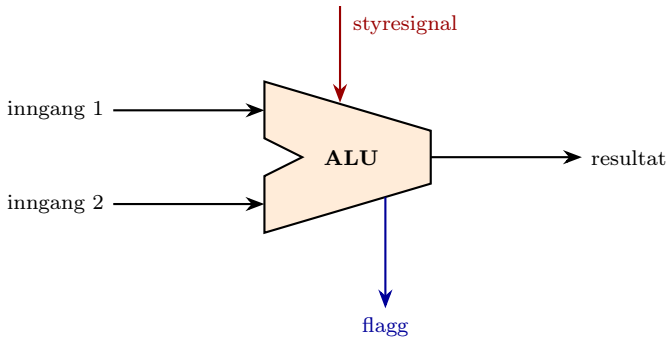
4.7 ALU-en

Et sted i maskinen blir to tall til ett. Det stedet heter **ALU**, etter engelsk *arithmetic logic unit*. På norsk: den aritmetisk-logiske enheten.

I LMC var dette kalkulatoren på pulten. Nå ser vi på den utenfra.

4.7.1 Tre inn, to ut

ALU-en tegnes tradisjonelt som en bred V. Formen har ingen dypere mening, men den er lett å kjenne igjen, så alle bruker den.



Figur 4.5 ALU-en sett utenfra. To tall inn fra siden, én beskjed inn ovenfra, ett svar og noen flagg ut.

Tell inngangene. Det er tre, ikke to.

De to fra siden er data. Det er tallene som skal behandles, og de kommer fra registre eller fra minne-dataregisteret.

Den tredje kommer ovenfra, og den er noe helt annet. Den er ikke et tall som skal regnes med. Den er beskjeden om hva som skal gjøres: legg sammen, trekk fra, sammenlign bit for bit.

Tankevekker

Her møtes de to tunnelgangene.
Styresignalet kommer fra *opkoden*. De fire bitene øverst i instruksjonsordet, de samme som i figur 4.1 på side 81, blir til den ledningen som forteller ALU-en hva den skal gjøre.
Dataene kommer fra registrene. Beskjeden kommer fra ordren. Det er hele koblingen mellom programmet du skrev og kretsene som utfører det.

Ut kommer to ting. Resultatet er det åpenbare. Flaggene er det som skal vise seg å bety mest, og de får en egen seksjon.

4.7.2 Hva den kan

En ALU gjør færre ting enn du skulle tro.

- **Addisjon og subtraksjon.**
- **Logiske operasjoner bit for bit:** OG, ELLER, IKKE, ENTEN-ELLER.
- **Skift og rotasjon:** flytte alle bitene ett eller flere hakk til siden.

Det er omtrent det. Multiplikasjon, divisjon, kvadratrot og sinus står ikke på lista i den klassiske ALU-en vi tegner her. Mange moderne prosessorer har egne kretser for multiplikasjon ved siden av.

4.7.3 Hvorfor addisjon holder så langt

Mye kan bygges av addisjon alene.

Subtraksjon er allerede løst. I kapittel 2 så du at $a - b$ kan regnes som $a + (-b)$ når negative tall skrives i toerkomplement. Maskinen trenger ingen egen subtraksjonskrets; den inverterer og legger til.

Multiplikasjon kan gjøres med gjentatt addisjon. $5 \cdot 3$ er $5 + 5 + 5$. I praksis gjør maskinen noe smartere, med skift og addisjon, men grunnoperasjonen er den samme.

Heltallsdivisjon kan gjøres med gjentatt subtraksjon. Du teller hvor mange ganger du får trukket fra.

Advarsel :)

Her er det lett å ta for hardt i. Du vil høre påstanden om at «alle beregninger kan brytes ned til addisjon», og den holder ikke helt.
Desimaltall, kvadratrøtter og trigonometri lar seg ikke løse med heltallsaddisjon på noen praktisk måte. Det er nettopp derfor moderne prosessorer har en egen **flyttallsenhet** ved siden av ALU-en. Hadde addisjon vært nok, hadde den enheten vært overflødig.

4.7.4 Mange små, satt sammen

ALU-en i figuren over behandler hele tall. Inni er den bygget av enheter som behandler én bit hver.

En 64-bits ALU består av 64 slike. For addisjon må de snakke sammen: menten fra én posisjon må videre til neste, akkurat som når du legger sammen på papir. Det gir en kjede.

For de logiske operasjonene finnes ingen slik kjede. ELLER mellom to tall regnes ut bit for bit, helt uavhengig, så alle 64 posisjonene blir ferdige samtidig. Signalene har dermed kortere vei gjennom kretsen enn ved en addisjon.

Det betyr ikke uten videre at ordren tar færre klokkeslag. Klokka er stilt etter det tregeste steget uansett. Men det er en av tingene som bestemmer hvor fort klokka kan gå.

Note-to-myself

Innmaten i ALU-en åpner vi ikke her. Hvordan en enkelt bit faktisk legges sammen, og hvorfor mentekjeden koster tid, er kapittel 5. Dette kapitlet stopper ved grensesnittet: tre inn, to ut. Det er det andre av de tre stedene vi peker nedover.

4.8 Flaggene

ALU-en gir fra seg svaret. Den gir også fra seg noen bemerkninger om svaret.

Et **flagg** er et register på nøyaktig én bit. Det er enten oppe eller nede, ett eller null, ja eller nei. Flaggene ligger samlet i et **statusregister**.

4.8.1 De fire vanlige

Flagg	Navn	Settes når
Z	null	resultatet ble null
N	negativ	resultatet ble negativt, altså når øverste bit er 1
C	mente	det ble mente ut av øverste posisjon
V	overflyt	svaret ikke fikk plass, og fortegnet ble feil

Tabell 4.4 De fire vanlige flaggene. Bokstavene kommer fra engelsk: zero, negative, carry, overflow.

Flaggene settes på nytt hver gang ALU-en regner. Du ber ikke om dem. De kommer uansett, som en kvittering.

Note-to-myself

Pensumboka bruker ikke bokstavene Z, N, C og V. Englander skriver «carry flag» og «overflow flag», og omtaler de to andre bare som negativ eller positiv, null eller ikke null. Innholdet er det samme, men søker du etter bokstavene i boka, finner du dem ikke.

4.8.2 Slik bruker et betinget hopp dem

Nå henger to ting sammen som hittil har stått hver for seg.

Et betinget hopp spør om noe. I en ekte maskin spør det ikke om et tall. Det spør om et flagg.

Vil du sammenligne to tall, trekker du det ene fra det andre. Du bryr deg ikke om differansen. Du bryr deg om flaggene som ble satt underveis.

Ble Z satt, var tallene like. Så langt er det greit.

Er de ulike, må du se på N og V sammen. Er de like, var det første tallet størst. Er de ulike, var det minst.

At V må være med, kan virke rart. Grunnen ser du i tabellen lenger ned: overflyt snur fortegnsbitet, og da lyver N. Sammenligner du $+7$ med -2 i fire bit, blir svaret negativt selv om sju er størst, og N alene ville gitt feil konklusjon.

Dette er grunn god nok til at ingen regner det ut for hånd. Ekte instruksjonssett har egne hoppordrer som gjør hele vurderingen, slik at programmet bare sier «hopp hvis det første var størst».

Derfor finnes det også ordrer som trekker fra, kaster svaret og beholder flaggene. De heter gjerne noe med «sammenlign».

Advarsel :)

LMC har ingen flagg i det hele tatt.
`BRZ` og `BRP` ser rett på akkumulatoren. Kontoristen går bort til kalkulatoren og titter på tallet som står der. Det er en forenkling i modellen, og den er bevisst: ett begrep mindre å holde styr på. Du bør likevel vite at ekte maskiner gjør det annerledes, for det forklarer hvorfor sammenligning og hopp er to adskilte ordrer der, mens de er smeltet sammen i LMC.

4.8.3 Mente og overflyt er ikke det samme

De to siste flaggene forveksles hele tiden. Forskjellen er verdt fire linjer i en tabell.

Vi regner i toerkomplement med fire bit. Da rekker tallene fra -8 til $+7$.

Regnestykke	**Bit**	**Flagg**	**Resultat**
$(+4) + (+2)$	$0100 + 0010 = 0110$	ingen	$+6$, riktig
$(+4) + (+6)$	$0100 + 0110 = 1010$	V	-6, galt
$(-4) + (-2)$	$1100 + 1110 = 1\,1010$	C	-6, riktig
$(-4) + (-6)$	$1100 + 1010 = 1\,0110$	C og V	$+6$, galt

Tabell 4.5 Alle fire kombinasjonene er mulige. Legg merke til hvilke rader som gir galt svar.

Les den nedenfra og opp, så faller det på plass:

Mente alene er ufarlig. Rad tre fikk mente, og svaret ble riktig. Menten falt ut i den ene enden og ble kastet, slik den skal når vi regner med fortegn.

Overflyt betyr feil. Rad to og fire fikk overflyt, og begge ga galt svar. $4 + 6$ ble -6. Det er ikke en avrunding; det er helt feil tall, med feil fortegn.

Regelen bak rad to er grei å huske. Ved addisjon kan overflyt bare skje når begge tallene har samme fortegn, og den kjennes igjen på at svaret har motsatt fortegn av dem. Ved subtraksjon er det omvendt: der må fortegnene være ulike. Det følger av at maskinen uansett regner $a - b$ som $a + (-b)$.

Tenk om...

... ingen sjekker flagget.
Et program legger sammen to store tall. Overflyt-flagget går opp. Maskinen gjør ingenting med det av seg selv; den legger bare det gale svaret i akkumulatoren og går videre til neste ordre.
Python sier fra når slikt skjer. Grunnen er at noen har skrevet kode som leser flagget og kaster en feilmelding. Det skjer ikke gratis, og det skjer ikke i alle språk. Under der ligger det et flagg og vifter, og det er opp til programmet om noen ser etter.
Dette er ikke en teoretisk bekymring. I kapittel 2 leste du om en rakett som gikk tapt fordi et tall ikke fikk plass.

4.8.4 Hva mente brukes til

Hvis menten bare kastes, hvorfor har vi flagget?

Fordi den ikke alltid kastes. Skal en maskin med 32-bits ALU legge sammen to tall på 64 bit, må den gjøre det i to omganger. Først de nederste 32 bitene, så de øverste. Menten fra den første addisjonen må med inn i den andre.

Det er nøyaktig det samme du gjør når du legger sammen store tall på papir, bare med 32 bit per kolonne i stedet for ett siffer.

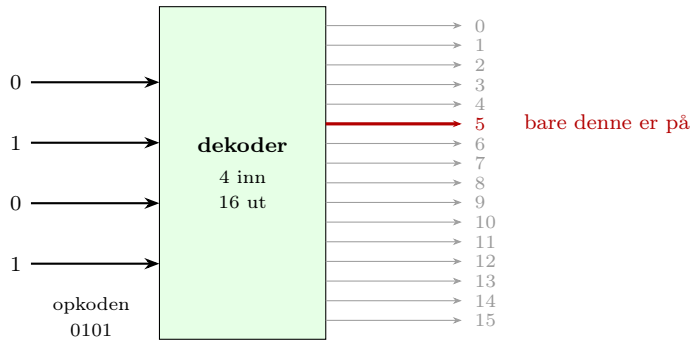
4.9 Kontrollenheten og datastien

Nå har vi delene. Det som mangler, er det som får dem til å samarbeide.

4.9.1 Fra opkode til styresignal

Kontrollenheten leser opkoden og slår på de riktige ledningene. Spørsmålet er hvordan.

Svaret er mindre magisk enn man skulle tro. En **dekoder** er en krets som tar inn noen få bit og har én utgang per mulig kombinasjon. Til enhver tid er nøyaktig én utgang på.



Figur 4.6 En dekoder med fire innganger har seksten utganger. Opkoden 0101 slår på linje nummer fem, og bare den.

Fire bit opkode gir seksten utganger, én per mulig ordre. Opkoden 0101 slår på linje fem, som er `LDA`-linja. Den linja åpner akkurat de bryterne `LDA` trenger, og lar resten være.

Englander beskriver det slik: bitene i instruksjonsregisteret, kombinert med klokkepulsen, åpner og lukker elektroniske brytere i riktig rekkefølge. Ingenting mer mystisk enn det.

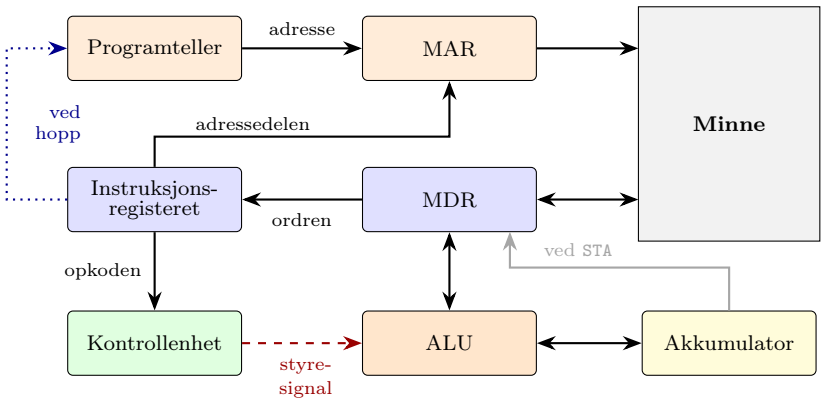
Note-to-myself

Dekoderen bygges av logiske porter. Hvordan, er kapittel 5. Dette er det tredje og siste stedet i kapitlet der vi stopper rett over porten og peker nedover. Det er med vilje: alt henger sammen, og det neste kapitlet er der det knytes.

4.9.2 Datastien

Datastien, eller *datapath* som den ofte kalles også på norsk, er samlingen av veier data kan ta inne i prosessoren. Registrene, ALU-en, minnegrensesnittet, og bryterne mellom dem.

Kontrollenheten bygger ingen nye veier. Den velger hvilke som er åpne i hvert øyeblikk.



Figur 4.7 En forenklet datasti. Den prikkede veien brukes bare ved hopp, og den grå bare ved `STA`. En ekte datasti har flere veier enn disse.

Legg merke til pila fra instruksjonsregisteret tilbake til MAR. Den er lett å overse og viktig å få med.

MAR brukes nemlig to ganger per ordre. Først får det adressen fra programtelleren, for å hente selve ordren. Så får det adressen fra ordrens eget adressefelt, for å hente det ordren peker på. Samme register, to jobber, rett etter hverandre.

4.9.3 Én ordre, steg for steg

Vi følger `LDA 42` gjennom datastien. Anta at ordren ligger i adresse 17, og at tallet 83 ligger i adresse 42.

1. Programtelleren inneholder 17. Tallet kopieres over i MAR. Minnet slår opp adresse 17, og ordren havner i MDR. Samtidig klikkes programtelleren fram til 18.
2. Ordren kopieres fra MDR over i instruksjonsregisteret. Der blir den liggende til den er utført, og minneregistrene er ledige igjen.
3. Ordren deles. Opkoden 0101 går til kontrollenheten, som slår på `LDA`-linja. Adressedelen 00101010, altså 42, kopieres til MAR.
4. Minnet slår opp adresse 42. Verdien 83 havner i MDR.
5. Verdien kopieres fra MDR til akkumulatoren. Ordren er utført.

Note-to-myself

Dette er de samme fire stegene som i kapittel 3, bare finere oppdelt. Steg 1 er «hent». Steg 2 og 3 er «del». Steg 4 og 5 er «utfør». Og så begynner det på nytt.
Ulike framstillinger teller ulikt. Pensumboka setter opp fem steg for `LDA`, men der er det siste steget å øke programtelleren, som vi har lagt inn i steg 1. Boka sier selv at det kan gjøres når som

helst etter hentingen, og at det vanligvis gjøres tidlig. Det er en måte å telle på, ikke en uenighet om hva maskinen gjør.

4.9.4 De samme fem stegene, sett i registrene

Den letteste måten å få tak i hva som skjer, er å se på hva som står i hvert register underveis. En strek betyr at innholdet er uten betydning akkurat da.

Etter steg	Teller	MAR	MDR	Instr.reg.	Akk.
Før vi begynner	17	—	—	—	9
1. Ordren hentet	18	17	542	—	9
2. Ordren lagret	18	17	542	542	9
3. Ordren delt	18	**42**	542	542	9
4. Tallet hentet	18	42	**83**	542	9
5. Ordren utført	18	42	83	542	**83**

Tabell 4.6 `LDA 42` sporet gjennom registrene. Verdien som nettopp endret seg, står i fet skrift.

To ting er verdt å stoppe ved i tabellen.

Programtelleren endrer seg i steg 1 og aldri mer. Den ble klikket fram i det ordren ble hentet, akkurat som i kapittel 3, og så er den ferdig for denne runden.

MAR endrer seg i steg 1 og i steg 3. Først holder det adressen til ordren, så holder det adressen ordren peker på. Dette er den doble bruken vi nevnte over, og tabellen gjør den synlig.

Noe er også verdt å merke seg ved lista. Det meste av arbeidet er flytting. Tre av de fem stegene gjør ikke annet enn å kopiere fra ett register til et annet, og de to øvrige er minneoppslag. Det eneste regnestykket i hele runden er at telleren økes med én.

4.9.5 De andre ordrene

En `ADD 42` ville vært helt lik, bortsett fra steg 5. Der ville verdien i MDR og verdien i akkumulatoren gått inn i ALU-en, og summen tilbake til akkumulatoren.

En `STA 42` går likt til og med steg 3. Så snur retningen. Akkumulatoren sender verdien til MDR. Kontrollenheten setter lese/skriv-linja til skriv. Nå er det minnet som tar imot i stedet for å gi fra seg, og det som lå i postkasse 42 fra før, er borte.

Note-to-myself

Legg merke til hvor lite som skiller de tre ordrene. De tre første stegene er helt like, og for `LDA` og `ADD` er det bare det siste som skiller.
Det er derfor en prosessor kan bygges av så få deler: det meste av arbeidet er likt uansett hva ordren ber om.

4.9.6 Et hopp går en annen vei

Hoppene er annerledes, og forskjellen er lærerik.

Vi følger `BRZ 7`, som ligger i adresse 18. Ordren betyr: hopp til adresse 7 hvis akkumulatoren er null.

1. Programtelleren står på 18. Tallet går til MAR, ordren kommer inn i MDR, og telleren klikker fram til 19.
2. Ordren går videre fra MDR til instruksjonsregisteret.
3. Opkoden 0111 går til kontrollenheten, som slår på `BRZ`-linja.
4. Nå ser kontrollenheten på akkumulatoren. Er den null, går adressedelen 7 inn i *programtelleren*, som dermed står på 7. Er den ikke null, skjer ingenting. Telleren blir stående på 19.

Se hvor adressedelen havnet.

I `LDA` gikk den til MAR, for å hente noe. Her går den til programtelleren, for å bestemme hvor neste ordre skal hentes fra. Samme felt i ordren, helt ulik mottaker. Hvilken av dem som åpnes, avgjøres av opkoden.

Minnet røres ikke etter at ordren er hentet. Et hopp trenger bare fire steg, mens `LDA` trengte fem.

Tankevekker

Nå ser du hvorfor telleren måtte klikkes fram allerede i steg 1.
Hadde maskinen ventet med å øke telleren til slutten av runden, ville den først satt den til 7 og deretter økt den til 8. Hoppet ville bommet med én, hver eneste gang.
Kapittel 3 slo fast regelen og ba deg stole på den. Her er maskineriet som gjør at den må være slik.

Tankevekker

Legg merke til hvor lite som egentlig skjer. En prosessor er for det meste en maskin som flytter tall mellom små lagringsplasser i en bestemt rekkefølge. Regningen er et lite hjørne av arbeidet.
Forrige kapittel sa at kraften i en datamaskin kommer av gjentakelse, ikke av at operasjonene er avanserte. Her ser du hvorfor det måtte bli slik. En maskin som bare flytter og legger sammen, kan bygges av få og like deler, og få og like deler kan gjøres små og raske. Enkelheten er ikke en begrensning maskinen lever med. Den er betingelsen for farten.

Fun Fact

Et spørsmål som melder seg av seg selv: hva står i programtelleren i det strømmen slås på?
Svaret er en fast adresse, bygget inn i maskinvaren. Prosessoren begynner alltid på samme sted. Der ligger starten på et lite program som er lagret i et minne som ikke glemmer når strømmen går. Det programmet sjekker maskinvaren og laster inn operativsystemet.
En prosessor kan altså ikke starte med ingenting. Det må alltid ligge noe på den ene adressen den våkner på.

4.9.7 Klokka

Hvert steg må være ferdig før neste begynner. Noe må holde takten, og det er **klokka**.

Klokka er et signal som veksler mellom null og ett i et fast tempo. Hvert slag slipper neste steg løs.

Tempoet kan ikke settes fritt. Når data kopieres fra ett register til et annet, tar det en liten stund før verdien har falt til ro. Noen operasjoner tar lengre tid enn andre; en addisjon tar mer enn en ren

flytting, og et minneoppslag tar mest. Klokkeslagene må ligge langt nok fra hverandre til at det tregeste steget rekker over.

Maskin	Klokkefrekvens	Omtrent antall steg per sekund
Tidlig hjemmedatamaskin	4,77 MHz	4,8 millioner
Mobiltelefon i dag	1,5–3 GHz	1,5–3 milliarder
Bærbar maskin i dag	1,5–5,5 GHz	opptil 5,5 milliarder

Tabell 4.7 Klokkefrekvens er antall slag per sekund. Ett slag er grovt sett ett steg.

Tar en ordre ti steg, klarer den tidlige maskinen rundt en halv million ordrer i sekundet. Det hørtes raskt ut den gangen.

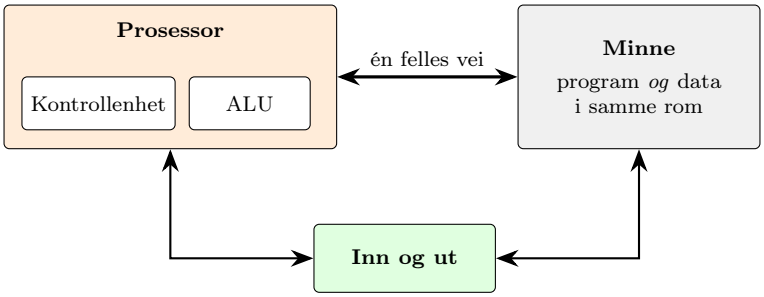
Advarsel :)

Klokkefrekvens er et dårlig mål på hvor rask en maskin er, og du vil se den brukt som om den var et godt et.
To maskiner med samme frekvens kan bruke ulikt antall steg på den samme ordren. Og en maskin som jobber med flere ordrer samtidig, gjør mer per slag enn en som tar én om gangen. Hvordan det går til, er kapittel 7.

4.10 Modellen som binder det sammen

Alt i dette kapitlet følger ett og samme oppsett. Det har et navn, og nå som du har sett delene, kan vi sette det på.

Von Neumann-arkitekturen er et sett retningslinjer for hvordan en datamaskin bygges. De ble skrevet ned i årene rundt 1945, og de gjelder fortsatt.



Figur 4.8 Von Neumann-modellen. Program og data deler minne, og de deler veien dit.

4.10.1 De fire reglene

1. **Minnet holder både program og data.** Dette kalles *lagret program*, og det er den viktigste av dem.
2. **Minnet adresseres lineært.** Hver plass har ett nummer, og numrene går i rekkefølge.
3. **Adressen sier ingenting om innholdet.** Plass 42 er plass 42 enten det ligger en ordre eller et tall der.
4. **Ordrene utføres i rekkefølge**, til noe får maskinen til å hoppe.

Alle fire kjenner du igjen. Hele kapittel 3 var en lang øvelse i regel 1 og 3.

4.10.2 Kjøkkenet

Et bilde som holder hele veien.

Amina lager middag. Oppskriften ligger i en perm, og råvarene står i det samme skapet som permen. Hun leser ett steg av gangen, finner det som trengs i skapet, gjør det steget sier, og går tilbake for neste linje.

Det geniale ved oppsettet er at oppskriften og råvarene behandles likt. Begge deler er bare ting i et skap. Det er derfor den samme maskinen kan kjøre en tekstbehandler det ene øyeblikket og spille musikk det neste: du bytter bare ut permen.

Tankevekker

Her lukkes sirkelen fra forrige kapittel.
Programmet som redigerte seg selv, virket fordi oppskriften lå i samme skap som råvarene. Maskinen kan skrive til en plass i minnet uten å bry seg om hva som lå der, og neste gang telleren peker dit, leses det som en ordre.
Lagret program er altså ikke en detalj. Det er regelen som gjør alt det rare i kapittel 3 mulig.

4.10.3 Flaskehalsen

Oppsettet har en pris, og den ser du først nå som du vet hva en buss er.

Program og data ligger i samme minne. De må derfor dele den samme veien inn til prosessoren. Hver eneste ordre må hentes over den bussen, og hvert eneste tall ordren trenger, må hentes over den samme bussen.

Prosessoren kan gjøres aldri så rask. Alt må uansett gjennom den ene åpningen.

Dette kalles **von Neumann-flaskehalsen**. En flaskehals er stedet som bestemmer farten på hele systemet, uansett hvor god resten er. Navnet kommer av at innholdet i en flaske ikke renner fortere ut enn halsen tillater.

Note-to-myself

Uttrykket står ikke i pensumboka. Englander beskriver problemet, men bruker ikke navnet. Det er et vanlig begrep i faget ellers, og det er greit å kjenne.

Store deler av kapittel 7 handler om å leve med denne flaskehalsen: legge ofte brukte data nærmere prosessoren, og sette flere ordrer i arbeid samtidig.

4.10.4 En merknad om navnet

Arkitekturen bærer ett navn, og den fortjener flere.

Rapporten som beskrev lagret program, kom i juni 1945. Den ble skrevet ved University of Pennsylvania, og John von Neumann sto som eneste navngitte forfatter. Innholdet bygget på arbeid gjort av en hel gruppe, der blant andre J. Presper Eckert og John Mauchly hadde sentrale bidrag.

Fun Fact

Dette er en ganske vanlig historie i teknologihistorien. Navnet som henger ved en idé, er ofte navnet på den som skrev den ned, ikke på alle som fant den. Det er verdt å ha i bakhodet hver gang en oppfinnelse har nøyaktig én oppfinner.

4.11 Oppsummering

Kapitlet begynte med et fjell som bores fra to sider, og endte med et kjøkken. Imellom gikk vi gjennom det ene møtepunktet: ordren, skrevet i bit.

4.11.1 Hva som egentlig var nytt

Mindre enn det kan virke som. Det meste av dette kapitlet er kapittel 3 med riktige navn på.

I LMC	**I en ekte maskin**
Rommet med hundre postkasser	Minnet, adressert lineært
Kontoristen	Kontrollenheten
Kalkulatoren	ALU-en
Telleren	Programtelleren
Instruksjonsregisteret	Instruksjonsregisteret
Adresseregisteret	MAR, men brukt to ganger per ordre
Ingenting tilsvarende	MDR, som LMC ikke trengte
Ett siffer opkode, to adresse	Felt av bit, med bredder arkitekten valgte
Akkumulatoren, alene	En registerfil med 16 eller 32
`BRZ` ser på akkumulatoren	Hoppet ser på et flagg
Ingenting tilsvarende	Bussene, som alt må gjennom

Tabell 4.8 Modellen fra forrige kapittel, satt opp mot virkeligheten.

De to radene uten motstykke i venstre kolonne er de to viktigste. Minnedataregisteret og bussene finnes ikke i LMC, fordi LMC ikke trengte å forklare hvordan noe *flytter* seg. Kontoristen bare gikk bort og hentet.

Begrep	Kort sagt
ISA	Avtalen. Hvilke ordrer maskinen forstår.
Mikroarkitektur	Byggingen. Hvordan den faktisk gjør det.
Instruksjonsord	Ordren som bitmønster, delt i felt med fast bredde.
Opkodebredde	Bestemmer hvor mange ulike ordrer som finnes.
Adressebredde	Bestemmer hvor langt en ordre kan peke. Tar plass fra opkoden.
Register	Fast lagringsplass inne i prosessoren, viet én oppgave. Har ingen adresse.
MAR	Holder adressen minnet skal slå opp. Enveis. Bredden setter taket for minnet.
MDR	Holder verdien på vei inn eller ut. Toveis.
De tre bussene	Adressebuss: hvilken plass. Databuss: hva som ligger der. Kontrollbuss: når og hvilken vei.
Bussbredde	Antall ledninger. Bredere gir mer per gang, men tåler lavere fart.
ALU	To tall inn, ett styresignal ovenfra, resultat og flagg ut.
Flagg	Ettbits kvittering på siste regnestykke: Z, N, C, V.
Mente mot overflyt	Mente alene er ufarlig. Overflyt betyr galt svar.
Dekoder	Noen bit inn, én linje på av mange. Slik blir opkoden til styresignaler.
Datasti	Veiene data kan ta inne i prosessoren.
Von Neumann	Program og data i samme minne, adressert etter nummer.
Flaskehalsen	Alt må gjennom den samme veien til minnet.

Tabell 4.9 Begrepene fra dette kapitlet.

Den gjennomgående ideen kom igjen fire ganger. Opkoden mot adressen i instruksjonsordet. Antall registre mot plassen de koster i ordren. Bussbredde mot klokkefart. Og til slutt hele von Neumann-modellen, som er enkel og fleksibel og derfor har en flaskehals.

Hver gang er det den samme formen på valget: noe du gjerne vil ha mer av, som bare kan vokse på bekostning av noe annet.

4.11.2 Videre til neste kapittel

Vi har stoppet tre ganger og pekt nedover.

Vi åpnet ikke ALU-en. Vi åpnet ikke dekoderen. Vi sa at registre kan huske, uten å si hvordan noe kan huske så lenge strømmen står på.

Alle tre svarene er det samme svaret, og det ligger ett lag ned. Kapittel 5 bygger kretser som kan regne og velge, av byggeklosser som bare kan si ja og nei.

Den andre tunnelgjengen er på vei oppover. Nå går vi ned og møter den.

4.12 Videre lesning

Englander, kapittel 7. Hovedkilden til dette kapitlet. Seksjon 7.1 til 7.4 dekker komponentene, registrene, minnet og hent-utfør-syklusen. Seksjon 7.6 og 7.7 dekker instruksjonsordet. Fire ting å være forberedt på:

- Boka skriver hent-utfør med en pilnotasjon vi har unngått her.
- Den viser aldri syklusen for et hopp, men lar det stå som oppgave.
- Den bruker ikke bokstavene Z, N, C og V om flaggene.
- Den definerer programtelleren som adressen til ordren som utføres nå. Vi har holdt oss til at den peker på den neste, slik kapittel 3 gjorde.

Englander, kapittel 8, avsnittet om klokka. Her står beskrivelsen av hvordan klokkepulsene kombineres med innholdet i instruksjonsregisteret for å styre bryterne, og tallene i tabell 4.7 på side 99.

Englander, kapittel 6.6. Von Neumann-arkitekturen, kort og presist. Merk at flaskehalsen ikke nevnes der.

Englander, kapittel 9.6. Busser. Her tegnes det som én buss med flere slags linjer, ikke som tre busser slik vi har gjort.

Englander, kapittel 5, side 127. Mente og overflyt, med tabellen vi lånte.

4.13 Refleksjon

1. En arkitekt skal lage en maskin med instruksjonsord på 16 bit. Hun vurderer 5 bit opkode mot 6 bit opkode. Hva taler for hver av dem, og hvilke opplysninger trenger hun for å velge?
2. LMC lar betingede hopp se rett på akkumulatoren, mens ekte maskiner går veien om flagg. Hva vinner en ekte maskin på den omveien?
3. Vi har sagt at bredere buss gir mer data per gang, men lavere mulig klokkefart. Finn et eksempel på den samme avveiningen fra et helt annet område enn datateknologi.
4. Von Neumann-modellen har vært nærmest uendret i over sytti år, i et fag der alt annet skiftes ut. Hva tror du det kommer av?

4.14 Selv-test

Fasit står i fasitdelen bakerst.

1. Forklar med egne ord forskjellen på en ISA og en mikroarkitektur.
2. To prosessorer kjører de samme programmene, men den ene er dobbelt så rask. Hva er likt, og hva er forskjellig?
3. Et instruksjonsord er 16 bit, med 5 bit opkode. Hvor mange ulike ordrer kan maskinen ha, og hvor mange adresser kan en ordre peke på?
4. Skriv `ADD 13` som tolv bit, med fire bit opkode og åtte bit adresse.
5. Bitmønsteret 0011 00001010 er en LMC-ordre i tolv bit. Hvilken?
6. Hvorfor kan ikke en deling på 6 bit opkode og 6 bit adresse kjøre LMC?
7. I LMC øker programtelleren med én. I en 32-bits maskin øker den med fire. Forklar hvorfor.
8. Nevn to ting som skiller et register fra en plass i minnet.
9. Hvorfor har moderne prosessorer mange generelle registre, og hva koster det i instruksjonsordet?
10. Hvorfor går adresser bare én vei inn i MAR?

11. Hva gjør aktiveringslinja, og hvorfor trengs den?
12. En maskin har et MAR på 20 bit. Hvor mange adresser kan den nå?
13. Forklar hvorfor en 32-bits maskin ikke kunne bruke mer enn fire gigabyte minne.
14. Hvilken buss snakker MAR på, og hvilken snakker MDR på?
15. En parallell buss skal bære 32 bit data og 24 bit adresse samtidig. Hvor mange linjer trengs, før kontrollinjene telles med?
16. Hva er skjevhet, og hvorfor er den et argument for serielle busser?
17. Hvor mange innganger har en ALU, og hva slags informasjon bærer hver av dem?
18. Hvor kommer styresignalet til ALU-en fra?
19. Hvorfor kan en ALU gjøre ELLER på 64 bit raskere enn en addisjon på 64 bit?
20. En ALU med fire bit regner $(-5) + (-6)$ i toerkomplement. Hvilke flagg blir satt, og er svaret riktig?
21. Et program legger sammen to tall og får et svar som er helt feil, uten at noe stopper. Hvilket flagg burde vært sjekket, og hvorfor stoppet ikke maskinen av seg selv?
22. Hvor mange utganger har en dekoder med fem innganger, og hvor mange av dem er på samtidig?
23. Følg `ADD 30` gjennom datastien, steg for steg. Hvilket steg er det eneste som skiller den fra `LDA 30`?
24. MAR brukes to ganger i løpet av én ordre. Hva holder det hver gang?
25. Nevn de fire reglene som definerer von Neumann-arkitekturen.
26. Forklar von Neumann-flaskehalsen, og hvorfor den ikke forsvinner av at prosessoren blir raskere.

Kapittel 5

Porter og logiske funksjoner

Innhold

5.1 Læringsmål . . . 106
5.2 Det begynner med en bryter . . . 107
5.3 Portene . . . 112
5.4 Sannhetsverditabellen . . . 117
5.5 Fra tabell til uttrykk . . . 120
5.6 Forenkling . . . 122
5.7 Kretsen som legger sammen . . . 126
5.8 Mentekjeden . . . 131
5.9 Kretser som velger . . . 135
5.10 Kretsen som husker . . . 138
5.11 Oppsummering . . . 139
5.12 Videre lesning . . . 140
5.13 Refleksjon . . . 141
5.14 Selv-test . . . 141

Et sted i koden din står det en linje som ligner på denne:

```
if alder >= 18 and har_billett:
    slipp_inn()
```

Du har skrevet noe lignende selv. Jobben gjøres av tre små ord: **AND**, **OR** og **NOT**. De virker like selvsagte som komma og punktum, og du brukte dem riktig lenge før noen forklarte hvordan de fungerer.

Note-to-myself

I Python skriver du dem med små bokstaver, fordi språket krever det. I denne boka skriver vi dem med **store**: AND, OR, NOT. Da holder vi den logiske operasjonen adskilt fra kodebiten, og det er også slik de står på portsymbolene og i faglitteraturen.

Men hvordan vet egentlig maskinen hva AND betyr?

Den er laget av sand, metall og strøm. Ingen sitter inni og leser koden din, og likevel blir svaret riktig hver eneste gang, milliarder av ganger i sekundet. Da må AND finnes som en fysisk *ting* der nede, noe du kunne pekt på hvis du hadde et godt nok mikroskop og visste hvor du skulle se.

Den tingen skal vi finne i dette kapitlet. Den er mindre og dummere enn du tror.

Tankevekker

Alt maskinen gjør er bygget av deler som bare kan svare ja eller nei på ett eller to signaler. Ingen av dem vet hva et tall er, og ingen vet hva addisjon er. Likevel klarer de å legge sammen, og før du er ferdig med dette kapitlet har du selv bygget kretsen som gjør det.

Vi starter med noe alle har vært borti: en bryter og en lampe. Derfra tar vi ett lite skritt om gangen, og hvert skritt er kort nok til at du kan følge det i hodet.

Du trenger ingen matematikk fra før. Det eneste du må godta, er at en påstand enten er sann eller usann. Har du følt at tidligere kapitler forutsatte noe du ikke hadde, kan du slappe av her. Vi lager oss riktignok en liten skrivemåte underveis, med noen få tegn som sparer plass, men den lærer du på stedet, og den er kortere enn alfabetet.

Stoffet er nytt og ligner ikke på noe annet i faget. De fleste må se det to ganger før det sitter, og det sier ingenting om hvor flink du er. Ta det i ditt eget tempo, og ha gjerne et ark ved siden av deg der du tegner med.

Vi kommer også med noe vi skylder deg. Kapittel 4 stoppet tre ganger og pekte nedover. Vi åpnet verken ALU-en eller dekoderen, og vi sa at et register kan huske, uten å forklare hvordan noe kan huske så lenge strømmen står på. Her får du alle tre svarene, og det er det samme svaret hver gang.

Når du er ferdig med kapitlet, har du bygget en krets som legger sammen to tall, ut fra en tabell du selv har satt opp. Du får også vite hvorfor en addisjon tar lengre vei gjennom maskinen enn en sammenligning, og hvor to av flaggene fra forrige kapittel kommer fra, rent fysisk. Det viser seg å være to ledninger og én port.

Dette er også nyttig utenfor faget. Telefonen i lomma har flere milliarder av disse små delene. Hver av dem tar plass, bruker strøm og blir varm, og hvis du bruker dobbelt så mange som du trenger, betaler du for det på alle tre måtene samtidig. En stor del av kapitlet handler derfor om å ikke sløse. Det er den samme vanen som når du skriver en betingelse uten å sjekke det samme to ganger.

Vi begynner med lampa.

5.1 Læringsmål

Etter å ha arbeidet med dette kapitlet skal du kunne:

1. **Forklare forskjellen på en funksjon og en krets.** Dette skillet er grunnlaget for alt det andre i kapitlet.
2. **Kjenne igjen de sju portene og si hva hver av dem gjør.** Du skal kunne gå fra symbol til tabell til uttrykk og tilbake, uten å pugge noe av det.
3. **Sette opp en sannhetsverditabell for en krets, og lese en krets begge veier.** Framlengs fra inngangene og baklengs fra utgangen. De to måtene avslører ulike ting.
4. **Skrive funksjonsuttrykket som hører til en tabell.** Dette er metoden resten av kapitlet bygger på.

5. **Forenkle et boolsk uttrykk med et lite sett regler.** Og vite når du er ferdig, som er vanskeligere enn det høres ut.
6. **Forklare hvordan en krets legger sammen to bit.** Halvadderen og fulladderen, utledet fra tabellen i stedet for bare vist fram.
7. **Forklare hvorfor mentekjeden koster tid.** Her ligger svaret på hvorfor en addisjon kan ta lengre tid enn en ELLER, og hvor to av flaggene fra forrige kapittel fysisk kommer fra.
8. **Bygge en dekoder av porter.** Da ser du hele veien fra opkoden i et instruksjonsord til ett enkelt styresignal.

5.2 Det begynner med en bryter

5.2.1 Sant og usant, skrevet som ett bit

Når maskinen kommer til linja `if alder >= 18`, må den finne ut én ting: er påstanden sann eller ikke?

Det finnes bare to mulige svar. Ikke «nesten», ikke «litt», ikke «kommer an på». Enten er den sann, eller så er den usann.

To svar er akkurat det et bit rommer. Maskinen skriver derfor sann som **1** og usann som **0**, og hele spørsmålet er dermed redusert til ett eneste bit.

Dette er den samme ideen som bar kapittel 2: et bitmønster betyr ingenting i seg selv, det er avtalen om hva det skal bety som gir det mening. Der var avtalen at bitene var et tall eller en bokstav. Her er avtalen at ett bit er et ja eller et nei. Samme bit, ny avtale.

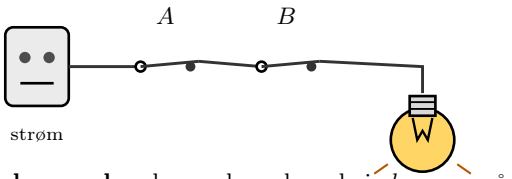
Resten av kapitlet handler om hva du kan bygge når det er alt du har å gå på.

5.2.2 To brytere og en lampe

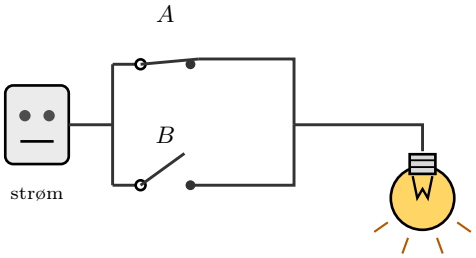
Glem maskinen et øyeblikk. Vi tar en stikkontakt, en lyspære og to brytere.

En bryter slipper enten strøm gjennom eller ikke. En lyspære lyser når det går strøm gjennom den. Mer er det ikke.

Nå kan vi koble de to bryterne på to ulike måter, og det er verdt å se hvor stor forskjell det gjør.



etter hverandre: lampa lyser bare hvis *begge* er på



ved siden av hverandre: lampa lyser hvis *minst én* er på

Figur 5.1 De to måtene å koble to brytere på. Øverst må strømmen gjennom begge for å nå fram til pæra. Nederst har den to veier å velge mellom, og her er A på mens B er av, noe som likevel holder. Returledningen tilbake til kontakten er utelatt i begge tegningene; den finnes, men har ingenting med saken å gjøre.

Øverst står bryterne **etter hverandre**. Strømmen må gjennom den første for å komme til den andre, så slår du av én av dem, er veien brutt og pæra mørk. Begge må være på.

Nederst står de **ved siden av hverandre**, og strømmen har to veier å velge mellom. Er bare én av dem åpen, kommer strømmen likevel fram. Det holder at minst én er på.

Les nå de to beskrivelsene som en programmerer:

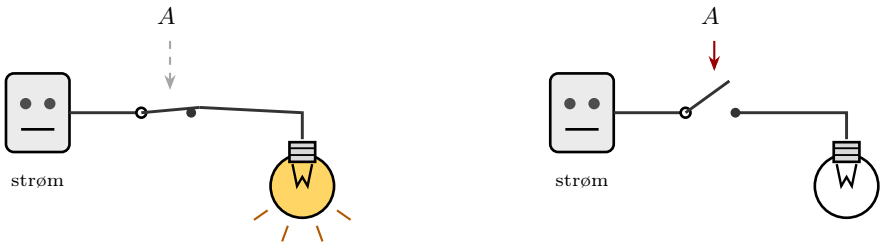
- *Begge* må være på. Det er **AND**.
- *Minst én* må være på. Det er **OR**.

Du har altså bygget to av de tre ordene fra kodelinja, av en stikkontakt, en lyspære og to brytere. Det er ingen sammenligning eller forenkling. Det er det samme.

5.2.3 Og den som gjør det motsatte

Det tredje ordet var NOT, og for å lage det må vi bytte ut en av delene.

Bryterne vi har brukt til nå er åpne når ingen rører dem. Det finnes også brytere som er lukket i hvilestilling, og som bryter strømmen når du trykker dem inn. Setter du en slik bryter i kretsen, lyser lampa helt til noen trykker.



Figur 5.2 En bryter som er lukket i hvilestilling. Den gjør det motsatte av de andre: trykker du den inn, bryter du strømmen. Pila viser hvor du trykker.

Du har sett dette i praksis. Lyset inne i kjøleskapet lyser så lenge døra er åpen, fordi det er døra som trykker inn bryteren når den går igjen.

Nå har vi alle tre: AND, OR og NOT, laget av ledninger og to slags brytere.

5.2.4 En som ikke lar seg lage slik

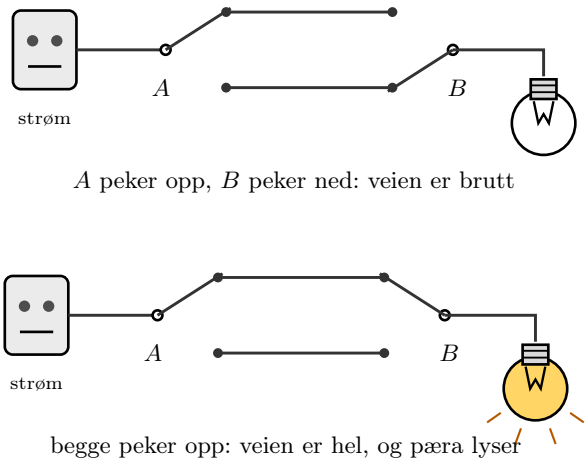
Men det finnes en kobling du har brukt mange ganger, som ikke passer inn i noen av de to mønstrene.

I trappeoppgangen henger det to brytere. Du kommer inn nederst, slår på lyset og går opp. Oppe i andre etasje slår du det av med den andre bryteren. Neste gang kommer kanskje noen ovenfra og gjør det motsatte, og det virker like bra.

Tenk litt over hva som er rart her. Ingen av de to bryterne har noen egen på-stilling eller av-stilling. Hver av dem har bare to stillinger, opp og ned, og ingen av dem betyr «lys» i seg selv.

Prøv å forklare det med det vi har hittil. Er det «begge må stå opp»? Nei, lyset kan være på med én oppe og én nede. Er det «minst én må stå opp»? Heller ikke, for med begge oppe kan lyset være av.

Det som avgjør, er om de to står i *samme* stilling eller ikke. Årsaken er lett å se når kretsen er tegnet.



Figur 5.3 Trappekoblingen. Mellom de to bryterne går det *to* ledninger, ikke én, og hver bryter velger hvilken av dem den skal bruke. Snu én av bryterne i den nederste tegningen, så er du tilbake i den øverste.

Det går to ledninger mellom bryterne, og hver bryter velger hvilken av dem den bruker. Peker begge på samme ledning, er veien hel. Peker de på hver sin, er den brutt.

Snu nå én av dem. Sto de likt, står de ulikt. Sto de ulikt, står de likt. Uansett hvilken vei det går, skifter lyset. Derfor virker koblingen uansett hvilken bryter du tar, og uansett hvem som var der før deg.

Note-to-myself

Er dette «lyser når de står likt» eller «lyser når de står ulikt»?
Det kommer an på hva du bestemmer deg for å kalle hva. Sier du at «opp» er 1 på begge bryterne, lyser lampa når de er like. Sier du at «opp» er 1 på den ene og «ned» er 1 på den andre, lyser den når de er ulike. Det er samme kobling og samme ledninger, bare en annen avtale om hva tallene betyr.
Det er akkurat poenget fra kapittel 2: bitene betyr det vi blir enige om at de skal bety. Det som er sant uansett avtale, er at ett bytte alltid snur lyset.

Uansett hvilken av lesemåtene du velger, er dette en tredje type oppførsel. Den lar seg ikke lage med brytere etter hverandre, og heller ikke med brytere ved siden av hverandre.

Det finnes altså flere oppførsler enn de to første. Hvor mange er det til sammen? Vi kommer tilbake til det, og svaret er mindre enn du tror.

5.2.5 Porten: et navn på oppførselen

Vi kunne fortsatt å tegne brytere gjennom hele kapitlet, men det ville vært en dårlig idé av to grunner.

Den første er at det blir for mange. En prosessor har milliarder av slike småting, og du ville aldri blitt ferdig med tegningen.

Den andre er at bryterne inni en brikke ikke ser slik ut. Det sitter faktisk brytere der, milliarder av dem, men de har verken vippearm eller bevegelige deler. De styres av spenning i stedet for av en finger, og de heter transistorer. Hvordan de er laget, kommer i kapittel 6.

Modellen du nettopp bygde er altså ikke feil. Den er bare tegnet med en annen type bryter.

Så vi gjør som folk alltid gjør når noe blir for detaljert: vi tegner et symbol i stedet, ett for hver oppførsel. Symbolet kaller vi en **port**.

Navnet passer best på den første av dem, der bildet stemmer: en port som enten slipper noe gjennom eller stopper det. For de andre har navnet bare blitt hengende ved, så du skal ikke lese noe særlig inn i det.

Du kan tenke på en port som en dørvakt med én fast regel. Vakta ved AND-døra slipper bare inn folk som både har fylt 18 *og* har billett. Vakta ved OR-døra nøyer seg med ett av de to. Vaktene bruker aldri skjønn, og de glemmer aldri regelen sin.

Alle porter har tre egenskaper til felles:

- Den har noen få innganger, som regel to. Hver inngang er 0 eller 1.
- Den har *én* utgang, som også er 0 eller 1.
- Den har en fast regel for hva utgangen blir, og regelen endrer seg aldri.

Mer trenger du ikke for å bygge. Vi bruker sju porttyper i denne boka, og fire av dem har du allerede møtt som brytere og lamper.

Note-to-myself

På engelsk heter en port *gate*, og det er det ordet du finner i all faglitteratur og i alle kretsprogrammer: «AND gate», «NOR gate».
Vi bruker det norske ordet *port* om selve tingen, og de engelske forkortelsene AND, OR, NOT og så videre om de enkelte typene. Det er slik de står på lysarkene, og slik de står på brikkene.

5.2.6 Funksjon: ordet som betyr tre ting

Nå trenger vi et ord til, og det er det mest forvirrende i kapitlet. Det er ikke fordi det betyr noe komplisert, men fordi du har sett det brukt på to andre måter før.

Vi trenger et ord for *selve oppførselen*, for «lampa lyser bare når begge er på», uavhengig av om den er laget av brytere, en port eller noe helt annet.

Ordet er **funksjon**.

Du så en funksjon allerede i kodelinja på første side: `slipp_inn()` er en Python-funksjon. I Python gir du en funksjon noen verdier og får en verdi tilbake. Fra matematikken kjenner du kanskje $f(x) = 2x+1$, som ser annerledes ut, men fungerer likt.

Den logiske funksjonen ligger nærmest den matematiske. Den er en regel, ikke en handling. Setter du inn verdier, følger svaret av seg selv, ikke fordi noe kjører, men fordi det bare er slik.

Dette er verdt å tenke over. En Python-funksjon *gjør* noe: den starter, bruker tid og blir ferdig. En port gjør ingenting. Den står der med noen signaler på inngangene, og utgangen er det den er. Det finnes ingen startknapp.

Ellers er likheten med Python god nok til å støtte seg på: noe går inn, og én verdi kommer ut. Det er to forskjeller:

- Alt som går inn og ut er bare 0 eller 1. Ingen tekst, ingen desimaltall, ingen lister.
- Det finnes så få muligheter at du kan skrive opp *alle*.

Det siste gjør dette stoffet uvanlig håndfast. En Python-funksjon som tar et tall kan du aldri teste for alle tall, for det er uendelig mange. En logisk funksjon med to innganger har fire tilfeller. Fire. Du kan skrive dem alle opp på en serviett, og da er du ferdig for alltid, med full sikkerhet.

Lista har et navn, *sannhetsverditabell*, og vi kommer til den i seksjon 5.4. Den er det viktigste verktøyet i hele kapitlet.

Note-to-myself

Samme ord, tre betydninger. De er i slekt, men ikke like:

- **I Python:** en kodeblokk med `def` foran.
- **I matematikken:** en regel som gir ett svar for hver verdi du setter inn.
- **Her:** oppførselen til en krets, altså hva som kommer ut for hver eneste kombinasjon av det som går inn.

I denne boka betyr «funksjon» alltid den siste. Mener vi en av de andre, sier vi det.

5.2.7 Krets: veien, ikke målet

Det tredje ordet trenger minst forklaring. En **krets** er porter koblet sammen med ledninger, altså den fysiske delen.

Og her er hovedpoenget i kapitlet, i én setning: **den samme funksjonen kan bygges av mange ulike kretser.**

Du har sett det allerede. De to bryterne i serie og en AND-port er helt forskjellige ting. Den ene er av metall og plast, den andre er et symbol på en brikke. Men de oppfører seg likt i alle fire tilfellene, så funksjonen er den samme.

Tenk på funksjonen som stedet du skal, og kretsen som veien dit. Skal du fra Blindern til sentrum, kan du ta T-banen, bussen eller sykkelen. Du kommer fram uansett, men turene er ikke like lange eller like dyre.

Tankevekker

Dette er den samme forskjellen som i kapittel 4, der skillet gikk mellom hva en maskin kan gjøre og hvordan den gjør det.
Nå dukker den opp igjen, flere lag lenger ned. Funksjonen er avtalen om hva som skal skje, og kretsen er måten det bygges på. Du står fritt til å bygge billigere så lenge avtalen holdes.
Skillet kommer tilbake på hvert eneste nivå i faget. Det er ikke tilfeldig; slik er hele stabelen satt sammen.

5.2.8 Derfor er det verdt å spare porter

Hvis flere veier fører fram, hvorfor bry seg om hvilken du velger?

Fordi hver port koster noe. Det er lett å glemme når kretsen er tegnet på papir og har tre symboler. Det er verre å glemme når den skal trykkes i milliarder av eksemplarer på en brikke.

- **Plass.** Porten tar areal, og areal på en brikke er det dyreste du har.
- **Strøm.** Hver port bruker strøm, også når den ikke gjør noe spesielt.
- **Varme.** Strømmen blir til varme, og varmen må ut et sted.
- **Tid.** Et signal bruker tid på å gå gjennom en port. Flere porter etter hverandre gir lengre vei, og lengre vei betyr at klokka må gå saktere.

Den siste kobler dette kapitlet til det forrige. Kapittel 4 sa at klokka er stilt etter det tregeste steget i maskinen. Nå skal vi se hva det tregeste steget faktisk består av.

Forenkling er derfor ikke pynt, og heller ingen skoleøvelse. Å fjerne en port gjør brikken litt mindre, litt kjøligere og litt raskere, alt på én gang.

5.3 Portene

Nå møter du hele samlingen: sju porter, fire grunnleggende og tre som er de samme med en boble på utgangen.

Fire av dem kjenner du allerede fra forrige seksjon, du har bare ikke sett symbolene. De kommer først.

For hver port får du tre ting: symbolet, tabellen over hva den gjør i hvert enkelt tilfelle, og en kort skrivemåte.

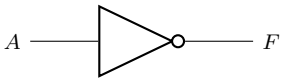
Skrivemåten trenger en forklaring. Å skrive «lampa lyser bare hvis begge er på» hver gang blir tungvint, så vi forkorter. Inngangene kaller vi A og B, og utgangen F. OG skrives som et gangetegn, eller ved at bokstavene settes inntil hverandre: $A \cdot B$ eller AB. ELLER skrives som pluss: $A + B$. IKKE skrives som en strek over: $\overline{A}$.

Det er hele skrivemåten. Tre tegn, og du har sett dem alle nå.

5.3.1 IKKE-porten

NOT-porten, også kalt **inverteren**, er kjøleskapslyset fra forrige seksjon. Den er den eneste porten med bare én inngang, og den snur signalet: inn 0 gir ut 1, og inn 1 gir ut 0.

$$F = \overline{A}$$



A	F
0	1
1	0

Figur 5.4 NOT-porten. Trekanten er selve porten, og den lille sirkelen i spissen er inverteringen.

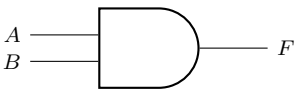
Note-to-myself

Du kommer til å se en annen skrivemåte enn vår. På lysarkene og på portarket skrives «ikke A» med **apostrof**: A'. Det betyr det samme som $\overline{A}$, og forskjellen er bare typografi.
Overstreken har en fordel. Sammenlign $\overline{A}\,\overline{B}$ med $\overline{AB}$. Den første betyr «ikke A, og ikke B». Den andre betyr «ikke (A og B)». Det er to helt forskjellige ting, og streken viser forskjellen med én gang. Med apostrof blir det $A'B'$ mot $(AB)'$, og da henger alt på en parentes som er lett å overse. Hvorfor de to er forskjellige, kommer vi tilbake til. Det er en av fellene i dette stoffet.

5.3.2 OG-porten

AND-porten er de to bryterne etter hverandre. Den gir 1 bare når begge inngangene er 1, og 0 i alle de tre andre tilfellene.

$$F = A \cdot B$$



A	B	F
0	0	0
0	1	0
1	0	0
1	1	1

Figur 5.5 AND-porten. Flat venstreside og rund høyreside, som en D.

En mikrobølgeovn starter bare når døra er lukket *og* noen har trykket på start. Begge deler må være oppfylt. Det er en AND, og det er grunnen til at ovnen stopper i det du åpner døra.

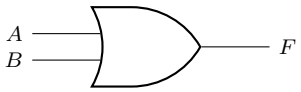
Fun Fact

AND er den eneste av de sju portene som er vanlig multiplikasjon. Se i tabellen: $0 \cdot 0 = 0$, $0 \cdot 1 = 0$, $1 \cdot 0 = 0$ og $1 \cdot 1 = 1$. Det stemmer for hver eneste rad.
Gangetegnet er derfor ikke noe vi har lånt for å få det til å passe, slik plusstegnet er for ELLER. For AND stemmer det på ordentlig.

5.3.3 ELLER-porten

OR-porten er de to bryterne ved siden av hverandre. Den gir 1 når minst én av inngangene er 1, og bare 0 når begge er 0.

$$F = A + B$$



A	B	F
0	0	0
0	1	1
1	0	1
1	1	1

Figur 5.6 OR-porten. Buet venstreside, og den møtes i en spiss til høyre.

Lyset i en fellesgang slås på når bevegelsessensoren merker noe *eller* noen trykker på bryteren. Én av delene er nok.

Her er det en liten språkfelle. På norsk sier vi gjerne «enten kaffe eller te» og mener én av dem, ikke begge. Det boolske ELLER er mer raust: det er sant også når begge er sanne. Den strenge varianten, der nøyaktig én skal være sann, er neste port.

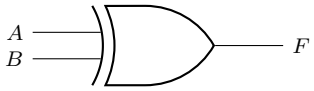
Advarsel :)

Plusstegnet i $A + B$ er ikke addisjon.
Det er lånt fra matematikken fordi ELLER oppfører seg som pluss i tre av fire tilfeller: $0 + 0 = 0$, $0 + 1 = 1$ og $1 + 0 = 1$. I det fjerde går det galt. Her er $1 + 1 = 1$, fordi «minst én av dem er sann» fortsatt bare er sant. Svaret er verken 2 eller 0 med mente.
Si «A eller B» når du leser det høyt, ikke «A pluss B». Det koster ingenting og sparer deg for den vanligste feilen i dette stoffet.

5.3.4 ENTEN-ELLER-porten

XOR-porten, fra engelsk *exclusive or*, gir 1 når inngangene er *forskjellige*. Står de likt, gir den 0.

$$F = A \oplus B$$



A	B	F
0	0	0
0	1	1
1	0	1
1	1	0

Figur 5.7 XOR-porten. Som OR, men med en ekstra bue utenfor venstresiden.

Dette er trappelyset. Koblingen som ikke lot seg lage av brytere etter hverandre eller ved siden av hverandre, har nå fått sin egen port.

Husker du notatet om at det kom an på hva vi kalte hva? Her får du oppgjøret. Velger du avtalen der lampa lyser når bryterne står *ulikt*, er trappekoblingen en XOR. Velger du den andre avtalen, er den en XNOR, som vi møter om et par sider. De to er hverandres motsetninger, mens koblingen i veggen er den samme uansett.

Egenskapen vi fant gjelder for begge: endrer du én av de to verdiene, snur svaret. Derfor virker bryteren i oppgangen uansett hvem som var der før deg.

Elektrikere kaller koblingen en toveisbryter og tenker neppe på boolsk algebra mens de monterer den. Funksjonen er like fullt den samme.

XOR er porten du kommer til å se oftest resten av kapitlet. Den dukker opp fire ganger: i halvadderen, i fulladderen, i kretsen som oppdager overflyt og i subtraksjonskretsen.

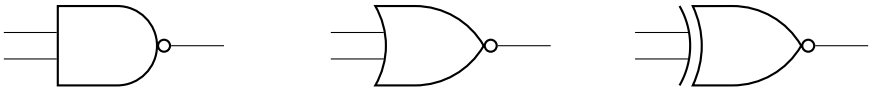
Grunnen er at XOR er den eneste porten som svarer på «er disse to forskjellige?». Merkelig nok er det også spørsmålet «hva blir summen?» når du legger sammen to bit. Vi kommer tilbake til det i seksjon 5.7.

5.3.5 De tre med boble

De tre siste portene er ikke nye funksjoner du må lære utenat. De er de tre du nettopp har sett, med utgangen snudd.

- **NAND** er AND med invertert utgang: $F = \overline{A \cdot B}$.
- **NOR** er OR med invertert utgang: $F = \overline{A + B}$.
- **XNOR** er XOR med invertert utgang: $F = \overline{A \oplus B}$.

Symbolene følger samme logikk. Du tegner grunnporten og setter en liten sirkel på utgangen.



Figur 5.8 De tre inverterte portene. Sirkelen på utgangen er hele forskjellen fra AND, OR og XOR.

Sirkelen betyr invertering, uansett hvor den står. Ser du den i en kretstegning, vet du at signalet snus akkurat der. Det er den eneste huskeregelen du trenger for de tre siste portene, så du har egentlig bare fire porttyper å holde styr på, ikke sju.

XNOR fortjener en ekstra setning. Den er det motsatte av XOR og svarer dermed på «er disse to *like*?». En krets som sammenligner to bit og sier fra om de er identiske, er altså bare én port. Den dukker opp igjen i seksjon 5.6.

5.3.6 Oversikten

Tabell 5.1 viser hele byggesettet samlet.

Port	Uttrykk	00	01	10	11
AND	$F = A \cdot B$	0	0	0	1
OR	$F = A + B$	0	1	1	1
XOR	$F = A \oplus B$	0	1	1	0
NAND	$F = \overline{A \cdot B}$	1	1	1	0
NOR	$F = \overline{A + B}$	1	0	0	0
XNOR	$F = \overline{A \oplus B}$	1	0	0	1

Tabell 5.1 De seks toinngangsportene. Kolonnene er de fire kombinasjonene av A og B. NOT står utenfor, siden den bare har én inngang.

Sammenlign radene to og to: AND mot NAND, OR mot NOR, XOR mot XNOR. I hvert par er alle tallene snudd. Det er den samme observasjonen som boblen i symbolet viser.

Fun Fact

Tidligere spurte vi hvor mange oppførsler det finnes i det hele tatt. Nå kan vi svare, og det krever bare litt telling.
En funksjon med to innganger har fire tilfeller. For hvert av dem kan utgangen settes til 0 eller 1, helt fritt. Det gir $2 \cdot 2 \cdot 2 \cdot 2 = 16$ mulige oppførsler, og flere finnes ikke.
Seks av de seksten har fått navn og symbol som toinngangsporter, og de står i tabellen over. De ti andre er ikke uinteressante, bare mindre brukt. Fire av dem ser bort fra den ene inngangen, og da holder det med en NOT-port eller en vanlig ledning. To gir samme svar uansett hva du mater inn. De fire siste har navn i logikken, men står sjelden som egne porter på en brikke.
Alle toinngangsporter som finnes er altså seksten stykker, og det er ingenting mer der ute.

Note-to-myself

Denne tabellen skal du ikke pugge.
Du skal kunne *bruke* den: gå fra en krets til en tabell, fra en tabell til et uttrykk, og se at to kretser gjør det samme. Selve oversikten er et oppslagsverk, og det er lov å ha den foran seg.
Det du bør ha i hodet uten å slå opp, er de tre grunnportene, og at en boble snur utgangen.

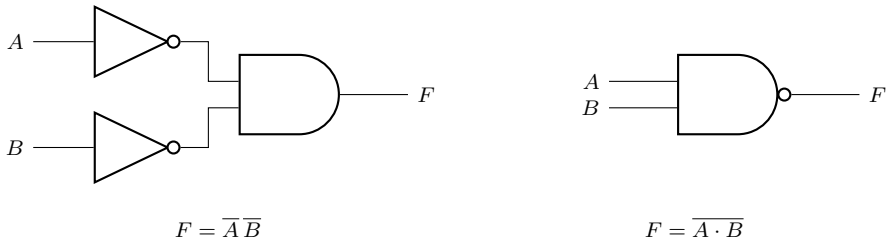
5.3.7 To feller

To feil går igjen år etter år. Begge er lette å unngå når du vet om dem.

Den første er plusstegnet, som vi allerede har nevnt: $1 + 1 = 1$.

Den andre er viktigere, og den koster flest poeng. Det er forskjell på å invertere inngangene og å invertere utgangen.

Se på disse to kretsene.



Figur 5.9 To kretser som ser beslektet ut og gjør helt forskjellige ting. Til venstre snus signalene før porten, til høyre snus svaret etter.

Til venstre snus A og B hver for seg, og så spør vi om begge de snudde er 1. Til høyre spør vi først om begge de opprinnelige er 1, og snur deretter svaret.

Regn gjennom ett tilfelle: $A = 1$, $B = 0$.

Venstre krets: $\overline{A} = 0$ og $\overline{B} = 1$. OG mellom 0 og 1 er 0, så utgangen er **0**.

Høyre krets: $A \cdot B = 0$, og snudd blir det **1**.

Samme innganger, motsatt svar.

Advarsel :)

Ser du en krets med invertere i, er det første du bør spørre om: *hvor* står de?
Står de foran porten, hører de til hver sin inngang, og du skriver $\overline{A}$ og $\overline{B}$ hver for seg. Står boblen på utgangen, hører inverteringen til hele uttrykket, og overstreken går over alt sammen.
Dette er den feilen flest gjør i dette stoffet. Den oppstår fordi øyet ser «det er en invertering her et sted» og hopper rett til et svar.

5.4 Sannhetsverditabellen

En **sannhetsverditabell** lister opp alle mulige kombinasjoner av innganger og viser hva utgangen blir for hver av dem.

Den beskriver alt. Når tabellen er fylt ut, finnes det ingen situasjon kretsen kan havne i som ikke står der. Det er noe av det fineste ved digital logikk: antall tilfeller er endelig, så du kan sjekke dem alle.

5.4.1 Hvor mange rader

Hver inngang kan være 0 eller 1. Det er to valg per inngang, og valgene er uavhengige av hverandre.

Med n innganger blir det 2^n rader.

Antall innganger	Antall rader
1	2
2	4
3	8
4	16
10	1 024

Tabell 5.2 Tabellen dobles for hver inngang du legger til.

Dette er den samme doblingen som i kapittel 2: ett bit til gir dobbelt så mange mønstre. Konsekvensen her er at tabeller fort blir uhåndterlige. Ti innganger gir over tusen rader, og ingen setter opp det for hånd.

I dette kurset holder vi oss til høyst tre innganger. Åtte rader er nok til å vise alt som er verdt å vise, og få nok til at du kan sjekke hver eneste en.

5.4.2 Slik glemmer du ingen rader

Det vanligste problemet med tabeller er ikke at man regner feil. Det er at man glemmer en rad eller skriver den samme kombinasjonen to ganger.

Det finnes et enkelt triks som fjerner problemet: **skriv radene som binære tall, i tellerekkefølge.**

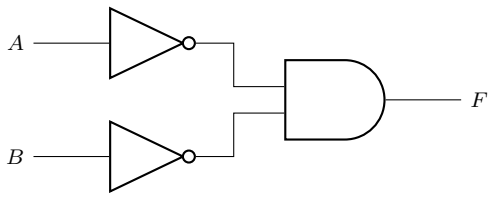
Med to innganger teller du 0, 1, 2, 3, altså 00, 01, 10, 11. Med tre innganger teller du fra 0 til 7: 000, 001, 010, 011, 100, 101, 110, 111.

Da får du alle kombinasjonene, hver av dem én gang, og i en rekkefølge alle bruker. To personer som setter opp samme tabell, får radene i samme orden og kan sammenligne direkte.

Se også på kolonnene når du teller slik. Den til høyre bytter verdi for hver rad, den ved siden av for annenhver rad, den neste for hver fjerde. Det er bare slik telling i totallssystemet ser ut, men det er praktisk, for du kan fylle ut inngangskolonnene nedover uten å tenke og bruke hodet på utgangen.

5.4.3 Å lese en krets forlengs

Her er en krets med tre porter og to innganger.



Figur 5.10 Tre porter. To invertere og en AND.

Å lese **framlengs** betyr at du starter til venstre og følger signalene mot høyre. Underveis gir du hver ledning et navn.

1. Inn kommer A og B.

2. A går gjennom en inverter, og ut kommer $\overline{A}$.
3. B går gjennom en inverter, og ut kommer $\overline{B}$.
4. De to møtes i en AND-port, som gir OG mellom dem.
5. Utgangen er $F = \overline{A} \cdot \overline{B}$.

Metoden er mekanisk, og det er poenget. Du trenger ikke forstå hva kretsen gjør for å skrive opp uttrykket. Du følger ledningene og skriver ned hva hver port gjør med det den får inn.

5.4.4 Og baklengs

Å lese **baklengs** betyr at du starter ved utgangen og spør: hva er den siste porten?

Her er den siste porten en AND, så F er et produkt av to ting. Da spør du hva de to tingene er, og går ett hakk til venstre for hver av dem. Begge viser seg å være en invertert inngang.

Du kommer fram til samme svar. Forskjellen er hva metodene passer til.

- **Framlengs** passer når du vil finne ut hva en krets faktisk gjør, og når du leter etter feil. Du følger signalet slik det går.
- **Baklengs** passer når du har et uttrykk og skal tegne kretsen, eller når du leter etter porter som ikke trengs. Du begynner med målet.

Begge er verdt å kunne, og de tar fem minutter å lære.

5.4.5 Tre porter som var én

Sett opp tabellen for kretsen i figur 5.10 på forrige side. Regn rad for rad: først hva inverterne gir, og så hva AND-porten gjør med det.

A	**B**	$\overline{A}$	$\overline{B}$	**F**
0	0	1	1	1
0	1	1	0	0
1	0	0	1	0
1	1	0	0	0

Tabell 5.3 Kretsen regnet ut rad for rad. Mellomkolonnene er inverterne.

Utgangen er 1 i én eneste rad, den der begge inngangene er 0.

Slå opp i tabell 5.1 på side 116 og let etter mønsteret $1, 0, 0, 0$.

Det er NOR.

Tankevekker

Tre porter gjør akkurat det samme som én.
$\overline{A} \cdot \overline{B}$ og $\overline{A + B}$ er to uttrykk for samme funksjon. Den ene kretsen bruker to invertere og en AND,

altså tre porter og mer plass. Den andre bruker én NOR-port.
Dette er hele kapitlet i miniatyr. Funksjonen er den samme, men kretsen er ikke det, og den ene er tre ganger så dyr som den andre.

Du har nettopp brukt en av de mest kjente reglene i boolsk algebra uten å vite hva den heter. Den får navn i seksjon 5.6.

Tenk om...

... du skulle finne ut dette ved å prøve deg fram på en ferdig brikke, uten tabellen. Du måtte måle utgangen for hver kombinasjon av innganger, skrive ned resultatene og sammenligne med noe du kjente fra før.
Det er akkurat det tabellen gjør, bare på papir og på et halvt minutt. Derfor er sannhetsverditabellen det første du tyr til når en krets er ukjent: den gjør gjetting om til oppslag.

5.5 Fra tabell til uttrykk

Hittil har vi gått fra krets til tabell. Nå går vi motsatt vei, fra tabell til uttrykk.

Dette er den viktigste metoden i kapitlet. Den er oppskriften som gjør et ønske om hva en krets *skal* gjøre, om til noe du kan bygge. Vi bruker den også til å utlede adderkretsen litt senere.

5.5.1 Vi leter etter ettallene

Regelen er enkel: **se bare på radene der utgangen er 1.**

Grunnen er praktisk. Når vi beskriver noe digitalt, beskriver vi som oftest når noe skal *skje*. Når skal alarmen gå? Når skal motoren starte? Når skal mente sendes videre? Det er lettere å liste opp de få tilfellene der svaret er ja enn alle de andre.

Tenk på en lampe. Du forklarer den ved å si «den lyser når bryteren står på». Du ramser ikke opp alle situasjonene der den ikke lyser.

5.5.2 Ett ledd per rad

For hver rad der $F = 1$ skriver du ett ledd. Leddet er en AND mellom alle inngangene, og hver inngang skrives slik at den blir 1 akkurat i den raden.

- Er inngangen 1 i raden, skriver du bokstaven som den er.
- Er inngangen 0 i raden, skriver du bokstaven med overstrek.

Hvorfor blir en null til en overstrek? Fordi leddet skal være sant akkurat i den raden. Når $A = 0$, er $\overline{A} = 1$. Det er altså $\overline{A}$ som er «på» i den raden, ikke A.

