IN1020 - “wanna-be” lærebok

Omid og Yngvar

Høsten 2025

sist oppdatert: 19. oktober 2025

Innhold

I Maskinvare 1

1 Hvordan håndtere kompleksitet? Abstraksjonsnivåers betydning 3

2 Hvordan oppnå høyere hastighet? Instruksjonsdekode, Pipeline og minnehierarki 9

3 Koder møter kretser 21

4 Kretser som driver koden 33

5 Bit'enes ballett 55

6 Transistorer 69

II Fasit for selv-test 77

F1 Abstraksjonsnivåer 79

III Appendix 83

A Synkronisering 85

Del I

Maskinvare

Kapittel 1

Hvordan håndtere kompleksitet? Abstraksjonsnivåers betydning

Innhold

1.1 Læringsmål 3
1.2 Hva er abstraksjonsnivåer? 4
1.3 Abstraksjon i datamaskiner og maskinvare 4
1.4 Eksempler på abstraksjon i praksis 5
1.5 Konklusjon 5
1.6 Videre lesning 6
1.7 Refleksjon 6
1.8 Selv-test 7

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

Hvordan oppnå høyere hastighet? Instruksjonsdekode, Pipeline og minnehierarki

Innhold

2.1 Læringsmål 10
2.2 Instruksjonsdekode 10
2.3 Pipeline 11
2.4 Minnehierarki 15

I dagens digitale verden er hastighet en nøkkelfaktor for effektiv databehandling. For å forstå hvordan vi kan øke hastigheten i datamaskiner, er det essensielt å forstå hvordan data prosesseres og viktigheten av minne og lagring. Det er derfor avgjørende å starte med en grundig forståelse av hvordan en prosessor gjennomfører en instruksjon, og hvordan en instruksjon styrer ulike deler av en CPU. Ettersom instruksjonene ofte henviser til et minneelement, er det også viktig å ha en forståelse for hvordan minne er designet og hvorfor man har et minnehierarki å forholde seg til.

Dette kapittelet tar for seg tre fundamentale konsepter innen datamaskinarkitektur som er avgjørende for å oppnå høyere hastighet: instruksjonsdekode, pipeline og minnehierarki.

Instruksjonsdekode er prosessen der maskinkodeinstruksjoner tolkes og utføres av prosessoren. Dette er det første steget i å forstå hvordan en datamaskin faktisk utfører oppgaver. Vi skal utforske hvordan prosessoren tolker og eksekverer instruksjoner, og hvordan dette påvirker ytelsen. Ved å forstå denne prosessen, kan vi bedre forstå hvordan ulike deler av CPUen aktiveres og samhandler under utførelsen av et program.

Pipeline-teknikken er en revolusjonerende metode innen prosessordesign som muliggjør parallell utførelse av flere instruksjoner samtidig. Dette konseptet bygger videre på vår forståelse av instruksjonsdekode, og viser hvordan moderne prosessorer kan øke effektiviteten ved å utføre flere steg av forskjellige instruksjoner parallelt. Vi skal se på hvordan denne teknikken betydelig kan øke gjennomstrømningen av instruksjoner og dermed oppnå høyere ytelse.

Minnehierarkiet er en strukturert tilnærming til organisering av datamaskinsystemer som utnytter prinsippet om lokalitet for å optimalisere tilgangen til data og instruksjoner. Dette konseptet er kritisk for å forstå hvorfor og hvordan datamaskiner håndterer lagring og henting av data og instruksjoner. Vi vil utforske de ulike nivåene i minnehierarkiet, fra hurtige men små cache-minner nær prosessoren

til større, men tregere hovedminne og sekundærlagring. Forståelsen av minnehierarkiet er essensiell for å kunne designe og optimalisere både maskinvare og programvare for best mulig ytelse.

Gjennom dette kapittelet vil vi dykke dypt inn i hvert av disse konseptene, utforske deres prinsipper, implementeringer og hvordan de samvirker for å øke datamaskinens ytelse. Vi vil også se på de utfordringene og begrensningene som følger med disse teknikkene, og diskutere fremtidige trender innen datamaskinarkitektur. Målet er å gi deg en solid forståelse av hvordan moderne datamaskiner oppnår sine imponerende hastigheter og hvordan du kan utnytte denne kunnskapen i din fremtidige karriere som informatiker.

2.1 Læringsmål

Etter å ha fullført dette kapittelet, skal du være i stand til å:

- Forklare prosessen for instruksjonsdekode og dens betydning for prosessorytelse.
- Beskrive konseptet med pipeline i prosessorarkitektur og hvordan det øker instruksjonsgjennomstrømningen.
- Analysere fordelene og utfordringene ved bruk av pipeline-teknikker i moderne prosessorer.
- Identifisere de ulike nivåene i et typisk minnehierarki og forklare deres roller.
- Diskutere hvordan minnehierarkiet påvirker systemets ytelse og kostnadseffektivitet.
- Vurdere ytelsesgevinster og potensielle flaskehalser i et datasystem basert på dets arkitektur.
- Anvende kunnskapen om disse konseptene til å optimalisere programvaredesign for bedre ytelse.

2.2 Instruksjonsdekode

Instruksjonsdekode er startpunktet for å forstå hvordan en datamaskin utfører oppgaver. Når et program kjøres, består det i virkeligheten bare av tall – binære instruksjoner som prosessoren henter én etter én. Men tallene i seg selv har ingen mening før prosessoren **dekoder** dem. Instruksjonsdekode er derfor det øyeblikket der datamaskinen forstår **hva den skal gjøre**. Det er prosessen der maskinkodeinstruksjoner tolkes og utføres av prosessoren. En typisk instruksjonssyklus består av flere trinn:

1. Instruction Fetch (IF): CPU-en henter neste instruksjon fra minnet.
2. Instruction Decode (DE): Den hentede instruksjonen dekodes for å bestemme hvilken operasjon som skal utføres og hvilke operander som er involvert.
3. Execute (EX): Instruksjonen utføres, som kan involvere aritmetiske operasjoner, logiske operasjoner, eller adresseberegninger.
4. Write Back (WB): Resultatet av operasjonen skrives tilbake til et register eller minne.

Denne prosessen gjentas for hver instruksjon i et program. I en sekvensiell prosessor, også kjent som en ikke-pipeline prosessor, fullføres hvert trinn i instruksjonssyklusen før neste instruksjon påbegynnes. Dette er en enkel tilnærming, men ikke særlig effektiv i forhold til utnyttelse av prosessorens ressurser. En spesiell type sekvensiell prosessor er en-syklus prosessoren, hvor alle trinn i instruksjonssyklusen utføres innenfor en enkelt klokkesyklus. Mens dette kan virke effektivt ved første øyekast, har det noen betydelige ulemper:

- Klokkesyklusen må være lang nok til å kunne inneholde den lengste mulige instruksjonssekvensen.
- Maskinvareressurser står ofte ubrukt siden ikke alle instruksjoner bruker alle komponenter.
- Det er vanskelig å oppnå høye klokkehastigheter, noe som begrenser den totale ytelsen.

For å analysere hvordan instruksjoner beveger seg gjennom prosessoren og hvordan forskjellige komponenter interagerer med hverandre, er det viktig å forstå datapath. Datapath refererer til den serien av komponenter i en prosessor som data flyter gjennom under utførelsen av en instruksjon, inkludert enheter som aritmetisk logisk enhet (ALU), instruksjonsregister, programteller og minne-grensesnitt.

Ved å analysere datapath, kan designere identifisere flaskehalser og muligheter for optimalisering. For eksempel kan effektiv instruksjonsdekoding redusere tiden som brukes på å tolke instruksjoner, smarte teknikker for operand-henting kan minimere ventetiden på data, optimalisering av utførelsesfasen kan akselerere beregninger, og effektiv minneaksess og write-back kan redusere forsinkelser relatert til dataoverføring.

Datapath refererer til den serien av komponenter i en prosessor som data flyter gjennom under utførelsen av en instruksjon. Den inkluderer enheter som:

- Aritmetisk logisk enhet (ALU)
- Instruksjonsregister
- Programteller
- Minne-grensesnitt

Instruksjonsdekode handler altså ikke om å øke hastigheten direkte, men om å bygge forståelsen for hvordan maskinen tenker. Når vi forstår dette trinnet, kan vi også forstå hvordan moderne prosessorer prøver å gjøre flere slike trinn samtidig. Forståelsen av instruksjonsdekode og de forskjellige trinnene i instruksjonssyklusen er kritisk for å forbedre prosessorytelsen, og leder oss til konseptet pipeline, som vi vil utforske i neste seksjon.

2.3 Pipeline

Pipeline er en revolusjonerende teknikk innen datamaskinarkitektur som har transformert prosessorytelse ved å muliggjøre parallell utførelse av flere instruksjoner. Konseptet med pipeline kan best forstås ved å sammenligne det med et samlebånd i en fabrikk, hvor flere produkter kan være i ulike stadier av produksjon samtidig.

I en tradisjonell sekvensiell prosessor, utføres hver instruksjon fullstendig før neste instruksjon påbegynnes. Dette kan sammenlignes med å ha én person som utfører alle trinn i en oppgave før han går videre til neste oppgave. Selv om dette er en enkel tilnærming, er det ikke særlig effektivt i forhold til utnyttelse av prosessorens ressurser.

Pipeline-teknikken, derimot, deler opp prosesseringen av hver instruksjon i flere trinn som kan utføres samtidig for forskjellige instruksjoner. Ved å dele opp utførelsen av instruksjoner på denne måten, kan en pipeline-prosessor oppnå en betydelig økning i gjennomstrømningen av instruksjoner og dermed høyere ytelse.

En pipeline betyr at prosessoren er delt opp i flere deloperasjoner som kan utføres samtidig, fordi hver del får sitt eget stykke maskinvare og sitt eget pipeline-register. Forskjellen mellom for eksempel en 2-trinns og en 4-trinns pipeline handler derfor ikke bare om antall steg i teorien – men om hvor mange

separate kretser og mellomlagre prosessoren faktisk har. En enkel analogi som illustrerer forskjell på en 2-trinns pipeline og en 4-trinns pipeline er:

Tenk deg to personer som skal lage pannekaker. Først ser vi for oss en 2-trinns pipeline:

1. Person 1 både blander røra og steker den.
2. Person 2 bare legger pannekakene på tallerken.

De jobber parallelt, men store deler av jobben er fortsatt i samme hender.

Så er ser vi på en 4-trinns pipeline:

1. Person 1 henter ingredienser.
2. Person 2 blander røra.
3. Person 3 steker pannekaker.
4. Person 4 legger dem på tallerkenen.

Nå kan du få én ferdig pannekake per minutt, selv om det tar fire minutter å lage den første.

En annen nyttig analogi for å forstå en 4-trinns pipeline er prosessen med å vaske klær. La oss se på fire trinn i klesvasken:

1. Laste inn: Putte klær i vaskemaskinen
2. Vaske: Selve vaskeprosessen
3. Tørke: Tørke klærne i tørketrommelen
4. Brette og legge bort: Brette klærne og legge dem i skapet

I en sekvensiell prosess ville vi fullføre alle fire trinn for én last med klær før vi begynte på neste last. Men med en pipeline-tilnærming kan vi ha fire laster med klær i ulike stadier samtidig:

- Mens den første lasten brettes og legges bort,
- er den andre lasten i tørketrommelen,
- den tredje lasten vaskes,
- og vi kan laste inn en fjerde last i vaskemaskinen.

Dette øker effektiviteten betydelig, da vi kan fullføre en last med klær mye oftere enn hvis vi måtte vente på at alle trinn ble fullført for hver last.

2.3.1 Pipeline-trinn og synkronisering

En grunnleggende 4-trinns pipeline i en prosessor består av følgende trinn:

1. Instruction Fetch (IF): Henter neste instruksjon fra minnet.

2. Instruction Decode (DE): Dekoder instruksjonen for å bestemme hvilken operasjon som skal utføres og hvilke operander som er involvert.

3. Execute (EX): Utfører den spesifiserte operasjonen.

4. Write Back (WB): Skriver resultatet tilbake til registre eller minne.

Valget av en 4-trinns pipeline er en balanse mellom kompleksitet og ytelse[1]. Disse fire trinnene representerer de grunnleggende operasjonene en prosessor må utføre for hver instruksjon, og gir en god balanse mellom økt gjennomstrømning og kompleksitet i design.

Synkronisering er en kritisk aspekt ved pipeline-design. Hvert trinn i pipeline må fullføres innen én klokkesyklus, og overføringen av data mellom trinnene må være nøye timet. Dette krever sofistikerte klokkesignaler og presis kontroll av datamengden som flyter gjennom hvert trinn.

For å illustrere hvordan en 4-trinns pipeline fungerer, kan vi se på et tidslinjediagram. La oss se på hvordan fem instruksjoner (I1 til I5) ville blitt utført:

```
Syklus:  1   2   3   4   5   6   7   8
I1:     IF  DE  EX  WB
I2:         IF  DE  EX  WB
I3:             IF  DE  EX  WB
I4:                 IF  DE  EX  WB
I5:                     IF  DE  EX  WB
```

Som vi ser, starter en ny instruksjon i hver klokkesyklus, og etter den fjerde syklusen fullfører vi én instruksjon per syklus. Dette illustrerer den betydelige økningen i gjennomstrømning som pipeline-teknikken muliggjør.

2.3.2 Fordeler og utfordringer med pipeline

Pipeline-teknikken bringer med seg en rekke betydelige fordeler for prosessorytelse:

- **Økt gjennomstrømning**: Ved å ha flere instruksjoner under utførelse samtidig, kan en pipeline-prosessor oppnå en betydelig økning i gjennomstrømningen av instruksjoner sammenlignet med en sekvensiell prosessor.
- **Bedre utnyttelse av maskinvareressurser**: I en pipeline-prosessor er hver del av prosessoren aktiv i hver syklus, noe som fører til en bedre utnyttelse av tilgjengelige ressurser.
- **Mulighet for høyere klokkehastighet**: Siden hvert trinn i en pipeline utfører mindre arbeid enn en fullstendig instruksjonsutførelse, kan syklustiden reduseres, noe som muliggjør høyere klokkehastigheter.

Til tross for disse fordelene, kommer pipeline-teknikken også med noen utfordringer som må håndteres:

- **Datahazards**: Disse oppstår når en instruksjon er avhengig av resultatet fra en tidligere instruksjon som ennå ikke er fullført. Datahazards kan føre til forsinkelser i pipeline mens den venter på at den nødvendige dataen skal bli tilgjengelig.

[1] En 4-trinns pipeline er også et godt valg for å pedagogisk forklare prinsippet.

- **Kontrollhazards**: Forgreningsinstruksjoner, som kan endre programflyten basert på en betingelse, kan forårsake kontrollhazards. Hvis en forgrening tas, kan instruksjoner som allerede er i pipeline bli ugyldige, noe som krever at pipeline tømmes og fylles på nytt.
- **Strukturelle hazards**: Disse oppstår når to instruksjoner trenger samme maskinvareressurs samtidig, for eksempel hvis to instruksjoner trenger tilgang til minnet i samme syklus. Strukturelle hazards kan føre til forsinkelser mens den ene instruksjonen venter på at ressursen skal bli tilgjengelig.

For å håndtere disse hazardene, kan pipeline-prosessorer implementere teknikker som forwarding (også kjent som data bypassing), spekulativ utførelse og out-of-order utførelse. Disse teknikkene bidrar til å minimere virkningen av hazards og opprettholde høy ytelse.

2.3.3 Avanserte pipeline-teknikker

Moderne prosessorer har ofte mer komplekse pipeline-strukturer enn den grunnleggende 4-trinns modellen. Disse avanserte teknikkene tar sikte på å ytterligere øke ytelsen og effektiviteten til pipelineprosessering.

En vanlig teknikk er å øke **dybden på pipeline** ved å dele opp trinnene i mindre deler. Mange moderne CPUer har pipelines med 10-20 eller flere trinn, noen har også nested-pipeline som betyr at de har pipeline i en eller flere trinn i (hoved-)pipelinen. Dette muliggjør høyere klokkehastigheter, siden hvert trinn utfører enda mindre arbeid. Imidlertid kan dypere pipelines også øke kompleksiteten og potensielt forverre virkningen av hazards.

Pipeline-design basert på CPU-formål kan variere betydelig avhengig av prosessorens tiltenkte bruk:

- *Høyytelsesprosessorer*: Disse har ofte dypere pipelines og mer aggressive spekulasjonsteknikker.
- *Laveffektprosessorer*: Disse kan ha kortere pipelines for å redusere strømforbruket.
- *Sanntidssystemer*: Disse kan ha enklere, mer forutsigbare pipeline-strukturer.
- *VLIW*[2] *arkitekturer*: Disse legger mer av ansvaret for instruksjonsparallelisme på kompilatoren.

Speed-up og Amdahl's lov - Konseptet "speed-up" er kritisk for å forstå effektiviteten av pipeline og andre parallelliseringsteknikker. Speed-up måler hvor mye raskere en oppgave kan utføres med en forbedret teknikk sammenlignet med en basislinje. Amdahl's lov beskriver den potensielle speed-up av et system når bare en del av systemet forbedres. Den sier at den totale speed-up er begrenset av den delen av systemet som ikke kan forbedres.

Superskalær arkitektur er en annen viktig teknikk i moderne prosessorer. I en superskalær prosessor kan flere instruksjoner startes i hver klokkesyklus, noe som øker graden av parallellitet. Dette krever duplikasjon av noen pipeline-ressurser, som ALUer og dekodere, men kan føre til betydelige ytelsesgevinster.

Out-of-order utførelse og spekulativ utførelse er også vanlige teknikker i moderne pipeline-prosessorer. Out-of-order utførelse tillater instruksjoner å utføres i en annen rekkefølge enn de opprinnelig ble skrevet i programmet, så lenge dataavhengighetene overholdes. Dette kan bidra til å skjule forsinkelser forårsaket av hazards. Spekulativ utførelse involverer å gjette utfallet av forgreninger og begynne å

[2]VLIW = Very long instruction word

utføre instruksjoner basert på gjetningen. Hvis gjetningen viser seg å være feil, må resultatene forkastes og pipeline tømmes, men hvis gjetningen er riktig, kan betydelige ytelsesgevinster oppnås.

Simultaneous Multithreading (SMT) er en teknikk som lar flere tråder dele ressursene i en enkelt prosessorkjerne. Dette kan bidra til å øke utnyttelsen av pipeline-ressurser ved å tillate instruksjoner fra forskjellige tråder å fylle ledige plasser i pipeline.

2.3.4 Selv-test oppgaver Pipeline

1. Tenk på en bilvask som en pipeline. Hvilke trinn ville du inkludere, og hvordan ville dette ligne på en prosessor-pipeline?
2. Forestill deg et restaurantkjøkken som en form for pipeline. Hvordan kan ulike stasjoner for matlaging representere trinnene i en prosessor-pipeline?
3. Hvordan kan en flyplassikkerhetskontroll ses på som en pipeline? Hvilke trinn ville være involvert, og hvordan relaterer dette til databehandling?

2.4 Minnehierarki

Minnehierarkiet er en strukturert tilnærming til organisering av datamaskinsystemer som utnytter prinsippet om lokalitet for å optimalisere tilgangen til data og instruksjoner. Det er en kritisk komponent i moderne datamaskinarkitektur, og forståelsen av minnehierarkiet er essensiell for å kunne designe og optimalisere både maskinvare og programvare for best mulig ytelse.

2.4.1 Oversikt over minneelementer

Et typisk minnehierarki består av flere nivåer, hver med sine egne egenskaper når det gjelder hastighet, pris og størrelse. Disse nivåene kan visualiseres som en hierarkisk trekant, med de raskeste, dyreste og minste minneelementene på toppen, og de tregere, billigere og større minneelementene nederst.

- **Registre**: Dette er de raskeste, men også de minste minneelementene, som er direkte integrert i prosessoren. De brukes til å lagre midlertidige data og mellomresultater under utførelsen av instruksjoner. Registre har typisk en aksess-tid på under 1 nanosekund og en størrelse på noen hundre bytes.
- **Cache**: Cache er et raskt minneelement som fungerer som en buffer mellom prosessoren og hovedminnet. Det er delt inn i flere nivåer (L1, L2, L3, etc.), hvor hvert nivå er større, men tregere enn det forrige. Cachens rolle er å lagre ofte brukte data og instruksjoner, slik at prosessoren kan aksessere dem raskt uten å måtte gå til hovedminnet. L1 cache har typisk en aksess-tid på 1-5 nanosekunder og en størrelse på noen titalls kilobyte, mens L2 og L3 cache kan ha aksess-tider på 5-20 nanosekunder og størrelser på hundrevis av kilobyte til noen megabyte.
- **Hovedminne (RAM)**: Random Access Memory (RAM) er hvor aktive programmer og data lagres under utførelse. Det er betydelig større enn cache, men også tregere og billigere per byte. RAM har typisk en aksess-tid på 50-100 nanosekunder og størrelser fra noen gigabyte til titalls gigabyte i moderne systemer.
- **Sekundærlagring**: Dette er det nederste nivået i minnehierarkiet og inkluderer lagringsenheter som harddisker, SSD-er og optiske disker. Sekundærlagring har stor kapasitet, men er også de tregeste minneelementene. De brukes til langtidslagring av programmer og data når de ikke

er i aktiv bruk. Aksess-tider for sekundærlagring kan være i størrelsesorden millisekunder til sekunder, mens kapasiteten kan være fra hundrevis av gigabyte til flere terabyte.

