Grunnleggende digitalteknikk

Grunnleggende digitalteknikk

Per Thorvaldsen

FAGBOKFORLAGET

1. utgave / 1. opplag 2019

ISBN: 978-82-450-2498-2

Grafisk produksjon: John Grieg, Bergen

Omslagsdesign ved forlaget
Omslagsfoto © Shutterstock/VikaSuh

Forfatteren har mottatt støtte fra Det faglitterære fond

Spørsmål om denne boken kan rettes til:
Fagbokforlaget
Kanalveien 51
5068 Bergen
Tlf.: 55 38 88 00 Faks: 55 38 88 01
e-post: fagbokforlaget@fagbokforlaget.no
www.fagbokforlaget.no

Pålogga

Livet er en periode
Logga på vår ensomme klode
Hen har du vært, hvor skal du dra?

Tida har sin egen kode
Lagra i vårt glemsomme hode
Når er du på og når logger du av?

Kjerstin Aune (1971–)

Til leseren

«Digital? Every idiot can count to one.»

Bob Widlar (1937–1991)

Kjære leser
Er du glad i tynne bøker? Har du lyst til å lære grunnleggende digitalteknikk relativt fort og smertefritt? Da er denne boken som skapt for deg. Dersom du skal studere grunnleggende elektrofag samtidig, kan den gjerne nytes sammen med sin analoge tvilling *Grunnleggende elektroteknikk – kort og godt fortalt* [13].

Denne boken er et ærlig forsøk på å skrive en kort og underholdende innføring i digitalteknikk. Boken er for deg som skal studere digitalteknikk i høyere utdanning og er interessert i å få en god forståelse av emnet. Boken kan selvfølgelig også leses av andre som ønsker en rask og grundig innføring i digitalteknikk eller ønsker å oppfriske gammel kunnskap. Smugles gjerne kapittelet «Møt familien» før du kjøper boken. Da ser du hva du får.

Hvor skal en så begynne? Det er de som hevder at digitalteknikk i dag er redusert til kun programvare, og at en kan lære emnet ved å starte med høynivåprogrammering og drille seg ned til transistorene og de logiske kretsene på bunnen. En slags variant av Reodor Felgens «Dra meg baklengs inn i verktøykassa». Andre mener at en må starte med de grunnleggende prinsipper og så bygge stein på stein. Jeg valgte etter mye om og men å følge en sekvensiell logikk fra bunn til topps. Dersom du er tilhenger av i bakvendtland-metoden, foreslår jeg at du starter med utfordringen i kapittelet «UARTig?» og leser det du trenger basert på behov.

Intensjonen med *Grunnleggende digitalteknikk* er at den skal inneholde gode og utfyllende forklaringer av det essensielle innholdet i emnet. Les boken etter lystprinsippet. Den kan leses fra A til Å, men du kan også lese litt her og der basert på det du trenger. Ønsker du for eksempel å lære mer om hvordan en datamaskin virker, så leser du kapittelet «Følg instruksen!». Dersom du har det travelt og ønsker et kræsjkurs i digitalteknikk, er det bare å starte med kapittelet «UARTig?» og ta det derfra.

I møte med studenter opplever jeg ofte at vi har problemer med og lurer på det samme. For min egen del skyldes det tretti år i næringslivet som har gjort meg litt rusten i de grunnleggende kunnskaper og ferdigheter. Vel, når det gjelder digitalteknikk, så var mine forkunnskaper nærmest lik null. Jeg startet som noen studenter gjør, med å prøve å lære det hele ved kun å lese forelesningsnotater, og der hvor jeg møtte motstand, startet jeg jakten på gode forklaringer andre steder. De finnes både her og der, og noen har jeg faktisk tenkt ut selv. Husk, vi har alle vært i samme båt, og for å sitere en tidligere lærerkollega: «Nå har jeg stått her i tjue år og sagt det samme, men de har enda ikke forstått det.»

Min utfordring er din fordel. Du går jo som kjent ikke til en øyenlege som ikke bruker briller, og heller ikke til en skomaker som ikke vet hvor skoen trykker. Nå får du læreren som har slitt med å forstå, og som sammen med gode kollegaer og studenter har funnet forståelige forklaringer.

All erfaring tilsier at ingenting kommer av seg selv. Skal du bli god i grunnleggende digitalteknikk, må du arbeide. Det må leses og øves fra første dag. «Øvelse gjør mester!» husker jeg var tittelen på norskboken i barneskolen. Det gjelder fremdeles. Følg gjerne oppskriften: studier – hvile og refleksjonstid – mer studier. En effektiv måte å avdekke manglende kunnskaper og ferdigheter på er å regne eksamensoppgaver. Det bør du starte med umiddelbart. I denne boken brukes eksamensoppgaver som døråpnere for teori.

Det viktigste er hva du lærer, ikke hvordan du tilegner deg lærdommen. Bruk gjerne alternative kilder dersom du har utfordringer med å forstå noe. Finn den forklaringen du forstår best. Denne boken er ment å være én kompakt kilde til kunnskap innen grunnleggende digitalteknikk. Det finnes et vell av andre bøker der ute som går mer i dybden, har flere regneøvelser og tusenvis av sider. Der kan du finne mye bra, og jeg har i appendiks gitt noen forslag til videre lesning basert på bøker jeg selv har hatt stort utbytte av å lese.

Hva med andre læringsressurser? Absolutt! Forelesninger gir en grei innføring i stoffet, særlig hvis du har forberedt deg på forhånd. Internett er en utømmelig kilde, og særlig kan enkle animasjoner være en god innfallsport til dypere forståelse. Selv lærer jeg nye ting ved å starte med gode populærvitenskapelige bøker og lærebøker jeg kjenner godt. Etter det brukes så nettet for alt det det er verdt til å utvide kunnskapen.

«Seeing is believing.» I grunnleggende digitalteknikk er en så heldig at mesteparten av teorien kan enten simuleres eller etterprøves på laboratoriet. Få deg en simulator og lek med kretser. Dersom du er så heldig å ha et laboratorium i nærheten eller et utviklingskit, bruk det for å øke forståelsen samt få praktisk erfaring. Den beste måten

å lære digitalt design på er å arbeide med konkrete prosjekt slik som det for eksempel er gjort i kapittelet «UARTig?».

Jeg vil få takke mine kollegaer ved Høgskolen på Vestlandet for gode diskusjoner. Jeg vil spesielt få berømme høgskolelektor Svein Haustveit for forelesningsnotater, forklaringer og oppgaver til en utålmodig og nysgjerrig kunnskapssøkende. Stor takk også til bokens konsulent, Ragnhild Terese Veimo Larsen.

Jeg vil også få rette en takk til Fagbokforlaget og redaktør Hilde Kristoffersen for å oppfordre meg til å gi opp den skjønnlitterære forfatterdrømmen og heller satse på fagbøker. Et absolutt fornuftig råd. Selv ikke jeg leser romaner lenger. Jeg vil også få takke Norsk faglitterær forfatter- og oversetterforening for et raust skrivestipend som gjorde det mulig å få realisert boken.

Det er vanlig i slike innledninger at forfatteren påtar seg ansvar for alt som måtte være feil i boken. Det får så være. Trykkfeil, eventuelt dårlige forklaringer eller flauser legges fortløpende ut på bokens hjemmeside.

Til slutt uttrykker jeg i tillegg et håp om at du vil ha like stor glede av å lese boken som jeg hadde av å skrive den.

Hilsen
Per Thorvaldsen
www.fagbokforlaget.no/grunnleggende_digitalteknikk
PS! Ta gjerne kontakt dersom du har kommentarer, forslag til forbedringer eller annet.
Per.Eilif.Thorvaldsen@hvl.no

QUIZ: Melding utenfra
Dersom følgende melding hadde blitt mottatt fra det ytre rom, hvordan kunne vi ha gjettet at det var sendt ut av menneskelignende vennligsinnede vesener med én arm dobbelt så lang som den andre (tips: antall siffer i meldingen er produkt av to primtall)?

00110001011000111111100110010011001001100101111000100100010010001001
01100110

Du kan sjekke ditt svar med fasit i appendiks dersom du ønsker det.

Til læreren

Hensikten med denne boken er å gi leseren en god forståelse av emnet digitalteknikk. Den inneholder derfor en blanding av grunnleggende digitalteknikk, matematikk og litt fysikk. I utgangspunktet er boken ment som tilleggslitteratur til emnet digitalteknikk slik det undervises i høyere utdanning. Boken kan selvfølgelig også anvendes sammen med boken *Grunnleggende elektroteknikk – kort og godt fortalt* som pensumlitteratur til et to-semesters innføringskurs i grunnleggende elektrofag.

Boken har en litt annerledes stil enn typiske lærebøker i digitalteknikk. Det brukes humor, analogier, anekdoter og historiske merknader. Dette er gjort bevisst for å fenge leseren og for å øke forståelsen. Boken er nærmest en antitese av en formelsamling.

Kapittelet «UARTig?» er viktig, da det viser hvordan en kan bruke grunnleggende kunnskaper til syntese av et nytt design.

Mye av teorien blir innført ved hjelp av å løse eksamensoppgaver. Hensikten er å vise at teorien har relevans, og at å løse oppgaver er en god måte å tilegne seg nytt stoff på. Boken mangler oppgavesett. Det må suppleres fra andre kilder.

I innholdsfortegnelsen er det angitt læringsutbytter for de enkelte kapitler, og det er naturlig å bruke kapitlene frem til eller med «UARTig?» som pensum for et emne i grunnleggende digitalteknikk.

Kapitlene «Kun i minne finner hjertet fred», «Følg instruksen!», «Fremskritt og tilbakesteg» og «Fra A til B» er stoff som tradisjonelt ikke undervises på innføringsnivå i grunnleggende digitalteknikk. Disse kapitlene om minne, datamaskiner, digital signalprosessering og transmisjon er skrevet for å hjelpe studentene når de senere i sine studier møter disse emnene.

Deler av teksten i boken er merket med rasterbakgrunn. Det er supplerende kunnskap som kan leses av den som ønsker seg et dypdykk i teori.

Innholdsfortegnelse

APPENDIKS

1

Kapittel 1

Kjært barn har mange navn

«'Om dem, hvis Navn ender på -sen,' sagde hun,
'dem kan der nu aldrig I Verden blive Noget af!'»

Hans Christian Andersen (1805–1875)

LÆRINGSUTBYTTE: Konvensjoner, symboler, notasjon, enheter, definisjoner og konstanter

«Kanskje den viktigste og mest berømte masteroppgave i det tjuende århundre.» Det er ikke mine ord. De stammer fra psykolog Howard Gardners bok *The Mind's New Science: A History of the Cognitive Revolution*. Hadde Gardner lest min oppgave? Nei, han refererte til Claude Shannons «A Symbolic Analysis of Relay and Switching Ciruits» fra 1937. I en alder av 21 år endret unge Shannon verdenshistorien. Han foreslo å bruke elektriske kretser til å løse logiske og matematiske problemer, og viste hvordan det kunne gjøres på en matematisk velfundert måte. Grunnlaget for den digitale revolusjonen var lagt.

Oppgaven til Shannon, som bærer preg av å ha vært skrevet med en litt sliten skrivemaskin med problemer å holde både linjene og fargebåndet, ligger lett tilgjengelig på Internettet. Den er overkommelig å lese og vel verdt et studium. Det er bare en liten hake. På side 5 innfører han en notasjon som er stikk motsatt den vi bruker i dag. Hans binære enhet 0 er vår 1 og vice versa. I tillegg er funksjonen OG definert som $+$ hvor vi foretrekker $\cdot$, og ELLER som $\cdot$ hvor konvensjonen nå er $+$. Claude Shannon var som sagt først ute, så han kunne gjøre hva han ville. At de som kom etter ham, kom på andre tanker, er ikke hans feil. Ja, ja, samme kan det være. Den underliggende logikken forandres ikke med notasjonen.

Og det stopper ikke der. Du skal ikke dra lenger enn til Danmark for å finne ut at det er ulike måter å telle på. De har rester av et tyvetallsystem hvor tress er $3 \cdot 20 = 60$, firs $4 \cdot 20 = 80$ og fems $5 \cdot 20 = 100$. Så har du de artige variantene halv tres, halv firs og halv fems. Det betyr at du skal ta med bare halve det siste sneset, og for eksempel blir halv firs $4 \cdot 20 - 10 = 70$. Når jeg handler i Danmark, har jeg for lengst gitt opp å klare å følge med. Jeg gir bare en stor nok pengeseddel. Problemet med den strategien er at en returnerer til hjemlandet med halve Danmarks myntbeholdning. Åkkesom, samme hva slags tellesystem en velger, er heldigvis den matematiske basisen den samme.

Så var det symbolene da. Har en klart å bli enig der? Nei, dessverre. I figur 1.1 ser du to ulike symboler for den samme OG-port. Det til venstre er den amerikanske varianten, mens det til høyre er den europeiske definert av en IEC norm. Jeg har valgt det amerikanske systemet.

Figur 1.1 Symbol for OG-funksjonen

Det triste faktum er at en alltid vil måtte leve med ulike konvensjoner, notasjoner, symboler og enheter. Trøsten er at de bakenforliggende fenomener er helt uavhengige av hvordan vi beskriver dem. Det er kun i lærebøker du vil finne konsistent bruk av notasjon. I virkeligheten må du være forberedt på å møte det meste.

Bare slik at vi kan være enige om begrepene, har jeg hentet følgende definisjoner fra leksikon.

- Konvensjon: Vedtatte eller alminnelig anerkjente retningslinjer
- Symbol: Et symbol er et tegn som har en dypere mening
- Notasjon: Tegnsett med tilhørende anvendelsesregler
- Enhet: Sammenligningsstørrelse i et målesystem
- Definisjon: En definisjon er en setning som forklarer og avgrenser betydningsinnholdet i et ord, uttrykk eller begrep så nøyaktig som mulig.

Definisjonen av definisjon er jo en definisjon i seg selv. Da disse formelle tingene kanskje ikke er det mest spennende å starte med, er konvensjoner, symboler, notasjon, enheter, definisjoner og konstanter samlet oversiktlig i appendiks. Skulle du underveis i lesningen trenge en definisjon eller lure på notasjon, er det bare å gå dit.

2

Kapittel 2

Møt familien

«Through the computer, the heralds say, we will make education better, religion better, politics better, our minds better. This is of course, nonsense, and only the young or the ignorant or the foolish could believe it.»