Til slutt kobler du alle leddene med ELLER. Logikken er at F er 1 hvis den første situasjonen inntreffer, *eller* den andre, *eller* den tredje.

Vi prøver det på en tabell.

A	B	F	Ledd
0	0	1	$\overline{A}\,\overline{B}$
0	1	1	$\overline{A}B$
1	0	0	—
1	1	1	AB

Tabell 5.4 Tre rader har 1 på utgangen, og hver av dem gir ett ledd.

$$F = \overline{A}\,\overline{B} + \overline{A}B + AB$$

Sjekk at det stemmer ved å prøve en rad. Sett $A = 0$ og $B = 1$. Første ledd: $1 \cdot 0 = 0$. Andre ledd: $1 \cdot 1 = 1$. Tredje ledd: $0 \cdot 1 = 0$. ELLER mellom 0, 1 og 0 er 1, som er det tabellen sier.

Legg merke til at det i hver rad der F er 1, er nøyaktig ett ledd som er 1. Slik må det være: hvert ledd er skreddersydd for sin ene rad og er 0 i alle de andre. I radene der F er 0, er alle leddene 0.

Note-to-myself

Formen du nettopp skrev har et navn: **sum av produkter**, eller SOP etter engelsk *sum of products*. «Produktene» er AND-leddene, og «summen» er ELLER-en som binder dem sammen. Navnet er greit å kjenne, for det dukker opp overalt i dette fagfeltet. Men metoden er viktigere enn navnet.

5.5.3 Tre innganger

Oppskriften endrer seg ikke når det kommer en inngang til. Hvert ledd får bare tre bokstaver i stedet for to.

A	B	C	F	Ledd
0	0	0	1	$\overline{A}\,\overline{B}\,\overline{C}$
0	0	1	0	—
0	1	0	1	$\overline{A}B\overline{C}$
0	1	1	0	—
1	0	0	1	$A\overline{B}\,\overline{C}$
1	0	1	0	—
1	1	0	0	—
1	1	1	1	ABC

Tabell 5.5 Åtte rader, fire av dem med 1 på utgangen.

$$F = \overline{A}\,\overline{B}\,\overline{C} + \overline{A}B\overline{C} + A\overline{B}\,\overline{C} + ABC$$

Uttrykket er stygt. Det har fire ledd og tolv bokstaver, og hvis du bygger det slik det står, trenger du tre invertere, fire AND-porter med tre innganger hver og en OR-port med fire innganger.

Det er åtte porter for noe som, skal det vise seg, bare trenger to. Men vi må ha uttrykket først, før vi kan gjøre noe med det.

5.5.4 Hva om vi hadde tatt nullene?

Et rimelig spørsmål er hvorfor det skal være ettallene. Kunne vi like gjerne beskrevet når utgangen er 0?

Ja, det går helt fint. Formen har et eget navn, *produkt av summer*. Der skriver du ett ELLER-ledd for hver nullrad og binder dem sammen med OG. Resultatet ser annerledes ut, men beskriver samme funksjon.

Vi holder oss til ettallene i denne boka, av to grunner. Den ene er at det passer måten vi tenker på: vi beskriver når noe skal aktiveres. Den andre er at det i praksis ofte er færre ettall enn nuller, og da blir uttrykket kortere.

Det er likevel greit å vite at valget finnes. Møter du en tabell med sytten ettall og tre nuller, er det nullene du vil beskrive.

5.6 Forenkling

Vi har et uttrykk, og nå skal vi gjøre det mindre.

Dette er delen av kapitlet som ligner mest på vanlig matematikk, og den gode nyheten er at reglene stort sett er de du allerede kan. Du setter utenfor parentes, stryker ledd som ikke bidrar og leter etter mønstre.

5.6.1 Reglene

Sju regler tar deg langt. De tre første er nesten for opplagte til å skrive ned, og nettopp derfor glemmer man dem.

Navn	**Regel**	**I ord**
Nøytralt element	$X + 0 = X \quad X \cdot 1 = X$	Legger ingenting til
Dominans	$X + 1 = 1 \quad X \cdot 0 = 0$	Avgjør alene
Idempotens	$X + X = X \quad X \cdot X = X$	Å si det to ganger hjelper ikke
Komplement	$X + \overline{X} = 1 \quad X \cdot \overline{X} = 0$	Én må være sann, aldri begge
Felles faktor	$XY + XZ = X(Y + Z)$	Sett utenfor parentes
Absorpsjon	$X + XY = X$	Det lange leddet er overflødig
Absorpsjon, variant	$\overline{X} + XY = \overline{X} + Y$	X i andre ledd kan strykes

Tabell 5.6 Regelsettet. De to siste er de som faktisk sparer porter.

De fire første kan du kontrollere ved å sette inn 0 og 1 for X og regne etter. Felles faktor er den

distributive loven, akkurat som i vanlig algebra.

De to absorpsjonsreglene er de som gjør det virkelige arbeidet, så det er verdt å overbevise seg om at de stemmer.

$X + XY = X$**.** Hvis $X = 1$, er hele uttrykket 1, uansett hva Y er. Hvis $X = 0$, faller første ledd bort, og andre ledd blir $0 \cdot Y = 0$. Svaret er altså alltid lik X, og leddet XY endrer aldri noe.

$\overline{X} + XY = \overline{X} + Y$**.** Hvis $X = 0$, er første ledd 1, og begge sider er 1. Hvis $X = 1$, faller første ledd bort på begge sider, og vi sitter igjen med Y mot Y. De to sidene er altså like.

Den siste er den som oftest blir oversett, og den som sparer mest.

5.6.2 Første eksempel, helt ut

Vi tar uttrykket fra tabell 5.4 på side 121. Underveis er det **fargede** leddet alltid det vi jobber med akkurat da.

$$F = \overline{A}\,\overline{B} + \overline{A}B + AB$$

Steg 1. De to første leddene har $\overline{A}$ felles:

$$F = \overline{A}\,\overline{B} + \overline{A}B \;+\; AB$$

Sett $\overline{A}$ utenfor parentes:

$$F = \overline{A}\,(\overline{B} + B) \;+\; AB$$

Steg 2. Inni parentesen står $\overline{B} + B$. Den er 1 etter komplementregelen: enten er B sann, eller så er den ikke det, og en tredje mulighet finnes ikke.

$$F = \overline{A} \cdot 1 \;+\; AB$$

Og $\overline{A} \cdot 1$ er bare $\overline{A}$:

$$F = \overline{A} + AB$$

Steg 3. Her stopper mange, men se på formen på det som står igjen:

$$F = \overline{A} + AB$$

Det er akkurat $\overline{X} + XY$, med $X = A$ og $Y = B$. Absorpsjonsregelen sier at A-en inni produktet kan strykes:

$$F = \overline{A} + B$$

Vi startet med tre ledd og seks bokstaver og endte med to bokstaver. Kretsen er nå én inverter og én OR-port.

Sjekk mot tabellen: A=0, B=0 gir $1 + 0 = 1$. A=0, B=1 gir 1. A=1, B=0 gir $0 + 0 = 0$. A=1, B=1 gir $0 + 1 = 1$. Mønsteret er $1, 1, 0, 1$, og det er tabellen.

Fun Fact

$\overline{A} + B$ har en tolkning det er verdt å stoppe ved: det er påstanden «hvis A, så B».
Prøv selv. Utsagnet «hvis det regner, tar jeg med paraply» er bare galt i ett tilfelle: det regner, og du har ikke paraply. Det er $A = 1$ og $B = 0$, og det er akkurat den ene raden der uttrykket vårt gir 0.
Implikasjon fra logikken og en krets med to porter viser seg å være det samme.

5.6.3 Andre eksempel, og en overraskelse

Nå tar vi det stygge uttrykket fra tabell 5.5 på side 121. Vi bruker samme framgangsmåte, og det **fargede** er det vi jobber med.

$$F = \overline{A}\,\overline{B}\,\overline{C} + \overline{A}B\overline{C} + A\overline{B}\,\overline{C} + ABC$$

Steg 1. De tre første leddene har $\overline{C}$ felles:

$$F = \overline{A}\,\overline{B}\,\overline{C} + \overline{A}B\overline{C} + A\overline{B}\,\overline{C} \; + \; ABC$$

Sett $\overline{C}$ utenfor:

$$F = \overline{C}\,(\overline{A}\,\overline{B} + \overline{A}B + A\overline{B}) + ABC$$

Steg 2. Inni parentesen har de to første $\overline{A}$ felles:

$$F = \overline{C}\big(\overline{A}\,\overline{B} + \overline{A}B + A\overline{B}\big) + ABC$$

$$F = \overline{C}\big(\overline{A}(\overline{B} + B) + A\overline{B}\big) + ABC$$

Og $\overline{B} + B = 1$, så den indre parentesen faller bort:

$$F = \overline{C}(\overline{A} + A\overline{B}) + ABC$$

Steg 3. Her er absorpsjonsregelen igjen, nå med $X = A$ og $Y = \overline{B}$:

$$F = \overline{C}(\overline{A} + A\overline{B}) + ABC$$

$$F = \overline{C}(\overline{A} + \overline{B}) + ABC$$

Stopp her og se på det som står. Det er $\overline{C}$ ganget med «ikke A eller ikke B», pluss AB ganget med C.

Sett $D = A \cdot B$. Da er $\overline{A} + \overline{B}$ det samme som $\overline{D}$. Det er De Morgans lov, som vi tar i neste underseksjon. Uttrykket blir:

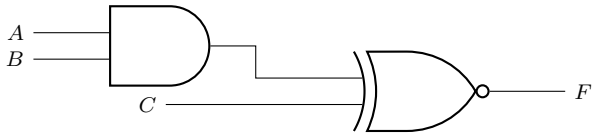
$$F = \overline{C}\,\overline{D} + DC$$

Det betyr «C og D er like», altså XNOR.

$$F = \overline{C \oplus (A \cdot B)}$$

Tankevekker

Fire ledd, tolv bokstaver og åtte porter ble til **to porter**: én AND som regner ut $A \cdot B$, og én XNOR som sammenligner resultatet med C.
Og nå kan du lese funksjonen i klartekst, noe som var umulig før. Kretsen svarer på spørsmålet: «er C lik OG-en av A og B?»
Det er dette forenkling er til for. Den sparer porter, men den gjør også at kretsen får en mening du kan si høyt.



Figur 5.11 Den samme funksjonen som på forrige side, bygget på nytt. To porter i stedet for åtte.

5.6.4 De Morgan

Vi har nå brukt den samme regelen to ganger uten å nevne navnet. Første gang var da $\overline{A}\,\overline{B}$ viste seg å være en NOR-port. Andre gang var nettopp nå, da $\overline{A} + \overline{B}$ ble til $\overline{AB}$.

Regelen heter **De Morgans lover**, etter en britisk matematiker fra 1800-tallet, og den finnes i to utgaver:

$$\overline{A \cdot B} = \overline{A} + \overline{B}$$
$$\overline{A + B} = \overline{A} \cdot \overline{B}$$

I ord: når du drar en overstrek fra utsiden og inn, snur du operatoren. OG blir ELLER, og ELLER blir OG. Hver variabel får sin egen overstrek.

Du kan sjekke den med dagligtale. «Det er ikke sant at både Nora og Ahmed er hjemme» betyr det samme som «enten er ikke Nora hjemme, eller så er ikke Ahmed det». Og «ingen av dem er hjemme» betyr «Nora er ikke hjemme, og Ahmed er ikke hjemme».

Note-to-myself

De Morgan er grunnen til at det finnes to måter å tegne de samme portene på, og at erfarne folk bytter mellom dem midt i en tegning.
En NAND-port kan tegnes som en AND med boble på utgangen, eller som en OR med bobler på begge inngangene. Funksjonen er den samme, og hvilken du velger avhenger av hva du vil at leseren skal se.
Vi bruker bare den første formen i denne boka. Men ser du en gang en OR-port med to bobler foran seg, vet du nå hva det er.

5.6.5 Hvor langt er langt nok

Et ærlig spørsmål er hvordan du vet at du er ferdig.

Det korte svaret er at du ikke vet det sikkert med metoden i dette kapitlet. Du leter etter felles faktorer og prøver absorpsjonsreglene, og når du ikke finner flere, stopper du. Det er fullt mulig å gå glipp av noe, og det skjer ofte.

Det finnes en systematisk metode som garanterer at du er i mål for små uttrykk. Den heter Karnaugh-diagram og står i appendiks B. Den er ikke pensum, men den er morsom, og den gjør det samme arbeidet på et rutenett i stedet for i en lang rekke omskrivninger.

Advarsel :)

Vær forsiktig med å si at et uttrykk ikke kan forenkles.
Prøv $F = AB + AC + D$. Ved første blikk ser det ikke ut til at noe gjentar seg, og det er fristende å si seg ferdig. Men A står i begge de to første leddene, og felles faktor gir $F = A(B + C) + D$. Det sparer en AND-port.
Gå gjennom listen i tabell 5.6 på side 122 én gang til før du konkluderer. Særlig felles faktor, den enkleste regelen, blir ofte oversett når uttrykket ser rotete ut.

5.7 Kretsen som legger sammen

Nå har vi verktøyet. Det som gjenstår er å bruke det på noe maskinen faktisk trenger.

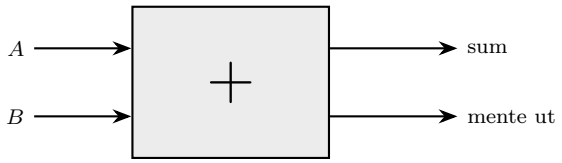
Kapittel 4 tegnet ALU-en som en bred V med tre innganger og to utganger, og sa rett ut at vi ikke åpnet den. Figur 4.5 på side 91 er den figuren. Her åpner vi den.

Vi begynner på det smaleste stedet: *hvordan legger man sammen to enkeltbit?* Når det er løst, er resten bare kopiering.

5.7.1 Den svarte boksen

Vi vet hva vi vil ha før vi vet hvordan vi skal få det.

Inn skal det gå to bit, A og B. Ut skal vi få svaret. Men svaret på $1 + 1$ får ikke plass i ett bit, så vi trenger to utganger: summen og en mente som sendes videre.



Figur 5.12 Det vi er ute etter. To bit inn, og et svar som ikke får plass i ett bit.

Binær addisjon på papir går kolonne for kolonne, akkurat som i titallssystemet: legg sammen sifrene i én kolonne, skriv ned det som får plass og send resten videre som mente. Det er akkurat det du gjør når du regner $9 + 4$: du skriver 3 og husker 1. Kapittel 2 regnet gjennom flere slike oppstillinger i toerkomplement. Det er den ene kolonnen vi skal bygge nå.

5.7.2 Tabellen, og det du kjenner igjen

Det er fire kombinasjoner, og du kan svarene allerede.

$0 + 0 = 0$. $0 + 1 = 1$. $1 + 0 = 1$. Og $1 + 1$ er 10 i totallssystemet: null i denne kolonnen og én i mente.

A	B	Sum	Mente ut
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

Tabell 5.7 Addisjon av to enkeltbit. To utganger, og hver av dem er sin egen funksjon.

Her er grepet: **behandle de to utgangskolonnene som to separate funksjoner.** Vi har to tabeller ved siden av hverandre, og vi løser dem hver for seg.

Les sum-kolonnen nedover: $0, 1, 1, 0$. Slå opp i tabell 5.1 på side 116.

Det er XOR.

Les mente-kolonnen: $0, 0, 0, 1$. Det er AND.

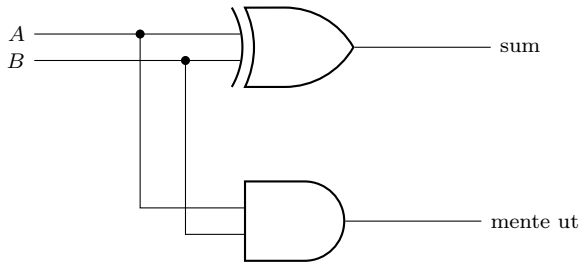
$$\text{Sum} = A \oplus B \qquad \text{Mente ut} = A \cdot B$$

Tankevekker

Dette er ingen tilfeldighet, og det fortjener et avsnitt.
Summen av to bit er 1 når nøyaktig én av dem er 1. «Nøyaktig én» er definisjonen på XOR. Det er trappelyset fra begynnelsen, brukt til å regne.
Og mente oppstår bare når begge er 1, som er definisjonen på AND. Begge må bidra, ellers blir det ingen overflyt til neste kolonne.
Maskinen legger sammen ved å stille to spørsmål om to bit: er de forskjellige, og er de begge 1? Det er hele addisjonen.

5.7.3 Halvadderen

To porter, og kretsen er ferdig.



Figur 5.13 Halvadderen. A og B går til begge portene. De fylte prikkene betyr at ledningen deler seg; der linjer krysser uten prikk, er det ingen forbindelse.

Dette er en **halvadder**.

Note-to-myself

Om prikkene: en fylt sirkel der to ledninger møtes betyr at de er koblet sammen og fører samme signal. Krysser to streker hverandre uten prikk, går de bare over hverandre uten å røre.
Dette er verdt å merke seg, for det er den eneste måten en flat tegning kan vise noe som i virkeligheten ligger i flere lag.

5.7.4 Det den mangler

Hvorfor kalles den «halv»?

Tenk på hvordan du legger sammen tall med flere sifre på papir. Kolonnen helt til høyre har to sifre å forholde seg til. Alle de andre har tre: de to sifrene, pluss menten fra kolonnen til høyre.

Halvadderen har bare to innganger, så den kan ikke ta imot mente. Den duger til den aller første kolonnen og ingen andre.

Vi trenger en versjon med tre innganger. Den heter **fulladder**.

5.7.5 Fulladderen, utledet

Først en liten navnesak, så du ikke snubler i den.

Menten heter **carry** på engelsk, og det er ordet du møter i all litteratur og alle kretstegninger. Vi forkorter den derfor med bokstaven C. Menten som kommer *inn* i kretsen fra posisjonen til høyre skriver vi C_{inn}, og den som går *ut* til posisjonen til venstre skriver vi C_{ut}.

Det er samme ting som «mente» hele veien. Vi bruker det norske ordet i teksten og den engelske forkortelsen i tabeller og formler, rett og slett fordi C_{inn} tar mindre plass enn «menten som kommer inn».

Tre innganger gir åtte rader. Vi setter dem opp i tellerekkefølge og fyller ut hva summen av de tre bitene blir.

A	B	C_{inn}	Sum	C_{ut}	Hvor mange ettall inn
0	0	0	0	0	ingen
0	0	1	1	0	ett
0	1	0	1	0	ett
0	1	1	0	1	to
1	0	0	1	0	ett
1	0	1	0	1	to
1	1	0	0	1	to
1	1	1	1	1	tre

Tabell 5.8 Fulladderen. Den siste kolonnen er bare til kontroll: null ettall gir 0, ett gir 1, to gir 0 med mente, og tre gir 1 med mente.

Nå bruker vi metoden fra seksjon 5.5 på hver av de to utgangene.

Mente ut er 1 i fire rader. Det gir:

$$C_{ut} = \overline{A}BC_{inn} + A\overline{B}C_{inn} + AB\overline{C_{inn}} + ABC_{inn}$$

Her kommer idempotensregelen til nytte, den som ser ut som den aldri skal brukes til noe. $X + X = X$ betyr også at vi kan skrive det samme leddet flere ganger uten å endre noe. Vi skriver ABC_{inn} tre ganger i stedet for én:

$$C_{ut} = (\overline{A}BC_{inn} + ABC_{inn}) + (A\overline{B}C_{inn} + ABC_{inn}) + (AB\overline{C_{inn}} + ABC_{inn})$$

Nå har hver parentes en felles faktor, og i hver av dem står et komplementpar som blir 1:

$$\begin{aligned}
\overline{A}BC_{inn} + ABC_{inn} &= BC_{inn}(\overline{A} + A) = BC_{inn} \\
A\overline{B}C_{inn} + ABC_{inn} &= AC_{inn}(\overline{B} + B) = AC_{inn} \\
AB\overline{C_{inn}} + ABC_{inn} &= AB(\overline{C_{inn}} + C_{inn}) = AB
\end{aligned}$$

Altså:

$$C_{ut} = AB + AC_{inn} + BC_{inn}$$

Les det høyt: mente ut er 1 når A og B er 1, *eller* A og menten inn er 1, *eller* B og menten inn er 1.

Med andre ord: **mente ut er 1 når minst to av de tre inngangene er 1.** Sjekk mot den siste kolonnen i tabell 5.8 på forrige side, så ser du at det stemmer for hver rad.

Summen er 1 i de fire andre radene:

$$\text{Sum} = \overline{A}\,\overline{B}C_{inn} + \overline{A}B\overline{C_{inn}} + A\overline{B}\,\overline{C_{inn}} + ABC_{inn}$$

Grupper de to første og de to siste etter om A er invertert:

$$\text{Sum} = \overline{A}(\overline{B}C_{inn} + B\overline{C_{inn}}) + A(\overline{B}\,\overline{C_{inn}} + BC_{inn})$$

Se på de to parentesene. Den første betyr «B og C_{inn} er forskjellige», altså $B \oplus C_{inn}$. Den andre betyr «de er like», altså $\overline{B \oplus C_{inn}}$.

Kall den første parentesen X. Da er den andre $\overline{X}$, og vi har:

$$\text{Sum} = \overline{A}X + A\overline{X}$$

Det betyr «A og X er forskjellige», altså:

$$\text{Sum} = A \oplus B \oplus C_{inn}$$

Tankevekker

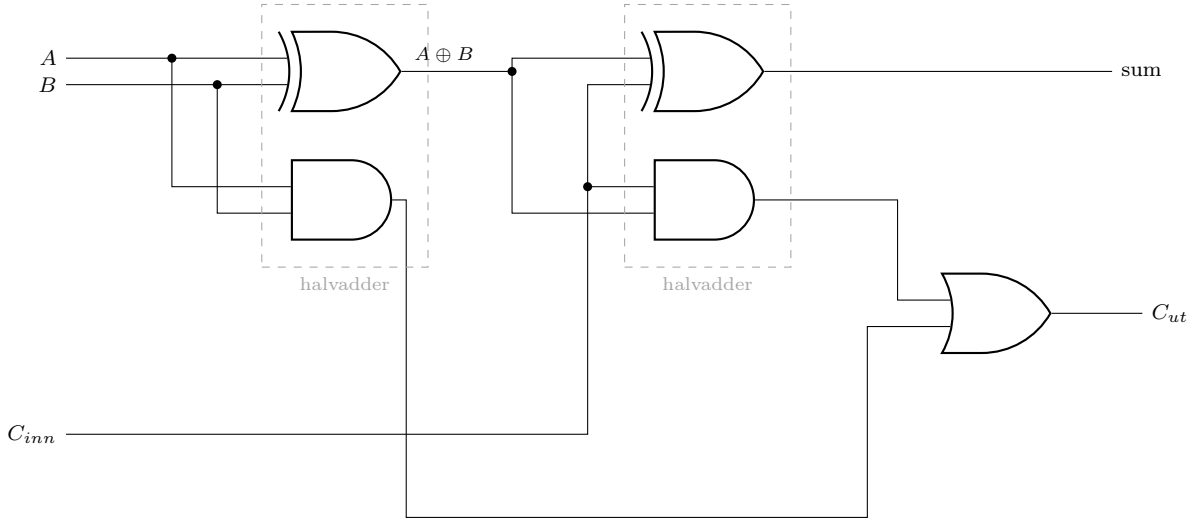
Kapittel 4 sa at vi ikke åpnet ALU-en. Nå er den åpen, og det som lå inni var to formler:

$$\text{Sum} = A \oplus B \oplus C_{inn} \qquad C_{ut} = AB + AC_{inn} + BC_{inn}$$

Ingen har gitt oss dem. Vi skrev opp hva vi ville ha i en tabell med åtte rader og regnet oss fram til formlene med reglene fra forrige seksjon.
Det er dette hele kapitlet har bygget mot: fra «slik skal det oppføre seg» til «slik skal det bygges», og det krevde ikke mer enn tabellen og sju regler.

5.7.6 Fulladderen, tegnet

Du kan bygge kretsen rett fra formlene. Men det finnes en penere vei: en fulladder er to halvaddere og en OR-port.



Figur 5.14 Fulladderen, bygget av to halvaddere. Den første legger sammen A og B. Den andre legger menten inn til resultatet. OR-porten fanger opp at mente kan oppstå i begge trinnene, men aldri i begge samtidig.

Tankegangen er den samme som når du legger sammen tre tall på papir: ta to av dem først, og legg så det tredje til summen.

Den første halvadderen regner ut $A+B$. Den andre legger C_{inn} til resultatet. Mente kan oppstå i begge trinnene, og OR-porten slipper den videre uansett hvilket av dem det var.

Note-to-myself

Hvorfor holder det med en OR, når det kunne vært to menter å legge sammen?
Fordi de aldri oppstår samtidig. Gir den første halvadderen mente, var både A og B lik 1, og da er summen deres 0. En halvadder med 0 på inngangen kan ikke gi mente, så situasjonen «mente fra begge» finnes ikke.
Regn gjennom alle åtte radene, så ser du at det aldri skjer.

Kretsen bruker fem porter: to XOR, to AND og en OR. Det er prisen for å legge sammen ett bit.

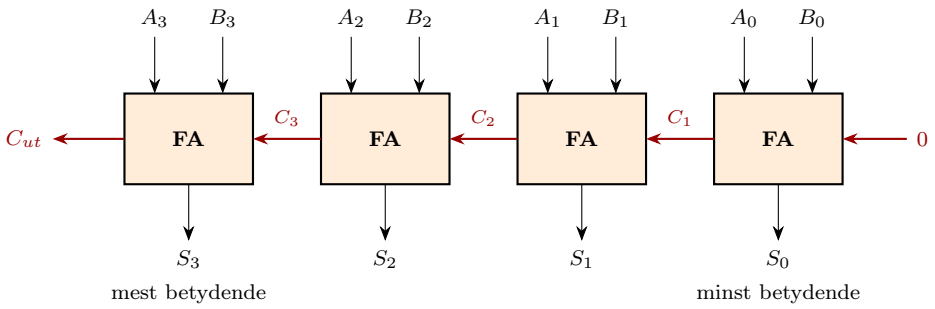
5.8 Mentekjeden

Én fulladder legger sammen ett bit, men et tall har flere.

Resten av arbeidet er kopiering: sett inn én fulladder per bitposisjon, og koble menten fra hver av dem inn i den neste.

5.8.1 Fire på rad

Her er fire bit. Hver boks er en hel fulladder, med de fem portene fra forrige figur inni.



Figur 5.15 Fire fulladdere i rekke. Menten går mot venstre, samme vei som når du legger sammen på papir. Den første får 0 inn, siden det ikke finnes noen kolonne til høyre for den.

Legg merke til retningen. Bitene står slik tall skrives, med det minst betydende til høyre, og menten beveger seg mot venstre. Det er samme vei som blyanten din går når du legger sammen i margen.

Kretsen har to navn. På engelsk heter den *ripple carry adder*, fordi menten «rippler», altså bølger, gjennom rekken. På norsk sier mange *seriell adder*. Vi kaller den **mentekjeden** når vi snakker om selve problemet, siden det er kjeden som er poenget.

5.8.2 Svaret er ikke riktig med én gang

Nå kommer det som gjør denne kretsen interessant.

Slå på alle inngangene samtidig. Hver fulladder begynner å regne i samme øyeblikk, og hver av dem gir fra seg et svar nesten umiddelbart.

Problemet er at de fleste av svarene er feil.

Den høyre fulladderen får 0 som mente inn og vet det helt fra start, så den kan regne seg ferdig. S_0 er riktig.

Den neste venter på C_1. Den har riktignok noe på menteinngangen hele tiden, men det er en verdi som ennå ikke har satt seg. S_1 er først riktig etter at C_1 er ferdig beregnet.

Slik fortsetter det. Hver posisjon må vente på den til høyre.

Det er som en stafett. Hver løper står klar med hånda utstrakt, men kan ikke starte før stafettpinnen kommer. Den første løperen har pinnen fra begynnelsen, mens den siste ikke kan gjøre noe før alle de andre har løpt sin etappe.

Tankevekker

Dette er dominoeffekten, bokstavelig talt. Den første brikken faller med én gang, men den siste faller ikke før alle de foran har falt.
Og det er ikke bare tregt. Det er en *feilkilde* hvis du leser av for tidlig. Ser du på utgangene like etter at inngangene ble satt, finner du et tall som ser helt rimelig ut og som er galt. Ingenting varsler deg.
Derfor må maskinen vente et fast antall klokkeslag før den bruker svaret. Kapittel 4 sa at klokka er stilt etter det tregeste steget. Mentekjeden er ofte det steget.

5.8.3 Hva det koster

Si at én fulladder bruker tiden t på å lage menten sin. Da bruker rekken omtrent

$$T \approx n \cdot t$$

der n er antall bit.

Tiden vokser altså rett proporsjonalt med bredden. Dobler du fra 32 til 64 bit, dobler du ventetiden.

Note-to-myself

«Omtrent» står der med vilje. Den siste fulladderen i rekken trenger bare å lage summen sin, ikke en ny mente som noen skal vente på. Den eksakte veien er derfor litt kortere enn $n \cdot t$.
Det som betyr noe er formen på svaret: tiden vokser *lineært* med antall bit. Det er den egenskapen som gjør konstruksjonen til et problem når tallene blir brede.

Det verste tilfellet er lett å lage. Legg sammen 0111 og 0001:

```
  0 1 1 1
+ 0 0 0 1
---------
  1 0 0 0
```

Menten som oppstår helt til høyre må videre gjennom hver eneste posisjon. Ingen av dem kan bli ferdig før naboen er det, så hele kjeden må gjennomløpes.

Nå kan vi også forklare noe kapittel 4 bare slo fast: hvorfor en ELLER mellom to 64-bits tall kan gå raskere enn en addisjon av de samme tallene.

En ELLER har ingen kjede. Hver bitposisjon har sin egen OR-port, og ingen av dem trenger å vite hva naboen kom fram til. Alle 64 blir ferdige samtidig. Addisjonen må derimot vente på menten, posisjon for posisjon.

Tenk om. . .

. . . du skulle gjøre addisjonen raskere. Hva ville du gjort?
Den nærliggende ideen er å finne ut hva menten blir *før* den rekker fram. Det går an: hvis A_0 og B_0 begge er 1, vet du at det blir mente ut derfra, uten å vente på noen.
Når denne tanken settes i system, kalles den *carry look-ahead*, og det er den moderne prosessorer bruker. Prisen er mange flere porter og mer strøm, for å slippe å vente. Det er samme avveining som har gått igjen i hele boka: plass mot tid.

5.8.4 Her kommer C-flagget fra

Kapittel 4 innførte fire flagg og forklarte hva de betyr. Tabell 4.4 på side 93 står der fortsatt. Det som manglet, var hvor de fysisk kommer fra.

For to av dem finner du svaret i figur 5.15 på side 131.

C-flagget er mente ut av den venstre fulladderen. Det er bare den ene ledningen som stikker ut til venstre i figuren, og som ingen flere fulladdere finnes til å ta imot.

Mer er det ikke. Når det ble mente ut av den øverste posisjonen, var svaret for stort for bitbredden, og den ledningen går rett inn i statusregisteret.

5.8.5 Og her kommer V-flagget fra

Overflyt er litt vanskeligere å få tak i, men kretsen er fin og liten.

V skal settes når svaret er feil *som fortegnstall*. Kapittel 2 forklarte toerkomplement, og kapittel 4 forklarte hvorfor mente og overflyt ikke er det samme. Spørsmålet her er bare hvordan kretsen oppdager det.

Svaret er å sammenligne menten *inn i* den øverste posisjonen med menten *ut av* den.

$$V = C_{inn,\,\text{øverst}} \oplus C_{ut,\,\text{øverst}}$$

Er de like, gikk det bra. Er de forskjellige, er det overflyt.

Hele overflytdeteksjonen er altså **én XOR-port**, koblet til de to menteledningene rundt den venstre fulladderen.

La oss sjekke det på eksempelet fra selv-testen i kapittel 4: $-5 + (-6)$ i fire bit. I toerkomplement er $-5 = 1011$ og $-6 = 1010$.

Posisjon	A	B	mente inn	sum	mente ut
0 (høyre)	1	0	0	1	0
1	1	1	0	0	1
2	0	0	1	1	0
3 (venstre)	1	1	0	0	1

Tabell 5.9 $1011 + 1010$, regnet posisjon for posisjon. Summen blir 0101.

Øverste posisjon har mente inn $= 0$ og mente ut $= 1$. De er forskjellige, så $V = 1$. Og $C = 1$, siden det ble mente ut.

Svaret 0101 er $+5$. Vi la sammen to negative tall og fikk et positivt, og flagget hadde rett.

Note-to-myself

Se hvor lite dette kostet. Fire fulladdere og så én XOR-port på toppen, og maskinen har et pålitelig varsel om at svaret er galt.
Det som er dyrt, er ikke å oppdage overflyt, men å gjøre noe med det. Kretsen sier fra, og om noen hører etter, er opp til programmet. Det var poenget i kapittel 2, og nå ser du at advarselen

bokstavelig talt henger der som en ledning.

5.8.6 Subtraksjon, nesten gratis

Kapittel 2 lovet at en maskin ikke trenger en egen subtraksjonskrets, fordi $a-b$ kan regnes som $a+(-b)$. I toerkomplement lager du $-b$ ved å invertere alle bitene i b og legge til 1.

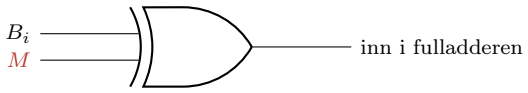
Begge delene har vi allerede.

Å invertere hvert bit i b krever én NOT-port per posisjon. Å legge til 1 betyr å sette menten inn til den høyre fulladderen til 1 i stedet for 0, og den inngangen står der allerede, ubrukt, i figur 5.15 på side 131.

Det siste trinnet er smart. Vi vil kunne velge mellom addisjon og subtraksjon, så vi trenger en bryter. Vi bytter hver NOT-port ut med en XOR-port, der den ene inngangen er biten fra b og den andre er en styreledning vi kaller M.

- Er $M = 0$, slipper XOR-porten biten uendret gjennom, og vi setter menten inn til 0. Kretsen legger sammen.
- Er $M = 1$, gir XOR-porten den inverterte biten, og vi setter menten inn til 1. Kretsen trekker fra.

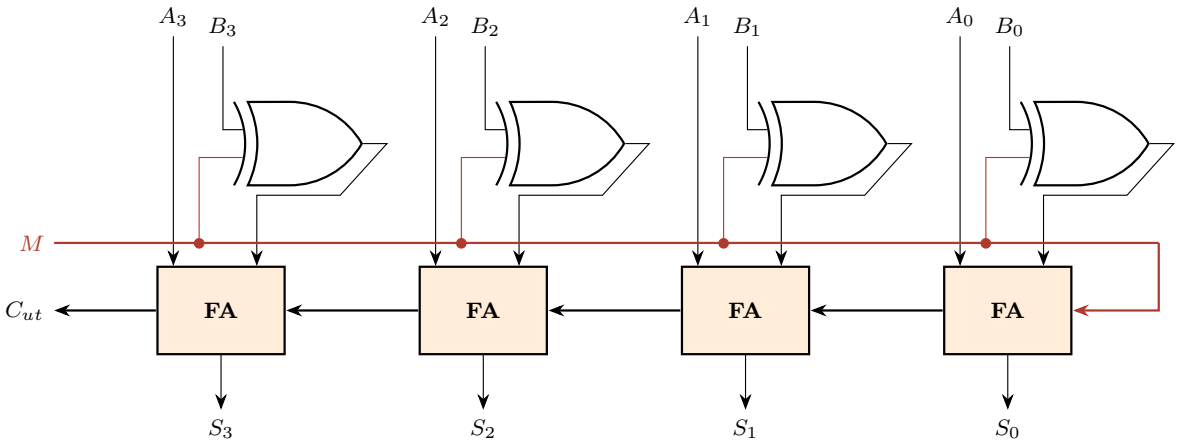
Den samme ledningen M styrer begge deler. Ett bit avgjør om kretsen er en adder eller en subtraher.



M	B_i	ut av XOR
0	0	0 (uendret)
0	1	1 (uendret)
1	0	1 (snudd)
1	1	0 (snudd)

Figur 5.16 XOR-porten brukt som *styrt* inverter. Står M på 0, slipper den B_i rett gjennom. Står M på 1, snur den. Det er hele trikset.

Slik ser hele kretsen ut når den er satt sammen.



Figur 5.17 Den samme mentekjeden som før, med én XOR-port lagt til foran hver B-inngang. Den røde ledningen M gjør to ting samtidig: den forteller alle XOR-portene om de skal snu, og den går helt til høyre og inn som mente til den første fulladderen. Står M på 0, legger kretsen sammen. Står den på 1, trekker den fra.

Tell hva som er lagt til: fire XOR-porter og én ledning. Det er hele prisen for at maskinen også kan trekke fra.

Advarsel :)

Når kretsen trekker fra, betyr C-flagget noe annet.
Ved addisjon er mente ut av den øverste posisjonen et varsel om at svaret ble for stort. Ved subtraksjon er den samme ledningen 1 akkurat når $a \geq b$, altså når regnestykket *ikke* trengte å låne.
Det er samme ledning og samme krets, men motsatt tolkning. Prosessoren vet hvilken som gjelder, fordi den vet hva M sto på.

Tankevekker

Nå lukkes en sirkel som har vært åpen i tre kapitler.
Kapittel 2 valgte toerkomplement, og begrunnelsen var at addisjon da virker uendret for negative tall. Kapittel 4 sa at ALU-en slipper unna uten egen subtraksjonskrets. Her ser du regningen: det koster *én XOR-port per bit*, og en styreledning som uansett måtte finnes.
Og styreledningen M er ingen tilfeldig ledning. Den kommer fra opkoden i instruksjonsordet, via kontrollenheten. Skriver du `SUB` i stedet for `ADD`, er forskjellen til slutt at én ledning inni ALU-en blir 1 i stedet for 0.

5.9 Kretser som velger

Vi har bygget kretser som regner. Nå skal vi bygge to som *velger*, og den første er det tredje løftet fra kapittel 4.

5.9.1 Dekoderen

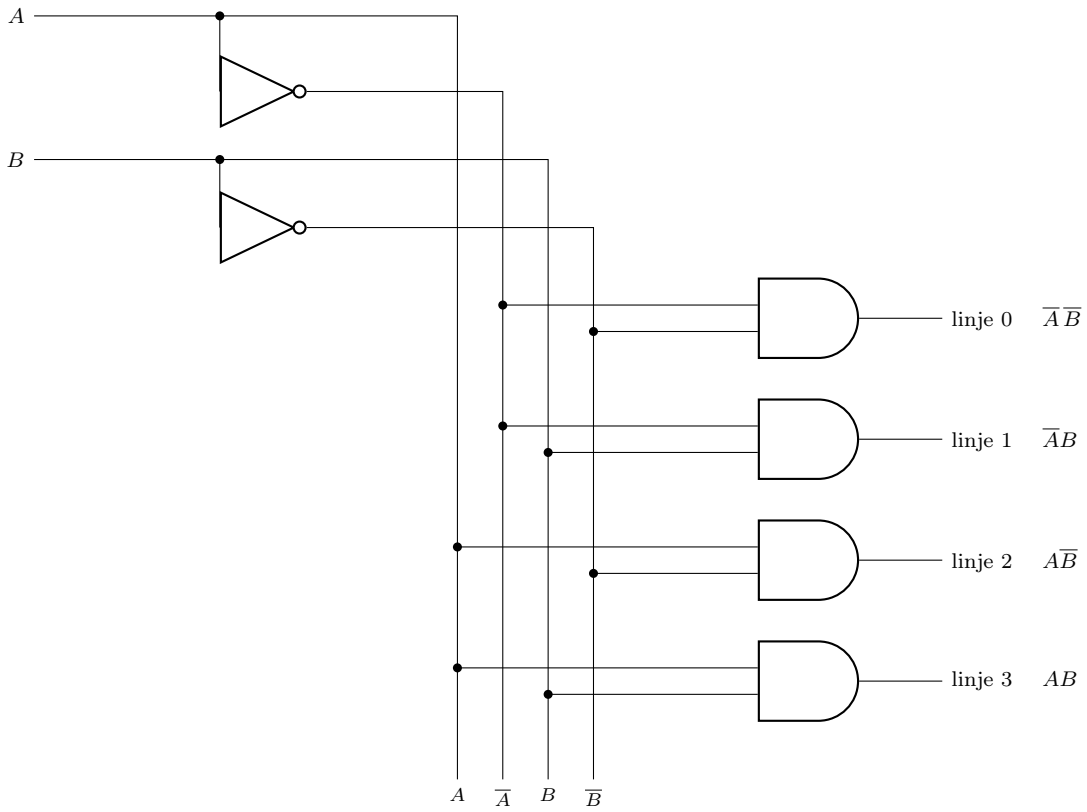
Kapittel 4 definerte en **dekoder** som en krets med noen få innganger og én utgang for hver mulige kombinasjon, der nøyaktig én utgang er på til enhver tid. Figur 4.6 på side 95 tegnet den med fire innganger og seksten utganger.

Spørsmålet var hvordan den bygges. Du har allerede skrevet svaret, uten å vite det.

Gå tilbake til seksjon 5.5. Der skrev vi ett ledd per rad i tabellen, og hvert ledd var en AND mellom alle inngangene, invertert der raden hadde 0.

En dekoder er alle de leddene på én gang, hvert med sin egen utgang.

Tenk deg en tavle i en resepsjon med én lampe for hvert rom. Taster du inn romnummeret, lyser bare lampa for akkurat det rommet. Det er en dekoder: et nummer inn, og én linje ut.



Figur 5.18 En dekoder med to innganger og fire utganger. De fire loddrette ledningene bærer A, $\overline{A}$, B og $\overline{B}$. Hver AND-port henter fra to av dem, og svarer dermed til én rad i sannhetsverditabellen.

Sett $A = 1$ og $B = 0$. Da er $A = 1$, $\overline{A} = 0$, $B = 0$ og $\overline{B} = 1$.

Porten for linje 2 henter fra A-bussen og $\overline{B}$-bussen, altså 1 og 1. Den slår ut.

De tre andre får minst én null inn, og en AND med en null er alltid 0. De er mørke.

Nøyaktig én utgang er på. Det var definisjonen, og nå ser du hvorfor den alltid holder: hver port er skreddersydd for sin ene kombinasjon.

5.9.2 Opkoden inn, styresignalet ut

Utvid figuren til fire innganger, så har du seksten AND-porter med fire innganger hver. Det er figuren fra kapittel 4.

Nå kan vi følge hele veien.

Kapittel 3 skrev `LDA 42`. Kapittel 4 gjorde det om til bitmønsteret 0101 00101010, der de fire første bitene er opkoden 0101.

De fire bitene går inn i dekoderen. Én AND-port har mønsteret $\overline{b_3} \cdot b_2 \cdot \overline{b_1} \cdot b_0$, og den får fire ettall inn. Den slår ut, mens alle de femten andre er mørke.

Utgangen fra den ene porten er `LDA`-linja. Derfra går den til bryterne i datastien og åpner de veiene som skal åpnes.

Tankevekker

Dette er hele koblingen mellom et program og en krets, og den er ikke lengre enn dette.
Du skrev en ordre. Assembleren gjorde den om til et tall. Tallet er et bitmønster, og fire av bitene traff én AND-port, som slo ut. Den porten er grunnen til at maskinen gjør `LDA` og ikke `ADD`.
Ingen tolker noe underveis. Ingen leser instruksjonen og forstår hva den betyr. Det er bare en port som får fire ettall inn.

5.9.3 Multiplekseren

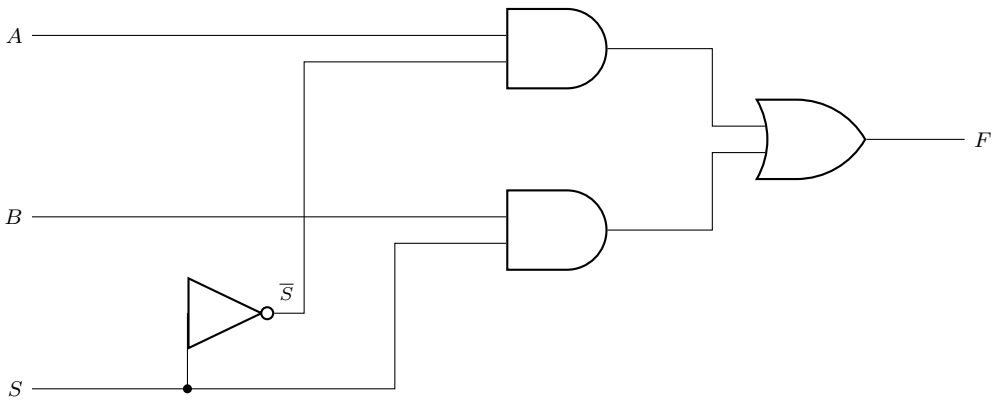
Den motsatte kretsen heter **multiplekser**, ofte forkortet mux. En dekoder tar et tall og velger én linje. En multiplekser tar flere linjer og velger ut én av dem.

Med to datainnganger og én styreinngang blir uttrykket:

$$F = \overline{S} \cdot A + S \cdot B$$

Er $S = 0$, faller andre ledd bort og F blir A. Er $S = 1$, faller første ledd bort og F blir B.

Det er en sporveksel: to spor kommer inn, og styresignalet bestemmer hvilket av dem toget får fortsette fra. Her er den bygget av to AND-porter, en OR-port og en inverter.



Figur 5.19 En multiplekser med to datainnganger. Styresignalet S åpner den nedre AND-porten og stenger den øvre, eller omvendt. Nøyaktig én av de to slipper noe gjennom til OR-porten.

Legg merke til at ideen er den samme som i dekoderen: en AND-port brukes som en grind, der det ene signalet bestemmer om det andre får slippe gjennom.

Note-to-myself

Multipleksere finnes overalt inni en prosessor, selv om de sjelden tegnes. Hver gang datastien har to mulige kilder til samme sted, sitter det en multiplekser der og velger.
Et eksempel du har sett er subtraksjonskretsen i forrige seksjon, som valgte mellom b og $\overline{b}$. Vi løste det med en XOR-port, som er billigere her, men en multiplekser ville gjort samme jobb.

5.10 Kretsen som husker

Alt vi har bygget til nå har én ting felles: utgangen følger inngangene hele tiden. Endrer du en inngang, endrer utgangen seg rett etterpå. Kretsen har ingen fortid.

Slike kretser kalles **kombinatoriske**. Portene, adderen, dekoderen og multiplekseren er alle kombinatoriske.

Men kapittel 4 sa at et register holder på en verdi så lenge strømmen står på. Det kan ingen kombinatorisk krets gjøre. Tar du bort inngangen, forsvinner svaret.

Så hvordan kan noe huske?

Ideen får plass i én setning: **koble utgangen tilbake til inngangen.**

Da er kretsen ikke lenger bare avhengig av det som kommer inn nå, men også av hva den selv sa sist. Er tilbakekoblingen riktig satt opp, kan kretsen holde seg selv i ro i den tilstanden den er i, helt til noen med vilje dytter den over i den andre.

Du har sett noe lignende hvis du noen gang har holdt en mikrofon for nær høyttaleren. Lyden går ut, kommer inn igjen og forsterker seg selv, og kretsen «husker» hvinet helt til noen flytter mikrofonen. Å lage noe som holder seg rolig i stedet for å hvine er et spørsmål om å koble det riktig.

To NOR-porter som mater hverandre holder akkurat ett bit på denne måten. Registrene i kapittel 4 er bygget av dette, og det samme er cachen.

Kretser med tilbakekobling kalles **sekvensielle**, fordi utgangen avhenger av rekkefølgen på det som har skjedd, ikke bare av hva som står på inngangene nå. Alt annet i dette kapitlet har vært kombinatorisk.

Her stopper vi, men historien fortsetter, og resten ligger rett bak i boka.

Vil du se resten? Appendiks C

Appendiks C, «Kretser som husker», tar denne ene siden videre til seks. Veien dit går slik:

- **SR-latchen.** De to NOR-portene tegnet opp, med en forklaring på hvorfor de blir stående i ro, og hva som skjer i den ene kombinasjonen som er forbudt.
- **D-latchen.** Hvordan du gjør det umulig å havne i den forbudte tilstanden, og hvordan du bestemmer *når* kretsen skal lytte.
- **Vippa.** Forskjellen på en dør som står åpen og en som smeller igjen i et bestemt øyeblikk. Dette er byggeklossen alt synkront utstyr er laget av.
- **Registeret.** Åtte vipper med felles klokkeledning, og dermed svaret på hvorfor kapittel 4 sa at registre er raske, dyre og få.

Det er ikke pensum i IN1020, og du kan hoppe over det uten å miste noe du blir prøvd i. Men det er her løftet fra kapittel 4 blir innfridd helt, og det er en fin halvtime hvis du vil se hvordan minne egentlig virker.

5.11 Oppsummering

Kapitlet begynte med to brytere i en trappeoppgang og endte med en krets som kan huske. Underveis har vi bygget noe av det maskinen faktisk består av, av deler som bare kan si ja og nei.

5.11.1 Hva som egentlig var nytt

Begrep	**Kort sagt**
Logisk port	Krets med én utgang som utfører én logisk operasjon. Sju typer.
Logisk funksjon	*Hva* som skal skje. Kan skrives som uttrykk eller tabell.
Krets	*Hvordan* det skjer. Samme funksjon kan bygges på mange måter.
Sannhetsverditabell	Alle 2^n kombinasjoner og hva utgangen blir. Uttømmende.
Sum av produkter	Ett AND-ledd per rad med 1 på utgangen, bundet sammen med ELLER.
Absorpsjon	$\overline{X} + XY = \overline{X} + Y$. Regelen som oftest blir glemt.
De Morgans lover	Overstrek inn gjennom en parentes snur operatoren.
Halvadder	To bit inn, sum og mente ut. XOR og AND. Mangler mente inn.
Fulladder	Tre bit inn, sum og mente ut. Fem porter. Byggeklossen i en ALU.
Mentekjede	Fulladdere i rekke. Tiden vokser lineært med antall bit.
C-flagget	Mente ut av den øverste posisjonen. Én ledning.
V-flagget	XOR av mente inn og mente ut i øverste posisjon. Én port.
Dekoder	Én AND-port per kombinasjon. Nøyaktig én utgang på om gangen.
Multiplekser	Velger én av flere innganger. $F = \overline{S}A + SB$.
Kombinatorisk krets	Utgangen følger bare inngangene. Ingen fortid.
Sekvensiell krets	Har tilbakekobling, og dermed hukommelse.

Tabell 5.10 Begrepene fra dette kapitlet.

Den gjennomgående ideen kom igjen fire ganger. Tre porter ble til én NOR. Fire ledd og åtte porter ble til én AND og én XNOR. Overflytdeteksjonen ble til én XOR. Subtraksjon ble til én XOR per bit

og en styreledning som allerede fantes.

Hver gang var det samme mønsteret: det du først skriver ned, er ikke det du til slutt bygger. Mellom de to ligger et lite sett regler og litt tålmodighet.

Hver gang ga det også mer enn sparte porter. Den forenklede kretsen lot seg lese. $\overline{A} + B$ er «hvis A, så B». $\overline{C \oplus AB}$ er «er C lik OG-en av A og B?». Det var umulig å se i det opprinnelige uttrykket.

5.11.2 Videre til neste kapittel

Kapittel 4 pekte tre ganger nedover. ALU-en er åpnet, dekoderen er bygget, og hukommelse viste seg å være tilbakekobling. Den siste fikk bare ideen her, og selve kretsene står i appendiks C.

Men én ting står igjen, og den er litt ubehagelig.

Portene finnes ikke.

Det er ingen spissfindighet. Inni prosessoren din finnes det ikke én eneste AND-port. Går du ett lag ned, finner du transistorer, små brytere av halvleder, og noen av dem er koblet sammen slik at gruppen oppfører seg som en AND.

Porten er et symbol vi har funnet opp fordi det er umulig å tegne en milliard brytere. Den er en abstraksjon, på linje med dem kapittel 1 handlet om, like nyttig og like uvirkelig som alle de andre.

Får du et kretsskjema med porter, kan du lese det. Får du det samme skjemaet tegnet med transistorer, er det uleselig. Det er hele grunnen til at porten finnes.

Kapittel 6 tar det siste steget ned, til selve bryteren, og til strømmen og spenningen som gjør at den er en bryter.

Det er bunnen. Lenger ned går vi ikke.

5.12 Videre lesning

Englander dekker ikke dette stoffet i boka. Det er greit å si rett ut, siden pensumboka har vært hovedkilden i alle tidligere kapitler. Boolsk algebra og digital logikk ligger i *Supplementary Chapter 1*, som forlaget legger ut på nett og ikke trykker. Finner du den, dekker den det samme som dette kapitlet, med flere eksempler og mer sekvensiell logikk.

Harris & Harris, *Digital Design and Computer Architecture*, kapittel 2 og 5. Dette er den beste neste boka hvis du vil videre. Kapittel 2 tar porter, boolsk algebra og Karnaugh grundig. Kapittel 5 bygger addere, multiplikatorer og en hel ALU. Vær klar over at den bruker andre portsymboler enkelte steder, og at den går raskt fram.

Nand2Tetris, fritt tilgjengelig på nett. Et kurs der du bygger en hel datamaskin fra NAND-porter og oppover, i en simulator. Du lager ingen fysiske kretser, men du må løse hvert trinn selv. Det er den beste måten å oppdage at alt faktisk henger sammen.

Appendiks B tar forenkling systematisk med Karnaugh-diagram, og **appendiks C** tar sekvensiell logikk fra tilbakekobling til ferdig register. Ingen av dem er pensum.

Videre i denne boka: kapittel 6 viser hva en port er bygget av. Appendiks E forklarer hvorfor alt må skje i takt med en klokke, og det blir først helt tydelig etter at du har sett mentekjeden.

5.13 Refleksjon

1. Vi så at tre porter kunne erstattes av én, og at åtte kunne erstattes av to. Likevel bygger ingen kretser for hånd lenger, for det gjør programvare. Hva er da vitsen med å kunne forenkle selv?
2. En mentekjede på 64 bit er treg, men enkel og liten. En carry look-ahead er rask, men stor og strømsulten. En designer må velge. Hvilke opplysninger om bruken trenger hun for å velge riktig?
3. Overflytdeteksjonen koster én port. Likevel har alvorlige feil oppstått fordi ingen sjekket flagget. Hvis varselet er nesten gratis å lage, hvorfor blir det ikke sjekket?
4. Porter finnes ikke, sa vi. De er et abstraksjonsnivå vi har funnet opp for å kunne tenke. Finn et annet felt enn datateknologi der noe tilsvarende gjelder: et begrep alle bruker og regner med, som ikke svarer til noe fysisk.

5.14 Selv-test

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

1. Forklar med egne ord forskjellen på en logisk funksjon og en krets.
2. Hvor mange rader har en sannhetsverditabell med fire innganger? Og med seks?
3. NOT-porten har én inngang. Hvor mange innganger tegner vi de seks portene i tabell 5.1 på side 116 med, og hvorfor er ikke det en grense for hva de kan ha?
4. Hvilken port gir 1 bare når de to inngangene er forskjellige? Og hvilken gir 1 bare når de er like?
5. Skriv sannhetsverditabellen for en NAND-port.
6. Sett $A = 1$ og $B = 1$. Hva gir hver av de seks portene i tabell 5.1 på side 116?
7. Hvorfor er $1 + 1 = 1$ i boolsk algebra, når det ikke er det i vanlig regning?
8. Hva er forskjellen på $\overline{A}\,\overline{B}$ og $\overline{A \cdot B}$? Gi et par inngangsverdier der de to gir ulikt svar.
9. Tegn kretsen for $F = \overline{A}B + A\overline{B}$ med grunnleggende porter. Hvilken enkeltport gjør akkurat det samme?
10. En krets har denne tabellen. Skriv funksjonsuttrykket på sum-av-produkter-form, og forenkle det så langt du kommer.

A	B	F
0	0	0
0	1	1
1	0	1
1	1	1

11. Sett opp sannhetsverditabellen for $F = \overline{A} + B$.
12. Forenkle $F = AB + A\overline{B}$.
13. Forenkle $F = \overline{A} + AB$. Hvilken regel brukte du?
14. Forenkle $F = ABC + AB\overline{C}$.
15. Forenkle $F = AB + AC + A\overline{B}\,\overline{C}$. Svaret er kortere enn du tror.
16. Bruk De Morgans lov på $\overline{A + \overline{B}}$, og forenkle.
17. I hvor mange rader av en tabell med tre innganger er leddet $\overline{A}\,\overline{B}\,\overline{C}$ lik 1? Hvilken rad er det?

18. En krets gir $1, 0, 0, 1$ for radene 00, 01, 10, 11. Hvilken port er det, og hva blir uttrykket på sum-av-produkter-form?
19. **Finn feilen.** En student har en krets med en inverter på A, en inverter på B, og begge inn i en OR-port. Studenten skriver $F = \overline{A + B}$. Hva er det riktige uttrykket, og hvilken enkeltport svarer studentens uttrykk til?
20. Hvorfor heter halvadderen «halv»?
21. Hvor mange porter er det i en fulladder bygget av to halvaddere og en OR-port, og hvilke typer?
22. Hvorfor holder det med én OR-port for å lage mente ut i fulladderen, i stedet for en ny adder?
23. Skriv sum-av-produkter-uttrykket for mente ut i fulladderen ut fra tabell 5.8 på side 128, og forenkle det til tre ledd.
24. En 4-bits mentekjede legger sammen 0110 og 0101. Sett opp mente inn og mente ut for hver av de fire posisjonene. Hva blir summen, og hva blir C og V?
25. En mentekjede på 8 bit bruker omtrent 8 nanosekunder. Hvor lang tid bruker en på 32 bit, omtrent, og hvorfor?
26. Hvorfor kan en ELLER mellom to 64-bits tall gå raskere enn en addisjon av de samme to tallene?
27. Hvor mange utganger har en dekoder med tre innganger? Hvor mange AND-porter og hvor mange invertere trengs?
28. Opkoden 0101 kommer inn i en dekoder med fire innganger. Hvilken linje slår ut, og hvilke fire signaler får AND-porten for den linja inn?
29. Hva skiller en kombinatorisk krets fra en sekvensiell? Gi ett eksempel på hver fra dette kapitlet.
30. I subtraksjonskretsen styrer ledningen M både XOR-portene og menten inn til den første fulladderen. Hva ville kretsen regnet ut hvis du satte $M = 1$ på XOR-portene, men glemte å sette menten inn til 1?

Kapittel 6

Transistor, strøm og spenning

Innhold

6.1 Læringsmål . . . 144
6.2 Bryteren som ikke trenger en finger . . . 145
6.3 Strøm, spenning og jord . . . 147
6.4 Inverteren: den første porten vi faktisk bygger . . . 149
6.5 NAND, NOR, og hvorfor AND er dyrere . . . 151
6.6 Null og ett er områder, ikke tall . . . 154
6.7 Det tar tid . . . 156
6.8 Den lengste veien . . . 158
6.9 Klokka . . . 160
6.10 Hvorfor klokka stoppet på rundt 3 GHz . . . 163
6.11 Oppsummering . . . 167
6.12 Videre lesning . . . 169
6.13 Refleksjon . . . 169
6.14 Selv-test . . . 170

Telefonen din blir varm.

Du har kjent det. Etter en halvtimes spilling eller en lang videosamtale er baksiden lun, og har du laptop, hører du viften spinne opp og bli stående og suse. Det er litt rart når du tenker etter. Det beveger seg ingenting der inne: ingen motor, ingen tannhjul, ingenting som gnisser mot noe annet. Prosessoren i en telefon har ikke én eneste bevegelig del.

Hva er det da som blir varmt?

Du har ikke svaret ennå, og det er helt greit. Det ligger et lag lenger ned enn vi har vært så langt.

Tankevekker

Varmen er ikke en bieffekt vi kunne konstruert oss bort fra. Den *er* regningen for arbeidet. Hver gang maskinen bytter et bit fra 0 til 1, tar den en liten slurk strøm, og hele slurken ender som varme.
En moderne prosessor gjør det noen milliarder ganger i sekundet, på noen milliarder steder samtidig. Da blir slurkene til et måltid.

Vi kommer hit med en gjeld. Kapittel 5 endte med en ubehagelig setning: portene finnes ikke. Du brukte et helt kapittel på å bygge med AND, OR og NOT, og så viste det seg at ingen av dem sitter

i maskinen. De er tegninger, symboler vi har funnet på fordi ingen orker å tegne en milliard brytere. Da gjenstår spørsmålet om hva som *faktisk* sitter der.

Det du leter etter, kjenner du allerede. Du brukte det gjennom hele kapittel 5, og du bruker det hver morgen når du slår på lyset.

Du trenger ingen fysikk for å lese dette kapitlet: ingen elektronikk, ingen kjemi, ingenting om atomer. Vi snakker om strøm hele veien, men vi gjør det med vann. Vannbildet er ikke helt presist, men det bærer deg lenger enn du skulle tro, og akkurat så langt som dette faget går.

Vi regner litt. Det er bare gange og dele med runde tall, og svaret står rett ved siden av, så du kan lese videre selv om du ikke orker å regne etter. Det eneste du må ha med deg fra før, er at «pico» og «nano» er navn på svært små tall, og du får en tabell over dem når vi kommer dit. Formler får du lite av. Den du får mest bruk for, kobler klokkeperiode til frekvens. En til, om hvorfor brikker blir varme, står i en egen boks som du kan hoppe over.

Dette kapitlet handler om et lag du aldri har tenkt på, så det kommer til å føles fremmed de første sidene. Det er normalt, og det sier ingenting om hvor flink du er. Det går fortere hvis du leser rolig og har et ark ved siden av deg.

Hva du sitter igjen med

Når du er ferdig med kapitlet, kan du svare på tre spørsmål som de fleste ikke kan svare på.
Hvorfor blir en datamaskin varm, når ingenting beveger seg i den?
Hvorfor sluttet prosessorer å bli raskere rundt 2004, etter å ha blitt raskere hvert eneste år i tretti år?
Og hvorfor har maskinen din åtte kjerner i stedet for én kjerne som går åtte ganger så fort?
Alle tre har det samme svaret, og du finner det i bunnen av stabelen.

Her er vi nemlig nede. Kapittel 1 tegnet opp lagene, og siden har vi gått nedover, ett lag om gangen. Nå stopper vi. Under transistoren ligger halvlederfysikken, som hører hjemme i et annet fag, og fra neste kapittel går veien oppover igjen.

6.1 Læringsmål

Når du har arbeidet deg gjennom kapitlet, skal du kunne:

1. **Forklare hva en transistor gjør.** Den er en bryter som styres av et elektrisk signal i stedet for av en finger. Resten av kapitlet er følger av det.
2. **Kjenne igjen de to typene og si hva som skiller dem.** Den ene leder når styresignalet er 1, den andre når det er 0.
3. **Forklare hvorfor en port trenger strøm.** Og hvor nullene og ettallene egentlig kommer fra, som ikke er der du tror.
4. **Lese en inverter bygget av to brytere.** Du skal kunne si hva utgangen blir i begge tilstandene, uten å pugge noe.
5. **Telle transistorer i en port og forklare hvorfor NAND er billigere enn AND.** Her får du svar på et spørsmål kapittel 5 stilte og ikke besvarte.
6. **Forklare hvorfor 0 og 1 er spenningsområder og ikke tall.** Det er også grunnen til at digitale signaler tåler støy og analoge ikke gjør det.
7. **Regne ut hvor lang tid en krets bruker.** Finn den lengste veien gjennom kretsen, tell portene og gang med portforsinkelsen.

8. **Lese et klokkesignal og regne mellom periode og frekvens.** Av alle målene her er det dette du kommer til å bruke mest.
9. **Forklare hvorfor klokkefrekvensen sluttet å øke.** Og hvorfor svaret ble flere kjerner i stedet for raskere kjerner.

6.2 Bryteren som ikke trenger en finger

6.2.1 Noen må trykke på bryteren

Kapittel 5 begynte med to brytere og en lyspære, og alt som kom etterpå hvilte på dem. Brytere i serie ble OG, brytere i parallell ble ELLER, og derfra bygde vi oss helt opp til en krets som legger sammen tall.

Historien hadde ett hull, og vi lot det ligge: noen må trykke på bryterne. En bryter er et stykke metall som enten berører et annet stykke metall eller ikke, og for å flytte det må noe ta i. En finger, en fjær, en knapp. En datamaskin har ingen fingre.

Det verste er ikke engang det. Se på en hvilken som helst krets fra forrige kapittel. Utgangen fra én port går rett inn som inngang på den neste, og da er det utgangen fra den første porten som må trykke på bryterne i den andre. Vi trenger altså en bryter som et signal kan trykke på.

Hvorfor dette er det eneste som mangler

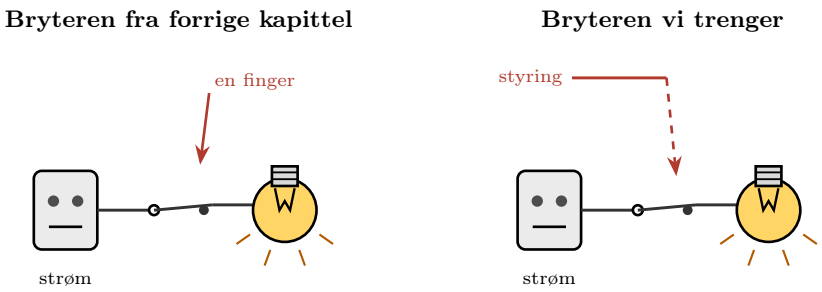
Alt annet i kapittel 5 fungerer allerede med vanlige brytere. Du kan bygge en fulladder av lysbrytere i kjelleren, koble den til et batteri, og den regner riktig. Folk har gjort det.
Den ville bare vært ubrukelig, fordi noen måtte stått der og trykket på bryterne i riktig rekkefølge. Akkurat den jobben skal vi bli kvitt.

6.2.2 En tredje ledning

Tenk deg bryteren fra forrige kapittel en gang til: to kontakter og et metallstykke som enten forbinder dem eller ikke. Gi den nå en tredje ledning, en som strømmen ikke går gjennom, men som bare *bestemmer*.

Vi sier at det er **spenning** på en ledning når den er koblet til strømkilden, og ingen spenning når den er koblet til jord. Det er de to tilstandene vi skriver som **1** og **0**. Hva spenning egentlig er, tar vi skikkelig i seksjon 6.3. Foreløpig kan du tenke på det som «det er trykk på ledningen».

Er det spenning på den tredje ledningen, leder bryteren. Er det ingen spenning, er den stengt.



Figur 6.1 Til venstre bryteren fra forrige kapittel, som noen må trykke inn. Til høyre den samme bryteren med en tredje ledning: spenningen på styreledningen avgjør om den leder. Da kan utgangen fra én port styre den neste.

Ingen bevegelige deler, ingen finger, ingen knapp: et signal på én ledning avgjør om strømmen får gå på en annen. Les den setningen en gang til. Hele kapitlet hviler på den, og den er ikke åpenbar første gang man leser den.

Nå kan du koble utgangen fra én slik bryter til styreledningen på den neste, og da bygger kretsen seg videre av seg selv, uten at noen står ved siden av og trykker.

6.2.3 Og en som gjør det motsatte

Det finnes to slags. Den vi nettopp beskrev, leder når det er spenning på styreledningen. Den andre gjør det motsatte: spenning på styreledningen *stenger* den, og uten spenning leder den.

Det kan høres ut som en kuriositet, men kapitlet henger på at vi har begge to. Hvorfor, ser du i seksjon 6.4.

To ord vi holder oss unna

I elektronikk sier man gjerne at en bryter er *åpen* når den *ikke* slipper gjennom. Det er stikk motsatt av en åpen kran, og siden vi skal bruke vann som bilde hele veien, er det en felle.
Vi sier derfor **leder** og **stengt** om transistorer i denne boka, aldri åpen og lukket. Møter du «åpen bryter» i en annen bok, betyr det det samme som «stengt» her.

Kapittel 5 hadde allerede et tegn for «denne gjør det motsatte»: den lille sirkelen, boblen, på et portsymbol. Den bruker vi igjen. En bryter med boble på styreledningen er den som gjør det motsatte.

6.2.4 Den heter transistor

Nå kan den få et navn. En **transistor** er en bryter som styres av et elektrisk signal i stedet for av en finger. Den er laget av silisium, har ingen bevegelige deler og er så liten at det er plass til noen titalls milliarder av dem i en telefon.

De to typene heter **nMOS** og **pMOS**. nMOS-transistoren leder når styresignalet er 1. pMOS-transistoren leder når styresignalet er 0, og det er den med boblen.

Navnene, for ordens skyld

MOS står for *metal-oxide-semiconductor* og beskriver hvordan transistoren er bygget opp i lag. Bokstavene n og p viser til to måter å behandle silisiumet på, slik at det leder strøm på hver sin måte.
Du trenger ikke kunne noe av dette. Det du må huske, er at den ene leder ved 1 og den andre ved 0. Vil du vite hva som skjer inne i silisiumet, står det i appendiks D.

En transistor har tre tilkoblinger: styreledningen og de to som strømmen eventuelt går mellom. I engelsk faglitteratur heter styreledningen *gate*, og de to andre heter *source* og *drain*. Det kan være greit å ha sett ordene, siden de står i alt du kommer til å lese videre. I denne boka kaller vi dem styreledningen og de to kontaktene, og lar det bli med det.

6.3 Strøm, spenning og jord

6.3.1 Hvorfor en port trenger strøm

Før vi kan bygge noe av bryteren, må vi vite hva det er som går gjennom den. Og vi kan begynne med et spørsmål som er lurt å stille høyt: hvorfor må en datamaskin ha strøm?

Det virker selvsagt, men det er det ikke. En bok trenger ingen strøm for å lagre tekst, og et ferdiglagt puslespill trenger ingen strøm for å bli liggende. Hvorfor kan ikke maskinen bare stå der med nullene og ettallene sine?

Fordi et bit i en datamaskin ikke er en ting. Det er en *tilstand*, en spenning som må holdes oppe et sted i kretsen, og slipper du den, faller den. Dessuten gjør en port mer enn å holde en verdi. Den lager et nytt signal som er sterkt nok til å styre neste port, og det koster energi, på samme måte som det koster energi å løfte noe opp.

6.3.2 Spenning er høyde, strøm er mengde

Vi bruker vann hele veien her, og da trenger vi to ord.

Spenning er høydeforskjellen. Står vannet høyt, er det trykk og vannet vil nedover, mens det ikke skjer noe når det står lavt. Spenning måles i *volt*.

Strøm er hvor mye vann som faktisk renner, og det avhenger både av høydeforskjellen og av hvor mye krana er skrudd opp.

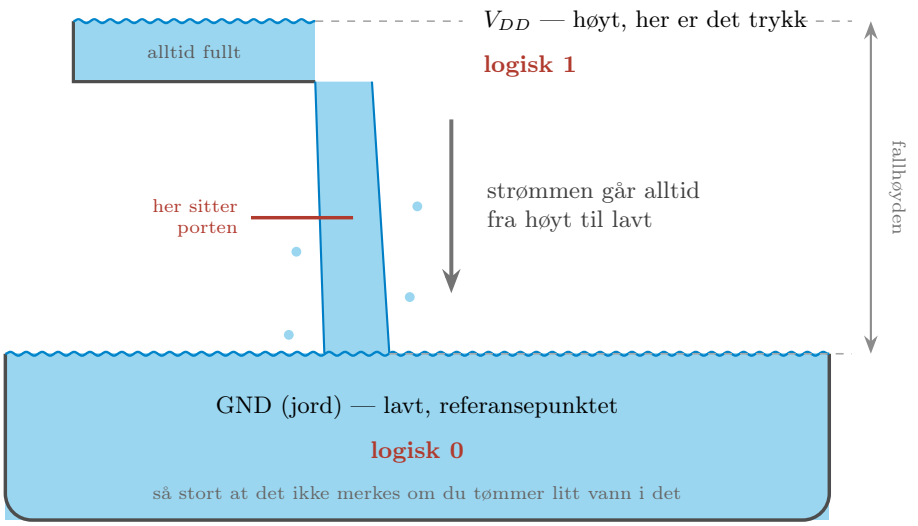
De to er forskjellige ting. Du kan ha stor høydeforskjell og likevel ingen strøm hvis krana er stengt, og akkurat den situasjonen skaper en stengt transistor.

6.3.3 Toppen og bunnen

En krets har et høyt punkt og et lavt punkt, og alt annet ligger et sted imellom.

Det høye punktet heter **VDD**. Det er forsyningsspenningen, stedet der strømmen kommer inn i kretsen. Navnet kommer av *drain supply voltage*, spenningen som mater transistorene, men i denne boka er det bare et navn på toppen. I en moderne prosessor ligger VDD rundt 0,8 volt. Av og til sier vi «1 volt» for å slippe desimaler.