2.4.2 Primær- og sekundærminne

Minneelementene i datamaskinarkitekturen kan videre deles inn i primærminne og sekundærminne. Primærminne refererer til minne som er direkte tilgjengelig for prosessoren, som registre, cache og RAM. Disse minneelementene er karakterisert ved rask aksess-tid, men begrenset kapasitet og høyere kostnad per byte.

Sekundærminne, på den annen side, refererer til lagringsenheter som harddisker, SSD-er og optiske disker. Disse enhetene har mye større kapasitet enn primærminne, men også betydelig tregere aksess-tider. Sekundærminne brukes hovedsakelig til langtidslagring av data og programmer når de ikke er i aktiv bruk.

En annen viktig distinksjon er mellom volatile (flyktig) og non-volatile (ikke-flyktig) minneelementer. Volatile minne, som SRAM og DRAM (brukt i cache og RAM), mister innholdet når strømmen slås av. Non-volatile minne, som flash-minne (brukt i SSD-er) og magnetiske disker (brukt i harddisker), beholder innholdet selv uten strøm.

2.4.3 Analogi for minnehierarkiet

En god analogi for å forstå minnehierarkiet er å tenke på det som en bibliotekstruktur. Registrene kan sammenlignes med et lite skrivebord hvor du har de bøkene og notatene du jobber med akkurat nå. Cache er som en bokhylle rett ved skrivebordet ditt, hvor du har de bøkene du ofte referer til. Hovedminnet er som bibliotekets hovedsamling - det inneholder alle bøkene du kan trenge, men det tar litt tid å hente dem. Sekundærlagring er som et fjernlager utenfor biblioteket, hvor bøker som sjelden brukes, blir oppbevart.

2.4.4 Eksempel på minnebruk i et program

For å illustrere hvordan de forskjellige minneelementene brukes under utførelsen av et program, la oss se på et forenklet eksempel med en for-løkke i Python (forutsatt at vi har veldig liten minne):

```
sum = 0
for i in range(100):
    sum += i
```

I dette eksemplet vil variablene 'sum' og 'i', samt selve for-løkken, sannsynligvis være lagret i cache, siden de aksesseres ofte. Hele programmet vil være lagret i hovedminnet (RAM).

2.4.5 Datamaskinarkitektur og minnehierarki

Minnehierarkiet er tett knyttet til datamaskinarkitekturen, spesielt i forhold til prosessoren (CPU). I den klassiske Von Neumann-arkitekturen er prosessoren koblet til minnet gjennom en minnebuss, og instruksjoner og data hentes fra minnet etter behov.

Minnebussen er en kritisk komponent i datamaskinarkitekturen, da den muliggjør kommunikasjon mellom prosessoren og minneelementene. Bussens bredde (antall bits som kan overføres samtidig) og hastighet har en direkte innvirkning på systemets ytelse. Moderne minnebusser, som DDR (Double Data

Rate) og HBM (High Bandwidth Memory), er designet for å maksimere dataoverføringshastigheten og minimere forsinkelser.

I moderne systemer med flerkjerne prosessorer har hver kjerne ofte sin egen L1 og L2 cache, mens L3 cache (og høyere nivåer, hvis de er tilstede) deles mellom kjernene. Denne delingen av cache introduserer behovet for cache-koherens, som sikrer at data i cache forblir konsistent mellom kjernene.

2.4.6 Cache-nivåer og koherens

Cache-nivåene (L1, L2, L3, etc.) har forskjellige egenskaper og bruksområder:

- **L1 cache**: Dette er det raskeste og minste cache-nivået, ofte delt inn i separate instruksjons- og data-cache. L1 cache er typisk på noen titalls kilobyte og har en aksess-tid på bare noen få klokkesykler.
- **L2 cache**: L2 cache er større og litt tregere enn L1, typisk på noen hundre kilobyte til noen megabyte. Det fungerer som en buffer mellom L1 cache og høyere nivåer av cache eller hovedminnet.
- **L3 cache og høyere nivåer**: Disse cache-nivåene er enda større og tregere enn L2, og deles ofte mellom flere prosessorkjerner. De fungerer som en siste buffer før hovedminnet og kan være på flere titalls megabyte.

Cache-koherens er en viktig mekanisme i systemer med delt cache. Det sikrer at hvis en kjerne oppdaterer en cache-linje, blir denne oppdateringen reflektert i cachene til de andre kjernene. Dette oppnås gjennom protokoller som MESI (Modified, Exclusive, Shared, Invalid), som holder styr på tilstanden til hver cache-linje.

2.4.7 Cache-hit, cache-miss, RAM-hit og RAM-miss

Når prosessoren trenger å lese data eller instruksjoner, sjekker den først i cache. Hvis den nødvendige dataen finnes i cache, kalles det en cache-hit, og dataen kan leses raskt. Hvis dataen ikke finnes i cache, kalles det en cache-miss, og prosessoren må gå til neste nivå i minnehierarkiet (enten et høyere cache-nivå eller hovedminnet) for å hente dataen.

På samme måte, når prosessoren går til hovedminnet (RAM), kan det være en RAM-hit hvis dataen finnes i RAM, eller en RAM-miss hvis dataen må hentes fra sekundærlagring.

Cache-misser og RAM-misser kan ha en betydelig innvirkning på ytelsen, siden det tar mye lengre tid å hente data fra høyere nivåer i minnehierarkiet.

2.4.8 Lese- og skriveoperasjoner i minnet

En viktig egenskap ved minneelementer er at det ofte tar lengre tid å skrive data enn å lese data. Dette skyldes flere faktorer:

- **Skriveoperasjoner krever mer strøm**: For å endre tilstanden til en minnecelle kreves det mer strøm enn bare å lese tilstanden. Dette gjelder spesielt for volatile minneteknologier som SRAM og DRAM, hvor minnecellene må kontinuerlig oppdateres for å beholde innholdet.

- **Skriveoperasjoner kan kreve ekstra steg**: I noen minneteknologier, som NAND flash, krever skriveoperasjoner at minnecellen først slettes før den kan skrives til. Dette introduserer ekstra forsinkelse sammenlignet med leseoperasjoner.

- **Cache-koherens**: I systemer med delt cache kan en skriveoperasjon kreve at cache-linjen oppdateres i cachene til flere prosessorkjerner, noe som introduserer ekstra forsinkelse og kommunikasjon over minnebussen.

Disse faktorene bidrar til at skriveoperasjoner ofte har høyere latens og energiforbruk enn leseoperasjoner, noe som kan påvirke ytelsen og strømforbruket til programmer som gjør mye skriving til minnet.

2.4.9 Dataoverføring og minneadressering

Når data overføres mellom prosessoren og minnet, skjer dette i enheter kalt ord (word). Ordlengden er antall bits som kan overføres samtidig over minnebussen, og er typisk 32 eller 64 bits i moderne systemer. Ordlengden påvirker både hastigheten og effektiviteten til dataoverføringen.

Minnestørrelsen refererer til den totale mengden data som kan lagres i et minneelement, og er typisk målt i bytes (8 bits). For eksempel kan et system ha 8 GB RAM, noe som betyr at hovedminnet kan lagre 8 milliarder bytes med data.

Adresserommet er det totale antallet unike adresser som kan brukes til å aksessere minnet. Hver minnecelle har en unik adresse, og prosessoren bruker disse adressene til å lese fra og skrive til spesifikke lokasjoner i minnet. Adresserommet er avhengig av antall adresselinjer i systemet, og er typisk 32 eller 64 bits i moderne prosessorer, noe som gir et adresserom på henholdsvis 4 GB eller 16 EB (eksabyte).

2.4.10 Videre lesing

For de mer avanserte og interesserte kan det være nyttig å dykke dypere inn i detaljene rundt minnehierarkiet og minneteknologier.

En viktig distinksjon i RAM-teknologi er mellom **SRAM (Static RAM) og DRAM (Dynamic RAM)**. SRAM bruker en mer kompleks minnecelle med flere transistorer, noe som gjør den raskere, men også dyrere og mer plasskrevende enn DRAM. DRAM bruker en enklere minnecelle med færre transistorer og en kondensator for å lagre data, noe som gjør den billigere og mer kompakt, men også tregere og mer energikrevende (på grunn av behovet for periodisk oppdatering) enn SRAM. I moderne systemer brukes SRAM typisk for cache, mens DRAM brukes for hovedminne.

En annen interessant utvikling er bruken av ikke-volatile minneteknologier, som **PCM (Phase-Change Memory) og MRAM (Magnetoresistive RAM)**, som kombinerer fordelene til både DRAM (høy tetthet, lav kostnad) og flash (ikke-volatilitet, høy utholdenhet). Disse teknologiene kan potensielt brukes til å lage "universal minne" som kan erstatte både DRAM og flash i fremtidige systemer.

Videre kan det være interessant å se på mer avanserte **cachekoherens-protokoller**, som MOESI og MESIF, som legger til ekstra tilstander for å optimalisere ytelsen i systemer med mange kjerner og delt minne.

Til slutt kan det være nyttig å utforske minnehierarkiet fra et programvareperspektiv, med teknikker som **cachebevisst programmering, minneallokering og garbage collection**, som kan ha en betydelig innvirkning på ytelsen til programmer som arbeider med store mengder data.

2.4.11 Selv-test oppgaver - Minne

1. Forklar forskjellen mellom primærminne og sekundærminne, og gi eksempler på hver.
2. Hva er forskjellen mellom en cache-hit og en cache-miss? Hvordan påvirker cache-misser ytelsen til et system?
3. Forklar begrepet cachekoherens.
4. Hvorfor tar skriveoperasjoner i minnet ofte lengre tid enn leseoperasjoner? Nevn minst to faktorer som bidrar til denne forskjellen.
5. Beskriv forholdet mellom ordlengde, minnestørrelse og adresserom i et datamaskinsystem. Hvordan påvirker disse parameterne ytelsen og kapasiteten til systemet?
6. Tenk deg et program som utfører mange skriveoperasjoner til minnet. Hvordan kan du optimalisere programmets ytelse og energiforbruk, gitt det du vet om minnehierarkiet og egenskapene til skriveoperasjoner?
7. Forklar begrepet "ikke-volatile minneteknologier" og gi eksempler på slike teknologier. Hvordan kan disse teknologiene potensielt endre minnehierarkiet i fremtidige systemer?

2.4.12 Videre lesing

For de som ønsker å fordype seg videre i pipeline-teknikker og avansert prosessorarkitektur, anbefales følgende ressurser:

- "Computer Architecture: A Quantitative Approach" av John L. Hennessy og David A. Patterson, som gir en omfattende behandling av moderne prosessordesign og ytelsesanalyse.
- "Modern Processor Design: Fundamentals of Superscalar Processors" av John Paul Shen og Mikko H. Lipasti, som fokuserer på design og implementering av superskalære prosessorer.
- IEEE Micro og ACM Transactions on Computer Systems, tidsskrifter som ofte publiserer cutting-edge forskning innen prosessorarkitektur.
- Tekniske "white papers" fra prosessorprodusenter som Intel, AMD og ARM, som gir innsikt i de nyeste pipeline-teknikkene og arkitekturene som brukes i kommersielle prosessorer.

Kapittel 3

Koder møter kretser

Innhold

3.1 Læringsmål 22
3.2 Datamaskinarkitektur 22
3.3 BUSarkitektur: Datamaskinens motorveisystem 24
3.4 Cache og RAM: Balansegangen mellom hastighet og kapasitet 25
3.5 Moderne prosessorarkitektur: Mer enn bare flere kjerner 26
3.6 Selv-test oppgaver 26
3.7 Fasit til selv-test oppgaver 26
3.8 Refleksjoner over trade-offs i maskinvaredesign 27
3.9 Refleksjonsspørsmål 28
3.10 Videre lesing 31

I (nesten) enhver datamaskin ligger en fascinerende verden der abstrakte koder møter fysiske kretser. Dette kapittelet[1] tar deg med på en reise gjennom datamaskinarkitekturens evolusjon og nåværende tilstand, med et spesielt fokus på forståelsen av maskinvare på dypet. Vi skal utforske hvordan moderne datamaskiner er bygget opp, fra de minste byggesteinene til komplekse systemer som håndterer milliarder av operasjoner hvert sekund.

Datamaskinarkitektur har gjennomgått en stor utvikling siden de første elektroniske datamaskinene ble konstruert på 1940-tallet. Fra rørbaserte giganter som fylte hele rom, har vi nå kraftige datamaskiner som passer i lommen. Denne utviklingen har vært drevet av stadig mer sofistikerte komponenter og designprinsipper.

La oss begynne med hjernen i enhver datamaskin: prosessoren. Moderne prosessorer er teknologiske underverk, kapable til å utføre milliarder av instruksjoner per sekund. De er bygget opp av transistorer[2] - bittesmå elektroniske brytere som kan være enten på eller av, representerende de binære tilstandene 1 og 0. En moderne prosessor kan inneholde milliarder av slike transistorer, noe som gir dem enorm regnekraft. Mer om dette i kapitellene senere.

Prosessoren arbeider tett sammen med flere typer minne, hver med sin spesielle rolle i systemet. Registre er de raskeste minneenhetene, plassert direkte i prosessoren. De brukes til å lagre data som prosessoren trenger umiddelbar tilgang til. Cache-minne fungerer som en bro mellom de lynraske

[1] Det er mye overlapp mellom dette kapittelet og den forrige, men vi har valgt å ha det slik nå da det passer bedre med forelesningene i høst 2024

[2] vi kommer til å snakke mer om transistorer senere i kurset.

registrene og det tregere hovedminnet (RAM). Det finnes flere nivåer av cache, vanligvis betegnet som L1, L2 og L3, hvor L1 er det raskeste og minste, mens L3 er det største og tregeste.

RAM (Random Access Memory) er datamaskinens arbeidsminne, hvor aktive programmer og data lagres mens maskinen er i bruk. I motsetning til registre og cache, som er bygget inn i prosessoren, er RAM separate chips på hovedkortet. RAM er mye tregere enn cache, men har betydelig større kapasitet.

For å binde sammen disse ulike komponentene brukes ulike typer BUS´er. En BUS kan sammenlignes med en motorvei for data, hvor informasjon flyter mellom ulike deler av systemet. Systembus er hovedveien som knytter sammen prosessor, minne og andre sentrale komponenter. I/O-BUSer håndterer kommunikasjon med eksterne enheter som harddisker, skjermkort og nettverkskort.

3.1 Læringsmål

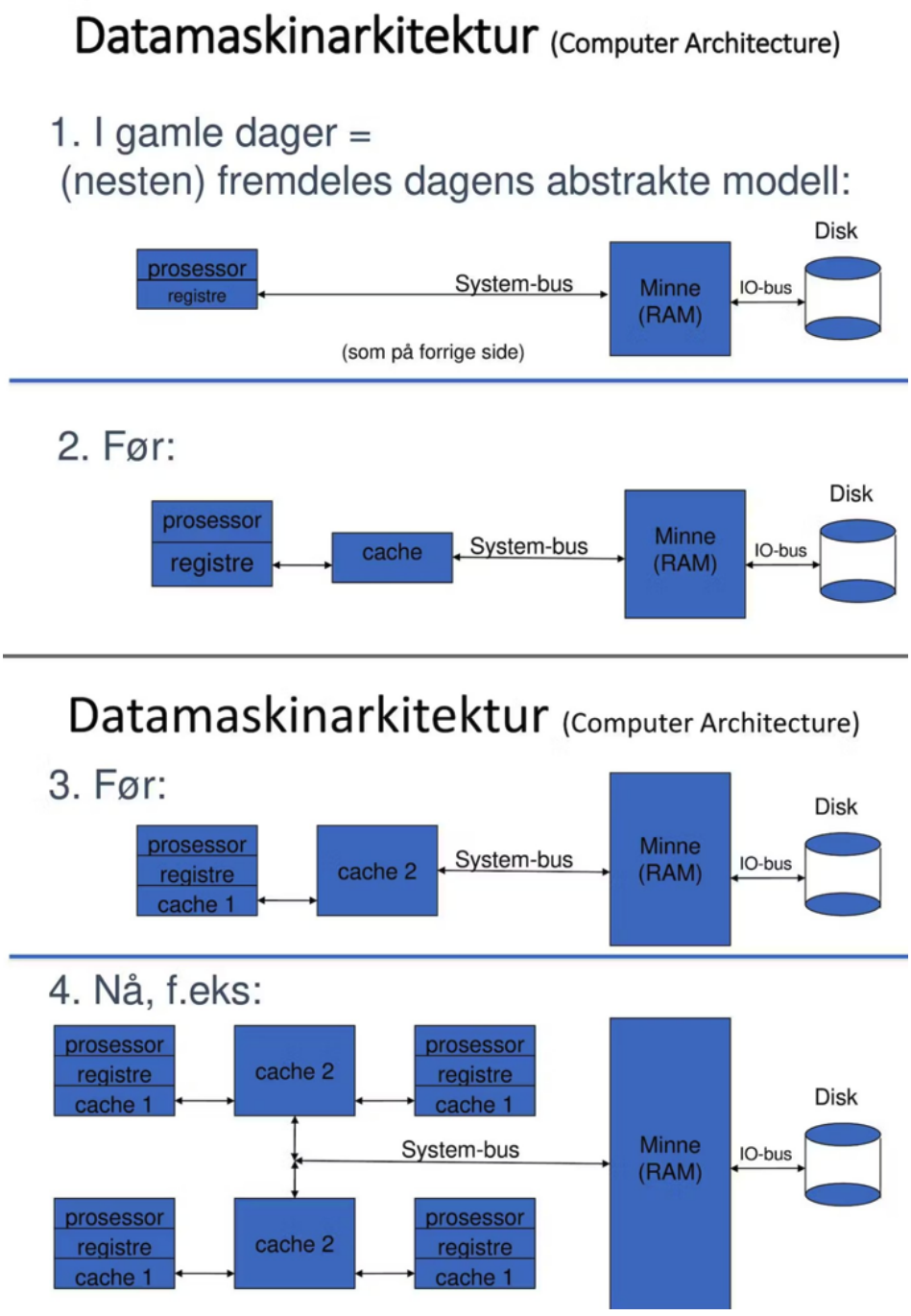
Etter å ha fullført dette kapittelet skal du kunne:

- Kunnskap:
  - Beskrive hovedkomponentene i en moderne datamaskinarkitektur
  - Gjenkjenne og forklare ulike typer busser og deres roller i en datamaskin
  - Redegjøre for betydningen av cache i moderne prosessorarkitektur
  - Kjenne til forskjeller mellom lese- og skriveoperasjoner i minne
- Ferdigheter:
  - Identifisere hvordan cache-hierarkiet påvirker systemets ytelse
  - Vurdere fordeler og ulemper ved ulike arkitekturvalg
- Generell kompetanse:
  - Evaluere trade-offs i maskinvaredesign basert på ulike krav og begrensninger
  - Reflektere over hvordan arkitekturvalg påvirker systemets helhetlige ytelse og effektivitet
- Refleksjon og analyse:
  - Kritisk vurdere konsekvensene av ulike designvalg i maskinvarearkitektur
  - Forstå hvordan teknologiske fremskritt påvirker mulighetene og begrensningene i maskinvaredesign
  - Reflektere over balansen mellom ytelse, energieffektivitet og kompleksitet

3.2 Datamaskinarkitektur

Datamaskinarkitektur har gjennomgått en evolusjon siden de første elektroniske datamaskinene ble konstruert. Figur 3.1 on the facing page illustrerer denne utviklingen gjennom fire stadier, fra en grunnleggende modell til dagens komplekse, flerkjernede systemer.

1. Grunnleggende modell: Dette representerer den tidligste og fortsatt grunnleggende abstrakte modellen for datamaskinarkitektur, ofte referert til som von Neumann-arkitekturen. Her ser vi en enkel prosessor med registre koblet direkte til minnet (RAM) via en system-bus. Disken er koblet til via en I/O-bus.