The Nature of Technology, Neil Postman (1931–2003)

LÆRINGSUTBYTTE: Nytten og gleden av grunnleggende digitalteknikk, fra iPhone 4 til NAND-porter

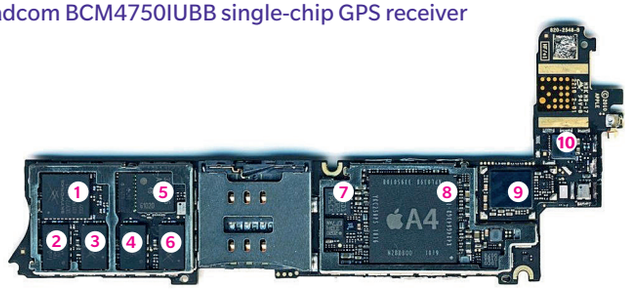
Det hendte i de dager da studenter måtte dele tastatur. Min medstudent og jeg trykket nølende på det oransje Tandberg-tastaturet som senere skulle gi liv til mange av Jan Kjærstads romaner og fantasier. Vi hadde aldri berørt et tastatur før, og var fryktelig redde for å gjøre noe galt da lærekreftene glimret med sitt fravær. Vi skulle lære oss det norske programmeringsspråket Simula som senere satte standarden for alle objektorienterte programmeringsspråk. Å trylle frem de mest fantastiske beregninger ved bare å skrive noen få linjer med grønne bokstaver fortonte seg helt magisk. Min medstudent var etter hvert ikke fornøyd med bare å kunne bruke datamaskinen. Han ville forstå hvordan den virket. Det fant han ut etter møysommelig arbeid, men strøk dessverre til eksamen i programmering. Nysgjerrigheten ga ham kunnskap og en strålende en karriere. Jeg fikk vitnemål.

«Hvor er disse NAND-portene?» Spørsmålet slynges i fortvilelse ut i klasserommet av en student som ikke helt ser sammenhengen mellom sin iPhone og elektronikkens byggesteiner. La oss dekonstruere iPhonen. Foreta såkalt «reverse engineering» og se om vi kan komme til bunns i saken. iPhonen var et strålende eksempel på nytenkning som endret hele telekommunikasjonsbransjen fra å være styrt av teleoperatørene til å bli en pengemaskin for Jobs' selskap Apple sine produkter og tjenester.

La oss se på iPhone 4 som er vist i figur 2.1. Hvorfor den utdaterte iPhone 4? Vel, hadde vi valgt den nyeste, hadde også den vært utdatert før vi hadde vært oppdatert. Det som kjennetegner den digitale teknikken, er nettopp den raske utviklingen. I løpet av mindre enn hundre år har digitalteknikken tatt oss fra ingeniøren Claude Shannons spede tanker om elektriske logiske kretser til kunstig intelligens.

Figur 2.1 Hvor er NAND-portene i iPhone 4?

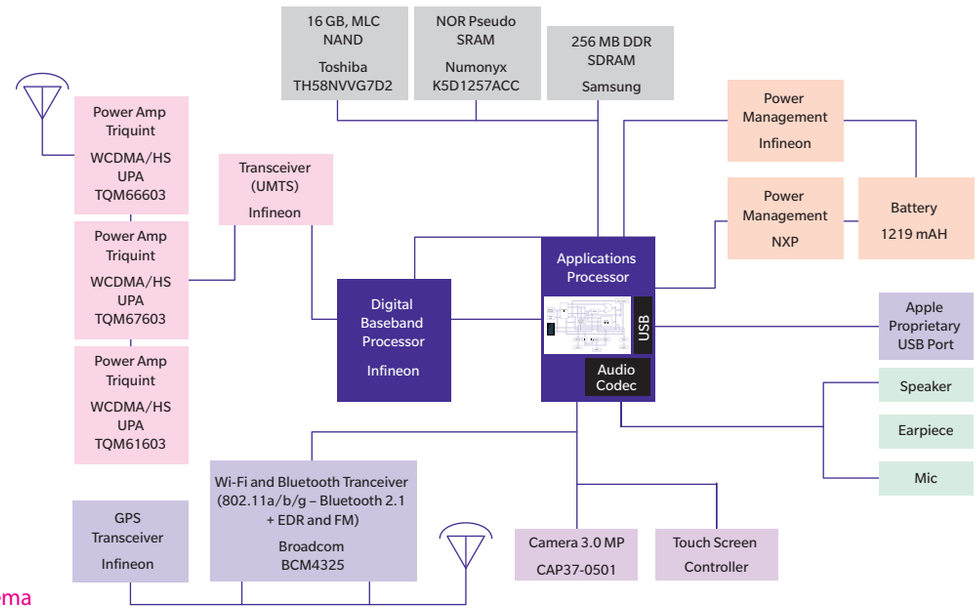
Apple iPhone 4 – Front
1 Skyworks SKY77541 GSM/GRPS Font End Module
2 Triquint TQM666092 Power Amp
3 Skyworks SKY77452 W-CDMA FEM
4 Triquint TQM676091 Power Amp
5 Apple 338S0626 Infineon GSM/W-CDMA Transciever
6 Skyworks SKY77459 Tx-Rx FEM for Quad-Band GSM / GPRS / EDGE
7 Apple AGD1 STMicro 3-axis digital gyroscope
8 Apple A4 Processor
9 Broadcom BCM4329FKUBG 802.11n with Bluetooth 2.1 + EDR and FM receiver
10 Broadcom BCM4750IUBB single-chip GPS receiver



Figur 2.2 iPhone 4 innmat

Hva vil åpenbare seg hvis vi åpner iPhone 4? Det kan gjøres enten fysisk, men da ryker garantien, eller ved å trykke appen Settings nede i høyre hjørnet på telefonen i figur 2.1. Vi velger appen. Da får vi opp et bilde, vist i figur 2.2, av smarttelefonens indre kvaliteter.

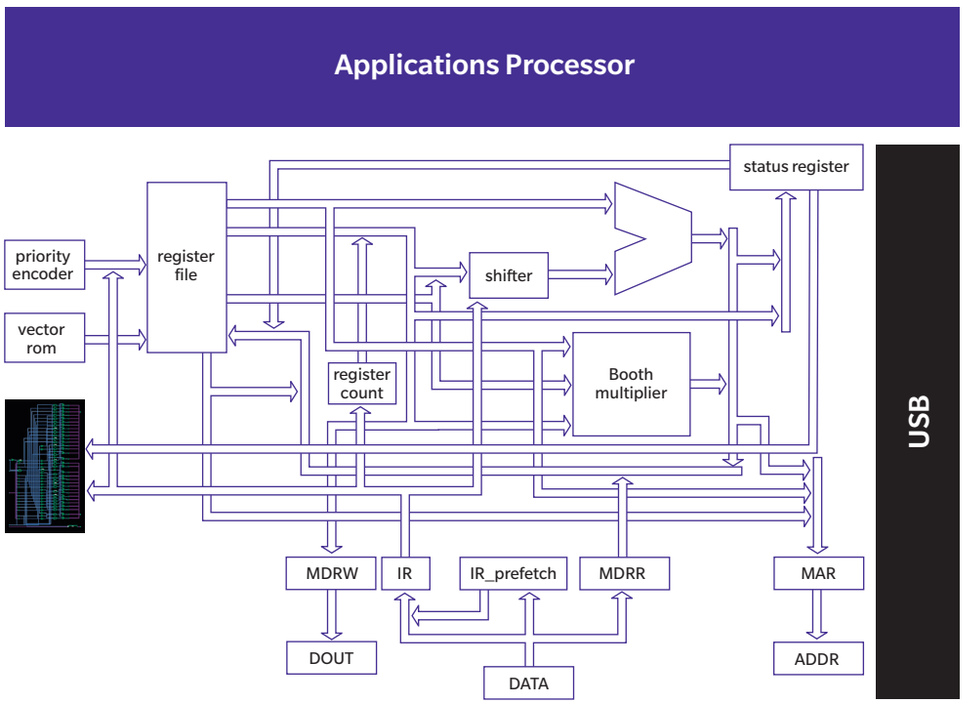
Innmaten til iPhone 4 består av en rekke moduler som gjør ulike ting. Vi finner telefonmoduler med GSM, GPRS, EDGE og W-CDMA, en forsterker, et digitalt gyroskop, 802.11 WiFi med blåtann, FM mottaker, GPS mottaker, diverse antenner og Apples A4 prosessoren som er selve hjernen til iPhonen. Det er i det hele tatt mye funksjonalitet på en liten flate. Om du ikke skulle kjenne alle de oppramsede forkortelser og begreper like godt, så fortvil ikke. Hensikten med denne dekonstruksjonen er å vise hva en kan få til med digitalteknikk. For å forstå hvordan de ulike delene samspiller, er det kjekt med et blokkskjema som fokuserer på funksjonalitet. Figur 2.3 viser et slikt forenklet blokkskjema.



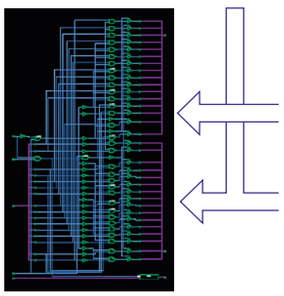
Figur 2.3 Forenklet blokkskjema

I blokkskjemaet har ulike funksjoner fått ulike farger. En har moduler for minne, strømforsyning, lyd, kontroller for berøringsskjerm, kamera, prosessorer for applikasjoner og signalbehandling og sist, men ikke minst en telefonidel.

I hjertet av iPhone 4 finner man applikasjonsprosessoren, figur 2.4, og den kan også beskrives med et blokkskjema.



Figur 2.4 Applikasjonsprosessoren



Figur 2.5 Vårt først møte med NAND-portene

Inne i applikasjonsprosessoren er det en rekke blokker som utfører ulike operasjoner for å kunne kjøre applikasjonsprogrammer. Oppe i høyre hjørne er det en navnløs blokk som ser ut som en pil. Det er den aritmetisk logiske enheten som er selve hjernen i prosessoren, og som utfører regne- og logiske operasjoner. Hvordan prosessorer virker, skal vi se nærmere på i kapittelet «Følg instruksen!». Nederst i venstre hjørne har vi gleden av å se innmaten i en blokk, og vi begynner å nærme oss NAND-portene med stormskritt.

Figur 2.5 viser en logisk krets bygd opp av porter. En logisk krets kan motta et sett med binære verdier 0 og 1, behandle disse og levere et resultat av 0 og 1 som er en

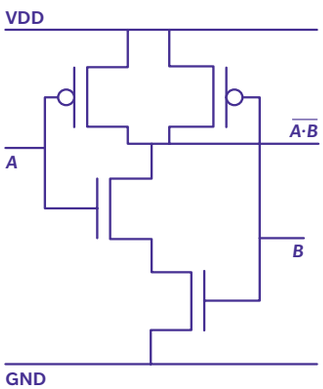
funksjon av det som er mottatt. Portene utfører boolske operasjoner på 0 og 1, og boolske funksjoner skal vi se nærmere på i slutten av neste kapittel.

Arbeidshesten i nesten all elektronikk er den såkalte NAND-porten, fordi alle logiske operasjoner en kan ønske seg, som NOT, AND, OR og deres kombinasjoner, kan avledes fra den. Symbolet for en NAND-port er gitt i figur 2.6, og den tilhørende sannhetstabell vises i tabell 2.1. Sannhetstabellen viser alle mulige inngangsverdier og tilhørende resultater. For eksempel viser første linje av sannhetstabellen at dersom A og B begge er 0, vil det som kommer ut være 1. Sannhetstabellen beskriver den boolske funksjonen. I dette tilfellet for NAND-porten.

Figur 2.6 Symbol for NAND-porten

Tabell 2.1 Sannhetstabell for NAND

A	B	$\overline{AB}$
0	0	1
0	1	1
1	0	1
1	1	0

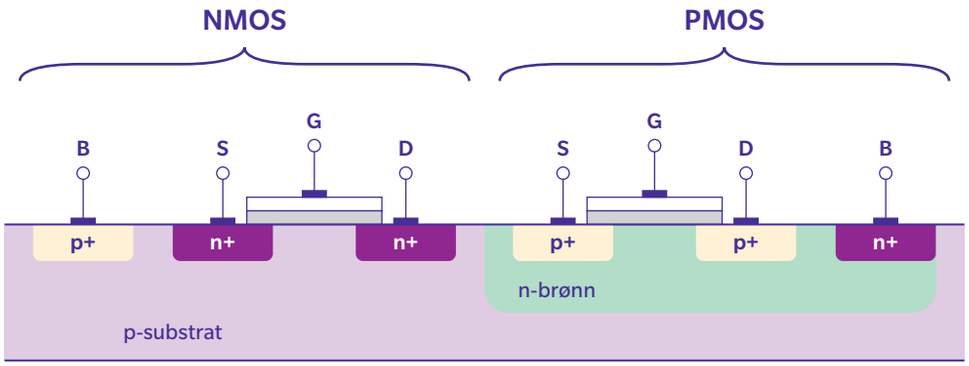


Figur 2.7 CMOS NAND

Symbolet for NAND er sammensatt av et AND-symbol og et NOT-symbol. NOT-symbolet er den lille rundingen. Sannhetstabellen viser hva som kommer ut av porten $\overline{AB}$ for ulike verdier av A og B. Streken over AB forteller oss at en har tatt negasjonen (invertert) av en AND-port AB. Frykt ikke dersom dette kan virke litt uforståelig. Det er kun en smakebit. Vi skal senere i kapittelet «En del elementære logisk emner» foreta et dypdykk i virkemåten til porter og boolsk algebra.

Hvordan lages NAND-porter? Med transistorer. Figur 2.7 gir en skjematisk fremstilling.

De vertikale parallelle strekene er fire transistorer. De to øverste har rundinger på seg for å fortelle oss at de er av typen PMOS. De to andre er NMOS. Teknologi som blander disse, kalles komplementær MOS (CMOS). CMOS er den dominerende teknologi i digitalteknikk. VDD er forsyningsspenning og GND jord. MOS står for metalloksyd



Figur 2.8 NMOS + PMOS = CMOS

felteffekt transistor, og P og N står for hvilke dopingegenskaper substratet transistorene er bygd på har. Figur 2.8 viser PMOS og NMOS i lykkelig CMOS forening.

Dermed er vi kommet til bunns i saken og har funnet våre NAND-porter. Du ser med all ønskelig tydelighet at dersom du lærer deg grunnleggende digitalteknikk, så er alt mulig. Dersom du ønsker å vite mer om den fysiske realiseringen av kretser, kan du kanskje allerede nå ta en titt i kapittelet «Store tanker gir små kretser» i appendiks eller «Elektronikkens DNA» i *Grunnleggende elektroteknikk – kort og godt fortalt* [13] side 247. Den 29. desember 1959 holdt den amerikanske fysikeren Richard Feynman et foredrag med den visjonære tittelen «There's plenty of room at the bottom» hvor han foreslo at en skulle lage ting ved å manipulere atomer direkte. Vel, vi er snart der, og det er bare å delta med liv og lyst.

3

Kapittel 3

Det er mengden det kommer an på

«Så var det barberen som barberer alle som ikke barberer seg selv. Barberer barbereren seg selv?»

Russels paradoks (1901), Bertrand Russell (1872–1970)

LÆRINGSUTBYTTE: Mengder, medlemskap, den tomme mengde $\varnothing$, universalmengden, komplementærmengden, delmengder, operasjoner på mengder, den boolske mengden

«Du kan bare glemme å få hjelp av meg!» Min far hadde vært på orienteringsmøte om moderne matematikk som vi, de søte små, ble utsatt for på 1970-tallet. Med disse fyndord meldte han seg ut som medlem av mengden av de som lærte seg mengdelære. Min kommende kone og jeg ble overlatt til oss selv og skred til verket med liv og lyst. Det hemmelige matematiske brorskapet Bourbaki førte oss sammen, og ordet moderne gjorde at vi ikke engang spurte etter nytten. Først tiår senere ble jeg oppmerksom på at den moderne matematikken er grunnlaget for all digitalteknikk.