Det lave punktet heter **GND**, kort for *ground*, altså jord. Det er 0 volt og referansepunktet alt annet måles mot.



Figur 6.2 V_{DD} og GND som toppen og bunnen av et fall. Strømmen går alltid nedover. Det nederste bassenget er så stort at nivået ikke rikker seg om du tømmer litt i det, og derfor kan jord brukes som en stabil nullstrek.

VDD er logisk 1, og GND er logisk 0. Mer er det ikke i oversettelsen mellom elektrisitet og logikk. Les den en gang til, for det er denne koblingen som gjør at alt du lærte i kapittel 5 faktisk kan sitte i en brikke.

Er «jord» virkelig jorda?

Spørsmålet kommer hvert år, og det er et godt et.
Svaret er: opprinnelig ja, og nå nesten. I et hus går jordlederen i stikkontakten faktisk ned i bakken. I en telefon på batteri finnes det ingen slik ledning, og «jord» er bare den ene polen på batteriet, som alt annet måles fra.
I begge tilfeller er poenget det samme. Jord er et sted som er så stort at det ikke merkes om du heller litt ladning inn i det, på samme måte som havet står like høyt etter at du har tømt et badekar i Atlanterhavet. Derfor kan jord fungere som en stabil nullstrek, og en stabil nullstrek trenger du for at «0 volt» skal bety det samme over hele brikken.

6.3.4 To tall, og ingenmannsland imellom

Én ting til før vi bygger noe. Strømmen går alltid fra VDD mot GND, aldri andre veien, så enhver ledning i kretsen har en spenning et sted mellom 0 og 0,8 volt. I prinsippet kan den ha hvilken som helst verdi i det intervallet. Logikken kjenner derimot bare to verdier. Hvordan går det i hop?

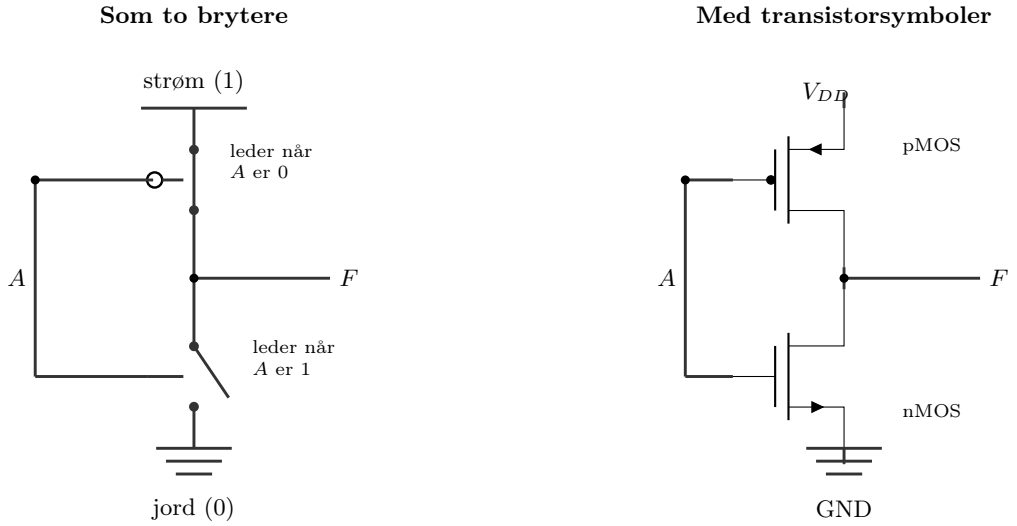
Det går i hop ved at 0 og 1 får hvert sitt *område* å være innenfor, i stedet for hvert sitt tall. Vi kommer tilbake til det i seksjon 6.6, når vi har noe konkret å vise det på. Foreløpig holder det å vite at lavt betyr 0 og høyt betyr 1.

6.4 Inverteren: den første porten vi faktisk bygger

6.4.1 To brytere i serie

Nå har vi det vi trenger, og vi kan bygge den enkleste porten som finnes: NOT-porten, som snur signalet.

Oppskriften er kort: sett de to transistortypene over hverandre mellom VDD og GND, og ta utgangen ut midt imellom. Begge styres av det samme inngangssignalet.



Figur 6.3 Inverteren tegnet på to måter. Til venstre som to brytere, slik vi tegner den i resten av kapitlet. Til høyre med symbolene du møter i faglitteraturen. Det er samme krets.

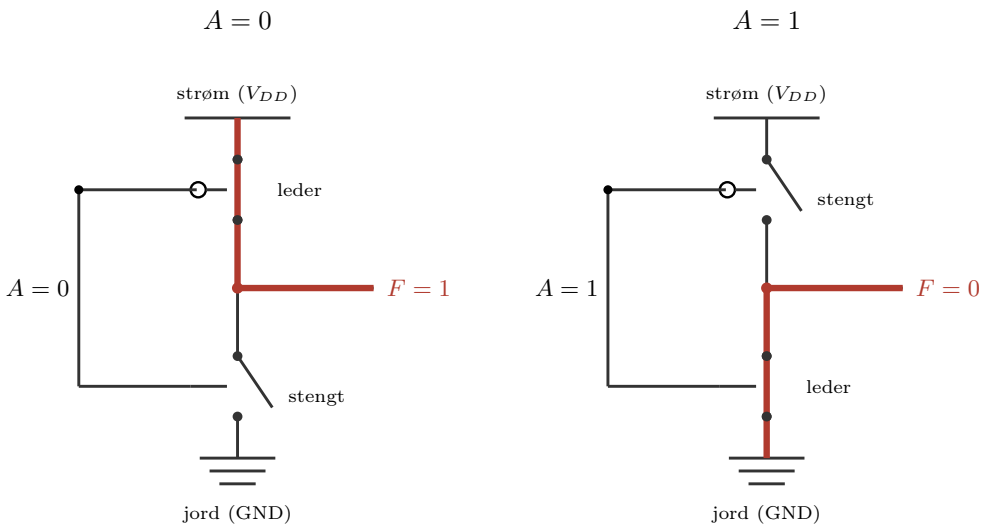
Den øverste er pMOS-en, den med boblen, som leder når inngangen er 0. Den nederste er nMOS-en, som leder når inngangen er 1.

6.4.2 Begge tilfellene, i ro og mak

Det finnes bare to muligheter, så vi går gjennom begge.

Inngangen er 0. Da leder pMOS-en øverst, fordi den leder ved 0, mens nMOS-en nederst er stengt, fordi den trenger 1 for å lede. Utgangen har fri vei opp til VDD og ingen vei ned til GND. Den blir høy, altså 1.

Inngangen er 1. Nå er det motsatt. pMOS-en er stengt, nMOS-en leder, og utgangen har fri vei ned til GND og ingen vei opp. Den blir 0.



Ettallet kommer fra forsyningen, ikke fra inngangen.
Inngangen velger bare hvilken av de to veiene som åpnes.

Figur 6.4 Inverteren i begge tilstandene. Den røde streken viser hvor verdien på utgangen kommer fra. Inngangen rører aldri utgangen, den åpner bare én av de to veiene.

Inn med 0, ut med 1. Inn med 1, ut med 0. Det er en NOT-port bygget av to brytere.

6.4.3 Det som er lett å gå glipp av

Les de to tilfellene en gang til, og se etter hvor ettallet kom fra.

Det kom ikke fra inngangen. Inngangen rørte aldri utgangen, og på tegningen ser du at det ikke går noen ledning mellom dem. Inngangen gikk bare til de to styreledningene. Det den gjorde, var å velge *hvilken av de to veiene som skulle åpnes*, og verdien på utgangen kom fra VDD eller fra GND, altså fra forsyningen.

Hvor nullene og ettallene kommer fra

Hver eneste ener i maskinen din er hentet rett fra strømforsyningen, og hver eneste null er hentet fra jord. Signalet som kommer inn i en port, bærer ikke verdien videre. Det åpner bare en kran. Verdien som renner ut er ny, og like sterk uansett hvor slapp inngangen var.
Det er derfor du kan koble tusen porter etter hverandre uten at signalet blir utvannet. Vi kommer tilbake til det i seksjon 6.6, og det er viktigere enn det ser ut nå.

6.4.4 Aldri begge samtidig

Én ting til i figuren: så lenge inngangen ligger i ro, leder de to transistorene aldri samtidig. Inngangen er enten 0 eller 1, de to typene reagerer motsatt på den, og leder den ene, er den andre stengt. Det er bra. Hvis begge ledet samtidig, ville det gått en rak strøm fra VDD til GND uten å gjøre noe nyttig, og det kalles en kortslutning.

Det gir en følge som er lett å overse. Når kretsen står i ro, går det nesten ingen strøm gjennom den. Strømmen går bare i det korte øyeblikket utgangen skifter.

Med ett lite unntak

I selve omslaget er begge transistorene så vidt ledende samtidig, i de få pikosekundene inngangen er på vei fra 0 til 1 eller omvendt. Da renner det en liten, rak strøm fra forsyningen til jord. Den varer kort, men den kommer hver eneste gang, og er en av grunnene til at det koster å skifte. Appendiks D går nærmere inn på det.

Fun Fact

Det er denne egenskapen som gjør at en telefon kan ligge på bordet i en uke og fortsatt ha batteri. Milliarder av transistorer holder på verdiene sine, og nesten ingen av dem trekker strøm mens de gjør det.
Egenskapen har også gitt teknologien navnet sitt. De to transistortypene utfyller hverandre, og *complementary* betyr nettopp det. Derav **CMOS**, complementary MOS, teknologien praktisk talt all digital elektronikk er bygget i.

Husk ordet «nesten». Vi kommer tilbake til det i seksjon 6.10, og da skjønner du hvor mye det betyr.

6.5 NAND, NOR, og hvorfor AND er dyrere

6.5.1 To i serie, eller to i parallell

Inverteren hadde én inngang og trengte derfor én transistor oppe og én nede. Med to innganger trenger vi to oppe og to nede, til sammen fire. Spørsmålet er hvordan de skal kobles, og det finnes bare to muligheter: etter hverandre eller ved siden av hverandre. Du kjenner dem fra kapittel 5, der det var lysbrytere i en stue.

To brytere *i serie*: strømmen må gjennom begge, så begge må lede. To brytere *i parallell*: strømmen har to veier å velge mellom, så det holder at én leder.

Se på NAND-porten til venstre. Nede står de to nMOS-ene i serie, så veien ned til jord åpner seg bare når begge inngangene er 1. Oppe står de to pMOS-ene i parallell, og siden de leder ved 0, holder det at *én* av inngangene er 0 for at utgangen kobles til VDD. Utgangen er altså 0 bare når begge inngangene er 1, og det er sannhetsverditabellen til NAND.

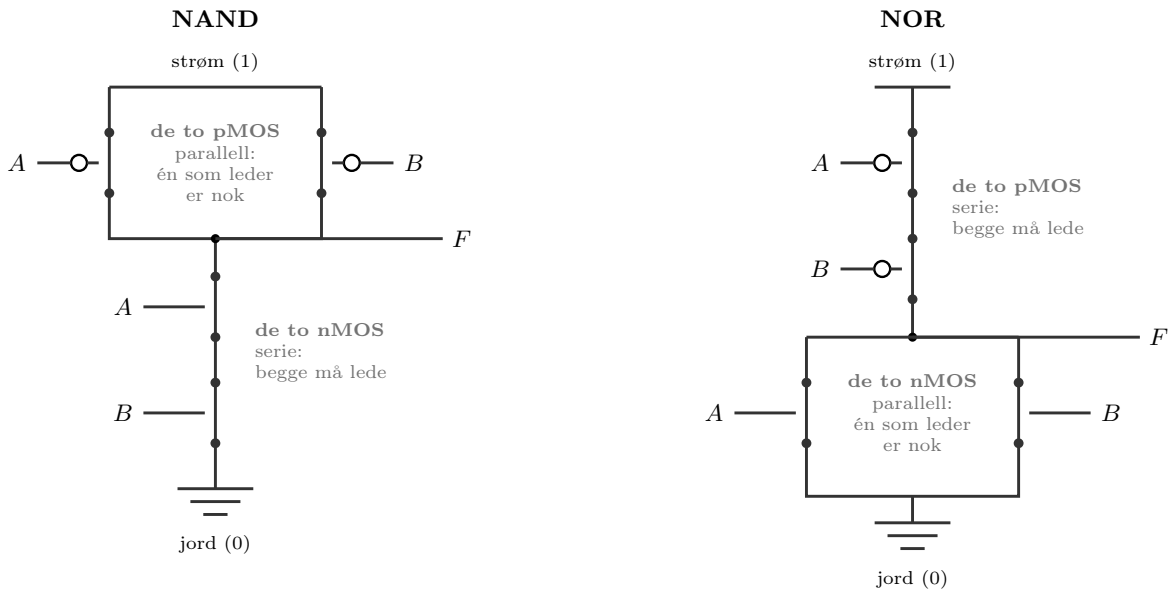
La oss sjekke ett tilfelle, slik vi gjorde med inverteren. Sett $A = 1$ og $B = 0$. Nede leder den ene nMOS-en (A er 1) og den andre er stengt (B er 0). De står i serie, så veien til jord er brutt. Oppe leder pMOS-en som styres av B, siden B er 0, mens den andre er stengt. De står i parallell, så den ene som leder er nok, og utgangen kobles til VDD.

Utgangen blir 1, og det stemmer: NAND skal gi 1 i alt annet enn tilfellet der begge er 1. Prøv de tre andre radene selv. Det tar to minutter, og det er den beste måten å se at det henger sammen.

NOR-porten til høyre er samme idé speilvendt: serie oppe, parallell nede.

Styreledningene er ikke tegnet

Figuren viser hvilken bryter som styres av A og hvilken som styres av B, men den tegner ikke ledningene som går dit. De ville krysset hverandre tre ganger og gjort figuren uleselig.
Det er samme slags forenkling som returledningen vi utelot i kapittel 5. Ledningene finnes, de er bare ikke det figuren handler om.



Figur 6.5 NAND og NOR, fire transistorer hver. Forskjellen ligger i hvem som står i serie og hvem som står i parallell. Styreledningene er utelatt for å holde figuren lesbar, slik returledningen var i forrige kapittel.

6.5.2 Hvert trinn snur signalet

Da kan du spørre: hvor er AND-porten?

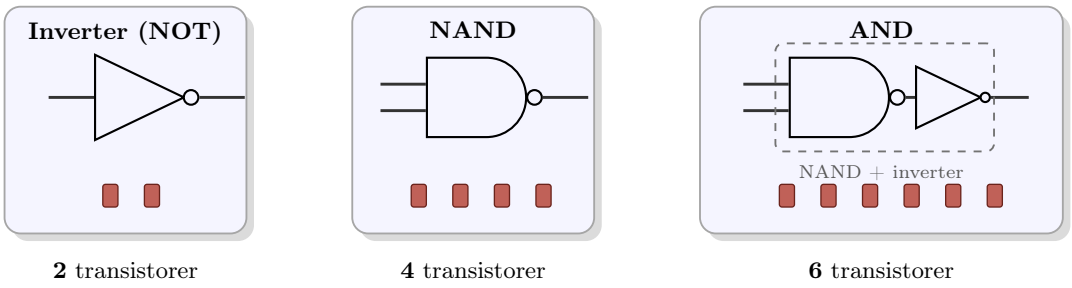
Prøv å bygge den. Setter du de to nMOS-ene i serie nede, trekkes utgangen til *jord* når begge er 1, og du får 0 ut når begge er 1. Det er NAND, ikke AND. Hvordan du enn kobler dem, blir det slik. Transistorene oppe kobler utgangen til VDD, transistorene nede kobler den til GND, og de to gruppene gjør alltid det motsatte av hverandre. Et slikt trinn snur derfor alltid signalet.

Dette betyr ikke at AND-porten ikke finnes

AND-porten finnes. Den ligger i alle kretsbiblioteker, og du tegner den som ett symbol.
Det som er sant, er at hvert enkelt *trinn* av denne typen snur signalet. En AND-port er derfor bygget som en NAND-port med en inverter etter seg, to trinn der det andre snur tilbake. Porten er like virkelig som før. Den er bare satt sammen av mer enn du trodde.

6.5.3 Regnskapet

Nå kan vi telle, og da blir tallene interessante.



Figur 6.6 Hva portene koster. AND er en NAND med en inverter etter seg, og koster derfor halvannen gang så mye som NAND.

En inverter koster to transistorer, en NAND fire og en AND seks, fordi den er en NAND pluss en inverter.

Det er motsatt av det de fleste gjetter. AND føles som den enkle porten og NAND som den rare med boble på. I silisium er det omvendt: NAND er den enkle, og AND er en NAND med et ekstra trinn.

Port	Transistorer	Hvordan
NOT	2	étt trinn
NAND	4	étt trinn
NOR	4	étt trinn
AND	6	NAND + inverter
OR	6	NOR + inverter
XOR	8–12	flere trinn, avhengig av oppskrift

Tabell 6.1 Hva portene koster i transistorer. Tallene gjelder porter med to innganger.

Kapittel 5 sa at det lønner seg å spare porter, og brukte mye plass på å forenkle uttrykk. Nå ser du hva det betales i. Hver port du sparer, er et håndfast antall transistorer som slipper å ta plass, trekke strøm og bli varme. Og nå forstår du hvorfor en kretsdesigner heller skriver om et uttrykk til NAND-er enn til AND-er. Det er ingen vane. Det er to transistorer per port.

Regn på det selv

En krets bruker tolv AND-porter med to innganger. En designer finner en måte å få samme resultat med tolv NAND-porter i stedet.
Tolv ganger seks mot tolv ganger fire: du sparte 24 transistorer av 72, altså en tredel. Gjør det samme i en krets med noen hundre millioner porter, og du har spart noe som merkes på strømregningen til et datasenter.

6.6 Null og ett er områder, ikke tall

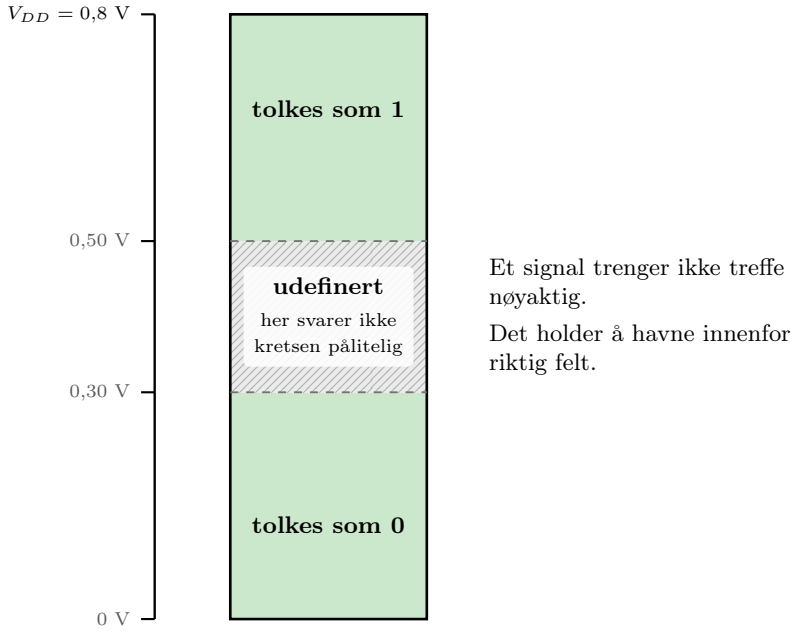
6.6.1 Verden er analog, og det er et problem

Vi lovet å komme tilbake til dette, og nå har vi noe konkret å henge det på.

En ledning fører en spenning som kan være hva som helst mellom 0 og 0,8 volt. Den kan være 0,37 volt, den kan dirre litt opp og ned, og den blir dratt på av naboledninger, av varme og av at noe annet på brikken skifter i samme øyeblikk. Logikken kjenner derimot bare 0 og 1. Hvordan kan et system med to verdier bygges av noe som kan ha uendelig mange?

6.6.2 Svaret: gi hvert tall et rom å bo i

Løsningen er enkel, og hele den digitale teknologien hviler på den. I stedet for å kreve at 0 betyr nøyaktig 0,000 volt, sier vi at alt under en viss grense *teller som* 0, og at alt over en annen grense teller som 1. Mellom de to grensene ligger et belte der kretsen ikke lover noe.



Figur 6.7 0 og 1 er områder, ikke tall. Havner et signal i det skraverte beltet, har kretsen ikke lovet noe om hva den gjør med det.

Signalet trenger altså ikke treffe presist. Det holder at det havner i riktig felt. En ener på 0,74 volt og en ener på 0,79 volt er den samme eneren.

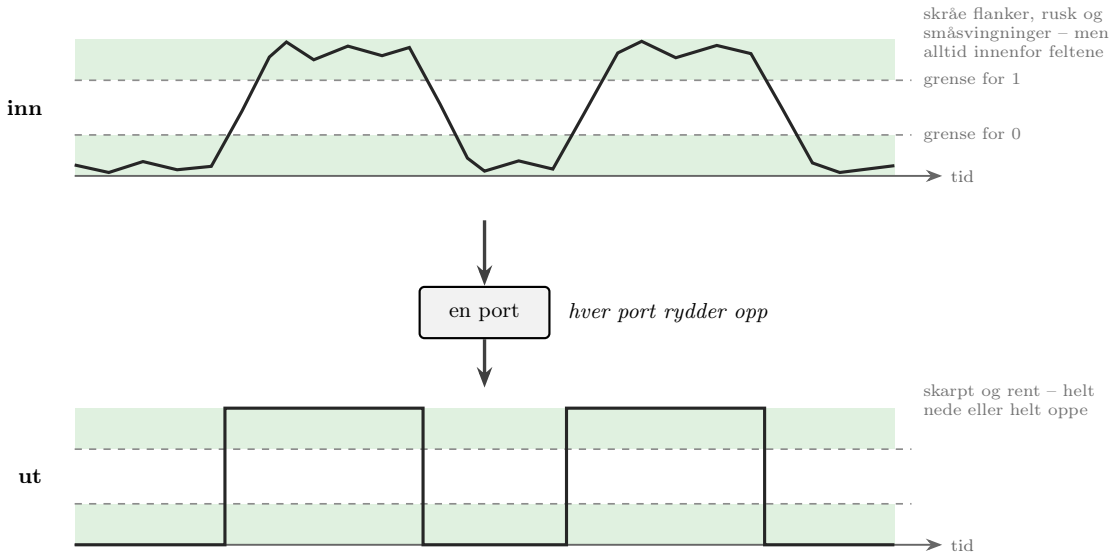
Hvorfor det er et belte i midten

Du kan spørre hvorfor vi ikke bare setter én grense midt på, slik at alt under er 0 og alt over er 1. Problemet er at en krets som får et signal helt på grensa, må velge, og den kan velge ulikt fra gang til gang. Med et bredt ingenmannsland i midten havner ingen der ved et uhell.
Beltet er altså med vilje, og bredden på det er en designbeslutning. Det tekniske navnet er **støymargin**: hvor mye juks signalet tåler før det blir misforstått. Tallene bak står i appendiks D.

6.6.3 Hver port vasker signalet rent

Nå følger noe som gjør digital teknologi mulig i praksis, og det henger sammen med det vi så på inverteren.

Når et slapt, forvrengt signal kommer inn i en port, bærer ikke porten det videre. Den *leser* det, avgjør om det er over eller under grensa, og lager et helt nytt signal fra VDD eller GND. Utgangen er skarp, selv om inngangen var slurvete.



Figur 6.8 Et forvrengt signal inn, et skarpt signal ut. Porten bærer ikke signalet videre, men leser det og lager et nytt fra forsyningen.

Signalet blir vasket rent i hvert eneste ledd. Rusk som samler seg underveis, forsvinner med en gang, og regnskapet starter på nytt etter hver port. Fagordet for dette er **signalgjenoppretting**.

6.6.4 Derfor overlever en kopi av en kopi

Du har kanskje lagt merke til følgen av dette uten å tenke over det. Kopierer du en lydfil og kopierer kopien, og gjør det tusen ganger, er den siste fila bit for bit lik den første. Hvert eneste bit ble lest som enten over eller under grensa, og skrevet på nytt like skarpt. En kassett kopiert tusen ganger blir bare susing, fordi hver liten unøyaktighet fulgte med videre og la seg oppå den forrige.

Forskjellen er ikke at digitalt er mer nøyaktig. Digitalt *retter seg selv* hele tiden, fordi hver port kaster alt som ikke er en ren 0 eller 1.

Tankevekker

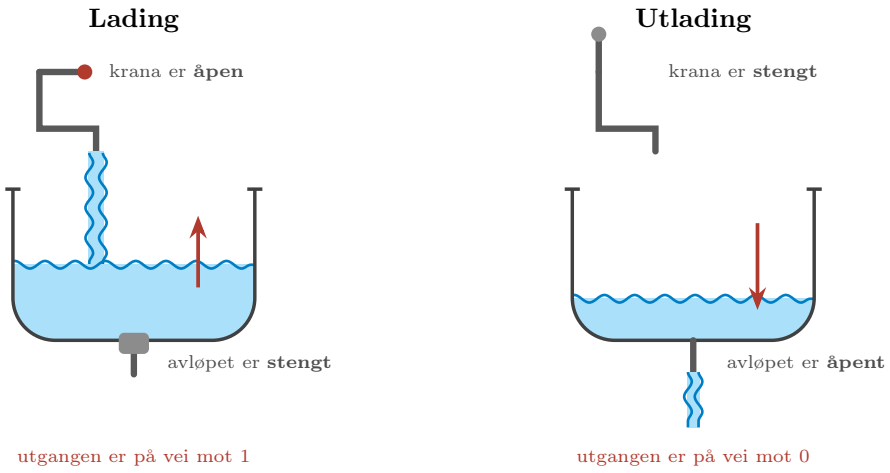
Kapittel 1 sa at en datamaskin er lag på lag med abstraksjoner, og at abstraksjoner lekker. Her står du ved den nederste, og den lekker ikke. «Et bit er en 0 eller en 1» er en påstand om noe fysisk, og den holder fordi hver port tvinger signalet tilbake i form, noen milliarder ganger i sekundet. Resten av boka hviler på dette. Kunne en ener bli litt slapp underveis, ville ingenting høyere opp fungert.

6.7 Det tar tid

6.7.1 Badekaret

Til nå har vi beskrevet alt som om det skjer momentant: bryteren begynner å lede, og utgangen er 1. Så enkelt er det ikke, og forskjellen er det resten av kapitlet handler om.

Tenk deg utgangen på en port som et badekar. Skal porten sette utgangen til 1, åpner den krana øverst og fyller karet. Skal den sette utgangen til 0, stenger den krana og åpner avløpet.



Jo større kran, jo raskere fylles karet. Jo større kar, jo lengre tid tar det.

Figur 6.9 Utgangen på en port, tegnet som et badekar. Å sette utgangen til 1 er å fylle karet, å sette den til 0 er å tømme det. Begge deler tar tid.

Det tar tid å fylle et badekar, og det tar tid å tømme det. Hvor lang tid, avhenger av to ting: hvor stort karet er, og hvor stor krana er.

6.7.2 Hvorfor er det et badekar der?

Her må vi være presise, for det er lett å misforstå. Ingen har *satt inn* et badekar, det oppstår av seg selv.

En ledning som skal føre en spenning, må lades opp til den spenningen, på samme måte som et kar må fylles før vannstanden er høy. Det samme gjelder styreledningen inn til neste port, siden det er spenningen på den som avgjør om transistoren leder. Summen av alt dette, ledningen og styreledningene til portene den går til, er badekaret. Fagordet er **kapasitans**, og det betyr omtrent «hvor mye som må fylles på».

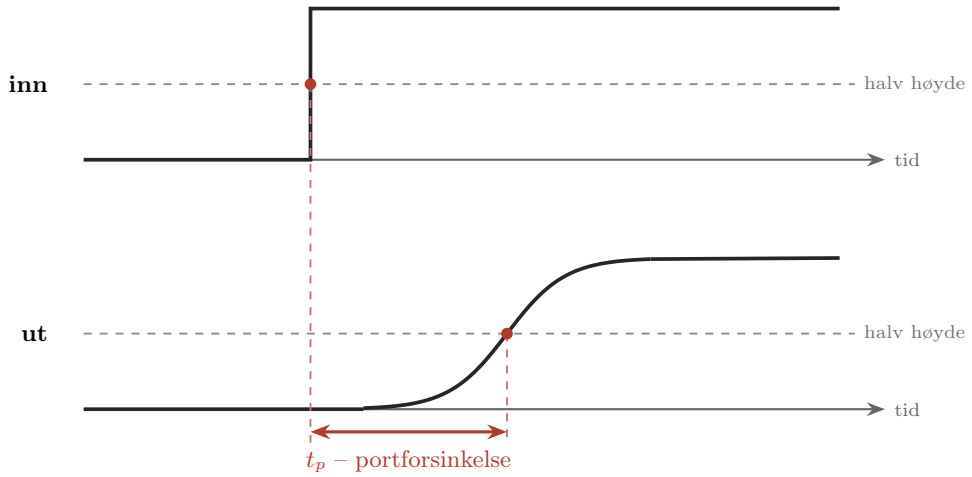
En vanlig misforståelse

Kapasitansen er ikke en komponent noen har plassert der fordi den trengs. Den er en uunngåelig følge av at ledninger og transistorer er fysiske ting som tar plass. Ingen designer er glad i den, og alle jobber for å få den ned, fordi mindre kar gir raskere kretser.
Merk også at denne lille mengden lagret ladning *ikke* er minne. Den hører ikke til minnehierarkiet

du møter i kapittel 7. Den er bare den uunngåelige tregheten i at et signal må bygge seg opp.

6.7.3 Portforsinkelse

Nå kan vi gi tiden et navn. **Portforsinkelse** er tiden fra inngangen til en port endrer seg til utgangen har kommet over på den nye verdien.



Utgangen begynner først å røre seg litt etter inngangen, og bruker så tid på å komme helt opp.

Figur 6.10 Portforsinkelse. Utgangen begynner først å røre seg litt etter inngangen, og bruker så tid på å komme helt opp.

Portforsinkelsen er ikke lik for alle porter. En XOR har flere trinn inni seg enn en NAND, og bruker derfor lengre tid. Vi regner likevel med *samme* tall for alle porter i dette kapitlet, siden poenget er hvordan man teller og ikke hvor presis summen blir.

I en moderne prosess ligger portforsinkelsen på noen få *pikosekunder*, og et pikosekund er en billiondel av et sekund. Vi regner med 5 pikosekunder per port i resten av kapitlet. Tallet er rundt for at regningen skal bli grei, og det ligger i riktig område.

Størrelsesordenene, så du har dem

Enhet	Hvor lenge	Hva som skjer der
pikosekund (ps)	10^{-12} s	én port skifter
nanosekund (ns)	10^{-9} s	en instruksjon kjøres, eller et tall hentes fra arbeidsminnet
mikrosekund (μs)	10^{-6} s	en lesing fra en SSD
millisekund (ms)	10^{-3} s	en lesing fra en treg disk

Tabell 6.2 Hvert trinn er tusen ganger lengre enn det forrige.

Avstanden fra et pikosekund til et sekund er omtrent like stor som fra ett sekund til tretti tusen år.

6.7.4 Større kran, raskere port, og en regning

Hvorfor gjør man ikke bare alle transistorene større, slik at de fyller karet fortere? Det gjør man faktisk, der det haster. En bredere transistor leverer mer strøm og skifter raskere, på samme måte som en bredere kran fyller badekaret fortere.

Men den koster. Den tar mer plass, den trekker mer strøm, og, her kommer den kjipe delen, den er selv et større kar for porten som står foran. Gjør du én port raskere, gjør du porten før den tregere.

Dette er den første av flere avveininger i kapitlet der du ikke kan få alt. Fart, plass og strøm drar hver sin vei, og en kretsdesigner velger hele dagen.

6.8 Den lengste veien

6.8.1 Portene står ikke i kø

Kapittel 5 bygde en fulladder av fem porter. Hvis hver port tar 5 pikosekunder, hvor lang tid bruker fulladderen?

Det fristende svaret er fem ganger fem, altså 25. Det er feil, og grunnen er viktig. Portene jobber nemlig ikke etter tur. De jobber alle sammen hele tiden, og en port begynner i samme øyeblikk inngangene er klare, uten å bry seg om hva de andre holder på med. Tiden gjennom kretsen er derfor ikke summen av alle portene. Den er lengden på den *lengste veien* fra en inngang til en utgang.

6.8.2 Fulladderen, målt

La oss se på det i kretsen du allerede kjenner.

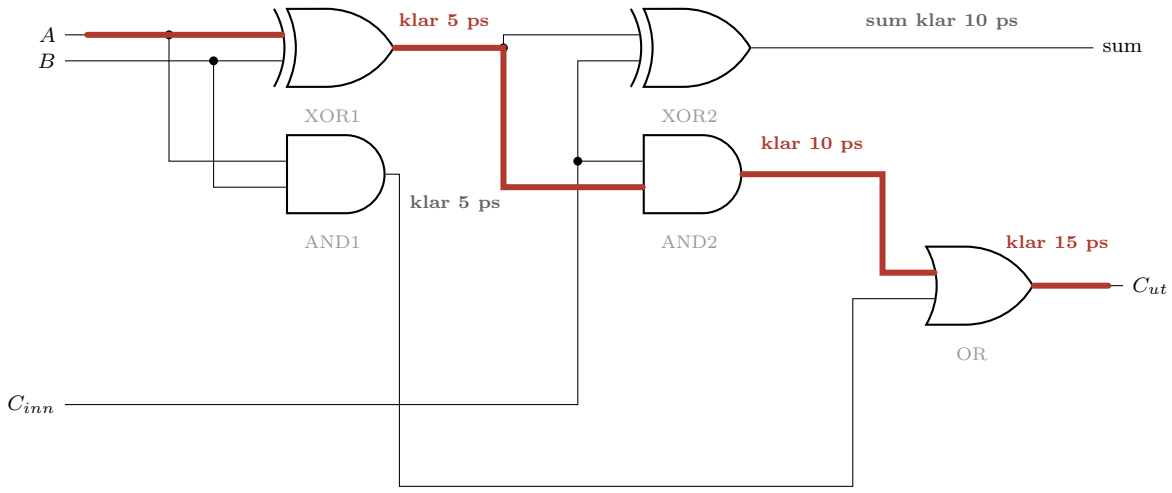
Følg den røde veien. Signalet går først gjennom XOR1, som er ferdig etter 5 pikosekunder, så gjennom AND2, ferdig etter 10, og så gjennom OR-porten, ferdig etter 15. Tre porter på rad gir 15 pikosekunder.

AND1 var ferdig allerede etter 5, men det hjalp ikke. OR-porten har to innganger og kan ikke svare før begge er på plass, så den raske inngangen må stå og vente på den trege.

Denne lengste veien kalles den **kritiske veien**. Det er den som bestemmer hvor rask kretsen er, og den eneste en designer bryr seg om å gjøre kortere. Å gjøre noe på en av de andre veiene raskere, endrer ingenting. Den samme tankegangen kommer du til å bruke når du senere skal få et program til å gå raskere: finn det som faktisk bestemmer tiden, og la resten ligge.

Et bilde fra en helt annen verden

Fire personer lager middag sammen. Potetene tar 20 minutter, kjøttet 15, sausen 10 og salaten 4, og alle jobber samtidig, hver med sitt.
Middagen er klar etter 20 minutter, ikke etter 49. Og skal dere spise tidligere, hjelper det ikke å hakke salaten raskere. Potetene er den kritiske veien.



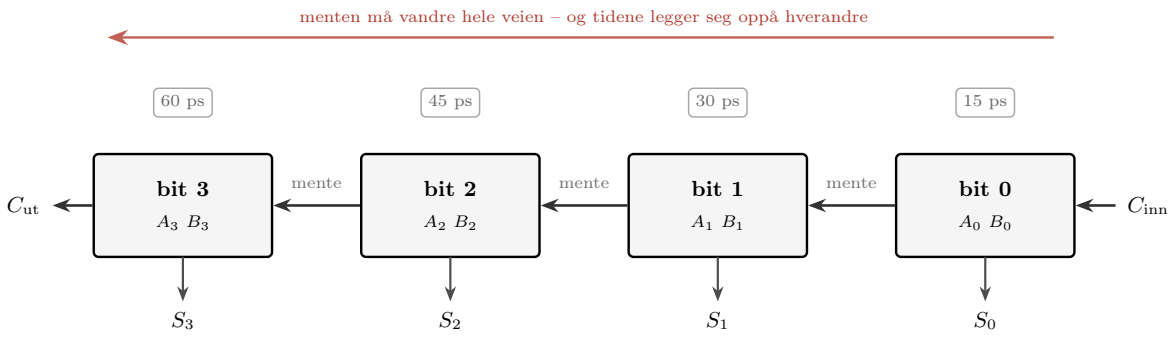
Den lengste veien: XOR1 → AND2 → OR. Tre porter, 15 ps.

OR-porten har to innganger. Den ene er klar etter 5 ps, den andre først etter 10. Porten kan ikke svare før begge er på plass, så den starter etter 10 og er ferdig etter 15. Veiene går samtidig — tidene legges ikke sammen, det er den lengste som bestemmer.

Figur 6.11 Den kritiske veien gjennom fulladderen fra kapittel 5. Kretsen har fem porter, men bare tre av dem ligger etter hverandre.

6.8.3 Og så legger vi dem i rekke

Kapittel 5 sa at mentekjeden koster tid, uten å si hvor mye. Nå kan vi regne.



64 bit: 64×15 ps $= 960$ ps, altså omtrent 1 GHz.

Figur 6.12 Mentekjeden. Her hjelper det ikke at portene jobber samtidig: hvert ledd må vente på menten fra det forrige, så tidene legger seg oppå hverandre.

I en mentekjede må menten ut av den første fulladderen før den andre kan bli ferdig, og den andre må bli ferdig før den tredje. Ingenting går i parallell. Det er en kø, der hvert ledd venter på det forrige. Fire bit koster dermed fire ganger 15, altså 60 pikosekunder.

Hva med en kjede på 64 bit, som er det en vanlig prosessor regner med?

$$64 \times 15 \text{ ps} = 960 \text{ ps}$$

Det er nesten et helt nanosekund for én addisjon. Deler vi ett sekund på 960 pikosekunder, får vi plass til omtrent **en milliard** slike addisjoner i sekundet.

Nå ser du hvorfor kapittel 5 brukte plass på å klage over mentekjeden. En krets som gjør alt annet på 15 pikosekunder, bruker nesten tusen på å legge sammen to tall.

Overslaget er for pessimistisk. Finner du ut hvorfor?

Vi ganget 15 pikosekunder med 64. Men prøv å bruke det du lærte i forrige seksjon på ledd nummer to i kjeden.
Der er A og B klare fra første stund, og det eneste som kommer sent, er menten fra naboen. Menten trenger ikke gjennom hele fulladderen, bare gjennom AND2 og OR, altså to porter og 10 pikosekunder. Det er bare det *første* leddet som koster 15. De 63 andre koster 10 hver:

$$15 + 63 \times 10 = 645 \text{ ps}$$

Det er fortsatt tregt, fortsatt den trege delen av maskinen, og tiden vokser fortsatt rett opp med antall bit. Men eksempelet viser at den kritiske veien ikke alltid er den du ser først, og at det lønner seg å spørre hvilket signal som faktisk kommer sist.

Hvorfor maskiner likevel er raskere enn 1 GHz

Ekte prosessorer bruker ikke en rett mentekjede på 64 bit. De bruker kretser som gjetter menten på forhånd i stedet for å vente på den, og slipper dermed køen.
Den vanligste heter *carry look-ahead*. Den er mye raskere og bruker betydelig flere transistorer, så den er enda et eksempel på samme avveining: fart mot plass og strøm. Du trenger ikke vite hvordan den virker, men du bør vite at den finnes, og hvorfor noen gadd å finne den opp.

6.9 Klokka

6.9.1 Problemet som må løses

Ulike veier gjennom en krets blir ferdige til ulik tid. Noen svar er klare etter 5 pikosekunder, andre først etter 960. I mellomtiden står det tall på ledningene som ikke er ferdig regnet ut, halvferdige verdier som skifter mens signalene arbeider seg gjennom. Hvis noe leser av en ledning i feil øyeblikk, leser det søppel.

Maskinen trenger derfor en avtale om *når* det er trygt å se. Den avtalen er klokka.

En forklaring du kan møte, som ikke stemmer

Noen ganger forklares klokka med at verdiene lekker ut hvis man venter for lenge, så man må rekke å lagre dem før de forsvinner.
Det stemmer ikke for vanlige logikkretser. Så lenge inngangen ligger fast, holder porten utgangen aktivt oppe, fordi én av transistorene leder og kobler den til VDD eller GND. Verdien renner ikke ut.
Grunnen til at vi trenger klokka er en annen: ulike veier blir ferdige til ulik tid, og noen må si fra når alle er det.

6.9.2 Alle leverer samtidig

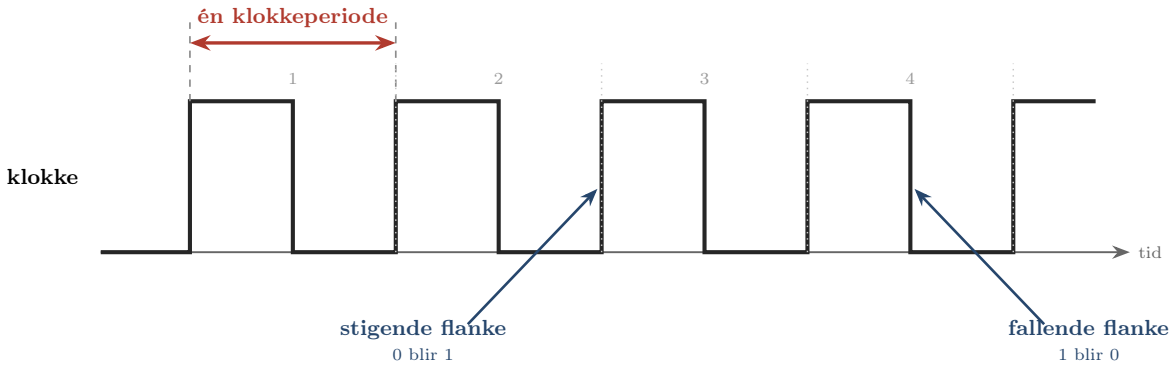
Tenk på en skriftlig prøve. Alle begynner når det ringer inn, alle jobber i sitt eget tempo, og alle leverer når det ringer ut. Den som ble ferdig tidlig, sitter og venter, for tiden settes etter den som trenger lengst.

Klokka i en datamaskin gjør det samme: den deler tiden i like lange biter, innenfor hver bit får alle kretsene jobbe fritt, og på slutten av biten lagres alle svarene samtidig.

Strengt tatt er det sløsing, for de raske kretsene står og venter mesteparten av tiden. Til gjengjeld slipper man å holde rede på hvem som blir ferdig når, og det er en enorm forenkling.

6.9.3 Hvordan et klokkesignal ser ut

Klokka er bare en ledning til, og den går til nesten alt på brikken. På den ledningen veksler spenningen mellom 0 og 1 i et helt jevnt mønster.



frekvens = 1 delt på klokkeperioden

Figur 6.13 Klokkesignalet. Dette er figuren det lønner seg å kunne utenat.

Tre ord hører til figuren, og de bør du kunne.

Klokkeperioden er tiden fra ett sted i mønsteret til samme sted neste gang. Du kan måle fra en stigende flanke til neste stigende flanke, eller fra en fallende til neste fallende. Det gir samme svar.

Stigende flanke er øyeblikket signalet går fra 0 til 1. **Fallende flanke** er det motsatte. Det er på flankene ting skjer: det er da resultatene lagres.

6.9.4 Fra periode til frekvens

Frekvensen er hvor mange perioder som går per sekund. Den måles i *hertz*, forkortet Hz, etter fysikeren Heinrich Hertz. En hertz er én periode i sekundet, og en *gigahertz* er en milliard av dem.

Sammenhengen mellom periode og frekvens får du mest bruk for i hele kapitlet:

$$\text{frekvens} = \frac{1}{\text{klokkeperioden}}$$

Det er samme slags sammenheng som mellom fart og tid. Kjører du strekningen på halvparten av tiden, kjørte du dobbelt så fort.

La oss sette inn et tall. En prosessor på 3 GHz gjør tre milliarder perioder i sekundet, og hver periode varer da:

$$\frac{1}{3 \cdot 10^9}\ \text{s} = 333\ \text{ps}$$

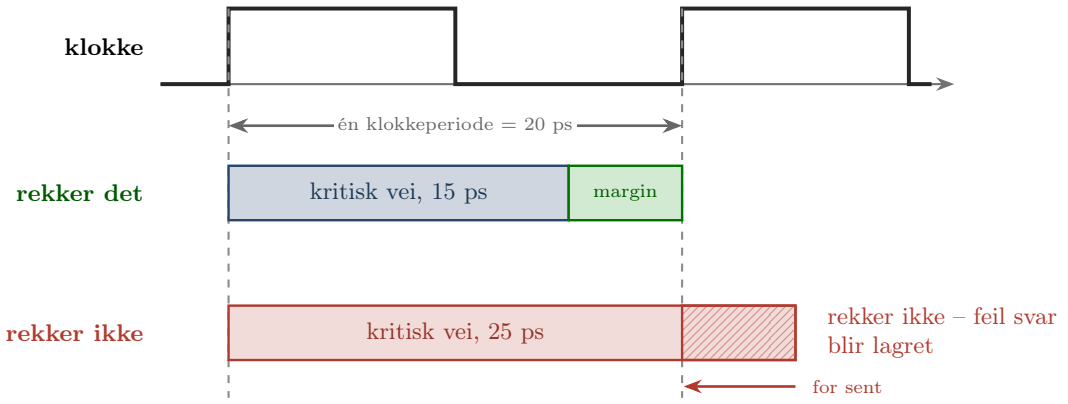
Det er en tredjedels nanosekund. I løpet av den tiden skal alt som skal bli ferdig, bli ferdig.

Regn den andre veien

En prosessor har en klokkeperiode på 500 pikosekunder. Hva er frekvensen?
$1/(500 \cdot 10^{-12}) = 2 \cdot 10^9$, altså 2 GHz.
Prøv også 250 ps, som gir 4 GHz, og 1 ns, som gir 1 GHz. Jo kortere periode, jo høyere frekvens. Det er samme tall snudd på hodet.

6.9.5 Hva som setter taket

Nå kan vi sette de to siste seksjonene sammen, og det gir en av de mest nyttige innsiktene i kapitlet. Alt som skal bli ferdig innen neste flanke, må rekke det. Klokkeperioden må derfor være minst like lang som den kritiske veien.



Figur 6.14 Den kritiske veien må få plass innenfor én klokkeperiode. Rekker den ikke fram, blir et halvferdig svar lagret, og maskinen regner feil.

Er perioden for kort, rekker ikke den trege veien fram. Da lagres et halvferdig svar, og maskinen regner feil. Ikke litt feil, men helt feil, og på en måte som er svært vanskelig å finne i etterkant.

Det er derfor man ikke bare kan skru opp klokkefrekvensen og få en raskere maskin. Over et visst punkt begynner den å regne galt.

Fun Fact

Dette er nettopp det folk gjør når de *overklokker* en maskin. De setter klokkeperioden kortere enn produsenten anbefaler og håper at marginen var romsligere enn oppgitt.
Ofte går det bra, fordi produsenten legger inn margin for varme dager og dårlige eksemplarer. Går det ikke bra, får man tilfeldige krasj. Det er ikke programvarefeil, men kretser som ikke rakk fram.

6.9.6 Hvorfor ikke 200 GHz?

Her er en regning som er fristende å gjøre, og som fører feil. En port tar 5 pikosekunder. Setter vi klokkeperioden til 5 pikosekunder, får vi en frekvens på $1/(5 \cdot 10^{-12})$, altså 200 GHz. Hvorfor går ikke prosessorer i 200 GHz?

Det er tre grunner, og alle tre er greie å ha med seg.

For det første er den kritiske veien aldri bare én port. Den er mange porter på rad, og vi har sett en på 64 ledd. Perioden må romme hele veien.

For det andre trenger en klokkeperiode mer enn selve regnetiden. Den trenger margin for at brikken er varm, for at akkurat dette eksemplaret er litt tregere enn gjennomsnittet, og for at klokkesignalet ikke kommer helt samtidig overalt.

Den tredje grunnen er den avgjørende: brikken ville brent opp. Det er temaet i neste seksjon.

Global og lokal klokke

En moderne brikke har ikke én klokke, men flere. Prosessorkjernene kan gå i én takt, minnegrensesnittet i en annen og grafikken i en tredje.
Det gir to fordeler. Deler som ikke trenger å være raske, slipper å bruke strøm på å være det, og deler som ikke er i bruk, kan få klokka slått helt av. Da står de stille og slutter å fylle badekar. De lekker fortsatt, som vi kommer tilbake til, men den største utgiften er borte.
Når noen oppgir at en maskin går i 3,5 GHz, er det den globale klokka til prosessorkjernene de snakker om.
Hvordan klokkesignalet fordeles til milliarder av transistorer, er et kapittel for seg. Det står i appendiks E og er ikke pensum.

6.10 Hvorfor klokka stoppet på rundt 3 GHz

6.10.1 Femti år med krymping

I 1965 skrev Gordon Moore, som senere var med på å grunnlegge Intel, en kort artikkel i bladet *Electronics*. Han hadde lagt merke til at antallet komponenter man fikk plass til på en brikke, hadde doblet seg hvert år siden teknologien kom, og han gjettet at det ville fortsette en stund.

Det fortsatte i over femti år. I 1975 justerte han anslaget til en dobling hvert annet år, og med den justeringen har observasjonen truffet forbløffende godt helt fram til vår tid. Den kalles **Moores lov**, og navnet er misvisende.

Det er ingen naturlov

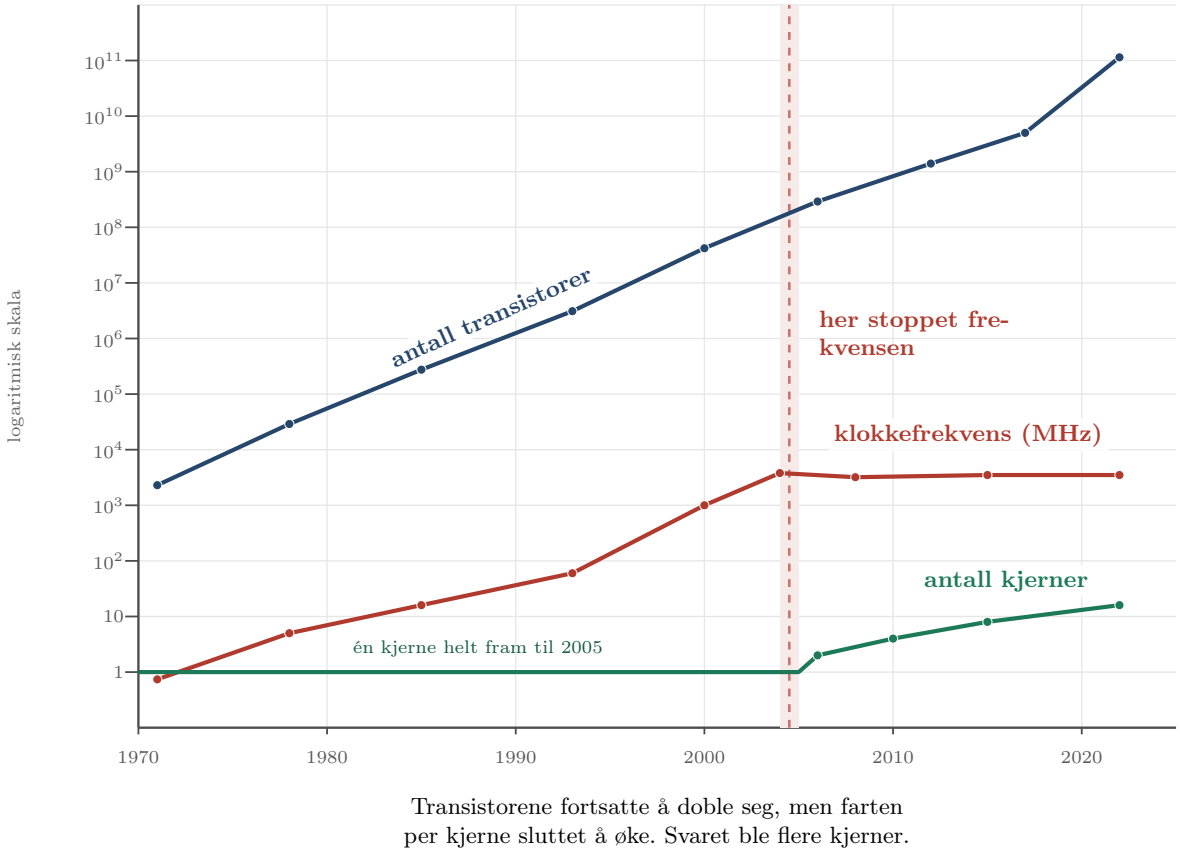
Ingenting i fysikken sier at man skal få plass til dobbelt så mange transistorer annethvert år. Moore observerte en trend og spådde at den ville holde.
Så skjedde noe interessant: hele bransjen begynte å planlegge etter spådommen. Produsenter la toårsplaner, utstyrsleverandører siktet mot de samme målene, og kundene ventet det.
Moores lov ble dermed i stor grad en selvoppfyllende profeti, en felles tidsplan som alle holdt seg til fordi alle andre gjorde det. Det gjør den ikke mindre imponerende, men det forklarer hvordan den kunne stoppe uten at noen naturlov ble brutt.

Hva fikk man ut av krympingen? To ting, og lenge fikk man begge gratis. Man fikk *flere* transistorer på samme flate, altså mer maskineri. Og man fikk *raskere* transistorer, fordi en mindre transistor har

et mindre badekar å fylle.

I tretti år gikk det slik at nye brikker hadde både flere transistorer og høyere klokkefrekvens. Maskinen du kjøpte i år var målbart raskere enn den fra i fjor, og du måtte ikke gjøre noe for å nyte godt av det. Programmene ble raskere av seg selv.

Så sluttet det.



Figur 6.15 Tre kurver fra 1971 til i dag. Antall transistorer fortsetter oppover, men klokkefrekvensen flater ut rundt 2004 — og i samme øyeblikk begynner antall kjerner å stige. Merk at y-aksen er logaritmisk, og at de tre kurvene måler hver sin ting. Det er formen på dem som er poenget, ikke at de kan sammenlignes tall mot tall.

Først en bruksanvisning, for aksen er uvanlig. Hvert hakk oppover er *ti ganger* så mye som hakket under. Uten det grepet ville ikke 2 300 transistorer og 114 milliarder fått plass på samme ark. Prisen er at et lite hopp på papiret er et stort hopp i virkeligheten, og at en kurve som ser ut til å stige jevnt, i virkeligheten dobler seg om og om igjen.

Se så på de tre kurvene. Antall transistorer fortsetter oppover gjennom hele perioden. Klokkefrekvensen følger tett etter fram til rundt 2004 og flater så nesten helt ut. I samme øyeblikk begynner antall kjerner å stige, fra én til to til fire.

Noe skjedde rundt 2004. Resten av seksjonen handler om hva.

6.10.2 Hver eneste vending koster

Gå tilbake til badekaret. Hver gang en utgang skal gå fra 0 til 1, må karet fylles. Vannet kommer fra VDD, og det koster energi å løfte det dit. Når utgangen senere skal tilbake til 0, tømmes karet ut i jord, og energien som lå i det, er borte. Ingenting av det kommer tilbake. Alt ender som varme.

Dette er svaret på spørsmålet vi åpnet kapitlet med. Telefonen blir varm fordi den fyller og tømmer milliarder av små badekar, milliarder av ganger i sekundet, og hvert eneste kar koster en slurk som blir til varme.

Hvor mye av strømmen blir varme?

Alt sammen.
Ikke «det meste», men alt. En datamaskin tar inn elektrisk energi og leverer ikke noe tilbake. Den løfter ingenting, og den lager ingen bevegelse. Det bitte lille som slipper ut som lyd fra viften eller lys fra skjermen, blir også til varme i rommet like etterpå.
En laptop som trekker 45 watt, varmer rommet like mye som en varmeovn på 45 watt. Det er energibevaring, og et av de få stedene i faget der fysikken gir deg et helt presist svar gratis.

6.10.3 Hva som bestemmer hvor varmt det blir

Tre ting avgjør varmen, og alle tre følger av badekarbildet.

Hvor mange kar. Flere transistorer betyr flere kar å fylle, og det er derfor en stor brikke bruker mer strøm enn en liten.

Hvor høyt vannet må løftes. Det er forsyningsspenningen, og den teller dobbelt, fordi du både løfter mer vann og løfter det høyere.

Hvor ofte. Det er klokkefrekvensen. Dobler du frekvensen, fyller og tømmer du dobbelt så mange ganger i sekundet, og bruker dobbelt så mye strøm.

Formelen, hvis du vil se den

Du kan hoppe over denne boksen uten å miste noe. Sammenhengen over skrives gjerne slik:

$$\text{effekt} \;\approx\; \text{kapasitans} \times \text{spenning}^2 \times \text{frekvens}$$

Det eneste du bør merke deg, er at spenningen står i andre potens. Halverer du spenningen, får du en firedel av effekten, og det gjør spenningen til den desidert mest effektive knappen å skru på. Et eksempel, fra en litt eldre brikke der spenningene var høyere enn i dag: den går i 2 GHz på 1,0 volt, og du vil ha den til 4 GHz. Frekvensen dobles, så effekten dobles. Men en høyere frekvens krever som regel også litt høyere spenning for å rekke fram i tide, la oss si 1,2 volt, og da ganges effekten i tillegg med $1{,}2^2/1{,}0^2 = 1{,}44$.
Til sammen: $2 \times 1{,}44 \approx 2{,}9$. Du doblet farten og nesten tredoblet varmen.

6.10.4 Triksen som sluttet å virke

Nå er vi framme ved det som skjedde rundt 2004.

Så lenge transistorene krympet, kunne man senke forsyningsspenningen samtidig, for mindre transistorer trenger mindre spenning. Siden spenningen teller dobbelt, falt effekten per transistor raskt nok

til at man kunne pakke inn flere og kjøre dem fortere uten at brikken ble varmere. Den balansen ga tretti år med gratis fart.

Så traff spenningen gulvet. En transistor trenger en viss spenning for å skille klart mellom på og av. Går du under den, slutter den å være en god bryter: den leder litt når den skal være stengt, og den blir treg. Støymarginen fra seksjon 6.6 krymper også, slik at signalene lettere blir misforstått.

Rundt 2004 sluttet altså spenningen å følge krympingen nedover. Før den tid falt den i takt med transistorene. Etterpå falt den mye saktere, fra rundt 1,3 volt den gangen, via 1,0 volt rundt 2010, til de 0,7 til 0,8 voltene vi ligger på nå. Fra det øyeblikket ble hver ny gigahertz betalt nesten rett i varme.

Dette punktet har fått navnet **varmeveggen**. Det er ingen vegg man kan bryte gjennom med bedre ingeniørkunst, for den står der to ønsker kolliderer: lavere spenning for å spare strøm, og høy nok spenning til at bryteren fortsatt er en bryter.

6.10.5 Og krana som drypper

Det kommer en ting til, og den blir verre jo mindre transistorene er. En stengt transistor er ikke helt stengt.



Figur 6.16 Lekkasje. En stengt transistor slipper likevel gjennom litt, og den regningen løper hele tiden — også når brikken ikke gjør noe.

Det lekker alltid litt strøm gjennom, og jo tynnere og mindre transistoren er, jo mer lekker den. Dette kalles **lekkasje**, og det er en utgift som løper hele tiden, også når brikken ikke gjør noe som helst. Da transistorene var store, var lekkasjen forsvinnende liten. I dagens størrelser er den en betydelig del av regnskapet.

Hvor lite det har blitt

For noen tiår siden kunne du tenke på utgangen til en port som et svømmebasseng. Du hadde god tid, og en liten lekkasje betydde ingenting.
I dag er det samme karet på størrelse med en teskje. Det fylles raskt, og det er poenget, men nå er lekkasjen stor i forhold til karet. Det er vanskelig å holde noe i en teskje med hull i.

6.10.6 Svaret ble flere kjerner

Da spenningen ikke kunne senkes mer, hadde bransjen et valg. De kunne fortsette å skru opp frekvensen og la brikkene bli varmere og varmere. Det gikk en liten stund, og rundt 2004 lå noen prosessorer på 3,8 GHz og krevde kjølere som fylte halve kabinettet. Eller de kunne slutte å skru opp frekvensen og bruke de stadig flere transistorene til noe annet.

De valgte det siste. I stedet for én kjerne som går dobbelt så fort, fikk vi to kjerner som går like fort som før. På papiret gir det like mange beregninger i sekundet. To kjerner har riktignok dobbelt så mange badekar å fylle, så den delen av regningen er uendret, men man slipper spenningsøkningen den doble frekvensen ville krevd, og det er nettopp faktoren 1,44 fra boksen over.

Se på figuren en gang til. Kurven for antall kjerner begynner å stige i samme øyeblikk som frekvenskurven flater ut. De to henger sammen.

Regningen ble sendt videre

Løsningen har en hake, og det er grunnen til at du som skal programmere, bør vite om dette.
En raskere kjerne gjør alle programmer raskere uten at noen trenger å gjøre noe. To kjerner gjør bare de programmene raskere som er skrevet for å bruke begge. Da frekvensen stoppet i 2004, ble regningen altså sendt fra maskinvaren til programvaren, og det er derfor parallellprogrammering gikk fra å være et spesialfelt til noe alle må forholde seg til.
Hvor langt man kommer med flere kjerner, og hvorfor det finnes en grense der også, er tema i kapittel 8.

6.10.7 Om tallene du leser i nyhetene

En siste ting, siden du kommer til å møte den.

«2 nanometer» er et navn, ikke et mål

Når en produsent sier at de lager brikker på en «2 nm-prosess», er det lett å tro at noe i transistoren er 2 nanometer langt. Det er det ikke. En gang viste tallet til en faktisk lengde i transistoren, men den koblingen ble brutt for lenge siden. I det som markedsføres som en 2 nm-prosess, er den tilsvarende lengden snarere 12 til 20 nanometer.
Nodenavnet er i dag et generasjonsnavn, omtrent som modellåret på en bil. Det forteller at denne generasjonen er tettere og bedre enn den forrige, og det stemmer, men det er ikke et mål på noe. Du trenger ikke tallene. Du trenger bare å vite at tallet ikke betyr det det ser ut som.

Og hva kommer etter? Når transistorene ikke kan bli stort mindre i flaten, begynner man å stable dem i høyden. Man bygger spesialiserte kretser som gjør én oppgave svært effektivt, i stedet for generelle kretser som gjør alt litt, og det forskes på helt andre materialer og helt andre måter å regne på. Det hører hjemme i kapittel 8, og i appendiks A finner du historien om hvordan vi kom hit.

6.11 Oppsummering

Kapitlet begynte med en varm telefon og endte med en kurve som flater ut i 2004. Underveis fant vi det kapittel 5 ikke kunne vise fram: bryteren uten finger.

6.11.1 Hva som egentlig var nytt

6.11.2 Den ene ideen, fire ganger

Det samme svaret dukket opp fire ganger.

Hvorfor kommer ettallet fra forsyningen og ikke fra inngangen? Fordi transistoren er en bryter, og en bryter slipper gjennom noe annet enn seg selv.

Begrep	Kort sagt
Transistor	Bryter som styres av et elektrisk signal, ikke av en finger.
nMOS	Leder når styresignalet er 1.
pMOS	Leder når styresignalet er 0. Tegnes med boble.
VDD	Det høye punktet i kretsen. Logisk 1. Rundt 0,8 volt i dag.
GND (jord)	Det lave punktet. Logisk 0, og referansen alt måles fra.
CMOS	De to typene brukt sammen. All digital elektronikk i dag.
Inverter	To transistorer. Inngangen velger hvilken vei som åpnes.
Spenningsområde	0 og 1 er intervaller, ikke tall. Imellom ligger et belte der ingenting loves.
Støymargin	Hvor mye juks et signal tåler før det blir misforstått.
Signalgjenoppretting	Hver port lager et nytt, skarpt signal fra forsyningen.
Kapasitans	Badekaret som må fylles. Oppstår av seg selv, ingen vil ha den.
Portforsinkelse	Tiden en port bruker på å skifte. Rundt 5 ps.
Kritisk vei	Den lengste veien gjennom kretsen. Den bestemmer farten.
Klokkeperiode	Tiden mellom to like flanker. Må romme den kritiske veien.
Frekvens	én delt på klokkeperioden. Måles i hertz.
Moores lov	Dobling annethvert år. En bransjeplan, ikke en naturlov.
Lekkasje	Stengt er ikke helt stengt. Koster strøm hele tiden.
Varmeveggen	Spenningen kan ikke senkes mer, så frekvensen kunne ikke økes mer.

Tabell 6.3 Begrepene fra dette kapitlet.

Hvorfor tåler digitale signaler støy? Fordi hver port henter en fersk verdi fra forsyningen og kaster den slitne som kom inn.

Hvorfor tar en port tid? Fordi bryteren må fylle et kar, og kar tar tid å fylle.

Hvorfor blir brikken varm? Fordi hvert kar som fylles og tømmes koster en slurk, og slurken blir til varme.

Alle fire er følger av én setning: det sitter en bryter der, og den flytter ladning.

6.11.3 Videre til neste kapittel

Dette var bunnen. Vi gikk ned fra kodelinja i kapittel 1, gjennom tallene, instruksjonene, arkitekturen og portene, og nå står vi på silisiumet. Under oss ligger halvlederfysikken, som hører hjemme i et annet fag. Herfra går veien oppover igjen.

Vi tar med oss et problem. Du så at en addisjon kan ta 960 pikosekunder, og at klokka må romme den tregeste veien. Da er det naturlig å spørre hvor lang tid det tar å hente et tall fra minnet. Svaret er ubehagelig: rundt hundre ganger lengre enn å regne med det. En prosessor bruker mesteparten av tiden sin på å vente.

Kapittel 7 handler om de to triksene som gjør at den likevel kommer noen vei: å legge det man bruker ofte nærmest, og å begynne på neste oppgave før den forrige er ferdig.

6.12 Videre lesning

Englander dekker ikke dette stoffet. Det er verdt å si rett ut, som i forrige kapittel. Boka nevner transistoren noen få ganger i historiekapitlet og forklarer den aldri. Den sier det til og med selv i avsnitt 8.6: hensikten med boka er ikke å diskutere den elektroniske implementasjonen, og den henviser leseren videre til en ingeniørlærebok.

Det er egentlig et poeng i seg selv. Englander trekker abstraksjonsgrensa akkurat der dette kapitlet begynner. Du har nettopp vært ett lag lenger ned enn pensumboka går.

Harris & Harris, *Digital Design and Computer Architecture*, kapittel 1. Den beste neste boka, og du ble allerede sendt dit i forrige kapittel — denne gangen skal du bla *tilbake* til kapittel 1. Avsnitt 1.6 tar spenningsnivåer og støymargin med tall, 1.7 tar CMOS-transistorene og bygger NAND og NOR på transistornivå, og 1.8 tar strømforbruk. Den krever ingen fysikkbakgrunn.

Patterson & Hennessy, *Computer Organization and Design*, avsnitt 1.5, 1.7 og 1.8. Her står den kanoniske framstillingen av akkurat det Englander mangler: hvordan brikker lages, Moores lov, varmeveggen med den berømte figuren, og hvorfor bransjen gikk over til flere kjerner. Hvis du bare skal lese én ting etter dette kapitlet, les 1.7.

Petzold, *Code*. Den vennligste inngangen hvis dette kapitlet føltes bratt. Petzold bygger hele veien fra brytere og releer opp til en fungerende datamaskin, uten en eneste formel. Han bruker releer der vi bruker transistorer, og det er en fordel: mekanismen er synlig.

Ben Eater, «Making logic gates from transistors», på nett. Et kvarter der han bygger porter av løse transistorer på et koblingsbrett og måler spenningene underveis. Det er nøyaktig broa mellom kapittel 5 og dette kapitlet, gjort fysisk.

Chris Mack, «Fifty Years of Moore's Law». En kort og fritt tilgjengelig artikkel om hvorfor Moores lov aldri var en naturlov, og hvilke deler av den som sluttet å gjelde når.

Videre i denne boka: appendiks D tar fysikken under transistoren — Ohms lov, halvledere, terskelspenning og transistoren brukt som forsterker. Appendiks E tar klokkefordeling på en hel brikke. Appendiks G fortsetter der dette kapitlet slapp, og forklarer hvorfor telefonen din har en egen krets for kunstig intelligens. Ingen av dem er pensum. Appendiks A forteller historien fra radiorør til silisium.

6.13 Refleksjon

1. Vi sa at kapasitansen er noe ingen vil ha, men som ikke kan unngås. Finn et eksempel fra et helt annet felt på en uunngåelig kostnad som følger av at noe er fysisk, og som alle jobber for å gjøre mindre.
2. Klokka får raske kretser til å vente på trege. Det sløser med tid, men gjør alt enklere å konstruere. Hvor ellers i datateknologien betaler man med ytelse for å få enklere regler?

3. Moores lov ble en selvoppfyllende profeti: bransjen planla etter spådommen, og derfor slo den til. Er det en styrke eller en svakhet ved en spådom at den virker på denne måten?
4. Da frekvensen stoppet i 2004, ble regningen sendt fra maskinvare til programvare. Hvem bestemte det? Og hva slags ansvar har en bransje for å fortelle brukerne sine at reglene har endret seg?

6.14 Selv-test

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

1. Hva er en transistor, sagt med én setning?
2. Hvor mange typer transistorer bruker vi i dette kapitlet, og hva skiller dem?
3. Hva betyr VDD, og hva betyr GND?
4. Hvilken logisk verdi svarer VDD til, og hvilken svarer GND til?
5. Hva betyr den lille boblen på styreledningen til en transistor?
6. Hvor mange transistorer er det i en inverter?
7. En inverter får inn 0. Hvilken av de to transistorene leder, og hva blir utgangen?
8. Hva står CMOS for, og hvorfor er navnet beskrivende?
9. Hvorfor går det nesten ingen strøm gjennom en CMOS-port som står i ro?
10. Hvor mange transistorer er det i en NAND-port? Og i en AND-port?
11. Hvorfor koster AND mer enn NAND, når AND ser ut som den enklere porten?
12. I en NAND-port står to transistorer i serie og to i parallell. Hvilke står hvor, og hvorfor gir det riktig oppførsel?
13. Forklar hvorfor 0 og 1 er spenningsområder og ikke enkelttall.
14. Hva er poenget med det udefinerte beltet mellom de to områdene?
15. En port får inn et signal som er slapt og litt rusket, men innenfor 1-området. Hvordan ser utgangen ut, og hvorfor?
16. Hvorfor blir en lydfil kopiert tusen ganger identisk med originalen, mens en kassett kopiert tusen ganger blir ubrukelig?
17. Hva er kapasitansen til en utgang, forklart med badekarbildet? Og hvorfor er det feil å si at noen har satt den inn med vilje?
18. Hva er portforsinkelse?
19. En portforsinkelse er 5 pikosekunder. Hvor mange pikosekunder går det på ett nanosekund, og hvor mange porter rekker å skifte på den tiden?
20. Hva er den kritiske veien i en krets, og hvorfor er den den eneste som betyr noe for farten?
21. En krets har tre veier fra inngang til utgang: én på 2 porter, én på 5 porter og én på 3 porter. Hver port tar 5 ps. Hvor lang tid bruker kretsen?
22. Hvorfor er svaret på en fulladder 15 ps og ikke 25 ps, når den har fem porter?
23. En OR-port har to innganger. Den ene er klar etter 5 ps, den andre etter 10 ps. Når er utgangen klar?
24. En mentekjede på 8 bit bruker 120 ps. Hvor lang tid bruker en på 32 bit, og hvorfor er svaret annerledes enn for de parallelle veiene i oppgave 21?

25. Hva er en klokkeperiode, og hva er en stigende flanke?
26. En prosessor går i 4 GHz. Hva er klokkeperioden i pikosekunder?
27. En klokkeperiode er 400 ps. Hva er frekvensen?
28. Hvorfor må klokkeperioden være minst like lang som den kritiske veien? Hva skjer hvis den ikke er det?
29. En port tar 5 ps, som skulle tilsi en frekvens på 200 GHz. Gi tre grunner til at ingen prosessor går så fort.
30. Hva sier Moores lov, og hvorfor er «lov» et misvisende ord?
31. Hvorfor teller forsyningsspenningen dobbelt i effektregnskapet, mens frekvensen bare teller én gang?
32. En brikke går i 2 GHz. Du dobler frekvensen og lar spenningen stå. Hva skjer med effekten?
33. Hva er lekkasje, og hvorfor er den et større problem i dag enn for tretti år siden?
34. Hvorfor kan man ikke bare fortsette å senke forsyningsspenningen for å spare strøm?
35. Forklar med egne ord hvorfor klokkefrekvensen sluttet å øke rundt 2004, og hvorfor svaret ble flere kjerner.
36. En laptop trekker 60 watt. Hvor mye varmer den rommet, og hvorfor er svaret så enkelt?
37. Hva betyr det egentlig når en produsent sier at en brikke er laget på en «3 nm-prosess»?
38. **Finn feilen.** En student skriver: «Klokka trengs fordi verdiene i kretsen lekker ut i løpet av noen nanosekunder, så vi må rekke å lagre dem før de forsvinner.» Hva er galt, og hva er den riktige grunnen?
39. **Finn feilen.** En annen student skriver: «AND-porter finnes ikke i en prosessor, bare NAND-porter.» Hva er riktig i påstanden, og hva er galt?
40. Kapittel 5 sa at det er verdt å spare porter. Forklar nå, med det du vet fra dette kapitlet, hva man faktisk sparer — og nevn tre ting en spart port koster mindre av.