Figur 3.1: Datamaskinarkitektur - historisk utvikling

Teorien bak denne modellen er fundamental: prosessoren henter instruksjoner og data fra minnet, utfører operasjoner, og lagrer resultater tilbake i minnet. Dette konseptet med lagret program revolusjonerte databehandling ved å tillate at både instruksjoner og data kunne lagres i samme minnestruktur.

2. Introduksjon av cache: Det neste steget i utviklingen var introduksjonen av cache-minne. Cache

ble lagt til mellom prosessoren og hovedminnet for å redusere den gjennomsnittlige tiden det tar å aksessere data.

Dette bygger på lokalitetsprinsippet, som består av to nøkkelkonsepter: - Temporal lokalitet: Hvis en dataelement aksesseres, er det sannsynlig at det vil bli aksessert igjen snart. - Spatial lokalitet: Hvis et dataelement aksesseres, er det sannsynlig at nærliggende dataelementer også vil bli aksessert.

Cache utnytter disse prinsippene for å holde ofte brukte data og instruksjoner nærmere prosessoren, noe som drastisk reduserer gjennomsnittlig aksesseringstid.

3. Flernivås cache: Etter hvert som prosessorene ble raskere, ble behovet for enda mer effektiv minnetilgang tydelig. Dette førte til utviklingen av flernivås cache-systemer.

Denne modellen introduserer konseptet med minnehierarki, hvor mindre, raskere minneenheter supplerer større, tregere enheter. L1-cache, integrert direkte i prosessoren, er ekstremt rask men liten. L2-cache er større men litt tregere. Dette hierarkiet balanserer hastighet og kapasitet, og utnytter lokalitetsprinsippene enda mer effektivt.

4. Moderne flerkjernearkitektur: Den siste modellen viser en typisk moderne datamaskinarkitektur med flere prosessorkjerner.

Denne arkitekturen introduserer parallellisme på hardware-nivå. Flerkjerneprosessorer kan utføre flere instruksjoner samtidig, noe som drastisk øker ytelsen for mange typer applikasjoner. Dette reiser imidlertid nye utfordringer, særlig innen minnekoherens - å sikre at alle kjerner har en konsistent visning av data i sine respektive cacher.

Moderne arkitekturer implementerer sofistikerte koherens-protokoller som MESI (Modified, Exclusive, Shared, Invalid) for å håndtere dette. Disse protokollene sikrer at endringer i data reflekteres konsistent på tvers av alle cacher og hovedminnet.

Gjennom hele denne utviklingen har et sentralt tema vært å overvinne den såkalte 'von Neumann-flaskehalsen' - begrensningen i dataoverføringshastighet mellom prosessor og minne. Hver fase i arkitekturevolusjonen har adressert dette problemet på ulike måter:

- Cache-minne reduserer behovet for hyppige minneaksesser. - Flernivås cache optimaliserer ytterligere for ulike aksessmønstre og datamengder. - Flerkjernearkitekturer muliggjør parallell databehandling, effektivt multipliserende systemets evne til å overkomme minnebegrensninger.

Moderne datamaskinarkitektur fortsetter å evolere, med fokus på å balansere regnekraft, energieffektivitet og evnen til å håndtere stadig mer komplekse og dataintensive oppgaver. Nye trender inkluderer heterogene systemer som kombinerer CPUer med spesialiserte prosessorer som GPUer og TPUer, samt utforskning av nye paradigmer som nevromorfe datamaskinarkitekturer (også kalt KI-arkitektur).

Forståelsen av denne historiske utviklingen og de underliggende teoretiske prinsippene er essensiell for å gripe de grunnleggende konseptene i datamaskinarkitektur og for å kunne forutse og forme fremtidige trender i feltet.

3.3 BUSarkitektur: Datamaskinens motorveisystem

Forestill deg en travel motorvei med flere filer, der biler (data) beveger seg raskt mellom ulike destinasjoner (komponenter). Dette er essensen av en BUS i datamaskinarkitektur. BUSer er kritiske for effektiv dataflyt i en datamaskin, og deres design har stor innvirkning på systemets ytelse.

Det finnes flere typer BUSer, hver med sin spesielle funksjon:

1. AdresseBUS: Denne BUSen bærer adressen til minnet eller I/O-enheten som prosessoren ønsker å kommunisere med. Bredden på adresseBUSen bestemmer hvor mye minne systemet kan adressere.

2. DataBUS: Her flyter selve dataene mellom komponentene. Bredden på dataBUSen påvirker hvor mye data som kan overføres i hver operasjon.

3. KontrollBUS: Denne BUSen bærer signaler som koordinerer aktivitetene til ulike komponenter, som lese- og skriveoperasjoner.

En interessant analogi for BUSarkitektur er et postsystem. AdresseBUSen kan sammenlignes med adressene på konvoluttene, dataBUSen med selve innholdet i brevene, og kontrollBUSen med postbudene som sørger for at brevene kommer frem til riktig sted til riktig tid.

Moderne BUSarkitekturer som PCI Express har revolusjonert dataoverføring ved å introdusere seriell, punkt-til-punkt kommunikasjon. Dette kan sammenlignes med å gå fra et system med felles motorveier til et nettverk av dedikerte, høyhastighets toglinjer mellom hver komponent.

3.4 Cache og RAM: Balansegangen mellom hastighet og kapasitet

I moderne datamaskinarkitektur ligger et intrikat samspill mellom ulike typer minne, hver med sine unike egenskaper. Dette samspillet er nøkkelen til å forstå begreper som 'hit' og 'miss' i cache og RAM, samt viktigheten av minnekoherens.

Når prosessoren trenger data, søker den først i cache. Hvis dataene finnes der, kalles det en 'cache hit'. Dette er ideelt, da cache-minnet er ekstremt raskt. Hvis dataene ikke finnes i cache, oppstår en 'cache miss', og systemet må lete i neste nivå av minnet, vanligvis RAM.

La oss bruke en biblioteksanalogi for å illustrere dette: Tenk deg at du er en forsker (prosessoren) som arbeider i et stort bibliotek. Ditt personlige arbeidsområde (L1 cache) har plass til noen få bøker. Rett ved siden av deg har du en liten bokhylle (L2 cache) med plass til flere bøker. I samme rom finnes en større bokhylle (L3 cache). Utenfor rommet ligger bibliotekets hovedsal (RAM) med tusenvis av bøker.

Når du trenger en bok, ser du først på arbeidsbordet ditt (L1 cache hit). Hvis den ikke er der, sjekker du den lille bokhyllen (L2 cache), så den store bokhyllen (L3 cache). Hvis boken ikke finnes i noen av disse, må du gå ut i hovedsalen (RAM) for å finne den. Hver gang du må gå lenger bort for å finne en bok, tar det mer tid - dette representerer ulike nivåer av 'cache miss'.

Minnekoherens blir viktig når vi introduserer flere 'forskere' (prosessorkjerner) som arbeider samtidig. Hvis en forsker oppdaterer informasjon i en bok, må denne oppdateringen reflekteres i alle kopier av boken som andre forskere måtte ha. Dette sikrer at alle arbeider med den mest oppdaterte informasjonen.

En interessant egenskap ved datamaskinminne er at det ofte tar lengre tid å skrive data enn å lese den. Dette kan virke kontraintuitivt, men har flere årsaker:

1. Fysiske begrensninger: Å endre tilstanden i en minnecelle (skriving) krever mer energi og tid enn bare å lese den eksisterende tilstanden.

2. Verifisering: Etter en skriveoperasjon må systemet ofte verifisere at dataene ble korrekt skrevet, noe som tar ekstra tid.

3. Cache-oppdateringer: Når data skrives til hovedminnet, må eventuelle kopier i cache også oppdateres for å opprettholde koherens.

4. Wear-leveling: I noen typer minne, som SSD-er, brukes teknikker for å fordele slitasje jevnt over alle minneceller, noe som kan introdusere ekstra forsinkelse ved skriving.

Denne forskjellen mellom lese- og skriveoperasjoner har implikasjoner for systemdesign og program-

meringsteknikker. For eksempel kan det være fordelaktig å bufre[3] skriveoperasjoner og utføre dem i større blokker, snarere enn å skrive små datamengder ofte.

3.5 Moderne prosessorarkitektur: Mer enn bare flere kjerner

Dagens prosessorer er langt mer enn bare en samling av transistorer. De inkorporerer avanserte teknikker for å maksimere ytelsen:

1. Flerkjernearkitektur: Moderne prosessorer har ofte flere kjerner, hver kapabel til å kjøre egne instruksjonsstrømmer. Dette muliggjør ekte parallell prosessering, hvor flere oppgaver kan utføres samtidig.

2. Hyperthreading: Denne teknikken lar en enkelt fysisk kjerne fremstå som to (eller flere) logiske kjerner for operativsystemet, noe som kan øke effektiviteten i multitasking-scenarioer.

3. Out-of-order execution: Prosessoren kan omorganisere rekkefølgen av instruksjoner for å maksimere ressursutnyttelsen, så lenge det ikke påvirker det endelige resultatet.

4. Spekulativ utførelse: Prosessoren kan 'gjette' utfallet av en betinget instruksjon[4] og begynne å utføre instruksjoner basert på denne gjetningen, noe som kan spare tid hvis gjetningen er korrekt.

Disse avanserte teknikkene har dramatisk økt prosessorenes ytelse, men de har også introdusert nye utfordringer, spesielt når det gjelder energieffektivitet og varmehåndtering.

3.6 Selv-test oppgaver

1. Hva menes med en 'cache hit' og en 'cache miss'? Gi et eksempel på hvordan dette kan påvirke ytelsen til et program.

2. Beskriv konseptet med en databuss. Hvordan påvirker bussens bredde datamaskinens ytelse?

3. Hvorfor tar skriveoperasjoner ofte lengre tid enn leseoperasjoner i datamaskinminne? Nevn minst to årsaker.

4. Forklar begrepet 'minnekoherens' i kontekst av flerkjerneprosessorer. Hvorfor er dette viktig?

5. Gi en kort beskrivelse av en avansert teknikk som brukes i moderne prosessorer for å øke ytelsen.

6. Hvordan kan bufring forbedre ytelsen ved skriveoperasjoner til minne? Gi et praktisk eksempel.

7. Hvordan fungerer hyperthreading, og hvilke fordeler gir det i et flerkjernesystem?

3.7 Fasit til selv-test oppgaver

1. En 'cache hit' oppstår når data som prosessoren trenger finnes i cache-minnet, mens en 'cache miss' oppstår når dataene ikke finnes i cache og må hentes fra tregere minne (som RAM). Eksempel: I et program som gjentatte ganger aksesserer samme data, vil første aksess kunne resultere i en cache miss (tregere), men påfølgende akseser vil være cache hits (raskere), noe som dramatisk kan forbedre programmets ytelse.

[3] En teknikk hvor små datamengder samles opp i et midlertidig, raskt minne (buffer) før de skrives samlet til tregere minne. Dette reduserer antall individuelle skriveoperasjoner og forbedrer systemets ytelse.

[4] typisk jump

2. En databuss er en kommunikasjonskanal som overfører data mellom komponenter i en datamaskin. Bussens bredde refererer til antall bits som kan overføres samtidig. En bredere buss kan overføre mer data per klokkesyklus, noe som kan øke datamaskinens ytelse ved å redusere tiden det tar å overføre store mengder data mellom for eksempel prosessor og minne.

3. Skriveoperasjoner tar ofte lengre tid enn leseoperasjoner av flere årsaker: - Fysiske begrensninger: Å endre tilstanden i en minnecelle krever mer energi og tid enn å lese den. - Verifisering: Etter skriving må systemet ofte verifisere at dataene ble korrekt skrevet. - Cache-oppdateringer: Ved skriving til hovedminnet må eventuelle kopier i cache også oppdateres for å opprettholde koherens.

4. Minnekoherens refererer til konsistensen av data lagret i lokale cacher til forskjellige prosessorkjerner i et flerkjernesystem. Det er viktig fordi det sikrer at alle kjerner arbeider med den mest oppdaterte versjonen av data, noe som forhindrer feil og uventede resultater i parallelle beregninger.

5. To typiske avanserte teknikker i moderne prosessorer: - Out-of-order execution: Omorganiserer rekkefølgen av instruksjoner for å maksimere ressursutnyttelsen. - Spekulativ utførelse: 'Gjetter' utfallet av betingede instruksjoner for å spare tid.

6. Bufring forbedrer ytelsen ved skriveoperasjoner ved å samle opp flere små skriveoperasjoner i et raskt minne (buffer) før de skrives samlet til tregere minne. Praktisk eksempel: Ved skriving av en stor fil til disk, kan operativsystemet bufre data i RAM og skrive større blokker til disken sjeldnere, noe som reduserer antall tregere I/O-operasjoner og forbedrer den opplevde ytelsen.

7. Hyperthreading er en teknikk der en enkelt fysisk prosessorkjerne fremstår som to logiske kjerner for operativsystemet. Dette oppnås ved at kjernen har dupliserte registersett og kan veksle raskt mellom to instruksjonsstrømmer. Fordeler i et flerkjernesystem inkluderer forbedret utnyttelse av prosessorressurser, økt gjennomstrømning av instruksjoner, og forbedret ytelse i multitasking-scenarioer, spesielt for applikasjoner som ikke fullt ut utnytter alle tilgjengelige kjerner.

3.8 Refleksjoner over trade-offs i maskinvaredesign

Når vi studerer datamaskinarkitektur, er det lett å bli fanget av de tekniske detaljene og glemme det større bildet. Men essensen av god maskinvaredesign ligger ikke bare i forståelsen av individuelle komponenter, men i evnen til å se helheten og vurdere trade-offs. Dette er kjernen i det vi ønsker at dere (studenter) skal ta med seg fra dette emnet - en dypere forståelse av hvordan valg i maskinvaredesign påvirker hele systemets ytelse, effektivitet og funksjonalitet.

La oss reflektere over følgende scenario:

'Du får tilgang til en nyere teknologi som gjør at transistorene blir vesentlig mindre og at ditt design dermed tar mindre plass. Hva vil du bruke den ekstra plassen til? Og hvorfor?'

Dette spørsmålet åpner for en fascinerende utforskning av prioriteringer og trade-offs i maskinvaredesign. La oss filosofere litt over noen mulige tilnærminger:

1. **Øke antall kjerner:** En åpenbar mulighet er å bruke den ekstra plassen til å legge til flere prosessorkjerner. Dette kunne drastisk øke systemets evne til parallell prosessering. Men er dette alltid den beste løsningen? Flere kjerner betyr økt kompleksitet i håndtering av parallellitet og minnekoherens. Dessuten, ikke alle applikasjoner kan effektivt utnytte et stort antall kjerner. Her ser vi et klassisk trade-off mellom rå regnekraft og systemkompleksitet.

2. **Utvide cache-minnet:** En annen tilnærming kunne være å dedikere mer plass til cache-minne. Dette kunne redusere latens og forbedre ytelsen for mange applikasjoner. Men hvor mye cache er nok? På et visst punkt gir mer cache diminishing returns. Dessuten øker større cache energiforbruket. Her balanserer vi mellom ytelse og energieffektivitet.

3. **Integrere spesialiserte enheter:** Vi kunne bruke plassen til å integrere spesialiserte enheter for oppgaver som kryptering, signalbehandling eller kunstig intelligens. Dette kunne gi dramatiske ytelsesforbedringer for spesifikke arbeidsbelastninger. Men det kommer på bekostning av generalitet. Er det verdt å ofre fleksibilitet for spesialisert ytelse?

4. **Forbedre energieffektivitet:** I stedet for å øke ytelsen, kunne vi bruke den ekstra plassen til å implementere mer sofistikerte strømstyringssystemer. Dette kunne gi lengre batterilevetid i bærbare enheter eller reduserte driftskostnader i datasentre. Men er energieffektivitet viktigere enn rå ytelse?

5. **Øke pålitelighet og feiltoleranse:** Vi kunne bruke plassen til å implementere redundante systemer eller mer robuste feilkorrigeringsmekanismer. Dette kunne være kritisk i systemer hvor pålitelighet er avgjørende, som i medisinsk utstyr eller romfartsteknologi. Men i hvor stor grad skal vi prioritere pålitelighet over ytelse eller energieffektivitet?

Hver av disse tilnærmingene representerer en unik prioritering og et sett med trade-offs. Det fascinerende med maskinvaredesign er at det ikke finnes noe universelt 'riktig' svar. Den optimale løsningen avhenger av konteksten - bruksområdet for systemet, målgruppen, budsjettet, og mange andre faktorer.

Denne typen refleksjon tvinger oss til å tenke utover de tekniske detaljene og vurdere større spørsmål: Hva er formålet med teknologi? Hvordan balanserer vi ytelse mot energieffektivitet? Spesialisering mot fleksibilitet? Kompleksitet mot pålitelighet?

Videre minner det oss om at teknologisk fremskritt ikke bare handler om å gjøre ting mindre, raskere eller mer kraftfulle. Det handler om å gjøre kloke valg om hvordan vi bruker de ressursene vi har tilgjengelig. I en verden med begrensede ressurser og økende bekymringer om energiforbruk og bærekraft, blir disse vurderingene stadig viktigere.

Som fremtidige ingeniører og teknologer, er det essensielt at dere utvikler evnen til å se disse større sammenhengene. Det er ikke nok å forstå hvordan man bygger en datamaskin; man må også forstå hvorfor man bygger den på en bestemt måte, og hvilke konsekvenser disse valgene har.

Til syvende og sist er maskinvaredesign en øvelse i kompromisser og kreativ problemløsning. Det handler om å finne den rette balansen mellom motstridende krav og begrensninger. Ved å reflektere over slike scenarioer, håper vi å inspirere dere til å tenke kritisk og holistisk om teknologi og dens rolle i samfunnet.

3.9 Refleksjonsspørsmål

Disse refleksjonsspørsmålene er ment å stimulere til dypere tenkning og diskusjon rundt temaene vi har gjennomgått. De kan brukes som utgangspunkt for gruppearbeid, individuelle oppgaver, eller bare som en måte å utforske stoffet på egen hånd. Husk at det ofte er i diskusjonen og refleksjonen rundt disse spørsmålene at den dypeste læringen skjer.

1. **Mer er ikke alltid bedre** - Tenk på et scenario hvor det å legge til flere prosessorkjerner faktisk kunne redusere systemets ytelse. Hvilke faktorer ville spille inn? Hvordan ville du argumentere for eller imot å øke cache-størrelsen i et system med begrenset strømtilgang?

2. **Spesialisering versus fleksibilitet** - I hvilke situasjoner ville du prioritere spesialiserte hardwarekomponenter fremfor generelle løsninger? Hvordan påvirker valget mellom spesialisert og generell hardware fremtidige oppgraderingsmuligheter?

3. **Helhetlig systemtenkning** - Hvordan ville du forklare sammenhengen mellom maskinvarearkitektur og programvareytelse til en ikke-teknisk person? På hvilke måter kan god forståelse av hardware påvirke hvordan du skriver programmer?

4. **Fremtidens utfordringer** - Hvilke begrensninger i dagens datamaskinarkitektur tror du vil bli mest kritiske å løse i fremtiden? Hvordan tror du økt fokus på energieffektivitet vil påvirke fremtidige arkitekturvalg?

5. **Kostnad versus ytelse** - I en situasjon med begrenset budsjett, hvilke komponenter i systemarkitekturen ville du prioritere å oppgradere? Hvorfor? Hvordan ville dine prioriteringer endres hvis hovedfokus var energieffektivitet istedenfor ytelse?

6. **Miljøpåvirkning** - Hvordan kan valg i maskinvarearkitektur påvirke et systems miljøfotavtrykk? Hvilke avveininger må vi gjøre mellom ytelse og bærekraft?

7. **Læring og utvikling** - Hvordan har din forståelse av datamaskinarkitektur endret måten du tenker om teknologi på? Hvilke aspekter av datamaskinarkitektur synes du er mest fascinerende eller utfordrende?

3.9.1 Veiledende svar for refleksjonsoppgavene

Her er noen veiledende svar til utvalgte refleksjonsspørsmål. Merk at disse svarene ikke er ment å være uttømmende, men heller å guide diskusjon og videre refleksjon.