Hva er en mengde? Det er en samling av elementer. Det kan være tall, norske konger, nye mengder eller andre ting. En vanlig måte å skrive mengder på er på listeform.

$$A = \{2, 4, 6, 8\}$$

Mengden A er her alle positive partall under 10. Antallet elementer i en mengde kalles kardinalitet. For mengden A er kardinaliteten $n(A) = 4$. Mengden av alle naturlige tall kan skrives på listeform som $\mathsf{N} = \{1, 2, 3, 4, 5, \ldots\}$ hvor prikkene gir et signal om at det fortsetter i det uendelige. At et element er medlem av en mengde, oppgis ved $x \in A$ hvor x er elementet og A er mengden. I vårt eksempel med positive partall under 10 er $x = 2$ et element, mens $x = 10$ ikke er det. At x ikke er et element i mengden A, angis på følgende måte: $x \notin A$.

En mengde kan også defineres med en slik-at-definisjon. Da starter man med en bestemt mengde, og så danner man en ny mengde ved å plukke ut de elementer som oppfyller en bestemt egenskap. Et eksempel i så henseende er:

$$B = \{x \in \mathsf{N} \mid x \text{ er delelig på } 2\} = \{2, 4, 6, 8, 10, \ldots\}$$

B er mengden partall, og N er mengden av alle naturlige tall $\{1, 2, 3, 4, 5, \ldots\}$.

Mengden som ikke inneholder noen elementer, kalles den tomme mengden Ø.

$$Ø = \{\}$$

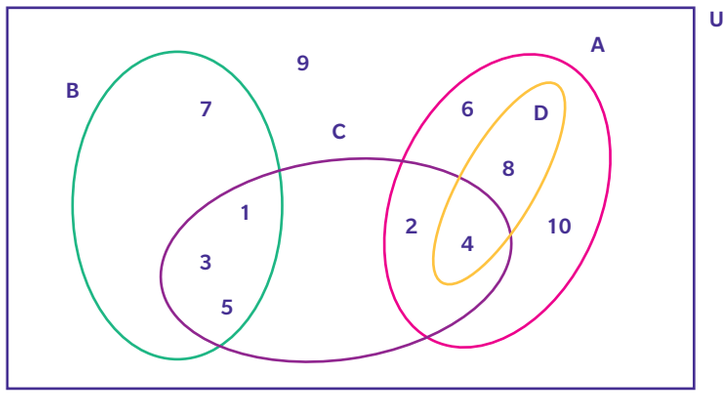
En grei måte å illustrere mengder og forhold mellom mengder på er såkalte Venn-diagram. Riktignok ble jeg venn med min kone over slike diagrammer, men betegnelsen Venn-diagram kommer fra logikeren John Venn. Typisk starter en med å tegne et rektangel som skal symbolisere universalmengden (også kalt grunnmengden). Figur 3.1 viser en universalmengde U som består av alle positive heltall fra 1 til 10. I tillegg har en fire mengder A, B, C og D definert som:

$$A = \{x \in U \mid x \text{ er partall}\} = \{2,4,6,8,10\}$$

$$B = \{x \in U \mid x \text{ er oddetall mindre enn } 9\} = \{1,3,5,7\}$$

$$C = \{x \in U \mid x \leq 5\} = \{1,2,3,4,5\}$$

$$D = \{x \in U \mid x \text{ er delelig på } 4\} = \{4,8\}$$

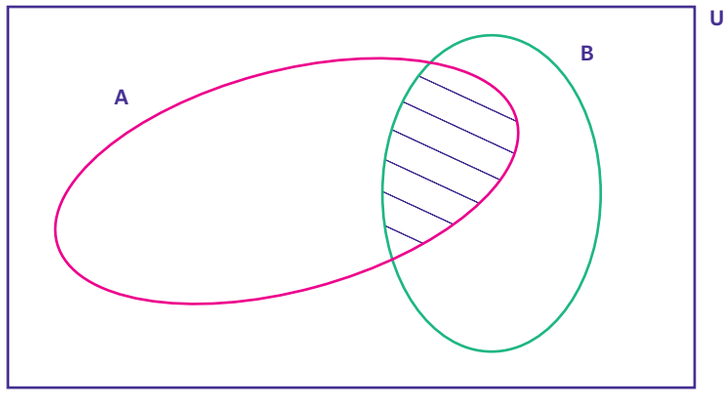


Figur 3.1 Vår universalmengde U med de fire mengdene A, B, C og D

Komplementærmengden til en mengde er alt som er med i universalmengden, men som ikke er med i mengden selv. I vårt tilfelle er for eksempel komplementærmengden til D lik $\overline{D} = \{1,2,3,5,6,7,9,10\}$. For å vise at det er komplementærmengden til D det er snakk om, setter man en strek $\overline{D}$. Komplementærmengder skal vi få stor glede av senere når vi begynner å se på logikk og logiske kretser.

Dersom en mengde er fullstendig inneholdt i en annen mengde, kalles den en delmengde. I vårt tilfelle er for eksempel D en delmengde av A, og det angis på følgende form $D \subseteq A$. Ingen av de andre mengdene A, B og C er delmengde av hverandre.

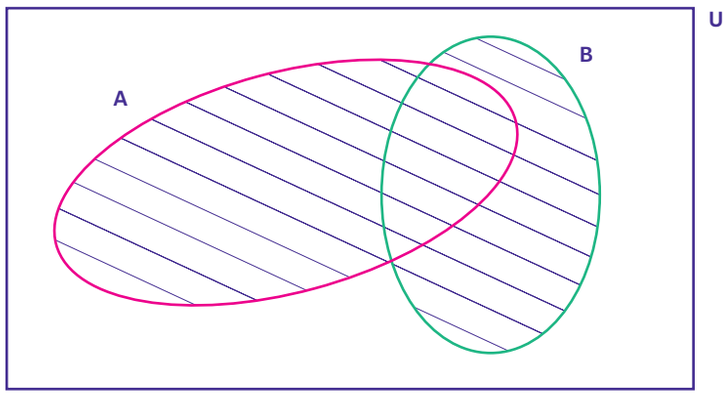
Det går an å definere operasjoner på mengder – såkalt mengdealgebra. To mengder A og B kan kombineres på ulike måter. Snittet mellom A og B betegnet $A \cap B$ består av alle elementer x som både er med i A og B.



Figur 3.2 Snittet **A** ∩ **B** er skravert

Dersom vi ser på mengdene vi hadde i figur 3.1, så er $A \cap B = \emptyset$. Det betyr at A og B ikke har noen felles elementer, og de kalles da disjunkte mengder. Videre er $B \cap C = \{1,3,5\}$, $C \cap D = \{4\}$ og $A \cap D = \{4,8\}$ for å nevne noen.

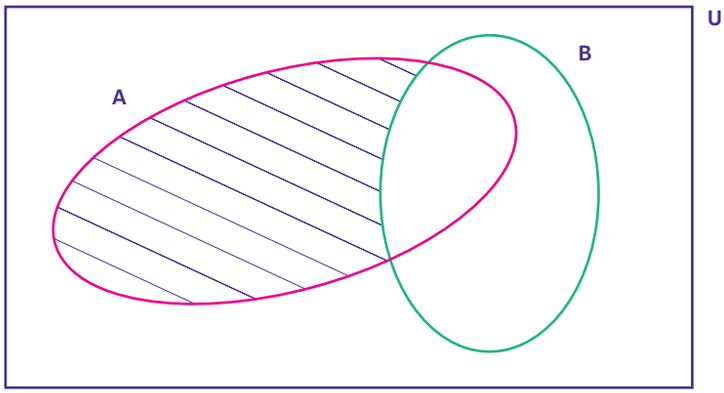
Unionen mellom A og B betegnet $A \cup B$ består av alle elementer x som er med i minst én av mengdene A og B.



Figur 3.3 Unionen **A** ∪ **B** er skravert

Ved igjen å se på mengdene vi hadde i figur 3.1, så er $A \cup B = \{1,2,3,4,5,6,7,8,10\}$, $B \cup C = \{1,2,3,4,5,7\}$, $C \cup D = \{1,2,3,4,5,8\}$ og $A \cup D = A = \{2,4,6,8,10\}$.

Differansen mellom to mengder $A \setminus B$ består av alle elementer i A som ikke er elementer i B.



Figur 3.4 Differansen **A \ B** er skravert

For mengdene i figur 3.1 gir det $A \setminus B = A = \{2,4,6,8,10\}$, $B \setminus C = \{7\}$, $C \setminus D = \{1,2,3,5\}$ og $A \setminus D = \{2,6,10\}$.

Med disse definisjonene av snitt og union kan en konstruere en mengdealgebra.

Kommutative lover:

$$A \cup B = B \cup A$$

$$A \cap B = B \cap A$$

Assosiative lover:

$$(A \cup B) \cup C = A \cup (B \cup C)$$

$$(A \cap B) \cap C = A \cap (B \cap C)$$

Distributive lover:

$$A \cup (B \cap C) = (A \cup B) \cap (A \cup C)$$

$$A \cap (B \cup C) = (A \cap B) \cup (A \cap C)$$

De fleste av disse lovene sier seg selv, mens en må finmyse litt for å akseptere andre. La oss ta en nærmere titt på den siste distributive loven.

Vis at $A \cap (B \cup C) = (A \cap B) \cup (A \cap C)$. En starter da med å vise at $A \cap (B \cup C)$ er en delmengde av $(A \cap B) \cup (A \cap C)$.

$$A \cap (B \cup C) \subseteq (A \cap B) \cup (A \cap C)$$

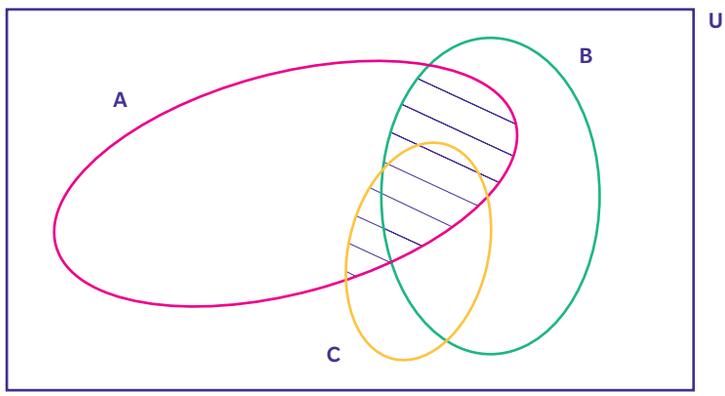
En starter med et element på venstre side, $x \in A \cap (B \cup C)$. Da vet en at $x \in A$ og samtidig at $x \in B \cup C$. Det siste betyr at $x \in B$ eller $x \in C$. Hvis $x \in B$, er også $x \in A \cap B$, og hvis $x \in C$,er også $x \in A \cap C$. I begge tilfeller vil $x \in (A \cap B) \cup (A \cap C)$.

Da gjenstår det å vise også det motsatte.

$$(A \cap B) \cup (A \cap C) \subseteq A \cap (B \cup C)$$

Igjen starter en med et element på venstre side, $x \in (A \cap B) \cup (A \cap C)$. Da har en to muligheter, enten er $x \in A \cap B$ eller $x \in A \cap C$. I begge tilfeller er $x \in A$. I tillegg har vi i det første tilfellet at $x \in B$, og i det andre tilfellet at $x \in C$. Derfor har en uansett at $x \in B \cup C$. Da både $A \cap (B \cup C)$ og $(A \cap B) \cup (A \cap C)$ er delmengder av hverandre, betyr det at de er like, og dermed har en bevist den siste distributive loven.

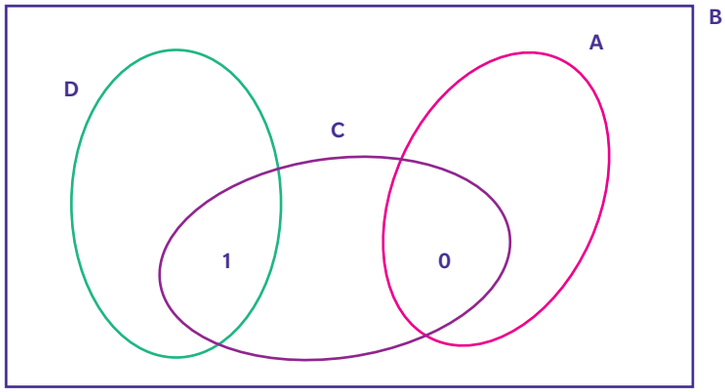
Dette beviset kan også illustreres med Venn-diagram. Det er bare å overbevise seg selv om at det skraverte feltet i figur 3.5 er likt $A \cap (B \cup C) = (A \cap B) \cup (A \cap C)$.



Figur 3.5 Mengden **A**∩(**B**∪**C**)=(**A**∩**B**)∪(**A**∩**C**) er skravert

Dette minner ikke mye om digitalteknikk. Hvordan kan vi bruke mengdelære på en fornuftig måte i digitalteknikken? Ved å gå til en av de enkleste av alle mengder – den boolske mengde $\boldsymbol{B}=\{0,1\}$. Som en ser, består denne boolske universalmengden **B** kun

av to elementer (verdier), nemlig 0 og 1. En boolsk mengde er en mengde som kun består av to elementer som kan defineres som SANT og USANT. I kapittelet «En del elementære logiske emner» skal vi se hvordan vi kan lage en algebra (regneregler) for den boolske mengden.



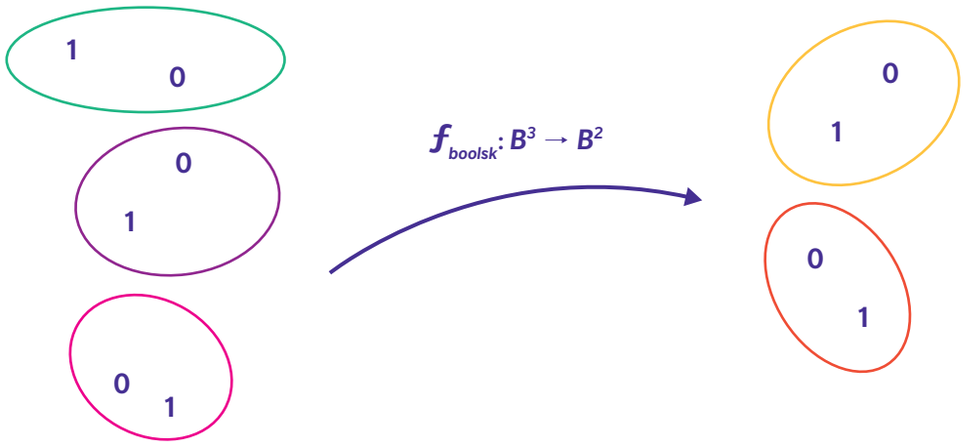
Figur 3.6 Den boolske mengden **B** med sine undermengder **A** , **C** og **D**

Dette var jo fascinerende lite å ta stilling til. Kun tre mulige delmengder og noen få komplementærmengder $\overline{A} = D = \{1\}, \overline{D} = A = \{0\}$ og $\overline{C} = \emptyset$. $\boldsymbol{B}$={0,1} er den grunnleggende byggeklossen i all digitalteknikk. Denne lille boolske mengden med tilhørende boolsk algebra kan brukes til å konstruere logiske kretser hvor 1 representerer SANT og 0 USANT. Typisk brukes transistorer eller dioders evne til å lede eller ikke lede for å angi 0 eller 1. Typisk blir 0 gitt ved en spenning på 0 volt og 1 som en spenning på enten 1,8 volt, 3,3 volt eller 5 volt avhengig av hvordan den logiske kretsen er konstruert.