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.

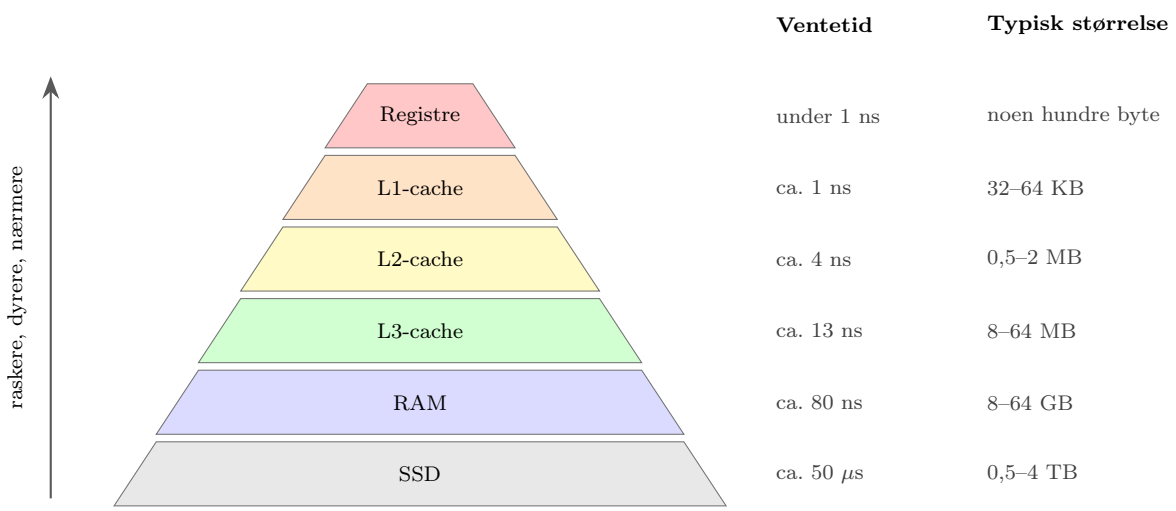
I stedet gjør du noe annet, helt uten å tenke over det: du flytter ting oppover etter hvert som du trenger dem. Boka du henter fra biblioteket, blir liggende på hylla. Siden du leser akkurat nå, ligger på pulten. Når du er ferdig med den, havner den på hylla igjen, og noe annet tar plassen.

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.

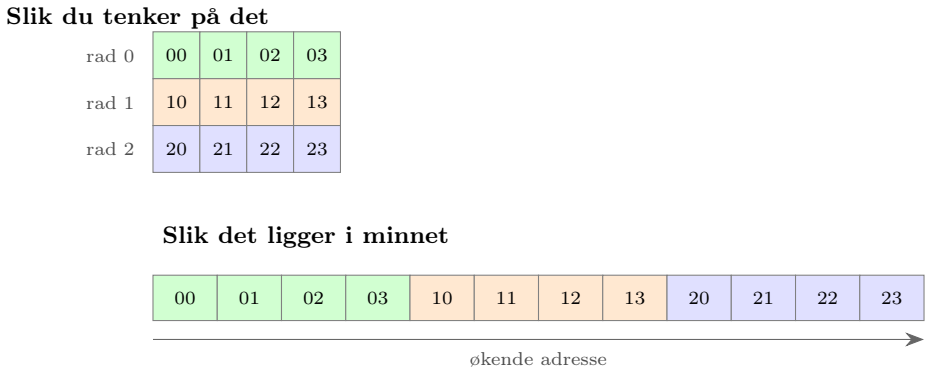
Det er verdt å merke seg at instruksjonene har lokalitet helt på egen hånd. En løkke på ti linjer som kjøres en million ganger, er ti linjer med ekstrem tidslokalitet. Det er derfor cachen virker like godt på kode som på data — og, som vi kommer til i avsnitt 7.4, derfor maskinen ofte har en egen cache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

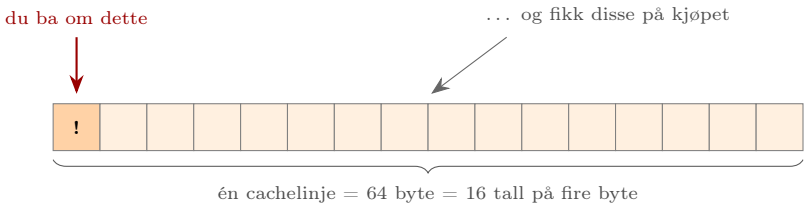
Du går ned, finner den, og går opp. Neste gang trenger du en tang. Ned igjen, opp igjen. Så en skrujern til. Ned, opp.

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

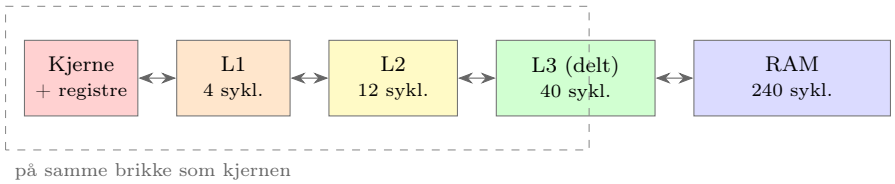
En cachelinje rommer fire tall. Et program leser en liste med tolv tall, i rekkefølge, én gang. Cachen er tom når programmet starter, og lista begynner akkurat der en linje begynner.
Hvor mange bommer blir det? Hvor mange treff? Hva er treffraten?
Og så det som er verdt å tenke på: hva skjer med svaret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

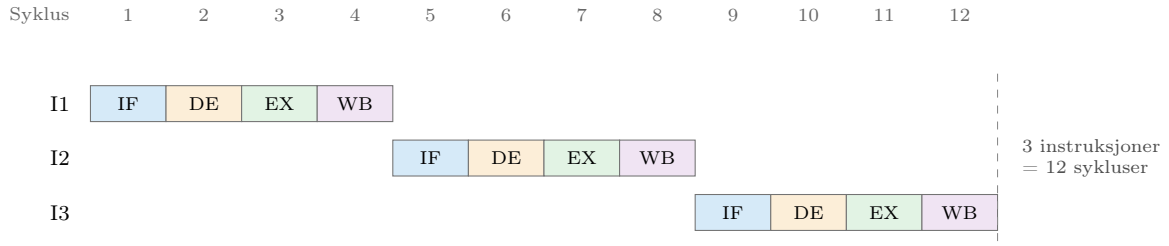
I kapittel 3 og 4 fulgte du en instruksjon gjennom maskinen. Den ble hentet, tydet, utført, og resultatet ble lagret.

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.

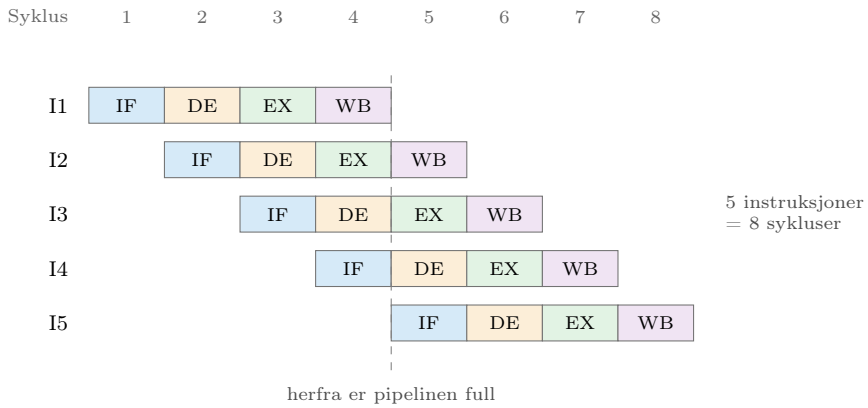
Uten pipeline gjør maskinen én instruksjon helt ferdig før den begynner på neste. Figur 7.5 viser det.



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.

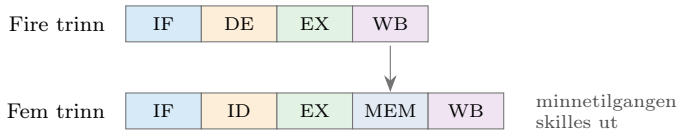
«Fire er den korteste pipelinen som finnes»

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.

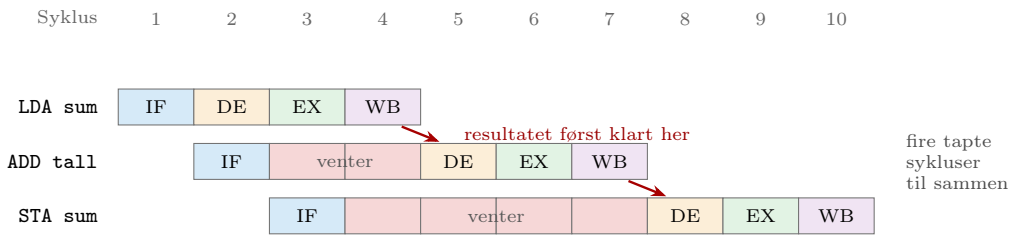
Her er tre instruksjoner i stil med dem fra kapittel 3. De legger et tall til en sum:

```
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.

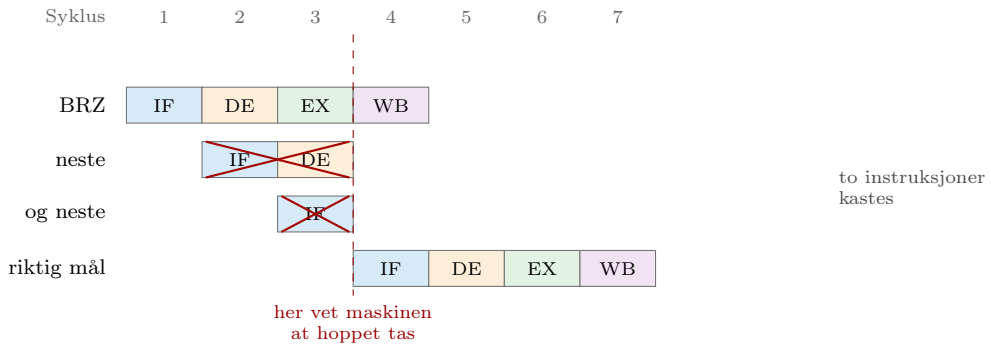
Men nå er de et problem.

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?

Kapittel 8

Moderne maskiner og hele stabelen

Innhold

8.1 Læringsmål 203
8.2 Maskinen som ikke venter 203
8.3 Flere kjerner, og grensen for dem 210
8.4 Når oppgaven ikke ligner på et program 213
8.5 Brikken for nevrale nett 215
8.6 Grensesnittene mot verden 217
8.7 Når abstraksjonen lekker 220
8.8 Hvor skal grensen gå? 223
8.9 Hele stabelen 224
8.10 Lagkartet 227
8.11 Oppsummering 228
8.12 Videre lesning 230
8.13 Refleksjon 230
8.14 Selv-test 231
Og det var hele reisen 233

Du har sett en slik liste før.

```
Prosessor      8 kjerner, 4,0 GHz
Buffer         12 MB delt L3
Minne          16 GB
Grafikk        integrert, 10 kjerner
Nevral motor   16 kjerner
```

Den sto i nettbutikken sist noen i familien skulle kjøpe telefon eller PC, eller på et ark fra Universitetets innkjøpsavtale. Den er skrevet for å imponere, og den gjør en elendig jobb med å forklare noe.

Les den en gang til. For du kan den nå.

Du vet hva en kjerne er, fordi kapittel 7 trengte én for å forklare cache-koherens. Du vet hva 4,0 gigahertz betyr, fordi du regnet ut klokkeperioden selv i kapittel 6. Du vet hvorfor det står en buffer på 12 megabyte mellom prosessoren og de 16 gigabytene. Du kan til og med anslå hvor mange klokkesykluser maskinen sparer hver gang den finner det den leter etter der.

For et par måneder siden var dette fem linjer med tall og engelske ord.

Men her er det som fortsatt står igjen, og det er et bedre spørsmål enn «hva betyr ordene».

Hvorfor åtte kjerner, og ikke én som er åtte ganger raskere? Du vet halve svaret fra kapittel 6: klokka stoppet fordi brikken ble for varm. Men det forklarer bare hvorfor de *måtte* gjøre noe annet. Det forklarer ikke om det virket.

Hvorfor står det tre forskjellige slags regneenheter på samme brikke? Åtte vanlige kjerner, ti grafikkjerner og seksten nevrale. Alle sammen er laget av de samme transistorene, de kjører på den samme klokka, og de ligger på det samme stykket silisium. Hvorfor ikke bare lage trettifire like?

Og hvorfor kan en maskin som gjør milliarder av addisjoner i sekundet, lures til å røpe et passord den aldri ble bedt om å vise fram?

Ingen av de tre spørsmålene handler om nye komponenter. De handler om valg — om hvorfor maskinen ble som den ble. Det er det dette kapitlet svarer på.

Du trenger nesten ingenting nytt for å lese videre.

Dette er det eneste kapitlet i boka som ikke går ned et lag. Du kommer til å møte en god del nye *navn*, men ingen ny byggestein: alt sammen er satt sammen av porter, transistorer og pipelinetrinn du allerede har sett. Det bygger av deler du allerede har. Pipelinen og cachen fra kapittel 7. Porten fra kapittel 5. Transistoren og varmeregnskapet fra kapittel 6. Instruksjonsformatet fra kapittel 4, og lagene fra kapittel 1.

Kapitlet er langt. Det er langt fordi det er mange svar, ikke fordi det er tungt. Og det er bygget i korte bolker som står på egne bein. Hver seksjon sier med en gang hvilket problem den løser. Da kan du lese dem i den rekkefølgen du vil, og slå opp den ene du trenger.

Vi regner lite, og vi regner i ord og tall — ingen formler med symboler i. To steder er det verdt å lese sakte: en deling i avsnitt 8.2.5, og et regnestykke i tre steg i avsnitt 8.3.3. Begge er skrevet ut.

Og hvis du vil hoppe: seksjonene kan leses i den rekkefølgen du vil, med étt unntak. Avsnitt 8.7 forutsetter avsnitt 8.2.

Hvis noe har føltes uferdig, var det med vilje

Kapittel 7 endte med to problemer vi navnga og ikke løste. Datahasarden som koster to sykluser hver gang, og kontrollhasarden som tømmer hele pipelinen ved hvert hopp. Vi sa begge ganger at det finnes løsninger, og at de kommer senere.
Kapittel 6 endte med en vegg: klokka kunne ikke gå fortere uten at brikken brant opp. Svaret ble flere kjerner, og vi sa at det er en grense der også, uten å si hvilken.
Og vi har lovet deg hva en spesialisert krets er, hva virtuelt minne er, og hvorfor L1-cachen er delt i to — det siste fikk du riktignok svar på.
Dette er kapitlet der regningene gjøres opp. Ingen av dem krever at du lærer noe nytt først.

Hva du sitter igjen med

To ting, og begge er noe du kan sjekke selv på siste side.
Du skal kunne lese databladet over linje for linje og si *hvorfor* hver linje står der — ikke bare hva ordet betyr, men hvilket problem akkurat den komponenten løser.
Og du skal kunne ta én linje Python, følge den ned gjennom alle lagene til transistorene som faktisk bytter, og tilbake opp igjen med svaret. Hele veien, uten hull.
Det siste er hele boka på fem sider, og det er avsnitt 8.9.

8.1 Læringsmål

Etter å ha arbeidet med dette kapitlet skal du kunne:

1. **Forklare hvordan maskinen slipper unna de fleste datahasardene.** Svaret er enklere enn problemet, og det tar én figur.
2. **Forklare hva forgreningsprediksjon og spekulativ utføring er.** Og hvorfor en maskin som gjetter, gjetter riktig så ofte.
3. **Skille superskalar fra pipelining, og kjerne fra tråd.** Fire ord som ofte blandes, og som betyr fire forskjellige ting.
4. **Regne på Amdahls lov med konkrete tall.** Og svare på hvorfor et program ikke blir åtte ganger raskere av åtte kjerner.
5. **Avgjøre om en oppgave passer best på CPU, GPU eller en spesialisert krets.** Og begrunne valget med noe annet enn «GPU-en er raskere».
6. **Si hva et nevralt nett regner ut, og hvorfor det gir en egen brikke.** Kort her; appendiks G tar hele historien.
7. **Forklare hva et avbrudd er, og hvorfor maskinen ikke bare sjekker selv.** Dette er grensesnittet mot resten av verden.
8. **Forklare hva virtuelt minne gir deg, og hvorfor det først og fremst handler om isolasjon.** Og hvor det passer inn i minnehierarkiet.
9. **Forklare et sikkerhetshull som stammer fra maskinvaren.** Du har allerede begge delene som trengs for å forstå Spectre.
10. **Følge én operasjon gjennom hele stabelen, fra kodelinje til transistor og opp igjen.** Og si hva abstraksjonen koster, med et tall.

8.2 Maskinen som ikke venter

Veggen: i kapittel 7 fant vi at pipelinen stopper hele tiden. En datahasard koster to sykluser, og `LDA`, `ADD`, `STA` er noe av det vanligste som finnes. En kontrollhasard tømmer pipelinen, og hvert eneste `if` er et hopp.

Vi lovet løsninger. Her er de, og den første er nesten pinlig enkel.

8.2.1 Resultatet trenger ikke ta omveien

Se på problemet en gang til.

```
I1:  LDA sum
I2:  ADD tall
```

Instruksjon 2 trenger verdien instruksjon 1 henter. Vi sa at den måtte vente til syklus 4, fordi det er da instruksjon 1 skriver verdien tilbake i WB-trinnet.

(én liten justering av modellen underveis: kapittel 7 lot instruksjon 2 hente operandene sine i DE-trinnet. Fra nå av henter den dem i starten av EX. Det er der videresendingen leverer, og det er slik ekte pipelines gjør det.)

Men stopp ved det et øyeblikk. *Hvorfor* må den vente til da?

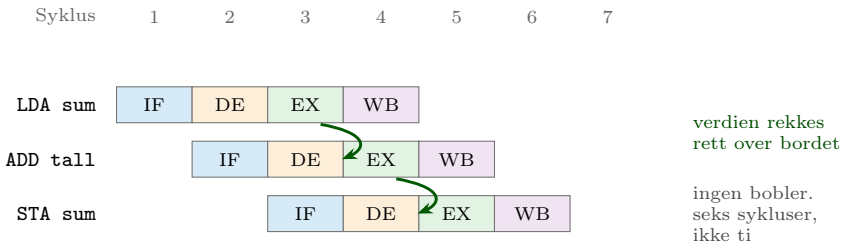
Verdien finnes jo. Den kom ut av minnet allerede i syklus 3, mens instruksjon 1 sto i EX. Den ligger og venter i en ledning inne i maskinen. Det eneste som gjenstår, er at den skal legges i akkumulatoren — og så skal instruksjon 2 hente den ut av akkumulatoren igjen.

Det er to ekstra skritt for en verdi som allerede er der.

Tenk på det slik. Du og en medstudent sitter ved samme bord og jobber på hver deres del av en innlevering. Du regner ut et tall hun trenger. Du kan skrive tallet på et ark, legge arket i hennes hylle nede i gangen, og så kan hun gå ned og hente det. Eller du kan rekke det over bordet.

Maskinen kan rekke det over bordet. Det legges inn en ledning fra utgangen av EX-trinnet rett tilbake til inngangen av EX-trinnet, slik at resultatet kan brukes i neste syklus — før det er skrevet noe sted.

Det heter **forwarding**, eller videresending. Det engelske ordet brukes mest, også på norsk.



Figur 8.1 Videresending. Samme tre instruksjoner som i figur 7.8 på side 193, men nå går resultatet rett fra utgangen av ett EX-trinn til inngangen av det neste. Avhengigheten er der fortsatt. Hasarden er borte.

Sammenlign med figur 7.8 på side 193. Der brukte de tre instruksjonene ti sykluser. Her bruker de seks, som er nøyaktig det en pipeline uten stans skal bruke.

Her ser du hvorfor skillet i kapittel 7 var verdt å holde på

Vi insisterte på at en *avhengighet* og en *hasard* er to forskjellige ting, og at det hørtes ut som ordkløveri.
Nå kan du se hvorfor det ikke er det.
Avhengigheten i koden over er fullstendig uendret. Instruksjon 2 trenger fortsatt resultatet fra instruksjon 1, og det kommer den alltid til å gjøre — det er en egenskap ved programmet, og ingen maskin kan trylle den bort.
Hasarden er borte. Den var en egenskap ved maskinvaren, og maskinvaren ble endret.
Dette er mønsteret for hele kapitlet, og det er verdt å ta med seg: de moderne triksene fjerner nesten aldri problemet. De fjerner konsekvensen av det.

Løser videresending *alle* datahasarder? I vår firetrinnsmodell gjør den omtrent det, og grunnen er at vi leser fra minnet inne i EX-trinnet. Verdien er derfor klar akkurat i tide, også når den kommer fra minnet — det er nettopp tilfellet figur 8.1 viser.

I en pipeline med eget minnetrinn er det annerledes. Der kommer lasteverdien først ut etter MEM, altså én syklus for sent for instruksjonen rett bak. Det tilfellet heter *last-bruk*, og det er den ene boblen selv en moderne prosessor ikke klarer å fjerne helt. Den er grunnen til at rekkefølgen på to instruksjoner av og til har noe å si, selv når de ikke rører hverandre.

8.2.2 Å gjette hvor det bærer

Den andre hasarden er verre.

Problemet var at maskinen ikke vet om et betinget hopp skal tas før hoppet er utført, og at den innen da har hentet inn to instruksjoner som kanskje er feil. I vår firetrinnsmodell koster det to instruksjoner. I en ekte prosessor med femten eller tjue trinn koster det femten eller tjue.

Her kan ingen ledning hjelpe. Informasjonen finnes ikke ennå — ingen vet hvor programmet skal før sammenligningen er gjort.

Så maskinen gjør det eneste som er igjen. Den gjetter.

Det høres ut som en dårlig idé, og førstegangsreaksjonen er som regel: hvor ofte kan det gå bra? Svaret er at det går bra overraskende ofte, og grunnen er at hopp i programmer er alt annet enn tilfeldige.

Tenk på en løkke som går tusen runder. På slutten av hver runde står det et hopp tilbake til toppen. Det hoppet tas 999 ganger og ikke-tas én gang. En maskin som bare antar «hopp bakover tas alltid», treffer 99,9 prosent av gangene på den løkka.

Det er den enkleste varianten, og den brukes faktisk: hopp bakover antas tatt, hopp framover antas ikke tatt. Hopp bakover er som regel løkker. Hopp framover er som regel feilhåndtering som sjelden utløses.

Maskinen i telefonen din gjør noe bedre. Den *husker*.

Du kjenner allerede prinsippet

Du bor i kollektiv med fire andre, og dere har en felles kalender. Hver morgen skal du gjette om noen har brukt opp melka.
Du kunne gjettet likt hver gang. Men etter to uker har du lagt merke til noe enkelt: som regel er det i dag som det var i går. Så du slutter å gjette likt, og begynner å gjette det samme som sist.
Du har ikke fått ny informasjon om kjøleskapet. Du har bare begynt å bruke historikken du allerede hadde.
En prosessor gjør akkurat det. Den fører en liten tabell over hvert hopp den har sett, og hva som skjedde sist. Står hoppet på en adresse den kjenner igjen, gjetter den det samme som forrige gang.

Mekanismen heter **forgreningsprediksjon**, på engelsk *branch prediction*. Tabellen er fysisk maskinvare inne på brikka: noen tusen små tellere som hver husker hva et bestemt hopp pleier å gjøre.

Prediktorene i dag gjør mer av det samme, ikke noe annet. De ser på mønsteret i de siste gangene i stedet for bare på den forrige, og på hva *andre* hopp gjorde rett før — altså nøyaktig det du gjorde da du begynte å bruke ukedagen på melka. Prinsippet er uendret: bruk historikken du allerede har.

Treffraten på vanlige programmer ligger over 95 prosent, og på forutsigbar kode over 99.

Løftet fra kapittel 7, innfridd

Vi skrev dette i boksen om tester inne i løkker:
«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.»
Nå har du grunnen. En `if` som nesten alltid går samme vei, lærer prediktoren seg på et par runder, og etter det koster den ingenting. En `if` som veksler tilfeldig, er umulig å lære — og da bommer maskinen omtrent annenhver gang og kaster alt som sto i pipelinen.
Det er derfor to programmer som gjør nøyaktig det samme kan skille betydelig i fart, og hele forskjellen ligger i om testene er forutsigbare. Det er samme slags håndtak som lokaliteten i kapittel 7:

noe du styrer, som ikke endrer hva programmet regner ut.

8.2.3 Å gjøre arbeidet før man vet om man får lov

Og her kommer et skritt til, som er større enn det ser ut.

Maskinen gjetter hvilken vei hoppet går. Men den stopper ikke der og venter på å få bekreftet gjetningen. Den *utfører* instruksjonene den har gjettet på — regner ut verdiene, henter fra minnet, fyller pipelinen helt opp.

Går gjetningen inn, er alt klart og arbeidet teller. Går den ikke inn, kastes alt sammen, og maskinen begynner på nytt fra riktig sted.

Det kalles **spekulativ utføring**.

Det er en handel: maskinen bruker strøm på arbeid som kanskje må kastes, og slipper til gjengjeld å stå stille i de tilfellene der gjetningen var riktig. Med en treffrate over 95 prosent er handelen god.

Hvorfor «kastes» er det viktige ordet her

Når gjetningen var feil, skal det ikke være mulig å se at maskinen gjettet.
Registrene settes tilbake. Resultatene forkastes. Programmet ditt får nøyaktig de samme svarene som det ville fått på en maskin som aldri spekulerte. Det er hele kontrakten — spekulasjon er lov nettopp fordi den er usynlig.
Legg den setningen på minnet. Vi kommer tilbake til den i avsnitt 8.7, og da skal vi se at den er sann på én måte og usann på en annen. Det er hele hemmeligheten bak et av de mest kjente sikkerhetshullene som finnes.

8.2.4 Flere av alt: superskalar

Så langt har vi reparert pipelinen. Nå utvider vi den.

En pipeline med fire trinn kan i beste fall bli ferdig med én instruksjon per syklus. Det er taket, og det er et tak som ligger i strukturen: det er bare étt EX-trinn, og dermed bare én instruksjon som kan regne av gangen.

Men det er ingen naturlov at det bare skal være ett. Hvorfor ikke bygge to ALU-er? Eller fire?

Da kan maskinen hente to instruksjoner samtidig, tyde begge, og utføre begge i samme syklus. En prosessor som gjør dette, kalles **superskalar**.

Her er skillet som er verdt å holde rett, fordi det er et klassisk eksamensspørsmål og et klassisk rot:

	Pipelining	Superskalar
Hva maskinen har flere av	Trinn	Hele regneenheter
Hva som skjer samtidig	Fire instruksjoner, men i *hvert sitt* trinn	Flere instruksjoner i *samme* trinn
Taket	én ferdig instruksjon per syklus	Flere ferdige per syklus
Analogien	Samlebånd	Flere samlebånd ved siden av hverandre

Tabell 8.1 Pipelining mot superskalar. De to er uavhengige ideer, og en moderne prosessor bruker begge samtidig.

Og nå det fine: du kan måle dette selv, på din egen maskin, på noen sekunder.

8.2.5 Fire addisjoner i samme klokkeslag, målt

Her er to små programmer. Begge gjør nøyaktig to milliarder addisjoner.

Det første legger til én om gangen i det samme tallet. Hver addisjon må vente på den forrige — du kan ikke legge til den tusende før den nihundreognittiniende er ferdig.

Det andre teller fire tall oppover samtidig. Fortsatt to milliarder addisjoner, men nå henger de ikke sammen: de fire er helt uavhengige av hverandre.

```
/* A: lenket. hver addisjon venter på den forrige */
for (i = 0; i < N; i++) { x += 1; }

/* B: fire uavhengige kjeder, like mange addisjoner */
for (i = 0; i < N/4; i++) { a += 1; b += 1; c += 1; d += 1; }
```

Samme antall addisjoner. Samme maskin. Her er tiden:

Variant	Tid	Per addisjon	Per syklus
Siste kolonne regnet om fra kolonnen foran, med 0,249 ns som klokkeperiode.			
A — lenket	0,497 s	0,249 ns	1,00
B — fire uavhengige	0,124 s	0,062 ns	4,02

Tabell 8.2 To milliarder addisjoner, målt to ganger. Den eneste forskjellen mellom A og B er om addisjonene er avhengige av hverandre.

Fire ganger raskere, med nøyaktig like mye arbeid.

Og legg merke til tallet helt til høyre i rad B. Maskinen blir ferdig med litt over fire addisjoner i hver eneste klokkesyklus. Det er ikke én per syklus, som en vanlig pipeline lover. Det er fire.

Du ser altså superskalaren direkte. Rad B viser at brikka klarer *minst* fire addisjoner i samme syklus — med fire kjeder kan målingen ikke vise mer, uansett hvor bred maskinen er. Flere kan den godt ha, og det finner du ut ved å øke antall kjeder til tallet slutter å stige.

I rad A kan bare én enhet arbeide på selve summen, fordi kjeden ikke gir de andre noe å gjøre med den.

Der klokkefrekvensen kom fra

Rad A er mer enn en sammenligning. Den er en måling av klokka.
En lenket addisjonskjede kan ikke gå fortere enn én addisjon per klokkesyklus. Hver av dem venter på resultatet fra den forrige, og et resultat finnes tidligst ved neste klokkeflanke.
Måler en slik kjede 0,249 nanosekunder per addisjon, går klokka altså minst så fort. Og siden en heltallsaddisjon er noe av det raskeste en maskin gjør — den tar nøyaktig én syklus — er klokkeperioden 0,249 nanosekunder.
Og da er frekvensen én delt på det. Ett sekund rommer rundt fire milliarder perioder på 0,249 nanosekunder, så klokka går omtrent 4,0 gigahertz. Vi slo den ikke opp. Vi målte den, med en `for`-løkke og en stoppeklokke, med formelen fra avsnitt 6.9.
Tallene over er fra maskinen boka ble skrevet på. Kjør programmene på din egen, så får du din egen klokkefrekvens og din egen nedre grense for antall regneenheter.
én advarsel hvis du gjør det: en kompilator som får lov, ser at løkke A bare legger til én to milliarder ganger, og bytter hele løkka med én addisjon. Da måler du null. Variablene må merkes slik at kompilatoren ikke har lov til å ta snarveien — i C gjøres det med `volatile`. Det står i den ferdige filen.

Hvorfor ble det ikke åtte?

Hvis det hjelper å ha fire kjeder, burde vel åtte hjelpe mer?
Prøv det. Svaret er nei — det flater ut, og det flater ut på et tall som forteller deg hvor mange regneenheter brikka faktisk har.
Hva er det som setter grensen? Og hva ville skjedd med rad A hvis du i stedet gjorde addisjonene *tyngre*, for eksempel byttet dem med divisjoner?

8.2.6 Ut av rekkefølge

Det er en ting til, og den følger naturlig av målingen du nettopp så.

I rad A sto tre regneenheter ubrukt fordi instruksjonene hang sammen. Ekte programmer ligger et sted imellom: noen instruksjoner henger sammen, andre gjør ikke det, og de ligger hulter til bulter om hverandre.

Si at instruksjon 5 må vente på et tall fra minnet. Instruksjon 6 venter på instruksjon 5. Men instruksjon 7 er helt uavhengig — den regner på noe annet.

En maskin som holder seg strengt til rekkefølgen, står stille og venter sammen med instruksjon 5, selv om instruksjon 7 kunne vært gjort for lengst.

En maskin som ikke gjør det, kalles **out-of-order**, eller ute av rekkefølge. Den ser på et vindu av de nærmeste instruksjonene — gjerne hundre eller flere — plukker ut dem som har alt de trenger, og setter dem i gang.

Du gjør det samme på kjøkkenet. Oppskriften sier at du skal koke pastaen, så lage sausen, så dekke bordet. Du setter på vannet først og dekker bordet mens det koker opp, fordi bordet ikke er avhengig av pastaen. Du følger ikke oppskriften i rekkefølge. Du følger den i avhengighetsrekkefølge.

Men resultatet må se ut som om rekkefølgen ble fulgt

Dette er punktet som gjør hele ideen lovlig, og det er verdt å lese sakte.
Instruksjonene *utføres* i den rekkefølgen som passer maskinen. Men de *fullføres* — altså får lov til å endre noe som er synlig — i den rekkefølgen programmet står i.
Maskinen holder resultatene tilbake i et eget område til tur-en kommer. Først da skrives de inn der programmet kan se dem.
Hvorfor er det så viktig? Fordi programmet ditt ellers ville fått feil svar. Og fordi det er nøyaktig den samme kontrakten som i boksen om spekulasjon. Maskinen får gjøre hva den vil på innsiden, så lenge utsiden ser ut som den enkle modellen fra kapittel 3 lovte.
Du skrev et program som gjør én ting av gangen. Du får svarene til et program som gjør én ting av gangen. Det som skjedde imellom, angår deg ikke.
Helt til avsnitt 8.7.

8.2.7 Og når det likevel blir ventetid: to tråder på én kjerne

En siste teknikk, og den er et svar på noe du allerede vet.

Selv med forwarding, prediksjon, spekulasjon, fire regneenheter og out-of-order, står prosessoren stille mye av tiden. Grunnen er kapittel 7: en bom som må helt til RAM koster rundt 240 sykluser, og i de 240 syklusene finner ikke maskinen nok uavhengig arbeid til å fylle fire regneenheter. (Syklustallene derfra er regnet ved 3 GHz. På en raskere maskin blir de litt større, siden ventetiden i nanosekunder er den samme — men størrelsesordenen holder.)

Så hva gjør man med en kjerne som har ledig kapasitet, men ikke nok arbeid?

Man gir den et program til.

Et slikt uavhengig arbeidsstykke heter en **tråd**. Det kan være to forskjellige programmer, eller to deler av det samme som jobber hver for seg — avsnitt 8.3 rydder ordentlig opp i ordet. Her trengs bare at to tråder er to ting som kan gjøres uavhengig av hverandre.

Svaret heter **SMT**, *simultaneous multithreading*. Hos Intel markedsføres det som hyperthreading.

En slik kjerne holder to sett registre i stedet for ett. Da kan den ha to uavhengige programmer inne i maskineriet samtidig. Når det ene venter på minnet, har det andre som regel noe å gjøre. De deler på de samme regneenhetene.

Operativsystemet ser to kjerner der det fysisk er én. Derfor viser en firekjerners maskin med SMT gjerne åtte i systemovervåkingen.

Men du fikk ikke åtte kjerner

Dette er det mest misforståtte tallet i hele databladet, og misforståelsen er lett å havne i.
De to trådene *deler* regneenheter, cache og alt annet. Det eneste som er doblet, er registrene og litt bokholderi. Får begge trådene mye å gjøre samtidig, konkurrerer de om de samme fire ALU-ene. Gevinsten er derfor ikke det dobbelte. Den ligger typisk på mellom ti og tretti prosent, og på enkelte oppgaver er den negativ, fordi de to trådene dytter hverandres data ut av L1-cachen.
SMT er et triks for å fylle hull. Det er ikke en kjerne til.

8.2.8 Hva det kostet

La oss stoppe og se på hva vi nettopp har bygget.

Videresendingsledninger. En tabell med tusenvis av tellere som husker hva hvert hopp pleier å gjøre. Et maskineri som utfører arbeid det kanskje må kaste. Fire sett regneenheter. Et vindu på hundre instruksjoner som sorteres etter avhengighet. To sett registre.

Ingenting av dette regner ut noe som helst. Ikke én av de tingene gjør en eneste addisjon. De er alle sammen bokholderi — maskineri som finnes bare for å holde regnemaskineriet i arbeid.

Og det er dyrt. Det koster transistorer, det koster areal, og etter kapittel 6 vet du hva det betyr: det koster strøm, og strømmen blir til varme.

I en moderne CPU-kjerne går størstedelen av arealet til dette bokholderiet, ikke til regning.

Og dette er hele grunnen til at GPU-en finnes

Hold på det regnskapet, for det er avsnitt 8.4 i et nøtteskall.
Alt bokholderiet over finnes av én grunn: å få mest mulig ut av én instruksjonsstrøm som maskinen ikke visste noe om på forhånd. Prosessoren leter desperat etter arbeid som kan gjøres parallelt, i en kode som er skrevet som om alt skulle skje etter hverandre.
Men hva om du *på forhånd* vet at du har åtte millioner helt like, helt uavhengige oppgaver?
Da trenger du ikke lete. Og da kan du rive ut hele letemaskineriet og bruke arealet på noe som faktisk regner.

8.3 Flere kjerner, og grensen for dem

Veggen: kapittel 6 endte med at klokka stoppet rundt 2004. Spenningen kunne ikke senkes mer, og da kunne ikke frekvensen økes mer uten at brikken brant opp. Svaret ble å bruke de stadig flere transistorene på flere kjerner i stedet for på én raskere.

Vi sa den gangen at det finnes en grense for den løsningen også. Nå skal vi finne den, og den er ikke der folk flest tror.

8.3.1 Kjerne, tråd og prosess: tre ord som blandes

Først en opprydding, for disse tre ordene brukes om hverandre i dagligtale og betyr helt forskjellige ting.

En **kjerne** er fysisk. Det er en komplett prosessor — egne registre, egen ALU, egen pipeline, egen L1-cache — og den tar plass på silisiumet. Du kan se kjernene på et bilde av brikka: de ser like ut og ligger ved siden av hverandre.

En **prosess** er et program som kjører. Nettleseren din er en prosess. Den har sitt eget minne, og den vet ingenting om de andre prosessene i maskinen.

En **tråd** er en utføringsvei inne i en prosess. Ett program kan ha flere tråder som jobber på hver sin del av oppgaven og deler det samme minnet. Nettleseren har gjerne en tråd som tegner sida, en som laster ned bilder, og en per fane.

Forholdet mellom dem er enkelt å si og lett å glemme: **trådene er arbeidet, kjernene er arbeiderne.** Operativsystemet fordeler tråder på kjerner, og det kan godt være tusen tråder og åtte kjerner — da bytter operativsystemet mellom dem så fort at det ser samtidig ut.

Derfor henger ikke maskinen selv om et program henger

Et program som går i en evig løkke, legger beslag på én kjerne. De sju andre er ledige, og operativsystemet fortsetter å gi dem til alt annet.
På en enkjerners maskin — og de fantes, i mange år — var dette merkbart annerledes. Et program som ikke oppførte seg, kunne gjøre hele maskinen ubrukelig.
Det er ikke operativsystemet som har blitt mye smartere. Det er maskinvaren som har blitt flere.

8.3.2 Det åtte kjerner faktisk gir deg

Her er den ubehagelige delen, og den er lettere å se med et eksempel enn med en forklaring.

Du skal lage middag til tjue personer. Jobben tar fire timer alene. Du får sju til å hjelpe deg.

Blir dere ferdige på en halvtime?

Nei. For noe av arbeidet lar seg ikke dele. Gryta med suppe skal koke i førti minutter uansett hvor mange som står rundt den. Ovnen tar den tiden den tar. Og til slutt må én person smake, justere og bestemme når maten er ferdig — det går ikke an å være åtte om.

Resten — skrelling, kutting, dekking — lar seg dele fint.

Si at en time av de fire er slik udelelig jobb, og tre timer lar seg dele på åtte.

Steg 1. Den delelige delen: $3 \text{ timer} \div 8 = 22{,}5$ minutter
Steg 2. Den udelelige delen: 60 minutter, uendret
Steg 3. Til sammen: $60 + 22{,}5 = 82{,}5$ minutter

Fra 240 minutter til 82,5. Det er 2,9 ganger raskere, ikke 8.

Og nå det virkelig ubehagelige. Si at du fikk tusen hjelpere i stedet for sju.

Den delelige delen går mot null. Den udelelige timen står der. Du kommer aldri under 60 minutter, uansett hvor mange du er — og 240 delt på 60 er fire.

Fire ganger er taket. Ikke tusen. Fire. Og det eneste som bestemte taket, var hvor stor andel av jobben som ikke lot seg dele.

8.3.3 Amdahls lov

Det du nettopp regnet ut, har et navn. Det heter **Amdahls lov**, etter Gene Amdahl, som skrev det ned i 1967.

Loven sier bare dette: **den delen av jobben som ikke lar seg dele, bestemmer hvor raskt det kan gå, uansett hvor mange som hjelper til.**

Regnestykket er det samme hver gang, og det er de tre stegene du nettopp gjorde. Del den delelige delen på antall kjerner. Legg til den udelelige delen urørt. Sammenlign med tiden du startet med.

Her er hva det gir for et program der nitti prosent lar seg dele — altså et program de fleste ville kalt godt parallellisert:

Kjerner	Delelig del	Udelelig del	Gevinst
1	90	10	1,0×
2	45	10	1,8×
4	22,5	10	3,1×
8	11,25	10	4,7×
16	5,6	10	6,4×
64	1,4	10	8,8×
1000	0,09	10	9,9×

Tabell 8.3 Amdahls lov for et program der 90 prosent lar seg dele. Tallene er tidsenheter: programmet tar 100 på én kjerne. Taket er ti ganger, og de siste 936 kjernene bidrar med 1,1 av dem.

Les kolonnen til høyre nedover. Første dobling gir 1,8 ganger. Andre dobling gir 3,1. Tredje gir 4,7. Deretter begynner det å flate ut, og fra 64 til 1000 kjerner vinner du tretten prosent.

De åtte kjernene i databladet

Gå tilbake til lista på første side i kapitlet.
De åtte kjernene gir deg i beste fall 4,7 ganger på et program som er bra parallellisert. På et program der halvparten lar seg dele, gir de 1,8. Og på et program som er skrevet som en rett strek — altså de fleste programmer noen skriver i sitt første semester — gir de nøyaktig ingenting. Programmet kjører på én kjerne, og de sju andre står og ser på.
Dette er regningen fra kapittel 6, og nå kan du sette tall på den. Da klokka sluttet å øke, sluttet programmer å bli raskere av seg selv. Siden har gevinsten ligget hos den som skriver programmet. Men merk hva som *ikke* følger av dette. Åtte kjerner er ikke bortkastet selv om ett program ikke bruker dem. Maskinen din kjører aldri bare ett program — nettleseren, musikken, oppdateringen i bakgrunnen og tekstbehandleren er fire prosesser som gjerne kan ligge på hver sin kjerne. Det er derfor maskinen føles mye jevnere enn Amdahl skulle tilsi.

8.3.4 Hvorfor den udelelige delen er større enn du tror

Amdahls lov forutsetter at delingen er gratis. Det er den ikke, og det er den andre grensen.

Noen må dele ut arbeidet og samle det inn igjen. Å starte åtte tråder, gi dem hver sin bit, og vente på at alle er ferdige, tar tid i seg selv. Er jobben liten nok, bruker du mer tid på å organisere enn på å gjøre.

Trådene må vente på hverandre. To tråder som skal legge til i samme sum, kan ikke gjøre det samtidig — da kan den ene lese tallet før den andre har skrevet sitt, og en av addisjonene forsvinner sporløst. Løsningen er at bare én slipper til av gangen, og da står de andre i kø. Hver gang tråder må stille seg i kø for hverandre, vokser den udelelige delen.

Og så er det cachen. Her møter vi igjen noe vi bare navnga i kapittel 7.

Cache-koherens, og hva den koster

Kapittel 7 sa at hver kjerne har sin egen L1-cache, at to kjerner derfor kan ha hver sin kopi av samme cachelinje, og at maskinvaren rydder opp. Vi lovet å si hva det koster.

Det koster dette: når én kjerne skriver til en linje, må alle de andre kjernenes kopi av den linja merkes ugyldig. Neste gang en av dem leser, bommer den — og må hente på nytt.
Og nå det som overrasker. Cachelinja er på 64 byte. Den dekker seksten tall på fire byte. Hvis to kjerner skriver til to *forskjellige* tall som tilfeldigvis ligger i den samme linja, får de full koherenstrafikk likevel — selv om de aldri rører hverandres data.
Maskinvaren holder nemlig styr på linjer, ikke på tall. De to kjernene kaster kopien fram og tilbake mellom seg, og hver runde koster en tur ut av L1.
Fenomenet heter *falsk deling*, og det er en av de få gangene et parallelt program blir *tregere* enn det var med én tråd. Fiksen er å flytte de to tallene fra hverandre i minnet, slik at de havner i hver sin linje. Legg merke til hva slags fiks det er: du endrer ikke hva programmet regner ut, bare hvor tallene ligger — nøyaktig som med lokaliteten i kapittel 7.

8.4 Når oppgaven ikke ligner på et program

Veggen: vi avsluttet avsnitt 8.2 med et regnskap. Mesteparten av arealet i en moderne CPU-kjerne går til bokholderi — prediksjon, spekulasjon, omsortering — og ingenting av det regner ut noe som helst. Det finnes bare for å lete etter arbeid som kan gjøres parallelt, i kode som er skrevet som om alt skulle skje etter hverandre.

Så hva om du vet at arbeidet er parallelt på forhånd?

8.4.1 Åtte millioner like oppgaver

Skjermen foran deg har kanskje 3840 ganger 2160 piksler. Det er rundt åtte millioner.

Skal bildet gjøres litt lysere, er jobben å gange hver piksel med samme tall. Åtte millioner ganginger. De er helt like, og — dette er det avgjørende — de er helt uavhengige av hverandre. Piksel nummer fire millioner bryr seg ikke om hva piksel nummer tre ble.

Sammenlign med det CPU-en driver med. Der sitter et maskineri på hundre instruksjoner og leter etter to som ikke henger sammen. Her har du åtte millioner som garantert ikke gjør det, og du visste det før du begynte.

Da er hele letemaskineriet bortkastet.

8.4.2 Så river man det ut

Tanken bak en grafikkprosessor — en **GPU** — er å fjerne alt det CPU-en bruker plassen sin på, og bruke arealet på regneenheter i stedet.

Ut går forgreningsprediktoren, spekulasjonen, det store omsorteringsvinduet og de dype cachene. Inn kommer noen tusen små, enkle regneenheter som alle gjør det samme på hver sine data, styrt av én felles kontrollenhet.

Her er bildet som pleier å sitte. En CPU er åtte dyktige håndverkere som hver kan hele faget og kan løse et problem de aldri har sett før.

En GPU er en skoleklasse på tusen elever som alle har fått den samme regneoppskriften, og som hver regner ut sin egen rad av den samme tabellen. Læreren sier «nå ganger alle med tre», og tusen svar kommer ut samtidig. Alle gjør det *samme*, men hver på sine *egne* tall.

Skal du bygge et hus, vil du ha håndverkerne. Skal du fylle inn en tabell med en million rader, vil du ha klassen.

	CPU	GPU
Antall regneenheter	Noen få, store	Tusenvis, små
Bruker arealet på	Bokholderi: prediksjon, spekulasjon, omsortering	Regning
Egen kontrollenhet per enhet	Ja	Nei — mange deler én
Bra på	én oppgave som må bli fort ferdig, med mange hopp og valg	Mange like oppgaver uten valg
Skjuler ventetid ved å	Cache — holde data nær	Bytte til en annen oppgave mens den venter
Dårlig når	Det er åtte millioner like småjobber	Oppgavene er ulike, eller henger sammen

Tabell 8.4 To måter å bruke det samme silisiumet på. Ingen av dem er bedre; de er svar på hver sin slags oppgave.

Legg særlig merke til nest siste rad, for den er en gammel kjenning i ny drakt.

CPU-en løser ventingen på minnet med cache: hold dataene nær, så blir turen kort. GPU-en gjør noe helt annet. Den har tusenvis av oppgaver liggende, og når en av dem må vente på minnet, bytter den bare til en annen som er klar. Ventetiden forsvinner ikke — den fylles med annet arbeid.

Det er latens mot gjennomstrømning igjen, fra kapittel 7. CPU-en prøver å gjøre ventetiden kort. GPU-en bryr seg ikke om at den er lang, så lenge noen alltid har noe å gjøre.

«GPU-en er raskere enn CPU-en» er ikke et svar

Dette er det vanligste rotet, og nå har du verktøyet til å unngå det.
En GPU er ikke raskere. En GPU-kjerne er faktisk *tregere* enn en CPU-kjerne — den er enklere, den har mindre cache, og den har ingen av triksene fra avsnitt 8.2.
Det GPU-en har, er mange av dem. Og «mange» hjelper bare hvis oppgaven din består av mange like, uavhengige biter.
Prøver du å kjøre et vanlig program på en GPU — et med `if`-er, med funksjonskall, med steg som må skje i rekkefølge — blir det ikke raskere. Det blir mye tregere.
Spørsmålet er aldri «hva er raskest». Spørsmålet er «hvordan ser oppgaven min ut».

Hvor ville du lagt disse?

Fire oppgaver. Avgjør for hver av dem om den hører hjemme på CPU eller GPU, og si hvorfor — med ordene fra tabell 8.4, ikke med «den er størst».

1. Legge et filter på et bilde med tolv megapiksler.
2. Finne ut om et tall er et primtall ved å prøve alle delere etter tur.
3. Sortere en liste med en million navn alfabetisk.
4. Simulere hvordan temperaturen fordeler seg i en metallplate, der hver rute påvirkes av naboene sine.

Nummer 3 og 4 er de interessante. Begge har et svar, og begge har et «ja, men».

8.4.3 Og så den tredje typen: kretser som bare gjør én ting

Det finnes en grad til av spesialisering, og den sitter i maskinen din akkurat nå.

Tenk på hva som skjer når du ser en film på telefonen. Videoen er komprimert, og hver eneste bildrute må pakkes ut — det er tung regning, og den må gjøres tjuefire ganger i sekundet i flere timer på et batteri.

Hvis CPU-en gjorde den jobben, ville telefonen bli varm og batteriet tomt på en times tid. Det gjør den ikke. Grunnen er at det sitter en egen liten krets på brikka som ikke kan gjøre noe annet enn å pakke ut akkurat det videoformatet. (Det sitter en tilsvarende krets for å pakke *sammen* video også — den brukes når du filmer.)

Den kan ikke programmeres. Den kan ikke brukes til noe annet. Og den er flere hundre ganger mer strømgjerrig enn CPU-en på nøyaktig den ene oppgaven.

Det kalles en **fastfunksjonskrets**, eller en akselerator. Og du har akkurat sett hvorfor den vinner, i avsnitt 8.2: den har ikke noe bokholderi. Den henter ikke instruksjoner, den tyder dem ikke, den gjetter ikke på hopp, den sorterer ingenting. Den gjør bare det ene den er bygget for.

Den samme avveiningen hele veien

Det er én linje gjennom hele denne seksjonen, og den er verdt å se samlet.
Fleksibilitet koster. Jo flere forskjellige ting en krets kan gjøre, jo mer maskineri må den ha for å finne ut hva den skal gjøre akkurat nå. Og det maskineriet koster areal, strøm og varme. Det regner ikke ut noe.
CPU-en kan alt, og betaler mest. GPU-en kan én *slags* ting, og betaler mindre. Fastfunksjonskretsen kan én ting, og betaler nesten ingenting.
Det er derfor det står tre forskjellige slags regneenheter i databladet på første side. De er ikke tre forsøk på det samme. De er tre punkter på den samme kurven.
Og i neste avsnitt kommer det fjerde.

8.5 Brikken for nevrale nett

Veggen: den siste linja i databladet. «Nevral motor, 16 kjerner.» Hvorfor trenger en telefon enda en slags regneenhet?

Dette avsnittet er kort med vilje. Hele historien står i appendiks G, og den er verdt å lese — men her tar vi bare det du trenger for å lese databladet ferdig.

8.5.1 Hva et nevralt nett regner ut

Det korte svaret er at det ganger og legger sammen, og at det gjør det veldig mange ganger.

Et nevralt nett er bygget av enheter som hver gjør det samme: de tar inn en håndfull tall, ganger hvert av dem med en vekt, legger produktene sammen, og sammenligner summen med en terskel.

Stopp ved den setningen, for du har sett den før.

I kapittel 5 bygde du AND og OR. Se på dem på nytt med de nye ordene: AND gir 1 når *begge* inngangene er 1, altså når summen av dem er minst 2. OR gir 1 når summen er minst 1. Begge ganger er vektene de samme — én på hver inngang — og det eneste som skiller de to portene, er hvor terskelen ligger.

En port kan altså *beskrives* som en vektet sum med en terskel. Merk ordet: dette er en måte å skrive sannhetsverditabellen på, ikke en påstand om at det sitter en summeringskrets inne i porten — den er fortsatt de fire transistorene fra kapittel 6.

Men beskrivelsen er presis, og den peker rett framover. Forskjellen på porten og den nevrale enheten er at portens vekter er bestemt én gang for alle, mens nettets kan stilles inn.

Det er hele avstanden mellom kapittel 5 og en KI-brikke, og appendiks G går den strekningen ordentlig.

8.5.2 Hvorfor det gir en egen brikke

Et nett har millioner eller milliarder av slike vekter. Å kjøre det én gang er millioner av ganginger og addisjoner — og de er alle sammen uavhengige av hverandre, akkurat som pikslene.

Det peker mot en GPU, og GPU-er brukes mye til dette. Men man kan gå ett skritt lenger, og grunnen er et tall fra kapittel 7 som du allerede kjenner formen på.

Å *hente* et tall koster mye mer enn å regne med det. I appendiks G settes forholdet opp ordentlig, og det er på flere tusen ganger. Når regnestykket er så enkelt som «gang og legg til», er det altså ikke regningen som koster. Det er transporten.

En brikke bygget for dette gjør derfor to ting. Den har veldig mange, veldig små gangeenheter. Og den sørger for at tallene flytter seg så kort som mulig: de sendes videre fra enhet til enhet i stedet for å hentes fra minnet hver gang.

Den kalles en **NPU**, en nevral prosesseringsenhet, og i databladet heter den «nevral motor».

Hvorfor den sitter i telefonen og ikke bare i skyen

Telefonen kunne sendt alt til en tjener — en server — og fått svaret tilbake. For noen ting gjør den det.
Men tre ting taler imot. Det går ikke uten nett — og at tastaturet gjetter neste ord, eller at telefonen finner tekst i et bilde, må virke i flymodus. Det tar tid å sende fram og tilbake, og et kamera som skal kjenne igjen ansikter mens du sikter, har noen millisekunder på seg. Og dataene forlater telefonen, noe som ikke alltid er greit.
Derfor sitter brikka der den sitter.

Vil du vite hvorfor brikka ser slik ut? Appendiks G

Dette avsnittet har sagt *at* det finnes en egen brikke, og grovt sett hvorfor. Appendiks G forklarer *hvordan den ble som den ble*, og det er det lengste appendikset i boka — 23 sider, og det stedet flest tråder fra resten av boka møtes.
Der står:

- veien fra porten i kapittel 5 til perseptronen, via vekter som kan stilles inn
- hvorfor én enhet ikke klarer XOR, og hva som måtte til
- regnskapet som viser at papirarbeidet i en vanlig prosessor koster mer enn selve jobben
- hva det faktisk koster å flytte et tall, i energi — og hvordan hele brikka formes av det ene forholdet
- hvorfor færre siffer gir mye mindre krets, og hvorfor KI regner med åtte bit der du lærte seksten og trettito
- hvorfor en telefon på tre og et halvt watt må løse oppgaven helt annerledes enn et grafikkort på tusen

- og hva som skjer hvis man gir opp den digitale abstraksjonen helt og lar Ohms lov gjøre gangingen

Ikke pensum. Men hvis du har lurt på hva som egentlig ligger bak ordet i databladet, er det der svaret står.

8.6 Grensesnittene mot verden

Veggen: alt vi har sett på til nå har vært inne i brikka. Men maskinen er ikke alene. Den har et tastatur, en skjerm, en disk og et nettverkskort, og den kjører tjue programmer samtidig som ikke skal kunne rote i hverandres ting.

Hvordan henger det sammen med hent–utfør-sløyfa fra kapittel 3? Den sløyfa gjør én ting: henter en instruksjon, utfører den, og gjentar. Det er ikke åpenbart hvor et tastetrykk skulle få plass.

8.6.1 Hvordan maskinen vet at du trykket på en tast

Her er problemet, og det er verdt å tenke over i noen sekunder før du leser svaret.

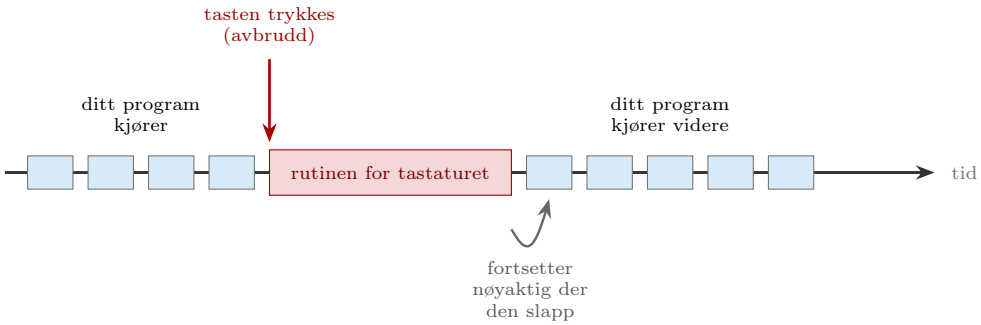
Prosessoren kjører et program. Du trykker på en tast. Hvordan får prosessoren vite det?

Den enkleste løsningen er at programmet spør. Hundre ganger i sekundet stikker det innom og sjekker om det har kommet et tastetrykk.

Det virker, og det brukes noen steder. Men det er to ting galt med det. Hvis det går lenge mellom hver sjekk, blir maskinen treg å bruke. Og hvis den sjekker ofte, går det med masse tid til å spørre om noe som nesten alltid ikke har skjedd.

Det er som å ta opp mobilen hvert femte minutt for å se om noen har skrevet. Enten sjekker du for sjelden, og svarer for sent — eller du sjekker så ofte at du ikke får gjort noe annet.

Løsningen er den samme som på mobilen: slå på varsling, og la beskjeden komme til deg.



Figur 8.2 Et avbrudd. Programmet merker ingenting — maskinen legger unna alt den holdt på med, kjører rutinen for den enheten som ringte på, og setter alt tilbake slik det var.

Et **avbrudd** er en ledning. Tastaturet trekker i den når noe har skjedd.

Og her er det som gjør det enkelt å forstå: prosessoren sjekker den ledningen én gang per instruksjon. Det koster ingenting, for det er bare å se på et signal som allerede ligger der. Ser den at ledningen er trukket, gjør den følgende, i denne rekkefølgen:

Den gjør ferdig instruksjonen den holder på med. Den lagrer unna programtelleren og registrene, slik at den husker nøyaktig hvor den var. Den slår opp i en tabell hvilken rutine som hører til akkurat den enheten, og setter programtelleren dit. Når rutinen er ferdig, legges alt tilbake, og programmet fortsetter som om ingenting hadde skjedd.

Det er altså et funksjonskall maskinvaren tvinger igjennom. Programmet ditt kalte ingenting, og programmet ditt merker ingenting.

Hvorfor dette er en av de viktigste ideene i faget

Uten avbrudd finnes ikke flere programmer samtidig.
Tenk etter hva operativsystemet egentlig gjør når det bytter fra nettleseren til musikkspilleren. Det må ta kontrollen tilbake fra nettleseren. Men nettleseren kjører jo — den kommer aldri til å gi fra seg prosessoren frivillig, og et program som går i evig løkke kommer aldri til å spørre om lov til å slutte.
Så operativsystemet ber om et avbrudd fra en klokkekrets. Noen hundre ganger i sekundet ringer den klokka, operativsystemet får kontrollen i noen mikrosekunder, og det bestemmer hvem som skal få kjøre videre.
Hele multitaskingen hviler altså på en ledning og et tidsur. Det er også dette som gjør at en maskin kan tas ut av en programvarefeil — maskinvaren har en vei inn som programvaren ikke kan stenge.

8.6.2 Og hvem som flytter dataene

Avbruddet sier *at* noe har skjedd. Det gjenstår å flytte selve dataene.

For en tast er det én byte, og det er ikke noe problem. For en fil på disken er det kanskje hundre megabyte.

Hvis prosessoren skulle flyttet dem selv, byte for byte, ville den brukt all sin tid på det. Og den ville gjort det i disken sitt tempo. Det så du i kapittel 7: millioner av klokkesykluser med venting.

Derfor gjør den det ikke. Det sitter en egen liten krets som bare flytter data mellom en enhet og minnet, uten å involvere prosessoren. Prosessoren sier «hent hundre megabyte fra disken og legg dem her», og går videre med noe annet. Når kretsen er ferdig, sender den et avbrudd.

Det heter **DMA**, direkte minnetilgang. Og legg merke til hva slags løsning det er: nok en spesialisert krets som gjør én ting, slik at den generelle maskinen slipper. Samme mønster som i avsnitt 8.4.

Hvordan prosessoren snakker med en enhet

Det er én ting til som overrasker, og den er kort å si.
Det finnes som regel ingen egne instruksjoner for «skriv til skjermen» eller «les fra disken». I stedet får hver enhet tildelt noen adresser i det vanlige adresserommet, og prosessoren bruker helt vanlige lese- og skriveinstruksjoner på dem.
Skriver du til adresse X, havner verdien ikke i minnet — den havner i en kontrollkrets på skjermkortet. Leser du fra adresse Y, får du statusen til disken.
Det kalles *minneavbildet inn- og utdata*. Hele poenget er at maskinen slipper å ha to sett instruksjoner. De samme `LDA` og `STA` du lærte i kapittel 3 gjør begge jobbene; det er bare adressen som avgjør hvor verdien faktisk går.

8.6.3 Virtuelt minne: hvorfor to programmer ikke kan rote i hverandre

Nå skal vi lukke hullet i minnehierarkiet fra kapittel 7.

Begynn med et problem du kan se med en gang.

Gå tilbake til rommet med hundre postkasser i kapittel 3. Nå kjører to programmer der inne. Begge er skrevet av hver sin person, og ingen av dem visste om den andre. Begge bruker postkasse 42 til å lagre en mellomregning.

Hva skjer?

Det ene programmet skriver i kasse 42. Det andre leser den, får et tall det ikke kjenner igjen, og regner feil. Eller verre: et program med dårlige hensikter leser kasse 42 med vilje, og finner passordet ditt der.

I LMC-rommet finnes ingen løsning på dette. Enhver instruksjon kan nevne enhver adresse.

Løsningen er å la programmene lyve om hvor de er.

Hvert program får sitt eget sett adresser som begynner på null og går oppover, og programmet tror det har hele maskinen for seg selv. Men adressene er ikke ekte. Hver gang programmet nevner en adresse, oversetter maskinvaren den til en annen adresse, et helt annet sted i det fysiske minnet.

Tenk deg at det ikke er étt rom med hundre postkasser, men en hel etasje med mange slike rom. Begge programmene sier «kasse 42», men det ene blir sendt til rom 3 og det andre til rom 9. De bruker det samme tallet, og de havner i hver sin kasse.

Det kalles **virtuelt minne**. Adressene programmet bruker er *virtuelle*; de maskinen faktisk slår opp i, er *fysiske*.

Hva du får	Hva det betyr
Isolasjon	Program A *kan ikke uttrykke* en adresse som tilhører program B. Ikke fordi det er forbudt — fordi det ikke finnes et tall som peker dit.
Enkel programmering	Hvert program kan starte på adresse null hver gang, uansett hvor i maskinen det faktisk havner.
Mer minne enn du har	Sider som ikke er brukt på en stund, kan legges ut på disken. Her kobles det til minnehierarkiet: dette er laget under RAM.
Deling når du vil ha den	To programmer kan få samme fysiske side avbildet inn hos begge, og da deler de med vilje i stedet for ved uhell.

Tabell 8.5 De fire tingene virtuelt minne gir. Den første er den viktigste, og den er grunnen til at mekanismen finnes i det hele tatt.

Et par ord til, så du kjenner dem igjen. Oversettelsen skjer ikke adresse for adresse — det ville krevd en tabell like stor som minnet. Minnet deles i biter på typisk fire kilobyte som heter **sider**, og oversettelsen skjer per side. Tabellen som holder styr på det heter *sidetabellen*. Og leter programmet etter en side som er lagt ut på disken, kalles det en *sidefeil*; da henter operativsystemet den inn igjen.

Og så det som gjør at det går an i praksis. Oversettelsen må skje ved *hver eneste* minnetilgang, altså milliarder av ganger i sekundet. Et oppslag i en tabell i minnet for hvert oppslag i minnet ville doblet arbeidet.

Så maskinen gjør det du nå forventer at den gjør: den setter en liten, rask cache foran. Den heter TLB, og den holder de siste oversettelsene. Treffraten er over 99 prosent, av nøyaktig samme grunn

som cachen virker i det hele tatt — lokalitet.

Nok et lag du ikke visste var der

Tell etter hvor mange usynlige lag som nå ligger mellom `x = minliste[5]` i Python og de transistorene som faktisk bytter.
Tolkeren oversetter linja til instruksjoner. Adressen instruksjonen nevner er virtuell og må oversettes. Oversettelsen slås opp i en cache du ikke visste fantes. Den fysiske adressen slås opp i L1, kanskje L2, kanskje L3. Instruksjonen selv ble kanskje utført ut av rekkefølge, kanskje spekulativt, kanskje samtidig med tre andre.
Ingen av lagene endrer svaret. Alle endrer tiden.
Og hvert eneste av dem er et eksempel på det kapittel 1 kalte informasjonsskjuling: et lag som gjør jobben sin og lar deg slippe å vite om det.
Neste avsnitt handler om hva som skjer når ett av dem lekker.

Hva vi har tatt, og hva vi har latt ligge

Vi har tatt begrepet og poenget: illusjonen, isolasjonen, og at oversettelsen koster et oppslag. Det er det du trenger, og det holder til planeten.
Mekanikken — hvordan sidetabellen er bygget, og hva som skjer i detalj ved en sidefeil — hører like mye til operativsystemfaget som til dette, og du møter den igjen der. Englander kapittel 19.7 har den hvis du vil ha den nå.

8.7 Når abstraksjonen lekker

Veggen: du har nå sett maskinen gjøre arbeid den kanskje må kaste, og du har sett at den later som om ingenting skjedde når den kaster det. Og du har sett et minne som ligger og husker hva som nettopp ble lest.

Sett de to tingene ved siden av hverandre, og det finnes et hull.

Dette er det siste nye i kapitlet, og det er ikke ny maskinvare. Alt som trengs, har du allerede.

8.7.1 To setninger fra tidligere i boka

Den første sto i kapittel 7, i avsnittet om at cachen er usynlig:

> «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.*»

Den andre sto i avsnitt 8.2, om spekulativ utføring:

> «Registrene settes tilbake. Resultatene forkastes. Programmet ditt får nøyaktig de samme svarene som det ville fått på en maskin som aldri spekulerte.»

Begge setningene er sanne. Og lest sammen er de et sikkerhetshull.

For hva betyr egentlig «resultatene forkastes»? Det betyr at *registrene* ryddes. Men mens maskinen spekulerte, leste den fra minnet — og den lesingen gikk gjennom cachen. Cachen ble ikke ryddet. Den husker fortsatt hvilken linje som ble hentet inn.

Og en cachelinje som ligger der, er målbar. Det vet du, for du målte det selv i kapittel 7: forskjellen mellom treff og bom var hele forklaringen på de trettito gangene.

8.7.2 Hvordan hullet oppstår

Her er mønsteret, i fem steg. Det er verdt å lese dem sakte, for hvert enkelt steg er noe du allerede har sett.

Steg 1. Et program har en sjekk: «er denne indeksen innenfor lista?» Sjekken er riktig skrevet, og den virker.

Steg 2. Sjekken er et betinget hopp, og maskinen gjetter på utfallet. Angriperen har først gjort tusen helt lovlige oppslag gjennom den samme sjekken, slik at prediktoren har lært seg å gjette «innenfor». Så gir hun en indeks som ligger langt utenfor — og maskinen gjetter som den er opplært til.

Steg 3. Mens den gjetter, utfører den oppslaget. Den leser en verdi den egentlig ikke skulle lest, og bruker den som *adresse* for et oppslag til.

Her er hele knepet, og det er verdt en setning for seg: adressen det andre oppslaget bruker, bestemmes av den hemmelige verdien. Er verdien 65, hentes linja som hører til 65. Er den 200, hentes linja som hører til 200.

Steg 4. Noen sykluser senere er sammenligningen ferdig, og gjetningen var feil. Maskinen rydder opp. Registrene settes tilbake, resultatene forkastes, og programmet oppfører seg helt riktig — oppslaget blir aldri gjort.

Så langt har alt gått helt etter planen. Programmet er korrekt. Maskinen er korrekt. Ingen regel er brutt.

Steg 5. Men cachen holder fortsatt på linja fra steg 3.

Angriperen trenger nå ikke lese verdien. Det får hun ikke lov til. Men hun kan måle tiden på en rekke helt lovlige oppslag — og det ene som går raskere enn de andre, er det som allerede lå i cachen.

Og siden linja er valgt av den hemmelige verdien, forteller den hvilken verdi det var. Går oppslag nummer 65 raskest, var verdien 65.

én byte av gangen, tusenvis av ganger i sekundet.

Les dette som forklaring, ikke som advarsel

Før vi går videre: hullet ble offentliggjort i 2018, og maskinen din er for lengst lappet. Du trenger ikke gjøre noe. Resten av avsnittet handler om hvorfor det er interessant, ikke om hva du bør passe deg for.

Det er ingen feil her, og det er hele poenget

Dette er det som gjør Spectre interessant, og det er grunnen til at det står i denne boka og ikke bare i et sikkerhetskurs.
Det er ingen programmeringsfeil. Sjekken er riktig skrevet. Det er ingen feil i prosessoren heller — den gjør nøyaktig det den ble designet for å gjøre, og den gir alltid riktige svar.
Lekkasjen ligger i *avstanden mellom to beskrivelser av den samme maskinen.* Modellen sier: instruksjonene utføres én etter én, og spekulasjon er usynlig. Virkeligheten sier: de utføres hulter til bulter, og spekulasjonen etterlater et spor i tiden.
Begge beskrivelsene er riktige. De er bare riktige om forskjellige ting — den ene om hvilke svar du

får, den andre om hvor lang tid du bruker på å få dem. Og den som ikke har lov til å se svarene, kan likevel se tiden.
Kapittel 1 nevnte ordet *abstraksjonslekkasje*: at detaljer fra et lavere lag siver opp gjennom et lag som skulle skjult dem. Dette er en lekkasje av akkurat den typen, og den siver opp hele veien fra mikroarkitekturen til nettleseren din.

8.7.3 Hva man gjør med noe slikt

Det ubehagelige med denne typen hull er at det ikke lar seg fikse med en oppdatering på vanlig måte. Problemet sitter i selve ideen om å spekulere, og den ideen er grunnen til at maskinen er rask.

Derfor er svaret blitt tredelt, og alle tre er kompromisser.

Maskinvaren. For flere av variantene er nyere brikker designet slik at spekulasjonen ikke får etterlate spor på tvers av grensene som betyr noe. Det tar år å få ut i verden, fordi det krever nye brikker — og for akkurat den varianten vi har beskrevet, finnes det ingen generell maskinvarefiks. Den må håndteres i koden.

Programvaren. Operativsystemer og nettlesere ble endret for å gjøre angrepet upraktisk — blant annet ved å gjøre klokkene som programmer har tilgang til, mindre nøyaktige. Måler du tid grovt nok, forsvinner forskjellen på treff og bom i støyen.

Og prisen. Flere av tiltakene koster ytelse, i enkelte tilfeller flere prosent. Det er en tydelig påminnelse om at sikkerhet og fart er to ting som trekker i hver sin retning, og at noen må velge hvor grensen går.

Maskinen din er i orden

La oss si det rett ut, siden dette er den typen avsnitt som kan gjøre en urolig. Hullet ble offentliggjort i 2018. Alle maskiner og nettlesere som har fått oppdateringer siden, har fått tiltakene. Du trenger ikke gjøre noe.
Grunnen til at det står her, er ikke at du skal passe deg. Det er at Spectre er det beste eksempelet som finnes på noe dette faget handler om. Lagene i en datamaskin er ekte og nyttige, men de er ikke tette. Den som forstår laget under, kan se ting den som bare kjenner laget over ikke vet finnes. Det er også derfor du har brukt et semester på å gå ned gjennom lagene.

Hvor ellers kan tid røpe noe?

Spectre bruker tid som kanal. Det finnes flere av dem, og prinsippet er alltid det samme: noe som ikke skulle være synlig, blir målbart gjennom en bivirkning.
Tenk over disse tre, og avgjør for hver om det er en reell lekkasje:

1. En innloggingsside som svarer raskere når brukernavnet ikke finnes, enn når det finnes men passordet er feil.
2. Et program som bruker målbart mer strøm når det regner på store tall enn på små.
3. En nettside som laster raskere andre gang du besøker den.

Den tredje er den mest interessante. Den er helt vanlig oppførsel — og den har vært brukt til å finne ut hvilke nettsteder noen har besøkt.

8.8 Hvor skal grensen gå?

Veggen: vi har nettopp sett et lag lekke. Da er det på sin plass å spørre hvor lagene egentlig bør ligge — og det finnes én grense i en datamaskin som har vært diskutert hardere enn alle andre.

Det er grensen mellom instruksjonssettet og maskineriet som utfører det. Kapittel 4 kalte dem instruksjonssettarkitekturen og mikroarkitekturen, og sa at den første er løftet og den andre er hvordan løftet innfris.

Spørsmålet er hvor mye løftet skal inneholde.

8.8.1 To svar

Det ene svaret: la instruksjonene gjøre mye. Hvis en vanlig oppgave krever ti enkle instruksjoner, hvorfor ikke lage én instruksjon som gjør hele jobben? Da blir programmet kortere, det tar mindre plass i minnet, og den som skriver assembler får mindre å skrive.

Det er **CISC**, *complex instruction set computer*. Intels x86-familie er det store eksempelet, med flere hundre instruksjoner i grunnformene sine — noen av dem kan kopiere en hel datablokk eller lete gjennom en tekststreng, helt alene.

Og det var et godt svar da det ble gitt. Minne var dyrt, og et program som tok halvparten så mye plass var en reell gevinst.

Det andre svaret: la instruksjonene gjøre lite, men fort. Hold instruksjonene få, like lange og enkle. La hver av dem gjøre én ting. Gi maskinen mange registre, slik at den slipper å gå til minnet hele tiden.

Det er **RISC**, *reduced instruction set computer*. ARM-familien er det store eksempelet, og den sitter i telefonen din og i stadig flere PC-er.

8.8.2 Hvorfor RISC vant der det telte

Nå kan du se grunnen selv, for den står i avsnitt 8.2.

Pipelining krever at alle trinn tar like lang tid. Det krever at instruksjonene er omtrent like store jobber.

En CISC-instruksjon som kopierer en datablokk, er ikke en jobb som passer inn i et EX-trinn. Den kan ta tusen sykluser. Legger du den i en pipeline, stopper hele samlebåndet mens den holder på — nøyaktig det problemet kapittel 7 brukte en halv seksjon på.

RISC-instruksjoner er alle sammen omtrent like store. De passer i samme trinn. De har samme lengde, så IF-trinnet vet hvor neste instruksjon begynner uten å tyde den forrige. Alt det som gjorde pipelining vanskelig, ble designet bort.

Men x86 forsvant jo ikke

Og her kommer vendingen, som er det beste i hele denne seksjonen.
Du bruker sannsynligvis en x86-maskin eller en ARM-maskin akkurat nå, og begge er raske. Hvordan kan en CISC-prosessor være rask, hvis CISC er dårlig egnet for pipelining?
Svaret er at den ikke *er* en CISC-prosessor på innsiden.
En moderne x86-prosessor tar imot CISC-instruksjonene og oversetter dem — i maskinvare, i dekodetrinnet — til små, jevnstore mikro-operasjoner. De fleste blir til én hver; de sammensatte blir

til flere. Det er mikro-operasjonene som går gjennom pipelinen, som videresendes, som spekuleres på og sorteres ut av rekkefølge.
Innsiden arbeider altså RISC-aktig: pipelinen ser bare jevne, enkle biter, akkurat slik den trenger. Utsiden er fortsatt x86, fordi all programvaren i verden er skrevet for x86.
Dette er ikke et kompromiss. Det er et abstraksjonslag, nøyaktig som alle de andre i denne boka: et grensesnitt som holdes uforandret utad, mens alt bak det er skiftet ut. Det er den samme ideen som kapittel 1 åpnet med — at man kan forbedre ett nivå uten å røre de andre — og her er den brukt til å redde førti år med programvare.
Prisen er at oversettelsen koster transistorer og strøm. Hvor mye det betyr i det store regnskapet, er omdiskutert — ARMs forsprang på batteridrevne maskiner handler først og fremst om hva brikkene er designet for.

Så hvem vant? Ingen, og begge. I praksis har leirene lånt av hverandre så lenge at ordene er mer nyttige som historie enn som beskrivelse. ARM har fått flere og mer sammensatte instruksjoner. x86 har fått en RISC-maskin på innsiden. Det som står igjen, er avveiningen — og den er den samme som i avsnitt 8.4: hvor mye skal én enhet kunne, og hva koster det å kunne mye?

En presisering om Little Man Computer

LMC har elleve instruksjoner, og det er fristende å kalle den en RISC-maskin. Det stemmer ikke helt.
LMC har få instruksjoner fordi den er en undervisningsmodell som skal få plass på en tavle. En RISC-maskin har få instruksjoner fordi det gjør dem raske å utføre og lette å pipelinere — og den har til gjengjeld mange registre, gjerne 32, mens LMC har akkumulatoren og ingenting annet.
LMC er en fin kontrast til x86 når du vil vise hvor lite som trengs. Men den er ikke et eksempel på RISC-filosofien.

8.9 Hele stabelen

Nå er vi framme.

Boka åpnet med en påstand i kapittel 1. En datamaskin er lag på lag. Hvert lag skjuler det under seg, og du kan arbeide på ett lag av gangen uten å vite hva de andre driver med.

Siden har vi gått nedover. Tall, instruksjoner, arkitektur, porter, transistor — og så opp igjen gjennom minnet, pipelinen og den moderne maskinen.

Det du aldri har sett, er hele veien på én gang.

Her er den. Én linje Python, fulgt helt ned til de transistorene som faktisk bytter, og tilbake opp igjen med svaret.