1. 'Mer er ikke alltid bedre'

- Et scenario hvor flere kjerner kan redusere ytelsen:
  - I systemer hvor minnebåndbredden er en flaskehals kan flere kjerner faktisk redusere ytelsen. Hvis kjernene konkurrerer om samme minneressurser, kan overhead fra minnekoherens og synkronisering føre til at systemet bruker mer tid på administrasjon enn faktisk prosessering.
  - Eksempel: Et program som er hovedsakelig sekvensielt vil ikke dra nytte av flere kjerner, men vil likevel lide under økt varmeproduksjon og energiforbruk.
- Argumentasjon rundt cache-størrelse i strømbegrenset system:
  - Større cache krever mer energi for både vedlikehold og aksessering
  - Det finnes et 'sweet spot' hvor ytelsesgevinsten fra større cache oppveier energikostnaden
  - Utover dette punktet vil ekstra cache-størrelse gi diminishing returns mens energiforbruket fortsetter å øke lineært

2. 'Spesialisering versus fleksibilitet'

- Prioritering av spesialiserte komponenter:
  - I systemer med veldig spesifikke og uforanderlige oppgaver (f.eks. nettverksrutere)
  - Når ekstrem ytelse er kritisk for en spesifikk type operasjon (f.eks. kryptografiske operasjoner)
  - I energibegrensede miljøer hvor spesialisert hardware kan være mer effektiv
- Økonomiske betraktninger:

- Utviklingskostnader for spesialisert hardware er betydelig høyere
- Produksjonskostnader per enhet er høyere ved små volum
- Vedlikeholds- og oppdateringskostnader kan være større for spesialiserte løsninger
- Break-even analyse må vurdere langsiktig TCO (Total Cost of Ownership)
- Markedsstørrelse og skalerbarhet påvirker økonomisk levedyktighet

- Påvirkning på oppgraderingsmuligheter:
  - Spesialisert hardware kan bli utdatert hvis kravene endres
  - Generell hardware tilbyr mer fleksibilitet for fremtidige behov
  - Balanse mellom kortsiktig ytelse og langsiktig tilpasningsevne

3. 'Helhetlig systemtenkning'

- Forklaring til ikke-teknisk person:
  - Bruk analogien med et kjøkken: Maskinvare er som kjøkkenets fysiske utforming og utstyr, mens programvare er som oppskriftene og måten man bruker kjøkkenet på
  - God kjøkkendesign (maskinvare) gjør det lettere å lage mat effektivt (kjøre programmer)
  - Dårlig planlagt kjøkkendesign kan gjøre selv enkle oppgaver tidkrevende og ineffektive

4. 'Fremtidens utfordringer'

- Kritiske begrensninger:
  - Varmeutvikling og energiforbruk
  - Minnebåndbredde og latens ('memory wall')
  - Fysiske begrensninger i transistorstørrelse
  - Kompleksitet i parallell prosessering

5. 'Kostnad versus ytelse'

- Prioritering med begrenset budsjett:
  - Balansert system er ofte viktigere enn ekstrem ytelse i én komponent
  - Flaskehalser bør identifiseres og adresseres først
  - For generell bruk: Fokuser på komponenter som gir merkbar forbedring i daglig bruk

- Energieffektivitetsfokus:
  - Investere i mer effektive strømforsyninger
  - Velge komponenter med god energi/ytelse-ratio
  - Implementere sofistikerte strømstyringssystemer

6. 'Miljøpåvirkning'

- Maskinvarearkitekturens påvirkning:
  - Direkte påvirkning gjennom energiforbruk og varmeproduksjon
  - Indirekte påvirkning gjennom komponentenes levetid og utskiftningsbehov
  - Materialbruk og produksjonsprosesser
- Ytelse versus bærekraft:
  - Optimalisere for energieffektivitet kan redusere maksimal ytelse
  - Design for lengre levetid kan kreve mer robuste (og dyrere) komponenter
  - Balansere behovet for kjøling mot energiforbruk

3.10 Videre lesing

For studenter som ønsker å dykke dypere i datamaskinarkitektur, anbefales følgende ressurser:

1. 'Computer Architecture: A Quantitative Approach' av John L. Hennessy og David A. Patterson - Dette er en klassiker innen feltet og gir en dyptgående behandling av moderne arkitekturteknikker.

2. 'Modern Processor Design: Fundamentals of Superscalar Processors' av John Paul Shen og Mikko H. Lipasti - Denne boken fokuserer på design av høyytelsesprosessorer.

For de som er interessert i den nyeste utviklingen innen prosessordesign, kan tekniske white papers fra selskaper som Intel, AMD og ARM være verdifulle ressurser.

Husk at datamaskinarkitektur er et felt i rask utvikling. Nye teknikker og teknologier dukker stadig opp, så det er viktig å holde seg oppdatert gjennom faglige tidsskrifter og konferanser.

Kapittel 4

Kretser som driver koden

Innhold

4.1 Læringsmål . . . 33
4.2 ISA vs Micro arkitektur . . . 34
4.3 Instruksjons dekode . . . 34
4.4 Micro arkitektur . . . 34
4.5 Binær addisjon . . . 39
4.6 Grunnleggende logiske porter (Logic Gates) . . . 40
4.7 Addisjonskretser . . . 43
4.8 Læringsmål . . . 48
4.9 Selv-test oppgaver . . . 48
4.10 Løsningsforslag til selv-test oppgaver . . . 49
4.11 Refleksjonsoppgaver . . . 50
4.12 Løsningsforslag for refleksjonsoppgavene . . . 51
4.13 Videre lesing . . . 53

I forrige kapittel tok vi for oss datamaskinarkitekturens grunnleggende komponenter - fra prosessorer og registre til busser og cache-hierarkier. Nå skal vi dykke dypere ned i abstraksjonsnivåene og utforske de faktiske kretsene som driver programkoden vår. Tenk på det som å åpne panseret på en bil - vi går fra å forstå hva de ulike delene gjør, til å se hvordan de faktisk fungerer sammen.

4.1 Læringsmål

Etter å ha gjennomgått dette kapittelet skal du kunne:

- Beskrive prosessen for instruksjonsdekoding og hvordan en instruksjon er bygget opp
- Forklare hvordan en ALU fungerer og hvorfor addisjon er en fundamental operasjon
- Gjenkjenne og beskrive funksjonen til de grunnleggende logiske portene (NOT, AND, OR, XOR, NAND, NOR, XNOR)
- Lese og tolke sannhetsverditabeller for logiske porter og kretser
- Utføre binær addisjon og forstå konseptet med mentepropagering
- Forklare hvordan en halv-adder og full-adder er konstruert og fungerer

4.2 ISA vs Micro arkitektur

Når vi beveger oss nedover i abstraksjonsnivåene, møter vi først ISA (Instruction Set Architecture) - grensesnittet mellom programvare og maskinvare. Dette kan sammenlignes med et lands lover og regler: ISA definerer hvilke instruksjoner en prosessor kan utføre, akkurat som lover definerer hva innbyggere kan og ikke kan gjøre. Under dette ligger mikroarkitekturen, som er den faktiske implementasjonen - hvordan prosessoren utfører disse instruksjonene, på samme måte som politiet og rettsvesenet håndhever lovene.

ISA er prosessorens "kontrakt" med programvaren, som spesifiserer: - Hvilke instruksjoner som er tilgjengelige - Hvordan instruksjonene er kodet - Hvilke registre som er tilgjengelige - Hvordan minnet er organisert og adressert

Mikroarkitekturen, derimot, bestemmer hvordan disse instruksjonene faktisk implementeres i hardware. To prosessorer kan ha samme ISA men helt forskjellig mikroarkitektur, akkurat som to land kan ha samme lover, men forskjellige måter å håndheve dem på.

En interessant og forenklet versjon av en ISA er LMC (Little Man Computer). Denne pedagogiske modellen representerer en grunnleggende datamaskin med et enkelt instruksjonssett, og fungerer som en utmerket introduksjon til hvordan ISA fungerer. I LMC ser vi kjernen av hva en ISA må inneholde - muligheten til å laste data, utføre operasjoner, og lagre resultater - uten alle kompleksitetene i moderne prosessorer.

4.3 Instruksjons dekode

Når en prosessor mottar en instruksjon, må den først "pakke ut" hva instruksjonen betyr. Dette er som å åpne et brev og forstå innholdet - først ser vi hvem det er til (operanden), hva som skal gjøres (operasjonskoden), og eventuelle ekstra detaljer (umiddelbare verdier eller adresser).

En typisk instruksjon i et moderne 64-bits system kan se slik ut:

```
[opcode (6 bit)] [destination (5 bit)] [source1 (5 bit)] [source2 (5 bit)] [immediate (43 bit)]
```

Sammenlign dette med LMC's enkle 3-sifrede instruksjoner, som 199 for addisjon: - 1: Operasjonskoden (ADD) - 99: Minneadressen

Prosessen med å utføre en instruksjon følger en fast syklus:

1. Fetch: Henter instruksjonen fra minnet
2. Decode: Tolker hva instruksjonen betyr
3. Execute: Utfører operasjonen
4. Write-back: Lagrer resultatet

Dette er som et kjøkken der kokken (prosessoren) først leser oppskriften (fetch), forstår ingrediensene og metodene som trengs (decode), lager retten (execute), og til slutt serverer den (write-back).

4.4 Micro arkitektur

Mikroarkitekturen er maskinens "muskulatur" - de faktiske kretsene som utfører arbeidet. I kjernen av denne strukturen finner vi dataflytbanen (datapath), som dirigerer data mellom ulike komponenter.

Se et eksempel på en datapath i Figur 4.1. La oss se nærmere på hvordan denne datapathen faktisk fungerer ved å følge veien til en instruksjon gjennom systemet.

Dataflytbanen i figuren viser en pipelinet arkitektur med fire hovedstadier, markert med vertikale søyler:

IF/ID (Instruction Fetch/Instruction Decode)

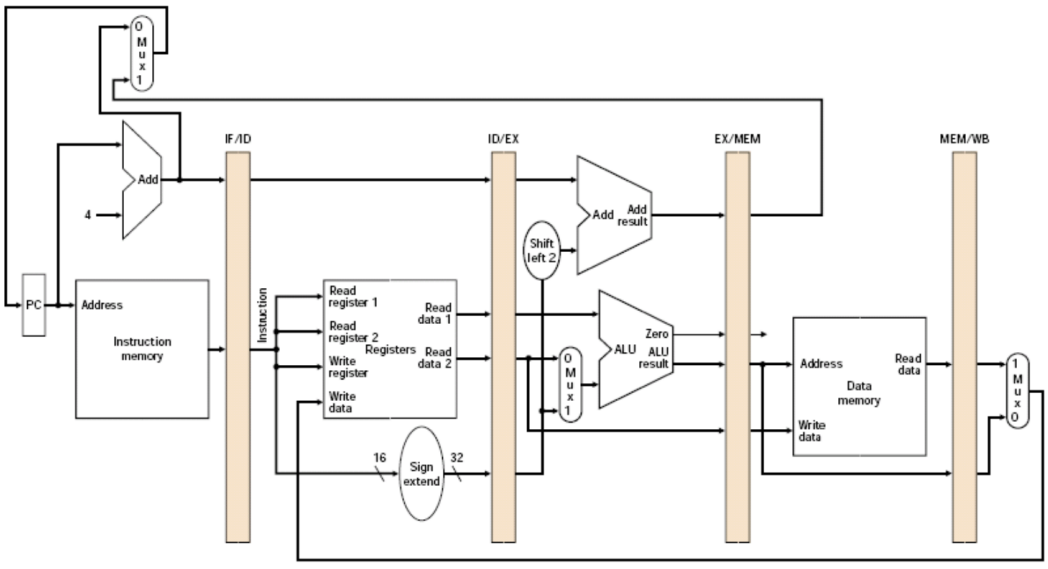
- Program Counter (PC) holder styr på hvilken instruksjon som skal hentes.
- Instruksjonen hentes fra instruksjonsminnet via adressen PC peker på.
- En adderingsoperasjon øker PC med 4 (for 32-bits instruksjoner) for å forberede neste instruksjonshenting.
- Multiplexeren[1] (MUX) øverst lar oss velge mellom normal sekventiell utførelse og hopp.

ID/EX (Instruction Decode/Execute)

- Instruksjonen dekodes og nødvendige registeroperasjoner utføres
- Register-filen håndterer:
  - Lesing av to kilderegistre (Read register 1 og 2)
  - Klargjøring av skriveregister (Write register)
  - Håndtering av data som skal skrives (Write data)
- Sign extend-enheten[2] konverterer 16-bits umiddelbare verdier til 32-bits ved å kopiere fortegnsbit

[1] dette er ikke pensum i IN1020, men en MUX er en komponent som basert på et styresignal velger en av inngangene som kan påvirke utgangen

[2] ikke pensum i IN1020



Figur 4.1: Eksempel på en datapath (32-bits arkitektur)

EX/MEM (Execute/Memory)

- ALU utfører den faktiske beregningen
- En shift left 2-enhet[3] muliggjør effektiv adresseberegning for hopp
- Resultatet fra ALU sendes videre eller brukes til adresseberegning
- Zero-flagget brukes for betingede hopp

MEM/WB (Memory/Write Back)

- Dataminnet aksesseres hvis nødvendig (for load/store operasjoner)
- Resultater skrives tilbake til registerfilen
- Multiplexere velger hvilke data som skal skrives tilbake

Dette designet illustrerer elegant hvordan prosessorer oppnår effektivitet gjennom pipelining - mens én instruksjon er i write-back-fasen, kan den neste være i memory-fasen, en tredje i execute-fasen, og en fjerde i fetch-fasen. Det er som et samlebånd i en fabrikk, hvor hvert stadium utfører sin spesialiserte oppgave samtidig som de andre stadiene jobber med andre instruksjoner, les mer i kapittel 2.3 on page 11.

4.4.1 ALU - Aritmetic and Logic Unit

ALU-en er prosessorens "regnehjerne" - den utfører alle aritmetiske og logiske operasjoner. Tenk på det som en avansert kalkulator som kan gjøre alt fra enkel addisjon til komplekse bitoperasjoner. Den 1-bits ALU-en vi ser i figure 4.2 on the next page er som en enkeltstående LEGO-kloss - den håndterer operasjoner på ett enkelt bit. For å bygge en 128-bits ALU, kobler vi sammen 128 slike enheter, som å bygge en større LEGO-konstruksjon.

Denne 1-bits ALU har følgende hovedkomponenter:

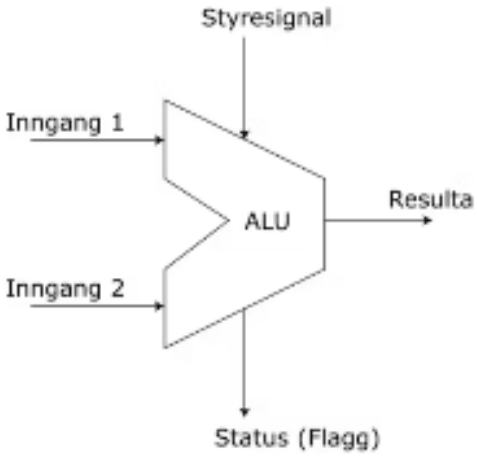
- To innganger - der inngangene er for operander i instruksjonen (tallene som skal behandles)
- Styresignal(er) (bestemmer hvilken operasjon som skal utføres)
- Utgang for resultatet
- Status/flagg som indikerer spesielle forhold (overflow, negative tall, etc.)

For å bygge en 32-bits, 64-bits eller 128-bits ALU, kobles disse 1-bits enhetene sammen i serie, der carry-bit fra én enhet mates inn i den neste [4]. Dette er som å bygge en lang kjede av minikalkulatorer der hver kalkulator håndterer ett siffer og sender eventuell overflyt til neste kalkulator i rekken.

En typisk enkel ALU støtter følgende operasjoner:

1. Aritmetiske operasjoner
    - Addisjon (ADD)

[3] ikke pensum i IN1020
[4] mer om dette i seksjonen om addisjon litt senere



Figur 4.2: 1-bits ALU

- Subtraksjon (SUB)
- Increment (INC)
- Decrement (DEC)

2. Logiske operasjoner
   - AND
   - OR
   - XOR
   - NOT
3. Shift-operasjoner[5]
   - Logical shift left (LSL)
   - Logical shift right (LSR)
   - Arithmetic shift right (ASR)
   - Rotate

4.4.2 ALU i moderne kontekst (ikke pensum i IN1020)

Mens vi til nå har fokusert på den grunnleggende aritmetisk-logiske enheten, har moderne prosessorer utviklet seg til å inkludere en hel familie av spesialiserte ALU-er. Dette er som å gå fra å ha én universalverktøy til å ha en hel verktøykasse full av spesialverktøy - hver optimalisert for sin oppgave, men alle basert på de samme grunnprinsippene.

Floating Point Unit (FPU) er kanskje den mest kjente spesialiseringen. Denne enheten håndterer desimaltall og komplekse matematiske operasjoner som sin, cos og kvadratrot. Tenk på det som en avansert kalkulator innebygd i prosessoren. Der den vanlige ALU-en er begrenset til heltallsoperasjoner,

[5] ikke pensum IN1020

kan FPU-en håndtere tall med desimaler med høy presisjon - kritisk for alt fra værprognoser til 3D-grafikk.

SIMD-enheter (Single Instruction, Multiple Data) representerer en annen fascinerende spesialisering. Disse enhetene kan utføre samme operasjon på flere dataelementer samtidig - som å ha flere samlebånd som kjører parallelt. Dette er særlig nyttig i multimedia-applikasjoner hvor samme operasjon ofte må utføres på store mengder data. For eksempel, når du justerer lysstyrken i et bilde, må samme matematiske operasjon utføres på hver eneste piksel. SIMD-enheter kan håndtere flere piksler samtidig, noe som gir en dramatisk ytelsesøkning.

Kryptografiske ALU-er er et nyere tilskudd til familien. Disse er spesialdesignet for å håndtere de matematiske operasjonene som kreves for moderne kryptering og dekryptering. De inkluderer ofte hardwareoptimaliserte implementasjoner av populære krypteringsalgoritmer, noe som både øker hastigheten og reduserer energiforbruket sammenlignet med programvarebaserte løsninger.

GPU-er (Graphics Processing Units) tar konseptet enda lenger ved å implementere hundrevis eller tusenvis av spesialiserte ALU-er i parallell. Hver av disse er optimalisert for de matematiske operasjonene som er vanlige i 3D-grafikk og maskinlæring. Det fascinerende er at selv disse høyt spesialiserte enhetene fortsatt bygger på de samme grunnleggende prinsippene som vår enkle 1-bits ALU - de adderer, subtraherer og utfører logiske operasjoner, bare i en helt annen skala og med optimalisering for spesifikke bruksområder.

Utviklingen av disse spesialiserte enhetene reflekterer en større trend i datateknologi: balansen mellom generalitet og spesialisering. En generell ALU kan utføre alle nødvendige beregninger, men spesialiserte enheter kan gjøre spesifikke operasjoner mye raskere og mer energieffektivt. Dette er særlig viktig i moderne enheter hvor energieffektivitet er kritisk - fra mobiltelefoner til datasentre.

Framtiden peker mot enda mer spesialisering, med dedikerte enheter for maskinlæring (TPU - Tensor Processing Units), signalbehandling, og andre spesifikke anvendelser. Likevel er det fascinerende å se hvordan alle disse avanserte enhetene fortsatt bygger på de samme grunnleggende prinsippene som vi finner i en enkel ALU. Det er et perfekt eksempel på hvordan fundamentale konsepter i datamaskinarkitektur kan skaleres og tilpasses for å møte nye utfordringer og behov.

4.4.3 Moderne mikroarkitektur (ikke pensum i IN1020)

Dagens mikroarkitekturer, eller nærmere nanoarkitektur, har utviklet seg langt utover den enkle datapath vi tidligere har sett på. Tenk deg forskjellen mellom en enkel enfelts landevei og et moderne motorveisystem med flere filer, av- og påkjøringer, og smart trafikkstyring. På samme måte har moderne prosessorer evolvert til å håndtere flere instruksjoner samtidig, predikere fremtidige hendelser, og optimalisere ressursbruk på måter som ville vært utenkelige for bare noen tiår siden.

En av de mest fascinerende utviklingene er superscalar-arkitekturer. I stedet for å prosessere én instruksjon om gangen, kan disse prosessorene identifisere flere uavhengige instruksjoner og utføre dem parallelt. Det er som å ha flere kassører i en butikk som kan betjene kunder samtidig, i stedet for at alle må stå i samme kø. En moderne prosessor kan starte utførelsen av fire eller flere instruksjoner i hver klokkesykel, forutsatt at instruksjonene ikke er avhengige av hverandre.

Out-of-order execution er en annen revolusjonerende innovasjon. Tradisjonelt måtte instruksjoner utføres i den rekkefølgen de sto skrevet i programmet, som å følge en oppskrift til punkt og prikke. Moderne prosessorer kan derimot analysere instruksjonsstrømmen og identifisere hvilke instruksjoner som kan utføres tidligere uten å påvirke resultatet - som en erfaren kokk som vet at han kan begynne å koke pasta mens sausen fortsatt lages. Dette reduserer ventetiden drastisk og øker effektiviteten.