Når sant skal sies, så er logiske kretser funksjoner på boolske mengder. Disse funksjonene henter verdier fra en eller flere boolske mengder og leverer resultatet til en eller flere boolske mengder. En generell boolsk funksjon er gitt ved:

$$f_{boolsk} : \boldsymbol{B}^k \rightarrow \boldsymbol{B}^l \ \ hvor \ \boldsymbol{B} = \{0,1\}$$

Denne funksjonen tar et sammensatt utsagn med k boolske variabler og leverer en funksjonsverdi med l boolske verdier.



Figur 3.7 Et eksempel på en boolsk funksjon med tre variabler og to resultatverdier

Den boolske funksjonen kan sammenlignes med en liten maskin som spiser k boolske verdier, behandler dem og leverer l boolske funksjonsverdier. Den boolske funksjonen beskrives ved hjelp av en sannhetstabell hvor alle mulige inngangsverdier har tilhørende utgangsverdier. Det skal vi se nærmere på i kapittelet «En del elementære logiske emner». Med denne innledende øvelsen i mengdelære er vi klare til å begi oss inn i de logiske kretsers vakre verden.

4

Kapittel 4

Tenk på et tall

«Alt er tall!»

Pythagoras? (~580– ~500)

LÆRINGSUTBYTTE: Tallsystemer, det desimale tallsystemet, binære tall, base, konvertering fra en base til en annen, oktalt og heksadesimalt tallsystem, algebra og aritmetikk, komplement, binære tall med fortegn, overflyt, flyttall, BCD (Binary Coded Decimal), Gray-kode, ASCII-kode

«Mengder er det mest grunnleggende i matematikken. Egentlig burde vi hatt mengdelære før vi fikk vite noe som helst om tallene.» Min venn matematikeren fortsetter ivrig: «Når en spør et barn om det har like mange fingre på begge hender, kan den lille enten telle eller direkte sammenligne fingermengdene ved å holde fingrene på høyre og venstre hånd mot hverandre. Begge metoder vil få barnet til å utbryte 'ja', men mengdemetoden setter ingen krav til tellekompetanse.» Jeg tenker i mitt stille sinn mens jeg ser ned i hendene mine at selv om mengder er vel og bra, kan det kanskje bli litt abstrakt for en seksåring.

Jeg husker ikke når jeg lærte å telle, men det var lenge før jeg begynte på skolen. Min mor som drev og sydde, ga meg ofte et meterbånd å leke med for å få fred i arbeidet. Dermed trodde jeg lenge at tallene stoppet med 150. Da jeg begynte på skolen, ble jeg innviet i tallenes magiske verden. Vi lærte forskjellen mellom 1-ere og 10-ere ved å bunte sammen fyrstikker av ulik størrelse med gummistrikk. 83 besto av en bunke med åtte tjukke fyrstikker og en bunke med tre tynne fyrstikker. Det jeg lærte, var posisjonssystemet. Senere i livet har jeg lært å posisjonere meg. Det desimale tallsystemet (titallsystemet) har 10 som base og bruker de velkjente siffer fra 0 til 9. 2457 kan for eksempel skriver som:

$$2457 = 2 \cdot 10^3 + 4 \cdot 10^2 + 5 \cdot 10^1 + 7 \cdot 10^0 = 2 \cdot 1000 + 4 \cdot 100 + 5 \cdot 10 + 7$$

Det vi skal få gleden av å gjøre i dette kapittelet, er aritmetiske øvelser (tallregning) på mengden av reelle tall. Denne tallregningen følger de algebraiske regler for +, –, / og · som vi lærte på barneskolen. Det som kanskje er nytt, er at vi skal bruke ulike tallsystemer.

At det finnes ulike tallsystemer, lærte jeg i historietimene. Babylonerne hadde sekstitallsystem basert på årets lengde som de antok var 360 dager, Mayafolket et tyvetallsystem, Maoriene på New Zealand et ellevetallsystem, og danskene har rester av et tyvetallsystem som gjør enhver ferie i Danmark til en tallmessig fornøyelse. Vi har det desimale tallsystemet av den enkle grunn at vi er utstyrt med ti fingre. At datamaskinens binære system (totallsystem) med kun sifrene 0 og 1 er et resultat av at maskinen er bygd opp av logiske kretser med kun to tilstander, lærte jeg ingen ting om, men det var kanskje for nytt for historielæreren?

Det binære tallsystem har base 2. Dersom en bruker bokstaven B til å representere tallets base, vil en ha:

$$10 = 1 \cdot B^1 + 0 \cdot B^0$$

I det binære tallsystemet er $B = 2$, og 10 binært vil være det samme som 2 desimalt. Dersom en ønsker å få det inn med teskje, kan en for sikkerhets skyld skrive:

$$(10)_2 = (2)_{10}$$

Her forteller tallet utenfor parentesen hvilket tallsystem tallet innenfor parentesen er presentert i.

Hva så med et større binært tall?

$$(1011)_2 = (1\cdot 2^3 + 0\cdot 2^2 + 1\cdot 2^1 + 1\cdot 2^0)_{10} = (1\cdot 8 + 0\cdot 4 + 1\cdot 2 + 1\cdot 1)_{10} = (11)_{10}$$

Generelt kan et positivt helt tall i et hvilket som helst tallsystem med base B skrives på følgende form:

$$\ldots s_3 s_2 s_1 s_0 = \ldots + s_3 \cdot B^3 + s_2 \cdot B^2 + s_1 \cdot B^1 + s_0 \cdot B^0$$

Hvordan kan en så gå fra det ene til det andre? Vi har allerede hatt et eksempel på å gå fra det binære tallsystemet til det desimale, $(1011)_2 = (11)_{10}$. La oss prøve å gjøre det motsatte med et annet eksempel.

$$2175 = 2\cdot 10^3 + 1\cdot 10^2 + 7\cdot 10^1 + 5\cdot 10^0 = \sum_{i=N}^{0} s_n 2^n$$

Helt til høyre har vi skrevet tallet 2175 som en sum av potenser av 2. For å finne den binære ekvivalenten til 2175 må vi bare bestemme hvilke s_n som er 0 eller 1.

$$2175 = s_{11}\cdot 2048 + s_{10}\cdot 1024 + s_9\cdot 512 + s_8\cdot 256 + s_7\cdot 128 + s_6\cdot 64 + s_5\cdot 32 + s_4\cdot 16 + s_3\cdot 8 + s_2\cdot 4 + s_1\cdot 2 + s_0\cdot 1$$

Her har vi startet med $2^{11} = 2048$ fordi $2^{12} = 4096$ er større enn tallet 2175 som skal konverteres. Her ser vi lett hvilke s_n som må være henholdsvis null og en. Nei, det gjør vi ikke, men det finnes heldigvis en prosedyre som produserer svaret lekende lett. Begynn med det opprinnelige tallet og heltallsdivider suksessivt på 2 og skriv opp koeffisient og rest. Den første resten vil representere s_0. s_0 i binær sammenheng kalles «Least Significant Bit» (LSB). De neste restene er $s_1, s_2 \ldots s_n$. Avslutt prosessen når koeffisienten er null.

Tabell 4.1 Konvertering fra desimal- til binærtall

Heltallsdivisjon	Koeffisient	Rest	s_n
$\frac{2175}{2}$	1087	1	s_0
$\frac{1087}{2}$	543	1	s_1
$\frac{543}{2}$	271	1	s_2
$\frac{271}{2}$	135	1	s_3
$\frac{135}{2}$	67	1	s_4
$\frac{67}{2}$	33	1	s_5
$\frac{33}{2}$	16	1	s_6
$\frac{16}{2}$	8	0	s_7
$\frac{8}{2}$	4	0	s_8
$\frac{4}{2}$	2	0	s_9
$\frac{2}{2}$	1	0	s_{10}
$\frac{1}{2}$	0	1	s_{11}

Altså er $(2175)_{10} = (10000111111)_2$. Siden det er først gang vi gjør dette, så la oss sjekke resultatet med innsetting.

$$2175 = 1\cdot 2048 + 0\cdot 1024 + 0\cdot 512 + 0\cdot 256 + 0\cdot 128 + 1\cdot 64 + 1\cdot 32 + 1\cdot 16 + 1\cdot 8 + 1\cdot 4$$
$$+1\cdot 2 + 1\cdot 1 = 2048 + 64 + 32 + 16 + 8 + 4 + 2 + 1 = 2175$$

Strålende! Det stemmer, og vi har funnet en enkel måte å gå fra desimaltall til binære tall på.

Det var de naturlige tallene. Hva med kommatall? De kan skrives på følgende vis:

$$\ldots s_2 s_1 s_0, s_{-1} s_{-2} s_{-3} \ldots = \ldots + s_2 \cdot B^2 + s_1 \cdot B^1 + s_0 \cdot B^0 + s_{-1} \cdot B^{-1} + s_{-2} \cdot B^{-2} + s_{-3} \cdot B^{-3}$$

La oss gjøre om $(0{,}1011)_2$ til en desimalverdi.

$$(0{,}1011)_2 = 0 \cdot 2^0 + 1 \cdot 2^{-1} + 0 \cdot 2^{-2} + 1 \cdot 2^{-3} + 1 \cdot 2^{-4} =$$
$$0 \cdot 1 + 1 \cdot 0{,}5 + 0 \cdot 0{,}25 + 1 \cdot 0{,}125 + 1 \cdot 0{,}0625 = (0{,}6875)_{10}$$

For å gå den motsatte vei, fra et desimaltall til et binært tall, kan vi for sifrene etter komma bruke en avart av metoden vi brukte ovenfor hvor vi delte suksessivt med 2. Nå må vi derimot multiplisere med 2 og bruke det overskytende som $s_{-1}, s_{-2} \ldots s_{-n}$.

Tabell 4.2 Konvertering fra desimal- til binærtall

Multiplikasjon	Overskytende	Rest	s_n
$0{,}6875 \cdot 2 = 1{,}3750$	1	0,3750	s_{-1}
$0{,}3750 \cdot 2 = 0{,}7500$	0	0,7500	s_{-2}
$0{,}7500 \cdot 2 = 1{,}5000$	1	0,5000	s_{-3}
$0{,}5000 \cdot 2 = 1{,}0000$	1	0	s_{-4}

Vi ser at resultatet stemmer, da vi har kommet tilbake til utgangspunktet:

$$(0{,}6875)_{10} = 1 \cdot 2^{-1} + 0 \cdot 2^{-2} + 1 \cdot 2^{-3} + 1 \cdot 2^{-4} = (0{,}1011)_2$$

Det er ikke alltid slik at frem og tilbake gjør at en kommer tilbake til utgangspunktet. På samme måten som et desimaltall ikke behøver å ha noen eksakt representasjon, som for eksempel $\frac{2}{3}$, vil et desimaltall mellom 0 og 1 som ikke er en sum av 2^{-n} ledd, bare tilnærmet være lik et binært tall.

La oss finne den binære presentasjonen av $(0{,}42357)_{10}$.

Tabell 4.3 Konvertering fra desimal- til binærtall

Multiplikasjon	Overskytende	Rest	s_n
$0,42357 \cdot 2 = 0,84714$	0	$0,84714$	s_{-1}
$0,84714 \cdot 2 = 1,69428$	1	$0,69428$	s_{-2}
$0,69428 \cdot 2 = 1,38856$	1	$0,38856$	s_{-3}
$0,38856 \cdot 2 = 0,77712$	0	$0,77712$	s_{-4}
$0,77712 \cdot 2 = 1,55424$	1	$0,55424$	s_{-5}
$0,55424 \cdot 2 = \ldots$	$\cdots$	$\cdots$	s_{-6}

Dermed er $(0,42357)_{10} = (0,01101\ldots)_2$. Vi kan kontrollere ved å gå andre veien:

$$(0,01101\ldots)_2 = 0 \cdot 2^0 + 0 \cdot 2^{-1} + 1 \cdot 2^{-2} + 1 \cdot 2^{-3} + 0 \cdot 2^{-4} + 1 \cdot 2^{-5} = 0,25 + 0,125 + 0,03125 = (0,40652)_{10}$$

Ooops! Dette var ikke akkurat det samme som $0,42357$, og det skyldes det enkle faktum at vi har neglisjert resten $0,55254$ i Tabell 4.3. Denne resten multiplisert med den minste beregnede verdi gir den totale feilen:

$$(0,42357)_{10} - (0,40652)_{10} = (0,55254 \cdot 0,03125)_{10} = (0,01732)_{10}$$

Jo flere sifre vi hadde orket å regne ut etter komma i $(0,01101\ldots)_2$, jo mindre hadde feilen blitt, men helt borte hadde den aldri blitt, da $(0,42357)_{10}$ ikke går opp i en sum av 2^{-n}.

Andre tallsystemer

Er det andre tallsystemer som det kan være kjekt å kjenne til? Vel, en av ulempene med det binære tallsystemet er at tallene har en tendens til å bli meget lange. En måte å bøte på det på er å gruppere tallet i grupper på tre eller fire. La oss se på tallet:

110111101100

Ved å gruppere det i grupper på tre får en:

$$\begin{array}{cccc} 110 & 111 & 101 & 100 \\ 6 & 7 & 5 & 4 \end{array} = (6754)_8$$

Dette tallsystemet med base 8 kalles oktalt og består av tallene $0,1,2,3,4,5,6,7$. Sjarmen med det oktale tallsystemet er at det blir mer kompakte tall, og at det er lett å regne fra binært til oktalt og vice versa.

Enda mer hendig er det heksadesimale tallsystemet med base 16 hvor en grupperer binære tall i grupper på fire. Grunnen til det er at i datamaskiner lagres tall i såkalte ord som er multipler av byte som er binære tall som har lengde på åtte. Innholdet i en byte kan da representeres av et heksadesimalt tall med to siffer.