```
svar = a + b
```

Den samme linja sto i kapittel 3, i figur 3.1 på side 56. Da stoppet vi ved LMC. Nå går vi hele veien.

8.9.1 Nedstigningen

Lag 1 — det du skrev. En linje i en tekstfil. Den sier hva du vil, ikke hvordan. Den sier ingenting om hvor `a` ligger, hvor mange bit tallet har, eller hva som skal skje hvis summen blir for stor. Det er hele vitsen med laget: du får slippe.

Lag 2 — oversettelsen. Noe må gjøre linja om til ordrer maskinen forstår. Det er språkstigen fra kapittel 3. Her blir `svar = a + b` til tre ordrer: hent `a`, legg til `b`, lagre i `svar`. Du kjenner dem som `LDA`, `ADD` og `STA`.

Lag 3 — ordren blir et tall. Kapittel 4 viste at en instruksjon ikke er en tekst, men et tall delt i felter. Noen bit sier hva som skal gjøres, resten sier hvor. `ADD` er ikke bokstavene A, D og D — det er et bitmønster i et opkodefelt, og det ligger i minnet på nøyaktig samme måte som tallene det skal regne med.

Lag 4 — tallene er også bit. Kapittel 2 gjorde den jobben. Verdiene i `a` og `b` er bitmønstre, negative tall er toerkomplement, og «for stor sum» har en helt konkret betydning: menten fra det øverste trinnet falt utenfor.

Lag 5 — maskineriet henter og utfører. Programtelleren peker på instruksjonen. Adressen går til MAR, instruksjonen kommer tilbake via MDR og havner i IR. Dekoderen tyder opkodefeltet og slår på de riktige styreledningene. Akkumulatoren og verdien fra minnet går inn i ALU-en. Dette er figur 4.7 på side 96, og det er hent–utfør-runden fra kapittel 3.

Lag 6 — ALU-en er porter. Kapittel 5 bygde den. Addisjonen gjøres av fulladdere satt i rekke, menten vandrer fra trinn til trinn, og hver fulladder er XOR, AND og OR koblet sammen. Ingen regner. Det er bare ledninger som fører strøm eller ikke.

Lag 7 — portene er transistorer. Kapittel 6 tok det siste skrittet. Hver NAND er fire transistorer, to i serie og to i parallell. Ettallet kommer fra forsyningen, ikke fra inngangen, og hver port lager et friskt, skarpt signal.

Lag 8 — transistoren er en bryter som flytter ladning. Og her er bunnen. Hver gang et signal skifter, må et bittelite kar fylles eller tømmes. Det tar rundt fem pikosekunder, og det koster en slurk energi som blir til varme.

Det er hele veien. Åtte lag fra en linje tekst til ladning som flytter seg i silisium, og ikke ett hull.

8.9.2 Og opp igjen

Summen står nå som spenninger på utgangen av den siste fulladderen i rekka. Snaut ett nanosekund senere er alle mentene ferdige, og tallet er stabilt. Det var kapittel 6 sin kritiske vei, og der regnet du den til 960 pikosekunder.

Ved neste klokkeflanke fanges verdien i akkumulatoren. `STA` sender den til minnet, der den legges i den cellen Python har bestemt skal være `svar`. Neste gang programmet ditt nevner `svar`, leses den opp igjen, og du får et tall du kan skrive ut.

Du spurte om `a + b`. Du fikk `a + b`. Ingen av de åtte lagene gjorde seg bemerket.

8.9.3 Og alt det som skjedde på siden

Nedstigningen over er sann, men den er den ryddige versjonen. Her er hva som foregikk samtidig — og hvert punkt er et kapittel du har lest:

- Instruksjonen lå antakelig i L1-instruksjonscachen. Henting tok fire sykluser i stedet for 240.
- Adressen den nevnte var virtuell og måtte oversettes. Oversettelsen lå i TLB-en.
- Verdiene i `a` og `b` lå i cachen, fordi de nettopp var brukt.
- Instruksjonen var én av fire inne i pipelinen samtidig.

- Resultatet ble sannsynligvis rukket rett over bordet til neste instruksjon, i stedet for å gå veien om akkumulatoren.
- Sto det et `if` i nærheten, hadde maskinen alt gjettet hvilken vei det gikk, og regnet videre på gjetningen.
- Og hele tiden lå det en ledning klar til å avbryte alt sammen, hvis du trykket på en tast.

Ingen av dem endret svaret. Alle sammen endret tiden.

8.9.4 Hva det koster, målt

Og så det ærlige regnskapet, siden vi har en stoppeklokke.

Kapittel 1 sa at abstraksjon er et kompromiss: at høyere lag forenkler, men kan skjule ting som er viktige for ytelse. Nå kan vi sette et tall på det.

Samme addisjon, skrevet på tre måter	Tid	Klokkesykluser
Maskinen, fire uavhengige addisjoner samtidig	0,062 ns	0,25
Maskinen, addisjoner som venter på hverandre	0,249 ns	1
`svar = a + b` i Python	5,9 ns	ca. 24

Tabell 8.6 Prisen på abstraksjonen, målt på samme maskin. Alle tre regner ut det samme.

Først et ord om den øverste raden, for 0,25 sykluser ser umulig ut. En syklus *er* den minste takten — ingenting tar en fjerdedels syklus. Tallet betyr at maskinen blir ferdig med fire addisjoner i hver syklus, altså at det går én fjerdedels syklus med per addisjon. Det er superskalaren fra avsnitt 8.2.5, sett fra den andre siden.

Og så poenget: i de 24 syklusene Python-linja bruker, kunne maskinen gjort rundt nitti addisjoner.

Men les tallet riktig. De 24 syklusene går ikke med til å legge sammen to tall — selve addisjonen er én av dem. Resten går med til alt språket gjør for deg. Finne ut hva `a` og `b` er for slags ting akkurat nå. Sjekke at det går an å legge dem sammen. Lage et nytt objekt til svaret, og slå opp hvilken instruksjon som kommer etter denne.

Det er ikke sløsing. Det er tjenester. Det er derfor du kan skrive `a + b` uten å vite om de er heltall, desimaltall eller tekst, og det er derfor du ikke må fortelle maskinen hvor mange bit svaret trenger.

Hele boka i én tabell

Tabell 8.6 er grunnen til at dette emnet finnes.
Hvis du bare kjente det øverste laget, ville de tre radene vært et mysterium — eller verre, de ville vært usynlige. Du ville ikke visst at det var noe å måle.
Hvis du bare kjente det nederste, kunne du forklart hver syklus, men du ville brukt tre uker på å skrive noe som tar en linje i Python.
Poenget har aldri vært at det ene laget er bedre enn det andre. Poenget er at du nå kan bevege deg mellom dem — velge lag etter oppgave, og vite hva valget koster.
Det er det du kan nå, som du ikke kunne for noen måneder siden.

8.10 Lagkartet

Til slutt et oppslag, ment til å slås opp i.

Tabellen under er hele boka på én side. Venstre kolonne er lagene slik boka har gått gjennom dem, øverst og nedover. Høyre kolonne er hvor hvert av dem står.

Den har to rader mer enn nedstigningen i forrige avsnitt. Det er med vilje: operativsystemet og fysikken var ikke med på turen ned gjennom `svar = a + b`, men de hører med i kartet. Ellers er det samme stabel.

Lag	Hva som hører hjemme her	Kapittel
Program	Python, variabler, løkker, funksjoner	1
Operativsystem	Prosess, tråd, avbrudd, virtuelt minne, isolasjon	8
Språk og oversettelse	Språkstigen, assembler, etiketter	3
Instruksjonssett	Instruksjonen som tall, opkode, adressefelt, hent–utfør	3, 4
Mikroarkitektur	Datasti, registre, MAR og MDR, dekoder, pipeline, cache, spekulasjon, flere kjerner	4, 7, 8
Data	Bit, byte, toerkomplement, flyttall, tegn, bilde, lyd	2
Logikk	Port, sannhetsverditabell, adderer, mentekjede	5
Krets	Transistor, CMOS, spenningsområde, kapasitans, portforsinkelse, kritisk vei, klokke	6
Fysikk	Halvleder, doping, felteffekt	appendiks D

Tabell 8.7 Hele boka, lag for lag. Merk at mikroarkitekturen er det tykkeste laget — det er der mesteparten av de siste tre kapitlene ligger.

Og her er den andre veien å bruke kartet: hvilket problem hvert lag løste.

Problemet	Svaret, og hvor det står
Alt må være tall	Binær representasjon, toerkomplement, IEEE 754 (kap. 2)
Mennesker kan ikke skrive bit	Språkstigen, assembler, tolker (kap. 3)
Maskinen må vite hva den skal gjøre	Instruksjonen som tall, dekoderen (kap. 4)
Noe må faktisk regne	Porter, fulladder, mentekjede (kap. 5)
Porter trenger brytere uten fingre	Transistoren, CMOS (kap. 6)
Signaler blir slitne	Spenningsområder og signalgjenoppretting (kap. 6)
Alt må skje i takt	Klokka, kritisk vei (kap. 6)
Minnet er hundre ganger for tregt	Cache og lokalitet (kap. 7)
Maskinen gjør én ting av gangen	Pipeline (kap. 7)
Pipelinen stopper hele tiden	Forwarding, prediksjon, spekulasjon (8.2)
én instruksjon per syklus er et tak	Superskalar, out-of-order, SMT (8.2)
Klokka kan ikke gå fortere	Flere kjerner — og Amdahls grense (8.3)
Fleksibilitet koster strøm	GPU, NPU, fastfunksjonskretser (8.4, 8.5)
Verden utenfor må inn	Avbrudd, DMA, minneavbildet I/O (8.6)
Programmer må ikke rote i hverandre	Virtuelt minne (8.6.3)
Lagene er ikke tette	Spectre, og hva det koster å tette dem (8.7)

Tabell 8.8 Den samme stabelen lest som en rekke problemer og svar. Hvert svar skapte som regel det neste problemet.

8.11 Oppsummering

8.11.1 Begrepene du har fått navn på

8.11.2 Den ene ideen, hele kapitlet

Se tilbake på hva alle teknikkene i dette kapitlet gjør.

Forwarding fjerner ikke avhengigheten — den fjerner ventetiden avhengigheten skapte. Prediksjon fjerner ikke usikkerheten om hvor hoppet går — den gjetter, og tar kostnaden sjelden i stedet for alltid. Spekulasjon fjerner ikke ventingen — den fyller den med arbeid som kanskje er bortkastet. Flere kjerner fjerner ikke varmeveggen — de går rundt den. GPU-en fjerner ikke bokholderiet — den gjør oppgaven om til noe som ikke trenger det. Cachen, fra forrige kapittel, fjerner ikke avstanden til minnet — den flytter en kopi nærmere.

Ingen av dem løser problemet sitt. Alle sammen flytter konsekvensen et sted der den koster mindre.

Og alle sammen har samme pris: maskinen blir mer komplisert innvendig, mens den later som om

Begrep	Kort sagt
Forwarding	Resultatet rekkes rett over bordet i stedet for å ta omveien om et register. Fjerner de fleste datahasardene.
Forgreningsprediksjon	Maskinen husker hva hvert hopp pleier å gjøre, og gjetter det samme. Treffer over 95 % av gangene.
Spekulativ utføring	Maskinen gjør arbeidet før den vet om gjetningen var riktig, og kaster det hvis den bommet.
Superskalar	Flere hele regneenheter. Flere ferdige instruksjoner per syklus, ikke bare én.
Out-of-order	Instruksjoner utføres i avhengighetsrekkefølge, men fullføres i programrekkefølge.
SMT	To tråder deler én kjerne og fyller hverandres ventetid. Gir 10–30 %, ikke det dobbelte.
Kjerne	En hel prosessor. Fysisk, og du kan se den på brikka.
Tråd	En utføringsvei inne i et program. Trådene er arbeidet, kjernene er arbeiderne.
Amdahls lov	Den delen som ikke lar seg dele, setter taket — uansett hvor mange kjerner du har.
Falsk deling	To kjerner som skriver til samme cachelinje uten å dele data, og betaler for det likevel.
GPU	Tusenvis av små regneenheter med felles styring. Bra på mange like, uavhengige oppgaver.
Fastfunksjonskrets	Kan bare én ting, og gjør den svært strømgjerrig. Har ikke noe bokholderi.
NPU	Fastfunksjonstanken brukt på gange-og-legg-sammen. Se appendiks G.
Avbrudd	En ledning som lar omverdenen avbryte prosessoren. Hele multitaskingen hviler på den.
DMA	En egen krets som flytter data mellom enhet og minne uten å bruke prosessoren.
Virtuelt minne	Hvert program får sitt eget adresserom. Isolasjon først, mer minne etterpå.
Side og sidetabell	Minnet deles i biter på typisk 4 kB, og tabellen oversetter virtuell til fysisk.
TLB	Cache for oversettelsene. Finnes fordi oversettelsen skjer ved hver eneste minnetilgang.
CISC og RISC	Mange sammensatte instruksjoner mot få enkle. Moderne x86 er en RISC-maskin med CISC-grensesnitt.
Abstraksjonslekkasje	Når det nederste laget blir synlig i det øverste. Spectre er eksempelet.

Tabell 8.9 Begrepene fra dette kapitlet.

ingenting er endret utvendig. Det gikk bra i femti år. I avsnitt 8.7 så du hva det koster når forskjellen mellom innsiden og utsiden til slutt blir synlig.

8.12 Videre lesning

Englander, kapittel 8. Avsnitt 8.4 dekker flerkjerne og de moderne teknikkene, men kortere enn vi har gjort det her, og uten å knytte dem til hasardene. Les det som en oppsummering, ikke som en innføring. Kapittel 11 tar systemorganisering og I/O, og avsnittene om avbrudd og DMA passer rett på avsnitt 8.6.

Englander, kapittel 19.7. Her står virtuelt minne grundig, over atten sider: sidetabellen, flere nivåer, sidefeil og hva operativsystemet gjør. Dette er stedet å gå hvis avsnitt 8.6.3 gjorde deg nysgjerrig — vi tok begrepet og isolasjonen, og lot mekanikken ligge.

Patterson & Hennessy, *Computer Organization and Design*. Avsnitt 4.7 og 4.8 tar forwarding og forgreningsprediksjon med den detaljen vi har utelatt, inkludert hvordan prediktortabellen faktisk er bygget. Kapittel 6 tar parallellitet, Amdahls lov og GPU-er. Hvis du skal lese én bok videre etter denne, er det denne.

Om Spectre. Den opprinnelige artikkelen heter «Spectre Attacks: Exploiting Speculative Execution» og er fritt tilgjengelig. Den er skrevet for forskere, men de første fem sidene er overraskende lesbare, og de følger samme fem steg som avsnitt 8.7.

Videre i denne boka. Tre appendikser henger på dette kapitlet. Appendiks G tar KI-maskinvaren ordentlig, og er det lengste og mest sammenvevde i boka. Appendiks F viser hvordan cachen faktisk finner en linje. Appendiks A forteller hvordan vi kom hit, fra radiorør til milliarder av transistorer.

Og videre i studiet. Operativsystemdelen av dette kapitlet — prosesser, tråder, virtuelt minne, avbrudd — er et helt emne for seg. Det samme er nettverksdelen, som vi bare har tatt grensesnittet til. Begge bygger direkte på det du nå kan.

8.13 Refleksjon

1. Nesten alle teknikkene i dette kapitlet gjør maskinen mer komplisert innvendig for å holde grensesnittet utvendig uendret. Finn to eksempler på samme strategi utenfor datateknologien, og ett sted der man i stedet valgte å endre grensesnittet.
2. Spekulativ utføring bruker strøm på arbeid som kanskje kastes. Under hvilke omstendigheter ville du sagt at det ikke er verdt det? Tenk på en maskin som går på batteri, og på en som styrer noe der responstiden må være forutsigbar.
3. Amdahls lov sier at den udelelige delen setter taket. Tenk på en oppgave du har vært med på å løse sammen med andre — en gruppeinnlevering, en flytting, en dugnad. Hva var den udelelige delen der, og vokste den da dere ble flere?
4. Fastfunksjonskretsen er svært effektiv og helt ufleksibel. Hva skjer med en slik krets når standarden den er bygget for, byttes ut? Hvilken risiko tar en produsent som legger mye av brikka si på slike kretser?
5. Spectre oppsto fordi en ytelsesoptimalisering etterlot et spor. Er det mulig å bygge en rask maskin som ikke etterlater spor i det hele tatt? Hva ville man måttet gi opp?
6. Virtuelt minne gir isolasjon mellom programmer. Men to programmer på samme maskin deler fortsatt cache, regneenheter, strømbudsjett og varme. Hvor godt isolert er de egentlig?

7. Tabell 8.6 på side 226 viser at Python-linja koster rundt 24 sykluser, og at maskinen kunne gjort rundt nitti addisjoner på samme tid. Når er det en god handel, og når er det ikke det? Prøv å formulere en regel du kunne gitt til en som begynner på faget til neste år.
8. Se på lagkartet i tabell 8.7 på side 227. Hvilket lag føler du at du forstår best, og hvilket dårligst? Hvilket lag tror du du kommer til å jobbe på?

8.14 Selv-test

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

1. Hva er forwarding, forklart med analogien fra avsnitt 8.2?
2. Hvilken av de tre hasardtypene løser forwarding?
3. Forsvinner avhengigheten når maskinen bruker forwarding? Begrunn.
4. Det finnes ett datahasardtilfelle forwarding ikke fjerner helt. Hvilket, og hvorfor?
5. Hvor mange sykluser brukte de tre instruksjonene i figur 7.8 på side 193, og hvor mange bruker de i figur 8.1 på side 204?
6. Hva er forgreningsprediksjon?
7. Hvorfor er «hopp bakover tas alltid» en god tommelfingerregel?
8. Hva gjør en prediktor som husker, bedre enn en som bare bruker tommelfingerregelen?
9. En løkke går tusen runder. Hvor mange ganger tar hoppet på slutten feil for en prediktor som gjetter «samme som sist»?
10. Hva er spekulativ utføring, og hva skjer med arbeidet hvis gjetningen var feil?
11. Hvorfor er spekulasjon en god handel når treffraten er 95 prosent?
12. Forklar forskjellen på pipelining og superskalar med én setning om hver.
13. En maskin har fire regneenheter. Hva er det teoretiske taket for antall ferdige instruksjoner per syklus?
14. I tabell 8.2 på side 207 gjør A og B like mange addisjoner. Hvorfor er B fire ganger raskere?
15. Hvordan ble klokkefrekvensen på 4,0 GHz regnet ut fra rad A i den tabellen? Hvilken forutsetning hviler utregningen på?
16. Hva er out-of-order-utføring?
17. Hvorfor må instruksjonene likevel *fullføres* i programrekkefølge?
18. Hva er SMT, og hvorfor gir det ikke dobbelt ytelse?
19. Forklar forskjellen på en kjerne, en prosess og en tråd.
20. Et program har tusen tråder og maskinen har åtte kjerner. Går det? Hva skjer?
21. Hva sier Amdahls lov, med dine egne ord?
22. Et program bruker 100 sekunder. 80 sekunder lar seg dele på fire kjerner, 20 gjør ikke. Hvor lang tid tar det på fire kjerner?
23. Samme program på uendelig mange kjerner. Hva er det raskeste det kan bli, og hvorfor?
24. Hvorfor blir et program som er skrevet som en rett strek, ikke raskere av åtte kjerner?
25. Nevn tre grunner til at den udelelige delen i praksis er større enn man tror.
26. Hva er falsk deling, og hvorfor er det overraskende at det koster noe?

27. Hvorfor har en GPU tusenvis av regneenheter der en CPU har fire?
28. Hva er det en GPU *ikke* har, og hvorfor fikk den plass til så mange enheter uten det?
29. Hvordan skjuler en GPU ventetiden på minnet, og hvordan skiller det seg fra det CPU-en gjør?
30. Du skal sortere en million navn alfabetisk. CPU eller GPU? Begrunn med noe annet enn størrelse.
31. Hvorfor går batteriet fortere tomt av å *lage* en video enn av å se på en?
32. Hva er en fastfunksjonskrets, og hvorfor er den så strømgjerrig?
33. Hva regner en enhet i et nevralt nett ut, og hva har det med en AND-port å gjøre?
34. Hvorfor er det transporten av tall, og ikke regningen, som former en KI-brikke?
35. Hvorfor sitter det en NPU i telefonen i stedet for at alt sendes til en tjener? Nevn tre grunner.
36. Hva er et avbrudd, og hva er alternativet?
37. Hvorfor koster det prosessoren nesten ingenting å sjekke om det har kommet et avbrudd?
38. Beskriv de fire tingene maskinen gjør når et avbrudd kommer, i rekkefølge.
39. Hvorfor er avbrudd en forutsetning for at flere programmer kan kjøre samtidig?
40. Hva er DMA, og hvilket problem løser det?
41. Hvordan snakker prosessoren med en skjerm eller en disk, når det ikke finnes egne instruksjoner for det?
42. Hva er virtuelt minne, og hva er den viktigste tingen det gir deg?
43. To programmer bruker begge adresse 42. Hvorfor kolliderer de ikke?
44. Hvorfor er isolasjonen sterkere enn bare et forbud mot å lese andres minne?
45. Hva er en side, og hvorfor skjer oversettelsen per side i stedet for per adresse?
46. Hva er en TLB, og hvorfor måtte den finnes?
47. Hvorfor virker TLB-en så godt? Svaret står i kapittel 7.
48. Hvilke to setninger fra tidligere i boka utgjør til sammen Spectre-hullet?
49. Gå gjennom de fem stegene i avsnitt 8.7 med egne ord.
50. Hvorfor rydder ikke maskinen opp i cachen når den oppdager at gjetningen var feil?
51. Hvordan kan en angriper lese en verdi hen aldri får se?
52. Hvor er programmeringsfeilen i Spectre-eksempelet? Svar nøye.
53. Hvorfor lar ikke Spectre seg fikse uten at noe koster?
54. Hva er forskjellen på CISC og RISC?
55. Hvorfor passer RISC bedre til pipelining?
56. Hvordan kan en x86-prosessor være rask når CISC passer dårlig til pipelining?
57. Hvorfor er det galt å kalle Little Man Computer en RISC-maskin?
58. Ranger disse fra høyeste til laveste abstraksjonsnivå: cachelinje, Python-variabel, fulladder, instruksjonsfelt, transistor, prosess.
59. Følg `svar = a + b` gjennom de åtte lagene i avsnitt 8.9. Nevn hvert lag og hva som skjer der.
60. Nevn tre ting som skjedde «på siden» under den ene addisjonen, og si for hver av dem om den endret svaret eller bare tiden.
61. En Python-addisjon tar rundt 24 klokkesykluser. Hva går de 24 syklusene til, hvis bare én av dem er selve addisjonen?

62. **Finn feilen.** En student skriver: «Hyperthreading dobler antall kjerner.» Hva er galt?
63. **Finn feilen.** En annen skriver: «En GPU er raskere enn en CPU, så alt bør kjøres på GPU.» Hva er galt?
64. **Finn feilen.** En tredje skriver: «Åtte kjerner gjør alle programmer åtte ganger raskere.» Hva er galt, og hva er riktig?
65. **Finn feilen.** En fjerde skriver: «Spectre er en feil i prosessoren som ble rettet med en oppdatering.» Hva er galt med begge halvdelene?
66. **Finn feilen.** En femte skriver: «Virtuelt minne finnes for at maskinen skal kunne bruke mer minne enn den har.» Hva mangler?
67. Alle teknikkene i dette kapitlet har noe til felles. Hva, og hvordan slo det tilbake i avsnitt 8.7?

Og det var hele reisen

Vi begynte i kapittel 1 med en påstand om at en datamaskin er lag på lag.

Nå har du vært nede på bunnen og opp igjen. Du har bygd en adderer av porter, porter av transistorer, og du har regnet ut hvorfor den tar tid. Du har sett en instruksjon bli til et tall og et tall bli til spenninger. Du har sett at en maskin gjør fire addisjoner i samme klokkeslag, og at klokkefrekvensen kan leses ut av en `for`-løkke — og du har det du trenger for å måle begge deler på din egen maskin.

Og du kan lese databladet fra første side.

Åtte kjerner, fordi klokka ikke kunne gå fortere. 4,0 gigahertz, fordi varmen setter grensen — ikke portene, som ville tålt mye mer. 12 megabyte buffer, fordi minnet er hundre ganger for tregt. Integrert grafikk, fordi noen oppgaver består av åtte millioner like biter. Nevral motor, fordi det å flytte et tall koster mer enn å gange det.

Fem linjer. Ikke én av dem er en vilkårlig teknisk detalj. Hver av dem er et svar på et problem som ikke lot seg løse på noen annen måte.

Det er det du kan nå. Ingen av lagene under deg er magi lenger — de er valg som noen tok, av grunner du kan gjøre rede for.

Og det er ikke en liten ting å ta med seg videre.

Del I

Fasit for selv-test

Fasit til Selv-Test F1

Abstraksjonsnivåer

Svarene her refererer til spørsmålene 1.8 på side 11

1. Et abstraksjonsnivå er et lag i et system som skjuler kompleksiteten til underliggende lag og tilbyr et forenklet grensesnitt for interaksjon. For eksempel er et operativsystem et abstraksjonsnivå som skjuler kompleksiteten til maskinvaren og tilbyr et brukergrensesnitt og API-er for applikasjoner.
2. Korrekt rekkefølge fra høyeste til laveste abstraksjonsnivå:

   Applikasjonsprogrammer

   Høynivåprogrammeringsspråk

   Operativsystem

   Assemblykode

   Logiske porter
3. Abstraksjonsnivåer bidrar til å håndtere kompleksitet i store softwaresystemer ved å:
   - Dele systemet inn i håndterbare moduler eller lag
   - Skjule komplekse detaljer bak enkle grensesnitt
   - Muliggjøre fokus på ett aspekt av systemet om gangen
   - Tillate parallell utvikling av ulike systemkomponenter
   - Forenkle vedlikehold og oppdatering av spesifikke deler av systemet
4. Et eksempel på hvordan abstraksjonsnivåer kan fremme innovasjon i teknologiutvikling er utviklingen av høynivå programmeringsspråk. Ved å abstrahere bort kompleksiteten av maskinvare og lavnivå programmering, har høynivåspråk som Python eller JavaScript gjort det mulig for flere mennesker å lære programmering og skape innovative applikasjoner uten å måtte forstå de underliggende detaljene om hvordan datamaskinen fungerer.
5. Å arbeide på et høyt abstraksjonsnivå innebærer å fokusere på overordnede konsepter og funksjonalitet uten å bekymre seg for lavnivå detaljer. Dette er fordelaktig når man ønsker å utvikle raskt, fokusere på forretningslogikk, eller når ytelse ikke er kritisk. Å arbeide på et lavt abstraksjonsnivå innebærer å ha direkte kontroll over systemressurser og hardwarenære operasjoner. Dette er fordelaktig når man trenger maksimal ytelse, har strenge minnebegrensninger, eller når man utvikler systemprogramvare eller drivere.
6. Korrekt rekkefølge fra fra høyeste til laveste abstraksjonsnivå i et operativsystem:

Brukergrensesnitt

Filsystem

Prosesshåndtering

Systemkall

Drivere

Fasit til Selv-Test F2

Digital representasjon

Svarene her refererer til spørsmålene 2.16 på side 52

1. Et posisjonelt tallsystem er et system der verdien til et siffer avhenger av *hvor* sifferet står, ikke bare av hvilket siffer det er. Vekten til et siffer i posisjon k er basen opphøyd i k. Posisjonene telles fra høyre og starter på null.

   For 407_{10}: sifferet 4 står i posisjon 2 og har vekten $10^2 = 100$. Sifferet 0 står i posisjon 1 med vekten $10^1 = 10$. Sifferet 7 står i posisjon 0 med vekten $10^0 = 1$. Verdien er $4 \times 100 + 0 \times 10 + 7 \times 1 = 407$.

2. (a) $10110_2 = 16 + 0 + 4 + 2 + 0 = 22_{10}$
   (b) $2\mathrm{F}_{16} = 2 \times 16 + 15 = 47_{10}$
   (c) $351_8 = 3 \times 64 + 5 \times 8 + 1 = 192 + 40 + 1 = 233_{10}$

3. (a) Divisjonsmetoden:

$$\begin{aligned} 200 \div 2 &= 100 \quad \text{rest } 0 \\ 100 \div 2 &= 50 \quad \text{rest } 0 \\ 50 \div 2 &= 25 \quad \text{rest } 0 \\ 25 \div 2 &= 12 \quad \text{rest } 1 \\ 12 \div 2 &= 6 \quad \text{rest } 0 \\ 6 \div 2 &= 3 \quad \text{rest } 0 \\ 3 \div 2 &= 1 \quad \text{rest } 1 \\ 1 \div 2 &= 0 \quad \text{rest } 1 \end{aligned}$$

   Nedenfra og opp: $200_{10} = 11001000_2$.

   (b) Grupper binærtallet i firere: `1100 1000` $\rightarrow \mathrm{C8}_{16}$.

   (c) Grupper i treere fra høyre: `011 001 000` $\rightarrow 310_8$.

   Legg merke til at (b) og (c) ble rene oppslag så snart (a) var på plass.

4. B = 1011 og 3 = 0011, altså `10110011`.

   Du slipper unna med det fordi $16 = 2^4$. Ett heksadesimalt siffer svarer til nøyaktig fire bit, uten rest og uten avrunding. Å konvertere er derfor et oppslag i tabell 2.1, ikke en utregning.

5. Grupper fra høyre: `101 1010`. Den øverste gruppa har bare tre bit, så du fyller på med en null foran: `0101 1010`. Slår du opp, får du $5A_{16}$.

   Det er lov fordi nuller helt til venstre ikke endrer verdien. Akkurat som 0042 er det samme tallet som 42, er `0101` det samme som `101`. Kontroll: $5A_{16} = 5 \times 16 + 10 = 90$, og $1011010_2 = 64 + 16 + 8 + 2 = 90$.

6. $7 = 111$, $5 = 101$ og $0 = 000$, altså `111 101 000`.

   Eieren kan lese, skrive og utføre filen. Gruppen kan lese og utføre, men ikke skrive. Alle andre har ingen rettigheter i det hele tatt. Dette er et helt vanlig rettighetssett for et kjørbart skript som bare eieren skal kunne endre.

7. Tallene fra 0 til og med 1000 er 1001 ulike verdier. Med n bit får vi 2^n verdier, så vi trenger den minste n der $2^n \geq 1001$. Siden $2^9 = 512$ er for lite og $2^{10} = 1024$ er nok, trengs **10 bit**.

   Merk at 10 bit gir plass til 1024 verdier, altså 23 flere enn vi trenger. Det er vanlig. Bitantall kommer i hele trinn, og man må nesten alltid ta litt mer enn man strengt tatt behøver.

8. (a) Uten fortegn: $128 + 64 + 32 + 0 + 8 + 0 + 0 + 0 = 232$.

   (b) Toerkomplement: øverste bit har vekten -128, så $-128 + 64 + 32 + 8 = -24$.

   Legg merke til sammenhengen: $232 - 256 = -24$. De to tolkningene skiller seg alltid med 2^n, her 2^8. Det er klokkeskiva fra kapitlet.

9. $37 =$ `00100101`. Inverter alle bitene: `11011010`. Legg til 1: `11011011`.

   Kontroll: $-128 + 64 + 16 + 8 + 2 + 1 = -128 + 91 = -37$. Det stemmer.

10. $20 =$ `00010100`.

```
  11011011     (-37)
+ 00010100     (20)
----------
  11101111
```

    Verdien av `11101111` er $-128+64+32+8+4+2+1 = -128+111 = -17$, og $-37+20 = -17$. Det stemmer.

    Legg merke til at vi aldri trengte å vite at det ene tallet var negativt. Bitene ble lagt sammen som om begge var vanlige positive tall.

11. Negerer vi `00000000` i toerkomplement, inverterer vi først til `11111111` og legger så til 1. Det gir `1 00000000`, og menten faller utenfor de åtte bitene. Igjen står `00000000` – samme mønster vi startet med. Null er altså sin egen negasjon, og det finnes bare ett bitmønster for den.

    I fortegn-og-magnitude er `00000000` lik $+0$ og `10000000` lik -0. I enerkomplement er `00000000` lik $+0$ og `11111111` lik -0, siden negering der bare er inversjon.

    Fordelen er todelt. Maskinen slipper å teste for to ulike nullmønstre hver gang den skal sjekke om en verdi er null, og det må den gjøre etter praktisk talt hver operasjon. I tillegg blir ett bitmønster ledig, og toerkomplement bruker det på -128. Det er derfor området er -128 til 127 og ikke -127 til 127.

12. Java regner først ut $30000 + 10000 = 40000$ som en `int`, der det er god plass. Castingen til `short` beholder bare de 16 nederste bitene:

$$40000_{10} = \texttt{1001 1100 0100 0000}_2$$

Øverste bit er 1, og en `short` leses som toerkomplement. Verdien blir derfor

$$-32768 + 4096 + 2048 + 1024 + 64 = -25536.$$

Resultatet er altså -25536. Bitene er ikke feil – de er nøyaktig de 16 nederste bitene i 40000. Det som er «feil», er at vi ba om en tolkning der det øverste bitet betyr noe helt annet enn det gjorde i `int`-en.

13. Vennen har delvis rett, og det er nettopp derfor påstanden er farlig.

    Riktig: Pythons egne heltall har ingen fast bredde. Trenger tallet flere bit, skaffer Python flere bit. Skriver du ren Python og regner med vanlige tall, får du aldri overflyt.

    Galt: «Python» er ikke ett sted. Store deler av økosystemet – NumPy og det meste annet som skal gå fort – er skrevet i C, og bruker C-ens tall med faste bitbredder. Et NumPy-array av typen `int32` renner over akkurat som i Java, stille og uten feilmelding. Samme fil, samme språk, to helt ulike regelsett.

    I tillegg er det ikke gratis. Hver operasjon må sjekke om resultatet får plass, og kompilatoren kan ikke optimalisere når den ikke vet hvor stort et tall kan bli. Det er en stor del av grunnen til at Python er tregt.

    Konklusjonen: spør hvilken *datatype* du regner med, ikke hvilket språk du skriver i.

14. Strengen `Bodø"` består av **4 tegn**: B, o, d og ø.

    (a) I UTF-8 tar B, o og d én byte hver, siden de ligger under 128 og dermed i ASCII-området. Bokstaven ø har kodepunktet `U+00F8` = 248, som er større enn 127 og trenger toformen med to byte. Til sammen **5 byte**.

    (b) I UTF-32 tar hvert tegn fire byte, alltid. Til sammen **16 byte**.

    De tre tallene er ulike fordi de måler tre ulike ting. Antall tegn handler om skrift. Antall kodepunkter handler om Unicode-standarden. Antall byte handler om hvilken koding vi har valgt. Skillet mellom kodepunkt og koding er nettopp det som gjør at samme tegn kan ta ulikt antall byte uten at det er noen motsetning.

15. Det er **ett tegn**, og det består av alle tre bytene.

    Du ser det på prefiksbitene, uten å slå opp noe sted:

    - `0xE2` = `11100010`. Den begynner med `1110`, altså tre enere før første null. Det betyr en sekvens på tre byte.
    - `0x82` = `10000010` og `0xAC` = `10101100`. Begge begynner med `10`, som betyr «jeg er en fortsettelse av et tegn som startet tidligere».

    Tegnet er for øvrig eurosymbolet, `U+20AC`.

16. **Som tekst:** fire tegn – `3`, `8`, `,` og `5`. Hvert av dem har sitt eget kodepunkt i Unicode, og kommaet er et helt vanlig tegn på linje med sifrene. I UTF-8 blir det fire byte. Maskinen aner ikke at dette er et tall, og kan ikke regne med det før noen har gjort det om.

    Som `double`: et flyttall på 64 bit, med fortegn, eksponent og mantisse. Kommaet lagres *ikke i det hele tatt*. Hvor det står, er gitt av formatet, ikke av dataene. Til gjengjeld kan maskinen regne med verdien direkte.

    Samme tall på papiret. To helt ulike ting i minnet, med helt ulike regler. Dette er kapitlets bærende idé i praktisk form, og det er grunnen til at du må vite hvilken type du leser data inn i.

17. **Steg 1 – binær form.** $6{,}5 = 4 + 2 + 0{,}5$, altså $110{,}1_2$.

    Steg 2 – normalisering. Flytt binærpunktet to plasser til venstre:

    $$110{,}1_2 = 1{,}101_2 \times 2^2$$

    Steg 3 – feltene.

    - Fortegn: tallet er negativt, så $S = 1$.
    - Eksponent: virkelig eksponent er 2, lagret verdi er $2 + 127 = 129 =$ `10000001`.
    - Mantisse: sifrene etter det ledende ettallet er `101`, resten fylles med nuller.

    ```
    1  10000001  10100000000000000000000
    ```

    Gruppert i firere blir dette `0xC0D00000`.

18. Ingen av tallene 0,1, 0,2 og 0,3 kan skrives eksakt i base 2. Alle tre har uendelig binærutvikling, og maskinen lagrer den nærmeste verdien den klarer å lage av hver. Når den så legger sammen tilnærmingen til 0,1 og tilnærmingen til 0,2, faller summen ikke nødvendigvis på nøyaktig samme verdi som tilnærmingen til 0,3. Avrundingsfeilene opphever ikke hverandre. Resultatet i en `double` blir

    $$0{,}30000000000000004.$$

    $0{,}5 + 0{,}25 == 0{,}75$ gir derimot sant. Alle tre tallene er endelige summer av negative toerpotenser: $0{,}5 = 2^{-1}$, $0{,}25 = 2^{-2}$ og $0{,}75 = 2^{-1} + 2^{-2}$. Da finnes det ingen avrundingsfeil, og likheten holder eksakt.

    Den generelle regelen: en brøk har endelig antall siffer i base B bare hvis nevneren ikke har andre primfaktorer enn dem B har. Nevneren i $\frac{1}{10}$ er 2×5, og 5 går ikke opp i 2.

19. (a) $1920 \times 1080 = 2\,073\,600$ piksler. Med 3 byte per piksel blir det

    $$2\,073\,600 \times 3 = 6\,220\,800 \text{ byte} \approx 6{,}2 \text{ MB}.$$

    (b) 16 farger trenger indekser fra 0 til 15, altså 4 bit per piksel:

    $$2\,073\,600 \times 4 \text{ bit} = 8\,294\,400 \text{ bit} = 1\,036\,800 \text{ byte}$$

    pluss en fargetabell på $16 \times 3 = 48$ byte. Til sammen 1 036 848 byte, altså en sjettedel av det opprinnelige – vi gikk fra 24 til 4 bit per piksel.

    Og ja, dette er **tapsfritt**. Bildet inneholdt bare 16 farger fra før, så hver eneste piksel får nøyaktig sin opprinnelige farge tilbake når tabellen slås opp. Ingenting er kastet, bare skrevet mer kompakt.

20. Dataraten er

    $$48\,000 \times 24 \times 2 = 2\,304\,000 \text{ bit/s} = 288\,000 \text{ byte/s}.$$

    Ett minutt er 60 sekunder:

    $$60 \times 288\,000 = 17\,280\,000 \text{ byte} \approx 17{,}3 \text{ MB}.$$

    Etter samplingsteoremet kan opptaket gjengi frekvenser opp til halve samplingsfrekvensen, altså **24 000 Hz**. Det er godt over det et menneske kan høre, og det er meningen. Marginen gjør det lettere å filtrere bort det som ligger over, slik at høye frekvenser ikke folder seg ned som falske toner.

21. Det de har til felles, er at de *ikke* bygger på mønstre i signalet, men på begrensninger hos mottakeren. JPEG kaster detaljer øyet ikke legger merke til – raske fargevariasjoner og finstruktur. MP3 kaster lyd øret ikke hører – frekvenser utenfor hørselsområdet, og svake lyder som maskeres av kraftigere lyder i nærheten.

    Med andre ord: begge representasjonene er tilpasset et menneskelig sanseapparat, ikke innholdet. Det er en helt annen avtale enn tapsfri komprimering, som lover at dataene kommer tilbake bit for bit.

    Situasjoner der ingen av dem bør brukes: medisinske bilder som skal diagnostiseres eller analyseres av programvare, bilder som er bevismateriale, arkivmateriale som skal bevares, og opptak som skal redigeres og lagres på nytt flere ganger. Og all data som skal behandles av en maskin i stedet for oppfattes av et menneske – da er hele forutsetningen borte. En algoritme har ikke et øye som lar seg lure.

22. Du kan si at filen inneholder 32 bit, og at bitmønsteret er ett av 2^{32} mulige, altså omtrent 4,3 milliarder.

    Du kan *ikke* si om det er et heltall uten fortegn, et toerkomplementtall, en `float`, fire ASCII-tegn, ett UTF-32-tegn, en RGBA-farge, en minneadresse eller en maskininstruksjon. Alle tolkningene er like gyldige, og bitene skiller ikke mellom dem.

    Opplysningen må komme utenfra. I praksis kommer den fra filendelsen, fra et magisk tall i starten av filen, fra en formatspesifikasjon, fra en protokoll som ble avtalt på forhånd, eller fra typen variabelen ble erklært med i programmet som leser filen.

    Dette er kapitlets bærende idé: representasjonen er en avtale, og avtalen lagres ikke sammen med dataene.

Fasit til Selv-Test F3

Instruksjoner

Fasit til selv-testen i kapittel 3, 3.13 på side 76.

1. Hundre, nummerert 00 til 99. Grunnen er at adressen skal få plass i to sifre. To sifre gir nøyaktig hundre kombinasjoner, og en tresifret instruksjon har bare to sifre til overs når det første er brukt til operasjonskoden.
2. Operasjonskoden er 3, adressedelen er 47. Koden 3 er `STA`, så instruksjonen betyr: kopier tallet i akkumulatoren til postkasse 47.
3. Adressen er hvor postkassen er. Innholdet er tallet som ligger der. De er lette å forveksle når tallene ligner, for eksempel når postkasse 47 inneholder tallet 47. Det er også lett å forveksle dem i en instruksjon som `ADD 99`, som betyr «legg til innholdet i postkasse 99», ikke «legg til 99».
4. Hent instruksjonen fra postkassen telleren peker på. Del instruksjonen i operasjonskode og adressedel. Utfør det operasjonskoden sier. Begynn på nytt. Telleren klikkes fram i steg 1, samtidig med hentingen.
5. Den peker på den neste instruksjonen, ikke på den som utføres. Den ble klikket fram i det instruksjonen ble hentet.
6. Fordi ingen postkasse er innom. `901` leser fra innkurven, og adressedelen 01 er en forlengelse av operasjonskoden. Den skiller inn-instruksjonen fra ut-instruksjonen `902` og tegnutskriften `922`.
7. Utkurven inneholder 42. Postkasse 99 inneholder 40. `ADD` leser postkassen uten å endre den, så det første tallet blir liggende.
8. ```
           INP
           STA tall
           INP
           ADD tall
           OUT
           HLT
   tall    DAT 0
   ```
9. Opkode 4.
10. `BRP 20` hopper, fordi null regnes som positivt. `BRZ 20` hopper også, fordi akkumulatoren er null. De to skiller lag først når akkumulatoren er positiv: da hopper `BRP`, mens `BRZ` lar være. Er akkumulatoren negativ, hopper ingen av dem.
   ```

11. Programmet leser to tall, trekker det første fra det andre og bruker fortegnet til å velge hvilket som skal skrives ut.

```
          INP
          STA a
          INP
          STA b
          SUB a
          BRP bstorst
          LDA a
          BRA ut
bstorst   LDA b
ut        OUT
          HLT
a         DAT 0
b         DAT 0
```

    Er de to tallene like, skrives det ene ut, og det er riktig svar uansett hvilket.

12. Å doble er å legge tallet til seg selv. Tallet må lagres først, for akkumulatoren har bare plass til ett.

```
          INP
          STA x
          ADD x
          OUT
          HLT
x         DAT 0
```

13. Feilen er `BRZ` i postkasse 03. Den hopper bare når akkumulatoren er null, og etter første runde er akkumulatoren 4. Programmet faller derfor rett gjennom til `HLT`. Betingelsen er snudd: vi vil fortsette så lenge tallet er null eller større, altså `BRP`.