Branch prediction har blitt stadig mer sofistikert. Moderne prediktorer bruker avanserte algoritmer og maskinlæring for å gjette hvilken vei programmet vil gå ved en forgreining. De holder statistikk over

tidligere beslutninger og kan oppnå treffprosent på over 95%. Dette er som å ha en GPS som ikke bare vet hvor du skal, men også lærer seg dine vanlige ruter og preferanser over tid.

En spennende nyvinning er heterogen arkitektur, ofte kalt big.LITTLE i ARM-prosessorer eller Performance/ Efficient-kjerner i Intel-prosessorer. Dette kombinerer kraftige kjerner for krevende oppgaver med energieffektive kjerner for lettere arbeid - som å ha både en kraftig sportsbil og en økonomisk hybridbil i garasjen, og velge den som passer best for hver tur.

Energieffektivitet har fått økt fokus i moderne design. Prosessorer kan nå dynamisk justere klokkehastighet og spenning (DVFS - Dynamic Voltage and Frequency Scaling) basert på arbeidsmengde og temperatur. De kan også slå av deler av kretsen som ikke er i bruk, såkalt power gating. Dette kan sammenlignes med moderne bygg som automatisk justerer oppvarming og belysning basert på bruk og tid på døgnet.

Disse avanserte teknikkene jobber sammen i en intrikat dans av hardware og kontrolllogikk. Resultatet er prosessorer som ikke bare er raskere enn sine forgjengere, men også mer effektive og fleksible. Den virkelige magien ligger i hvordan all denne kompleksiteten er gjemt bak et relativt enkelt programmeringsgrensesnitt - programmeren trenger ikke bekymre seg for alle disse detaljene, men kan fortsatt dra nytte av ytelsen de gir.

4.5 Binær addisjon

Hvorfor er addisjon så viktig? Fordi praktisk talt alle andre operasjoner kan reduseres til en serie addisjoner. Multiplikasjon er gjentatt addisjon, divisjon er gjentatt subtraksjon (som er addisjon med negative tall), og selv logiske operasjoner kan implementeres ved hjelp av additive kretser. Det er som at addisjon er det grunnleggende verktøyet i prosessorens verktøykasse - med det kan vi bygge alt annet vi trenger.

Men hvordan fungerer egentlig addisjon når maskinen bare har tilgang til nuller og enere?

```
  38
+ 45
----
  83
```

Når vi utfører denne addisjonen, starter vi fra høyre og jobber oss mot venstre. I posisjonen lengst til høyre har vi $8 + 5 = 13$. Vi skriver ned 3 og "har i mente" 1 som vi tar med til neste posisjon. Der blir regnestykket 1 (mente) $+3 + 4 = 8$.

Binær addisjon følger nøyaktig samme prinsipp, men med bare to sifre (0 og 1):

```
   1 1 (mente)
   1 1 0 1
 + 0 1 1 0
---------
   1 0 0 1 1
```

La oss gå gjennom dette steg for steg:

1. Start fra høyre: $1 + 0 = 1$ (ingen mente)
2. Neste posisjon: $0 + 1 = 1$ (fortsatt ingen mente)
3. Neste: $1 + 1 = 10$ (binært) – skriv ned 0, har 1 i mente

4. Siste posisjon: 1 (mente) $+1 + 0 = 10$ (binært) – skriv ned 0, har 1 i mente

5. Mente-tallet blir det siste sifferet i svaret

I binær addisjon har vi bare fire mulige kombinasjoner:

```
0 + 0 = 0
0 + 1 = 1
1 + 0 = 1
1 + 1 = 10 (skriv ned 0, har 1 i mente)
```

Denne prosessen med å propagere mente-tallet gjennom addisjonen er essensielt for å forstå hvordan vi senere skal implementere dette i kretser.

4.6 Grunnleggende logiske porter (Logic Gates)

For å kunne implementere addisjon med kretser, trenger vi først å forstå de grunnleggende byggesteinene – logiske porter. Disse portene manipulerer binære signaler (0 og 1) for å utføre logiske operasjoner.

En **sannhetsverditabell** er en systematisk måte, tabell, å vise alle mulige kombinasjoner av inngangsverder og hvilke utgangsverdier de gir. For porter med to innganger A og B, vil vi ha fire mulige kombinasjoner: (0,0), (0,1), (1,0) og (1,1). Se Tabell 4.1 som et eksempel (F er utgang som foreløpig ikke er definert):

Inngang		Utgang
A	B	F
0	0	
0	1	
1	0	
1	1	

Tabell 4.1: Sannhetsverditabell

4.6.1 NOT (inverter)

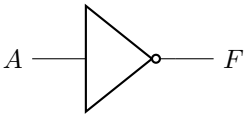
Den enkleste porten – snur verdien og har bare én inngang

$F = \overline{A}$

eller

$F = A'$

A	F
0	1
1	0



Figur 4.3: NOT-port (Inverter)

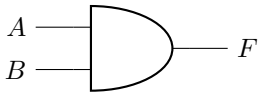
4.6.2 AND

$$F = A \cdot B$$

eller

$$F = AB$$

A	B	F
0	0	0
0	1	0
1	0	0
1	1	1

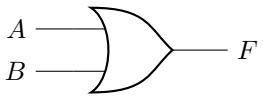


Figur 4.4: AND-port

4.6.3 OR

$$F = A + B$$

A	B	F
0	0	0
0	1	1
1	0	1
1	1	1

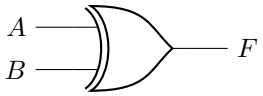


Figur 4.5: OR-port

4.6.4 XOR

$$F = A \oplus B$$

A	B	F
0	0	0
0	1	1
1	0	1
1	1	0



Figur 4.6: XOR-port (Exclusive OR)

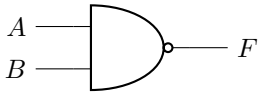
4.6.5 NAND

$$F = \overline{A \cdot B}$$

eller

$$F = (AB)'$$

A	B	F
0	0	1
0	1	1
1	0	1
1	1	0



Figur 4.7: NAND-port

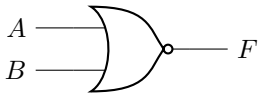
4.6.6 NOR

$$F = \overline{A + B}$$

eller

$$F = (A + B)'$$

A	B	F
0	0	1
0	1	0
1	0	0
1	1	0



Figur 4.8: NOR-port

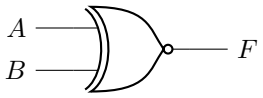
4.6.7 XNOR

$$F = \overline{A \oplus B}$$

eller

$$F = A \odot B$$

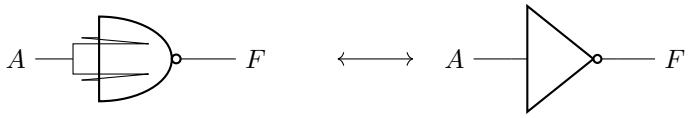
A	B	F
0	0	1
0	1	0
1	0	0
1	1	1



Figur 4.9: XNOR-port

Disse logiske portene er byggesteinene vi bruker for å konstruere mer komplekse kretser. Ved å kombinere dem på smarte måter, kan vi lage kretser som utfører addisjon, subtraksjon og alle andre operasjoner en datamaskin trenger. Det fascinerende er at alt dette – fra enkel addisjon til komplekse beregninger – kan oppnås ved å kombinere disse grunnleggende portene på ulike måter.

En spesielt interessant egenskap ved disse portene er at vi faktisk ikke trenger alle sammen. Enhver logisk funksjon kan konstrueres ved hjelp av bare tre porter: AND, OR og NOT. Dette kalles funksjonell kompletthet. Enda mer fascinerende er at vi kan oppnå det samme med bare én porttype: enten NAND eller NOR. Dette er grunnen til at disse kalles universelle porter. For eksempel kan en NOT-port lages ved å koble begge inngangene på en NAND-port sammen.



Figur 4.10: NOT-port implementert med NAND-port

Når vi ser på bruken av disse portene, er det nyttig å skille mellom logiske og aritmetiske operasjoner. Porter som AND, OR og NOT er primært logiske operatorer – de jobber med sannhetsverdier (true/false eller 1/0). XOR derimot, har en spesiell rolle siden den kan brukes både logisk og aritmetisk. I addisjonskretser brukes XOR for å beregne summen av bits, mens AND brukes for å håndtere mente-bit.

La oss se på forskjellen mellom logiske og aritmetiske operasjoner i en programmeringskontekst:

```
# Logisk operasjon i en for-løkke
for i in range(5):
    if (i > 2) and (i < 4):  # Logisk AND
        print(i)              # Skriver ut 3
```

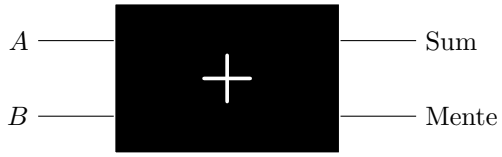
```
# Aritmetisk operasjon i en for-løkke
sum = 0
for i in range(5):
    sum = sum + i            # Aritmetisk addisjon
    # sum blir til slutt 10 (0+1+2+3+4)
```

I det første eksempelet bruker vi `and` som en logisk operator – den sjekker om en betingelse er sann eller usann. I det andre eksempelet bruker vi `+` som en aritmetisk operator – den utfører faktisk tallberegning. På kretsnivå implementeres den aritmetiske addisjonen ved hjelp av flere logiske porter koblet sammen i et bestemt mønster, noe vi skal se nærmere på i neste kapittel om addisjonskretser.

Denne dualiteten – at vi kan bygge aritmetiske operasjoner ved hjelp av logiske porter – er et av de mest elegante aspektene ved digital logikk og danner grunnlaget for hvordan moderne datamaskiner fungerer.

4.7 Addisjonskretser

La oss starte med den enkleste formen for addisjon: to binære tall som skal legges sammen. Vi kan tenke på dette som en svart boks med to innganger og to utganger:



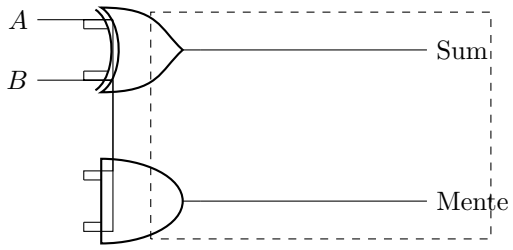
Figur 4.11: Addisjonskrets som svart boks

For å forstå hvordan denne kretsen bør fungere, må vi tenke gjennom alle mulige kombinasjoner av innganger og hvilke utganger vi ønsker. La oss sette opp en sannhetsverditabell:

A	B	Sum	Mente
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

Tabell 4.2: Sannhetsverditabell for addisjonskrets

Ser vi nærmere på denne tabellen, oppdager vi noe interessant. Utgangen "Sum" følger nøyaktig samme mønster som XOR-porten vi lærte om i forrige kapittel. Og hvis vi studerer "Mente"-utgangen, ser vi at den oppfører seg akkurat som en AND-port! Dette gir oss en idé om hvordan vi kan implementere denne kretsen med porter vi allerede kjenner:



Figur 4.12: Implementasjon av addisjonskrets med logiske porter

Men hva skjer når vi skal addere flere bits? I binær addisjon vil vi ofte ha en mente-bit fra forrige posisjon som må tas med i beregningen. La oss se på et eksempel:

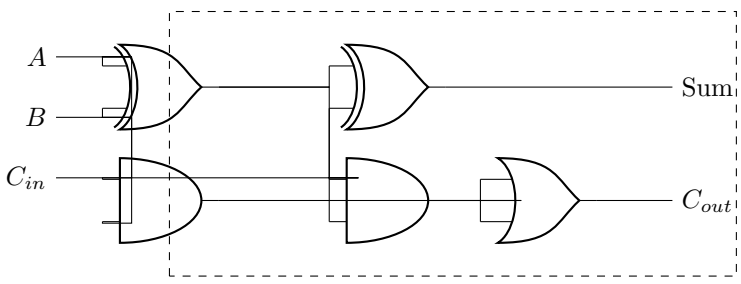
```
  1 1 (mente)
  0 0 1 1
+ 0 1 1 0
---------
  1 0 0 1
```

For å håndtere dette trenger vi en mer avansert krets som også kan ta hensyn til mente inn (Carry in, eller C_{in}). La oss utvide sannhetsverditabellen:

A	B	C_{in}	Sum	C_{out}
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

Tabell 4.3: Sannhetsverditabell for utvidet addisjonskrets

Nå ser vi at mønsteret blir mer komplekst. Den første kretsen vi laget kalles faktisk en halv-adder nettopp fordi den bare håndterer to innganger uten mente inn. Denne nye kretsen, som håndterer tre innganger, kalles en full-adder. Ved å se på sannhetsverditabellen er det ikke lenger åpenbart hvilke porter vi trenger – mønsteret er mer komplisert. Men vi kan faktisk bygge en full-adder ved å bruke to halv-addere og noen ekstra porter:

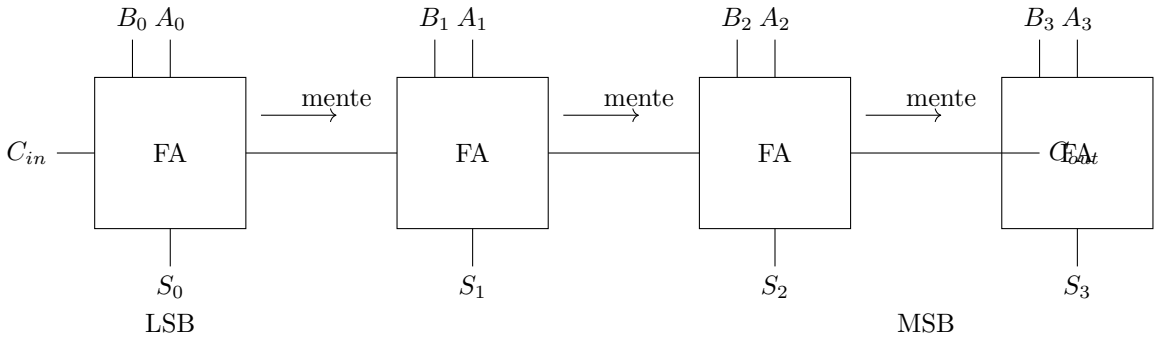


Figur 4.13: Implementasjon av full-adder med to halv-addere

Denne konstruksjonen viser hvordan vi kan bygge mer komplekse kretser ved å kombinere enklere byggeklosser. For å forstå nøyaktig hvorfor denne kombinasjonen av porter gir riktig resultat, trenger vi å se nærmere på boolsk algebra, noe vi skal utforske i neste kapittel.

4.7.1 Ripple Carry Adder

Når vi skal addere to flerbits tall, trenger vi flere full-addere som jobber sammen. Den enkleste måten å koble disse sammen på er i serie, der mente-utgangen fra én full-adder kobles til mente-inngangen på neste full-adder. Dette kalles en ripple carry adder fordi mente-biten "rippler" (bølger) gjennom systemet fra minst signifikante bit (LSB) til mest signifikante bit (MSB).



Figur 4.14: 4-bits ripple carry adder

La oss se hvordan denne kretsen virker ved å følge addisjonen av to 4-bits tall:

```
  1101  (13)
+ 0110  (6)
------
  0011  (19)
  ^
mente propagerer
```

Mentepropageringen er både styrken og svakheten til denne addertypen:

- **Styrke:** Enkel design og lite arealkrevende i hardware
- **Svakhet:** Tidsforsinkelse øker lineært med antall bits

For å forstå tidsforsinkelsen, la oss følge hva som skjer når vi adderer bitene:

1. LSB (minst signifikante bit) beregnes først: $1+0=1$, ingen mente 2. Neste bit må vente på eventuell mente fra forrige bit: $0+1=1$ 3. Tredje bit: $1+1=0$ med mente 4. MSB må vente på mente fra alle tidligere beregninger: $1+0+1=0$ med mente ut

Denne sekvensielle propagering av mente-biten skaper en forsinkelse som kalles "carry propagation delay". For en n-bits adder blir den totale forsinkelsen:

$$T_{total} = n \times T_{carry}$$

der T_{carry} er tiden det tar for én mente-bit å propagere gjennom én full-adder.

For små tall (få bits) er denne forsinkelsen akseptabel, men for større tall blir den problematisk. For eksempel, i en 32-bits adder må vi i verste fall vente på at mente-biten propagerer gjennom alle 32 trinn før vi får det endelige resultatet. Dette er grunnen til at moderne prosessorer bruker mer sofistikerte adderdesign som carry-lookahead og carry-skip addere for kritiske operasjoner.

En interessant analogi er å tenke på mentepropageringen som en rekke dominobrikker: Når første brikke faller, må vi vente på at hver brikke slår ned den neste før vi ser resultatet på slutten av rekken. Jo flere brikker, jo lengre må vi vente på det endelige resultatet.

4.7.2 Moderne addere (ikke pensum IN1020)

Mens ripple carry adderen er enkel å forstå og implementere, har behovet for raskere beregninger ført til utviklingen av mer sofistikerte adderdesign. La oss se på noen moderne tilnærminger som brukes i dagens prosessorer.

Carry Look-ahead Adder (CLA)

Den mest revolusjonerende forbedringen i adderdesign var introduksjonen av carry look-ahead prinsippet. I stedet for å vente på at mente-biten skal propagere gjennom hver full-adder, beregner CLA alle mente-bits parallelt.

For å forstå hvordan dette fungerer, introduserer vi to viktige signaler for hver bitstilling:

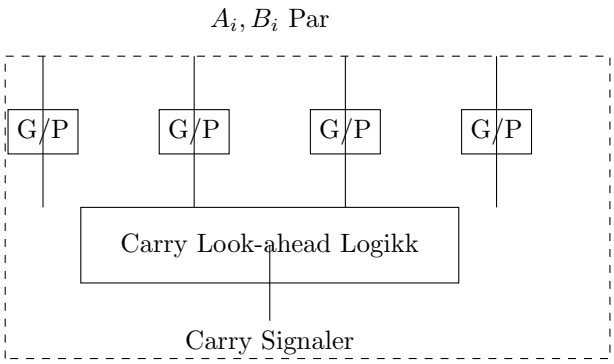
- Generate (G_i): Signaliserer at denne posisjonen vil generere en mente uavhengig av innkommende mente
- Propagate (P_i): Signaliserer at denne posisjonen vil propagere en innkommende mente videre

For hver bit i har vi:

$$G_i = A_i \cdot B_i$$
$$P_i = A_i \oplus B_i$$

Mentesignalet for hver posisjon kan da beregnes direkte:

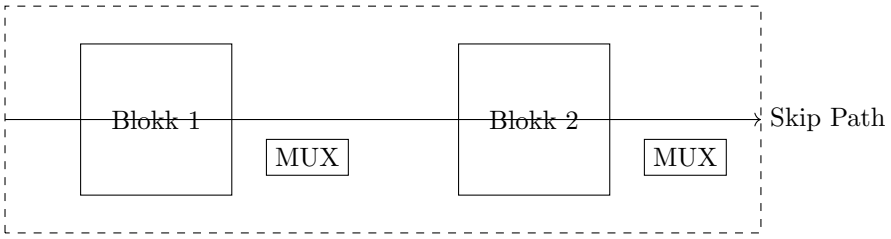
$$C_{i+1} = G_i + (P_i \cdot C_i)$$



Figur 4.15: Forenklet struktur av en Carry Look-ahead Adder

Carry Skip Adder

En annen elegant løsning er carry skip adderen, som representerer et kompromiss mellom kompleksitet og hastighet. Den deler addisjonsprosessen inn i blokker og kan "hoppe over" hele blokker hvis de bare propagerer mente uten å generere ny.



Figur 4.16: Prinsippskisse av en Carry Skip Adder

Carry Save Adder (CSA)

Når vi trenger å addere flere tall samtidig (for eksempel i multiplikasjon), bruker vi carry save addere. I stedet for å håndtere mente-bits umiddelbart, "sparer" denne adderen på mentene og håndterer dem i en separat operasjon.

Dette er spesielt nyttig i:

- Multiplikasjonskrester
- Flyttallsenheter (FPU)
- Digital signalbehandling (DSP)

Parallell Prefix Adder

Den mest avanserte formen for moderne addere er parallell prefix addere, som inkluderer varianter som:

- Kogge-Stone adder
- Brent-Kung adder
- Han-Carlson adder

Disse adderene bruker sofistikerte nettverk av logiske porter for å beregne alle mentesignaler med minimal forsinkelse, men med høyere kompleksitet i hardware.