La oss igjen ta utgangspunkt i det binære tallet:

$$110111101100$$

Ved å gruppere det i grupper på fire får en:

$$\begin{array}{cccc} 1101 & 1110 & 1100 \\ D & E & C \end{array} = (DEC)_{16}$$

Det heksadesimale tallsystemet omfatter tallene. $0,1,2,3,4,5,6,7,8,9,A,B,C,D,E,F$. For tallene over 9 har en brukt bokstaver. A er desimalt 10, og så har en til slutt F som er det samme som desimalt 15. Tallet DEC vekker gode minner om mitt første møte med en datamaskin, nemlig Digital Equipment Corporations DEC PDP-10 med det flotte operativsystemet TOPS-20. Tilfeldig? Neppe!

Tabell 4.4 Tallsystemer

Heksadesimalt	Desimalt	Oktalt	Binært
0	0	0	0000
1	1	1	0001
2	2	2	0010
3	3	3	0011
4	4	4	0100
5	5	5	0101
6	6	6	0110
7	7	7	0111
8	8	10	1000
9	9	11	1001
A	10	12	1010
B	11	13	1011
C	12	14	1100
D	13	15	1101
E	14	16	1110
F	15	17	1111

Regning med binære tall

Etter at vi hadde lært tallene å kjenne på barneskolen, begynte vi å regne. Vi startet med det enkleste som var å plusse (addere). Senere utvidet vi repertoaret med å trekke fra (subtrahere) og samtidig oppdage de negative tall. Det var vel i andre eller tredje klasse vi tok det helt ut med ganging (multiplikasjon), den lille multiplikasjonstabellen og deling (divisjon). Divisjonen ga tall med komma, og plutselig lå hele den matematiske verden åpen for oss. At vi hadde drevet med aritmetiske øvelser, fikk vi først vite da vi begynte å regne med bokstaver (algebra) istedenfor tall (aritmetikk). Først på gymnaset lærte jeg at regnereglene for pluss, minus, ganging og deling sammen med mengden av reelle tall definerte en algebra som angitt i tabell 4.5.

Tabell 4.5 Algebraiske regler på de reelle tall

Operasjon	Uttrykt	Kommutativ	Assosiativ	Identitets-element	Invers funksjon
Addisjon	$a+b$	$a+b=b+a$	$(a+b)+c=a+(b+c)$	0	Subtraksjon (–)
Multiplikasjon	$a\cdot b$	$a\cdot b=b\cdot a$	$(a\cdot b)\cdot c=a\cdot(b\cdot c)$	1	Divisjon (/)

Så hvorfor i herrens navn skal vi nå lære å regne med binære tall? Holder det ikke å mestre kunstarten i et tallsystem? Kanskje, men da datamaskiner regner med binære tall, er det vel verdt å lære seg hvordan de opererer.

La oss følge mønsteret fra barneskolen og starte vår binære aritmetikk med addisjon. Samtidig sammenligner vi med ordinær desimal addisjon:

Desimal addisjon	Binær addisjon	
	11	← Mente
3	0011	
+6	+0110	
= 9	=1001	

Så kjekt! Det ser ut som om desimal addisjon og binær addisjon er ekvivalente. La oss utfordre skjebnen med et litt mer avansert eksempel:

Desimal addisjon	Binær addisjon	
11	11 1	← Mente
208	11010000	
+ 92	+ 1011100	
= 300	=100101100	

La oss sjekke resultatet:

$$(100101100)_2 = 1\cdot 256+0\cdot 128+0\cdot 64+1\cdot 32+0\cdot 16+1\cdot 8+1\cdot 4+0\cdot 2+0\cdot 1 = 256+32+8+4 = (300)_{10}$$

Det stemmer, og det er selvfølgelig ikke så merkelig, da det tross alt bare er representasjonen av tallene som er forskjellig, og ikke selve regneregelen.

Hva så med subtraksjon? Det burde vel gå like bra? La oss starte enkelt:

Desimal subtraksjon	Binær subtraksjon	
	~~1~~010	← Mente
9	~~1~~001	
–2	–0010	
=7	=0111	

Suksess igjen. Obs! Tierne i mente for binær subtraksjon er 2 skrevet i titallsystemet. Hva med to større tall?

Desimal subtraksjon	Binær subtraksjon	
~~1~~010	10	← Mente
~~1~~22	111~~1~~010	
–68	–1000100	
=54	=0110110	

Det går jo som en lek. Det er bare én hake. Datamaskiner kan ikke subtrahere tall, de kan bare addere. Med mengden datamaskiner i omløp og deres evne til alle slags mulige beregninger må det være en alternativ måte for datamaskinene å addere på. Den alternative måten kalles 2ers komplement.

For å forstå 2ers komplement kan det være lurt først å ta en kikk på 10ers komplement. La oss starte med eksempelet vi allerede har vært borte i, nemlig: $9-2=7$. 10ers komplement til 2 er $(10-2)=8$. Den desimale subtraksjonen kan utføres ved å addere 10ers komplement:

Desimal subtraksjon	Desimal subtraksjon med 10ers komplement
9	9
–2	+8
=7	=17

17 er selvfølgelig ikke det samme som 7, men ved å ignorere mente så er siste siffer likt i de to tall. Subtraksjon kan altså utføres med addisjon.

La oss ta en større utfordring:

Desimal subtraksjon	Binær subtraksjon	
~~1~~010		← Mente
~~1~~22	122	
–68	+(100–68)	
=54	=154	

Siden tallet vi nå skulle trekke fra var –68, måtte vi bruke 100 i 10ers kompliment. Hadde tallet vi skulle trekke fra vært tresifret, hadde vi måttet bruke 1000, og på den måten fortsetter det for større tall. Igjen ser vi at vi får det samme resultatet så lenge mente blir ignorert.

Virker denne metoden også når tallet som trekkes fra (subtrahend) er større enn tallet (minuend) det trekkes fra? La oss starte med eksempelet vi er blitt så glad i, 7–9. Hvordan regnet vi det ut på barneskolen forresten? Vi snudde om på rekkefølgen $9-7=2$, og så tok vi det negative av resultatet og fikk -2. At det gjelder generelt, ser en ved at $a-b=x$ blir $b-a=-x$, og at x finnes ved å multiplisere med minus en til slutt.

Desimal subtraksjon	Desimal subtraksjon med 10ers komplement
2	2
–9	$+(10-9)$
$=-7$	$=3$

Det var et stykke unna, men fortvil ikke. Svaret vi har fått ved desimal subtraksjon med 10ers komplement når subtrahend er større minuend, kommer på 10ers komplement form. Det egentlige negative svaret finner vi ved å ta 10ers komplement av resultatet: $-(10-3)=-7$.

Bare for å teste at vi har fått dreisen på det, prøver vi oss med større tall:

Desimal subtraksjon	Desimal subtraksjon med 10ers komplement
68	68
− 122	$+\left(1000-122\right)$
$=-54$	$=946$

Resultatet finner vi igjen ved å ta 10ers komplement $-\left(1000-946\right)=-54$.

Da burde problemet med datamaskinens manglende evne til å subtrahere være løst. Det er bare å gå løs på binær subtraksjon med 2ers komplement. La oss starte stille og rolig med små tall hvor minuend er større enn subtrahend. Subtrahend i dette tilfellet er 01011, og vi må finne 2ers komplement av det tallet.

2ers komplement	
~~1010101~~010	← Mente
~~1~~00000	
−01011	
=10101	

Så kan vi bruke 2ers komplement og addere istedenfor å subtrahere.

	Binær subtraksjon	Binær subtraksjon med 2ers komplement	
	~~1010~~10	1 1	← Mente
Minuend	~~1~~0001	10001	
Subtrahend	−01011	+10101	
Differanse	=00110	=100110	

Vi ser at resultat er likt med vanlig binær subtraksjon dersom vi forkaster øverste mente.

Det finnes en enklere måte å finne 2ers komplement på, både for oss og for datamaskiner, enn den vi har brukt. Selve prosedyren er enkel å utføre, men verre å forstå enn det vi allerede har gjort med 10ers og 2ers komplement. Oppskriften er som følger: Ta såkalt 1ers komplement og så legge til 1. 1ers komplement av et tall får en ved å invertere alle siffer. La oss teste den nye metoden på subtrahenden 01011 vi hadde ovenfor.

Tall	01011
1ers komplement av tall	10100
	+1
2er komplement av tall	= 10101

Vi får det samme som tidligere, men hvorfor? Og ikke minst, gjelder det for alle tall? La meg prøve å forklare. La oss starte med et tall som slutter med 1, for eksempel $XXX1$. X her indikerer at de sifrene enten kan være 0 eller 1. Når vi tar 2ers komplement på den første måten, trekker vi $XXX1$ fra 10000. Sifrene XXX vil bli invertert (0 blir 1 og 1 blir 0) til $\overline{X}\overline{X}\overline{X}$, men ikke siste siffer. Resultatet blir altså $\overline{X}\overline{X}\overline{X}1$ som er det samme som vi ville fått om vi tok 1ers komplement av $XXX1$ og la til 1: $\overline{X}\overline{X}\overline{X}0+1$. Antallet X spiller ingen rolle for resultatet. En må bare sørge for at det en trekker fra, er en potens større. Det andre mulige tilfellet er et tall som slutter med en 1 og så en eller flere 0, for eksempel $XXX100$. 2ers komplement med første metode krever at vi trekker $XXX100$ fra 1000000. Resultatet blir $\overline{X}\overline{X}\overline{X}100$. Ta så 1ers komplement av $XXX100$. Det blir $\overline{X}\overline{X}\overline{X}011$. Ved å legge 1 til $\overline{X}\overline{X}\overline{X}011$ så har en $\overline{X}\overline{X}\overline{X}100$. Samme hvilken metode en velger, så får en 2ers komplement. Fra nå av satser vi på metoden med 1ers komplement pluss 1 i ren dovenskap.

Før vi blir helt fulle av oss selv på grunn av all suksessen, la oss prøve en binær subtraksjon hvor subtrahenden er større enn minuenden. Fra eksempelet med desimal subtraksjon av samme type vet vi at vi har utfordringer (les problemer) i vente.

	Binær subtraksjon	Binær subtraksjon med 2ers komplement
Minuend	101	101
Subtrahend	− 11011	+00101
Differanse	= −10110	= 01010

Med ordinær binær subtraksjon hvor subtrahend er størst, gjør en det samme som i det tilsvarende desimale eksempelet, nemlig trekker minuend fra subtrahend og setter så minus foran resultatet. For metoden med 2ers komplement må en finne 2ers komplement av subtrahend 11011 som er $00100+1=00101$. Med binær subtraksjon med 2ers komplement hvor subtrahend er størst, kommer resultatet i 2ers komplements form. En finner svaret ved å ta $-(10101+1)=-10110$.

Resultatet blir altså likt. En annen fin ting med 2ers komplement-metoden er at dersom en lagrer negative binære i 2er komplements form, så er de klare til bruk uten at en trenger å konvertere.

La oss starte med en såkalt byte bestående av åtte bit. Det minst signifikante bit (LSB) er til høyre og det mest signifikante bit (MSB) til venstre. MSB brukes for å indikere fortegn. Dersom MSB er 0, er tallet positivt, og er MSB 1, er tallet negativt. Vi har da igjen syv bit til selve tallet. Med åtte bit er det plass til $2^8 = 256$ ulike tall som starter fra -128 og går til $+127$. Asymmetrien skyldes det enkle faktum at vi må ha med oss null også.

Tabell 4.6 Binære tall med fortegn

Binær	Desimal	Heksadesimal
10000000	−128	80
10000001	−127	81
10000010	−126	82
10000011	−125	83
…	…	…
…	…	…
…	…	…
11111110	−2	*FE*
11111111	−1	*FF*
00000000	0	0
00000001	1	1
00000010	2	2
00000011	3	3
…	…	…
…	…	…
…	…	…
01111101	125	7*D*
01111110	126	7*E*
01111111	127	7*F*

Her ser en at de negative tallene er 2ers komplement av det tilsvarende positive tallet. Ta for eksempel 10000011 (desimalt -125) og utfør 2ers komplement $01111100+1=01111101$ (desimalt 125). Hva med andre veien? 01111101 (desimalt 125) gir 2ers komplement $10000010+1=10000011$ (desimalt -125). Ikke overraskende er frem og tilbake like langt. Tallene er 2ers komplement av hverandre.

En grei metode dersom en ønsker å konvertere et negativt binært tall til et desimaltall, er å tolke fortegnsbit som -128 når en har en byte. Da har en for eksempel:

$$(10000011)_2 = 1\cdot(-128)+0\cdot 64+0\cdot 32+0\cdot 16+0\cdot 8+0\cdot 4+1\cdot 2+1\cdot 1=-128+2+1=(-125)_{10}$$

For å få regning med tall med fortegn under huden samt lære oss å regne med heksadesimale tall, kan vi ta noen eksempler.

	Heksadesimal subtraksjon	Binær subtraksjon med 2ers komplement	
	^{10}	11 11	← Mente
Minuend	63	01100011	
Subtrahend	−3A	+11000110	
Differanse	=29	=100101001	

Bare for å ha sagt det: Mente i venstre kolonne, 10, er skrevet heksadesimalt og tilsvarer 16 desimalt.

Vi ser at vi får likt svar når vi husker at siste mente i 2ers komplement skal forkastes. 2ers komplement til $3A$ kan finnes på ulike måter.

1) Konverter $3A$ til det tilsvarende binære tall 00111010. Ta 1ers komplement og få 11000101. Legg til 1, $(11000101+1)=11000110,$ og en har 2ers komplement. På heksadesimal form er $(11000110)_2=(C6)_{16}$.
2) En variant av metoden ovenfor er å trekke 3A fra det maksimale heksadesimale tallet og legge til 1. $FF-3A=C5+1=C6$.
3) En siste metode er å ta tallet siffer for siffer og finne et tilsvarende siffer som gjør summen av sifrene lik F. For $3A$ vil $3+C=F$ og $A+5=F$, og tallet en får er $C5$. Til slutt legger en til 1 og får $C6$.

$-3A$ er lagret som $C6$ når vi regner med tall med fortegn, så da er det bare å addere med $C6$ for å få svaret. Igjen forkastes mente.

	Heksadesimal subtraksjon	Heksadesimal subtraksjon med 2ers komplement	
	10	1	← Mente
Minuend	~~6~~3	63	
Subtrahend	−3*A*	+*C*6	
Differanse	=29	=129	

La oss prøve å addere to negative tall som desimalt har verdiene verdiene –5 og –9.

	Desimal addisjon	Binær addisjon	Heksadesimal addisjon	
	1	11111111	11	← Mente
Minuend	–5	11111011	*FB*	
Addend	–9	+11110111	+*F*7	
Sum	= –14	=111110010	=1*F*2	

I den heksadesimale addisjonen er tallene med fortegnsbit gitt i 2ers komplements form, og igjen forkastes det siste mente. Er svaret til høyre lik -14 ? La oss sjekke:

$$(F2)_{16} = (11110010)_2 = 1\cdot(-128)+1\cdot 64+1\cdot 32+1\cdot 16+0\cdot 8+0\cdot 4+1\cdot 2+0\cdot 1 = -128+64+32+16+2 = (-14)_{10}$$

Når de negative tallene lagres i 2ers komplements form, blir alt addisjon, og en får fire mulige konstellasjoner.