```
00  LDA 05
01  OUT
02  SUB 06
03  BRP 01
04  HLT
05  5
06  1
```

    Nå skrives 5, 4, 3, 2, 1, 0 ut, og akkumulatoren står på -1 når maskinen stopper.

14. `BRZ` ser ikke på instruksjonen over. Den ser på akkumulatoren. At de to ofte henger sammen, er tilfeldig: det er bare fordi instruksjonen over som regel er den som sist endret akkumulatoren. Riktig formulering: «`BRZ` hopper hvis akkumulatoren er null.» Står det en `LDA`, en `ADD` eller en `INP` mellom testen og regnestykket, tester du på noe annet enn du tror. En `OUT` eller en `STA` mellom dem er derimot ufarlig, for de leser akkumulatoren uten å endre den.

15. Etter `svar DAT 7` står tallet 007 i postkassen. Etter `peker DAT svar` står tallet 031, altså adressen til `svar` og ikke innholdet der.

16. Fordi maskinen ikke har noen operasjonskode for den. `DAT` er en beskjed til assembleren om å sette av en postkasse, og eventuelt legge et tall i den. Når assembleren er ferdig, finnes det bare et tall i minnet, og ingenting som sier at tallet var ment som data.

17. Maskinen henter 250 og utfører det som en instruksjon. Operasjonskoden er 2, som er `SUB`, og adressedelen er 50. Maskinen trekker altså innholdet i postkasse 50 fra akkumulatoren, og fortsetter til neste postkasse. Hva som skjer videre avhenger både av hva som ligger i postkasse 50 og av hva som ligger etter 250. Ingen av delene er bestemt av programmet, og derfor kan utfallet ikke forutsies uten å vite hva hele minnet inneholder.

18. Etikettene løser problemet fordi de ikke inneholder noe tall. De er navn, og assembleren regner ut adressen på nytt hver gang den kjøres. Setter du inn en instruksjon i midten, får alle etterfølgende postkasser en ny adresse, og alle henvisninger til dem får det også. Derfor må assembleren kjøre på nytt: det er selve oversettelsen fra navn til tall som endrer seg, ikke koden du skrev.

19. Programmet ville skrevet `LDA 12` inn i postkasse 02 på nytt i hver eneste runde. Telleren ville aldri kommet forbi den første bokstaven. Resultatet blir en evig løkke som skriver `h` igjen og igjen. Instruksjonene i postkasse 00 og 01 er en oppstart som bare skal kjøre én gang.

20. Nullen brukes som stoppsignal, og testes før den blir lagt til.

    ```
    les     INP
            BRZ ferdig
            ADD sum
            STA sum
            BRA les
    ferdig  LDA sum
            OUT
            HLT
    sum     DAT 0
    ```

    Legg merke til at `sum` må initialiseres til null. Uten `DAT 0` ville summen begynt på det som tilfeldigvis lå i postkassen fra forrige kjøring.

21. Det er den samme påstanden, sett fra to kanter. Kapittel 2 sa at et mønster trenger en tolkning utenfra. Dette kapitlet har vist at maskinen selv er en slik tolkning, og at den ikke kan velge feil, for den vet ikke at det finnes et valg. Det nye er at tolkningen ikke bestemmes av hva tallet er, men av hvordan maskinen kom dit: for å hente en ordre, eller for å hente en verdi.

22. Lå det data der, får programmet feil svar, og feilen kan dukke opp langt unna stedet der den ble laget. Lå det en instruksjon der, kjører maskinen noe annet enn programmet, og oppførselen avhenger av hvilket tall som ble skrevet. Lå det en instruksjon der *og* en utenforstående kan styre hvilket tall som skrives, kan den utenforstående bestemme hva maskinen gjør. Det siste er kjernen i en stor andel av sikkerhetshullene i programvare.

Fasit til Selv-Test F4

Arkitektur og ALU

Fasit til selv-testen i kapittel 4, 4.14 på side 103.

1. ISA er avtalen mellom programvaren og maskinvaren: hvilke ordrer som finnes, hva hver av dem gjør, hvor mange registre maskinen har, og hvordan en ordre er bygget opp. Mikroarkitekturen er hvordan maskinen faktisk er bygget for å holde den avtalen. ISA er rattet og pedalene, mikroarkitekturen er motoren.

2. ISA-en er lik. Det er derfor de samme programmene kjører på begge. Mikroarkitekturen er forskjellig: den raske maskinen bruker færre klokkeslag per ordre, har høyere klokkefrekvens, eller arbeider med flere ordrer samtidig.

3. Fem bit opkode gir $2^5 = 32$ ulike ordrer. Da er det $16 - 5 = 11$ bit igjen til adressen, og $2^{11} = 2048$ adresser.

4. Opkoden for `ADD` er 1, altså 0001. Tallet 13 er 00001101 i åtte bit. Ordren blir 0001 00001101.

5. Opkoden 0011 er 3, som er `STA`. Adressen 00001010 er 10. Ordren er `STA 10`, altså `310`: kopier tallet i akkumulatoren til postkasse 10.

6. Fordi 6 bit adresse bare gir $2^6 = 64$ adresser, og LMC trenger hundre postkasser. Opkodene ville det vært rikelig plass til: 6 bit gir plass til 64 opkoder, og vi trenger ni.

7. Fordi minnet adresseres i byte. I LMC rommer én postkasse en hel ordre, så neste ordre ligger ett hakk unna. En ordre på 32 bit er fire byte og opptar fire adresser på rad, så neste ordre ligger fire hakk unna. Regelen er den samme i begge tilfeller: telleren skal peke på neste ordre.

8. For det første er hvert register kablet til én bestemt oppgave, mens alle plasser i minnet er like. For det andre har registre ingen adresse; de slås ikke opp, men nås direkte av kontrollenheten.

9. Fordi en operasjon mellom to registre slipper unna minneoppslaget, som er det trege steget. Prisen er at hvert registernummer må få plass i instruksjonsordet. Trettito registre krever fem bit å peke på, og de bitene tas fra de andre feltene.

10. Fordi prosessoren selv bestemmer hvilken adresse som skal brukes, og nettopp har lagt den dit. Det finnes ingen grunn til å lese tilbake en adresse man selv har skrevet.

11. Aktiveringslinja bestemmer *når* overføringen skjer. Den trengs fordi adressesignalene bruker et lite øyeblikk på å stabilisere seg når adressen endres. Underveis kan flere rader være halvveis påslått. Aktiveringslinja slås derfor ikke på før adressen har falt til ro.

12. $2^{20} = 1\,048\,576$ adresser, altså omtrent én million. Med én byte per adresse blir det 1 megabyte.

13. Fordi MAR var 32 bit bredt. Det gir $2^{32} = 4\,294\,967\,296$ ulike adresser, og med én byte per adresse er det fire gigabyte. Flere adresser fantes det ingen måte å uttrykke på, uansett hvor mange minnebrikker maskinen hadde.

14. MAR snakker på adressebussen. MDR snakker på databussen. Lese/skriv-linja og aktiveringslinja ligger på kontrollbussen.

15. $32 + 24 = 56$ linjer.

16. Skjevhet er at signalene på de ulike ledningene ikke kommer fram helt samtidig, fordi ledningene verken er like lange eller ligger helt likt. Maskinen må vente på det tregeste signalet før den kan lese av noe. Jo flere ledninger, jo større sjanse for at én henger etter, og dermed må klokka gå saktere. En seriell buss har ingen slik skjevhet å vente på, og kan klokkes mye fortere.

17. Tre. To datainnganger som bærer tallene det skal regnes med, og ett styresignal som bærer beskjeden om hvilken operasjon som skal utføres. Styresignalet er altså ikke data.

18. Fra opkoden i instruksjonsordet. Kontrollenheten dekoder opkoden og slår på den linja som svarer til ordren, og den linja styrer ALU-en. Det er her ordren du skrev møter kretsen som utfører den.

19. Fordi ELLER regnes ut bit for bit, helt uavhengig for hver posisjon. Alle 64 kan gjøres samtidig. En addisjon må sende menten videre fra posisjon til posisjon, og den kjeden tar tid.

20. I fire bit toerkomplement er $-5 = 1011$ og $-6 = 1010$. Summen er $1011 + 1010 = 1\,0101$. Det blir mente ut, altså C. Begge tallene var negative, mens resultatet 0101 er $+5$, altså positivt — da er det også overflyt, altså V. Svaret er galt: $-5 + (-6) = -11$, som ikke får plass i fire bit, der området er -8 til $+7$.

21. Overflyt-flagget, V. Maskinen stoppet ikke fordi flaggene bare settes; de gjør ingenting av seg selv. Det er opp til programmet å lese dem. Språk som Python sier fra fordi noen har skrevet kode som sjekker, og det er ikke noe maskinen gjør gratis.

22. $2^5 = 32$ utganger, og nøyaktig én av dem er på om gangen.

23. Steg 1: programtelleren gir adressen til MAR, ordren hentes inn i MDR, og telleren klikkes fram. Steg 2: ordren kopieres fra MDR til instruksjonsregisteret. Steg 3: opkoden går til kontrollenheten, og adressedelen 30 kopieres til MAR. Steg 4: minnet slår opp adresse 30, og verdien havner i MDR. Steg 5: verdien i MDR og verdien i akkumulatoren går inn i ALU-en, og summen legges tilbake i akkumulatoren.

    Bare steg 5 skiller `ADD` fra `LDA`. I `LDA` kopieres verdien rett fra MDR til akkumulatoren uten å gå om ALU-en.

24. Første gang holder MAR adressen til selve ordren, hentet fra programtelleren. Andre gang holder det adressen ordren peker på, hentet fra adressefeltet i ordren.

25. Minnet holder både program og data. Minnet adresseres lineært, med ett nummer per plass. Adressen sier ingenting om hva slags innhold som ligger der. Ordrene utføres i rekkefølge, til noe får maskinen til å hoppe.

26. Program og data ligger i samme minne og må dele den samme veien inn til prosessoren. Hver ordre må hentes over den bussen, og hvert tall ordren trenger, må hentes over den samme bussen. En raskere prosessor hjelper ikke, fordi alt uansett må gjennom den ene åpningen. Flaskehalsen ligger i veien, ikke i den som venter på den.

Fasit til Selv-Test F5

Porter og logiske funksjoner

Fasit til selv-testen i kapittel 5, 5.14 på side 141.

1. Funksjonen er *hva* som skal skje: hvilken utgang hver kombinasjon av innganger skal gi. Kretsen er *hvordan* det skjer: hvilke porter som er koblet sammen på hvilken måte. Den samme funksjonen kan bygges av mange ulike kretser, og noen av dem er langt billigere enn andre. Det er nettopp derfor forenkling gir mening.

2. Fire innganger gir $2^4 = 16$ rader. Seks innganger gir $2^6 = 64$.

3. Vi tegner dem alle seks med to innganger. Det er ikke en egenskap ved portene, bare et valg som holder eksemplene små: AND og OR finnes like gjerne med tre eller fire innganger, og kapitlet bruker nettopp slike når dekoderen bygges. Oppførselen er den samme — en AND med fire innganger gir 1 bare når alle fire er 1. XOR og XNOR generaliseres også, men sjeldnere, og NOT har alltid én inngang.

4. XOR gir 1 når inngangene er forskjellige. XNOR gir 1 når de er like.

5. NAND gir 0 bare når begge inngangene er 1, og 1 ellers:

A	**B**	**F**
0	0	1
0	1	1
1	0	1
1	1	0

6. AND gir 1. OR gir 1. XOR gir 0. NAND gir 0. NOR gir 0. XNOR gir 1.

7. Fordi + ikke betyr addisjon her. Det betyr «minst én av dem er 1». Når begge er 1, er påstanden fortsatt bare sann, og sann skrives som 1. Det finnes ingen verdi 2 i systemet, og det finnes ingen mente — en port har bare én utgang.

8. $\overline{A}\,\overline{B}$ er «ikke A, og ikke B», altså NOR. $\overline{A \cdot B}$ er «ikke (A og B)», altså NAND. De er ulike i to av fire rader: for A=0, B=1 gir den første 0 og den andre 1, og for A=1, B=0 gir den første 0 og den andre 1. De er bare like når begge inngangene er like.

9. Kretsen er: en inverter på A og en inverter på B; så en AND-port med $\overline{A}$ og B, og en annen AND-port med A og $\overline{B}$; og til slutt en OR-port som tar de to AND-utgangene. Til sammen fem porter. Den ene porten som gjør nøyaktig det samme, er XOR.

10. Tre rader har 1, så uttrykket blir

$$F = \overline{A}B + A\overline{B} + AB$$

De to siste leddene har A felles: $\overline{A}B + A(\overline{B} + B) = \overline{A}B + A$.

Nå står det $A + \overline{A}B$. Det er absorpsjonsregelen $\overline{X} + XY = \overline{X} + Y$, der det er $\overline{A}$ som spiller rollen til X. Leddet $\overline{A}$ i produktet kan dermed strykes:

$$F = A + B$$

Det er en ren OR-port, noe du også kan se direkte: utgangen er 1 i alle rader utenom 00.

11. Utgangen er 0 bare i den ene raden der A er 1 og B er 0:

A	**B**	**F**
0	0	1
0	1	1
1	0	0
1	1	1

12. $AB + A\overline{B} = A(B + \overline{B}) = A \cdot 1 = A$. Hele B faller bort: når A er 1 er utgangen 1 uansett hva B er, og når A er 0 er den 0.

13. $\overline{A} + AB = \overline{A} + B$. Regelen er absorpsjon i varianten $\overline{X} + XY = \overline{X} + Y$. Begrunnelsen: hvis $A = 0$ er begge sider 1, og hvis $A = 1$ faller det første leddet bort på begge sider og vi står igjen med B mot B.

14. $ABC + AB\overline{C} = AB(C + \overline{C}) = AB$.

15. Sett A utenfor parentes: $A(B + C + \overline{B}\,\overline{C})$. Innholdet i parentesen er alltid 1 — enten er minst én av B og C lik 1, og da er et av de to første leddene 1, eller så er begge 0, og da er $\overline{B}\,\overline{C}$ lik 1. Altså

$$F = A$$

16. De Morgan på en overstreket sum gir et produkt av de overstrekede leddene:

$$\overline{A + \overline{B}} = \overline{A} \cdot \overline{\overline{B}} = \overline{A} \cdot B$$

Den doble overstreken over B opphever seg selv.

17. I nøyaktig én rad, den der alle tre inngangene er 0, altså A=0, B=0, C=0. Det er generelt for slike ledd: hvert ledd i sum-av-produkter-formen er skreddersydd for én eneste rad og er 0 i alle de andre.

18. Det er XNOR. På sum-av-produkter-form blir det $F = \overline{A}\,\overline{B} + AB$, siden utgangen er 1 i rad 00 og rad 11.

19. Studentens svar er feil. Inverteringene står på *inngangene*, ikke på utgangen, så det riktige uttrykket er

$$F = \overline{A} + \overline{B}$$

Ved De Morgan er dette det samme som $\overline{A \cdot B}$, altså en **NAND**-port. Uttrykket studenten skrev, $\overline{A + B}$, er en NOR-port, og den bygges av en OR-port med én inverter *etter* seg. De to kretsene gir ulikt svar i to av fire rader.

20. Fordi den bare har to innganger, og dermed ikke kan ta imot mente fra kolonnen til høyre. Den duger bare til den aller første bitposisjonen i en addisjon. Alle de andre posisjonene har tre tall å legge sammen og trenger en fulladder.

21. Fem porter: to XOR, to AND og én OR. Hver halvadder bidrar med en XOR og en AND, og OR-porten samler de to mentene.

22. Fordi de to halvadderne aldri gir mente samtidig. Gir den første mente, var både A og B lik 1, og da er summen deres 0 — og en halvadder som får 0 inn kan ikke gi mente. Siden høyst én av de to kan være 1, holder det å spørre «er minst én av dem 1», og det er en OR.

23. Mente ut er 1 i fire rader, så

$$C_{ut} = \overline{A}BC_{inn} + A\overline{B}C_{inn} + AB\overline{C_{inn}} + ABC_{inn}$$

Trikset er å bruke $X + X = X$ og skrive ABC_{inn} tre ganger, slik at hvert av de tre andre leddene får en partner å slå seg sammen med. Da blir det $BC_{inn}(\overline{A}+A)+AC_{inn}(\overline{B}+B)+AB(\overline{C_{inn}}+C_{inn})$, altså

$$C_{ut} = AB + AC_{inn} + BC_{inn}$$

I ord: mente ut er 1 når minst to av de tre inngangene er 1.

24. Regn posisjon for posisjon fra høyre, og la menten følge med videre mot venstre:

Posisjon	**A**	**B**	**mente inn**	**sum**	**mente ut**
0 (høyre)	0	1	0	1	0
1	1	0	0	1	0
2	1	1	0	0	1
3 (venstre)	0	0	1	1	0

Summen er 1011. Det ble ingen mente ut av den venstre posisjonen, så $C = 0$. Men den venstre posisjonen hadde mente *inn* lik 1 og mente *ut* lik 0, og de er forskjellige, så $V = 1$.

Kontroll: som fortegnsløse tall er $0110 + 0101 = 6 + 5 = 11$, og 1011 er 11, så svaret er riktig. Som fortegnstall i toerkomplement er 1011 lik -5, og $6 + 5$ er ikke -5. Området i fire bit er -8 til $+7$, og 11 får ikke plass. Flagget har rett begge ganger: ingen mente, men overflyt.

25. Omtrent 32 nanosekunder, altså fire ganger så lenge. Tiden gjennom en mentekjede vokser rett proporsjonalt med antall bit, fordi hver posisjon må vente på menten fra naboen til høyre før den kan gjøre seg ferdig. Fire ganger så mange posisjoner gir fire ganger så lang kjede.

26. Fordi en ELLER ikke har noen kjede. Hver bitposisjon har sin egen OR-port, og ingen av dem trenger å vite noe om naboene. Alle 64 regner ferdig samtidig, og veien gjennom kretsen er én port lang. En addisjon må sende menten gjennom alle 64 posisjonene etter tur.

Merk at kortere vei gjennom kretsen ikke uten videre betyr færre klokkeslag. Klokka er stilt etter det tregeste steget uansett, slik kapittel 4 påpekte. Men det er nettopp slike lange veier som bestemmer hvor fort klokka kan gå.

27. Tre innganger gir $2^3 = 8$ utganger. Det trengs åtte AND-porter, én per utgang, hver med tre innganger. Og tre invertere, én per inngangssignal, slik at både A og $\overline{A}$ er tilgjengelige for portene.

28. Linje 5 slår ut, siden 0101 er 5 i totallssystemet. AND-porten for den linja får inn $\overline{b_3}$, b_2, $\overline{b_1}$ og b_0. Med opkoden 0101 er $b_3 = 0$, $b_2 = 1$, $b_1 = 0$ og $b_0 = 1$, så alle fire signalene inn i porten er 1, og bare den porten slår ut.

29. I en kombinatorisk krets avhenger utgangen bare av hva som står på inngangene akkurat nå. Setter du de samme verdiene inn igjen, får du alltid det samme ut. En sekvensiell krets har tilbakekobling, så utgangen avhenger også av hva som har skjedd før — den har en tilstand. Eksempler fra kapitlet: fulladderen og dekoderen er kombinatoriske, mens de to NOR-portene som holder hverandre er sekvensielle.

30. Da ville kretsen regnet ut $a + \overline{b}$, altså $a - b - 1$. Svaret blir én for lite. Grunnen er at $-b$ i toerkomplement er «inverter alle bitene *og legg til 1*», og menten inn er nettopp den ene-eren. Inverteringen alene gir enerkomplementet, som er én mindre enn det du ville ha.

Fasit til Selv-Test F6

Transistor, strøm og spenning

Fasit til selv-testen i kapittel 6, 6.14 på side 170.

1. En transistor er en bryter som styres av et elektrisk signal i stedet for av en finger.
2. To: nMOS og pMOS. nMOS leder når styresignalet er 1, pMOS leder når det er 0. De gjør altså motsatt ting av hverandre, og det er nettopp derfor de brukes sammen.
3. VDD er forsyningsspenningen, det høye punktet der strømmen kommer inn i kretsen. GND er jord, 0 volt, og referansepunktet alt annet måles mot.
4. VDD svarer til logisk 1, GND til logisk 0.
5. Boblen betyr «denne gjør det motsatte». På en transistor betyr den at den leder når styresignalet er 0. Det er samme tegn som på portsymbolene i kapittel 5, og det betyr det samme der.
6. To. Én pMOS opp mot VDD og én nMOS ned mot GND.
7. pMOS-en leder, fordi den leder ved 0. nMOS-en står åpen. Utgangen får dermed fri vei opp til VDD, og utgangen blir 1.
8. *Complementary MOS*. Navnet er beskrivende fordi de to transistortypene utfyller hverandre: står den ene i ledende stilling, står den andre i motsatt stilling, alltid.
9. Fordi de to transistorene aldri leder samtidig så lenge inngangen ligger i ro. Det finnes derfor ingen gjennomgående vei fra VDD til GND, og strømmen har ingen steds å gå. Strømmen går bare i det korte øyeblikket utgangen skifter — og da er begge så vidt innom ledende tilstand samtidig. I tillegg kommer den lille lekkasjen fra seksjon 6.10.
10. NAND har fire. AND har seks.
11. Fordi et slikt trinn alltid snur signalet. NAND er ett trinn. AND må være en NAND med en inverter etter seg, altså to trinn, og inverteren koster to transistorer til.
12. De to nMOS-ene nede står i serie, de to pMOS-ene oppe står i parallell. Serien nede gjør at utgangen bare kobles til jord når *begge* inngangene er 1, altså at utgangen bare blir 0 da. Parallellen oppe gjør at det holder at *én* inngang er 0 for at utgangen skal kobles til VDD. Til sammen er det akkurat NAND.
13. Fordi spenningen på en ledning kan ha hvilken som helst verdi mellom 0 og VDD. Skulle 0 bety nøyaktig 0,000 volt, ville nesten ingen signaler vært gyldige. I stedet får hvert av de to tallene et intervall å bo i: alt under en nedre grense teller som 0, alt over en øvre grense teller som 1.

14. Beltet gjør at ingen signaler havner på grensa ved et uhell. Et signal som ligger nøyaktig på en enkelt grense, måtte kretsen valgt for, og den kunne valgt ulikt fra gang til gang. Med et bredt ingenmannsland imellom er det god avstand til begge sider.

15. Utgangen blir skarp og ren. Porten bærer ikke signalet videre — den leser bare om det er over eller under grensa, og henter så en fersk verdi fra VDD eller GND. Utgangen er derfor like sterk uansett hvor slapp inngangen var.

16. Fordi hver kopi av en digital fil leser hvert bit som enten over eller under grensa, og skriver det på nytt helt skarpt. Småfeil blir kastet i hvert ledd og samler seg aldri opp. På en kassett blir hver liten unøyaktighet med videre og legger seg oppå den forrige, så etter tusen ledd er det bare susing igjen.

17. Kapasitansen er størrelsen på badekaret som må fylles for at utgangen skal komme seg opp til 1. Den består av ledningen selv og av styreledningene til alle portene signalet går til. Det er feil å si at noen har satt den inn, fordi den oppstår av seg selv av at ledninger og transistorer er fysiske ting som tar plass. Alle jobber for å få den ned.

18. Tiden fra inngangen til en port endrer seg, til utgangen har kommet seg over på den nye verdien.

19. Et nanosekund er 1000 pikosekunder. Med 5 ps per port rekker 200 porter å skifte.

20. Den kritiske veien er den lengste veien fra en inngang til en utgang. Den betyr alene noe for farten fordi alle portene jobber samtidig: kretsen er ferdig når den tregeste veien er ferdig. Å gjøre noe raskere på en kortere vei endrer ingenting, for den sto likevel og ventet.

21. Fem porter på den lengste veien gir $5 \times 5 = 25$ ps. De to andre veiene går samtidig og spiller ingen rolle.

22. Fordi bare tre av de fem portene ligger etter hverandre. Veien XOR1 $\rightarrow$ AND2 $\rightarrow$ OR er tre ledd, altså 15 ps. AND1 og XOR2 jobber samtidig med de andre, og tidene deres legges ikke til.

23. Etter 15 ps. Porten kan ikke begynne før begge inngangene er klare, altså ikke før etter 10 ps, og så bruker den sine egne 5.

24. Åtte bit på 120 ps betyr 15 ps per ledd. 32 bit gir da $32 \times 15 = 480$ ps. Svaret er annerledes enn i oppgave 21 fordi leddene her *ikke* går samtidig: hver fulladder må vente på menten fra den forrige, så tiden vokser med antall bit.

25. Klokkeperioden er tiden fra ett sted i klokkemønsteret til samme sted neste gang — for eksempel fra en stigende flanke til neste stigende flanke. En stigende flanke er øyeblikket signalet går fra 0 til 1.

26. $1/(4 \cdot 10^9) = 250 \cdot 10^{-12}$ s, altså 250 ps.

27. $1/(400 \cdot 10^{-12}) = 2{,}5 \cdot 10^9$ Hz, altså 2,5 GHz.

28. Fordi alt som skal bli ferdig innen neste flanke, må rekke det. Er perioden kortere enn den kritiske veien, rekker ikke den tregeste veien fram før resultatet lagres. Da lagres et halvferdig svar, og maskinen regner feil — ikke litt feil, men helt feil, og på en måte som er vanskelig å spore.

29. For det første er den kritiske veien aldri én port, men mange på rad. For det andre trenger perioden margin for varme, for produksjonsvariasjon og for at klokkesignalet ikke kommer helt samtidig overalt. For det tredje ville effekten blitt så høy at brikken brant opp, siden effekten vokser med frekvensen.

30. At antallet transistorer man får plass til på en brikke dobler seg omtrent annethvert år. «Lov» er misvisende fordi ingenting i fysikken krever det. Moore observerte en trend og spådde at den ville fortsette, og så la bransjen planene sine etter spådommen. Den ble dermed i stor grad selvoppfyllende.

31. Fordi høyere spenning gjør to ting samtidig: det er mer ladning som må flyttes, og hver ladning må løftes høyere. Frekvensen gjør bare én ting — den bestemmer hvor ofte flyttingen skjer. Derfor står spenningen i andre potens i effektuttrykket, mens frekvensen står i første.

32. Effekten dobles. Med spenningen uendret er effekten rett proporsjonal med frekvensen. (I praksis krever en dobling av frekvensen som regel litt høyere spenning også, og da blir økningen større enn det dobbelte.)

33. Lekkasje er strømmen som slipper gjennom en transistor som skal være stengt. Den er et større problem i dag fordi transistorene er blitt så små: jo tynnere og mindre de er, jo mer lekker de. Samtidig er regningen uavhengig av om brikken gjør noe — den løper hele tiden, også når maskinen står i ro.

34. Fordi en transistor trenger en viss spenning for å skille klart mellom på og av. Under den grensa leder den litt selv når den skal være stengt, og den blir treg. I tillegg krymper støymarginen fra seksjon 6.6, slik at signalene lettere blir misforstått. I dag ligger gulvet rundt 0,7 til 0,8 volt.

35. Så lenge transistorene krympet, kunne forsyningsspenningen senkes i takt. Siden spenningen teller dobbelt i effektregnskapet, falt effekten per transistor raskt nok til at man kunne pakke flere inn og kjøre dem fortere uten at brikken ble varmere. Rundt 2004 sluttet spenningen å kunne følge krympingen nedover, og fra da av ble hver ny gigahertz betalt nesten rett i varme. Gulvet den nærmet seg, rundt 0,7 til 0,8 volt, er der vi ligger i dag. Transistorene fortsatte likevel å bli flere, så bransjen brukte dem til å lage flere kjerner i stedet for én raskere.

36. Den varmer rommet med 60 watt, nøyaktig som en 60 watts varmeovn. Svaret er så enkelt fordi maskinen ikke leverer energi tilbake i noen annen form: den løfter ingenting og setter ingenting i varig bevegelse. All energien som går inn, ender som varme.

37. At brikken er laget i en bestemt produksjonsgenerasjon. Tallet var en gang en fysisk lengde i transistoren, men den koblingen er brutt. I en «3 nm-prosess» er den tilsvarende lengden snarere rundt 15 nanometer. Nodenavnet fungerer i dag som et generasjonsnavn, omtrent som et modellår.

38. Feilen er begrunnelsen. I vanlige, statiske logikkretser lekker ikke verdien ut mens inngangen ligger fast: en av transistorene står åpen og holder utgangen aktivt koblet til VDD eller GND. Den riktige grunnen til at vi trenger klokka er at ulike veier gjennom kretsen blir ferdige til ulik tid, og at noen må si fra når alle er det. (Studenten tenker trolig på dynamisk logikk eller på DRAM, der verdien faktisk må friskes opp.)

39. Det riktige er at hvert enkelt CMOS-*trinn* er inverterende. Du kan ikke bygge en AND av fire transistorer i ett trinn. Det gale er konklusjonen: AND-porter finnes, og de står i alle kretsbiblioteker. De er bare bygget som en NAND med en inverter etter seg, altså seks transistorer i to trinn.

40. Man sparer transistorer. Hver port du fjerner, er fire til seks transistorer som slipper å være der. En spart port koster mindre av tre ting samtidig: *plass* på brikken, *strøm* fordi det er ett badekar mindre å fylle og tømme, og *varme*, siden all den strømmen ender som varme. Ligger porten i tillegg på den kritiske veien, sparer du også *tid*.

Fasit til Selv-Test F7

Minnehierarki og pipeline

Fasit til selv-testen i kapittel 7, 7.10 på side 198.

1. Fordi stort og raskt trekker i hver sin retning. Et stort minne har flere adresser å dekode og lengre ledninger internt, og begge deler tar tid. I tillegg er den raske teknologien (SRAM) ti til tjue ganger så arealkrevende per bit som den tette (DRAM), så et stort og raskt minne ville også vært urimelig dyrt.
2. Raskt, stort og billig. Du får to av dem.
3. Register, L1-cache, L2-cache, RAM, SSD.
4. Primærminne er det prosessoren kan nå direkte — registre, cache og RAM — og det mister innholdet når strømmen går. Sekundærminne er SSD og harddisk, som husker uten strøm, men som er mange tusen ganger tregere.
5. Fordi det bare lå i primærminnet. Et dokument du ikke har lagret, finnes bare i RAM, og RAM er flyktig. Lagring betyr nettopp å flytte det ned til sekundærminnet.
6. Rundt 240 sykluser ved 3 GHz. Størrelsesordenen er det viktige: et par hundre, ikke et par.
7. For det første må signalet gå lenger fysisk, og en lengre ledning har mer kapasitans å fylle. For det andre trengs en større dekoder for å finne fram til riktig celle, og en større dekoder har flere porter på den kritiske veien.
8. Fordi et register ikke har noen adresse å slå opp. Instruksjonen peker rett på det. En cache må først avgjøre om verdien i det hele tatt er der, og hvor den ligger, og den letingen koster tid.
9. At ingenting flyttes bort fra laget under — det kopieres oppover. Cachen inneholder en kopi av deler av RAM, og RAM inneholder en kopi av deler av disken. Originalen blir liggende.
10. Gjennomsnittstiden en minnetilgang tar når både treff og bom er regnet med, vektet etter hvor ofte hver av dem skjer.
11. $0{,}9 \times 1 + 0{,}1 \times 100 = 0{,}9 + 10 = 10{,}9$ ns.
12. $0{,}98 \times 1 + 0{,}02 \times 100 = 0{,}98 + 2 = 2{,}98$ ns. Det er $10{,}9/2{,}98 \approx 3{,}7$ ganger raskere — av en forbedring på åtte prosentpoeng i treffrate.
13. Fordi en bom er hundre ganger dyrere enn et treff. Da må den være hundre ganger sjeldnere bare for å koste like mye. Ved 90 prosent treffrate er bommene ti ganger sjeldnere, men hver av dem koster hundre ganger mer — og da ender de opp med å dominere regnestykket fullstendig.

14. At noe du nettopp brukte, sannsynligvis brukes igjen snart. Et eksempel er en variabel som oppdateres hver runde i en løkke, som `sum` eller en teller.

15. At naboen til det du nettopp brukte, sannsynligvis brukes snart. Et eksempel er en liste som gås gjennom i rekkefølge.

16. Stedslokaliteten. Cachelinja henter inn naboene samtidig, og de er bare nyttige hvis programmet faktisk kommer til å be om dem.

17. Instruksjonene har lokalitet helt på egen hånd, og ofte sterkere enn dataene. En løkke på ti linjer som kjøres en million ganger, er ti linjer med ekstrem tidslokalitet. Og instruksjoner som ikke er hopp, ligger etter hverandre i minnet, noe som gir stedslokalitet.

18. Den minste enheten cachen håndterer — en sammenhengende klump av minnet som hentes og kastes ut under ett. Typisk 64 byte.

19. Seksten. $64 \div 4 = 16$.

20. Et treff betyr at verdien allerede lå i cachen. En bom betyr at den ikke gjorde det, og at den må hentes fra et lavere nivå.

21. Først sendes forespørselen videre til neste nivå: L1 spør L2, L2 spør L3, L3 spør RAM. Så hentes hele cachelinja som verdien ligger i, ikke bare den ene verdien. Til slutt legges linja inn i cachen, og noe annet må vike for å gi plass. Prosessoren venter mens alt dette skjer.

22. Tre bommer — én for hver linje som må hentes — og 21 treff. Treffraten er $21/24 = 87{,}5$ prosent.

23. Nå trengs seks linjer, altså seks bommer og 18 treff. Treffraten faller til $18/24 = 75$ prosent. Kortere linjer betyr at hver bom gir færre naboer på kjøpet, og da blir det flere bommer på samme datamengde.

24. Fordi avveiningen mellom fart og størrelse ikke forsvinner bare fordi man har bygget én cache. En cache som skal svare på fire sykluser, må være liten. En som skal romme flere megabyte, kan ikke svare på fire sykluser. Begge er nyttige, så maskinen har begge.

25. For å kunne hente en instruksjon og en dataverdi i samme syklus. Én cache kan bare svare på ett spørsmål om gangen, og pipelinen trenger begge deler samtidig. Se også oppgave 50.

26. Gjennomskriving oppdaterer både cachen og minnet med en gang. Tilbakeskriving skriver bare i cachen, merker linja som endret, og skriver den ned først når linja må vike.

27. Tilbakeskriving. Prisen er at det finnes et tidsrom der cachen og minnet er uenige om hva verdien er, og maskinen må holde orden på hvilke linjer som er endret og sørge for at ingen leser den utdaterte kopien.

28. Problemet med at flere kjerner kan ha hver sin kopi av den samme cachelinja. Endrer den ene sin kopi, er de andres kopier plutselig feil. Det oppstår fordi hver kjerne har sin egen private L1-cache.

29. Ikke i den forstand at du kan styre hva som ligger der. Noen instruksjonssett har *prefetch*-instruksjoner som sier fra om at en adresse snart trengs, men de er hint: de endrer bare farten, aldri hva programmet regner ut. Den praktiske konsekvensen er at håndtaket du har som programmerer, er *rekkefølgen* du går gjennom data i. Den bestemmer du helt selv, og den avgjør om du får treff eller bom.

30. Et rutenett ligger utbrettet radvis i minnet, så naboene i en rad ligger inntil hverandre mens naboene i en kolonne ligger 32 000 byte fra hverandre. Den radvise løkka får derfor femten gratis treff for hver bom, fordi cachelinja tar med naboene. Den kolonnevise bruker aldri naboene den fikk, og bommer på hver eneste tilgang.

31. Fordi hele rutenettet fikk plass i cachen. Når alt ligger der uansett, er det ingen bommer å spare, og rekkefølgen slutter å bety noe. Det er samtidig den beste bekreftelsen på at det faktisk er cachen vi måler.

32. At en oppgave deles i trinn, og at flere enheter er inne i hvert sitt trinn samtidig. Et samlebånd. Oppvaskeksempelet: én vasker, én skyller og én tørker, slik at tre tallerkener er under arbeid på én gang.

33. Instruction Fetch: instruksjonen hentes fra minnet og programtelleren økes. Decode: instruksjonen tydes, og maskinen finner ut hva den skal gjøre og hvilke verdier den trenger. Execute: arbeidet utføres, for eksempel i ALU-en. Write Back: resultatet skrives tilbake til et register eller til minnet.

34. $4 \times 50 = 200$ sykluser.

35. $3 + 50 = 53$ sykluser. Det er omtrent 3,8 ganger raskere.

36. Fordi de tre syklusene det tar å fylle pipelinen er en fast kostnad som aldri forsvinner. Gevinsten nærmer seg fire når antall instruksjoner vokser, men den når den aldri, for det vil alltid stå $3 + n$ mot $4n$.

37. Latens er hvor lang tid én enkelt instruksjon bruker fra start til slutt. Gjennomstrømning er hvor mange instruksjoner som blir ferdige per syklus. Pipelining endrer bare gjennomstrømningen — latensen er praktisk talt den samme, fire sykluser, med og uten. (Pipelineregistrene legger på en liten smule, men ingenting i nærheten av en firedel.)

38. Det er galt at hver instruksjon går fortere. Hver instruksjon bruker fortsatt fire sykluser fra start til slutt. Det som er riktig, er at instruksjonene blir ferdige mye oftere, fordi flere er inne i maskinen samtidig. Pipelining forbedrer gjennomstrømningen, ikke latensen.

39. Fordi klokka går i takt med det tregeste trinnet. Klokkeperioden må være lang nok til at alle trinnene rekker å bli ferdige, så et trinn som er dobbelt så tregt som de andre, tvinger alle de andre til å vente like lenge. Da forsvinner gevinsten.

40. Kapittel 6 viste at klokkeperioden må romme den kritiske veien gjennom kretsen. Sammenlign med en maskin som gjør hele instruksjonen i én lang klokkeperiode: der er den kritiske veien hele instruksjonen, hent, tyd, regn og lagre i én omgang, og det tvinger fram en lang periode. Deler du arbeidet i fire trinn med et pipelineregister mellom hvert, blir den kritiske veien bare ett trinn lang, altså rundt en firedel, og klokka kan gå omtrent fire ganger så fort. Pipelining er altså måten man forkortet den kritiske veien på, og dermed måten man fikk lov til å skru opp frekvensen. Merk at dette er den *samme* firedoblingen som i tabellen over sykluser, sett fra en annen kant — ikke en gevinst som kommer i tillegg.

41. For det første koster registrene mellom trinnene tid i hver eneste syklus, og til slutt blir det mer registerarbeid enn nyttig arbeid. For det andre lar ikke arbeidet seg dele vilkårlig — en addisjon kan ikke deles i hundre like store biter. For det tredje taper man mer hver gang noe går galt, fordi flere instruksjoner må kastes.

42. Fordi minnetilgang 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 i stedet for å gjemme den inne i utførelsen.

43. En situasjon der pipelinen ikke kan gå videre som planlagt, og må vente eller kaste arbeid.

44. Datahasard: en instruksjon trenger et resultat som ikke er klart ennå, for eksempel `ADD` rett etter `LDA`. Kontrollhasard: et betinget hopp gjør at instruksjonene som allerede er hentet, er feil. Strukturhasard: to trinn vil bruke den samme maskinvaren i samme syklus, for eksempel minnet.

45. En avhengighet er en egenskap ved programmet: instruksjon B trenger resultatet fra instruksjon A, og det er sant uansett hvilken maskin du kjører på. En hasard er en følge av maskinvaren: avhengigheten blir et problem bare hvis maskinen ikke rekker å levere verdien i tide. En smartere maskin kan fjerne hasarden uten at avhengigheten forsvinner.

46. En syklus der et trinn i pipelinen står uten nyttig arbeid fordi det venter.

47. Det er to. `ADD y` trenger verdien `LDA x` henter, og `STO x` trenger resultatet `ADD y` regner ut. Hver av dem koster to sykluser i vår modell, altså fire til sammen: sekvensen bruker ti sykluser i stedet for seks.

48. Fordi maskinen ikke vet hvor den skal hente neste instruksjon fra før hoppet er utført. Et regnestykke endrer ikke hvilken instruksjon som kommer etter, så pipelinen kan trygt hente videre i rekkefølge. Et betinget hopp kan sende programmet to forskjellige steder, og svaret foreligger først i EX-trinnet.

49. Fordi det er flere instruksjoner inne i maskinen samtidig. Med fire trinn rekker maskinen å hente to feil instruksjoner før den oppdager det. Med tjue trinn kan det være femten eller flere, og alle må kastes.

50. Ved å ha to cacher i stedet for én. L1 er delt i en instruksjonscache og en datacache, slik at IF-trinnet og de trinnene som leser data, kan betjenes samtidig uten å stå i kø for hverandre.

51. Cachen gjør ikke RAM raskere i det hele tatt. RAM er nøyaktig like treg som før, og en tur dit koster like mange sykluser. Det cachen gjør, er å sørge for at prosessoren slipper å gå til RAM så ofte. Det riktige er å si at cachen senker den *effektive* aksesstiden, altså gjennomsnittet.

52. De to firedoblingene er den samme gevinsten beskrevet to ganger, og de kan ikke ganges sammen. Enten sammenligner du med en maskin som allerede bruker fire like sykluser per instruksjon — da er klokka den samme, og du vinner på gjennomstrømningen. Eller du sammenligner med en maskin som bruker én lang syklus — da vinner du på klokkefrekvensen, men instruksjonen tar fortsatt fire sykluser. Samme firedobling, to måter å telle den på.

53. En bom i L1 betyr bare at verdien ikke lå i L1. Da spør maskinen L2, så L3, så RAM, og i praksis ligger verdien nesten alltid i RAM. Å måtte gå helt til disken er en helt annen mekanisme, den hører til virtuelt minne, og den er svært sjelden.

54. Begge nekter å vente. Cachen venter ikke på RAM, men holder en kopi nær. Pipelinen venter ikke på at forrige instruksjon skal bli ferdig, men setter i gang med den neste. Begge virker fordi programmer har et mønster — lokalitet i det ene tilfellet, en jevn strøm av uavhengige instruksjoner i det andre. Og begge svikter på samme måte: når mønsteret brytes. Et oppslag langt unna gir en bom, og et hopp gir en tømt pipeline. I begge tilfellene står maskinen stille.

Svar på regneboksene i kapitlet

Avsnitt 7.2 på side 175, «Regn på det selv». Ved 80 prosent treffrate: $0{,}8 \times 1 + 0{,}2 \times 100 = 0{,}8 + 20 = 20{,}8$ ns. Ved 95 prosent: $0{,}95 \times 1 + 0{,}05 \times 100 = 0{,}95 + 5 = 5{,}95$ ns. Programmet går altså rundt 3,5 ganger fortere. Legg merke til at selv ved 95 prosent treffrate kommer fem sjettedeler av tiden fra bommene.

Avsnitt 7.4 på side 183, «Regn en treffrate». Tolv tall, fire per linje: det trengs tre linjer, altså tre bommer og ni treff. Treffraten er $9/12 = 75$ prosent. Med åtte tall per linje trengs to linjer: to bommer og ti treff, altså $10/12 \approx 83$ prosent. Større linjer gir høyere treffrate når programmet går gjennom

data i rekkefølge — men bare da. Hopper programmet rundt, henter de store linjene inn naboer som aldri blir brukt, og da er gevinsten borte.

Avsnitt 7.6 på side 192, «Finn hasarden». Det er to avhengigheter. `ADD b` trenger verdien `LDA a` henter, og `STO c` trenger resultatet `ADD b` regner ut. `LDA d` trenger ingen verdi fra noen av de tre. Den henter inn noe nytt og bryr seg ikke om hva som lå i akkumulatoren fra før, så den kan gå rett gjennom uten å vente.

Fasit til Selv-Test F8

Moderne maskiner og hele stabelen

Fasit til selv-testen i kapittel 8, 8.14 på side 231.

1. Resultatet sendes rett fra utgangen av ett EX-trinn til inngangen av det neste, i stedet for å ta omveien via akkumulatoren. Analogien: du rekker tallet over bordet til medstudenten din i stedet for å legge det i hylla hennes nede i gangen.

2. Datahasarden.

3. Nei. Avhengigheten er en egenskap ved programmet — instruksjon 2 trenger fortsatt resultatet fra instruksjon 1, og det kan ingen maskin endre på. Det som forsvinner er hasarden, altså ventetiden, og den var en egenskap ved maskinvaren.

4. Last-bruk: når verdien skal hentes fra minnet og neste instruksjon trenger den med en gang. I vår firetrinnsmodell leses minnet i EX, så der rekker videresendingen det akkurat. I en pipeline med eget MEM-trinn kommer verdien først ut etter MEM, og da står én boble igjen.

5. Ti sykluser med bobler, seks med forwarding. Seks er det en pipeline uten stans skal bruke på tre instruksjoner: tre sykluser fylling pluss én per instruksjon.

6. At maskinen fører en tabell over hvert hopp den har sett og hva som skjedde sist, og gjetter det samme neste gang. Tabellen er fysisk maskinvare på brikka.

7. Fordi hopp bakover som regel er løkker. En løkke som går n runder, hopper tilbake $n-1$ ganger og lar være én gang, så regelen treffer nesten alltid.

8. Den tilpasser seg det enkelte hoppet i stedet for å behandle alle likt. En `if` som nesten alltid går samme vei, læres på et par runder — og tommelfingerregelen ville gjettet feil på den hver gang hvis den tilfeldigvis er et framoverhopp som nesten alltid tas.

9. To ganger. én gang første runde, før tabellen vet noe om hoppet, og én gang siste runde, når løkka slutter og hoppet ikke tas. 998 riktige av 1000. Dette forutsetter at tabellen starter på «ikke tatt»; starter den på «tatt» — som den gjerne gjør for hopp bakover — bommer den bare én gang.

10. At maskinen utfører instruksjonene den har gjettet på, i stedet for å vente på å få gjetningen bekreftet. Var gjetningen feil, settes registrene tilbake og resultatene forkastes, slik at programmet får samme svar som på en maskin uten spekulasjon.

11. Fordi kostnaden ved å bomme bare betales i de fem prosentene. I de nittifem andre slipper maskinen å stå stille i det hele tatt. Prisen er strøm brukt på arbeid som av og til kastes.

12. Pipelining deler én instruksjon i trinn, slik at flere instruksjoner er inne i maskinen samtidig — men i hvert sitt trinn. Superskalar har flere hele regneenheter, slik at flere instruksjoner kan være i *samme* trinn samtidig.

13. Fire. En superskalar maskin med fire regneenheter kan i beste fall bli ferdig med fire instruksjoner per syklus.

14. Fordi addisjonene i A henger sammen: hver av dem må vente på resultatet fra den forrige, så bare én regneenhet har noe å gjøre. I B er de fire kjedene uavhengige, og da kan alle fire regneenhetene arbeide samtidig.

15. En kjede der hver addisjon venter på den forrige, kan ikke gå fortere enn én addisjon per klokkesyklus. Måler den 0,249 nanosekunder per addisjon, er klokkeperioden 0,249 ns, og frekvensen er én delt på det, altså 4,0 GHz. To forutsetninger ligger under. Kjeden må faktisk være avhengig hele veien — klarer kompilatoren å bryte den, måler du noe annet. Og addisjonen må ta nøyaktig én syklus; hadde vi brukt divisjon, som tar mange, ville samme regnestykke gitt et helt galt svar.

16. At maskinen ser på et vindu av de nærmeste instruksjonene, plukker ut dem som har alt de trenger, og setter dem i gang — uavhengig av hvilken rekkefølge de står i i programmet.

17. Fordi programmet ellers ville fått feil svar. Resultatene holdes tilbake til tur-en kommer, og skrives først da inn der programmet kan se dem. Kontrakten er at utsiden skal se ut nøyaktig som den enkle modellen fra kapittel 3.

18. To sett registre på én kjerne, slik at to tråder kan være inne i maskineriet samtidig og fylle hverandres ventetid. Det gir ikke det dobbelte, fordi de to trådene *deler* regneenheter og cache — bare registrene og litt bokholderi er doblet. Gevinsten er typisk 10–30 prosent, og kan være negativ hvis trådene dytter hverandres data ut av L1.

19. En kjerne er fysisk maskinvare: en komplett prosessor med egne registre, egen ALU og egen pipeline. En prosess er et program som kjører, med sitt eget minne. En tråd er en utføringsvei inne i en prosess, og flere tråder i samme prosess deler minne. Kort: trådene er arbeidet, kjernene er arbeiderne.

20. Ja. Operativsystemet fordeler trådene på de åtte kjernene og bytter mellom dem så fort at det ser samtidig ut. Åtte av dem kjører faktisk samtidig i hvert øyeblikk — seksten hvis kjernene har SMT. Resten venter, enten på tur eller på at noe utenfor maskinen skal bli ferdig.

21. At den delen av jobben som ikke lar seg dele, setter taket for hvor raskt det kan gå — uansett hvor mange kjerner du legger til.

22. 40 sekunder. Den delelige delen blir $80 \div 4 = 20$, den udelelige blir stående på 20, og til sammen er det 40. Det er 2,5 ganger raskere.

23. 20 sekunder, altså fem ganger raskere. Den delelige delen går mot null, men de 20 udelelige sekundene står igjen uansett hvor mange kjerner du har.

24. Fordi det bare har én tråd. Det kjører på én kjerne, og de sju andre har ingenting å gjøre for akkurat det programmet. Hele den delelige andelen er null, og da sier Amdahls lov at gevinsten er én gang.

25. Å dele ut arbeid og samle det inn igjen tar tid i seg selv. Tråder som skal røre det samme, må vente på hverandre og stille seg i kø. Og cache-koherensen koster: hver gang én kjerne skriver til en linje, blir de andres kopi ugyldig.

26. At to kjerner skriver til to forskjellige tall som tilfeldigvis ligger i den samme cachelinja. Det overraskende er at de aldri rører hverandres data — men maskinvaren holder styr på linjer og ikke på enkelttall, så de får full koherenstrafikk likevel, og programmet kan bli tregere enn med én tråd.

27. Fordi den er bygget for oppgaver der man vet på forhånd at arbeidet er parallelt. Da trenger den ikke maskineriet som leter etter parallellitet, og arealet kan brukes på regning i stedet. En CPU-kjerne har en håndfull regneenheter og mye bokholderi; en GPU har byttet bokholderiet mot flere tusen enheter.

28. Den har ikke bokholderiet: forgreningsprediktor, spekulasjon, det store omsorteringsvinduet og de dype cachene. I en CPU går størstedelen av arealet til nettopp det, og ingenting av det regner ut noe.

29. GPU-en har tusenvis av oppgaver liggende, og bytter til en annen når én av dem må vente på minnet. Ventetiden fylles med annet arbeid. CPU-en prøver i stedet å gjøre ventetiden kort, ved å holde dataene nær i cachen. Det er latens mot gjennomstrømning fra kapittel 7.

30. CPU, i de fleste tilfeller. Sortering er full av sammenligninger, altså hopp som går forskjellig vei, og stegene henger sammen: resultatet av én sammenligning avgjør hva som skjer videre. Det er det GPU-en er dårligst på. (Det finnes sorteringsalgoritmer laget for GPU, og for svært store datamengder kan de vinne — men da er algoritmen skrevet om, ikke bare flyttet.)

31. Begge veier går gjennom en fastfunksjonskrets, så det er ikke der forskjellen ligger. Å pakke *sammen* video er en vesentlig tyngre jobb enn å pakke den ut — maskinen må lete etter hva som er likt mellom bilderutene. Og under opptak går i tillegg kamerasensoren, bildebehandlingen, skjermen og lagringen samtidig.

32. En krets som bare kan gjøre én ting, og ikke kan programmeres. Den er strømgjerrig fordi den ikke har noe bokholderi: den henter ikke instruksjoner, tyder dem ikke, gjetter ikke på hopp og sorterer ingenting.

33. Den tar inn en håndfull tall, ganger hvert av dem med en vekt, legger produktene sammen og sammenligner summen med en terskel. En AND-port gjør det samme: den gir 1 når summen av inngangene er minst 2. Forskjellen er at portens vekter og terskel ligger fast, mens nettets kan stilles inn.

34. Fordi regnestykket i seg selv er så enkelt — gange og legge til — at kostnaden ligger i å flytte tallene fram og tilbake, ikke i å regne på dem. Derfor bygges brikka med mange små gangeenheter som sender tallene videre til hverandre i stedet for å hente fra minnet hver gang.

35. Det virker uten nett, det går raskere når svaret trengs innen noen millisekunder, og dataene forlater ikke telefonen.

36. En ledning en enhet kan trekke i når noe har skjedd, slik at prosessoren får beskjed. Alternativet er at programmet spør med jevne mellomrom — enten for sjelden, så maskinen blir treg, eller for ofte, så det går med tid til å spørre om noe som nesten aldri har skjedd.

37. Fordi det bare er å se på et signal som allerede ligger der. Det er ingen ekstra minnetilgang og ingen ekstra instruksjon.

38. Den gjør ferdig instruksjonen den holder på med. Den lagrer unna programtelleren og registrene. Den slår opp i en tabell hvilken rutine som hører til enheten, og setter programtelleren dit. Når rutinen er ferdig, legges alt tilbake og programmet fortsetter.

39. Fordi operativsystemet ellers aldri ville fått kontrollen tilbake fra et program som kjører. En klokkekrets utløser et avbrudd noen tusen ganger i sekundet, og da kan operativsystemet bestemme hvem som skal få kjøre videre. Uten den ledningen ville et program i evig løkke eid maskinen.

40. En egen krets som flytter data mellom en enhet og minnet uten å involvere prosessoren. Den løser at prosessoren ellers ville brukt millioner av klokkesykluser på å flytte data i et tempo satt av disken.

41. Hver enhet får tildelt noen adresser i det vanlige adresserommet, og prosessoren bruker helt vanlige lese- og skriveinstruksjoner på dem. Skriver du til en bestemt adresse, havner verdien i en kontrollkrets i stedet for i minnet. Det kalles minneavbildet inn- og utdata.

42. At hvert program får sitt eget sett adresser som begynner på null, og at maskinvaren oversetter dem til fysiske adresser et helt annet sted. Det viktigste det gir er isolasjon: programmer kan ikke rote i hverandres minne.

43. Fordi de to adressene oversettes til hver sin fysiske adresse. Program A sin 42 kan bli fysisk 300 042, og program B sin 42 kan bli 900 042.

44. Fordi det ikke er et forbud i det hele tatt. Program A kan ikke *uttrykke* en adresse som tilhører program B — det finnes ikke noe tall A kan skrive som oversettes dit. Et forbud kan omgås; en manglende mulighet kan ikke.

45. En side er en bit av minnet på typisk fire kilobyte. Oversettelsen skjer per side fordi en tabell med én rad per adresse ville vært like stor som minnet selv.

46. En liten, rask cache for de siste oversettelsene. Den måtte finnes fordi oversettelsen skjer ved hver eneste minnetilgang — et tabelloppslag i minnet for hvert oppslag i minnet ville doblet arbeidet.

47. Lokalitet. Et program bruker de samme sidene om igjen, og naboadresser ligger som regel på samme side, så de siste oversettelsene er nesten alltid de man trenger igjen. Treffraten ligger over 99 prosent.

48. Fra kapittel 7: at cachen ikke endrer hvilket svar programmet får, bare hvor lang tid det tar. Og fra avsnitt 8.2: at spekulasjon ryddes bort og ikke er synlig i resultatet. Begge er sanne, og sammen er de et hull — for det som ryddes er registrene, ikke cachen, og cachen er målbar i tid.

49. Programmet har en sjekk på om en indeks er innenfor. Angriperen har trent prediktoren med tusen lovlige oppslag, så maskinen gjetter «innenfor» også når indeksen er langt utenfor. Mens den gjetter, utfører den oppslaget: den leser en verdi den ikke skulle lest, og bruker den som *adresse* for et oppslag til. Gjetningen viser seg å være feil, og maskinen rydder opp — programmet oppfører seg riktig, og oppslaget blir aldri gjort. Men cachen holder fortsatt på linja fra det tredje steget, og hvilken linje det er, ble bestemt av den hemmelige verdien.

50. Fordi opprydningen er laget for å gjenopprette *resultatet*, ikke sporene. Registrene settes tilbake fordi de bestemmer hva programmet regner ut. Cachen påvirker ikke svaret, bare farten, og ble derfor aldri regnet som noe som måtte ryddes.

51. Ved å måle tid. Hun kan ikke lese verdien, men hun kan gjøre en rekke helt lovlige oppslag og ta tiden på dem. Det ene som går raskest, lå allerede i cachen. Og siden adressen det forkastede oppslaget brukte ble bestemt av den hemmelige verdien, forteller det hvilken verdi det var: går oppslag nummer 65 raskest, var verdien 65.

52. Det finnes ingen. Sjekken er riktig skrevet, programmet er korrekt, og prosessoren gjør nøyaktig det den er designet for. Lekkasjen ligger i avstanden mellom to riktige beskrivelser av samme maskin: den ene om hvilke svar du får, den andre om hvor lang tid du bruker på å få dem.

53. Fordi problemet sitter i selve ideen om å spekulere, og det er den ideen som gjør maskinen rask. Tiltakene koster derfor enten ytelse, eller nye brikker, eller begge deler.

54. CISC har mange og sammensatte instruksjoner som hver gjør mye; RISC har få, enkle og like lange instruksjoner som hver gjør én ting, og mange registre.

55. Fordi pipelining krever at alle trinn tar like lang tid, og det krever at instruksjonene er omtrent like store jobber. RISC-instruksjoner er det. En CISC-instruksjon som kopierer en hel datablokk, kan ta tusen sykluser og stopper hele samlebåndet. At RISC-instruksjoner har samme lengde hjelper også: IF-trinnet vet hvor neste instruksjon begynner uten å tyde den forrige.

56. Fordi den ikke er en CISC-maskin på innsiden. Dekodetrinnet oversetter hver CISC-instruksjon til en håndfull små, like mikro-operasjoner, og det er de som går gjennom pipelinen. Innsiden er en RISC-maskin; utsiden er x86 fordi programvaren er skrevet for x86.

57. Fordi LMC har sine elleve instruksjoner av en helt annen grunn: den er en undervisningsmodell som skal få plass på en tavle. RISC-maskiner har få instruksjoner for å kunne pipelinere dem, og de har til gjengjeld mange registre — gjerne 32 — mens LMC har akkumulatoren og ingenting annet.

58. Python-variabel, prosess, instruksjonsfelt, cachelinje, fulladder, transistor. (Python-variabelen hører til programlaget, prosessen til operativsystemet under det, instruksjonsfeltet til instruksjonssettet, cachelinja til mikroarkitekturen, fulladderen til logikken og transistoren til kretslaget — samme rekkefølge som i tabell 8.7.)

59. (1) Linja du skrev, som sier hva og ikke hvordan. (2) Oversettelsen til tre ordrer: hent, legg til, lagre. (3) Ordren som tall, delt i opkodefelt og adressefelt. (4) Verdiene som bitmønstre, med toerkomplement for negative tall. (5) Hent–utfør i datastien: programteller, MAR, MDR, IR, dekoder, ALU, akkumulator. (6) ALU-en som fulladdere satt i rekke, bygget av XOR, AND og OR. (7) Hver port som fire transistorer i CMOS. (8) Hver transistor som en bryter som fyller eller tømmer et bittelite kar, på rundt fem pikosekunder.

60. For eksempel: instruksjonen lå i L1-cachen (bare tiden), adressen måtte oversettes fra virtuell til fysisk via TLB-en (bare tiden), og instruksjonen var én av fire inne i pipelinen samtidig (bare tiden). Alle tre endret bare tiden — det er hele poenget med dem.

61. Til alt språket gjør for deg: finne ut hva `a` og `b` er for slags verdier akkurat nå, sjekke at de kan legges sammen, lage et nytt objekt til svaret, og slå opp hvilken instruksjon som kommer etter denne. Det er ikke sløsing, det er tjenester — det er derfor du kan skrive `a + b` uten å vite hvor mange bit svaret trenger.

62. De to trådene deler regneenheter, cache og alt annet. Bare registrene og litt bokholderi er doblet. Du får 10–30 prosent, ikke 100, og av og til mindre enn null. Operativsystemet *ser* to kjerner, men det er ikke det samme som å ha dem.

63. En GPU-kjerne er faktisk tregere enn en CPU-kjerne — den er enklere og har ingen av triksene fra avsnitt 8.2. Det GPU-en har er mange av dem, og det hjelper bare når oppgaven består av mange like, uavhengige biter. Et vanlig program med `if`-er og steg som må skje i rekkefølge, blir mye tregere på en GPU.

64. Åtte kjerner gjør bare de programmene raskere som er skrevet for å bruke flere tråder, og selv da setter Amdahls lov et tak: med nitti prosent parallelliserbar kode gir åtte kjerner 4,7 ganger, ikke 8. Et program som er skrevet som en rett strek, blir ikke raskere i det hele tatt. Det kjernene derimot gjør, er at flere *programmer* kan kjøre samtidig uten å trenge seg om den samme kjernen.

65. Begge halvdelene er galt. Det er ingen feil i prosessoren — den gjør nøyaktig det den er designet for, og den gir riktige svar; problemet er at designet etterlater et spor. Og det ble ikke rettet med én oppdatering: svaret har vært tredelt, med endringer i maskinvare, i programvare, og en reell ytelseskostnad som fortsatt betales.

66. Isolasjonen mangler, og den er den viktigste grunnen til at mekanismen finnes. Å kunne bruke mer minne enn man har, er en nyttig bivirkning — men den delen brukes mindre i dag enn før, siden maskiner har mye RAM. Isolasjonen brukes hele tiden.

67. At ingen av dem løser problemet sitt. De flytter konsekvensen til et sted der den koster mindre: forwarding fjerner ventetiden men ikke avhengigheten, prediksjon tar kostnaden sjelden i stedet for alltid, spekulasjon fyller ventetiden med arbeid som kanskje kastes, og flere kjerner går rundt varmeveggen i stedet for gjennom den. Prisen er den samme hver gang: maskinen blir mer komplisert innvendig mens den later som om ingenting er endret utvendig. I avsnitt 8.7 ble nettopp den forskjellen mellom innsiden og utsiden synlig, og da var den et sikkerhetshull.

Register

adressebuss, 89
adresseringsmåte, 89
akkumulator, 62, 85
aktiveringslinje, 88
ALU, 91
Amdahls lov, 211
AND-port, 113
ASCII, 34
assembler, 57, 67
assemblerspråk, 56
avbrudd, 217

BCD, 26
bit, 14
boble, 193
bom, 179
buss, 89
byte, 24
bytekode, 57

cache, 175
cache-koherens, 186
cachelinje, 183
carry, 128
CISC, 223
CMOS, 151

databuss, 89
datasti, 96
De Morgans lover, 125
dekoder, 95, 136
DMA, 218
DRAM, 177

effektiv aksesstid, 179
etikett, 71

fargedybde, 45
fargetabell, 45
fastfunksjonskrets, 215
flagg, 93
flyttall, 39
flyttallsenhet, 92
forgreningsprediksjon, 205
fortegnsbit, 28
forwarding, 204
frekvens, 161
fulladder, 128

generelle registre, 85
GND, 148
GPU, 213

halvadder, 128
hasard, 192
heksadesimal, 22
hent-utfør-syklusen, 61
høynivåspråk, 56

IEEE 754, 40
indeksregister, 73
indirekte adressering, 73
instruksjonsregister, 85
instruksjonssettarkitektur, 80
inverter, 113
ISA, 80
ISO 8859-1, 36

JPEG, 46

kapasitans, 156
kjerne, 186, 210
klokke, 98
klokkeperiode, 161
kodepunkt, 36
koding, 36
kombinatorisk krets, 138
kompilator, 57
komprimering
    tapsfri, 46

kontrollbuss, 89
kontrollenhet, 95
krets, 112
kritisk vei, 158
kvantisering, 47

lagret program, 99
lekkasje, 166
lese/skriv-linje, 88
linje, 89
Little Man Computer, 57
logisk funksjon, 111
løpelengdekoding, 46

MAR, 87
maskinkode, 56
MDR, 87
mikroarkitektur, 80
minneadresseregister, 87
minnedataregister, 87
mnemonic, 64
modulo, 21
mojibake, 34
Moores lov, 163
multiplekser, 137

NAND-port, 115
nMOS, 146
NOR-port, 115
NOT-port, 113
NPU, 216
Nyquist, 48

oktal, 23
oktett, 25
opkode, 60
OR-port, 114
out-of-order, 208
overflyt, 31

piksel, 44
pipeline, 188
pipelineregister, 190
pMOS, 146
port, logisk, 110
portforsinkelse, 157
primærminne, 178
programteller, 62, 85
prosess, 210
pseudoinstruksjon, 72
psykoakustikk, 50

register, 62, 85
registerfil, 85
RGB, 44
ripple carry adder, 131
RISC, 223

sampling, 47
sannhetsverditabell, 117
sekundærminne, 179
sekvensiell krets, 138
seriell buss, 90
side, 219
signalgjenoppretting, 155
skjevhet, 90
SMT, 209
spekulativ utføring, 206
spenning, 147
SRAM, 177
statusregister, 93
stedslokalitet, 181
stikkordregister, 3
stopper opp, 193
strøm, 147
støymargin, 154
sum av produkter, 121
superskalar, 206

tidslokalitet, 181
toerkomplement, 4, 27
tolk, 57
transistor, 14, 146
treff, 179
treffrate, 179
tråd, 209, 210
tømmes, 194

Unicode, 36
UTF-8, 36

varmeveggen, 166
VDD, 147
vekt (i tallsystem), 15
vektorgrafikk, 47
virtuelt minne, 179, 219
von Neumann-arkitektur, 99
von Neumann-flaskehalsen, 100

XNOR-port, 115
XOR-port, 114

Del II

Appendiks

Tillegg A

Datahistorie

Appendikset er ikke skrevet

Historien finnes i dag to steder, fortalt to ganger: i `P05-AvanserteProsessorer` og i `M06-transistor`. Her skal den stå én gang.

Tillegg B

Forenkling med Karnaugh-diagram

Dette er ikke pensum

Ingenting i dette appendikset kommer på eksamen, og du trenger det ikke for å løse en eneste oppgave i IN1020.
Det står her fordi seksjon 5.6 på side 122 endte med en innrømmelse: med reglene i kapitlet vet du aldri helt sikkert når du er ferdig å forenkle. Karnaugh-diagrammet fjerner den usikkerheten for små uttrykk, og det gjør det på en måte som er overraskende visuell.
Regn med en halvtime.

Kapittel 5 forenklet boolske uttrykk ved å lete. Du ser etter felles faktorer, prøver absorpsjonsreglene, og stopper når du ikke finner mer. Metoden virker, men den er et søk, og søk kan gå glipp av ting.

Maurice Karnaugh arbeidet ved Bell Labs og publiserte i 1953 en metode som gjør det samme arbeidet uten leting. Ideen er å legge sannhetsverditabellen ut som et rutenett, ordnet slik at ruter som ligger inntil hverandre alltid hører sammen.

B.1 Rutenettet

Vi starter med to variabler. Tabellen har fire rader, og diagrammet får fire ruter.

	$\overline{B}$	B
$\overline{A}$	$\overline{A}\,\overline{B}$	$\overline{A}B$
A	$A\overline{B}$	AB

Figur B.1 Et Karnaugh-diagram for to variabler. Hver rute svarer til én rad i sannhetsverditabellen, og hver rute er merket med leddet som hører til den raden.

Hver rute er én rad i tabellen. Du fyller inn utgangsverdien, 0 eller 1, i hver rute.

Nøkkelen til at metoden virker, er plasseringen. **To ruter som ligger inntil hverandre skiller seg i nøyaktig én variabel.**

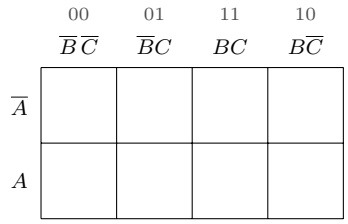
Se på de to øverste. Begge har $\overline{A}$; den ene har $\overline{B}$ og den andre har B. Det er nettopp formen $\overline{A}\,\overline{B} + \overline{A}B$, som vi i kapittel 5 forenklet til $\overline{A}$.

Det er hele metoden: **to naboruter med 1 i betyr at variabelen som skiller dem kan strykes.**

B.2 Tre variabler, og rekkefølgen som ser rar ut

Med tre variabler trenger vi åtte ruter. De settes opp som to rader og fire kolonner.

Her kommer det som forvirrer alle første gang. Kolonnene står ikke i tellerekkefølge. De står slik:



Figur B.2 Kolonnene teller 00, 01, 11, 10 — ikke 00, 01, 10, 11. De to siste er byttet om, med vilje.

Rekkefølgen $00, 01, 11, 10$ kalles **Gray-kode**, etter Frank Gray, en annen Bell Labs-ingeniør. Den har én egenskap: fra ett tall til det neste endrer nøyaktig én bit seg.

Prøv selv. $00 \to 01$: bare den høyre endret seg. $01 \to 11$: bare den venstre. $11 \to 10$: bare den høyre.

Hadde vi brukt vanlig telling, ville spranget $01 \to 10$ endret begge. Da ville naboruter ikke lenger hatt den egenskapen hele metoden hviler på.

Note-to-myself

Diagrammet er egentlig en sylinder. Den venstre og den høyre kolonnen er også naboer — $\overline{B}\,\overline{C}$ og $B\overline{C}$ skiller seg bare i B — selv om de ligger i hver sin ende av arket.
Det samme gjelder øverste og nederste rad når diagrammet har fire variabler. Papiret er flatt, men figuren er ikke.

B.3 Metoden

Tre steg.

1. Fyll inn utgangsverdien i hver rute.
2. Ring inn grupper av ettall. En gruppe må være rektangulær og inneholde 1, 2, 4 eller 8 ruter — aldri 3, 5 eller 6. Grupper får gjerne overlappe, og du skal lage dem så store som mulig. En rute som ikke har noen nabo med 1 i, blir stående som en gruppe på én.
3. Skriv ett ledd per gruppe. I leddet tar du bare med de variablene som er *like* i hele gruppen. De som skifter, stryker du.

Grunnen til at gruppene må ha 2, 4 eller 8 ruter er den samme som gjorde Gray-koden nødvendig: bare da forsvinner en hel variabel. En gruppe på to stryker én variabel, en gruppe på fire stryker to.

B.4 Eksempel: det vi gjorde for hånd

Vi tar uttrykket fra kapittel 5, tabell 5.5 på side 121:

$$F = \overline{A}\,\overline{B}\,\overline{C} + \overline{A}B\overline{C} + A\overline{B}\,\overline{C} + ABC$$

Fyll inn. Ettallene står i radene 000, 010, 100 og 111.

	$\overline{B}\,\overline{C}$	$\overline{B}C$	BC	$B\overline{C}$
$\overline{A}$	1	0	0	1
A	1	0	1	0

oransje: tre ruter som henger sammen over kanten blå: står alene

Figur B.3 Diagrammet for F. De tre oransje rutene er *ikke* én lovlig gruppe — tre er ikke en toerpotens — men de dekkes av to lovlige grupper på to.

Nå grupperer vi.

Gruppe 1: de to rutene i venstre kolonne. De har $\overline{B}\,\overline{C}$ felles, mens A skifter. Altså stryker vi A, og leddet blir $\overline{B}\,\overline{C}$.

Gruppe 2: ruten øverst til venstre og ruten øverst til høyre. De er naboer over kanten, siden diagrammet er en sylinder. Begge har $\overline{A}$ og $\overline{C}$; B skifter. Leddet blir $\overline{A}\,\overline{C}$.

Gruppe 3: den ene ruten i midten nederst. Den har ingen naboer med 1 i. En gruppe på én rute gir et ledd med alle tre variablene: ABC.

Resultatet:

$$F = \overline{B}\,\overline{C} + \overline{A}\,\overline{C} + ABC$$

Tre ledd og sju bokstaver, mot fire ledd og tolv bokstaver i utgangspunktet. Og du vet at det ikke går an å gjøre bedre på denne formen, fordi gruppene er så store de kan bli.

Tankevekker

Legg merke til at kapittel 5 kom fram til noe annet: $\overline{C \oplus AB}$, som bare er to porter.