Addertype	Forsinkelse	Kompleksitet
Ripple Carry	$O(n)$	Lav
Carry Look-ahead	$O(\log n)$	Høy
Carry Skip	$O(\sqrt{n})$	Middels
Parallel Prefix	$O(\log n)$	Meget høy

Tabell 4.4: Sammenligning av ulike adderarkitekturer

I moderne prosessorer brukes ofte en hybrid tilnærming, der forskjellige addertyper brukes for forskjellige formål:

- Ripple carry for ukritiske operasjoner hvor areal og energiforbruk er viktigst
- CLA eller parallel prefix for kritiske beregninger hvor hastighet er avgjørende
- Carry save for spesialiserte operasjoner som multiplikasjon

Dette viser hvordan moderne prosessordesign balanserer ulike hensyn som hastighet, energiforbruk, og kompleksitet for å oppnå optimal ytelse for ulike bruksscenarier.

4.8 Læringsmål

Etter å ha gjennomgått dette kapittelet skal du kunne:

- Forklare forskjellen mellom ISA og mikroarkitektur, og hvordan disse relaterer til hverandre
- Beskrive prosessen for instruksjonsdekoding og hvordan en instruksjon er bygget opp
- Identifisere de ulike delene i en datapath og forklare deres funksjon
- Forklare hvordan en ALU fungerer og hvorfor addisjon er en fundamental operasjon
- Gjenkjenne og beskrive funksjonen til de grunnleggende logiske portene (NOT, AND, OR, XOR, NAND, NOR, XNOR)
- Lese og tolke sannhetsverditabeller for logiske porter og kretser
- Utføre binær addisjon og forstå konseptet med mentepropagering
- Forklare hvordan en halv-adder og full-adder er konstruert og fungerer
- Beskrive virkemåten til en ripple carry adder og dens begrensninger
- Identifisere ulike typer moderne addere og deres fordeler/ulemper

4.9 Selv-test oppgaver

1. Hvilken logisk port vil gi output 1 kun når begge innganger er 1? a) OR b) AND c) XOR d) NAND

2. Tegn sannhetsverditabellen for en XOR-port.

3. Utfør følgende binære addisjon:

```
  1101
+ 0110
------
```

4. Hvorfor kalles den første adderkretsen vi lærte om for en "halv-adder"?

5. Du har en krets med to innganger A og B og én utgang F. Når du tester alle mulige kombinasjoner får du følgende resultat:

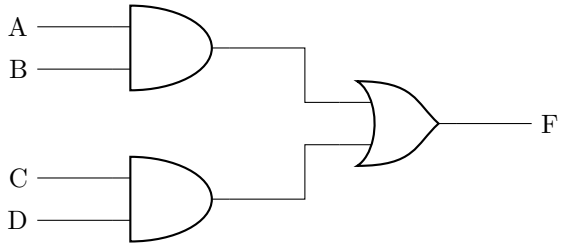
```
A B | F
0 0 | 1
0 1 | 1
1 0 | 1
1 1 | 0
```

Hvilken logisk port er dette?

6. Når vil en ripple carry adder være på sitt tregeste? Begrunn svaret.

7. Design en krets som sammenligner to bits A og B og gir output 1 kun hvis A = B. Hvilke porter vil du bruke?

8. Gitt følgende krets:



a) Lag sannhetsverditabellen for kretsen b) Kan denne kretsen forenkles? Hvis ja, hvordan?

4.10 Løsningsforslag til selv-test oppgaver

1. Svar: b) AND AND-porten er den eneste av de opplistede portene som kun gir 1 når begge innganger er 1.

2. Sannhetsverditabell for XOR:

A	B	F
0	0	0
0	1	1
1	0	1
1	1	0

3. Løsning:

```
  1101
+ 0110
------
 10011
```

4. Den kalles halv-adder fordi den kun kan håndtere to innganger (A og B), men ikke noen mente-inngang fra tidligere addisjon. En full-adder kan håndtere tre innganger: A, B og mente inn.

5. Dette er en NAND-port. Det er den eneste porten som gir 0 kun når begge innganger er 1, og 1 i alle andre tilfeller.

6. En ripple carry adder er tregest når mente-biten må propagere gjennom alle posisjonene, fra minst signifikante bit til mest signifikante bit. Dette skjer for eksempel når vi adderer 0111 + 0001, der mente-biten må "ripple" gjennom hele tallet for å gi 1000.

7. Vi kan bruke en XNOR-port, som gir 1 når inngangene er like. Alternativt kan vi bygge kretsen med basis-porter: (A·B) + (A'·B') Dette gir 1 når både A og B er 1, eller når både A og B er 0.

8. a) Sannhetsverditabell:

A	B	C	D	F
0	0	0	0	0
0	0	0	1	0
0	0	1	0	0
0	0	1	1	1
0	1	0	0	0
0	1	0	1	0
0	1	1	0	0
0	1	1	1	1
1	0	0	0	0
1	0	0	1	0
1	0	1	0	0
1	0	1	1	1
1	1	0	0	1
1	1	0	1	1
1	1	1	0	1
1	1	1	1	1

b) Uttrykket kan skrives som F = (A·B) + (C·D) = AB + CD
Dette er en sum-av-produkter form og kan ikke forenkles ytterligere.

4.11 Refleksjonsoppgaver

1. I kapittelet har vi sett hvordan alle komplekse operasjoner til slutt kan reduseres til addisjon. Tenk på noen hverdagslige eksempler der vi bruker multiplikasjon eller divisjon. Hvordan ville du forklart for en medstudent hvordan disse operasjonene kan utføres ved hjelp av bare addisjon?

2. Vi har lært at NAND-porten er en universell port, som betyr at alle andre porter kan bygges ved hjelp av bare NAND-porter. Hvilke fordeler og ulemper kan du se ved å designe en datamaskin som bare bruker NAND-porter? Tenk på faktorer som kompleksitet, antall komponenter og stabilitet.

3. I kapittelet har vi sett hvordan binær addisjon ligner på desimal addisjon med tanke på mentepropagering. Sammenlign hvordan du løser følgende to oppgaver:

   - 999 + 1 (desimalt)
   - 1111 + 0001 (binært)

   Hvilke likheter og forskjeller ser du i hvordan mentepropageringen fungerer?

4. Tenk på forskjellen mellom ISA og mikroarkitektur. Hvis du skulle forklare dette for noen som aldri har programmert, hvilket hverdagslig eksempel ville du brukt som analogi? Begrunn valget ditt.

5. I en datamaskin må alle beslutninger til slutt reduseres til kombinasjoner av logiske porter. Tenk på en enkel if-setning i et programmeringsspråk:

```
if (age >= 18) and (has_ticket == True):
    allow_entry = True
```

Hvordan ville du gått fram for å implementere denne logikken ved hjelp av porter vi har lært om? Hvilke utfordringer ser du?

6. Vi har sett at en halv-adder bruker en kombinasjon av XOR og AND porter. Prøv å tenke gjennom hvorfor akkurat disse to portene ble valgt:

   - Hva er det med XOR-portens oppførsel som gjør den egnet til å beregne sum?
   - Hvorfor er AND-porten perfekt for å håndtere mente?

   Finn gjerne på en analogi som illustrerer dette.

7. Når vi jobber med digitale kretser, opererer vi med perfekte 0-er og 1-ere. I den virkelige verden er ingen ting perfekt. Hvilke praktiske utfordringer kan du se for seg at dette kan skape? Hvordan tror du en datamaskin håndterer "usikre" signaler?

8. I kapittelet har vi sett hvordan en ripple carry adder bygges opp av flere full-addere i serie. Hvis du skulle redesigne denne kretsen for å gjøre den raskere, hvilke ideer ville du utforsket? Du trenger ikke å komme med en teknisk løsning, men forklar tankegangen din.

4.12 Løsningsforslag for refleksjonsoppgavene

1. Mulig løsning:
   For multiplikasjon kan vi bruke eksempelet 5×3:

   - Dette er det samme som $5 + 5 + 5$
   - Eller tolkes som legg til 5 tre ganger"

   For divisjon kan vi bruke $15 \div 3$:

   - Dette kan sees på som "hvor mange ganger kan vi trekke fra 3 fra 15?"
   - 15 - 3 = 12 (én gang)
   - 12 - 3 = 9 (to ganger)
   - 9 - 3 = 6 (tre ganger)
   - 6 - 3 = 3 (fire ganger)
   - 3 - 3 = 0 (fem ganger)

   Dermed har vi redusert både multiplikasjon og divisjon til gjentatte addisjoner eller subtraksjoner.

2. Mulig løsning:
   Fordeler med bare NAND-porter:

   - Enklere produksjonsprosess da man bare lager én type port
   - Lettere å standardisere testing og kvalitetskontroll
   - Potensielt høyere stabilitet da man har perfeksjonert én type komponent

   Ulemper:

- Flere komponenter trengs for å lage enkle operasjoner
- Mer komplekse kretser kan gi lengre signalforsinkelse
- Høyere strømforbruk da man trenger flere porter

3. Mulig løsning:
   Likheter:

   - Begge systemer krever at mente propagerer fra høyre mot venstre
   - I begge tilfeller må vi vente på at hver posisjon er ferdig før vi kan være sikre på neste
   - Begge viser "worst casescenario for mentepropagering

   Forskjeller:

   - I det desimale systemet må vi håndtere ti ulike verdier, mens binært bare har to
   - Mentepropagering i binært er enklere da vi bare har to mulige utfall

4. Mulig løsning:
   En god analogi er en bil:

   - ISA er som bilens kontroller (ratt, pedaler, girkasse) - grensesnittet sjåføren bruker
   - Mikroarkitektur er som motorens interne design - hvordan kontrollerene faktisk implementeres
   - To biler kan ha samme kontroller (ISA) men helt forskjellige motorer (mikroarkitektur)
   - Sjåføren (programmereren) trenger ikke vite hvordan motoren fungerer for å kjøre bilen (kjøre programmet)

5. Mulig løsning:
   For å implementere denne logikken trenger vi:

   - En komparator for å sjekke age >= 18 (kan bygges med flere porter)
   - En AND-port for å kombinere resultatet med has_ticket
   - Utfordringer inkluderer:
     - Hvordan representere tall som 18 i binær form
     - Bygge en komparator som kan sammenligne tall
     - Håndtere flere bits for aldersrepresentasjon

6. Mulig løsning:
   XOR for sum:

   - XOR gir 1 når inngangene er forskjellige, akkurat som i binær addisjon
   - Dette matcher hvordan vi adderer enkeltbits uten å tenke på mente
   - Som å kaste to mynter - resultatet er krone"hvis myntene viser forskjellig

   AND for mente:

   - Mente oppstår bare når begge bits er 1
   - AND-porten er perfekt siden den kun gir 1 når begge innganger er 1
   - Som to personer som må samarbeide for å løfte noe tungt - begge må bidra

7. Mulig løsning:
   Praktiske utfordringer:

- Elektrisk støy kan påvirke signalnivåer
- Temperaturvariasjoner kan påvirke komponentenes oppførsel
- Timing-problemer når signaler endrer seg

Løsninger:

- Bruk av terskelverdi for å definere hva som er 0 og 1
- Feilkorrigerende koder
- Buffer-soner i timing
- Redundante systemer

8. Mulig løsning:
   Ideer for forbedring:

   - Parallell prosessering av ulike deler av tallet
   - Prediksjon av mente-bits basert på inngangsverdier
   - Dele opp i mindre grupper som kan arbeide uavhengig
   - Som å ha flere kassører i en butikk i stedet for én lang kø

   Tankegangen er å identifisere flaskehalsen (sekvensiell mentepropagering) og finne måter å parallelisere arbeidet på, selv om det kan bety mer kompleks hardware.

4.13 Videre lesing

For studenter som ønsker å fordype seg ytterligere i emnet, anbefales følgende:

- "Computer Organization and Design" av Patterson og Hennessy - Dette er bibelen for datamaskinarkitektur
- "Digital Design and Computer Architecture" av Harris og Harris - Gir en grundig innføring i digital design
- "The Elements of Computing Systems" (også kjent som "Nand to Tetris") - En praktisk tilnærming til å bygge en datamaskin fra bunnen av

For de spesielt interesserte er det også verdt å utforske:

- RISC-V arkitekturen - et åpent instruksjonssett som blir stadig mer populært
- Moderne mikroarkitekturer som ARM's big.LITTLE og Intel's heterogene arkitekturer

Kapittel 5

Bit'enes ballett

Innhold

5.1 Læringsmål 55
5.2 Motivasjon for forenkling 56
5.3 Kretsanalyse: En systematisk tilnærming 57
5.4 Utlesing av en sannhetsverditabell 57
5.5 Forenkling 59
5.6 Refleksjonsspørsmål 60
5.7 Selv-test oppgaver 61
5.8 Fasit og løsningsveiledning for selv-test 62
5.9 Løsningsveiledning til refleksjonsspørsmål 64
5.10 Optimalisering og forenkling 66
5.11 Videre lesing 66

I den digitale verden danser bit'ene en intrikat ballett, hvor hver bevegelse er nøye koreografert gjennom logiske porter og kretser. Denne dansen, selv om den virker kompleks ved første øyekast, følger bestemte mønstre og regler som vi skal utforske i dette kapittelet. Akkurat som en ballettdanser må beherske grunnleggende trinn før hen kan utføre en hel forestilling, må vi forstå hvordan vi kan analysere og forenkle digitale kretser for å skape effektive systemer.

Etter å ha utforsket datamaskinens arkitektur og grunnleggende komponenter i forrige kapittel, skal vi nå dykke dypere inn i hvordan vi kan få bit'ene gjennom logiske kretser. Vi skal se hvordan vi kan analysere disse kretsene systematisk, og ikke minst, hvordan vi kan forenkle dem for å spare både kompleksitet og areal.

5.1 Læringsmål

Etter å ha jobbet med dette kapittelet skal du kunne:

- Forklare hvorfor forenkling av logiske kretser er viktig med tanke på både kompleksitet og arealforbruk
- Beskrive forskjellen mellom en logisk port og en logisk funksjon, og kunne gi eksempler på hver av dem

- Systematisk analysere logiske kretser ved å bruke både:
  - venstre-til-høyre-metoden (fra innganger til utganger)
  - høyre-til-venstre-metoden (fra utganger til innganger)
- Lese og tolke en sannhetsverditabell for å:
  - Identifisere når utgangen er aktiv (1)
  - Oversette hver rad med 1 på utgangen til et logisk uttrykk
  - Kombinere deluttrykkene til ett samlet funksjonsuttrykk
- Forenkle logiske uttrykk ved å:
  - Identifisere felles faktorer
  - Bruke grunnleggende regler som X + X = X og X' + X = 1
  - Faktorisere ut felles variabler
- Gjenkjenne når en kombinasjon av porter kan erstattes med en enklere løsning

5.2 Motivasjon for forenkling

Tenk deg et symfoniorkester hvor hvert instrument spiller sin egen melodi uten å ta hensyn til helheten - resultatet ville blitt kaotisk. På samme måte må vi i digital design[1] strebe etter elegante løsninger som bruker færrest mulig komponenter for å oppnå ønsket funksjonalitet. Hvorfor er dette så viktig?

La oss først definere noen grunnleggende begreper:

Port

En logisk port er en fundamental byggekloss i digitale systemer som utfører en spesifikk boolsk operasjon på én eller flere inngangssignaler for å produsere ett utgangssignal.

Funksjon

En logisk funksjon er et matematisk uttrykk som beskriver sammenhengen mellom inngangs- og utgangsverdier i en digital krets, ofte uttrykt ved hjelp av boolsk algebra.

Når vi designer digitale kretser, har hver port en kostnad i form av:

- Fysisk areal på brikken
- Strømforbruk
- Forsinkelse i signaloverføring
- Varmeutvikling

Ved å forenkle kretsene våre, kan vi redusere alle disse kostnadene. La oss se på et praktisk eksempel.

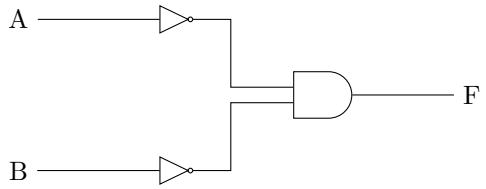
[1] Vi snakker bare om digital design da vi bruker kun 0 og 1. Det finnes også mange elektroniske kretser som er analoge hvor det bruker flere verdier enn 0 og 1.

5.3 Kretsanalyse: En systematisk tilnærming

Når vi møter en ny krets, kan vi analysere den på to måter:

1. Venstre-til-høyre analyse (innganger til utganger)
2. Høyre-til-venstre analyse (utganger til innganger)

La oss se på et konkret eksempel med funksjonen F = A'B'.



Figur 5.1: Krets for F = A'B'

La oss analysere denne kretsen steg for steg fra venstre til høyre:

1. Vi starter med inngangene A og B
2. Begge signalene går gjennom hver sin NOT-port
3. De inverterte signalene (A' og B') går inn i en AND-port
4. Utgangen F blir dermed A'B'

Hvis vi ser på sannhetsverditabellen for denne funksjonen:

A	B	F
0	0	1
0	1	0
1	0	0
1	1	0

Tabell 5.1: Sannhetsverditabell for F = A'B'

Vi ser at dette utgangsmønsteret er identisk med en NOR-port! Dette betyr at vi kan erstatte hele kretsen med én enkelt NOR-port, noe som sparer både areal og kompleksitet.

5.4 Utlesing av en sannhetsverditabell

La oss først se på hvordan vi kan lese ut en funksjon fra en sannhetsverditabell. Når vi skal skrive et uttrykk som beskriver en digital funksjon, ønsker vi å finne ut nøyaktig når utgangen blir 1. Dette er fordi i digitale systemer er vi ofte interessert i å vite under hvilke forhold noe skal aktiveres eller skrus på.

Tenk på det som en oppskrift: hvis vi skal beskrive når en lampe lyser, er det mer naturlig å si "lampen lyser når bryteren er på" enn å også beskrive alle situasjonene der lampen ikke lyser. På samme måte fokuserer vi på tilfellene der funksjonen vår gir 1 som utgang. La oss se på følgende tabell:

A	B	F
0	0	1
0	1	1
1	0	0
1	1	1

Tabell 5.2: Sannhetsverditabell med to innganger

For å bygge funksjonsuttrykket, følger vi disse stegene:

1. Vi ser først etter alle rader der F = 1. Dette er de situasjonene der funksjonen vår er "aktiv".
2. For hver slik rad må vi beskrive den eksakte kombinasjonen av innganger som gir denne 1-eren:
   - Når en inngang er 0, representerer vi dette med en NOT-operasjon (enten x' eller $\overline{x}$)
   - Når en inngang er 1, bruker vi variabelen som den er

For eksempel, la oss se på den første raden der F = 1:

- A = 0: Dette betyr at A må være 0 for at dette leddet skal bli 1
- B = 0: Dette betyr at B også må være 0 for at dette leddet skal bli 1
- Vi skriver dette som A'B' (eller $\overline{A} \cdot \overline{B}$)

Hvorfor blir A = 0 til A'?

1. Når vi skriver A', betyr det "IKKE A" eller "det motsatte av A"
2. Hvis A = 0, er det motsatte av A (altså A') = 1
3. Derfor, når vi vil at A skal være 0, skriver vi A' fordi dette blir 1 når A er 0

La oss nå gå gjennom alle radene der F = 1:

- Rad 1: A = 0, B = 0 gir leddet A'B'
- Rad 2: A = 0, B = 1 gir leddet A'B
- Rad 4: A = 1, B = 1 gir leddet AB

Siden vi ønsker at F skal være 1 når hvilken som helst av disse situasjonene inntreffer, kobler vi dem sammen med OR-operasjoner (+):

$$F = A'B' + A'B + AB \tag{5.1}$$

Samme uttrykk med overline-notasjon[2]:

$$F = \overline{A} \cdot \overline{B} + \overline{A}B + AB \tag{5.2}$$

[2]jeg bruker $\cdot$ mellom variablene for å vise at hver variabel er invertert. Fordi i teksten så blir det vanskelig å se forskjellen på $\overline{AB}$ som er (AB)' og $\overline{A}\overline{B}$ som er A'B'

La oss nå se på et eksempel med **tre** innganger, A, B og C:

A	B	C	F
0	0	0	1
0	0	1	0
0	1	0	1
0	1	1	0
1	0	0	1
1	0	1	0
1	1	0	0
1	1	1	1

Tabell 5.3: Sannhetsverditabell med tre innganger

Her ser vi at F = 1 i rad 1 (A=0,B=0,C=0), rad 3 (A=0,B=1,C=0), rad 5 (A=1,B=0,C=0) og rad 8 (A=1,B=1,C=1). Dette gir oss:

$$F = A'B'C' + A'BC' + AB'C' + ABC \tag{5.3}$$

Samme uttrykk med overline-notasjon:

$$F = \overline{A}\,\overline{B}\,\overline{C} + \overline{A}B\overline{C} + A\overline{B}\,\overline{C} + ABC \tag{5.4}$$

5.5 Forenkling

La oss nå se hvordan vi kan forenkle uttrykkene fra forrige seksjon. Vi starter med det første uttrykket fra likning 5.1:

$$F = A'B' + A'B + AB$$

Vi kan forenkle dette ved å se på de to første leddene. Her ser vi at A' er felles, og B opptrer både som B' og B. Dette kan vi skrive som:

$$F = A'(B' + B) + AB \tag{5.5}$$

Siden B' + B = 1 (dette er en grunnleggende regel som sier at en variabel eller dens negasjon må være sann), får vi:

$$F = A' + AB \tag{5.6}$$

For det andre uttrykket fra likning 5.3:

$$F = A'B'C' + A'BC' + AB'C' + ABC$$

Her kan vi først se på de tre første leddene som alle inneholder C'. Vi kan faktorsiere ut C':

$$F = C'(A'B' + A'B + AB') + ABC \tag{5.7}$$

Inni parentesen kan vi faktorsiere ut A' fra de to første leddene:

$$F = C'(A'(B' + B) + AB') + ABC \tag{5.8}$$

Siden B' + B = 1, får vi:

$$F = C'(A' + AB') + ABC \tag{5.9}$$

Dette er den enkleste formen vi kan oppnå uten å bruke mer avanserte metoder som Karnaugh-diagram.