1) Når en har to positive tall, blir summen positiv med 0 i tegnbitet og intet mente å forkaste.
2) Når et positivt tall legges sammen med et mindre negativt tall, blir summen positiv med 0 i tegnbitet og siste mente å forkaste.
3) Når et positivt tallet legges sammen med et større negativt tall, kommer summen som er negativ med 1 i tegnbit i 2ers komplements form.
4) Når begge tall er negative, er summen negativ med 1 i tegnbitet i 2ers komplements form og med siste mente å forkaste.

Her ville nok vår matematikervenn benyttet anledningen til å fortelle oss at det er modulo aritmetikk vi holder på med. Hva er så det? Et eksempel er $8+6\,mod\,12=2$. Vi ser at svaret er resten av 8 + 6 når vi heltallsdividerer med 12. Denne modulo aritmetikken ble vi alle utsatt for da vi lærte klokken. Med en byte vil det å kaste siste mente være det samme som å ta modulo 256 av summen. La oss se på konstellasjon

2) med tallene $(9–7)_{10} = (00001001 + (11111000 + 1))_2 = (00001001 + (11111001))_2$. La oss konvertere de to binære tallene uten å tenke på MSB som tegnbit. $(0000100 + (11111001))_2 = (9 + (249))_{10} = 258$. 258 mod 256 = 2 og stemmer fortreffelig. Hva med konstellasjon 4)? La oss gjøre det enkelt ved å se på $(–7–7)_{10} = ((11111001) + (11111001))_2 = (249 + 249)_{10} = 498$. 498 mod 256 = 242 og fortegnet får vi på plass ved å trekke 256 fra 242. 242 – 256 = –14.

Hva skjer dersom vi prøver å sprenge grenser?

	Desimal addisjon	Binær addisjon	Heksadesimal addisjon	
	1	1111	1	← Mente
Minuend	125	01111101	*7D*	
Addend	58	+00111010	+*3A*	
Sum	= 183	= 10110111	= *B7*	

Ooops!, her har vi fått tegnbit lik 1 og størrelsen $(0110111)_2 = 0{\cdot}64 + 1{\cdot}32 + 1{\cdot}16 + 0{\cdot}8 + 1{\cdot}4 + 1{\cdot}2 + 1{\cdot}1 = 32 + 16 + 4 + 2 + 1 = 55$. Vi har fått overflyt, siden summen 183 er større enn det vi har plass til, nemlig 127.

Overflyt kan vi bare få når vi adderer enten to positive tall hvis sum blir større enn 127, eller to negative tall hvis sum er mindre enn –128. La oss se hva som skjer dersom tallene ovenfor hadde vært negative.

	Desimal addisjon	Binær addisjon	Heksadesimal addisjon	
	1	1 11	1	← Mente
Minuend	–125	10000011	83	
Addend	–58	+11000110	+*C6*	
Sum	= –183	= 101001001	= 149	

Mer ooops!, her har vi fått tegnbit lik 0 og størrelsen $(1001001)_2 = 1{\cdot}64 + 0{\cdot}32 + 0 \cdot 16 + 1{\cdot}8 + 0 \cdot 4 + 0 \cdot 2 + 1 \cdot 1 = 64 + 8 + 1 = 73$. Vi har fått overflyt, siden summen –183 er mindre enn det vi har plass til, nemlig –128.

En datamaskin begrenset til å regne mellom –128 til 127 blir litt puslete, derfor settes bytes sammen til ord. To bytes gir 16 bit tilgjengelig. Da brukes MSB i det mest signifikante byte til tegnbit, området blir fra –32768 til 32767, og regningen blir som

før. Man må bare huske å la mente flyte over fra et byte til et annet. Skulle en ønske å regne med større tall, er det bare å pøse på med mer bytes og på den måten gjøre ordet større.

Det har vært en lang dags ferd mot natt bare å lære seg å addere og subtrahere med binære tall. Hva med de to andre regningsartene, multiplikasjon og divisjon? Igjen er det rett frem. La oss prøve multiplikasjon først:

$$\begin{array}{r} 00000101 \cdot 00000111 \quad = 00010011 \\ \hline 00000101 \\ 00000101 \\ +00000101 \\ \hline 00100011 \end{array}$$

Utfordringen er at det er ikke slik datamaskiner regner. Når de multipliserer, adderer de isteden. Regnestykket ovenfor utføres ved å addere 00000101 00000111 ganger. Resultatet blir selvfølgelig det samme. Dersom det multipliseres med 2^n, gjør datamaskiner det ved å flytte (skifte) bitene n plasser mot venstre og fylle på med nuller.

$$00000101 \cdot 00000100 = 00010100$$

Her multipliseres det med 100, og da er $n = 2$ og en flytter 00000101 to plasser til venstre og fyller på med to nuller. La oss sjekke for sikkerhets skyld:

$$\begin{array}{r} 00000101 \cdot 00000100 \quad = 0010100 \\ \hline 00000000 \\ 00000000 \\ +00000101 \\ \hline 00010100 \end{array}$$

Her ville nok Sherlock Holmes ha utbrutt «Elementært Watson!», men det er jo en første gang for alt.

Så er det divisjon som står for tur.

```
00000110 : 00000010  = 00000011
–00000010
--------
 00000010
–00000010
--------
 00000000
```

Her er vi heldige og får ingen rest. Hadde vi fått det, settes det bare komma, og så fortsetter en videre slik vi lærte på barneskolen. Riktignok må vi supplere med byte for den del av tallet som er etter komma.

Igjen er det slik at datamaskiner ikke gjør det på denne måten. De bruker subtraksjon, og i dette tilfellet ville 00000010 trekkes 00000011 ganger fra 00000110 til resten ble 00000000.

Dersom det divideres med 2^n, utfører datamaskiner det ved å skifte bitene n plasser mot høyre og fylle på med samme tall som sto i det mest signifikante bit (tegnbit). Årsaken til det siste er at en skal ta hensyn til at tallene er representert i 2ers komplement.

$$00000110 : 00000100 = 00000001{,}1$$

Her divideres det med 100, og da er $n = 2$ og en flytter 00000110 to plasser til høyre og fyller på med to nuller siden tallet er positivt. La oss igjen sjekke for å føle oss trygge:

```
00000110 : 00000100  = 0000001,1
–00000100
--------
 00000100
–00000100
--------
 00000000
```

Igjen blir det likt.

Flyttall

De fleste datamaskiner har en såkalt koprosessor for å utføre flyttallsberegninger. Flyttall er tall uttrykt ved hjelp av en desimalbrøk og en eksponent. Jeg husker mitt første møte med en slik en (8087) tidlig på 1980-tallet. Den var priset hinsides og så ut som et tusenbein med svake føtter som skulle ned i en sokkel som var satt av til den i IBM PC XT. Med angst og beven og svetteperler på pannen satte jeg den bokstavelig talt på plass. Hildrende du! Det var som å sette turbo på beregningene.

I slike prosessorer brukes flyttall som angis på følgende vis som i eksempelet her:

$$0,2315078 \cdot 10^{9}$$

$0,2315078$ kalles mantissen og 9 eksponenten. For binære flyttall er formatet definert av ANSI/IEEE Standard 754-1985 og kommer i tre størrelser: singel presisjon, dobbel presisjon og utvidet presisjon med henholdsvis 32, 64 og 80 bit.

Standard format for singel presisjon er gitt ved:

← 32 bit →

S	Eksponent (E)	Mantisse (F)
1 bit	8 bit	23 bit

Mantissen består av 23 bit, men da et binært tall alltid starter med 1, har en 24 bit til rådighet fordi en ikke trenger å bruke noe bit for den mest signifikante 1-er. I eksponenten bruker en en såkalt «biased» eksponent som finnes ved å addere 127 til den egentlige eksponent. På den måte kan eksponenter fra 126 til 128 representeres. La oss se på et eksempel:

$$1011010010001 = 1.011010010001 \cdot 2^{12}$$

«Biased» eksponent blir 12 + 127 = 139 som binært er 10001011 = 128 + 8 + 2 + 1, og mantissen 011010010001 siden den første 1-eren er underforstått.

0	10001011	01101001000100000000000
S	E	F

Generelt har en:

$$tall = (-1)^S (1+F)\left(2^{E-127}\right)$$

Et eksempel hvor vi prøver å tolke bitmønsteret:

1	1001001	10001110001000000000000
S	E	F

$$tall = (-1)^1 (1{,}10001110001000000000000)\left(2^{145-127}\right)$$
$$= (-1)(1{,}10001110001000000000000)\left(2^{18}\right) = -1100011100010000000$$

Helt til slutt: Det finnes to unntak fra formatet. 0,0. representeres ved at alle bitene er 0 og tallet uendelig med bare 0-er i mantissen og bare 1-ere i eksponenten.

BCD

Når begynte den moderne tid? 14:11 74 aug 11. Busstasjonen i Bergen, perrong 11. Med mitt nyinnkjøpte LED-ur startet jeg digitalalderen med et lett trykk. Bak det hemmelighetsfulle, rødbrune dekselet lyste fremtiden mot meg og en anselig mengde fremmede som fjetret tok imot den binære visjon. Når sluttet den moderne tid? Ikke like lett å tidfeste, men ca. fjorten dager senere var batteriet flatt. Etter at en begynte med mer strømgjerrig teknologi, har såkalte syvsegments elektriske tall dukket opp overalt i kalkulatorer, termometre og andre målere. Ja, de viser til og med prisen på varer i butikken. At det brukes BCD «Binary Coded Desimal», lærte jeg først nylig.

Den mest vanlige BCD-koden er den såkalte 8421 BCD-koden. Her signaliserer tallrekken at en bygger opp koden binært med 4 bit hvor de enkelte bit henholdsvis representerer desimalverdiene 8, 4, 2 og 1. En bruker 4 bit til å representere desimaltallene 0 til 9.

Tabell 4.7 Desimalt/BCD

Konvertering mellom desimalt og BCD										
Desimalt	0	1	2	3	4	5	6	7	8	9
BCD	0000	0001	0010	0011	0100	0101	0110	0111	1000	1001

Den observante leser vil her se at BCD følger mønsteret til oppbygging av vanlige binære tall, men kombinasjonene vi tidligere brukte for 10 til 15 (1010, 1011, 1100, 1101, 1110, 1111), er nå ulovlige. La oss se på noen eksempler for å lære å konvertere frem og tilbake mellom desimalt og BCD.

35=0011 0101

2469=0010 0100 0110 1001

10000110=1000 0110=86

1001010001110000=1001 0100 0111 0000=9470

Vi ser hvorfor denne koden er blitt populær. Det er rimelig lett å konvertere. Å regne med BCD gir litt større utfordringer. La oss se på addisjon, da det tross alt er den grunnleggende operasjonen for datamaskiner. Problemet er at vi har noen ulovlige koder, og det løses ved å hoppe over dem.

Følgende prosedyre hjelper oss på veien:

1) Adder to BCD tall ved å bruke de vanlige reglene for binære tall.
2) Hvis en 4 bit sum er lik eller mindre enn 9, så er det et lovlig BCD tall.
3) Hvis en 4 bit sum er større enn 9 eller gir mente, så er det et ulovlig resultat. Adder 6 (0110) til 4 bit summen for å hoppe over de ulovlige kodene og returnere til 8421 BCD kode. Hvis det blir mente når 6 adderes, ta mente med til neste 4 bit gruppe.

La oss ta et eksempel på det bekymringsløse først:

23	0010	0011
+15	+0001	0101
=38	=0011	1000

Så lenge tallene er mindre enn eller lik 9, er det rett frem. La oss utfordre skjebnen ved å gå litt ut av komfortsonen.

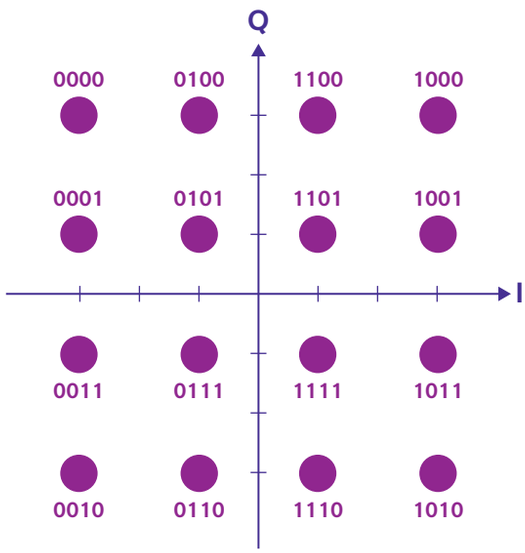
9		1001	
+4		+0100	
		1101	>9
		+0110	+6 gir mente
=13	=0001	0011	

16	0001	0110	
+15	+0001	+0101	
	0010	1011	> 9
		+0110	+6 gir mente i høyre kolonne
= 31	= 0011	0001	

Det er selvfølgelig mulig å utføre de tre andre regningsartene og andre C-momenter med BCD koder, men det overlates til spesielt interesserte.

Gray-kode

Gray-kode er en måte å kode binære tall på slik at et kun får forandring i 1 bit når en øker eller minsker det opprinnelige ukodete tallet med 1. Hensikten med denne koden er å minske sårbarheten for feil når det opprinnelige tallet forandrer seg med pluss minus 1. Gray-koder brukes i mange ulike applikasjoner. Jeg møtte dem første gang i forbindelse med koding av radiosignaler. Der hadde en for eksempel 16 ulike tilstander som ble sendt på luften med ulike analoge amplituder og faser. Disse 16 tilstandene ble Gray-kodet siden sannsynligheten var størst for at en feil ville gi en tilstøtende tilstand og dermed bare feil i 1 bit. Mer om dette finner du i kapittelet «Fra A til B».



Figur 4.1 Gray-kode for et radiosignal

Gray-kode produseres enkelt fra et binært tall med følgende prosedyre:

$$g_3 = b_3$$

$$g_2 = b_3 \oplus b_2$$

$$g_1 = b_2 \oplus b_1$$

$$g_0 = b_1 \oplus b_0$$

Her er g_n og b_n enkeltbit i posisjon n i henholdsvis Gray-kode og det binære tallet. Hva er $\oplus$? Her foregriper vi begivenhetenes gang og introduserer for første gang en boolsk funksjon. Den kalles for eksklusiv «or», XOR, hvor hvor $0 \oplus 0 = 0$, $0 \oplus 1 = 1$, $1 \oplus 0 = 1$, $1 \oplus 1 = 0$. Motsatt vei fra Gray-kode til binær:

$$b_3 = g_3$$

$$b_2 = b_3 \oplus g_2$$

$$b_1 = b_2 \oplus g_1$$

$$b_0 = b_1 \oplus g_0$$

Tabell 4.8 Fire bit Gray-kode

Desimal	Binær	Gray-kode	Desimal	Binær	Gray-kode
0	0000	0000	8	1000	1100
1	0001	0001	9	1001	1101
2	0010	0011	10	1010	1111
3	0011	0010	11	1011	1110
4	0100	0110	12	1100	1010
5	0101	0111	13	1101	1011
6	0110	0101	14	1110	1001
7	0111	0100	15	1111	1000

Finmysing på denne tabellen viser at Gray-koden har de ønskede egenskaper. Den endrer seg kun 1 bit når tallet endres pluss minus en, og den er entydig (dvs. to ulike tall har ikke samme Gray-kode).