Det er ingen motsetning. Karnaugh-diagrammet gir den minste *sum-av-produkter*-formen, altså den minste løsningen bygget av AND-porter inn i en OR-port. Den sier ingenting om hva som skjer hvis du også får bruke XOR og XNOR.
Metoden garanterer at du er i mål innenfor sine egne spilleregler. Den ser ikke XOR-mønstre, for de er spredt diagonalt i rutenettet, og diagonale ruter er ikke naboer.

B.5 Hvor langt rekker metoden

Fire variabler går fint: et rutenett på 4×4, med Gray-kode både bortover og nedover, og med naboskap over begge kantene.

Fem variabler krever to slike diagrammer ved siden av hverandre, der ruter med samme posisjon i de to er naboer. Det er fortsatt mulig, men man begynner å ta feil.

Seks og oppover gjør ingen for hånd. Og 64 variabler er helt utenfor rekkevidde — 2^{64} ruter er omtrent like mange som anslaget på antall sandkorn på alle strendene i verden.

Dette er grunnen til at ingen designer kretser slik lenger. Programvare gjør jobben, med algoritmer som arbeider på helt andre måter enn rutenett, og som klarer uttrykk med tusenvis av variabler.

Note-to-myself

Hvorfor da lære det?
Av samme grunn som du lærte å legge sammen i hodet selv om kalkulatoren er raskere: for å kjenne igjen formen på svaret. Når du ser et Karnaugh-diagram der ettallene ligger i en diagonal, vet du at det er en XOR, og det er en gjenkjennelse du tar med deg.
Og fordi det er fint. En metode som gjør algebra om til å se etter rektangler, er ikke hverdagskost.

Tillegg C

Kretser som husker

Dette er ikke pensum

Ingenting i dette appendikset kommer på eksamen.
Det står her fordi kapittel 4 ga leseren et løfte tre ganger: at vi skulle forklare hvordan et register kan holde på en verdi så lenge strømmen står på. Seksjon 5.10 på side 138 ga ideen på én side. Her er resten.
Det er den eneste delen av maskinvarestoffet der tid er en del av forklaringen, og det gjør det både vanskeligere og mer interessant enn alt annet i kapittel 5.

Alt i kapittel 5 var kombinatorisk. Utgangen fulgte inngangene, hele tiden, uten å huske noe.

Det er et problem, for en maskin må huske. Programtelleren må stå på det samme tallet mellom to klokkeslag. Akkumulatoren må holde på et mellomresultat mens ALU-en regner på noe annet. Minnet må holde på en milliard bit mens ingen ser på dem.

Spørsmålet i dette appendikset er hvordan noe som bare består av porter, kan gjøre det.

C.1 Tilbakekobling

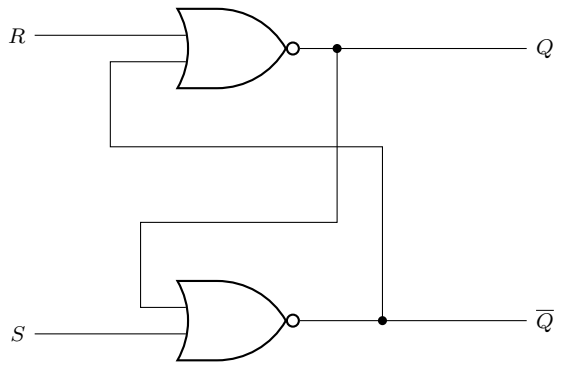
Svaret er én idé: **la utgangen være en av inngangene.**

Så lenge signalet bare går forlengs gjennom kretsen, er fortiden borte i samme øyeblikk inngangen endrer seg. Men går det en ledning fra utgangen og tilbake, kan kretsen holde seg selv oppe. Den som bestemmer hva utgangen skal være, er delvis utgangen selv.

Det høres sirkulært ut, og det er det. Poenget er at sirkelen kan være *stabil*: den kan hvile i en tilstand der alt stemmer med alt, og bli der til noen forstyrrer den.

C.2 SR-latchen

Den enkleste kretsen som gjør dette, er to NOR-porter koblet i kryss. Utgangen fra hver går inn som inngang på den andre.



Figur C.1 SR-latchen. To NOR-porter der hver utgang er inngang på den andre. S står for *set* og R for *reset*.

Kretsen har to utganger, Q og $\overline{Q}$, og de er normalt motsatte av hverandre. Verdien kretsen «husker», er Q.

La oss følge den gjennom.

C.2.1 Hvileposisjonen

Sett $S = 0$ og $R = 0$. Det er det normale: ingen prøver å endre noe.

Anta at Q er 1 akkurat nå. Da får den nedre NOR-porten inn $S = 0$ og $Q = 1$. En NOR gir 1 bare når begge inngangene er 0, så med en ener inne gir den 0. Altså er $\overline{Q} = 0$.

Den øvre NOR-porten får inn $R = 0$ og $\overline{Q} = 0$. Begge er 0, så den gir 1. Altså er $Q = 1$.

Det var det vi antok. Regnestykket går opp, og ingenting endrer seg. Kretsen hviler.

Gjør det samme med $Q = 0$ som antakelse, og du finner at også det går opp. **Med $S = R = 0$ har kretsen to stabile tilstander**, og den blir stående i den den er i.

Det er hukommelse. Ett bit, holdt oppe av to porter som bekrefter hverandre.

C.2.2 Å sette og å nullstille

Sett $S = 1$ et øyeblikk, mens R fortsatt er 0.

Den nedre NOR-porten får nå en ener inn og gir 0 ut, så $\overline{Q}$ blir 0. Da får den øvre porten inn $R = 0$ og $\overline{Q} = 0$, og gir 1. Altså blir $Q = 1$. Den enen går tilbake til den nedre porten, som fortsatt gir 0.

Slipp S tilbake til 0. Den nedre porten får nå $S = 0$ og $Q = 1$, og gir fortsatt 0. Ingenting rikker seg.

Endringen ble stående etter at inngangen gikk tilbake. Det er hele poenget.

$R = 1$ gjør det samme den andre veien, og setter Q til 0.

S	R	Hva skjer
0	0	Ingenting. Kretsen husker det den hadde.
1	0	Q blir 1, og blir stående der.
0	1	Q blir 0, og blir stående der.
1	1	Forbudt. Se under.

Tabell C.1 SR-latchen med NOR-porter. S for *set* setter Q til 1, R for *reset* setter den til 0, og null på begge betyr at kretsen holder på det den har.

C.2.3 Den forbudte kombinasjonen

Sett $S = 1$ og $R = 1$ samtidig. Begge NOR-portene får en ener inn, så begge gir 0. Q og $\overline{Q}$ er nå begge 0, og det er en selvmotsigelse — de skulle vært motsatte.

Det i seg selv er bare rart. Det farlige kommer når du slipper begge samtidig.

Da får begge portene to nuller inn, og begge begynner å gi 1. Men den ene enen rekker fram til den andre porten og slår den av. Hvilken av dem som vinner, avgjøres av hvilken port som er noen pikosekunder raskere.

Resultatet er *uforutsigbart*. Kretsen havner i en av de to tilstandene, og du kan ikke vite hvilken.

Advarsel :)

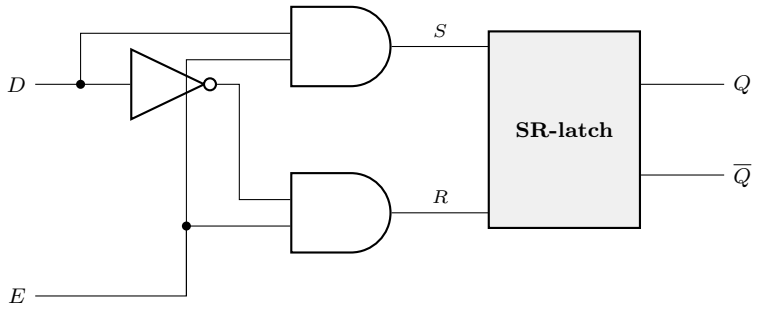
Dette er en av de få stedene i digital logikk der svaret ikke er bestemt av inngangene.
Det er grunnen til at ingen bruker en naken SR-latch i en ekte maskin. Alt som bygges videre herfra, har som hovedoppgave å gjøre det *umulig* å havne i den forbudte kombinasjonen.

C.3 D-latchen

Løsningen er å sørge for at S og R aldri kan være like.

Ta én datainngang, kall den D, send den rett inn på S, og send den gjennom en inverter inn på R. Da er de to alltid motsatte, og $S = R = 1$ kan ikke oppstå.

I tillegg vil vi kunne bestemme *når* latchen skal lytte. Vi legger inn en styreinngang, E for *enable*, og lar begge veiene gå gjennom hver sin AND-port med E.



Figur C.2 D-latchen. Inverteren gjør at S og R alltid er motsatte, så den forbudte kombinasjonen kan ikke oppstå. E bestemmer om latchen lytter.

Oppførselen er nå enkel å beskrive:

- Er $E = 1$, følger Q etter D. Endrer D seg, endrer Q seg med.
- Er $E = 0$, er begge AND-portene stengt. S og R er 0, latchen hviler, og Q står fast på den verdien den hadde da E gikk ned.

Dette er en brukbar lagringscelle. Du legger inn en verdi ved å sette D og la E være 1 et øyeblikk, og så holder den på verdien.

C.4 Fra latch til vippe

D-latchen har likevel et problem, og det handler om tid.

Så lenge E er 1, er latchen *gjennomsiktig*. Alt som skjer med D i det tidsrommet, slår rett gjennom til Q. Er E oppe i tjue nanosekunder og D vingler litt underveis, vingler Q også.

I en prosessor er det uakseptabelt. Der vil man at alle lagringsceller skal ta imot ny verdi i *nøyaktig samme øyeblikk*, og ellers stå helt stille.

Løsningen heter **vippe**, på engelsk *flip-flop*. En vippe tar imot ny verdi bare i det øyeblikket klokkesignalet skifter fra 0 til 1 — på *flanken*, ikke mens signalet er oppe.

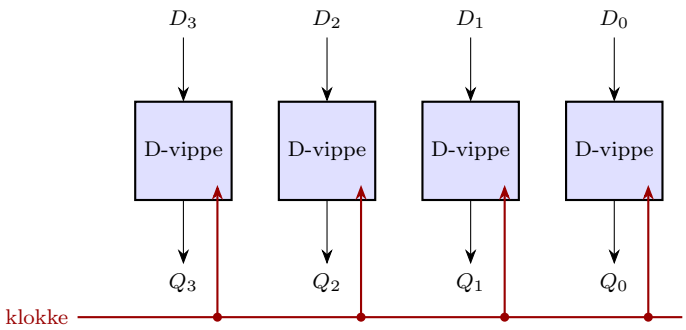
Note-to-myself

Forskjellen i ett bilde: en latch er en dør som står åpen så lenge du holder i håndtaket. En vippe er en dør som slipper gjennom akkurat det som står i åpningen i det sekundet den smeller igjen. Den vanligste måten å bygge en vippe på, er å sette to D-latcher etter hverandre, med motsatt E. Når klokka er lav, lytter den første og den andre holder. Når klokka går høy, stenger den første og den andre slipper gjennom det den første fanget. Det er aldri en åpen vei tvers gjennom begge.

Vippa er byggeklossen alt synkront utstyr er laget av. Og nå kan vi lukke sirkelen.

C.5 Registeret

Et register er én vippe per bit, med felles klokkeledning.



Figur C.3 Et register på fire bit. Fire vipper, én klokkeledning. Alle fire tar imot ny verdi i samme øyeblikk.

Det er alt et register er.

Kapittel 4 sa at registre er få, og at de finnes fordi de er raskere enn minnet. Nå kan du se hvorfor — og hvorfor de også er dyre.

Raske, fordi verdien allerede står i kretsen. Ingen skal slå opp noe, ingen adresse skal dekodes, ingen ledning skal ut av prosessoren. Utgangen fra vippa er tilgjengelig med én gang.

Dyre, fordi hver eneste bit koster to latcher. Hver latch er to NOR-porter, to AND-porter og en inverter, så en vippe er ti porter. Et 64-bits register er dermed godt over seks hundre porter som ikke gjør noen beregning i det hele tatt. De bare står der og holder.

Få, som følger av de to andre. Du legger ikke flere hundre porter på en bit du sjelden trenger.

Tankevekker

Og nå er den siste av de tre nedpekingene fra kapittel 4 besvart helt.
Et register husker fordi det består av kretser som bekrefter hverandre. To porter som peker på hverandre og sier det samme, blir stående og si det, til noen kommer og dytter dem over.
Det er ikke lagring i noen dypere forstand. Det er bare en sirkel som holder seg selv i gang, og som faller sammen i samme sekund strømmen går.

C.6 Og så kommer klokka

Vi har sagt «klokke» flere ganger uten å forklare den.

En vippe tar imot ny verdi på klokkeflanken. Det forutsetter at alle vipper i maskinen ser flanken omtrent samtidig, og at signalene som skal lagres, har rukket å sette seg før den kommer.

Begge forutsetningene er vanskelige å innfri i praksis, og de er grunnen til at klokkefrekvens er en grense i det hele tatt. Det er temaet i appendiks E.

Der blir også mentekjeden fra kapittel 5 interessant igjen. Kjeden må bli ferdig før neste klokkeflanke. Blir den ikke det, lagrer vippene et halvferdig svar, og ingen sier fra.

Tillegg D

Inni transistoren

Dette er ikke pensum

Ingenting i dette appendikset kommer på eksamen, og du trenger ikke en linje av det for å løse oppgavene i IN1020.
Det står her fordi kapittel 6 sa fra fem ganger at noe kom senere. Ohms lov kom aldri. Halvlederen kom aldri. Hvordan transistoren faktisk virker, kom aldri. Terskelspenningen ble brukt som forklaring uten å bli forklart selv. Og støymarginen fikk ideen, men ingen tall.
Her er alle fem. Til slutt gjør vi en ting kapitlet ikke nevnte med et ord: vi bruker den samme transistoren som forsterker.
Regn med en time. Du trenger ingen fysikk og ingen kjemi fra før.

Kapittel 6 lot ett spørsmål stå ubesvart hele veien.

Vi sa at transistoren er en bryter uten finger. Et signal på styreledningen avgjør om strømmen får gå. Men hvordan kan det ha seg? Ingenting beveger seg der inne. Ingen kontakt treffer en annen kontakt. Styreledningen rører ikke engang resten av transistoren — det ligger et isolerende lag imellom, og gjennom det går det ingen strøm.

Likevel bestemmer den.

Dette appendikset forklarer hvordan. Det tar en liten omvei først, for vi trenger tre ting på plass: hvordan strøm oppfører seg i en helt vanlig krets, hva silisium er, og hva som skjer når noen roter til silisiumet med vilje.

Alt er kvalitativt. Du får én formel som du har sett før uten å vite det, og noen regnestykker med små tall. Vannbildet fra kapittel 6 følger oss hele veien, og det holder lenger enn du skulle tro.

D.1 Spenning, strøm og motstand

Kapittel 6 klarte seg med to ord om elektrisitet. Spenning er høydeforskjellen i vannbildet. Strøm er hvor mye vann som faktisk renner.

Her trenger vi et tredje ord, og så trenger vi sammenhengen mellom de tre.

D.1.1 Det tredje ordet

Det tredje ordet er **motstand**, og i vannbildet er det hvor trangt røret er.

Tenk deg to rør fra samme tank. Det ene er en tynn slange, det andre er et avløpsrør. Høydeforskjellen er den samme. Likevel renner det mye mer gjennom det store røret, fordi vannet møter mindre motstand på veien.

Elektrisk motstand er det samme. Et tynt og langt stykke metall gir mye motstand. Et tykt og kort stykke gir lite. Og en transistor som står stengt, gir så enorm motstand at det nesten ikke renner noe.

Enhetene og navnene bak dem

De tre størrelsene har hver sin enhet, og alle tre er oppkalt etter personer som arbeidet med dette lenge før noen tenkte på datamaskiner.
Spenning måles i **volt** (V), etter Alessandro Volta. Strøm måles i **ampere** (A), etter André-Marie Ampère. Motstand måles i **ohm** (Ω), etter Georg Ohm.
Ett ampere er mye i denne sammenhengen. Tallene vi møter nede i en brikke er i million- og milliarddeler.

D.1.2 Sammenhengen

Georg Ohm var lærer i Tyskland og publiserte i 1827 det han hadde målt seg fram til. Sammenhengen er så enkel at den er lett å undervurdere:

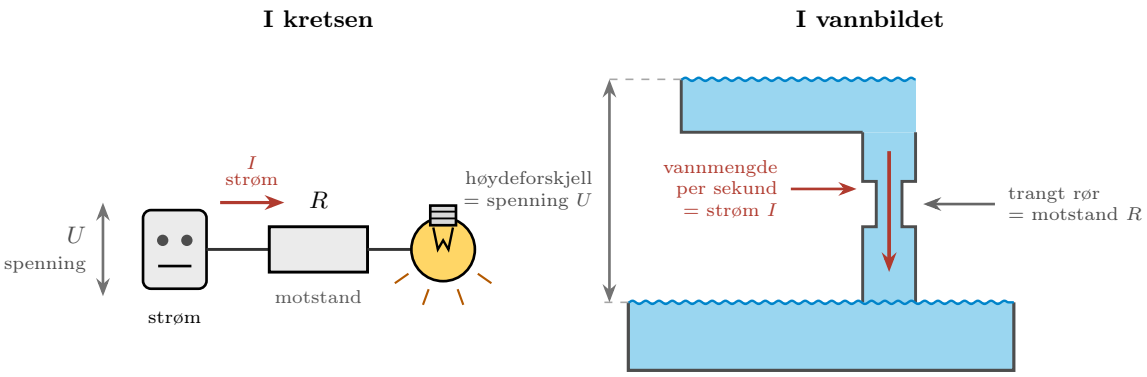
$$U = R \cdot I$$

Spenningen over noe er lik motstanden ganger strømmen gjennom det. Den kalles **Ohms lov**, og den er den mest brukte ligningen i all elektronikk.

Du får bruk for alle tre formene:

- $U = R \cdot I$ — hvor stor spenning må til?
- $I = U/R$ — hvor mye renner det?
- $R = U/I$ — hvor trangt er røret?

Les den i vannbildet, så gir den seg selv. Øker du høydeforskjellen, renner det mer. Gjør du røret trangere, renner det mindre.



Figur D.1 Ohms lov i to utgaver. Spenningen er høydeforskjellen, motstanden er hvor trangt røret er, og strømmen er hvor mye som faktisk renner. Øker du høyden, renner det mer. Gjør du røret trangere, renner det mindre.

D.1.3 To regnestykker

Det første er av den typen som står i en fysikkbok.

Tenk deg en liten lampe som får 3 volt, og som har en motstand på 15 ohm. Hvor mye strøm går gjennom den?

$$I = \frac{U}{R} = \frac{3}{15} = 0{,}2 \text{ A}$$

To tidels ampere. Det er en vanlig størrelse for en liten lyspære.

Det andre regnestykket er fra inni en brikke, og tallene ser helt annerledes ut.

En nMOS-transistor som leder, har en motstand på størrelsesorden 10 000 ohm. Over den står hele forsyningsspenningen på 0,8 volt. Da blir strømmen:

$$I = \frac{0{,}8}{10\,000} = 0{,}00008 \text{ A}$$

Altså 80 milliondeler av et ampere, eller 80 mikroampere. En enkelt transistor flytter forsvinnende lite.

Og den stengte transistoren

Prøv det samme på en transistor som står stengt. Motstanden er da rundt en milliard ohm.

$$I = \frac{0{,}8}{1\,000\,000\,000} = 0{,}0000000008 \text{ A}$$

Under et nanoampere. Det er så lite at det føles som null.
Gang det så med ti milliarder transistorer, som er det en stor brikke har. Da er vi oppe i flere ampere, og på 0,8 volt blir det watt som forsvinner uten at noen regner ut noe som helst.
Dette er et grovt overslag, og ekte brikker slår av strømmen til deler som ikke er i bruk. Men det er den regningen kapittel 6 kalte *lekkasje*, og nå vet du hvor den kommer fra: stengt gir høy motstand, og høy motstand gir liten strøm — ikke ingen.

D.1.4 Og effekten

Én ting til, siden kapittel 6 handlet så mye om varme.

Effekten er spenningen ganger strømmen:

$$P = U \cdot I$$

Lampa over trakk 3 volt ganger 0,2 ampere, altså 0,6 watt. Den varmer rommet like mye som en 0,6 watts varmeovn, og det er hele historien.

Formelen i kapittel 6 — kapasitans ganger spenning i andre ganger frekvens — sier det samme om en krets som skifter mange ganger i sekundet. Den regner ut gjennomsnittet over alle slurkene, i stedet for strømmen i ett øyeblikk.

D.2 Silisium, og hva vi gjør med det

Nå til stoffet transistoren er laget av.

D.2.1 Et stoff midt imellom

Kobber leder strøm. Glass leder ikke. De fleste stoffer er tydelig det ene eller det andre, og det gjør dem lite interessante.

Silisium ligger midt imellom. Rent silisium leder dårlig, men ikke så dårlig som glass. Og det som gjør det brukbart, er at vi kan styre hvor godt det leder.

Et stoff med den egenskapen har et eget navn: en **halvleder**.

Fun Fact

Silisium er det nest vanligste grunnstoffet i jordskorpa, etter oksygen. Det er hovedingrediensen i vanlig sand.
Å skaffe silisium koster nesten ingenting. Å rense det koster desto mer. Silisiumet en brikke hviler på, er blant de reneste tingene mennesker lager — i området ett fremmedatom per milliard.
Så tilsetter vi forurensninger igjen, med mikrometerpresisjon og i nøyaktig riktig mengde. Hele poenget med å rense så hardt er å kunne bestemme selv hva som skal være der.

D.2.2 Rent silisium er kjedelig

Et silisiumatom har fire elektroner å dele med naboene. I en silisiumkrystall sitter atomene i et fast mønster, og hvert atom deler ett elektron med hver av fire naboer.

Da går regnskapet opp. Alle plasser er fylt, og ingen elektroner er til overs.

Her er et bilde som hjelper. Tenk deg en rad med stoler der hver stol er opptatt. Ingen kan flytte seg, for det finnes ingen ledig plass. Raden er full, og alt står stille.

Strøm er elektroner som flytter seg. Når ingen kan flytte seg, går det ingen strøm.

D.2.3 Doping

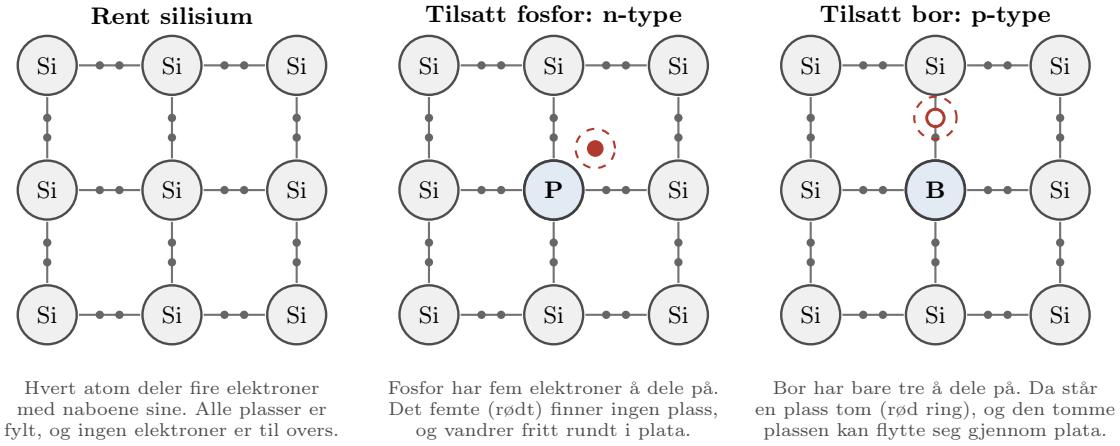
Nå kommer trikset, og det er overraskende grovt.

Vi blander inn noen få atomer av et annet grunnstoff. Det kalles **doping**, og mengden er liten: fra ett fremmedatom per hundre tusen til ett per ti millioner.

Det finnes to valg, og de gjør motsatt ting.

Fosfor har fem elektroner å dele med. Fire av dem finner en plass i mønsteret. Det femte finner ingen, og det blir gående løst. Silisium med fosfor i har altså noen frie elektroner, og elektroner er negativt ladet. Derfor kalles det **n-type**.

Bor har bare tre. Da blir det en plass i mønsteret som står tom. Den tomme plassen har et eget navn: et **hull**. Silisium med bor i kalles **p-type**.



Figur D.2 Silisium rent, og silisium med to slags tilsetning. Fosfor gir et elektron til overs, og plata leder med negativ ladning — det er n-type. Bor gir en tom plass, et hull, og plata leder ved at hullet flytter seg — det er p-type.

D.2.4 At et hull kan flytte seg

Hullet er den delen folk stopper opp ved, og det er verdt å bruke et avsnitt på.

Tilbake til stolraden. Nå er én stol ledig. Personen ved siden av flytter seg én plass bort, og da står den forrige stolen tom. Så flytter nestemann, og den ledige plassen har rykket enda et hakk.

Ingen av personene reiser langt. Likevel vandrer den ledige plassen hele veien gjennom raden.

Elektronene gjør nøyaktig det samme, og det er praktisk å snakke om hullet som om det var en ting som beveger seg. Hullet oppfører seg som en positiv ladning som vandrer motsatt vei av elektronene.

Plata er ikke ladet

Her er en misforståelse som er lett å få.
«n-type» høres ut som negativt ladet silisium, og «p-type» som positivt ladet silisium. Slik er det ikke. Fosforatomet tok med seg både det ekstra elektronet og den ekstra positive ladningen i kjernen sin. Regnskapet går opp.
Et stykke n-type silisium er elektrisk nøytralt, akkurat som et stykke rent silisium. Forskjellen er

bare at det har noen ladningsbærere som kan bevege seg fritt.
n og p sier altså hva slags ladning som *flytter seg*, ikke hvilken ladning plata har.

D.3 Felteffekten

Nå har vi delene. Her er mekanismen kapittel 6 utsatte.

D.3.1 To svingdører mot hverandre

Start med en transistor som står stengt.

En nMOS-transistor er en plate p-type silisium med to små områder n-type øverst. De to n-områdene er kontaktene, altså det strømmen eventuelt skal gå mellom. Mellom dem ligger p-silisium.

Et møte mellom n og p har en egenskap som er verdt å vite om: det slipper strøm bare den ene veien. En slik overgang virker som en svingdør som bare går én vei.

Og her står det to av dem, vendt mot hverandre. Prøver du å sende strøm fra venstre til høyre, stopper den andre døra deg. Prøver du andre veien, stopper den første.

Transistoren er altså stengt som utgangspunkt, og den er stengt begge veier.

D.3.2 Den tredje ledningen ligger oppå

Over gapet mellom de to kontaktene ligger styreelektroden. Den ligger ikke *på* silisiumet — det er et tynt isolerende lag imellom, laget av silisiumoksid, som er hovedingrediensen i vanlig glass.

Tre lag altså: metall øverst, oksid i midten, halvleder under. Der har du navnet kapittel 6 nevnte uten å forklare: MOS, for *metal-oxide-semiconductor*. Navnet er rett og slett en liste over hva transistoren består av, ovenfra og ned.

Det viktige er at oksidet isolerer. Styreelektroden henger over silisiumet uten å røre det, og det går ingen strøm fra den og ned.

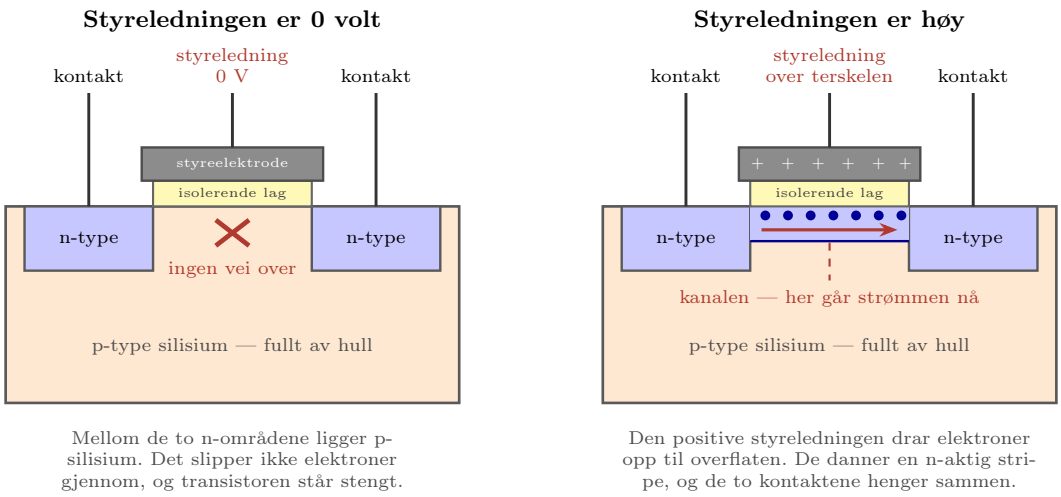
D.3.3 Så hva gjør den, da?

Den virker på avstand. Et elektrisk felt går rett gjennom det isolerende laget, selv om ingen ladning gjør det.

Sett en positiv spenning på styreelektroden. Den positive ladningen der oppe gjør to ting med silisiumet rett under: den dytter hullene vekk, og den drar elektroner til seg.

Elektronene samler seg i et tynt sjikt like under oksidet. Og når det har samlet seg nok av dem, har det tynne sjiktet blitt n-aktig — det ligner nå på de to kontaktene på hver side.

Da henger de to kontaktene sammen gjennom en stripe av fritt bevegelige elektroner. Stripa heter **kanalen**, og nå kan strømmen gå.



Figur D.3 Felteffekten. Styreelektroden ligger på et isolerende lag og rører ingenting. Den virker på avstand: er den positiv nok, trekker den elektroner opp under laget, og det blir en kanal mellom de to kontaktene. Slipper du styrespenningen, forsvinner kanalen igjen.

Slipp styrespenningen, og elektronene sprer seg igjen. Kanalen forsvinner, de to svingdørene er tilbake, og transistoren er stengt.

Dette er **felteffekten**: en spenning åpner og stenger en kanal uten at noe berører noe.

Du har sett det samme i dag

Telefonskjermen din virker på nøyaktig samme prinsipp.
Under glasset ligger et rutenett av ledere. Fingeren din rører aldri dem — glasset ligger imellom, og glass isolerer. Likevel merker skjermen hvor fingeren er, fordi fingeren endrer det elektriske feltet gjennom glasset.
Samme sak med en genser som drar i håret ditt på vei over hodet. Ladning på ett sted påvirker ladning på et annet, uten kontakt.
Felteffekten i en transistor er den samme fysikken, bare mye mindre og mye raskere.

D.3.4 Og pMOS er alt sammen speilvendt

pMOS-transistoren er den samme konstruksjonen med motsatt fortegn overalt.

Underlaget er n-type, de to kontaktene er p-type, og ladningsbærerne i kanalen er hull i stedet for elektroner. Styreelektroden må derfor være *negativ* i forhold til kontaktene for at kanalen skal dannes.

I en krets der det høyeste punktet er VDD, betyr det at en pMOS leder når styreledningen ligger lavt. Det er akkurat den oppførselen kapittel 6 beskrev, og boblen på symbolet er påminnelsen om den.

D.3.5 Hvorfor det er så fint at styreelektroden ikke rører noe

Her lukker vi en sirkel fra kapittel 6.

Siden oksidet isolerer, trekker styreelektroden ingen varig strøm. Du får en kort strøm mens du lader den opp eller ut, og så ingenting. Det er grunnen til at en CMOS-port som står i ro, nesten ikke bruker

strøm.

Men den korte ladestrømmen er også forklaringen på badekaret. En styreelektrode med et isolerende lag under, og silisium under der igjen, er akkurat det en kondensator er. Den må fylles med ladning før spenningen på den er høy.

Ingen ville ha det badekaret

Kapittel 6 slo fast at kapasitansen er en uunngåelig følge av at ting er fysiske, og ikke en komponent noen har satt inn. Nå kan du se hvorfor.
Oksidlaget må være der, ellers fungerer ikke felteffekten. To ledende flater med et isolerende lag imellom lagrer ladning, enten man vil eller ikke.
Lastkapasitansen er altså parasittisk. Den følger med på kjøpet, alle jobber for å få den mindre, og ingen har bruk for den.

D.4 Terskelspenningen

Nå kan vi svare på noe kapittel 6 påsto uten å begrunne: at forsyningsspenningen har et gulv den ikke kommer under.

D.4.1 Kanalen kommer ikke med én gang

Du kan ikke sette en bitteliten spenning på styreelektroden og få en bitteliten kanal.

Grunnen er at silisiumet rett under oksidet er p-type, og fullt av hull. Den første delen av styrespenningen går med til å dytte de hullene unna. Først når de er borte, begynner elektroner å samle seg.

Spenningen der kanalen virkelig begynner å danne seg, heter **terskelspenningen**. I en moderne prosess ligger den rundt 0,3 volt.

Bruk en lysdemper som bilde. Du vrir på knappen, og i den første biten av dreiningen skjer det ingenting i taket. Så tenner lampa, og derfra følger lysstyrken knappen. Terskelen er det punktet der noe begynner å skje.

D.4.2 Under terskelen er det ikke stille

Her er detaljen som gjør hele regnskapet vanskelig.

Under terskelen går strømmen ikke til null. Den faller bratt, men jevnt. Tommelfingerregelen er at strømmen blir omtrent ti ganger mindre for hver 0,1 volt du går under terskelen.

Det høres greit ut helt til du snur på det. Senker du terskelen med 0,1 volt, blir lekkasjen i *alle* de stengte transistorene ti ganger større.

Og de fleste transistorene i en brikke er stengt til enhver tid. Det er lekkasjen fra kapittel 6, og nå vet du hvorfor den ikke bare kan designes bort.

D.4.3 Hvorfor terskelen setter gulvet

Nå kan vi sette det sammen. To tall står mot hverandre: forsyningsspenningen og terskelspenningen.

En transistor leder godt først når styrespenningen ligger et godt stykke *over* terskelen. Avstanden mellom de to bestemmer hvor mye strøm transistoren klarer å levere, og dermed hvor fort den fyller badekaret.

Sett VDD til 0,8 volt og terskelen til 0,3. Da har en nMOS 0,5 volt å gå på, og det er nok.

Senk VDD til 0,4 volt, og det er bare 0,1 volt igjen. Strømmen stuper, badekaret fylles langsomt, og portforsinkelsen vokser kraftig. Brikken ble ikke bare mer sparsommelig — den ble treg.

Den nærliggende ideen er å senke terskelen tilsvarende. Og der står den andre veggen: terskelen ned betyr lekkasjen opp, ti ganger per 0,1 volt.

Tankevekker

To krav drar i hver sin retning, og de møtes et sted rundt 0,7 til 0,8 volt.
Over det punktet bruker brikken unødig mye strøm på hvert eneste skift, siden spenningen teller dobbelt i effektregnskapet.
Under det punktet taper du enten fart, hvis du holder terskelen oppe, eller strøm til lekkasje, hvis du senker den.
Dette er hele forklaringen bak setningen i seksjon 6.10 på side 163 om at spenningen traff gulvet. Gulvet er ikke en beslutning noen tok, men stedet der to kurver krysser hverandre.

D.4.4 Og oksidet som ikke kan bli tynnere

Det finnes en vei til. Et tynnere oksidlag gir sterkere felt for samme spenning, og dermed bedre kontroll over kanalen.

Den veien gikk bransjen helt til oksidet var noen få atomlag tykt. Da dukket det opp et nytt problem: elektroner begynte å smette rett gjennom laget, som om det ikke var der. Fenomenet heter tunnelering, og det er noe bare svært små ting kan gjøre.

Løsningen ble å bytte ut silisiumoksidet med andre stoffer som gir samme elektriske virkning i et fysisk tykkere lag. Men den slags triks er ikke gratis, og hvert av dem er en engangsgevinst.

Mens vi er ved små mål

Kapittel 6 advarte mot å lese nodenavn som lengder, og det er verdt å gjenta her, siden vi nå snakker om faktiske mål.
En «2 nm-prosess» har ingen del som er 2 nanometer lang. Den tilsvarende lengden i transistoren ligger snarere mellom 12 og 20 nanometer.
Nodenavnet er et generasjonsnavn. Blant de få delene av en moderne transistor som faktisk måles i enkeltsiffer nanometer, er isolasjonslaget under styreelektroden.

D.5 Tallene bak støymarginen

Seksjon 6.6 på side 154 sa at 0 og 1 er områder, og at det ligger et ingenmannsland imellom. Her er tallene.

D.5.1 Fire løfter

En port gir to løfter om hva den sender ut, og to løfter om hva den godtar å få inn. Til sammen fire, og alle fire har et navn.

Navn	Hva porten lover	Verdi
V_{OH}	Sender jeg en ener, er den minst så høy.	0,75 V
V_{OL}	Sender jeg en null, er den høyst så lav.	0,05 V
V_{IH}	Alt over dette leser jeg som en ener.	0,50 V
V_{IL}	Alt under dette leser jeg som en null.	0,30 V

Tabell D.1 De fire spenningsnivåene, med tall fra en brikke der forsyningsspenningen er 0,8 volt. O-en i navnet står for *output* og I-en for *input*, H for *high* og L for *low*.

Legg merke til at kravene til *inn* er slappere enn løftene om *ut*. Porten sender en ener på minst 0,75 volt, men den godtar alt over 0,50 volt som en ener.

Den forskjellen er hele poenget.

D.5.2 Marginen er differansen

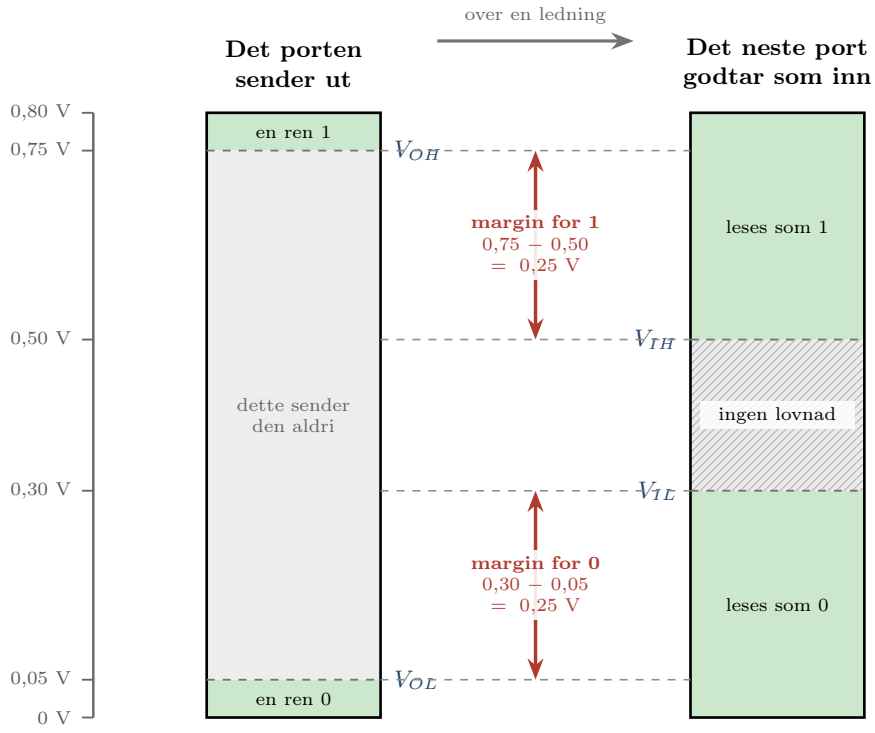
Støymarginen er avstanden mellom det porten lover å sende og det neste port krever å få.

Det gir to tall, ett i hver ende:

$$\text{margin for } 1 = V_{OH} - V_{IH} = 0{,}75 - 0{,}50 = 0{,}25 \text{ V}$$

$$\text{margin for } 0 = V_{IL} - V_{OL} = 0{,}30 - 0{,}05 = 0{,}25 \text{ V}$$

En ener kan altså miste et kvart volt på veien og fortsatt bli lest som en ener. En null kan plukke opp et kvart volt og fortsatt bli lest som en null.



Figur D.4 Støymarginen, med tall. Porten til venstre lover å sende en ener på minst 0,75 volt. Porten til høyre lover å lese alt over 0,50 volt som en ener. Avstanden mellom de to løftene er slingringsmonnet, og her er det 0,25 volt i begge ender.

Tenk på en samtale i en bråkete kantine. Du snakker høyere enn det som strengt tatt skal til for å bli hørt. Differansen er slingringsmonnet, og det er det som gjør at samtalen tåler at noen mister en tallerken i gulvet.

Hvor mye er et kvart volt?

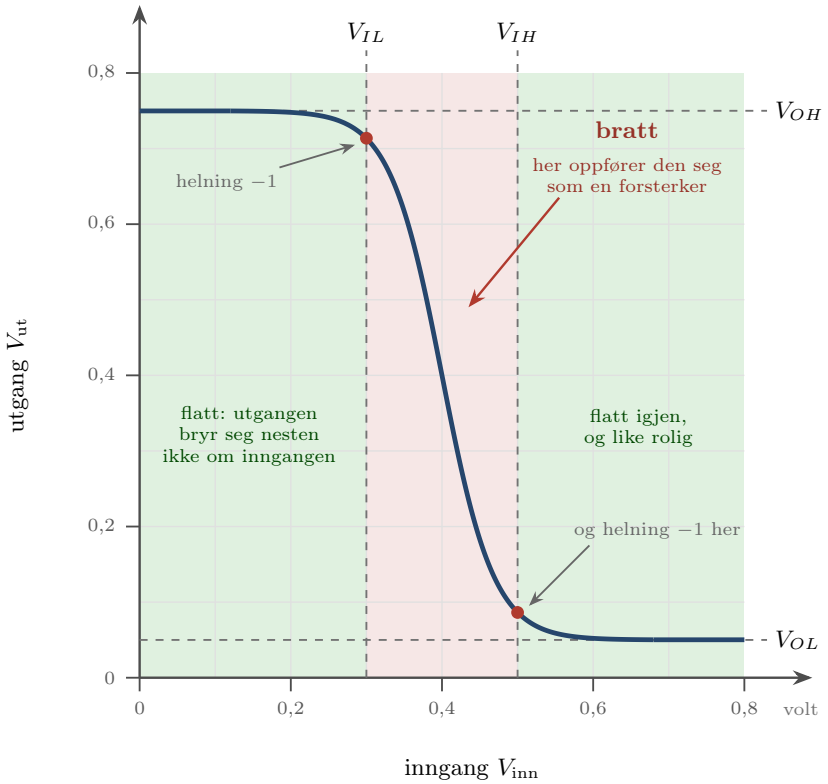
Det høres lite ut. Det er en firedel av et volt.
Men det skal sammenlignes med støyen på en ledning inne i en brikke, og den er typisk noen titalls millivolt. Da er et kvart volt romslig.
Støyen kommer fra at naboledninger skifter samtidig og drar litt i hverandre, fra at forsyningsspenningen dipper når mange porter arbeider i samme øyeblikk, og fra varme. Ingen av delene lar seg fjerne. Derfor bygger man inn slingringsmonn i stedet.

D.5.3 Hvor grensene kommer fra

Nå er et spørsmål verdt å stille: hvem bestemte at V_{IH} skulle være nettopp 0,50 volt?

Ingen bestemte det. Det følger av hvordan kretsen oppfører seg, og det kan måles.

Mål en inverter. Sett en spenning inn, les av spenningen ut, og gjenta for hele området fra 0 til VDD. Kurven du får, heter **overføringskarakteristikken**.



Figur D.5 Overføringskarakteristikken til en inverter. Den viser hva utgangen gjør for hver eneste inngangsspenning. Ytterst er kurven flat, og det er der logikken bor. I midten stuper den, og i det smale beltet oppfører transistoren seg som en forsterker.

Kurven har tre deler. Ytterst til venstre er den flat og høy: inngangen kan vingle en del uten at utgangen bryr seg. Ytterst til høyre er den flat og lav, like rolig. I midten stuper den.

Grensene ligger der helningen er nøyaktig -1. Den grensa er ikke satt på slump, og grunnen er verdt å tenke gjennom.

Så lenge helningen er slakkere enn -1, blir en forstyrrelse på inngangen *mindre* på utgangen. Porten krymper problemet. Blir helningen brattere enn -1, blir forstyrrelsen større på utgangen enn den var på inngangen, og da vokser den for hvert ledd i kjeden.

Punktene der helningen passerer -1 er altså vendepunktene mellom «porten rydder opp» og «porten gjør det verre». Det er nøyaktig det V_{IL} og V_{IH} betyr.

Tankevekker

Her er grunnen til at signalgjenopprettingen i seksjon 6.6 på side 154 faktisk virker.
Et signal som ligger trygt innenfor et av de flate områdene, kommer ut skarpere enn det kom inn. Neste port gjør det samme igjen. Rusk blir ikke bare stoppet — det blir mindre for hvert ledd. En kjede på tusen porter er dermed ikke tusen anledninger til å samle opp feil. Den er tusen anledninger til å vaske signalet rent.

D.5.4 Hva som skjer når forsyningen synker

De fire nivåene flytter seg med forsyningsspenningen. Halverer du VDD, blir også marginene omtrent halvert.

Gå fra 0,8 til 0,4 volt, og de 0,25 voltene blir til noe i nærheten av 0,12. Problemet er at støyen ikke følger med ned. Naboledninger drar like hardt som før, og varmen er den samme.

Dette er den tredje grunnen til at forsyningsspenningen har et gulv, ved siden av fart og lekkasje. Under et visst punkt er det rett og slett ikke nok slingringsmonn igjen.

D.6 Transistoren som forsterker

Nå skal vi bruke den samme kurven til noe helt annet.

D.6.1 Det bratte stykket

Se på midten av overføringskarakteristikken en gang til, og glem logikken et øyeblikk.

Der er utgangen ikke 0 og ikke 1. Den ligger et sted imellom, og den følger inngangen. Flytter inngangen seg litt, flytter utgangen seg mye.

Si at inngangen endrer seg med 0,02 volt. På det bratteste punktet svarer utgangen med en endring på rundt 0,10 volt. Utgangen beveger seg altså fem ganger så mye som inngangen.

Det har et navn. Forholdet mellom endringen ut og endringen inn er **forsterkningen**, og her er den fem.

Kranbildet fra kapittel 6 passer like godt her. En liten dreining på håndtaket endrer vannmengden kraftig. Den store strømmen kommer fra vannledningen, ikke fra fingeren din, og fingeren bestemmer bare hvor mye som slipper gjennom.

D.6.2 Det er dette radio og lyd er

En mikrofon lager en spenning som følger lyden, og den spenningen er bitteliten. En høyttaler trenger langt mer for å flytte luft.

Mellom dem står en forsterker, og i bunnen av den står transistorer som arbeider midt på kurven sin. Utgangen følger inngangen i form, men i en helt annen størrelse.

Det samme gjelder radiosignaler, sensorer, bilder fra et kamerachip og alt annet som kommer inn i en maskin fra verden utenfor. Ytterst i kjeden står alltid noe analogt.

Den var en forsterker først

Transistoren ble demonstrert i 1947, ved Bell Labs. Jobben den skulle gjøre, var å forsterke — den var tenkt som en erstatning for radiorøret i telefonnettet og i radioapparater. Digitale kretser var ikke det man hadde i tankene.
CMOS, altså de to transistortypene brukt sammen slik kapittel 6 viste, ble oppfunnet og patentsøkt i 1963.
Og Gordon Moores artikkel kom i 1965, med spådommen han justerte til en dobling hvert annet år i 1975.

Fra 1947 til 1963 gikk det altså seksten år før noen satte sammen de to typene på den måten hele den digitale verdenen seinere skulle bygge på.

D.6.3 Hvorfor digital design holder seg unna midten

Hvis midtområdet er så nyttig, hvorfor unngår digital logikk det med vilje?

Tre grunner, og alle tre er alvorlige.

Det finnes ingen margin der. Et signal midt i beltet er like nær 0 som 1. Den minste forstyrrelse flytter det over, og neste port kan lese det stikk motsatt av det forrige port gjorde.

Forstyrrelsen vokser. Helningen i midten er brattere enn -1. En forsterker gjør ikke forskjell på signal og støy, og i midtområdet blir rusket forsterket sammen med alt annet.

Og det blir varmt. Midt i overgangen leder begge transistorene samtidig. Da står det en rak vei fra VDD til GND gjennom begge to, og strømmen som renner der, gjør ingen nytte.

Det siste er lettest å se i badekarbildet. Krana og avløpet er åpne på én gang. Vannet renner rett gjennom karet og ut i sluket, uten å heve vannstanden med en millimeter.

D.6.4 Men innom er den hver eneste gang

Her kommer det som gjør seksjonen verdt å lese.

En port kan ikke hoppe fra 0 til 1. Den må gjennom alle verdiene imellom, og det betyr at den er innom forsterkerområdet ved hvert eneste skift.

Tiden den bruker der, er flanketiden — hvor bratt signalet stiger eller faller. Og nettopp i det tidsrommet leder begge transistorene, så det renner en rak strøm fra forsyningen til jord.

Derfor bryr kretsdesignere seg om bratte flanker. Et signal som stiger langsomt, er ikke bare sent ute. Det blir hengende i midtområdet, og hvert nanosekund der koster strøm som ikke utretter noe.

Regn på det

Kapittel 6 sa at effekten går opp med frekvensen, fordi du fyller og tømmer badekaret flere ganger i sekundet.
Nå har du en grunn til: hvert skift tar med seg en liten slurk gjennom begge transistorene i tillegg til det som havner i badekaret.
Begge postene vokser med antall skift per sekund. Begge står stille når signalet ligger i ro. Det er den samme logikken en gang til, nå fra en annen kant.

D.6.5 Og det som står stille når ingenting skjer

Et siste punkt, for å unngå en vanlig misforståelse.

Når et signal har kommet i mål, holder porten det der aktivt. Én av de to transistorene leder og kobler utgangen til VDD eller GND, og den forbindelsen står til inngangen endrer seg. Utgangen renner ikke ut, og den trenger ingen oppfriskning. Det er slik *statisk* CMOS virker, og det er den typen logikk alt i kapittel 5 og 6 er bygget av.

Det finnes andre familier der verdien faktisk lekker ut. Dynamisk logikk legger en verdi på en ledning og

lar den ligge, og den må brukes før den er borte. Det samme gjelder DRAM, som du møter i kapittel 7: hvert bit ligger som ladning i en bitteliten kondensator, og kretsen må lese og skrive det tilbake mange ganger i sekundet for at det skal overleve.

Blander man de to, får man den forklaringen kapittel 6 advarte mot: at klokka finnes for å redde verdiene før de forsvinner. For statisk CMOS stemmer det ikke.

D.7 Hva du kan ta med deg opp igjen

Tre ting er verdt å holde på.

Transistoren er en kran som styres av et felt. Spenningen på styreelektroden trekker ladning til overflaten i silisiumet og åpner en kanal. Ingenting berører noe, og derfor trekker styringen nesten ingen strøm.

Terskelen setter gulvet. Transistoren trenger en viss spenning for å åpne ordentlig, lekkasjen vokser dramatisk hvis man senker den terskelen, og støymarginen krymper i takt med forsyningen. Tre grunner som peker samme vei, og til sammen er de varmeveggen fra 2004.

Den samme komponenten kan gjøre begge deler. Midt på kurven er transistoren en forsterker, og ytterst er den en bryter. Digital teknologi har valgt å bo i ytterkantene, og hele påliteligheten vi bygger alt annet på, kommer av det valget.

D.8 Videre lesning

Harris & Harris, *Digital Design and Computer Architecture*, avsnitt 1.6 og 1.7. Samme stoff som dette appendikset, med flere tall og med figurer som ligner. Avsnitt 1.7.1 tar transistoren som bryter, og 1.6 tar de fire spenningsnivåene og støymarginen. Dette er det naturlige neste steget.

Neamen, *Semiconductor Physics and Devices*, kapittel 1 til 4. Her står det vi hoppet over: hvorfor silisium leder som det gjør, hva båndteori er, og hvordan pn-overgangen virker i detalj. Det krever litt fysikk, men de første kapitlene er skrevet for folk som ikke har den.

Sedra & Smith, *Microelectronic Circuits*. Standardverket om transistoren brukt analogt. Hvis forsterkerseksjonen var den mest interessante for deg, er det denne veien du skal.

Ben Eater, «Transistors», på nett. Han måler på ekte transistorer på et koblingsbrett, inkludert midtområdet. Å se utgangen bevege seg gradvis er noe annet enn å lese om det.

Videre i denne boka: appendiks A forteller hvordan vi kom fra radiorør til silisium. Appendiks E tar klokkefordelingen over en hel brikke, som er det andre stedet fysikken setter grenser for hva maskinvaren kan gjøre.

D.9 Oppgaver

Disse er ikke pensum, og det finnes ingen fasit. Flere av dem har mer enn ett rimelig svar.

1. En ledning har en motstand på 4 ohm. Over den står 1,2 volt. Hvor mye strøm går gjennom?
2. En transistor leder 50 mikroampere når det står 0,8 volt over den. Hvor stor er motstanden?
3. Forklar med vannbildet hvorfor en tynn og lang ledning gir mer motstand enn en tykk og kort.

4. Hvorfor er et stykke n-type silisium elektrisk nøytralt, når det har frie elektroner i seg?
5. Beskriv med dine egne ord hvordan et hull kan flytte seg, når et hull er en plass der det ikke er noe.
6. Hvorfor er det nødvendig at laget under styreelektroden isolerer? Hva ville skjedd hvis styreelektroden lå rett på silisiumet?
7. En nMOS har terskelspenning 0,3 volt. Forsyningsspenningen er 0,8. Hva skjer med farten hvis noen senker forsyningen til 0,5 volt, og hvorfor?
8. Noen foreslår å senke terskelspenningen til 0,1 volt slik at brikken kan gå på 0,4 volt. Hva er innvendingen, og hvor stor er den omtrent?
9. En port har $V_{OH} = 0{,}70$ V, $V_{OL} = 0{,}10$ V, $V_{IH} = 0{,}45$ V og $V_{IL} = 0{,}25$ V. Regn ut begge støymarginene. Er de like store?
10. Forklar hvorfor grensene V_{IL} og V_{IH} ligger nettopp der helningen på kurven er -1, og ikke et annet sted.
11. Hvorfor koster et signal som stiger langsomt, mer strøm enn et som stiger raskt — selv om begge ender på samme verdi?
12. En medstudent sier: «Transistoren er enten på eller av, så den kan ikke brukes til lyd.» Hva er galt i påstanden, og hva er riktig?

Tillegg E

Synkronisering

Dette er ikke pensum

Ingenting i dette appendikset kommer på eksamen, og du trenger ikke en linje av det for å løse en oppgave i IN1020.
Kapittel 6 ga deg klokka slik den ser ut fra utsiden: en jevn puls, en periode, og en frist alle kretsene må rekke. Den forklaringen holder. Den hopper bare over ett spørsmål.
Hvordan kommer det ene pulsslaget seg ut til alle som skal ha det?
Svaret er et lite ingeniørfag for seg. Det er her mye av strømmen i en brikke tar veien, og det er her de siste prosentene av klokkefrekvensen vinnes og tapes.
Regn med en halvtime.

Seksjon 6.9 på side 160 tegnet klokka som én ledning med et jevnt mønster på. Alle vippene henger på den ledningen, og alle ser flanken i samme øyeblikk.

Den siste setningen er en nyttig løgn.

Et pulsslag bruker tid på å komme fram, akkurat som alt annet i kapittel 6. Ledningen er lang. Den har et badekar å fylle. Og den som sitter lengst unna, hører slaget litt etter den som sitter nærmest.

Dette appendikset handler om den forskjellen, og om regningen den sender.

E.1 Ett slag, mange millioner mottakere

Klokkesignalet skal fram til hver eneste vippe på brikken. Det er mange av dem. En moderne prosessor har flere millioner, spredt over et areal på et par kvadratcentimeter.

Tenk deg at du skal rope et startsignal til en hel fotballstadion. Stemmen din har bare så mye kraft, og publikum lengst unna hører ingenting.

En port har nøyaktig samme problem. Hver vippe den skal nå, har en styreledning som må lades opp, og hver styreledning er et lite badekar. Ett tusen vipper er ett tusen kar. Én port kan ikke fylle dem alle.

E.1.1 Treet

Løsningen er den samme som på stadion: sett opp høyttalere, og la hver høyttaler ta sin del av publikum.

I en brikke heter høyttaleren en **forsterker**. Den tar imot klokkesignalet, og den sender det videre med full kraft til neste trinn. Underveis deler signalet seg i to, og to, og to igjen. Resultatet er et **klokketre**.

Forsterkeren er noe du allerede har sett

En forsterker i et klokketre er to invertere etter hverandre.
Den første snur signalet, den andre snur det tilbake. Ut kommer det samme signalet som gikk inn — men nå drevet av to friske transistorer, med full kraft og bratte flanker.
Det er den samme oppvaskingen som seksjon 6.6 på side 154 beskrev. Hvert trinn vasker signalet rent, og her bruker vi den egenskapen med vilje.

Et klokketre på en stor brikke har gjerne flere tusen slike forsterkere. De står i lag, og hvert lag driver det neste.

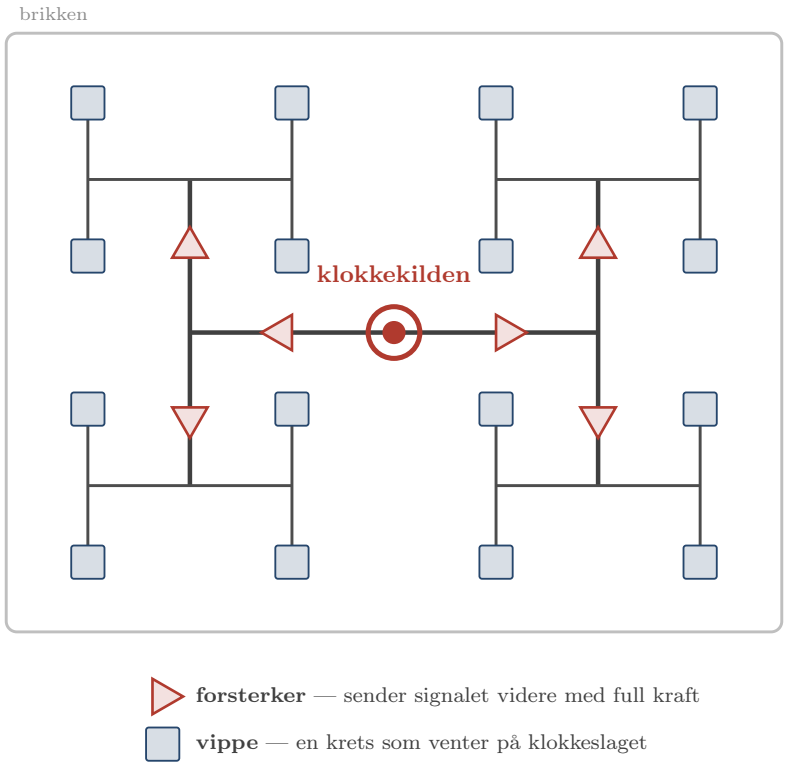
E.1.2 Hvorfor treet ser ut som en H

Nå kommer det som gjør klokketreet annerledes enn all annen ledning på brikken.

Alle veiene gjennom treet skal være like lange.

For andre signaler spiller det liten rolle om en ledning er litt lengre enn naboen. For klokka er det hele poenget. Kommer slaget fram til to vipper på ulik tid, begynner de to å arbeide i ulik takt, og da forsvinner avtalen klokka er til for.

Den vanligste løsningen er et **H-tre**. Formen gjentar seg: en strek deler seg i to armer, hver arm deler seg i to nye, og slik fortsetter det nedover. Figur E.1 på neste side viser to runder av mønsteret.



Figur E.1 Et H-tre. Klokkekilden står i midten, og signalet deler seg i to for hvert trinn utover. Veien fra midten ut til hvert eneste blad er like lang, og derfor kommer slaget omtrent samtidig fram overalt.

Tell etterpå: fra midten ut til et hvilket som helst blad går signalet like langt. Det er hele grunnen til at figuren ser så streng ut.

Ekte klokketrær er rotete. Brikken er ikke symmetrisk, blokkene har ulik størrelse, og noen hjørner trenger flere vipper enn andre. Verktøyene bygger derfor treet automatisk, og de bruker timer på å få det i balanse.

E.2 Skew: slaget som kommer litt for sent

Uansett hvor pent treet er bygget, kommer klokka aldri helt samtidig fram.

Tenk deg et kor på tjue personer i en stor kirke. Dirigenten slår an, og alle skal synge første tone i samme sekund. De bakerst ser hånda like tidlig som de fremste, men hører klangen litt senere, og noen av dem reagerer litt tregere enn andre. Resultatet er en tone som smører seg ut over et lite øyeblikk.

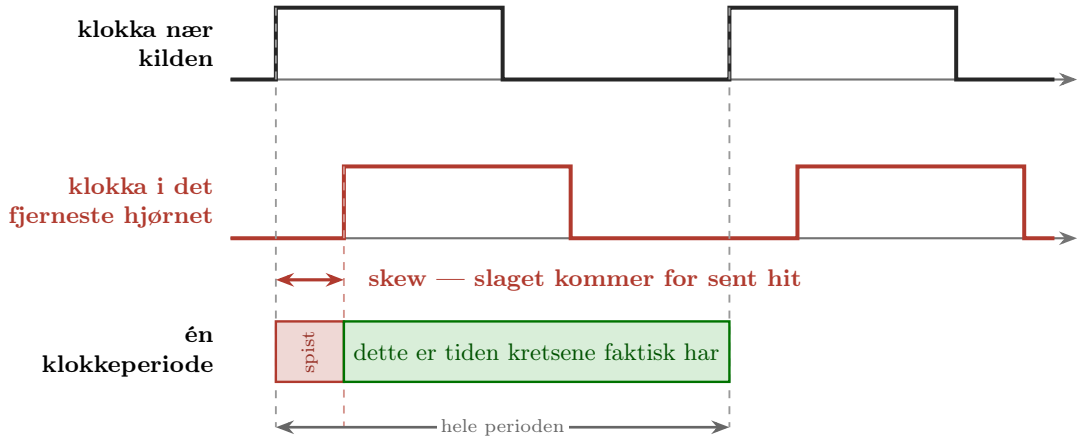
Forskjellen i ankomsttid mellom to punkter på brikken heter **clock skew**, eller bare forskyvning på norsk.

Fire ting lager skew. Ledningene er ikke helt like lange. Noen forsterkere driver flere vipper enn andre. Brikken er varmere i midten enn i kanten, og varme transistorer er tregere. Og to transistorer som skulle vært like, kommer aldri helt like ut av fabrikken.

E.2.1 Hva det koster

Skew spiser av klokkeperioden.

Si at klokka kommer 10 pikosekunder for sent fram til mottakeren. Da har kretsene mellom sender og mottaker 10 pikosekunder mindre på seg enn perioden sier. Figur E.2 viser det.



Klokka taper ingenting på forskyvningen. Regningen taper, for den må bli ferdig i det som er igjen.

Figur E.2 Clock skew. Den samme flanken, sett to steder på brikken. Forskyvningen i figuren er kraftig overdrevet — i praksis sikter designerne mot noen få pikosekunder.

Regn på det med tallene fra kapittel 6. En brikke på 3 GHz har en periode på 333 pikosekunder. Ti pikosekunder skew er tre prosent av den. Tre prosent av klokkefrekvensen er mye penger når hele bransjen slåss om de siste prosentene.

Derfor bruker designerne så mye arbeid på treet. Hvert pikosekund de får bort, kan de legge i regnetid i stedet.

Den skumle varianten

Skew kan også ødelegge på en måte som ikke lar seg fikse med en langsommere klokke.
Tenk deg en stafett. Den som skal ta imot pinnen, starter for tidlig. Da er veksleren ikke framme ennå, og laget mister pinnen helt.
Det samme kan skje i en krets. Kommer klokka *for tidlig* fram til den mottakende vippa, kan den nye verdien fra avsenderen rekke fram før vippa har fått låst den gamle. Da hopper verdien over et helt klokkeslag.
Senker du frekvensen, hjelper det ikke, for feilen handler om de to flankene i forhold til hverandre, ikke om hvor lang perioden er. En slik brikke er ødelagt, og den må rettes i tegningen og lages på nytt. Det er den dyreste feilen i hele faget.

Noen ganger bruker designerne skew med vilje. Har ett trinn i en pipeline for lite tid og nabotrinnet litt til overs, kan de forsinke klokka til den ene vippa en smule. Da låner det trange trinnet tid av det romslige. Trikset heter *useful skew*, og det er et av de få stedene der en feilkilde har blitt et verktøy.

E.3 Jitter: slaget som ikke kommer helt i takt

Skew er et avvik fra sted til sted. **Jitter** er et avvik fra gang til gang.

Prøv å klappe en helt jevn takt i tjue sekunder. Du klarer det nesten. Noen av klappene kommer et lite øyeblikk for tidlig, andre for sent, og ingen av dem treffer helt. En klokkekrets har samme svakhet, bare i en annen skala.

Klokka starter utenfor brikken, i en liten kvartskrystall som svinger med en fast takt, typisk rundt 25 megahertz. Inne på brikken ganger en krets som heter *faselåst sløyfe* det tallet opp til gigahertz. Oppgangeren er en analog krets, og den er følsom.

Tre ting rister på den. Forsyningsspenningen dupper når mange kretser skifter samtidig. Temperaturen svinger. Og nabosignaler drar litt i klokkeledningen gjennom ledningene rundt.

Jitter koster på samme måte som skew: designerne må trekke det verste tilfellet fra perioden før de gir resten til regningen. En brikke som er oppgitt til 333 pikosekunder, kan i praksis bare regne med vel 300 av dem, når skew, jitter og margin for varme er trukket fra.

E.4 Hva klokka koster i strøm

Nå kommer tallet som overrasker folk.

Klokkenettet alene bruker en stor del av strømmen i en brikke. Målinger på ulike design har landet mellom 20 og 45 prosent, og rundt en tredjedel er et vanlig svar.

Grunnen følger rett av badekarbildet i seksjon 6.7 på side 156. Effekten avhenger av hvor ofte noe skifter, og klokka skifter hele tiden. En vanlig dataledning ligger stille det meste av tida; den skifter kanskje hver tiende periode. Klokkeledningen skifter to ganger hver eneste periode, over hele brikken, hvert sekund maskinen står på.

Og den ledningen er lang. Et klokketre når ut i hvert hjørne, og hele den lengden er badekar som må fylles og tømmes milliarder av ganger i sekundet.

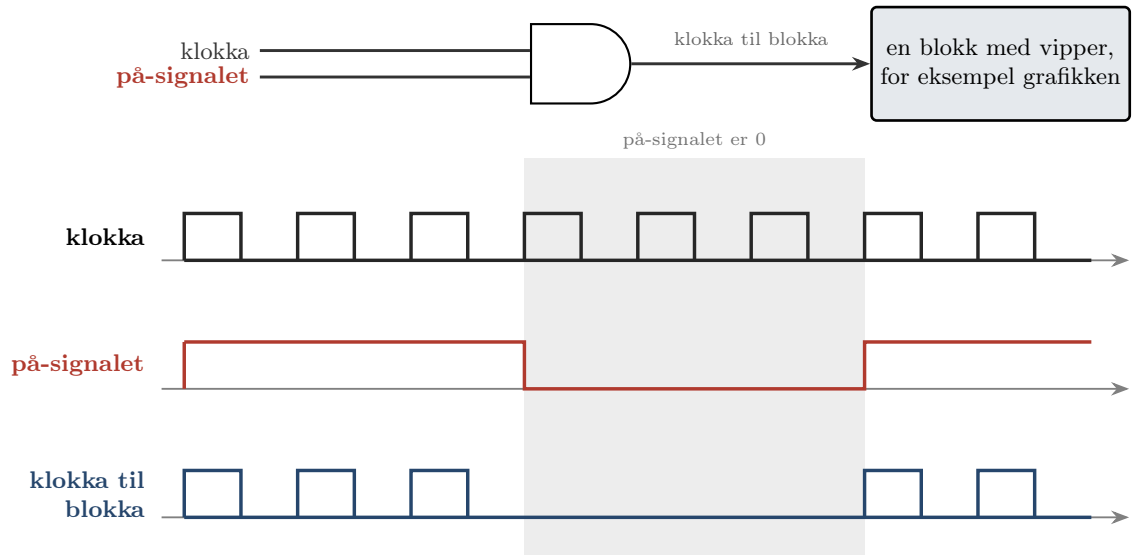
Lyset i et stort kontorbygg

Tenk deg et bygg med fire hundre kontorer. Taklyset i hver gang og hvert rom står på hele døgnet, hele året, uansett om noen er der.
Strømregningen for belysningen blir fort større enn for alt arbeidet som faktisk gjøres i bygget. Det var ikke meningen. Det er bare det som skjer når noe står på overalt, hele tiden, i tilfelle noen trenger det.

E.5 Clock gating: å slå av lyset i rom ingen bruker

Svaret på den regningen er like enkelt som i bygget. Slå av klokka til de delene som ikke gjør noe.

Teknikken heter **clock gating**. Prinsippet er en AND-port på klokkeledningen. Den ene inngangen er klokka, den andre er et på-signal fra styringslogikken. Er på-signalet 0, står utgangen stille, og vippene bak den får ingen flanker å reagere på.



I det grå feltet står vippene helt stille. Ingen kar fylles, ingen kar tømmes, og strømmen går nesten i null.

Figur E.3 Clock gating. En AND-port på klokkeledningen lar styringslogikken stenge klokka til en blokk som ikke er i bruk.

Gevinsten er stor fordi den slår to veier. Vippene slutter å skifte, og grenen av klokketreet som går dit, slutter også å skifte. Den grenen var ofte det dyreste av de to.

En moderne brikke gjør dette i stor skala. Verktøyene setter inn tusenvis av slike porter, helt ned til enkeltregistre som bare brukes av én instruksjonstype. Spiller du musikk på telefonen med skjermen av, står størstedelen av brikken med klokka stengt.

Figuren er en forenkling

En naken AND-port duger ikke i praksis, og grunnen er verdt et øyeblikk.
Skifter på-signalet midt i en klokkepuls, kutter porten pulsen på midten. Ut kommer en stump som er for kort til å virke som en flanke, men lang nok til å forvirre vippene bak.
Ekte gating-celler har derfor en liten latch som holder på-signalet fast mens klokka er høy. Da kan på-signalet bare endre seg i det trygge vinduet, og pulsene som slipper gjennom, er alltid hele.
Latchen er den samme kretsen som appendiks C bygde.

E.5.1 Og det gating ikke fikser

En blokk med stengt klokke står stille, men den har fortsatt strøm på.

Det betyr at lekkasjen fra seksjon 6.10 på side 163 løper videre. Krana drypper selv om ingen bruker vannet.

Vil du bli kvitt den også, må du kutte forsyningen til hele blokka. Det heter *power gating*, og det virker — men prisen er høy. Blokka glemmer alt den sto med, nøyaktig som appendiks C forklarte: et register er en sirkel som faller sammen i samme sekund strømmen går. Og det tar tid å våkne igjen, ofte mange mikrosekunder.

Derfor er valget et skjønn. Korte pauser tar clock gating. Lange pauser tar power gating.

E.6 Klokkedomener

En moderne brikke har flere klokker, slik notatboksen i seksjon 6.9 på side 160 nevnte. Her er detaljene.

Et **klokkedomene** er et område som deler én klokke og én frekvens. De typiske domenene i en PC-prosessor er prosessorkjernene, det som ligger rundt dem og betjener dem, minnegrensesnittet, grafikken og inn- og utgangskretsene.

En påstand som ikke stemmer

Det blir av og til sagt at hver funksjonsenhet i en prosessor har sin egen klokkefrekvens — at ALU-en går i én takt, dekoderen i en annen og registerfila i en tredje.
Det gjør de ikke. Domenene er grove. En stor brikke har en håndfull til et par dusin av dem, ikke tusenvis.
Det som finnes per enhet, er clock gating. Hver enhet kan få klokka stengt når den ikke er i bruk, men når den går, går den i takt med resten av domenet sitt.
Grunnen er at et domene koster. Hvert domene trenger sitt eget tre, sin egen oppganger og sin egen margin. Og hver grense mellom to domener trenger ekstra kretser, som du ser i neste avsnitt.

E.6.1 Å krysse en grense

Et signal som går fra ett domene til et annet, møter en klokke det ikke har noe forhold til. Flanken kan komme når som helst, også midt i en overgang.

Da kan en vippe havne i en tilstand som verken er 0 eller 1.

Tenk på en mynt du kaster på bordet. Nesten alltid lander den på en av sidene. Nå og da lander den på kanten, står og vaker, og bruker et øyeblikk på å bestemme seg. Hvor lenge den står, kan ingen forutsi.

Fenomenet heter **metastabilitet**, og det er ikke en teoretisk kuriositet. Ser kretsen bak på vippa mens den fortsatt vaker, leser den søppel.

Kuren er enkel og litt frekk: sett to vipper etter hverandre. Den første får lov til å vake. Den andre leser den først et helt klokkeslag senere, og da har den første nesten alltid bestemt seg. Du kan ikke gjøre sannsynligheten for feil null, men du kan gjøre den så liten at den forventede tiden mellom to feil er lengre enn brikkens levetid.

Prisen er tid. Hver kryssing koster minst et klokkeslag, ofte to eller tre. Det er en av grunnene til at designerne holder antall domener nede, og til at store datamengder krysser gjennom en kø i stedet for signal for signal.

E.7 Å skru på klokka mens maskinen går

Klokkefrekvensen er ikke spikret fast. Maskinen din endrer den flere ganger i sekundet.

Teknikken heter **dynamisk frekvens- og spenningsskalering**, forkortet DVFS. Operativsystemet og en liten styringskrets på brikken ser på hvor travelt det er, og justerer deretter.

Hvorfor begge deler på én gang? Svaret ligger i formelen i seksjon 6.10 på side 163. Effekten vokser med frekvensen, og den vokser med spenningen i andre potens. Senker du bare frekvensen, sparer du lineært. Men en lavere frekvens gir lengre periode, og med lengre periode rekker kretsene fram på lavere spenning også. Da kan du ta begge, og sparingen blir mye større.

Et eksempel med runde tall. En kjerne går i 3 GHz på 1,1 volt. Styringskretsen setter den ned til 1,5 GHz og 0,8 volt. Frekvensen er halvert, og spenningen ganger effekten med $0{,}8^2/1{,}1^2 \approx 0{,}53$. Til sammen bruker kjernen rundt en firedel så mye.

Den andre veien gjør brikken det samme. Er bare én kjerne travel, kan styringskretsen gi den mer spenning og høyere frekvens en stund, fordi de andre kjernene ligger stille og varmebudsjettet er felles. Det er dette Intel kaller Turbo Boost og AMD kaller Precision Boost. Blir brikken for varm, trekker den seg ned igjen av seg selv.

Lønner det seg alltid å gå sakte?

Nei, og grunnen er lekkasjen.
En brikke som står på, lekker uansett hva den holder på med. Går den sakte, bruker den lite effekt — men den bruker den lenger, og lekkasjeregningen løper hele tiden.
For mange oppgaver lønner det seg derfor å kjøre fort, bli ferdig, og legge seg til å sove. Strategien har fått navnet *race to idle*, og den er grunnen til at en telefon ofte sprinter i hundre millisekunder og så sovner igjen.
Hvilken strategi som vinner, avhenger av hvor mye brikken lekker og hvor dypt den klarer å sove. Begge svar finnes i ekte maskiner.

En ting DVFS ikke gjør, er å følge enkeltinstruksjoner. Å endre spenning tar noen mikrosekunder, og det er tusenvis av klokkeperioder. Styringen jobber i grovere tidsrom enn programmet gjør.

Mobiltelefoner tar ideen ett skritt lenger. Der sitter det gjerne to eller tre grupper kjerner på samme brikke: noen få store og raske, og noen flere små og sparsomme. Hver gruppe har sin egen klokke og sin egen spenning. Leser du e-post, går jobben på de små. Starter du et spill, våkner de store.

E.8 Å klare seg uten en felles klokke

Hele dette appendikset har handlet om hva det koster å holde en hel brikke i takt. Da er et spørsmål nærliggende: må vi?

Nei. Det finnes kretser uten klokke i det hele tatt.

E.8.1 Håndtrykket

Tenk deg at Aisha rekker en full kaffekopp til Birk. Ingen av dem teller til tre. Aisha holder koppen til hun kjenner at Birk har tak, og først da slipper hun. Birk sier ifra med grepet sitt, ikke med et ord.

Asynkron logikk gjør akkurat dette. En krets som er ferdig, sier «her kommer svaret». Mottakeren svarer «jeg har det». Først da begynner neste runde. To ekstra ledninger erstatter klokka, og de kalles *request* og *acknowledge*.

Fordelene er reelle. Det er ikke noe klokketre å bygge, og dermed ingen skew. Strømmen følger arbeidet helt presist: en krets som ingen spør, bruker ingenting, uten at noen må huske å stenge klokka til den.

Og farten følger den *faktiske* forsinkelsen. En addisjon med kort mente blir ferdig raskere enn en med lang, og mottakeren får svaret med en gang. En synkron krets går alltid i takt med sitt verste tilfelle, en asynkron med dagens tilfelle.

E.8.2 Hvorfor nesten ingen gjør det likevel

Ulempene er også reelle, og de er praktiske mer enn prinsipielle.

Håndtrykket koster kretser. Hver kobling trenger ekstra ledninger og ekstra logikk, og regnestykket spiser fort opp det du sparte på klokketreet.

Verktøyene er bygget for synkrone design. Femti år med programvare for syntese, tidsanalyse og testing forutsetter at det finnes en klokke å måle mot. Testingen blir vond av samme grunn: en synkron brikke kan du stoppe mellom to klokkeslag og se på, mens en asynkron brikke ikke har slike øyeblikk.

Et forskningsmiljø i Manchester bygde på 1990-tallet en serie asynkrone ARM-prosessorer som het Amulet. De virket. De var konkurransedyktige på strøm. De ble likevel aldri en kommersiell bølge, og ulempene over er grunnen.

E.8.3 GALS: kompromisset som vant

Løsningen de fleste har landet på, ligger i midten.

Del brikken i øyer. Inne på hver øy går alt i takt med én klokke, og alle de kjente verktøyene virker. Mellom øyene bruker du håndtrykk. Forkortelsen er **GALS**, for *globally asynchronous, locally synchronous*.

Tenk på en stor organisasjon. Hver avdeling har sitt eget morgenmøte til sin egen tid. Skal avdelingene snakke sammen, legger de en beskjed i en postkasse, og den andre henter den når det passer. Ingen prøver å få tusen personer til å møtes samtidig.

Og her lukkes sirkelen fra forrige seksjon. En moderne brikke med fem-seks klokkedomener og synkronisatorer på grensene *er* et GALS-system, selv om ingen kaller det det. Kompromisset vant så grundig at det ble usynlig.

E.9 To ekte brikker

Til slutt, kort, to eksempler som viser hvordan dette ser ut i praksis.

Intel Nehalem, 2008. Nehalem delte brikken i kjernene og det Intel kalte *uncore*: den delte L3-cachen, minnekontrolleren og koblingene ut. De to delene fikk hver sin klokke. En egen styringskrets på brikken holdt øye med strøm og temperatur, og den kunne gi en travel kjerne høyere frekvens mens de andre lå lavt. Nehalem kunne i tillegg kutte forsyningen helt til en kjerne som ikke var i bruk. Brikken er et ryddig eksempel på grove domener og fin gating i samme design.

AMD Zen, 2017. Zen bygger prosessoren av flere små blokker, og mellom dem går en forbindelse som heter Infinity Fabric. Fabricen har sin egen klokke, og den har lenge fulgt hastigheten på minnet i stedet for kjernene. Kjernene har sin. Overgangen mellom de to går gjennom synkronisatorer, og når de to frekvensene ikke passer sammen, koster hver kryssing ekstra klokkeslag. Det er en av grunnene til at raskere minne kan gjøre en Zen-maskin merkbart kjappere — ikke bare fordi minnet er raskere, men fordi grensa blir billigere å krysse.

E.10 Videre lesning

Harris & Harris, *Digital Design and Computer Architecture*, kapittel 3. Avsnittene om timing tar oppsett- og holdetid, skew og metastabilitet med tall og ligninger. Den regner også ut hvor

lenge en synkronisator må vente for at feilen skal bli sjelden nok. Dette er den naturlige neste boka.

Sparsø & Furber, *Principles of Asynchronous Circuit Design.* Standardverket om kretser uten klokke, skrevet av to av dem som sto bak Amulet-prosessorene. Fritt tilgjengelig på nett. Les kapittel 1 hvis du bare vil ha smaken.

Weste & Harris, *CMOS VLSI Design.* Tyngre, men den har de grundige kapitlene om klokkefordeling og om strømbudsjettet i en brikke, med de målingene som tallene i dette appendikset bygger på.

Vil du grave i forskningslitteraturen, er søkeordene *clock distribution network*, *clock gating* og *globally asynchronous locally synchronous*.

E.11 Oppgaver

Oppgavene under er ikke pensum, og det finnes ingen fasit til dem bakerst i boka. De som handler om klokkeperiode, flanker og portforsinkelse, finner du i selv-testen til kapittel 6 i stedet.

1. Hvorfor kan ikke én enkelt port drive klokkesignalet ut til alle vippene på en brikke? Svar med badekarbildet.
2. Hvilken egenskap ved H-treet er den viktige, og hvorfor spiller den egenskapen mindre rolle for vanlige dataledninger?
3. Nevn fire årsaker til clock skew. Hvilke av dem kan en designer gjøre noe med, og hvilke må en bare leve med?
4. En brikke går i 2,5 GHz. Skew og jitter spiser til sammen 18 pikosekunder. Hvor stor andel av perioden er det?
5. Forklar forskjellen mellom skew og jitter med dine egne ord, og gi en analogi til hver.
6. Hvorfor hjelper det ikke å senke klokkefrekvensen når skew får en verdi til å hoppe over et klokkeslag?
7. Klokkenettet bruker en stor del av strømmen i en brikke. Forklar hvorfor, med utgangspunkt i hvor ofte en ledning skifter.
8. Hvorfor duger ikke en naken AND-port som gating-celle i praksis?
9. En blokk har stengt klokke, men full forsyning. Hva bruker den fortsatt strøm på, og hva må til for å bli kvitt det?
10. En kollega påstår at hver funksjonsenhet i en prosessor kan ha sin egen klokkefrekvens. Hva er galt i påstanden, og hvilken teknikk tenker kollegaen sannsynligvis på?
11. Forklar metastabilitet med en analogi, og forklar hvorfor to vipper etter hverandre gjør problemet lite nok.
12. DVFS senker frekvens og spenning samtidig. Hvorfor begge to, når det er frekvensen som bestemmer farten?
13. Når lønner det seg å kjøre fort og sovne, framfor å kjøre sakte hele tiden? Hva i brikken avgjør svaret?
14. Nevn to fordeler og to ulemper ved asynkron design. Hvilken av ulempene tror du veier tyngst i dag, og hvorfor?
15. Forklar GALS for noen som akkurat har lært hva en klokke er. Bruk en analogi som ikke er postkassa.

Tillegg F

Hvordan cachen finner linja

Dette er ikke pensum

Ingenting i dette appendikset kommer på eksamen, og du trenger det ikke for å løse en eneste oppgave i IN1020.
Det står her fordi avsnitt 7.4 på side 183 sa at cachen «finner» eller «ikke finner» en linje, uten å si hvordan. Det er en reell forenkling, og den dekker over noe som er kort og ganske elegant.
Regn med tjue minutter.

Kapittel 7 stilte et spørsmål det ikke besvarte.

En L1-cache på 32 kilobyte rommer 512 linjer på 64 byte. Prosessoren spør etter en adresse, og cachen må svare på fire klokkesykluser om den har den eller ikke.

Fire sykluser er ikke mye. Hvordan rekker den å lete gjennom 512 plasser?

Svaret er at den ikke leter. Den regner ut hvor linja ville ligget hvis den fantes, og ser etter bare der.

F.1 Adressen er allerede delt opp

Trikset er at en minneadresse inneholder svaret, hvis du deler den riktig.

Start med noe du kan gjøre i hodet. En cachelinje er 64 byte. Da hører adressene 0 til 63 til den første linja, 64 til 127 til den andre, 128 til 191 til den tredje, og så videre.

Hvilken linje hører adresse 1088 til? Del på 64:

$$1088 \div 64 = 17, \text{ uten rest}$$

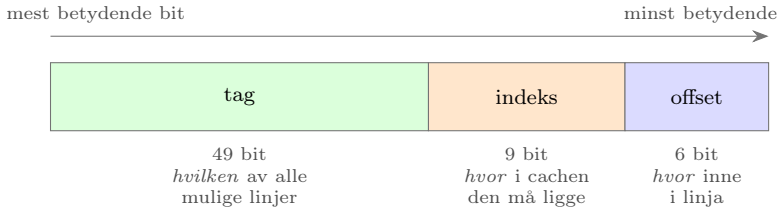
Adresse 1088 er altså det første byte i linje nummer 17. Og adresse 1090?

$$1090 \div 64 = 17, \text{ rest } 2$$

Samme linje, men to byte inn i den.

Dette er hele ideen. Å dele en adresse på 64 og se på kvotienten og resten gir deg linjenummeret og posisjonen inne i linja.

Og nå kommer det fine. I totallsystemet er det å dele på 64 det samme som å flytte seg seks sifre — akkurat som å dele på 100 i titallsystemet bare flytter kommaet to plasser. Maskinen trenger ikke dele i det hele tatt. Den bare ser på forskjellige deler av bitmønsteret.



Figur F.1 En 64-bits adresse delt i tre, for en cache på 32 KB med 64-byte linjer og 512 plasser.

Her er hvor de tre tallene kommer fra:

Offset: 6 bit. Linja er 64 byte, og $2^6 = 64$. Seks bit er nøyaktig det som trengs for å peke på ett av 64 byte inne i linja.

Indeks: 9 bit. Cachen har 512 plasser, og $2^9 = 512$. Ni bit peker på én av dem.

Tag: resten. På en maskin med 64-bits adresser blir det $64 - 9 - 6 = 49$ bit.

F.2 Direkte avbildning

Den enkleste cachen gjør nå dette: indeksen bestemmer plassen, helt og holdent. Linje 17 kan bare ligge på plass 17. Ikke noe annet sted.

Det kalles en **direkte avbildet** cache, og oppslaget blir svært raskt:

1. Les indeksbitene. Det peker ut én plass i cachen.
2. Se på taggen som er lagret på den plassen.
3. Er den lik taggen i adressen? Da er det treff. Ellers er det bom.

Ingen leting. Ett oppslag og én sammenligning, og begge kan gjøres samtidig i maskinvaren.

Men hvorfor trengs taggen? Fordi plass 17 ikke bare kan huse linje 17. Den huser også linje 529, og linje 1041, og alle andre linjenumre som gir 17 til overs når du deler på 512.

Cachen har 512 plasser og minnet har milliarder av linjer. Mange linjer må dele på hver plass. Taggen er det som forteller hvilken av dem som faktisk ligger der akkurat nå.

Plass	Gyldig	Tag	64 byte data
16	ja	0x4A	···
17	ja	0x00	···
18	nei	—	···

adresse 1088 gir
indeks 17, tag 0x00
⇒ treff

Figur F.2 Oppslag i en direkte avbildet cache. Indeksen peker på plass 17, og taggen der stemmer. Gyldigbitet sier om plassen i det hele tatt er tatt i bruk.

Legg merke til kolonnen som heter «gyldig». Når maskinen slås på, er cachen full av tilfeldig søppel, og taggene kan tilfeldigvis stemme. Gyldigbitet er ett bit per plass som sier om det står noe ekte der. Uten det ville cachen svart med søppel i stedet for å si at den ikke har verdien.

F.3 Problemet med direkte avbildning

Direkte avbildning er rask og billig. Den har også en ubehagelig svakhet.

Si at programmet ditt veksler mellom to tabeller, og at de tilfeldigvis ligger slik i minnet at de treffer samme plass i cachen. Adresse 1088 og adresse 33 856, for eksempel — de ligger 32 kilobyte fra hverandre, og begge gir indeks 17.

Da skjer følgende: programmet leser fra den første tabellen, og linja legges på plass 17. Så leser det fra den andre, og den linja må også ligge på plass 17. Den første kastes ut. Så leser programmet fra den første igjen, og den andre kastes ut. Og slik fortsetter det.

Resultatet er at cachen bommer hver eneste gang, selv om den har massevis av ledig plass. Alle de 511 andre plassene står tomme.

Dette kalles en **konfliktbom**, og det er en reell plage. Den er særlig ubehagelig fordi den er usynlig: to programmer som gjør akkurat det samme kan ha helt ulik fart, bare fordi dataene tilfeldigvis havnet på ulike adresser.

F.4 Assosiativitet: gi linja flere steder å bo

Løsningen er å slakke på regelen om at hver linje bare har ett sted å ligge.

I en **toveis assosiativ** cache deles plassene i par. En linje med indeks 17 kan ligge på hvilken som helst av de to plassene i par 17. Da kan begge tabellene i eksemplet over ligge inne samtidig, og konflikten forsvinner.

Merk at adressedelingen endrer seg. Med 512 plasser i par blir det 256 par, så indeksen krymper fra ni til åtte bit, og taggen vokser fra 49 til 50. Regelen fra forrige seksjon holder fortsatt: indeksen må være akkurat stor nok til å peke på ett sted å lete.

Prisen er at oppslaget blir dyrere. Nå må cachen sammenligne taggen mot to plasser i stedet for én. Det gjøres samtidig, i parallell maskinvare, men det koster transistorer og litt tid.

Jo flere veier, jo færre konflikter — og jo dyrere oppslag.

Type	Steder per linje	Vurdering
Direkte avbildet	1	Raskest og billigst. Sårbar for konflikter.
Toveis assosiativ	2	Fjerner mange av konfliktene. Vanlig i små og eldre cacher.
Åtte til seksten veier	8–16	Det vanlige i dag, på alle nivåer. Konflikter er sjeldne.
Fullassosiativ	alle	Ingen konflikter i det hele tatt. For dyr for alt annet enn svært små cacher.

Tabell F.1 Avveiningen i assosiativitet. Den samme avveiningen som resten av kapittel 7: fart mot fleksibilitet.

En fullassosiativ cache lar en linje ligge hvor som helst. Da er det ingen indeks i adressen i det hele tatt — hele adressen minus offset blir tag — og cachen må sammenligne mot alle plassene samtidig. Det er raskt nok hvis det er åtte plasser, og helt uaktuelt hvis det er 512.

F.5 Hvem må vike

Når en ny linje skal inn og alle de aktuelle plassene er opptatt, må noe ut.

I en direkte avbildet cache er det ikke noe valg: det er bare ett sted, og den som ligger der, må gå.

Med assosiativitet oppstår et valg, og det trengs en regel. Den vanligste er **minst nylig brukt** — på engelsk *least recently used*, forkortet LRU. Cachen kaster ut den linja det er lengst siden noen rørte.

Begrunnelsen er tidslokaliteten fra avsnitt 7.3 på side 180. Har en linje ikke vært brukt på lenge, er det den som sannsynligvis heller ikke blir brukt snart.

Å holde helt nøyaktig orden på rekkefølgen koster maskinvare, så ekte cacher bruker som regel en tilnærming som er nesten like god og mye billigere.

F.6 Hva du kan ta med deg

Tre ting er verdt å huske, selv om ingenting av dette er pensum.

Cachen leter ikke — den regner. Det er derfor den rekker å svare på fire sykluser. Indeksen i adressen peker rett på stedet, og så er det bare én sammenligning igjen.

Taggen er nødvendig fordi mange linjer deler plass. Cachen har noen hundre plasser og minnet har milliarder av linjer.

Og en cache kan bomme på data den faktisk har plass til. Konfliktbommen er grunnen til at assosiativitet finnes, og den er grunnen til at to programmer som gjør det samme kan ha ulik fart.

Vil du gå videre herfra, er Patterson & Hennessy kapittel 5 det naturlige neste steget. Der står det samme med flere regneeksempler, og der finner du også klassifiseringen av bommer i tre typer og formelen for effektiv aksesstid skrevet ut ordentlig.

Tillegg G

Hvorfor telefonen har en egen KI-brikke

Dette er ikke pensum

Ingenting i dette appendikset kommer på eksamen, og du trenger ikke en linje av det for å løse oppgavene i IN1020.
Det står her fordi kapittel 6 stoppet midt i en setning. Det sa at klokka stoppet på rundt 3 GHz, at svaret ble flere kjerner, og at man dessuten «bygger spesialiserte kretser som gjør én oppgave svært effektivt». Så gikk det videre. Her er den kretsen.
Du trenger ikke kunne noe om kunstig intelligens fra før. Du får vite hvorfor maskinvaren under den ser ut som den gjør.
Regn med to timer på lesingen. Oppgavene til slutt kommer i tillegg, og fire av dem krever penn og papir. Dette er det lengste appendikset i boka.

Slå på flymodus. Pek kameraet mot en side med tekst, og telefonen lar deg markere ordene og kopiere dem. Begynn å skrive en melding, og den gjetter neste ord før du har tenkt det.

Ingenting av dette går via nettet. Det skjer i hånda di, på noen millisekunder, og batteriprosenten rører seg knapt.

Det burde ikke stemme. Kapittel 6 på side 143 fortalte en ganske nedslående historie: frekvensen stoppet å stige rundt 2004, spenningen kan ikke senkes mer, og varmen setter grensen. Telefonen din har ingen vifte og et batteri på størrelse med et kakestykke. Den burde ikke kunne gjøre noe en bærbar maskin fra 2015 ikke kunne.

Likevel gjør den det. Og grunnen er at det sitter en egen krets inne i brikken som ikke kan gjøre stort annet enn én ting.

Dette appendikset svarer på fire spørsmål, i denne rekkefølgen. Hva kretsen regner ut, og hvorfor regnestykket er så enkelt at det nesten er flaut. Hvorfor en vanlig prosessor er elendig til akkurat det, selv om den kan alt annet. Hva de ulike operasjonene faktisk koster i energi — som er den delen alt det andre følger av. Og hva som skjer hvis man tar ideen helt ut og slutter å flytte tallene.

Underveis bygger vi mye på ting du allerede har. Portene som den vektede summen bygges av, kommer fra kapittel 5 på side 105. Regnestykket med energi og frekvens kommer fra kapittel 6 på side 143. At tall kan lagres med færre siffer enn man tror, kommer fra kapittel 2 på side 13. Og i den siste seksjonen får Ohms lov fra appendiks D endelig noe å gjøre.

To ord du trenger med én gang

En **modell** er i denne sammenhengen ikke et program, men en stor samling tall. Det er disse tallene — gjerne millioner eller milliarder av dem — som utgjør «det maskinen har lært». Å bruke en modell er å regne med tallene. Å lage dem er en helt annen jobb, og den foregår et annet sted. En **akselerator** er en krets som er bygget for én bestemt type arbeid i stedet for for alt mulig. Lydkort og grafikkort er eldre eksempler. Kretsen dette appendikset handler om, er en akselerator for nevrale nett.

Hva dette appendikset ikke er

Dette er ikke en innføring i kunstig intelligens eller maskinlæring. Du får ikke vite hvordan en modell lages, hva en transformer er, eller hvorfor en språkmodell svarer som den gjør. Alt det er store og spennende fag, og de har egne emner.
Her står bare maskinvaren. Spørsmålet gjennom hele appendikset er: *hvilke kretser må til, og hvorfor akkurat de?*

G.1 Et regnestykke, ikke en hjerne

Ordet «nevralt nettverk» lover mer enn det holder. Det høres ut som en hjerne, og det er det ikke. Det er en lang rekke ganginger og addisjoner, satt opp på en bestemt måte.

Vi tar oss god tid til å komme fram til akkurat hvor enkelt regnestykket er, fordi det er det som forklarer alt resten. Og vi starter et sted du har vært før.

G.1.1 Porten kan skrives med vekter

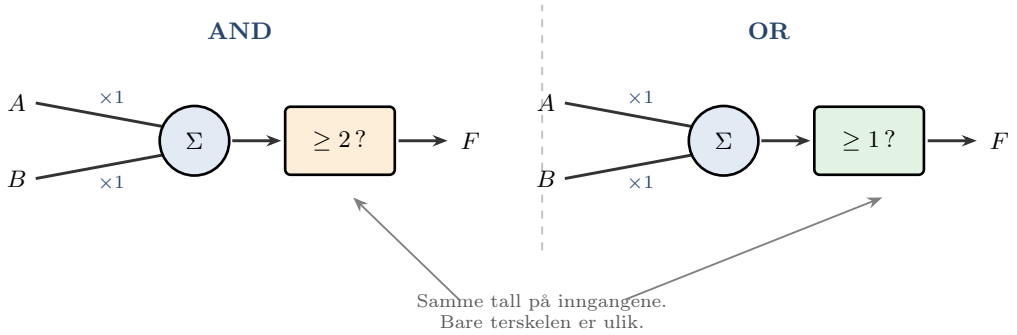
Hent fram AND-porten fra kapittel 5 på side 105. Den sender ut 1 bare når begge inngangene er 1.

Det kan skrives på en annen måte. Gi hver inngang tallet 1. Legg sammen inngangene. Send ut 1 hvis summen er minst 2.

Prøv den: med inngangene 1 og 0 blir summen 1, som ikke er minst 2, så utgangen er 0. Med 1 og 1 blir summen 2, og utgangen er 1. Det stemmer med sannhetstabellen hele veien.

Nå OR-porten. Den sender ut 1 når minst én inngang er 1. Gi hver inngang tallet 1. Legg sammen inngangene. Send ut 1 hvis summen er minst **1**.

Legg merke til hva som skjedde. De to portene fikk *nøyaktig de samme tallene* på inngangene. Det eneste som skiller dem, er hvor stor summen må være før utgangen slår om.



Figur G.1 AND og OR tegnet som vektede summer. Hver inngang ganges med et tall — en *vekt* — og summen sammenlignes med en *terskel*. De to portene har identiske vekter, og skiller seg bare i hvor høy summen må være.

Dette er ikke en tilfeldighet, og det gjelder flere porter enn disse to. De inverterende portene trenger bare at vekten snus til noe negativt.

Ta NAND, som skal gi 0 bare når begge inngangene er 1. Gi hver inngang tallet -1, og send ut 1 hvis summen er minst -1. Med inngangene 0 og 0 blir summen 0, som er større enn -1, så utgangen er 1. Med 1 og 0 blir summen -1, som akkurat holder, så utgangen er 1. Med 1 og 1 blir summen -2, som er for lavt, og utgangen er 0. Det er NAND.

En terskel kan altså godt være et negativt tall. «Minst -1» betyr bare «ikke lavere enn -1», og -2 er lavere enn -1.

Port	Vekter	Terskel
AND	$+1, +1$	≥ 2
OR	$+1, +1$	≥ 1
NOT	-1	≥ 0
NAND	$-1, -1$	≥ -1
NOR	$-1, -1$	≥ 0

Tabell G.1 Fem porter fra kapittel 5, skrevet som vektet sum med terskel. Negative vekter gir de inverterende portene. Legg merke til at AND og OR deler vekter, og at NAND og NOR gjør det samme.

Terskelen er altså en *innstilling*. Den kunne vært et annet tall, og da hadde kretsen gjort noe annet.

Og da melder neste spørsmål seg av seg selv: hva om vektene også kunne stilles inn?

To forbehold, og det andre er viktig

Dette er en nyttig måte å se portene på, men den har to grenser, og du bør kjenne begge.
Den er funksjonell, ikke fysisk. Inne i en ekte CMOS-port finnes det ingen summeringsnode og ingen sammenligning. Kapittel 6 viste hva som faktisk er der: transistorer i serie og parallell som leder eller ikke leder, og en utgang som trekkes mot forsyningen eller mot jord. Den vektede summen beskriver *sannhetstabellen*, ikke strømveiene. Den som sier at en NAND-port «er» en vektet sum, har byttet beskrivelsesnivå uten å si fra.
Den dekker ikke alle funksjoner. En sannhetstabell med to innganger har fire rader, og hver

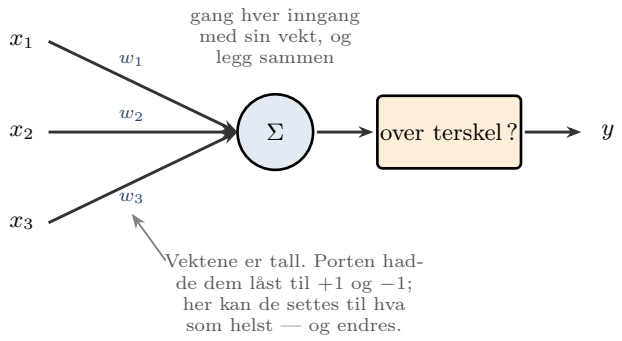
rad kan gi 0 eller 1. Det gir $2 \times 2 \times 2 \times 2 = 16$ mulige funksjoner til sammen — AND og OR og NAND er bare tre av dem. Fjorten av de seksten kan skrives som en vektet sum. De to som ikke kan, er XOR og XNOR.
Fjorten av seksten høres ut som nesten alle. Men regnestykket skalerer dårlig: med fire innganger er det bare 1882 av 65 536 funksjoner som lar seg skrive slik, altså under tre prosent. Jo flere innganger, desto sjeldnere blir det.
Det siste forbeholdet er ikke en detalj. De to unntakene XOR og XNOR er hovedsaken om to sider.

G.1.2 Vekter man kan stille inn

Spørsmålet ble stilt for lenge siden.

I 1943 beskrev Warren McCulloch og Walter Pitts nerveceller som noe som ligner logiske porter: en celle teller hvor mange av naboene som er aktive, og fyrer hvis det er mange nok. De gikk fra nervecellen til logikken. Vi går motsatt vei, men det er den samme broa.

Femten år senere snudde Frank Rosenblatt på det avgjørende punktet. Hos McCulloch og Pitts teller alle inngangene likt. Rosenblatt ga hver inngang sin egen vekt, og lot vektene være tall man kunne *endre*. Han kalte konstruksjonen en **perseptron**.



Figur G.2 Perseptronen. Den gjør akkurat det figur G.1 gjorde, men vektene er ikke lenger låst. Rosenblatt ga dessuten maskinen en regel for å finne vektene selv: gjetter den feil, flyttes vektene litt i retning av riktig svar.

Det Rosenblatt la til, og som er hele poenget, var en oppskrift for å finne vektene uten å regne dem ut på forhånd. Vis maskinen et eksempel. Gjetter den riktig, la vektene være. Gjetter den feil, flytt dem litt i retning av riktig svar. Gjenta mange tusen ganger.

Det er maskinlæring i én setning, og resten av dette appendikset trenger ikke mer enn det.

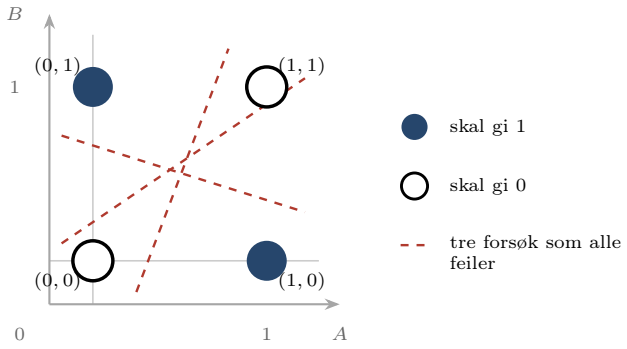
Vektene var fysiske motstander

Mark I Perceptron, bygget rundt 1958–1960, var en maskin man kunne ta på. Den hadde 400 fotoceller i et rutenett på 20 ganger 20, og vektene var *potensiometre* — justerbare motstander, som volumknappen på et gammelt radioapparat. Da maskinen skulle lære, vred små elektromotorer på knottene.
Legg den detaljen på minnet. I seksjon G.5 dukker den opp igjen, i en artikkel fra vår egen tid.

G.1.3 Vendingen: XOR

Advarselsboksen sa at XOR ikke kan skrives som en vektet sum. Her er hvorfor det betyr noe.

XOR skal gi 1 når nøyaktig én av inngangene er 1. Tegn de fire mulige inngangene som fire punkter, og fargelegg dem etter hva svaret skal være.



Figur G.3 XOR, tegnet som fire punkter. En vektet sum med terskel kan bare skille punktene med en rett linje, og de to fylte punktene ligger diagonalt. Uansett hvor linja legges, havner minst ett punkt på feil side.

En vektet sum med terskel kan bare trekke en rett linje gjennom dette bildet, og si «alt på den ene siden gir 1». De to punktene som skal gi 1, ligger diagonalt fra hverandre. Ingen rett linje klarer det.

Marvin Minsky og Seymour Papert skrev dette opp i boka *Perceptrons* i 1969, og det kjølnet interessen for feltet i mange år.

Hva Minsky og Papert faktisk mente

Påstanden deres blir ofte gjengitt som at nevrale nett ikke kan løse XOR. Det er ikke det de skrev, og de visste bedre.
De viste at *en enkelt* terskelenhet ikke kan. At flere lag kan, var klart for dem.
Innvendingen var en annen. Ingen visste hvordan man skulle få flere lag til å *lære*, siden Rosenblatts regel bare virker på ett lag. Og de trodde utvidelsen ville vise seg ufruktbar. Det siste skrev de rett ut, samtidig som de ba om å bli motbevist.
De tok feil i det. Men de tok ikke feil i at problemet var uløst i 1969.

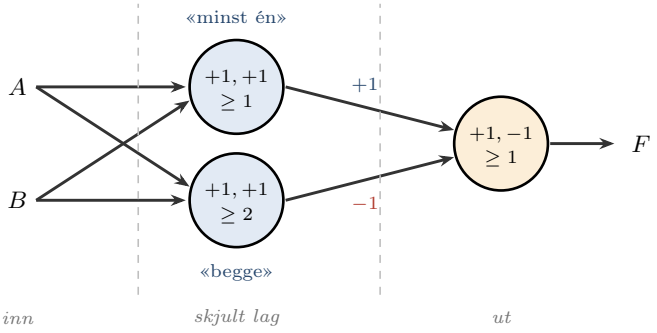
Løsningen er å sette enheter i to lag.

La den første enheten regne ut OR, altså «minst én». La den andre regne ut AND, altså «begge». Begge to er vektede summer, med vektene og tersklene fra tabell G.1.

Så setter man en tredje enhet til å lese svarene fra de to første. Den skal si ja når den første sier ja og den andre sier nei — altså er den også en vektet sum: vekt $+1$ på svaret fra OR, vekt -1 på svaret fra AND, og terskel ≥ 1.

Prøv den. Er begge inngangene 0, gir OR 0 og AND 0, og summen blir 0. For lavt. Er nøyaktig én inngang 1, gir OR 1 og AND 0, og summen blir 1. Høyt nok. Er begge 1, gir både OR og AND 1, og summen blir $1 - 1 = 0$. For lavt igjen.

XOR er altså «minst én, og ikke begge». Og hver eneste del av konstruksjonen er den samme vektede summen som før — det nye er bare at de står i to etasjer.



Figur G.4 XOR bygget av tre vektede summer. Tallene i hver sirkel er vektene over og terskelen under. Legg merke til den negative vekten inn til utgangsenheten: det er den som gjør «og ikke begge».

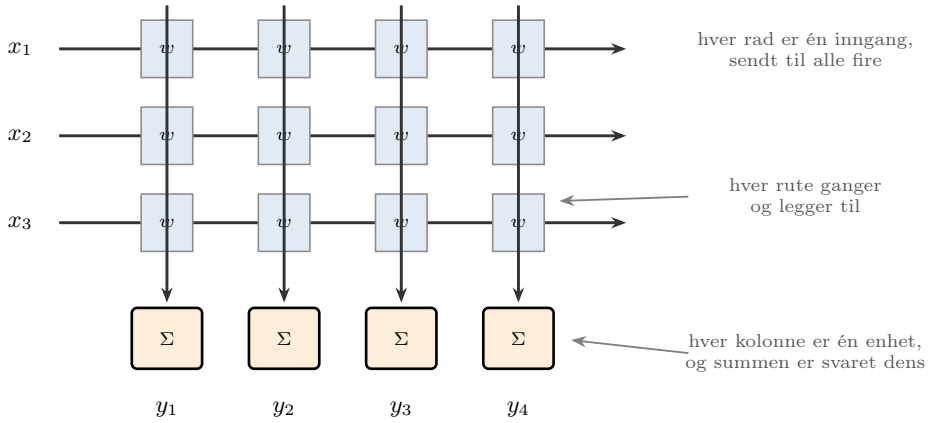
Det laget i midten — det som verken er inngang eller utgang — kalles et **skjult lag**. Ordet er misvisende, for det er ikke mer skjult enn resten. Det heter bare det fordi det ikke vender ut mot verden.

Alt som kalles et nevralt nettverk i dag, er lag på lag av dette.

G.1.4 Hele regnestykket, i en tabell

Nå kan vi se hva en maskin faktisk må gjøre.

Ta et lag med tre innganger og fire enheter. Hver enhet har sin egen vekt for hver inngang, så det er tolv vekter til sammen. Sett dem opp i et rutenett.



Figur G.5 Et lag med tre innganger og fire enheter. Tolv vekter, tolv ganginger, og fire summer. Matematikere kaller et slikt rutenett en *matrise*, og hele operasjonen en matrisemultiplikasjon — men du trenger ikke navnet for å se hva som skjer.

Følg én kolonne nedover. Hver rute henter inn en inngang, ganger den med sin vekt, og legger resultatet til det som kom ovenfra. Nederst står summen.

Det er alt. Gang, og legg til. Gang, og legg til.

Operasjonen har et navn, og det er verdt å lære fordi det går igjen i alt som er skrevet om dette: **MAC**, forkortelse for *multiply–accumulate*, altså gang-og-akkumuler. En MAC er én ganging pluss én addisjon.

Hvor mange ganger?

Rutenettet over har tolv ruter, altså tolv MAC-operasjoner. Det er ikke mye.
Men et lag i en modell som kjenner igjen hva som er på et bilde, har gjerne noen hundre innganger og noen hundre enheter. Det gir titusener av vekter i ett lag. En modell har mange titalls lag. Og hele seansen gjentas for hvert bilde.
En liten bildemodell som kjører på en telefon, bruker typisk noen hundre millioner MAC-operasjoner på én eneste knipsing. En stor språkmodell bruker mange milliarder på hvert ord den skriver.
Det er mange operasjoner. Men det er bare én *type* operasjon, den er kjent på forhånd, og ingen av dem avhenger av hva svaret på den forrige ble.
Hold fast på den siste setningen. Den er grunnen til at det går an å bygge en egen brikke.

Og treningen?

Alt over handler om å *bruke* en ferdig modell. Det kalles **inferens**, og det er det telefonen din gjør.
Å finne vektene i utgangspunktet er en annen og mye dyrere jobb. Den kalles **trening**, foregår i datasentre, og bruker en teknikk som heter tilbakepropagering: man måler hvor feil svaret ble, og regner bakover gjennom lagene for å finne ut hvilken vei hver enkelt vekt bør flyttes.
Trening er regnestykket fra figur G.5 kjørt forlengs og baklengs millioner av ganger, med høyere presisjon. Resten av dette appendikset handler om inferens, siden det er den delen som skjer i lomma di.

G.2 Hvorfor en vanlig prosessor er feil verktøy

Prosessoren fra kapittel 4 på side 79 kan gange. Den kan legge sammen. Hvorfor holder det ikke?

Svaret er at den bruker nesten all energien sin på noe annet enn regnestykket.

G.2.1 Papirarbeidet koster mer enn jobben

Hent fram hent-utfør-syklusen fra kapittel 3 på side 55. For hver eneste instruksjon må prosessoren hente instruksjonen fra minnet, finne ut hva den betyr, hente operandene fra registrene, utføre operasjonen, og skrive resultatet tilbake.

Fire av de fem trinnene handler om å administrere arbeidet. Bare ett er arbeidet.

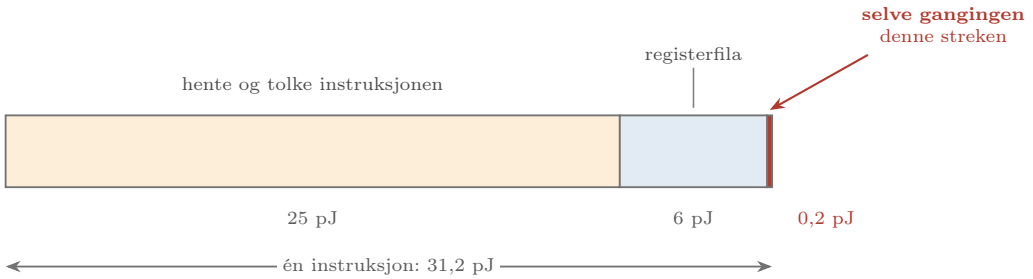
For en instruksjon som gjør noe stort, er det grei handel. For en MAC er det ikke det.

Nå trenger vi en enhet for energi, og det er bare én i hele appendikset. Energi måles i **joule**. én joule er omtrent det som skal til for å løfte et eple én meter opp. Kretsene vi snakker om, bruker langt mindre enn det, så tallene kommer i **pikojoule** — en milliondels milliondels joule. Og én watt, enheten kapittel 6 på side 143 brukte om varme, er rett og slett én joule i sekundet.

Tallene under kommer fra den samme måleserien som seksjon G.3 bygger på. Å hente instruksjonen og styre arbeidet rundt den koster rundt 25 pikojoule. Registerbruken koster omkring 6. Selve gangingen av to åttebitstall koster **0,2**.

De 6 gjelder hele registerfila for én instruksjon, med flere lesinger og en skriving. Det er altså ikke prisen på én enkelt lesing.

Papirarbeidet koster altså godt over hundre ganger mer enn jobben.



Figur G.6 Hva det koster en vanlig prosessor å utføre én eneste ganging. Den tynne røde streken helt til høyre er regnestykket. Alt det andre er administrasjon — å finne ut hva som skal gjøres, og å hente fram tallene det skal gjøres med.

Et rekneverk med for mye administrasjon

Tenk deg at du skal stemple hundre tusen konvolutter. Selve stemplingen tar et halvt sekund. Men reglene sier at du først må gå bort til hylla og hente arbeidsordren. Lese den. Hente stempelet fra skapet. Stemple. Levere ordren tilbake, og rydde stempelet på plass. Og så begynne på nytt for neste konvolutt.
Nå tar hver konvolutt to minutter, og nesten alt er gåing.
En prosessor som kjører et nevralt nettverk, gjør dette. Arbeidsordren er instruksjonen, stempelet er tallene den skal regne med, og hylla og skapet er minnet og registrene. Den henter en ny arbeidsordre for hver eneste ganging, selv om ordren er helt lik forrige gang og helt lik neste.

Prosessoren har triks mot dette. Flere kjerner tar flere konvolutter samtidig, og vektorinstruksjoner lar én ordre dekke åtte eller seksten ganginger i stedet for én. Det hjelper, og det er nettopp derfor et grafikkort er langt bedre til dette enn en vanlig prosessor: et grafikkort er bygget for å gjøre det samme med svært mange tall samtidig.

Men triksene flytter grensen, de fjerner den ikke.

G.2.2 Alt prosessoren er god til, er bortkastet her

Det er verre enn som så. Prosessoren er full av kretser som er der for å takle *uforutsigbarhet*, og et nevralt nettverk er ikke uforutsigbart i det hele tatt.

Den har sprangprediksjon, som gjetter hvilken vei en `if` går. Her er det ingen `if`-er.

Den har cache, som gjetter hvilke data du vil trenge snart, basert på hva du nettopp brukte. Her vet man nøyaktig hvilke tall som trengs, i hvilken rekkefølge, lenge før man begynner.

Den har kretser som kan stokke om på rekkefølgen av instruksjoner for å unngå venting. Her er rekkefølgen lagt i stein.

Alle disse kretsene opptar plass på brikken, og de koster strøm hver eneste klokkesykkel. For en nettleser eller et operativsystem er de gull verdt. For tolv ganginger i et rutenett er de rent sløseri.

Flaskehalsen har et navn

At maskinen bruker mer krefter på å flytte tall fram og tilbake enn på å regne med dem, er ikke et nytt problem. John Backus kalte det *von Neumann-flaskehalsen* i Turing-forelesningen sin, holdt i 1977 og trykt året etter.
Han mente riktignok noe videre enn maskinvaren. Hovedanliggendet hans var at selve *tenkemåten*

i programmeringsspråk tvinger oss til å behandle ett og ett ord om gangen. Men uttrykket festet seg om maskinvaren, og det er den betydningen du vil møte.
Kapittel 7 på side 173 tar den samme flaskehalsen fra en annen kant.

Konklusjonen er ubehagelig for en generell maskin: den er bygget for å kunne gjøre alt, og det den koster, betaler den hver gang den gjør én ting.

Så hva skjer hvis man gir slipp på «alt»?

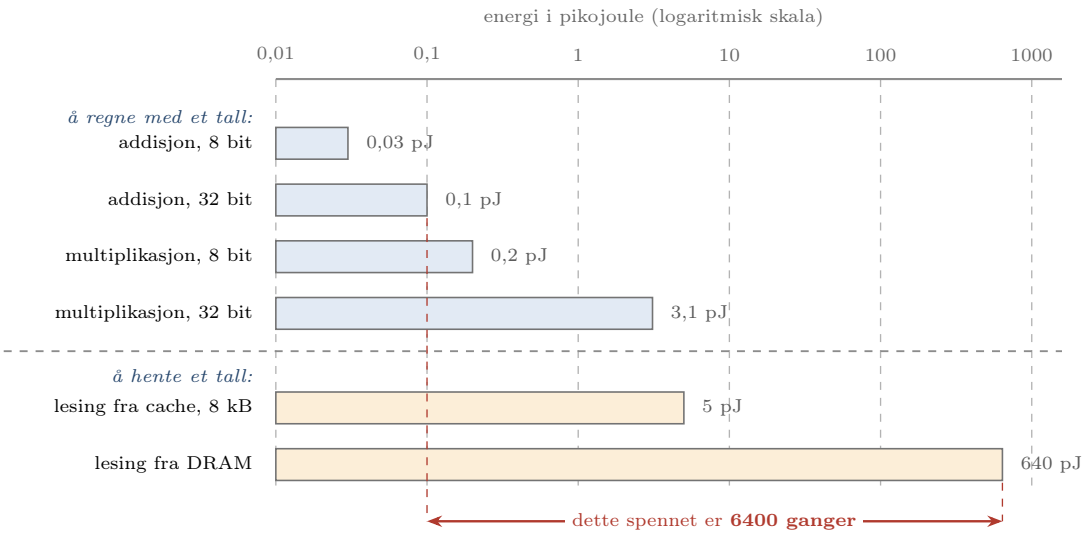
G.3 Hva ting koster

Kapittel 6 på side 143 slo fast at hver eneste vending koster en slurk energi, og at slurkene blir til varme. Nå skal vi telle slurker.

Dette er den viktigste seksjonen i appendikset. Alt som kommer etterpå, er følger av tallene her.

G.3.1 Tabellen som forklarer alt

I 2014 satte Mark Horowitz opp et regnskap over hva enkeltoperasjoner faktisk koster i energi. Tallene er blitt sitert i tusenvis av artikler siden, og de står fortsatt.



Figur G.7 Energi per operasjon, målt på en brikke med 45 nanometers prosess og 0,9 volt forsyning. Merk at skalaen er logaritmisk: hvert hakk bortover er ti ganger mer, så søylene ser langt jevnere ut enn tallene er. Spennet nederst dekker fire hakk, altså en faktor på over seks tusen. Tallene er fra 2014 og gjelder den prosessen. Forholdet mellom de to gruppene er om noe blitt verre siden.

Les figuren én gang til, og legg merke til hvor den stiplede linja går.

Alt over linja er regning. Alt under er henting. Og den dyreste regneoperasjonen koster mindre enn den billigste hentingen.

G.3.2 Tallet som bestemmer alt

Sammenlign de to ytterpunktene. En addisjon av to trettitobitstall koster 0,1 pikojoule. Å hente ett tall fra arbeidsminnet koster 640.

Det er **over seks tusen ganger**.

Kniven og butikken

Du skal hakke en løk. Selve hakkingen tar et halvt sekund. Å gå i butikken etter løken tar en time. Turen tar altså omtrent sju tusen ganger så lang tid som arbeidet. Det er samme forhold som mellom en henting og en addisjon.
Vil du bli fortere ferdig, er det ingen vits i å kjøpe en skarpere kniv. Selv en kniv som hakker på null tid, sparer deg for et halvt sekund av en time og et halvt sekund.
Det du må gjøre, er å slippe unna turen: ha løken i skapet fra før, eller handle inn til mange middager på én gang.
Dette er hele arkitekturen bak en KI-brikke, og det er også hele poenget med minnehierarkiet i kapittel 7 på side 173. To ganske ulike deler av faget, med nøyaktig samme innsikt under.

Og her kommer det som gjør nevrale nett spesielle: de er *særlig* utsatt for dette.

En modell består nesten bare av vekter, og hver eneste vekt må hentes før den kan brukes. Verre: kjører du modellen én gang, på én bildefil, blir hver vekt brukt nøyaktig én gang. Det er løkturen uten muligheten til å lage flere middager.

Henter maskinen hver vekt fra arbeidsminnet rett før den skal ganges med noe, bruker den 640 pikojoule på å hente og 0,2 på å gange. Da er over 99,9 prosent av energien brukt på transport.

Hele resten av appendikset handler i praksis om de to utveiene fra den situasjonen: å gjøre hver tur kortere, og å få mer ut av hver tur.

Om tallene i denne seksjonen

To ting er verdt å vite før du tar dem med videre.
De er fra 2014, og gjelder én bestemt produksjonsprosess. Regneoperasjoner har blitt billigere siden. Hentinger fra arbeidsminnet har blitt det i langt mindre grad, fordi arbeidsminnet sitter i en egen brikke og signalene må ut av silisiumet og over en kobberledning. Forholdet mellom de to gruppene er derfor ikke blitt bedre — det er blitt verre.
Tall som dette blir ofte feilsitert. Den vanligste feilen er å blande sammen målinger som gjelder ulike tallbredder. Originaltallene for minne gjelder hentinger på 64 bits, mens regnetallene gjelder 32. Alt i figuren er regnet om til 32. Blander man de to, bommer man med en faktor to.
Den nest vanligste feilen er å si «et par hundre ganger». Det er et helt annet regnestykke: det sammenligner en henting med en hel gange-og-legg-til, målt på en annen brikke.
Oppgir du tall herfra, ta med at de gjelder 45 nanometer og 32-bits verdier.

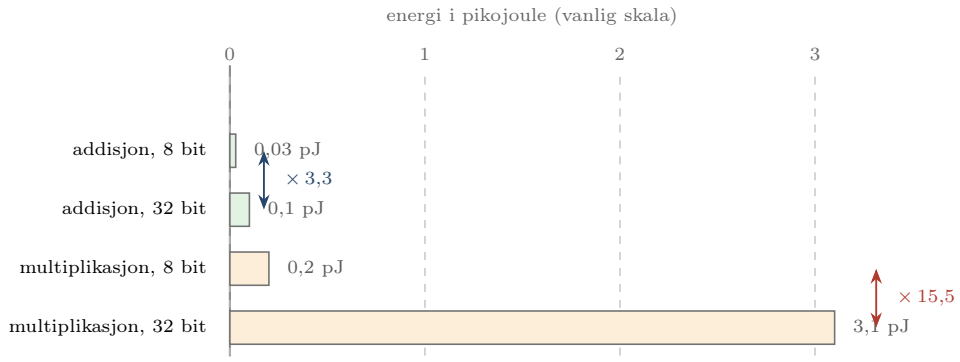
G.3.3 Færre siffer, mindre krets

Det er ett tall til i figur G.7 som fortjener oppmerksomhet, og det handler om noe kapittel 2 på side 13 tok opp: hvor mange bits et tall egentlig trenger.

Sammenlign de to multiplikasjonene. Åtte bits koster 0,2 pikojoule. Trettito bits koster 3,1. Fire ganger så mange siffer gir femten og en halv gang så mye energi.

Sammenlign så de to addisjonene. Åtte bits koster 0,03, trettito koster 0,1. Fire ganger så mange siffer

gir drøyt tre ganger så mye.



Figur G.8 Hva det koster å gå fra åtte til trettito bits, denne gangen på vanlig skala. De to addisjonssøylene er nesten usynlige, og det er hele poenget: ved siden av en trettitobits multiplikasjon er en addisjon gratis uansett hvor mange siffer den har. Multiplikasjonen vokser omtrent med kvadratet av bitbredden, addisjonen omtrent i takt med den.

Grunnen ligger i hvordan de to kretsene er bygget. En adder behandler ett sifferpar om gangen og sender menten videre, så fire ganger så mange siffer gir fire ganger så mange trinn. Det så du i kapittel 5 på side 105, der fulladderne sto på rekke. Målingen sier 3,3, og det er omtrent fire.

En multiplikator må derimot lage et delprodukt for *hvert par* av siffer. Gang 11 med 11 i hodet, så ser du det: to siffer ganger to siffer gir fire delprodukter som skal legges sammen. Firedobler du sifferantallet, blir det seksten ganger så mange par — og $4 \times 4 = 16$ er nettopp der målingen på 15,5 ligger.

Adderen vokser altså i takt med sifferantallet. Multiplikatoren vokser med kvadratet av det. I praksis litt svakere enn kvadratet, men nær nok til at tommelfingerregelen holder.

Og siden et nevralt nettverk nesten bare gjør multiplikasjoner, er det den kurven som gjelder.

Dette er grunnen til at KI-maskinvare regner med så få siffer som den slipper unna med. Teknikken heter **kvantisering**: man tar en modell som er trent med trettito bits per vekt, og skriver vektene om til åtte bits før de skal brukes.

Format	Bits	Oppdeling	Hva det brukes til
FP32	32	1 + 8 + 23	Trening, og alt som skal være trygt.
bfloat16	16	1 + 8 + 7	Trening. Samme område som FP32, færre gjeldende siffer.
FP16	16	1 + 5 + 10	Trening og inferens. Flere gjeldende siffer enn bfloat16, men mindre område.
INT8	8	heltall	Inferens. Det vanlige formatet i en telefon.
INT4	4	heltall	Inferens der plass betyr mest. Krever mer arbeid for å fungere.

Tabell G.2 Tallformatene du vil støte på. Oppdelingen er fortegn, eksponent og mantisse, slik kapittel 2 satte det opp. De to sekstenbitsformatene deler bitene ulikt, og det er hele forskjellen mellom dem.

bfloat16 mot FP16

De to har like mange bits og er likevel ikke like, og forskjellen er lærerik.
FP16 bruker fem bits på eksponenten og ti på mantissen. bfloat16 bruker åtte på eksponenten og sju på mantissen — altså nøyaktig samme eksponent som FP32.
Det betyr at bfloat16 dekker *samme tallområde* som FP32, men med grovere oppløsning. FP16 har finere oppløsning, men treffer taket og gulvet mye tidligere.
For trening viste det seg at området er viktigere enn presisjonen. Små verdier som rundes helt til null, ødelegger læringen; litt avrunding gjør det ikke. Derfor lagde Google bfloat16 til sine egne brikker, og formatet har siden spredt seg.
Legg merke til hva det betyr: bfloat16 er *mindre* presist enn FP16, ikke mer. Det er bare mer robust.

«Åtte bits er gratis» er oversalg

Du vil lese at kvantisering til åtte bits ikke koster noe. Det stemmer ofte, og det stemmer ikke alltid.
Først et ord om hva **skalering** er her. Åtte bits rekker bare til 256 ulike verdier. Før vektene kan skrives om, må man derfor bestemme hvilket tallområde de 256 trinnene skal dekke.
Velger man én slik faktor for hele laget, må den romme både den største og den minste vekten der. Er avstanden mellom dem stor, blir trinnene grove for alle. Velger man i stedet én faktor per liten gruppe vekter, blir trinnene finere der det trengs.
For modeller som kjenner igjen innholdet i bilder, holder resultatet seg da vanligvis innenfor et par prosentpoeng av originalen. Med «treffsikkerhet» menes her hvor stor andel av bildene modellen setter riktig navn på: faller den fra 76 til 74 prosent, er det tålelig.
For store språkmodeller kan det gå riktig galt. Forbi en viss størrelse dukker det opp enkelte tallverdier som er mye større enn alle de andre, og de er samlet i noen få posisjoner. Må skaleringen romme dem, mister alt det andre oppløsning. Et kjent eksempel: en stor språkmodell falt fra 71,6 til 32,3 prosent riktige svar med rett fram kvantisering. Det er ikke en liten forringelse, det er en ødelagt modell. Løsningen ble å kjøre de få store verdiene i seksten bits og resten i åtte.
En tredje ting som ofte glemmes: sifrene du *ganger* med, og sifrene du *summerer* i, trenger ikke

være like mange. En vanlig oppskrift ganger i åtte bits og legger sammen i trettito, nettopp fordi summen av mange tall vokser seg stor selv om hvert ledd er lite.
Den ærlige formuleringen er altså: åtte bits kan gjøres nesten gratis — for de riktige modellene, med riktig skalering, og med en sum som holdes i flere siffer enn det man ganger med.

G.3.4 Budsjettet

Nå kan vi svare på spørsmålet fra første side.

En telefon tåler omtrent tre og et halvt watt vedvarende. Ikke fordi batteriet ikke klarer mer, men fordi det ikke finnes noe sted å gjøre av varmen. Det er ingen vifte, ingen kjøleribbe, bare et glass- og metallskall som ligger mot hånda di. Blir den for varm, skrur telefonen ned farten på seg selv.

Et grafikkort i et datasenter tåler i størrelsesorden tusen watt. Det har en vifte, det sitter i et kabinett med flere vifter, og rommet det står i, blir kjølt ned av et eget anlegg.

Forholdet er et sted mellom hundre og tre hundre til én.

Det er altså ikke slik at telefonen gjør det samme som datasenteret, bare litt tregere. Den har et helt annet budsjett, og den må kjøpe seg ytelse et annet sted enn med effekt. Det eneste stedet som er igjen, er å få mer ut av hver pikojoule.

Og det er nøyaktig det en egen brikke gjør.

G.4 Brikken som er bygget for dette

En akselerator for nevrale nett følger tre regler, og alle tre er svar på tallene i forrige seksjon.

Bruk så få siffer som mulig. Ha så mange multiplikatorer som mulig. Og flytt tallene så kort som mulig.

G.4.1 Regel én: mange, små multiplikatorer

Hvis en åttebits multiplikator bruker en femtendel av energien til en trettitobits, kan man ha femten av dem for samme strømregning. Arealet følger omtrent samme mønster — litt svakere, men i samme størrelsesorden — så de får plass også.

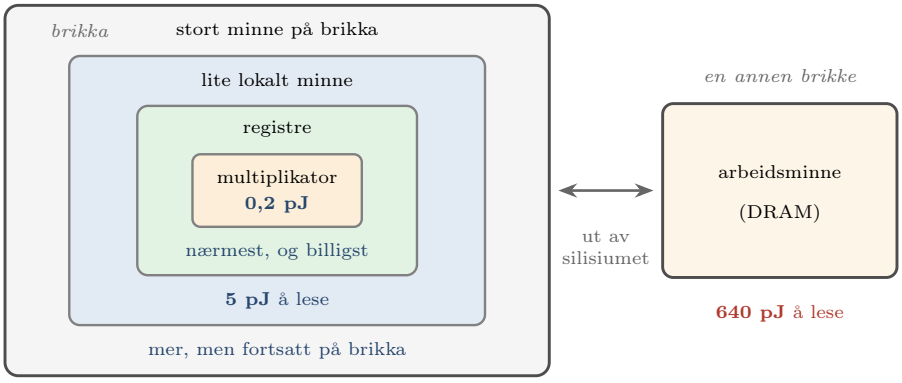
Det er en god handel, for arbeidet består nettopp av svært mange uavhengige ganginger. Hundre små multiplikatorer som jobber samtidig, slår én stor som jobber fort.

Her møter vi igjen noe kapittel 6 på side 143 regnet ut. Å øke frekvensen koster *mer* enn lineært i effekt, fordi høyere fart som regel også krever høyere spenning — og spenningen teller i annen. Eksempelet i kapittel 6 på side 143 endte på nær tre ganger effekten for dobbelt så høy frekvens. Å legge til flere enheter som går i samme takt, koster bare lineært.

Valget er derfor det samme som bransjen tok i 2004, men tatt mye lenger. Flere kjerner var svaret for generelle maskiner. For denne ene oppgaven kan man gå fra noen få kjerner til mange tusen regneceller, nettopp fordi oppgaven er så ensformig at cellene ikke trenger å være smarte.

G.4.2 Regel to: hold tallene nær

Figur G.7 viste at en henting fra arbeidsminnet koster over seks tusen ganger en addisjon. Men ikke alle hentinger er like dyre. Det avgjørende er hvor langt tallet må reise.



Figur G.9 Energien avhenger av avstanden. Jo lenger ut i figuren tallet må hentes fra, desto dyrere blir det, og spranget ut av brikka er det desidert største. Det lille lokale minnet her er den samme cachen på 8 kB som i figur G.7, og tallene er de samme.

Spranget som koster, er det ut av silisiumet. Inne på brikka går signalene noen millimeter gjennom korte ledninger. Ut til arbeidsminnet må de gjennom loddepunkter, over et kretskort og inn i en annen brikke. Hele den strekningen er en kapasitans som må fylles og tømmes — badekaret fra kapittel 6 på side 143, bare mye større.

Derfor bygger man akseleratoren slik at vektene lastes inn én gang og blir liggende, mens dataene renner forbi dem.

G.4.3 Regel tre: la tallene flyte

Og da er vi framme ved selve konstruksjonen.

I 1979 beskrev H. T. Kung og Charles Leiserson en måte å organisere regneceller på som de kalte en **systolisk matrise**. Navnet kommer fra *systole*, det medisinske ordet for hjertets sammentrekning, og bildet er godt valgt: data pumpes rytmisk gjennom rutenettet, ett steg for hvert slag.

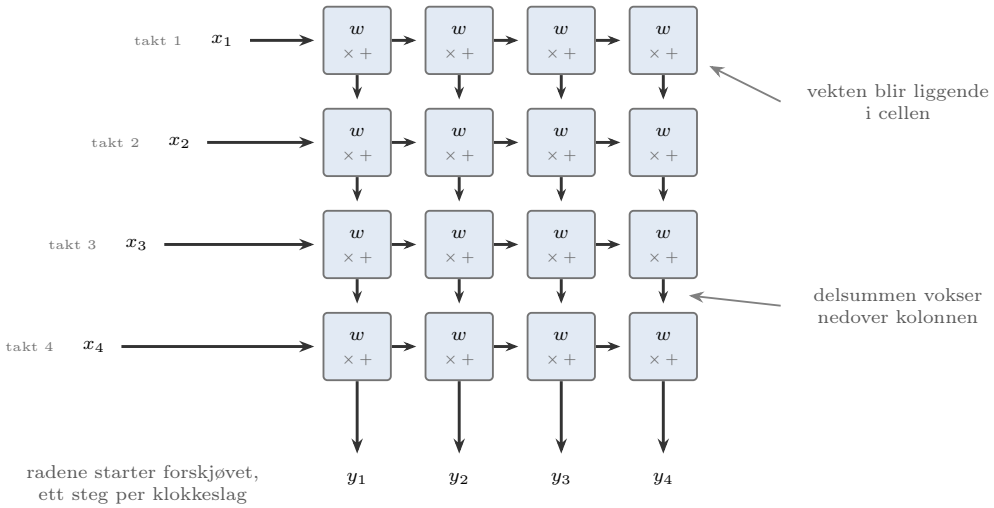
Gevinsten ligger i det siste, og det er verdt å være presis om hva som gjenbrukes.

Et inngangstall hentes én gang og brukes av *alle* cellene i raden. I figuren er det fire ganginger på én henting. I en ekte matrise er det flere hundre.

Vektene blir liggende i cellene og hentes aldri igjen så lenge samme lag kjører. Og sender man flere bilder gjennom etter hverandre uten å bytte vekter, betaler man for vektene én gang og får dem med på kjøpet for hvert bilde.

Det er kjøkkenanalogien satt i system: én tur i butikken, mange middager.

En celle har heller ingen kontakt med noe annet enn naboene sine. Det var hele motivasjonen i 1979, og den er den samme i dag: lange ledninger på en brikke er trege og dyre, mens korte er billige. Enda et ekko fra kapittel 6 på side 143.



Figur G.10 En systolisk matrise. Rutenettet ligner på figur G.5, men det avgjørende her er tiden: hver celle holder én vekt, tar imot et tall fra venstre og en delsum ovenfra, ganger, legger til, og sender begge videre på neste klokkeslag. Derfor starter radene forskjøvet — tallene i rad to skal møte delsummen fra rad én akkurat når den kommer ned.

Googles første forsøk, og hva som begrenset det

Google bygget en slik brikke og tok den i bruk internt fra 2015. Den hadde en systolisk matrise på 256 ganger 256, altså **65 536** regneceller, som ganget åttebitstall og akkumulerte i trettito. Klokka gikk på 700 MHz — langt under en vanlig prosessor.

Det interessante står i deres egen rapport. De kaller minnebåndbredden sin «meager», altså skrinn. Brikken hadde regnekraft nok, men slet med å få tallene fort nok inn.

Det er figur G.7 igjen. Selv når man bygger maskinen spesielt for oppgaven og fyller den med regneceller, er det fortsatt hentingen som er flaskehalsen. Neste generasjon fikk derfor en helt annen minneteknologi.

G.4.4 Hva som faktisk sitter i telefonen

Kretsen i telefonen din går under flere navn. NPU, for *neural processing unit*. Nevral motor. KI-motor. Markedsavdelingene er ikke enige med hverandre.

Under navnene er det stort sett samme sak. Et felt med mange tusen små multiplikatorer som regner i åtte bits. Lokalt minne tett på. Og en enkel styreenhet som mater dem med én og samme type ordre om og om igjen.

Kretsen sitter som en egen blokk inne i den samme brikken som prosessorkjernene og grafikkdelen, og deler arbeidsminne med dem.

Den kan ikke kjøre et operativsystem. Den kan ikke kjøre en nettleser. Den kan gjøre matrisemultiplikasjon i lav presisjon, og den gjør det med en brøkdel av energien prosessoren ville brukt.

Hvem var først?

Spørsmålet har ikke ett svar, og det er verdt å vite hvorfor.
Huawei annonserte en egen nevral kjerne i systembrikken Kirin 970 den 2. september 2017. Apple annonserte sin Neural Engine i A11 Bionic ti dager senere, den 12. september. Men Apples telefon kom i butikk 22. september, mens Huaweis kom i oktober.
Begge selskapene har markedsført seg som først, og begge har rett innenfor hver sin definisjon. Legger man til et tredje krav — at utviklere faktisk kunne bruke kretsen — forskyves hele diskusjonen til året etter, for Apples første generasjon var forbeholdt selskapets egne funksjoner.

G.4.5 Og hvorfor ikke bare bruke skyen?

Energiregnskapet er én begrunnelse for å ha kretsen lokalt. Det finnes fire til, og de er like viktige.

Svartiden. En tur til et datasenter og tilbake tar titalls millisekunder i beste fall, og langt mer på dårlig dekning. Skal kameraet justere fokus etter hva det ser mens du beveger deg, holder ikke det.

Dataene blir værende. En modell som kjører lokalt, trenger ikke sende bildet, stemmen eller teksten din noe sted. For helsedata, ansikter og tastetrykk er det forskjellen på hva som er mulig å tilby i det hele tatt.

Det virker uten nett. Oversettelse i en tunnel, diktering på et fly, kameraet på fjellet.

Noen må betale for datasenteret. Hvert svar som regnes ut i skyen, koster leverandøren strøm og maskintid. Regnes det ut på telefonen, er regningen allerede betalt av den som kjøpte telefonen.

Og det er et poeng til som hører hjemme i dette appendikset: *radioen koster også energi.* Å sende et bilde over mobilnettet bruker langt mer strøm enn å sende det noen millimeter bortover på en brikke. Velger man å regne i skyen for å spare strøm lokalt, kan man ende opp med å bruke mer.

G.5 Å slutte å flytte tallene

Alt i forrige seksjon var vanlig digital elektronikk. Transistorer som leder eller stenger, nuller og enere, en klokke som holder takten. Nøyaktig det kapittel 6 på side 143 beskrev, bare satt sammen på en annen måte.

Denne seksjonen handler om noe annet. Den er den mest spekulative delen av appendikset, og lite av det finnes i produkter du kan kjøpe. Men den er med, fordi den tar ideen fra forrige seksjon helt ut — og fordi den endelig gir Ohms lov noe å gjøre.

G.5.1 Det logiske neste steget

Argumentet så langt har vært: det koster mer å flytte et tall enn å regne med det, så flytt det så kort som mulig.

Ta det til sin ytterste konsekvens. Hvorfor flytte tallet i det hele tatt? Hvorfor ikke regne *der det ligger*?

Det høres ut som et ordspill, men det er en helt konkret idé, og den bygger på noe du allerede kan.

G.5.2 Ohms lov gjør en ganging

Appendiks D innførte Ohms lov. Den sier at strømmen gjennom noe er spenningen delt på motstanden.

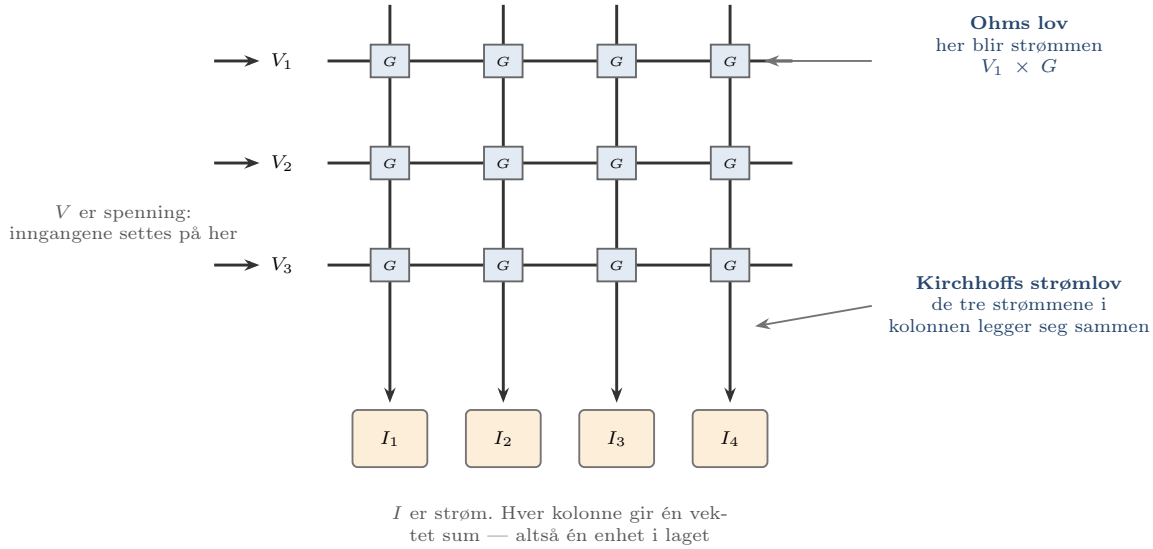
Motstand er et mål på hvor dårlig noe leder. Tar man én delt på motstanden, får man i stedet et mål på hvor *godt* det leder, og den størrelsen heter **konduktans**. Skriver man loven med konduktans i stedet for motstand, blir delingen til en ganging:

$$\text{strøm} \;=\; \text{spenning} \;\times\; \text{konduktans}$$

Legg merke til hva som skjedde. Dette er en multiplikasjon, og den utføres ikke av en krets med transistorer. Den utføres av fysikken, i det øyeblikket spenningen står på.

Og det finnes en lov til som er like nyttig. Den heter **Kirchhoffs strømlov**, og den sier at når flere strømmer møtes i samme ledning, legger de seg sammen. Det følger av at ladning ikke blir borte: det som renner inn i et punkt, må renne ut igjen.

To fysiske lover, og de gjør akkurat de to operasjonene et nevralt nettverk trenger: gang, og legg sammen.



Figur G.11 Et kryssfelt. Vektene er lagret som konduktanser i krysningene, inngangene legges på radene som spenninger, og strømmene som møtes i hver kolonne er ferdige vektede summer. Hele rutenettet fra figur G.5 skjer på én gang, i det øyeblikket spenningene settes på.

Studer figuren litt. Vektene ligger lagret som konduktanser — altså som hvor godt hvert lille element leder — i hver eneste krysning. Inngangene legges på som spenninger på radene. I hver krysning gjør Ohms lov en ganging helt av seg selv, og i hver kolonne legger strømmene seg sammen på veien ned.

Det som i figur G.5 var tolv ganginger og fire summeringer, skjer her i samme øyeblikk som spenningene settes på. Selve regnestykket trenger verken klokke, instruksjoner eller henting av vekter — men som vi straks skal se, trenger kretsene rundt det både styring og klokke.

Vektene ble uansett aldri flyttet noe sted, for de *er* kretsen.

Teknikken kalles **regning i minnet**, og komponentene er en type minne der motstanden kan endres og beholdes.

Hele sirkelen, på seksti år

Husker du potensiometrene i Mark I Perceptron?
Rosenblatts maskin lagret vektene sine som justerbare motstander, fordi det var det man hadde i 1958. Så kom den digitale elektronikken, vektene ble til tall i et minne, og motstandene forsvant. Og nå, etter seksti år med digitale brikker, foreslår forskere å lagre vektene som justerbare motstander igjen — fordi det er billigere enn å flytte tall.
Ideen var aldri dum. Den var bare fryktelig upraktisk helt til flytting av data ble det dyreste man gjorde.

Om navnet på komponenten

Elementene i krysningene blir ofte kalt *memristorer*. Det ordet har en historie som er verdt å kjenne hvis du leser videre.
Leon Chua postulerte i 1971 at det burde finnes et fjerde grunnleggende kretselement ved siden av motstanden, kondensatoren og spolen, og kalte det memristoren. I 2008 publiserte et forskningsmiljø ved HP en artikkel med tittelen «The missing memristor found».
Det har vært omstridt siden, og innvendingene er publisert i fagfellevurderte tidsskrifter. Kjernen i striden er at komponenten de fant, oppfyller en videre definisjon fra 1976, men neppe det opprinnelige postulatet fra 1971.
Boka bruker derfor det nøytrale *motstandssvitsjende minne*. Møter du ordet memristor, er det stort sett det samme som menes — men vit at navnet er omdiskutert.

G.5.3 Hvorfor det ikke har tatt over

Ideen er over femten år gammel og fortsatt ikke i telefonen din. Grunnene er lærerike, for de handler alle om noe kapittel 6 på side 143 var innom.

Det er analogt igjen. Kapittel 6 på side 143 brukte flere sider på hvorfor digital elektronikk er så robust, og ga oss to ord for det. Hver port henter en fersk verdi fra forsyningen og kaster den slitne som kom inn; det er *signalgjenoppretting*. Og avstanden mellom det en port lover og det neste port godtar, er *støymarginen*. Til sammen er de grunnen til at en kopi av en kopi av en kopi overlever.

Et kryssfelt gir begge deler opp. Strømmen i en kolonne er en analog størrelse, og det er ingen som vasker den ren.

Cellene er ikke like. To elementer som ble programmert til samme konduktans, leder litt ulikt. Verdien driver dessuten langsomt over tid, slik at regnefeilen *vokser* mens brikken står der. Den fysiske grunnen er at den ledende strukturen inne i cellen bare er noen få nanometer bred, så fluktuasjoner på atomnivå slår merkbart ut. I praksis får man typisk fire til seks bits som kan stoles på — langt under de åtte en digital brikke gir uten å blunke.

Strømmen tar snarveier. I et stort rutenett finner strømmen veier gjennom celler den ikke var ment å gå gjennom. Og ledningene har selv motstand nok til at en celle langt fra driveren ser en annen spenning enn en celle nær. Begge deler gjør store felt vanskeligere enn små.

Og den dyreste innvendingen til slutt. Resten av maskinen er digital. Inngangene må gjøres om fra tall til spenninger, og strømmene ut må gjøres om til tall igjen. De omformerne er analoge kretser, og i analysene av slike systemer står de for omtrent halvparten av energien og en tredel av arealet.

Det siste punktet er det virkelig alvorlige, og grunnen er verdt å merke seg. Moores lov fra kapittel 6 på side 143 — at transistorene krymper og blir flere for hver generasjon — gjelder digital logikk. *Analoge kretser krymper ikke på samme måte.* Den digitale delen av regnestykket blir billigere år for år, mens omformerne står omtrent stille. Gevinsten i kryssfeltet spises altså opp i periferien, og andelen blir

relativt verre med hver nye generasjon, ikke bedre.

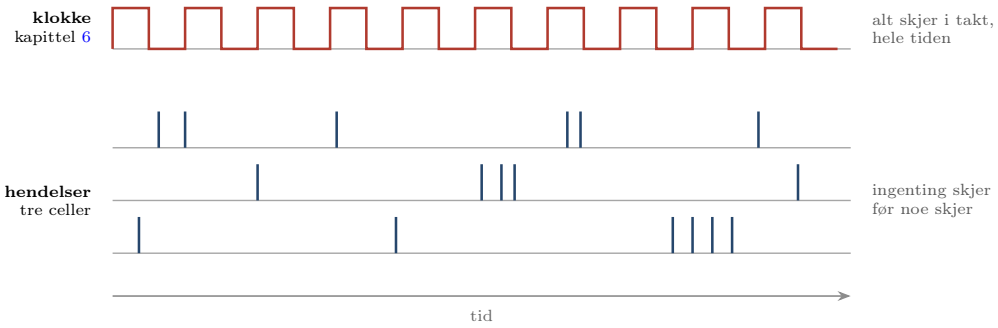
G.5.4 Neuromorf, og hva ordet faktisk betyr

Ordet **neuromorf** brukes på to måter, og de er svært ulike.

I løs forstand betyr det omtrent «hjerneinspirert», og da er enhver KI-brikke neuromorf. Det er en lite nyttig betydning.

I streng forstand betyr det maskinvare som etterligner noe konkret ved hvordan nerveceller arbeider. Tre ting: de kommuniserer med korte pulser i stedet for tall, en celle gjør bare noe når den får en puls, og minnet og regningen ligger på samme sted.

Resten av seksjonen bruker den strenge betydningen.



Figur G.12 To måter å organisere tid på. Klokkesignalet fra kapittel 6 slår uavbrutt, og hver eneste vending koster, også når ingenting skal regnes ut. I en hendelsesstyrt krets er linjene stille helt til en celle har noe å melde.

Figuren viser hvorfor ideen er fristende. Kapittel 6 på side 143 slo fast at effekten er omtrent kapasitans ganger spenning i annen ganger frekvens, og at frekvensen gjelder *hver vending som faktisk skjer*. En klokke som slår tre milliarder ganger i sekundet, betaler for hver av dem.

I et hendelsesstyrt system er linjene stille når ingenting skjer. I de fleste nevrale nett er de aller fleste verdiene null til enhver tid, og null er gratis å ikke sende.

De mest kjente systemene er forskningsmaskiner, og tre av dem er verdt å kjenne ved navn.

IBMs TrueNorth, fra 2014, har over en million kunstige nerveceller og 256 millioner forbindelser på én brikke, og bruker rundt 65 milliwatt på en typisk oppgave. Til sammenligning lå grafikkortet i seksjon G.3 på omkring tusen watt — TrueNorth bruker under en titusendel av det.

Intels Loihi, fra 2017, med en nyere utgave i 2021, legger vekt på at nervecellemodellen kan programmeres i stedet for å være støpt fast i kretsen.

SpiNNaker, fra Manchester, går motsatt vei: den bygger store systemer av helt vanlige prosessorkjerner, og lar det neuromorfe ligge i at kjernene bare sender pulser til hverandre.

«Uten klokke» er for sterkt

Du vil lese at neuromorfe brikker ikke har klokke. Det stemmer ikke helt for noen av de tre. Et ord først: en krets er *synkron* når alt i den skjer i takt med et felles klokkesignal, slik kapittel 6 på side 143 beskrev. Den er *asynkron* når delene i stedet venter på hverandre og handler når de får beskjed.

TrueNorth har asynkron logikk inne i seg, men et felles tikk-signal på rundt tusen ganger i sekundet som flytter tida framover for alle delene samtidig. Det er en global tidsreferanse, bare en svært langsom en. Loihi er helt vanlig synkron inne i hver kjerne, og asynkron bare mellom kjernene. SpiNNaker beskriver seg selv som globalt asynkron og lokalt synkron.
Den riktige formuleringen er derfor: *ingen global, taktgivende klokke slik en prosessor har*. Ikke «ingen klokke».

NPU-en i telefonen din er ikke neuromorf

Dette er den viktigste setningen i hele seksjonen, og den går ofte tapt fordi de to tingene omtales i samme avsnitt.
Kretsen i telefonen din er vanlig digital, synkron CMOS. Den har en klokke. Den regner med heltall i åtte bits. Den sender ikke pulser, den har ingen nervecellemodell, og den har ikke minnet og regningen på samme sted i den forstand denne seksjonen mener.
Den er en effektiv matrisemultiplikator, og det er alt den er. Det er ikke en nedvurdering — den er usedvanlig god til jobben sin.
Fellen er at «neuromorf» brukes i markedsføring om nær sagt hva som helst. Til og med en helt digital inferensbrikke fra IBM, som selskapet selv kaller digital, blir rutinemessig omtalt som neuromorf i pressen.

Hvor står det hele? Regnebrikkene er i all hovedsak forskningssystemer og tidlige utviklingsprøver. Markedet for dem er lite nok til å forsvinne i avrundingsfeilen ved siden av en milliard telefoner i året.

Men én ting har lyktes kommersielt, og den er verdt å nevne fordi den illustrerer prinsippet godt: hendelsesbaserte kameraer. I stedet for å ta tretti hele bilder i sekundet, melder hver piksel fra bare når lysstyrken der endrer seg. Står motivet stille, sendes nesten ingenting. Slike sensorer selges i dag som ordinære industrikameraer.

Det er den samme ideen som i figur G.12, og den virker.

G.6 Hva du kan ta med deg opp igjen

Fire ting er verdt å holde på.

Spesialisering er det andre svaret på varmeveggen. Kapittel 6 på side 143 fortalte om det første: da frekvensen stoppet, kom flere kjerner. Dette appendikset har fortalt om det andre. Når man heller ikke kommer lenger med flere generelle kjerner, bygger man kretser som ikke kan annet enn én ting — og som til gjengjeld gjør den tingen for en brøkdel av energien. Telefonen din gjør begge deler samtidig.

Det koster mer å hente et tall enn å regne med det. Over seks tusen ganger mer, hvis tallet ligger i arbeidsminnet. Dette ene forholdet forklarer alt det andre. Det forklarer hvorfor vektene får bli liggende mens dataene renner forbi. Det forklarer hvorfor noen vil gjøre regningen inne i minnet. Og det forklarer hvorfor telefonen regner lokalt i stedet for å sende bildet et sted. Det er dessuten hele begrunnelsen for minnehierarkiet i kapittel 7 på side 173.

Presisjon er en ressurs man kan bruke mindre av. Kapittel 2 på side 13 viste at et tall må representeres med et bestemt antall bits. Her har du sett hva de bitene koster: en multiplikator vokser omtrent med kvadratet av bitbredden. Derfor er «hvor få siffer slipper vi unna med» et av de viktigste spørsmålene i hele feltet — og derfor er svaret alltid «det kommer an på modellen».

Den digitale abstraksjonen er et valg, ikke en naturlov. Kryssfeltet i seksjon G.5 regner analogt, og det gir bort alt kapittel 6 på side 143 skrøt av: signalgjenoppretting, støymargin, en kopi som overlever å bli kopiert. I bytte får det en ganging gratis. Om den handelen lønner seg, er et åpent

spørsmål — men at det er en handel, og ikke en selvfølge, er verdt å ta med seg.

Begrep	Kort sagt
Vektet sum	Hver inngang ganges med sitt eget tall, og alt legges sammen.
Terskel	Grensen summen må over for at utgangen skal bli 1.
Perseptron	Vektet sum med terskel, der vektene kan endres.
Skjult lag	Et lag som verken er inngang eller utgang. Det som gjør XOR mulig.
MAC	én ganging pluss én addisjon. Arbeidsenheten i alt dette.
Inferens	Å bruke en ferdig modell. Det telefonen gjør.
Trening	Å finne vektene. Skjer i datasentre, og koster mye mer.
Kvantisering	Å skrive om vektene til færre bits før bruk.
Akselerator	Krets bygget for én oppgave i stedet for alle.
Systolisk matrise	Rutenett av regneceller der data flyter fra nabo til nabo.
Regning i minnet	Å utføre regnestykket der tallene allerede ligger.
Neuromorf	Maskinvare som etterligner nervecellens arbeidsmåte, med pulser og hendelser.

Tabell G.3 Begrepene fra dette appendikset.

Sju kapitler møtes her

Hvis du har lest boka i rekkefølge, er dette appendikset et knutepunkt, og det er verdt å se hvor trådene kommer fra.
Presisjonen og tallformatene er kapittel 2 på side 13. Hent-utfør-syklusen som koster mer enn regnestykket, er kapittel 3 på side 55. Flaskehalsen mellom prosessor og minne er kapittel 4 på side 79. Portene som den vektede summen bygges av, er kapittel 5 på side 105. Energi, klokke og varmevegg er kapittel 6 på side 143. At det koster å hente, er kapittel 7 på side 173. Og hvor det hele bærer hen, er kapittel 8 på side 201.
Ingen av delene er nye. Det nye er bare at de peker samme vei.

G.7 Videre lesning

Englander dekker ikke dette stoffet. Det sier seg selv — boka trekker abstraksjonsgrensa over transistoren, og dette appendikset ligger under den og ved siden av den samtidig.

Sze, Chen, Yang og Emer, «Efficient Processing of Deep Neural Networks: A Tutorial and Survey», *Proceedings of the IEEE*, 2017. Den beste enkeltkilden til alt i seksjon G.3 og G.4. Den er skrevet for folk med maskinvarebakgrunn, men de første tjue sidene krever ikke mer enn du har nå. De samme forfatterne har senere skrevet en hel bok med samme tittel.

Horowitz, «Computing's Energy Problem (and what we can do about it)», ISSCC 2014. Kilden til tallene i figur G.7. Det er en plenumsforelesning, ikke en artikkel, og den er kort og lesverdig.

Jouppi med flere, «In-Datacenter Performance Analysis of a Tensor Processing Unit», ISCA 2017. Beskrivelsen av Googles første akselerator, skrevet av dem som bygget den. Usedvanlig ærlig om hva som ikke fungerte. Fritt tilgjengelig.

Patterson og Hennessy, *Computer Organization and Design*, kapittel 6. Setter Googles akselerator inn i sammenhengen med resten av maskinen, og bruker den som gjennomgangseksempel. Vil du ha rammeverket rundt — hvorfor spesialiserte arkitekturer i det hele tatt vant fram — står det i kapittel 7 av de samme forfatternes *Computer Architecture: A Quantitative Approach*.

Sebastian, Le Gallo, Khaddam-Aljameh og Eleftheriou, «Memory devices and applications for in-memory computing», *Nature Nanotechnology*, 2020. Oversikten over alt i seksjon G.5, inkludert en nøktern gjennomgang av hvorfor det er vanskelig.

3Blue1Brown, serien om nevrale nettverk, på nett. Hvis det var *modellen* og ikke maskinvaren som vakte interessen din, er dette den beste visuelle innføringen som finnes. Fire korte videoer, og de forklarer også treningen dette appendikset hoppet over.

Videre i denne boka: kapittel 7 på side 173 tar dataflyttingen fra en annen kant og viser hva en vanlig prosessor gjør med problemet. Kapittel 8 på side 201 setter akseleratoren inn i den ferdige maskinen. Appendiks D har Ohms lov og fysikken bak kryssfeltet.

G.8 Oppgaver

Disse er ikke pensum. Svarskisser står i seksjon G.9, men flere av oppgavene har mer enn ett rimelig svar. Oppgave 6, 7, 10 og 14 krever penn og papir; resten kan tas i hodet.

1. Skriv en NOR-port med tre innganger som en vektet sum med terskel.
2. AND og OR har samme vekter og ulik terskel. Finnes det to porter i tabell G.1 som har samme *terskel*, men ulikt antall innganger?
3. Tegn de fire punktene for AND slik figur G.3 gjør for XOR, og tegn inn en linje som skiller dem.
4. Et lag har 200 innganger og 300 enheter. Hvor mange vekter har laget, og hvor mange MAC-operasjoner kreves for å kjøre det én gang?
5. En modell har 40 slike lag. Omtrent hvor mange MAC-operasjoner trengs per gjennomkjøring?
6. Bruk tallene i figur G.7. Hva koster det i energi å gjøre én million åttebits multiplikasjoner? Og hva koster det å hente én million tall fra arbeidsminnet?
7. Et regnestykke henter hver vekt fra arbeidsminnet rett før den brukes. Hvor stor andel av energien går til henting? Hva blir andelen hvis hver hentet vekt i stedet brukes hundre ganger?
8. Forklar med kjøkkenanalogien hvorfor det hjelper lite å gjøre multiplikatoren raskere hvis tallene kommer fra arbeidsminnet.
9. Hvorfor vokser en multiplikator omtrent med kvadratet av bitbredden, mens en adder vokser omtrent lineært? Svar med utgangspunkt i hvordan de to kretsene er bygget.
10. En brikke har plass til 2000 trettitobits multiplikatorer. Omtrent hvor mange åttebits multiplikatorer får plass på samme areal?
11. bfloat16 og FP16 har like mange bits. Hvilket av dem dekker det største tallområdet, og hvilket har flest gjeldende siffer? Hvorfor er det første viktigst under trening?
12. En medstudent sier: «Kvantisering til åtte bits er gratis, alle gjør det.» Hva er riktig i påstanden, og hva mangler?
13. Hvorfor er sprangprediksjon og cache til liten nytte når en prosessor kjører et nevralt nettverk?

14. En telefon tåler rundt 3,5 watt, et grafikkort i et datasenter rundt 1000. Hvis begge skal gjøre samme antall MAC-operasjoner i sekundet, hva må telefonen gjøre annerledes?
15. I en systolisk matrise hentes hvert inngangstall én gang og brukes av alle cellene i raden. Hvorfor er akkurat det den viktigste egenskapen?
16. Googles første akselerator hadde 65 536 regneceller og en klokke på 700 MHz — altså omtrent en fjerdedel av de 3 GHz en vanlig prosessor ligger på. Hvorfor var det et fornuftig valg?
17. Forklar hvorfor Ohms lov utfører en multiplikasjon i et kryssfelt, og hvorfor summeringen skjer av seg selv.
18. Et kryssfelt gir i praksis fire til seks bits som kan stoles på. Hva er det kryssfeltet gir avkall på, sammenlignet med en digital krets?
19. Hvorfor blir omformerne mellom analoge og digitale verdier et *større* problem for hver nye generasjon brikker, og ikke et mindre?
20. Hvorfor bruker en hendelsesstyrt krets mindre energi enn en klokkestyrt, selv når begge gjør like mye nyttig arbeid? Formelen for effekt fra kapittel 6 inneholder frekvensen — hva er det egentlig den frekvensen teller?
21. En telefonprodusent reklamerer med at brikken er «neuromorf». Hvilke tre spørsmål ville du stilt for å finne ut om det stemmer i streng forstand?
22. Nevn to grunner til å regne lokalt på telefonen som ikke handler om energi i det hele tatt.

G.9 Svarskisser

1. Vekter -1, -1, -1 og terskel ≥ 0. Er alle inngangene 0, blir summen 0 og utgangen 1. Er minst én inngang 1, blir summen negativ og utgangen 0.
2. Ja — NOT og NOR har begge vekten -1 og terskel ≥ 0, men NOT har én inngang og NOR har to. Poenget er at verken vektene eller terskelen alene bestemmer porten; antall innganger hører med.
3. Bare punktet $(1, 1)$ skal gi 1. En linje som går mellom $(1, 1)$ og de tre andre, skiller dem — for eksempel en linje fra omtrent (0,4; 1,2) til (1,2; 0,4). At det går an, er nettopp det som skiller AND fra XOR.
4. $200 \times 300 = 60\,000$ vekter, og like mange MAC-operasjoner.
5. Omtrent $40 \times 60\,000 = 2{,}4$ millioner. Ekte modeller er større, men størrelsesordenen millioner til milliarder er riktig.
6. En million multiplikasjoner: $10^6 \times 0{,}2$ pJ, altså 0,2 mikrojoule — en mikrojoule er en milliondels joule. En million hentinger: $10^6 \times 640$ pJ $= 640$ mikrojoule. Over tre tusen ganger mer. (Forholdet på over seks tusen i seksjon G.3 gjelder mot en *addisjon*, som er billigere enn en ganging. Begge tallene er riktige, de sammenligner bare ulike ting.)
7. Med én henting per ganging: 640 mot 0,2, altså rundt 99,97 prosent til henting. Brukes hver vekt hundre ganger, faller hentekostnaden per ganging til 6,4 pJ, og andelen til omtrent 97 prosent. Fortsatt høyt — men det er nettopp derfor gjenbruk er verdt så mye.
8. En skarpere kniv korter ned de tretti sekundene med hakking, ikke den halvtimen man bruker på å gå i butikken. Skal totaltiden ned, må man handle sjeldnere eller hente mer per tur.
9. Adderen behandler ett sifferpar om gangen og sender menten videre, slik fulladderne sto på rekke i kapittel 5: dobbelt så mange siffer gir dobbelt så mange trinn. Multiplikatoren må lage ett delprodukt for hvert *par* av siffer, og antall par firedobles når sifferantallet dobles.

10. Rundt tolv til seksten ganger så mange, altså i størrelsesorden 25 000 til 30 000. Både areal og energi vokser omtrent med kvadratet av bitbredden, men arealet vokser litt saktere enn energien, så svaret ligger i underkant av faktoren 15,5 fra figur G.8.

11. bfloat16 dekker det største området — både de største og de minste tallene — fordi den har åtte eksponentbits mot FP16s fem. FP16 har flest gjeldende siffer, med ti mantissebits mot sju. Under trening er området viktigst fordi svært små verdier som rundes helt til null, ødelegger læringen, mens litt avrunding bare bremser den.

12. Riktig: for mange modeller koster det lite, og gevinsten i energi og plass er stor. Mangler: at det forutsetter riktig skalering, at det kan slå helt feil ut for store språkmodeller, og at man gjerne akkumulerer i flere bits enn man ganger i.

13. Begge er kretser som gjetter. Sprangprediksjon gjetter hvilken vei en `if` går, men det finnes ingen `if`-er her. Cachen gjetter hvilke data som trengs snart, men her er rekkefølgen kjent på forhånd. De løser problemer oppgaven ikke har, og koster strøm likevel.

14. Telefonen må få mer ut av hver pikojoule. Konkret: færre bits per tall, mange små multiplikatorer i stedet for få raske, lav klokkefrekvens, og lokalt minne tett på regneenhetene så tallene slipper å reise.

15. Fordi henting er det dyreste som skjer. Brukes et hentet tall av alle cellene i raden, deles hentekostnaden på hele raden, og regnestykket forskyves fra transport til regning — som er den billige delen.

16. Effekten vokser lineært med frekvensen alene, men høyere fart krever som regel høyere spenning, og spenningen teller i annen. I eksempelet i kapittel 6 ble det nær tre ganger effekten for dobbelt så høy frekvens. Flere enheter i samme takt koster derimot bare lineært. Når oppgaven består av svært mange uavhengige ganginger, er det derfor langt billigere å regne bredt og langsomt enn smalt og fort.

17. Strømmen gjennom hvert element er spenningen ganger konduktansen, altså et produkt av inngangen og vekten. Når flere strømmer møtes i samme ledning, legger de seg sammen fordi ladning ikke forsvinner. Fysikken gjør begge operasjonene uten en eneste transistor.

18. Det gir avkall på signalgjenopprettingen og støymarginen. En digital port henter en fersk verdi fra forsyningen og kaster den slitne som kom inn, så feil hoper seg ikke opp. Et kryssfelt har ingen slik opprydding, så støy, celleforskjeller og drift går rett inn i svaret.

19. Fordi digital logikk krymper med hver generasjon — det er Moores lov fra kapittel 6 — mens analoge kretser i liten grad gjør det. Den digitale delen av regnestykket blir billigere år for år, og omformerne blir da en stadig større andel av både energien og arealet.

20. Effekten er omtrent kapasitans ganger spenning i annen ganger frekvens, og frekvensen teller de vendingene som faktisk skjer. En klokke betaler for hver eneste takt, også når det ikke er noe å gjøre. En hendelsesstyrt krets betaler bare når noe skjer — og i et nevralt nett er de aller fleste verdiene null til enhver tid.

21. Tre gode spørsmål: Sender den pulser i stedet for tall? Gjør den noe bare når den får en hendelse, eller går den på en global klokke? Ligger vektene der regningen skjer, eller hentes de fra et minne? Er svaret nei på alle tre, er det en vanlig akselerator med et pent navn.

22. For eksempel: dataene blir på telefonen i stedet for å sendes til en tjener, og det virker uten nettdekning. Svartiden er et tredje gyldig svar. Kostnaden for leverandøren teller derimot ikke her — den er i stor grad nettopp en strømregning.