Ved å følge slike enkle steg, der vi ser etter:

1. Felles variabler som kan faktoriseres ut
2. Uttrykk av typen X' + X som er lik 1
3. Uttrykk av typen X + X som er lik X

kan vi ofte forenkle komplekse uttrykk betydelig.

5.6 Refleksjonsspørsmål

Disse spørsmålene er ment å hjelpe deg med å reflektere over det du har lært i dette kapittelet. Det finnes ofte flere riktige svar, og hensikten er å få deg til å tenke dypere over konseptene.

1. **Kompleksitet og ytelse:**
   - Hvorfor er det ikke alltid best å bruke den enkleste mulige kretsen?
   - Kan du tenke på situasjoner hvor en mer kompleks krets faktisk kunne være bedre?
2. **Analogier til hverdagen:**
   - Kan du finne eksempler fra hverdagen der du bruker lignende logikk som AND, OR og NOT?
   - Hvordan ville du forklart forskjellen på en logisk port og en logisk funksjon til en venn som ikke kan noe om digital elektronikk?
3. **Analyse og forståelse:**
   - Hvilken analysemetode (venstre-til-høyre eller høyre-til-venstre) foretrekker du? Hvorfor?
   - Hvordan påvirker valget av analysemetode måten du tenker på når du ser på en ny krets?
4. **Sannhetsverditabeller:**
   - Hvorfor tror du vi fokuserer på 1'ere når vi leser ut fra en sannhetsverditabell?
   - Kunne vi like gjerne fokusert på nullene? Hva ville vært fordeler og ulemper med det?
5. **Sammenheng med tidligere kunnskap:**
   - Hvordan henger det du har lært i dette kapittelet sammen med det du lærte om addisjon i forrige kapittel?
   - Kan du se noen sammenheng mellom forenkling av logiske uttrykk og forenkling av algebraiske uttrykk fra matematikken?

6. **Praktisk anvendelse:**
    - Kan du tenke på noen eksempler fra din hverdag der digitale kretser spiller en viktig rolle?
    - Hvorfor er det viktig å kunne forenkle kretser når vi designer digitale systemer?
7. **Konseptuell forståelse:**
    - Når du ser en kompleks krets, hva er det første du ser etter?
    - Hvordan ville du gått frem for å forklare en kompleks krets for en medstudent?

Tips: Det kan være nyttig å diskutere disse spørsmålene med medstudenter. Ofte ser andre ting fra andre perspektiver, noe som kan gi deg en dypere forståelse av stoffet. Jeg vil absolutt anbefale deg å delta på gruppetimer.

5.7 Selv-test oppgaver

Disse oppgavene er laget for at du skal kunne teste din egen forståelse av stoffet. Under hver oppgave finner du et symbol som indikerer vanskelighetsgraden:

- ($\star$) = Grunnleggende forståelse
- ($\star\star$) = Middels vanskelig
- ($\star\star\star$) = Utfordrende

Oppgave 1: Grunnleggende begreper ($\star$)

a) Hva er forskjellen på en logisk port og en logisk funksjon?

b) Gitt funksjonen F = AB + A'B', tegn kretsen ved å bruke grunnleggende porter.

c) Sant eller usant:

- En NOT-port kan ha flere innganger
- A + A = A
- A' + A = 1

Oppgave 2: Sannhetsverditabell analyse $\star\star$

a) Skriv ut funksjonsuttrykket fra følgende sannhetsverditabell:

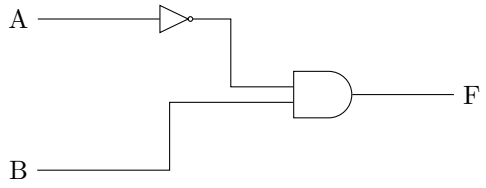
A	B	F
0	0	1
0	1	0
1	0	0
1	1	1

b) Forklar steg for steg hvordan du kom fram til svaret.

Oppgave 3: Kretsanalyse ⋆⋆

Analyser følgende krets og:

- Skriv funksjonsuttrykket
- Lag sannhetsverditabellen
- Beskriv med ord hva kretsen gjør



Figur 5.2: Krets for analyse

Oppgave 4: Forenkling ⋆⋆

Forenkle følgende uttrykk så mye som mulig:

$$F = ABC + ABC' + AB'C \tag{5.10}$$

Vis alle mellomregninger og begrunn hvert steg.

Oppgave 5: Sammensatt oppgave ⋆⋆⋆

En innbruddsalarm skal aktiveres (F = 1) under følgende forhold:

- Når bevegelsessensoren (A) er aktivert OG dørsensoren (B) er aktivert
- ELLER når bevegelsessensoren er aktivert OG det er natt (C)
- ELLER når nødsignalet (D) er aktivert, uansett andre forhold

a) Skriv funksjonsuttrykket for denne alarmen.

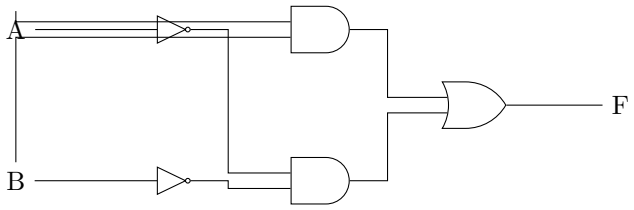
b) Tegn kretsen ved å bruke grunnleggende porter.

c) Kan uttrykket forenkles? Hvis ja, vis hvordan.

5.8 Fasit og løsningsveiledning for selv-test

Oppgave 1: a) En logisk port er en fysisk komponent som utfører én spesifikk operasjon, mens en logisk funksjon kan bestå av flere porter og beskriver en mer kompleks oppførsel.

b) Kretsløsning:



Figur 5.3: Krets for F = AB + A'B'

c)

- usant - NOT-porter har alltid kun én inngang
- Sant
- Sant

Oppgave 2: a) F = A'B' + AB

b) Fremgangsmåte:

- Identifiser rader med F = 1 (rad 1 og 4)
- For rad 1: A=0, B=0 gir leddet A'B'
- For rad 4: A=1, B=1 gir leddet AB
- Kombiner med OR (+)

Oppgave 3:

- Funksjonsuttrykk: F = A'B
- Sannhetsverditabell:

A	B	F
0	0	0
0	1	1
1	0	0
1	1	0

- Kretsen gir 1 kun når A er 0 og B er 1

Oppgave 4: F = ABC + ABC' + AB'C = AB(C + C') + AB'C = AB(1) + AB'C = AB + AB'C

Oppgave 5: a) F = AB + AC + D

b) [Kretsdiagram ville vært tegnet her]

c) Dette uttrykket kan ikke forenkles ytterligere med enkle metoder da det ikke finnes overlappende termer eller åpenbare fellesfaktorer.

Tips: Om du ikke får til en oppgave med en gang, prøv å bryte den ned i mindre deler. Start med det du er sikker på, og jobb deg gradvis mot en fullstendig løsning.

5.9 Løsningsveiledning til refleksjonsspørsmål

1. Kompleksitet og ytelse

Den enkleste kretsen er ikke alltid den beste fordi:

- **Forsinkelse:** En forenklet krets kan noen ganger introdusere flere portnivåer (større dybde) som øker forsinkelsen gjennom kretsen
- **Strømforbruk:** Enkelte "enkle" løsninger kan kreve porter som bruker mer strøm
- **Støyimmunitet:** Mer komplekse kretser kan gi bedre beskyttelse mot elektrisk støy
- **Testbarhet:** En mer kompleks krets kan være lettere å teste og feilsøke

2. Analogier til hverdagen

Eksempler på logiske operasjoner i hverdagen:

- **AND:**
  - For å lage kaffe trenger du både vann OG kaffebønner
  - En heis beveger seg kun når døren er lukket OG en etasje er valgt
- **OR:**
  - Du kan betale med enten kort ELLER kontanter
  - En brannalarm utløses av enten røyk ELLER varme
- **NOT:**
  - Kjøleskapet lyser når døren IKKE er lukket
  - Alarmen går hvis døren åpnes når systemet IKKE er deaktivert

Forskjell på port og funksjon kan forklares som: "En port er som en enkelt bryter eller sensor, mens en funksjon er som et helt system av brytere og sensorer som jobber sammen for å bestemme når noe skal skje."

3. Analyse og forståelse

Begge analysemetoder har sine fordeler:

- **Venstre-til-høyre:**
  - Følger signalenes naturlige flyt
  - Lettere å forstå hvordan endringer i innganger påvirker systemet
  - Nyttig når man skal feilsøke
- **Høyre-til-venstre:**
  - Starter med målet (utgangen) og jobber bakover
  - Ofte nyttig i design når man vet hva man vil oppnå
  - Kan være mer effektiv for å identifisere unødvendige komponenter

4. Sannhetsverditabeller

Vi fokuserer på ettallene fordi:

- Det er ofte færre ettalkombinasjoner enn nullkombinasjoner
- I digitale systemer beskriver vi ofte når noe skal aktiveres (1) heller enn når det ikke skal aktiveres (0)
- Det gir ofte enklere uttrykk matematisk
- Det matcher hvordan vi tenker på funksjonalitet ("når skal noe skje" vs "når skal noe ikke skje")

5. Sammenheng med tidligere kunnskap

Koblinger til addisjon og tidligere kapitler:

- Addisjonskretser bruker kombinasjoner av AND, OR og NOT porter
- Carry-bit i addisjon krever analyse av flere innganger, lignende det vi gjør med komplekse logiske funksjoner
- Forenkling av logiske uttrykk følger lignende prinsipper som algebraisk forenkling:
  - Faktorisering (ta ut fellesfaktorer)
  - Distributive lover
  - Kombinering av like ledd

6. Praktisk anvendelse

Digitale kretser i hverdagen:

- Smarttelefoner: Touch-sensorer, skjermkontroll
- Hvitevarer: Temperaturkontroll, timerfunksjoner
- Biler: ABS-systemer, motorstyringsenheter
- Betalingsterminaler: Validering av PIN-kode

Viktighet av forenkling:

- Reduserer produksjonskostnader
- Minimerer strømforbruk i mobile enheter
- Øker pålitelighet ved færre komponenter
- Forbedrer ytelse gjennom optimalisering

7. Konseptuell forståelse

Ved analyse av komplekse kretser:

- Start med å identifisere hovedkomponentene
- Se etter kjente mønstre eller delkretser
- Identifiser innganger og utganger
- Del kretsen inn i mindre, håndterbare deler
- Analyser hver del separat før du ser på helheten

For å forklare for medstudenter:

- Bruk analogier fra hverdagen
- Start med enkle deler og bygg opp kompleksitet gradvis
- Tegn og forklar steg for steg
- Bruk eksempler med konkrete inngangsverdier
- Oppfordre til aktiv deltakelse gjennom spørsmål

Merk: Denne veiledningen er ment som et utgangspunkt for diskusjon og refleksjon. Det kan finnes andre gode svar og perspektiver som ikke er nevnt her.

5.10 Optimalisering og forenkling

Når vi designer digitale kretser, kan vi tenke på det som å komponere musikk. En erfaren komponist vet hvordan han kan oppnå samme musikalske effekt med færre noter, akkurat som en erfaren kretsdesigner vet hvordan han kan oppnå samme funksjonalitet med færre porter.

Noen nyttige strategier for kretsforenkling inkluderer:

- Identifisering av kjente mønstre (som i eksempelet over)
- (ikke pensum) Bruk av Karnaugh-diagrammer for systematisk forenkling
- (ikke pensum) Anvendelse av DeMorgans lover
- (ikke pensum) Utnyttelse av don't care-tilstander

5.11 Videre lesing

For studenter som ønsker å fordype seg ytterligere i dette feltet, anbefaler vi følgende ressurser:

- "Digital Design and Computer Architecture" av David Money Harris og Sarah L. Harris - En utmerket bok som går dypere inn i digital design og optimalisering.

- "Contemporary Logic Design" av Randy H. Katz og Gaetano Borriello - Gir en moderne tilnærming til logisk design og analysemetoder.
- "Introduction to Logic Design" av Alan B. Marcovitz - Spesielt god på systematiske metoder for kretsanalyse og optimalisering.

For de spesielt interesserte anbefales også å utforske:

- Kvantemekaniske effekter i moderne digitale kretser
- Moderne synteseverktøy og deres optimeringsalgoritmer
- Maskinlæringsbaserte tilnærminger til kretsoptimalisering

Kapittel 6

Transistorer

Innhold

6.1 Fra vakuumrør til silisiumrevolusjonen 69
6.2 Moores lov og den eksponentielle veksten 70
6.3 Transistor 70
6.4 MEMS - Når transistorer møter mekanikk 72
6.5 Den fjerde industrielle revolusjon 73
6.6 Fremtidsperspektiver 73
6.7 Hastighet, last, lekasje og synkronisering 73
6.8 Den evige balansen 75
6.9 Videre lesing 76

Advarsel :)

Mye av dette kapittelet er ikke-pensum, men skrevet som støtte for dere og gir mange detaljer. Hva som er pensum kommer på forelesningen.

I en hver eneste elektronisk enhet vi bruker - fra den minste smartklokken til de mest kraftfulle superdatamaskinene - finnes det milliarder av mikroskopiske helter som utfører sitt arbeid i det stille. Disse usynlige heltene er transistorene, og deres historie er like fascinerende som deres betydning er omfattende. Praktisk talt er all komersiell digital halvleder teknologi CMOS (Complementary Metal On Oxide) teknologi.

Tenk deg et øyeblikk at du står på toppen av et høyt fjell og ser utover en stor by om kvelden. Millioner av lys blinker og skinner i mørket, hver og en representerer menneskelig aktivitet og liv. På samme måte fungerer en moderne prosessor, der milliarder av transistorer "blinker" av og på i et intrikat samspill som muliggjør alt fra enkel tekstbehandling til kompleks kunstig intelligens.

6.1 Fra vakuumrør til silisiumrevolusjonen

Da de første elektroniske datamaskinene ble bygget, brukte de vakuumrør - store, strømkrevende komponenter som ofte brant ut. ENIAC, en av de første elektroniske datamaskinene, brukte over 17,000 vakuumrør og krevde et helt rom for å utføre beregninger som i dag kan gjøres på en lommeregner.

I 1947 endret alt seg. Ved Bell Laboratories gjorde William Shockley, John Bardeen og Walter Brattain en oppdagelse som skulle revolusjonere verden: transistoren. Denne lille halvlederkomponenten kunne utføre samme jobb som vakuumrørene, men var mindre, mer pålitelig og brukte betydelig mindre strøm.

Det er interessant å merke seg at ikke alle så potensialet i den nye teknologien. Som Thomas Watson, styrelederen for IBM, uttalte i 1943: *"I think there is a world market for maybe five computers."* Denne uttalelsen minner oss om hvor vanskelig det kan være å forutse teknologiske revolusjoner, selv for eksperter i feltet.

6.2 Moores lov og den eksponentielle veksten

I 1965 gjorde Gordon Moore, en av grunnleggerne av Intel, en bemerkelsesverdig observasjon: antallet transistorer på en integrert krets doblet seg omtrent hver 24. måned. Denne observasjonen, kjent som Moores lov, har vist seg å være forbløffende presis i over et halvt århundre.

For å sette dette i perspektiv:

- 1971: Intel 4004 prosessoren hadde 2,300 transistorer
- 1993: Intel Pentium hadde 3.1 millioner transistorer
- 2022: Apple M1 Ultra har over 114 milliarder transistorer

Denne eksponentielle veksten kan være vanskelig å begripe. La oss bruke en analogi: Hvis bilteknologi hadde utviklet seg i samme tempo som datateknologi siden 1971, ville en moderne bil i dag:

- Kunne kjøre 300,000 km/t
- Koste under 1 krone
- Kunne kjøre jorden rundt på en dråpe bensin

6.3 Transistor

La oss nå dykke ned i transistorens fascinerende verden ved å studere den enkleste digitale byggeklossen vi har: **inverteren**. En inverter, også kjent som en NOT-port, har én enkel oppgave - den skal snu verdien av inngangssignalet. Når inngangen er 1 (høy), blir utgangen 0 (lav), og når inngangen er 0 (lav), blir utgangen 1 (høy).

Fun Fact

Visste du at antallet transistorer i hele verden siden 1971 til idag er like mange som det er sandkorn i Sahara x2.

Men hvordan bygger vi egentlig denne funksjonen ved hjelp av transistorer? For å forstå dette må vi først bli kjent med to viktige konsepter: VDD og GND.

6.3.1 VDD og GND - Teknologiens "Opp" og "Ned"

Tenk deg et vannfall. Vannet strømmer alltid fra toppen (høyt potensial) til bunnen (lavt potensial). I elektronikk har vi noe lignende:

- VDD er som toppen av vannfallet - det er vår positive spenningsforsyning, vanligvis 3.3 Volt eller 1.2 V i moderne kretser. Dette representerer vår logiske '1'. Vi kan for enkelhetsskyld tenke på VDD = 1 V.
- GND (Ground) er som bunnen av vannfallet - det er vårt referansepunkt, 0 V. Dette representerer vår logiske '0'.

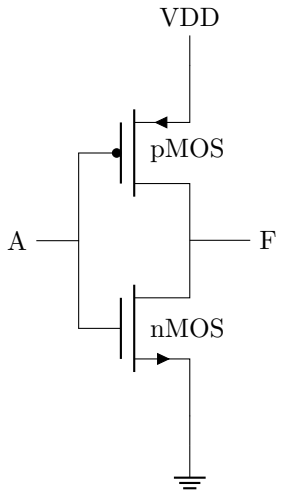
Strømmen i en elektronisk krets vil, akkurat som vann, alltid "falle" fra høyere til lavere potensial - fra VDD mot GND.

6.3.2 Inverteren på Transistornivå

En inverter består av to typer transistorer:

- En pMOS-transistor (p-type) - tenk på denne som en bryter som er PÅ når inngangen er 0
- En nMOS-transistor (n-type) - tenk på denne som en bryter som er PÅ når inngangen er 1

Her er hvordan vi kobler dem sammen:



La oss se hvordan denne kretsen fungerer, steg for steg:

Når inngangen A er 0 (GND):

1. pMOS-transistoren blir PÅ (som en lukket bryter)
2. nMOS-transistoren blir AV (som en åpen bryter)
3. Utgangen F kobles til VDD gjennom pMOS-transistoren
4. Resultatet blir at F = 1 (høy)

Når inngangen A er 1 (VDD):

1. pMOS-transistoren blir AV (som en åpen bryter)
2. nMOS-transistoren blir PÅ (som en lukket bryter)
3. Utgangen F kobles til GND gjennom nMOS-transistoren
4. Resultatet blir at F = 0 (lav)

For å gjøre dette enda mer konkret, kan vi tenke på transistorene som vannkraner:

- pMOS er en kran som er åpen når ingen trykker på den (0), og lukket når noen trykker på den (1)
- nMOS er en kran som er lukket når ingen trykker på den (0), og åpen når noen trykker på den (1)

Når vi setter disse kranene i serie, med VDD som vannkilde på toppen og GND som avløp på bunnen, får vi vår inverter!

6.3.3 Fra Én til milliarder

Det fantastiske med denne enkle byggeklossen er at den kan brukes som utgangspunkt for å bygge mer komplekse kretser. Ved å kombinere flere transistorer på lignende måter kan vi skape NAND-porter, NOR-porter, og til slutt hele prosessorer. En moderne prosessor inneholder milliarder av slike transistorer, alle basert på de samme grunnleggende prinsippene vi nettopp har lært om.