ASCII-kode

Tastaturet på datamaskinen din har en dedikert mikroprosessor som hele tiden detekterer eventuelle anslag. Mikroprosessoren oversetter bokstaven, kommandoen eller symbolet du har trykket på til en ASCII-kode. ASCII står for «American Standard Code for Information Interchange». Trykker du for eksempel på «a», sendes $(1100001)_2 = (97)_{10}$ til PC-en din. Som en ser, består ASCII-koden av 7 bit, men da IBM lanserte sin PC, utvidet de koden til 8 bit for blant annet å få plass til våre kjære æ, ø og å.

Tabell 4.9 ASCII-kode

Binært	Desimal	Hex	Forkortelse	Vises som	Navn/mening
000 0000	0	00	NUL	$^{N}{}_{U_L}$	Null character
000 0001	1	01	SOH	$^{S}{}_{O_H}$	Start of Header
000 0010	2	02	STX	$^{S}{}_{T_X}$	Start of Text
000 0011	3	03	ETX	$^{E}{}_{T_X}$	End of Text
000 0100	4	04	EOT	$^{E}{}_{O_T}$	End of Transmission
000 0101	5	05	ENQ	$^{E}{}_{N_Q}$	Enquiry
000 0110	6	06	ACK	$^{A}{}_{C_K}$	Acknowledgement
000 0111	7	07	BEL	$^{B}{}_{E_L}$	Bell
000 1000	8	08	BS	$^{B}{}_{S}$	Backspace
000 1001	9	09	HT	$^{H}{}_{T}$	Horizontal Tab
000 1010	10	0A	LF	$^{L}{}_{F}$	Line feed
000 1011	11	0B	VT	$^{V}{}_{T}$	Vertical Tab
000 1100	12	0C	FF	$^{F}{}_{F}$	Form Feed
000 1101	13	0D	CR	$^{C}{}_{R}$	Carriage return
000 1110	14	0E	SO	$^{S}{}_{0}$	Shift Out
000 1111	15	0F	SI	$^{S}{}_{I}$	Shift In
001 0000	16	10	DLE	$^{D}{}_{L_E}$	Data Link Escape
001 0001	17	11	DC1	$^{D}{}_{C_1}$	Device Control 1 – oft. XON
001 0010	18	12	DC2	$^{D}{}_{C_2}$	Device Control 2
001 0011	19	13	DC3	$^{D}{}_{C_3}$	Device Control 3 – oft. XOFF
001 0100	20	14	DC4	$^{D}{}_{C_4}$	Device Control 4
001 0101	21	15	NAK	$^{N}{}_{A_K}$	Negative Acknowledgement

Binært	Desimal	Hex	Forkortelse	Vises som	Navn/mening
001 0110	22	16	SYN	$^{S}_{N}$	Synchronous Idle
001 0111	23	17	ETB	$^{E}_{B}$	End of Trans. Block
001 1000	24	18	CAN	$^{C}_{N}$	Cancel
001 1001	25	19	EM	$^{E}_{M}$	End of Medium
001 1010	26	1A	SUB	$^{S}_{B}$	Substitute
001 1011	27	1B	ESC	$^{E}_{C}$	Escape
001 1100	28	1C	FS	$^{F}_{S}$	File Separator
001 1101	29	1D	GS	$^{G}_{S}$	Group Separator
001 1110	30	1E	RS	$^{R}_{S}$	Record Separator
001 1111	31	1F	US	$^{U}_{S}$	Unit Separator
111 1111	127	7F	DEL	$^{D}_{L}$	Delete

Binært	Desimal	Hex	Grafisk	Binært	Desimal	Hex	Grafisk	Binært	Desimal	Hex	Grafisk
010 0000	32	20	Mellomrom	100 0000	64	40	@	110 0000	96	60	`
010 0001	33	21	!	100 0001	65	41	A	110 0001	97	61	a
010 0010	34	22	“	100 0010	66	42	B	110 0010	98	62	b
010 0011	35	23	#	100 0011	67	43	C	110 0011	99	63	c
010 0100	36	24	$	100 0100	68	44	D	110 0100	100	64	d
010 0101	37	25	%	100 0101	69	45	E	110 0101	101	65	e
010 0110	38	26	&	100 0110	70	46	F	110 0110	102	66	f
010 0111	39	27	‘	100 0111	71	47	G	110 0111	103	67	g
010 1000	40	28	(	100 1000	72	48	H	110 1000	104	68	h
010 1001	41	29	)	100 1001	73	49	I	110 1001	105	69	i
010 1010	42	2A	*	100 1010	74	4A	J	110 1010	106	6A	j
010 1011	43	2B	+	100 1011	75	4B	K	110 1011	107	6B	k
010 1100	44	2C	,	100 1100	76	4C	L	110 1100	108	6C	l
010 1101	45	2D	-	100 1101	77	4D	M	110 1101	109	6D	m
010 1110	46	2E	.	100 1110	78	4E	N	110 1110	110	6E	n
010 1111	47	2F	/	100 1111	79	4F	O	110 1111	111	6F	o
011 0000	48	30	0	101 0000	80	50	P	111 0000	112	70	p
011 0001	49	31	1	101 0001	81	51	Q	111 0001	113	71	q

Binært	Desimal	Hex	Grafisk
011 0010	50	32	2
011 0011	51	33	3
011 0100	52	34	4
011 0101	53	35	5
011 0110	54	36	6
011 0111	55	37	7
011 1000	56	38	8
011 1001	57	39	9
011 1010	58	3A	:
011 1011	59	3B	;
011 1100	60	3C	<
011 1101	61	3D	=
011 1110	62	3E	>
011 1111	63	3F	?

Binært	Desimal	Hex	Grafisk
101 0010	82	52	R
101 0011	83	53	S
101 0100	84	54	T
101 0101	85	55	U
101 0110	86	56	V
101 0111	87	57	W
101 1000	88	58	X
101 1001	89	59	Y
101 1010	90	5A	Z
101 1011	91	5B	[
101 1100	92	5C	\
101 1101	93	5D	]
101 1110	94	5E	^
101 1111	95	5F	_

Binært	Desimal	Hex	Grafisk
111 0010	114	72	r
111 0011	115	73	s
111 0100	116	74	t
111 0101	117	75	u
111 0110	118	76	v
111 0111	119	77	w
111 1000	120	78	x
111 1001	121	79	y
111 1010	122	7A	z
111 1011	123	7B	{
111 1100	124	7C	|
111 1101	125	7D	}
111 1110	126	7E	~

Da gjenstår det kun å teste kunnskapen. Hva er mer naturlig enn å prøve seg på en eksamensoppgave?

Oppgave

a. Tallsystemkonvertering
   i. Konverter binærtallet 10011 til desimaltall
   ii. Konverter desimaltallet 131 til binærtall
   iii. Konverter binærtallet 1000010111 til heksadesimaltall

b. Representer tallene a = 38_{10} og b = -14_{10} på 2-komplement form (bruk 8 bit) og utfør den aritmetiske operasjonen a + b.

La oss starte med a. i., da den ser enklest ut:

$$(10011)_2 = 1\cdot 16 + 0\cdot 8 + 0\cdot 4 + 1\cdot 2 + 1\cdot 1 = (19)_{10}$$

For å være helt sikker i sin sak, kan det være fornuftig å kontrollregne ved å gå andre veien. Det gjør en ved suksessivt å heltallsdividere på 2 til koeffisienten er 0.

Heltallsdivisjon	Koeffisient	Rest	s_n
$19/2$	9	1	s_0
$9/2$	4	1	s_1
$4/2$	2	0	s_2
$2/2$	1	0	s_3
$1/2$	0	1	s_4

$$(19)_{10} = s_4 \cdot 16 + s_3 \cdot 8 + s_2 \cdot 4 + s_1 \cdot 2 + s_0 \cdot 1 = 1 \cdot 16 + 0 \cdot 8 + 0 \cdot 4 + 1 \cdot 2 + 1 \cdot 1 = (10011)_2$$

Det stemmer forbløffende. Et trenet øye som har lest hele kapittelet, ville nok ha sett at svaret var 19 bare ved å kaste et blikk på det binære tallet. Det er i dette tilfellet tross alt ikke så langt. Med smaken av suksess er det bare å fortsette ufortrødent videre.

I neste oppgave (a. ii.) skal vi på samme måte som da vi kontrollerte svaret i forrige oppgave, drive med heltallsdivisjon:

Heltallsdivisjon	Koeffisient	Rest	s_n
$131/2$	65	1	s_0
$65/2$	32	1	s_1.
$32/2$	16	0	s_2
$16/2$	8	0	s_3
$8/2$	4	0	s_4
$4/2$	2	0	s_5
$2/2$	1	0	s_6
$1/2$	0	1	s_7

$$(131)_{10} = s_7 \cdot 128 + s_6 \cdot 64 + s_5 \cdot 32 + s_4 \cdot 16 + s_3 \cdot 8 + s_2 \cdot 4 + s_1 \cdot 2 + s_0 \cdot 1$$
$$= 1 \cdot 128 + 0 \cdot 64 + 0 \cdot 32 + 0 \cdot 16 + 0 \cdot 8 + 0 \cdot 4 + 1 \cdot 2 + 1 \cdot 1 = (10000011)_2$$

Det er rimelig lett å se at dette stemmer, da $131 = 128 + 2 + 1$.

Dette er jo på grensen til morsomt. La oss utfordre skjebnen med a. iii. Der skal $(1000010111)_2$ konverteres til et heksadesimalt tall. Det er bare å gruppere de binære tallene fire og fire fra LSB.

$$(1000010111)_2 = (10\ 0001\ 0111)_2 = 2 \cdot 16^2 + 1 \cdot 16^1 + 7 \cdot 16^1 = (217)_{16}$$

Et lite skår i gleden er det at vi ikke får hilse på tallene A til F. Her kan vi for eksempel kontrollregne ved å gjøre både heksadesimal- og binærpresentasjonen om til desimal og se om vi får det samme resultat.

$$(1000010111)_2 = (512 + 16 + 4 + 2 + 1) = (535)_{10}$$

$$(217)_{16} = 2 \cdot 16^2 + 1 \cdot 16^1 + 7 \cdot 16^1 = (535)_{10}$$

Dobbel suksess!

Da er det kun en talloppgave igjen, b. Den er litt mer utfordrende. La oss starte med å beregne med desimaltall for på en enkel måte å finne ut hva svaret skal bli.

$$(38 - 14)_{10} = (24)_{10} = 0 \cdot (-128) + 0 \cdot 64 + 0 \cdot 32 + 1 \cdot 16 + 1 \cdot 8 + 0 \cdot 4 + 0 \cdot 2 + 0 \cdot 1 = (00011000)_2$$

Med fasit innabords er det bare å sette i gang. Som du ser, har vi brukt 8 bit, da vi har blitt bedt om det.

Tallene $a = (38)_{10}$ og $b = (-14)_{10}$ skal presenteres på 2ers komplements form. La oss ta det enkle først, nemlig det positive tallet. Det er rett frem:

$$a = (38)_{10} = 0 \cdot (-128) + 0 \cdot 64 + 1 \cdot 32 + 0 \cdot 16 + 0 \cdot 8 + 1 \cdot 4 + 1 \cdot 2 + 0 \cdot 1 = (00100110)_2$$

Det er mange måter å finne 2ers komplement av et negativt binært tall på. La oss gjøre det lettvint med å ta det tilsvarende positive binære tallet og så utføre 1ers komplement og addere 1. De positive og de negative tallene er jo 2ers komplement av hverandre, slik de ble definert i tabell 4.6. I vårt tilfelle har vi at $(14)_{10} = (00001110)_2$. $b = (-14)_{10}$ på 2ers komplements form blir:

Tall	00001110
1ers komplement av tall	11110001
	+1
2er komplement av tall	=11110010

La oss sjekke at det stemmer:

$$b=(11110010)_2=1\cdot(-128)+1\cdot 64+1\cdot 32+1\cdot 16+0\cdot 8+0\cdot 4+1\cdot 2+0\cdot 1=(-14)_{10}$$

Da er det bare å gjøre det siste vi blir bedt om, $a+b$:

Binær subtraksjon med 2ers komplement		
	111 11	← Mente
Minuend a	00100110	
Subtrahend b	+11110010	
Differanse	=100011000	

Igjen ser vi at det stemmer med fasit når vi bare husker å forkaste øverste mente.

Skal vi gjøre flere oppgaver? Nei, nå kan vi det. Jeg foreslår at vi heller setter datamaskinene til å telle for oss. Det er jo derfor de er skapt.

5

Kapittel 5

En del elementære logiske emner

«Jeg velger SANNHETEN!»

Herman Tønnessen (1918–2001)

LÆRINGSUTBYTTE: Utsagn, kombinerte utsagn, sannhetstabeller, ekvivalente utsagn, boolsk algebra, De Morgans lover, logiske funksjoner, SSOP (Standard Sum Of Products), minterm, logiske forenklinger, Karnaugh-diagram, «don't care», Quine-McCluskey, Espresso

Ifølge statsviteren Bernt Hagtvet er filosofen Arne Næss' bok *En del elementære logiske emner* den viktigste boken etter krigen. Boken, som er en kort og sylskarp introduksjon til semantikk og debattlære, ga ikke oss som måtte lese den under tvang som en del av Ex. Phil., de samme følelsene. Derimot ble filosofen Herman Tønnessens lille, pessimistiske eksistensfilosofiske bok *Jeg velger SANNHETEN!* lest med mye større interesse. Kanskje fordi den ikke var pensum? Den logikken jeg trenger, er samlet i en perm med sannhetstabeller som hviler mot Tønnessens mesterverk.

Påstanden til Hagtvet om at *En del elementære logiske emner* er den viktigste boken etter krigen, er problematisk fra et logisk ståsted, da det er umulig å fastslå sannhetsverdien. Det er kun synsing. Det en krever av et utsagn (påstand), er at det skal ha en sannhetsverdi, som kan være enten sann eller usann. I matematikken brukes ofte P eller Q til å betegne utsagn.

To eksempler på utsagn er:

$$P : \cos(0) = 1$$

$$Q : 39 \text{ er et primtall}$$

Sannhetsverdien til P er sann fordi cosinus til null grader er , mens sannhetsverdien til Q er usann siden 39 ikke er noe primtall.

Utsagn kan kombineres. Ved å bruke ordene «ikke», «og» og «eller» kan man skape nye utsagn.

Negasjon får en ved å ta et utsagn P og lage utsagnet «ikke P». Det skrives $\overline{P}$, i stil med komplementærmengden for mengder, og kalles negasjonen av P. Negasjonen til utsagnet Q ovenfor blir:

$$Q : 39 \text{ er ikke et primtall}$$

For å få oversikt over alle muligheter en har, kan en lage en såkalt sannhetstabell.