Dette er grunnen til at forståelsen av hvordan en enkel inverter fungerer er så viktig - den gir oss nøkkelen til å forstå hvordan alle moderne digitale enheter fungerer på sitt mest grunnleggende nivå.

6.4 MEMS - Når transistorer møter mekanikk

Mikroelektromekaniske systemer (MEMS) representerer neste steg i miniatyriseringen av teknologi. Disse enhetene kombinerer elektroniske og mekaniske komponenter på mikroskopisk nivå. Tenk på det som bittesmå maskiner, mindre enn et sandkorn, som kan måle bevegelse, trykk, eller til og med utføre fysiske handlinger.

MEMS-teknologi finner vi i:

- Moderne smarttelefoners gyroskoper
- Airbag-sensorer i biler
- Mikrofonene i høreapparater
- Projeksjonssystemer i moderne projektorer

6.5 Den fjerde industrielle revolusjon

Vi står nå midt i det mange kaller den fjerde industrielle revolusjon, eller "Information Evolution". Mens de tidligere revolusjonene handlet om damp, elektrisitet og automatisering, handler denne om sammensmelting av fysiske, digitale og biologiske systemer.

Julius Sextus Frontinus uttalte i år 10: "Inventions have long since reached their limit, and I see no hope for further development." To tusen år senere fortsetter vi å flytte grensene for hva som er mulig, mye takket være transistoren og dens evolusjon.

6.6 Fremtidsperspektiver

Mens tradisjonelle silisiumbaserte transistorer nærmer seg sine fysiske grenser, åpner nye teknologier spennende muligheter:

- Kvantedatamaskiner som utnytter subatomære fenomener
- Karbonbaserte transistorer som kan være enda mindre og raskere
- Nevromorfiske chip som etterligner hjernens arkitektur
- 3D-stablede transistorer som utnytter vertikal integrasjon

6.7 Hastighet, last, lekasje og synkronisering

For å forstå hvorfor moderne datamaskiner trenger klokker og synkronisering, må vi først dykke ned i transistorenes fysiske verden og tidsaspekter. Her beveger vi oss i en verden der tid måles i pikosekunder (10^{-12} sekunder) og avstander i nanometer (10^{-9} meter).

6.7.1 Den mystiske teknologilengden

Når vi hører at "Apple utvikler 2nm transistorer", hva betyr egentlig dette? Opprinnelig refererte dette målet til lengden av den såkalte gate-kanalen i en transistor - avstanden mellom source og drain. Men i moderne transistorer er dette tallet mer et markedsføringsnavn enn et faktisk fysisk mål.

Tenk på en transistor som en veldig liten lysbryter. I en tradisjonell lysbryter har du en fysisk avstand mellom kontaktpunktene som må dekkes for å skape forbindelse. I en transistor er gate-lengden tilsvarende denne avstanden. Men moderne transistorer er mer komplekse, med 3D-strukturer (såkalte FinFET eller Gate-All-Around strukturer) der den faktiske gate-lengden ofte er større enn den annonserte "teknologilengden".

6.7.2 Å endre en utgang - det handler om kapasitans

La oss gå tilbake til vår inverter og se hva som faktisk skjer når vi skal endre utgangen. Tenk deg en hageslag som er full av vann (representerer en logisk 1). For å tømme slangen (endre til logisk 0) må vi vente på at vannet renner ut. Jo lengre slange, jo lengre tid tar det.

I den elektriske verdenen kaller vi dette kapasitans. Hver transistor og ledning i en krets fungerer som en bitteliten kondensator som må lades opp eller ut. Dette tar tid, og denne tiden avhenger av:

1. Last-kapasitansen (C_L):
    - Størrelsen på transistorene som er koblet til utgangen
    - Lengden på metallbanen som kobler komponenter sammen
    - Antall komponenter som er koblet til utgangen
2. Drivstyrken til transistorene:
    - Hvor "sterk" transistoren er (W/L-forhold)
    - Forsyningsspenningen (VDD)

Forsinkelsen kan uttrykkes som:

$$t_d \propto (C_L \times V_{DD})/I_{drive} \tag{6.1}$$

Der I_{drive} er strømmen transistoren kan levere.

6.7.3 Kampen mot lekasje

Hvis vi tenker oss tilbake til hageslang-analogien - hva hvis slangen har små hull? Dette er nettopp hva som skjer i moderne transistorer. Det finnes tre hovedtyper lekasje:

1. Subterskel-lekasje:
    - Som en kran som drypper selv når den er stengt
    - Øker dramatisk når transistorene blir mindre
    - Særlig problematisk i mobile enheter der batterilevetid er viktig
2. Gate-lekasje:
    - Når isolasjonen mellom gate og kanal blir så tynn at elektroner kan "tunnelere" gjennom
    - Blir verre jo tynnere gate-oksiden er
3. Junction-lekasje:
    - Strøm som lekker mellom source/drain og substratet
    - Øker med temperatur

6.7.4 Tidsperspektiver og behovet for synkronisering

La oss sette dette i perspektiv med noen typiske tidsverdier i en 14nm prosessteknologi:

- Gate-forsinkelse (enkel inverter): ca 5 picosekunder
- Metallbane-forsinkelse (1 mm): 1 mikrosekunder
- Klokkeperiode (3GHz prosessor): 333 picosekunder
- Lekasjestrøm utladningstid: nanosekunder til mikrosekunder

Dette skaper flere utfordringer:

1. Varierende Forsinkelser:
    - Ulike baner gjennom kretsen tar ulik tid
    - Temperatur påvirker forsinkelser
    - Produksjonsvariasjon gir ulike karakteristikker
    - Spenningsvariasjoner påvirker hastighet
2. Race Conditions:
    - Signaler som skal ankomme samtidig kan komme i ulik rekkefølge
    - Kan føre til feil resultat eller ustabil oppførsel

Dette er grunnen til at vi trenger synkronisering. Ved å dele tiden inn i diskrete intervaller med en klokkesignal, kan vi:

- Vente på at alle signaler har stabilisert seg
- Garantere at data er gyldig når det samples
- Håndtere variasjon i forsinkelser

Tenk på det som et travløp der hestene må vente ved hver målstrek før de får fortsette til neste runde. Dette sikrer at ingen "løper i fra" hverandre, selv om noen er raskere enn andre.

6.8 Den evige balansen

Prosessorutvikling handler derfor om å balansere flere motstridende hensyn:

- Høyere hastighet $\iff$ Mer lekasje
- Mindre transistorer $\iff$ Større produksjonsvariasjon
- Lavere spenning $\iff$ Mer sårbar for støy
- Tettere pakking $\iff$ Varmeutvikling

Dette er grunnen til at moderne prosessorer bruker teknikker som:

- Flere frekvensdomener
- Dynamisk frekvens- og spenningsskalering
- Power gating for å slå av ubrukte deler
- Komplekse klokkenettverk for presis synkronisering

Ved å forstå disse fundamentale begrensningene og utfordringene, kan vi bedre sette pris på de utrolige ingeniørbragdene som moderne prosessorer representerer, og hvorfor synkronisering er så viktig i digital design.

6.9 Videre lesing

For studenter som ønsker å fordype seg ytterligere i emnet, anbefales følgende ressurser:

1. "The Chip: How Two Americans Invented the Microchip and Launched a Revolution" av T.R. Reid - En fascinerende historie om transistorens og mikrochipens utvikling.

2. "Fundamentals of Modern VLSI Devices" av Yuan Taur og Tak H. Ning - For de som ønsker å forstå de fysiske prinsippene bak moderne transistorer.

3. "Physics of Semiconductor Devices" av S.M. Sze - Et standardverk innen halvlederteknologi.

4. "Introduction to MEMS: Fabrication and Applications" av Nadim Maluf - En grundig innføring i MEMS-teknologi.

5. "The Chip War" - om viktigheten på politisk plan av å ha tilgang til å produsere chip.

For de spesielt interesserte anbefales også å følge med på forskningsjournaler som IEEE Electron Device Letters og Nature Electronics, hvor de nyeste fremskrittene innen transistorteknologi publiseres.

Del II

Fasit for selv-test

Fasit til Selv-Test F1

Abstraksjonsnivåer

Svarene her refererer til spørsmålene 1.8 on page 7

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

Register

Instruksjonsdekode, 9
Minnehierarki, 9
Pipeline, 9

Del III

Appendix

Tillegg A

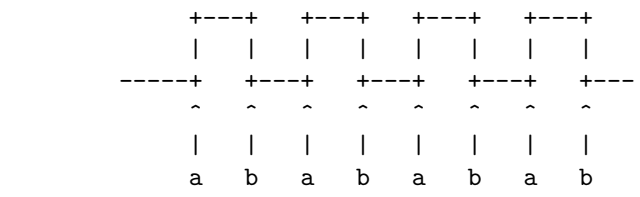
Synkronisering

I hjertet av enhver datamaskin ligger en grunnleggende mekanisme som holder orden på alt som skjer: synkronisering. Tenk deg et orkester uten dirigent, hvor hver musiker spiller i sitt eget tempo. Resultatet ville vært kaotisk. På samme måte trenger datamaskiner en dirigentfor å sikre at alle komponenter arbeider i harmoni. Dette er essensen av synkronisering i dataverdenen.

Før vi dykker dypere inn i synkroniseringens verden, la oss bli kjent med noen sentrale begreper:

- **Klokkesignal**: Dette er hjerterytmen i en datamaskin. Et klokkesignal er en regelmessig puls som styrer timingen av operasjoner i en digital krets.
- **Klokkeperiode**: Tiden mellom to påfølgende stigende (eller fallende) flanker i klokkesignalet. Dette definerer hvor raskt operasjoner kan utføres.
- **Stigende flanke**: Overgangen fra lav til høy spenning i klokkesignalet.
- **Fallende flanke**: Overgangen fra høy til lav spenning i klokkesignalet.

For å illustrere disse begrepene, la oss se på en enkel representasjon av et klokkesignal:



Denne illustrasjonen viser et typisk klokkesignal. Avstanden mellom to påfølgende stigende flanker representerer klokkeperioden.

Frekvensen til et klokkesignal måles i Hertz (Hz) og representerer antall svingninger per sekund. For å beregne frekvensen, bruker vi formelen:

$$\text{Frekvens} = \frac{1}{\text{Klokkeperiode}} \tag{A.1}$$

I moderne teknologi bestemmes frekvensen ofte av flere faktorer, som **Prosessorarkitektur** der ulike CPU-design har forskjellige optimale frekvensområder. **Varmeutvikling** hvor høyere frekvenser genererer mer varme, som må håndteres effektivt. **Strømforbruk** øker ved høyere frekvenser fordi det krever mer energi. **Produksjonsteknologi** og utviklingen av nyere produksjonsmetoder gir muligheter for høyere frekvenser. Det er fascinerende å tenke på at dagens smarttelefoner ofte opererer med klokkefrekvenser på flere gigahertz - milliarder av svingninger per sekund!

A.1 Global klokke og klokkestrategier

I en CPU er konseptet om en global klokke sentralt. Dette er hovedpulsen som synkroniserer alle operasjoner. Men å distribuere et enkelt klokkesignal over en hel chip er ikke alltid praktisk eller effektivt. Derfor har ulike klokkestrategier blitt utviklet:

1. **Synkron klokke**: En enkelt klokke distribueres over hele chipen. Dette er enklest, men kan være utfordrende i store systemer.
2. **Hierarkisk klokke**: Hovedklokken fordeles i et hierarki, med lokale buffere som regenererer signalet.
3. **Asynkron design**: Deler av chipen opererer uavhengig, med lokale klokker eller handshake-protokoller.

Valget av klokkestrategi påvirker både ytelse og energieffektivitet i moderne prosessorer. Hvorfor er synkronisering så viktig? La oss bruke en analogi:

> *Tenk deg et travelt veikryss uten trafikklys. I rushtiden ville det raskt oppstå kaos, med biler som kolliderer eller blokkerer hverandre. Trafikklysene fungerer som en synkroniseringsmekanisme, som sikrer at trafikken flyter jevnt og sikkert.*

På samme måte trenger datamaskiner synkronisering for å sikre at data er stabile før de leses eller skrives, for å koordinere kommunikasjon mellom ulike komponenter, for å unngå race conditions hvor resultatet avhenger av tilfeldige timingforskjeller, og for å muliggjøre pipelining og andre avanserte teknikker for ytelsesoptimalisering.

A.2 Portforsinkelse og synkroniseringens opprinnelse

Behovet for synkronisering oppstod tidlig i datamaskinens historie, hovedsakelig på grunn av et fenomen kjent som portforsinkelse. Hver logisk port i en digital krets, enten det er en AND-port, OR-port eller noe mer komplekst, trenger litt tid for å produsere et stabilt utgangssignal etter at inngangssignalene endres.

Flere faktorer påvirker portforsinkelsen som **Porttype**: Komplekse porter har generelt større forsinkelse enn enkle porter. **Fanout**: Jo flere porter en utgang driver, jo større blir forsinkelsen. **Temperatur**: Høyere temperaturer øker vanligvis forsinkelsen. **Produksjonsvariasjon**: Selv identiske porter på samme chip kan ha litt forskjellig forsinkelse.

La oss bruke en annen analogi for å illustrere dette:

> *Tenk deg en rekke med dominobrikker. Når du velter den første brikken, tar det litt tid før den neste faller, og så videre. Hvis du skulle telle antall falne brikker på et bestemt*

tidspunkt, ville resultatet variere avhengig av hvor i rekken du ser. På samme måte kan ulike deler av en digital krets være i forskjellige tilstander på et gitt øyeblikk på grunn av portforsinkelser.

Synkronisering løser dette problemet ved å introdusere **'ventepunkter'** - øyeblikk hvor vi vet at alle signaler har stabilisert seg. I vår dominoanalogi ville dette være som å vente til alle brikkene har falt før vi teller dem.

Mens synkronisering løser mange problemer, bringer det også med seg utfordringer. Klokketrær, som brukes for å distribuere klokkesignalet over en hel chip, krever betydelig strøm og areal. Klokkeforskyvning (clock skew), som er forskjeller i ankomsttiden for klokkesignalet på ulike deler av chipen, kan også være et problem. I tillegg kan jitter, små uønskede variasjoner i timingen av klokkeflanker, påvirke systemets ytelse.

Fremtidige trender i synkronisering inkluderer globalt asynkrone, lokalt synkrone (GALS) systemer, som kombinerer fordelene med synkrone og asynkrone design, adaptiv klokking, hvor klokkefrekvensen justeres basert på arbeidsmengde og termiske forhold, og optiske klokkedistribusjonssystemer, som bruker lys i stedet for elektriske signaler for å distribuere klokken, potensielt med lavere forsinkelse og strømforbruk.

Synkronisering er altså en fundamental del av datamaskinarkitekturen, og forståelsen av hvordan det fungerer er avgjørende for å designe og optimalisere effektive systemer. Ved å mestre disse konseptene og teknikkene, kan vi bygge datamaskiner som utnytter kraften i synkronisering for å levere imponerende ytelse og effektivitet.

A.3 Videre lesing

For deg som ønsker å fordype deg ytterligere i synkronisering og relaterte emner anbefales følgende ressurser:

1. Digital Design and Computer Architecture"av David Harris og Sarah Harris - Gir en grundig innføring i digital design, inkludert avanserte synkroniseringsteknikker.
2. Principles and Practices of Interconnection Networks"av William J. Dally og Brian Towles - Utforsker synkronisering i kontekst av nettverkskommunikasjon i datamaskiner.
3. "Modern VLSI Design"av Wayne Wolf - Diskuterer synkroniseringsutfordringer i moderne integrerte kretser og hvordan de håndteres i VLSI-design.
4. IEEE-tidsskrifter, spesielt "IEEE Transactions on Computers"og "IEEE Journal of Solid-State Circuits Disse inneholder ofte artikler om cutting-edge forskning innen synkronisering og relaterte områder.

A.4 For veldig interesserte ***

Her er noen eksempler fra den virkelige verden og casestudier som viser hvordan synkroniseringsutfordringer har blitt håndtert i design av spesifikke høyytelsesdatamaskinsystemer, med vekt på nye teknikker. (dette er ikke pensum)

Intel Nehalem-mikroarkitektur:
Intel Nehalem, lansert i 2008, introduserte en rekke innovasjoner for å håndtere synkroniseringsutfordringer. En av de viktigste var bruken av en "uncoreregion på chipen, som inkluderte en felles level

3-cache og en integrert minnekontroller. Dette reduserte klokkeforskyvning og jitter ved å minimere avstanden signalene måtte reise. Nehalem brukte også en teknikk kalt dynamisk kretsredundansfor å forbedre utbyttet under produksjon ved å deaktivere deler av chipen som ikke møtte timingkravene, i stedet for å forkaste hele chipen.

AMDs Zen-arkitektur:
AMDs Zen-mikroprosessorarkitektur, introdusert i 2017, tok i bruk en innovativ "Infinity Fabric-sammenkobling for å håndtere kommunikasjon og synkronisering mellom ulike deler av chipen. Infinity Fabric er en høyhastighets, lavforsinkelse, pakkesvitsjet nettverksarkitektur som muliggjør skalerbar ytelse og modulær design. Den tillater også asynkron kommunikasjon mellom ulike klokkedomener, noe som reduserer kompleksiteten og strømforbruket til klokkedistribusjonsnettverket.

IBMs z15-mainframe:
IBMs z15-mainframe, lansert i 2019, bruker en teknikk kalt "Autonomisk synkroniseringsnettverkfor å håndtere klokkedistribusjon og timing. Dette systemet overvåker klokkenettverket og justerer automatisk for variasjon og degradering over tid, slik at systemet kan opprettholde pålitelig drift over lengre perioder. z15 bruker også et faseløst klokkedistribusjonssystemsom eliminerer behovet for en ekstern referanseklokke og reduserer jitter.

Nvidia Volta GPU-arkitektur:
Nvidias Volta-arkitektur for grafikkprosessorer (GPU-er), introdusert i 2017, inneholder flere innovasjoner for å håndtere synkronisering i et høyt parallelt system. Volta introduserte et nytt konsept kalt "independent thread scheduling", som gjør det mulig for GPU-en å utføre synkroniseringsoperasjoner på trådnivå uten å involvere CPU-en, noe som drastisk reduserer synkroniseringskostnadene. Volta inkluderer også spesialiserte synkroniseringsprimitiver i maskinvaren, som barrierinstruksjoner, for effektiv inter-tråd-kommunikasjon.

Google TPU (Tensor Processing Unit):
Googles TPU er en spesialisert chip for maskinlæringsarbeidsbelastninger. TPU-ene er designet for å være svært energieffektive og skalerbare, noe som krever nøye håndtering av synkronisering og kommunikasjon. TPU-er bruker en teknikk kalt systolisk matrisemultiplikasjon", hvor data flyter synkront gjennom et gitter av prosesseringselementer, med lokale forbindelser som eliminerer behovet for en global klokke. Dette reduserer energiforbruket og gjør det mulig å skalere til svært store matriser.

A.5 Selv-test oppgaver

For å teste din forståelse av synkronisering, prøv å løse følgende oppgaver:

1. Forklar forskjellen mellom stigende flanke og fallende flanke i et klokkesignal.
2. En prosessor har en klokkefrekvens på 2.5 GHz. Beregn klokkeperioden i nanosekunder.
3. Nevn tre faktorer som påvirker valget av klokkefrekvens i moderne teknologi.
4. Tegn et enkelt klokkesignal som viser 4 komplette sykluser. Merk av stigende og fallende flanker.
5. Forklar konseptet global klokke"og diskuter hvorfor det kan være utfordrende å implementere i store systemer.
6. Gi et eksempel på en race condition"og forklar hvordan synkronisering kan forhindre dette problemet.
7. En digital krets har følgende portforsinkelser:
    - AND-port: 2 ns

- OR-port: 3 ns
- NOT-port: 1 ns

Hvis et signal må passere gjennom 2 AND-porter, 1 OR-port, og 3 NOT-porter, hva er den totale forsinkelsen?

8. Sammenlign synkron og asynkron design i digitale systemer. Nevn en fordel og en ulempe ved hver tilnærming.
9. Forklar konseptet "clock skew"og diskuter hvordan det kan påvirke ytelsen til en digital krets.
10. En prosessor bruker en hierarkisk klokkestrategi. Beskriv hvordan dette fungerer og hvilke fordeler det gir sammenlignet med en enkel global klokke.
11. Diskuter hvordan økende klokkefrekvenser påvirker varmeutvikling i moderne prosessorer. Hvordan håndteres dette problemet i praksis?
12. Gi et eksempel på hvordan adaptiv klokking kan forbedre energieffektiviteten i et digitalt system.