Tabell 5.1 Sannhetstabell for negasjon

P	$\overline{P}$
0	1
1	0

Her er 0 usann og 1 sann. Den øverste raden sier at dersom P er usann, så er $\overline{P}$ sann, og den nederste sier det motsatte. Det uttømmer alle mulighetene.

To utsagn P og Q kan kombineres med ordet «og» til utsagnet «P og Q». Det sammensatte utsagnet «P og Q» skrives som $P \wedge Q$. Vi kan for eksempel se på våre to tidligere utsagn:

$$P : \cos(0) = 1$$
$$Q : 39 \text{ er et primtall}$$

P er her sann og Q usann, dermed blir det sammensatte utsagnet $P \wedge Q$ usann. $P \wedge Q$ vil kun bli sann dersom både P og Q er sanne. Det hele kan oppsummeres i en sannhetstabell:

Tabell 5.2 Sannhetstabell for «og»

P	Q	$P \wedge Q$
0	0	0
0	1	0
1	0	0
1	1	1

De to utsagnene P og Q kan også kombineres med ordet «eller», til utsagnet «P eller Q». Det sammensatte utsagnet «P eller Q» skrives som $P \vee Q$. Vi ser igjen på våre to tidligere utsagn:

$$P : \cos(0) = 1$$
$$Q : 39 \text{ er et primtall}$$

P er her sann og Q usann, dermed blir det sammensatte utsagnet $P \vee Q$ sann. $P \vee Q$ vil kun bli usann dersom både P og Q er usanne. Igjen er det på sin plass med en liten sannhetstabell.

Tabell 5.3 Sannhetstabell for «eller»

P	Q	$P \vee Q$
0	0	0
0	1	1
1	0	1
1	1	1

Dette er jo rimelig overkommelig, men sannhetstabeller kommer først til sin rett når de logiske uttrykkene blir mer kompliserte. De blir da et fint verktøy for å holde oversikt over de logiske muligheter. Ta for eksempel det lett uoversiktlige sammensatte utsagnet:

5 er et primtall og et partall, eller så er ikke 5 et primtall eller partall

I det utsagnet er det to grunnutsagn:

P : 5 er et primtall

Q : 5 er et partall

Det sammensatte utsagnet uttrykt ved P og Q blir:

$$(P \wedge Q) \vee \overline{(P \vee Q)}$$

Parentesene er tilføyd bare for at vi skal utføre operasjonene i riktig rekkefølge. For å få oversikt lager vi en sannhetstabell som bygger seg opp mot det resulterende sammensatte utsagnet lengst mot høyre. Underveis har vi delutsagn som gjør det lettere å bestemme sluttutsagnet.

Tabell 5.4 Sannhetstabell for $(P \wedge Q) \vee \overline{(P \vee Q)}$

P	Q	$P \wedge Q$	$P \vee Q$	$\overline{P \vee Q}$	$(P \wedge Q) \vee \overline{(P \vee Q)}$
0	0	0	0	1	1
0	1	0	1	0	0
1	0	0	1	0	0
1	1	1	1	0	1

I vårt tilfelle er P sann da 5 er et primtall, og Q usann da 5 ikke er et partall. Vi må derfor bruke rad tre i tabellen og finner at det sammensatte utsagnet da er usant.

En kan også bruke sannhetstabeller til å undersøke om ulike sammensatte utsagn er ekvivalente (like). Det er ikke spesielt lett å se at for eksempel:

$$P \wedge (Q \vee R) = (P \wedge Q) \vee (P \wedge R)$$

Da kan en sannhetstabell være god å ha.

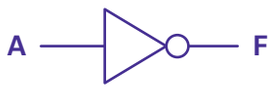
Tabell 5.5 Sannhetstabell for $P \wedge (Q \vee R) = (P \wedge Q) \vee (P \wedge R)$

P	Q	R	$Q \vee R$	$P \wedge (Q \vee R)$	$(P \wedge Q)$	$(P \wedge R)$	$(P \wedge Q) \vee (P \vee R)$
0	0	0	0	0	0	0	0
0	0	1	1	0	0	0	0
0	1	0	1	0	0	0	0
0	1	1	1	0	0	0	0
1	0	0	0	0	0	0	0
1	0	1	1	1	0	1	1
1	1	0	1	1	1	0	1
1	1	1	1	1	1	1	1

Siden en nå har tre utsagn, *P*, *Q* og *R*, blir det åtte rader i tabellen, og en ser at de to utsagnene vi skulle undersøke er ekvivalente.

En ser at jo flere logiske utsagn en har, jo større blir sannhetstabellen, og man kan lett miste oversikten. Det trengs hjelp, og her kommer digitalteknikken oss i møte. De grunnleggende logiske operasjonene «ikke», «og» og «eller» kan lages ved hjelp av elektroniske kretser og utsagnene 1 og 0 ved to ulike spenningsnivåer.

Den enkleste operasjonen «ikke» realiseres med en inverter og har følgende symbol:



Figur 5.1 IKKE-funksjonen (inverter)

Sannhetstabellen for IKKE-funksjonen er den samme som tabell 5.1 for negasjon. I figur 5.1 er A inngangen og F funksjonsverdien. IKKE-funksjonen er den enkleste boolske funksjonen vi har. Den henter sin inngangsverdi *A* fra den boolske mengden $\boldsymbol{B} = \{0,1\}$ og leverer funksjonsverdien F til en tilsvarende boolsk mengde $\boldsymbol{B}$. Dette kan skrives litt mer formelt som:

$$IKKE: \boldsymbol{B} \rightarrow \boldsymbol{B} \; hvor \; \boldsymbol{B} = \{0,1\}$$

En boolsk variabel x kan enten ha verdien 1 eller 0, og det angir en på følgende måte: $x \in \{0,1\}$.

Den neste funksjonen er OG-funksjonen. Sannhetstabellen er den samme som for den logiske operasjonen «og» gitt i tabell 5.2. I motsetning til IKKE-funksjonen har

OG-funksjonen to inngangsvariabler som leverer en funksjonsverdi. Det gir fire mulige kombinasjoner av inngangsverdier, og antallet rader i sannhetstabellen dobles i forhold til IKKE-funksjonens sannhetstabell. Den boolske funksjonen for OG kan skrives:

$$OG: \boldsymbol{B}^2 \rightarrow \boldsymbol{B} \; hvor \; \boldsymbol{B} = \{0,1\}$$

Det lille 2-tallet forteller oss at denne funksjonen har to inngangsvariabler A og B som den henter fra to ulike boolske mengder, og F er funksjonsverdien.

Figur 5.2 Symbol for OG-funksjonen

Som en ser, er det her to ulike symboler for den samme funksjonen. Det til venstre kalles ofte det amerikanske symbolet, mens det til høyre er definert av en IEC norm. For små kretser, som vi skal sysle med, er de amerikanske variantene av funksjonssymbolene de mest brukervennlige.

Den siste logiske operasjonen vi hadde definert, var «eller». På samme måte som for de to andre funksjonene kan den realiseres ved hjelp av elektronikk slik at den får en sannhetstabell lik «eller» gitt ved tabell 5.3. Symbolet for ELLER-funksjonen er gitt ved:

Figur 5.3 Symbol for ELLER-funksjonen

Den boolske funksjonen for ELLER kan skrives:

$$ELLER: \boldsymbol{B}^2 \rightarrow \boldsymbol{B} \; hvor \; \boldsymbol{B} = \{0,1\}$$

Med dette har en de grunnleggende funksjoner (operasjoner) for å kunne definere en algebra på den boolske mengden $\boldsymbol{B} = \{0,1\}$. For å gjøre forvirringen total, har en i elektronikken valgt å bruke andre tegn for «ikke», «og» og «eller» enn i den matematiske logikken. Ja, selv innenfor matematisk logikk har en ikke klart å bli enig om tegnene. Følgende lille tabell gir sammenhengen:

Tabell 5.6 Logiske tegn

Funksjon	Tegn – Matematisk logikk	Tegn – Digitalteknikk
IKKE	– eller ¬	‾
OG	∧	· eller ingen prikk
ELLER	∨	+

Hensikten med den boolske algebraen er at den skal reflektere logikk og dermed kunne brukes til kombinatorisk logikk hvor en ut av sammensatte utsagn kan beregne sannhetsverdien. En generell boolsk funksjon er gitt ved:

$$f_{boolsk} : \boldsymbol{B}^k \rightarrow \boldsymbol{B}^l \ hvor \ \boldsymbol{B} = \{0,1\}$$

Denne funksjonen tar et sammensatt utsagn med k boolske variabler og leverer en funksjonsverdi med l boolske verdier. En slik funksjon når den realiseres i digital elektronikk og $l = 1$, kalles en port.

Kommutative lover
Den boolske algebraen er rimelig lik algebra vi kjenner fra før. De kommutative lovene er gitt ved:

$$A + B = B + A$$

$$AB = BA$$

Her er A og B variabler hentet fra boolske mengder $\boldsymbol{B} = \{0,1\}$. Disse lovene er helt i tråd med sannhetstabellen for ELLER- og OG-funksjonen, da variablenes orden ikke spiller noen rolle i verken tabell 5.3 eller tabell 5.2.

Assosiative lover
De neste lovene en har i den boolske algebraen, er de assosiative lovene:

$$A + (B + C) = (A + B) + C$$

$$A(BC) = (AB)C$$

Dersom en lurer på om dette er riktig, kan en kontrollere det med en sannhetstabell. La oss sjekke den siste assosiative loven.

Tabell 5.7 Sannhetstabell for $A(BC) = (AB)C$

A	B	C	BC	A(BC)	AB	(AB)C
0	0	0	0	0	0	0
0	0	1	0	0	0	0
0	1	0	0	0	0	0
0	1	1	1	0	0	0
1	0	0	0	0	0	0
1	0	1	0	0	0	0
1	1	0	0	0	1	0
1	1	1	1	1	1	1

Ikke helt uventet stemmer loven. Den har tross alt blitt brukt siden George Boole introduserte den i sin bok *The Mathematical Analysis of Logic* i 1847.

Distributive lover

Boolsk algebra har også distributive lover:

$$A + (BC) = (A + B)(A + C)$$

$$A(B + C) = (AB) + (AC)$$

Igjen kan en overbevise seg selv om disse lovenes riktighet ved å lage sannhetstabeller, men kanskje vi heller i ren dovenskap skal stole på Boole.

I den boolske logikken kommer lovene parvis for ELLER og OG. Det er for eksempel ikke tilfellet for den vanlige algebraen med tall. Dersom vi hadde brukt $A + (BC) = (A + B)(A + C)$ på tall, hadde resultatet blitt fornøyelig, men alt annet enn riktig. Bare prøv med $A = 7$, $B = 4$ og $C = 2$, så ser du hvor det ender: 15 = 99.

Det finnes totalt ti par med boolske lover, og de er gitt i tabell 5.8.

Tabell 5.8 De boolske lovene

Nummer	ELLER	OG	Navn
B1	$A+A=A$	$AA=A$	Idempotent
B2	$A+(B+C)=(A+B)+C$	$A(BC)=(AB)C$	Assosiativ
B3	$A+B=B+A$	$AB=BA$	Kommutativ
B4	$A+(AB)=A$	$A(A+B)=A$	Absorpsjon
B5	$A+(BC)=(A+B)(A+C)$	$A(B+C)=(AB)(AC)$	Distribusjon
B6	$A+1=1$	$A0=0$	Bundet
B7	$A+0=A$	$A1=A$	Identitet
B8	$A+\overline{A}=1$	$A\overline{A}=0$	Komplement
B9	$\overline{0}=1$	$\overline{1}=0$	0 og 1
B10	$\overline{(A+B)}=\overline{A}\,\overline{B}$	$\overline{(AB)}=\overline{A}+\overline{B}$	De Morgan

De lovene som ikke gjelder i ordinær algebra, men i boolsk algebra, er angitt med farge. Alle de boolske lovene kunne vi ha vist gyldigheten av ved hjelp av sannhetstabeller. La oss innskrenke det til å undersøke De Morgans lover.

Tabell 5.9 De Morgans OG lov $\overline{(A+B)}=\overline{A}\,\overline{B}$

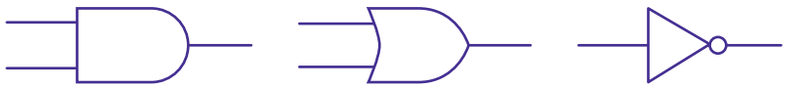
A	B	$A+B$	$\overline{(A+B)}$	$\overline{A}$	$\overline{B}$	$\overline{A}\,\overline{B}$
0	0	0	1	1	1	1
0	1	1	0	1	0	0
1	0	1	0	0	1	0
1	1	1	0	0	0	0

Tabell 5.10 De Morgans ELLER lov $\overline{(AB)}=\overline{A}+\overline{B}$

A	B	AB	$\overline{(AB)}$	$\overline{A}$	$\overline{B}$	$\overline{A}+\overline{B}$
0	0	0	1	1	1	1
0	1	0	1	1	0	1
1	0	0	1	0	1	1
1	1	1	0	0	0	0

De Morgans lover er nærmest trivielle, men han har allikevel skrevet seg inn i historieboken. Det skyldes det enkle faktum at de er meget anvendelige når boolske uttrykk skal manipuleres og forenkles.

OG-, ELLER- og IKKE-funksjonene realiseres i fysiske kretser som AND, OR og NOT.



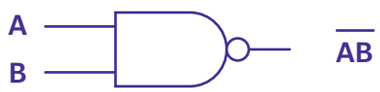
Figur 5.4 Symboler for AND, OR og NOT

Med IKKE-, OG- og ELLER-funksjonene har en sammen med den boolske algebraen det som trengs for å lage logiske kretser. Allikevel har en valgt å lage noen grunnfunksjoner til, nemlig NAND, NOR, XOR og XNOR.

NAND-funksjonen er kun en OG-funksjon med invertert utgang. Sannhetstabellen for NAND blir:

Tabell 5.11 Sannhetstabell for NAND

A	B	$\overline{AB}$
0	0	1
0	1	1
1	0	1
1	1	0



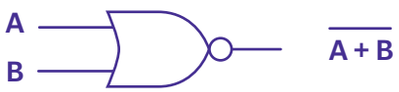
Figur 5.5 Symbol for NAND-funksjonen

Her forteller den lille rundingen etter AND-funksjonen at utgangen inverteres.

NOR-funksjonen er en ELLER-funksjon med invertert utgang. Sannhetstabellen for NOR blir:

Tabell 5.12 Sannhetstabell for NOR

A	B	$\overline{A+B}$
0	0	1
0	1	0
1	0	0
1	1	0



Figur 5.6 Symbol for NOR-funksjonen

Det som er så kjekt med både NAND og NOR, er at de er såkalt komplette. Det vil si at en kan utlede de tre basisfunksjonene IKKE, OG og ELLER fra enten NAND eller NOR ved å bruke den boolske algebra. En kan altså bygge opp en logisk krets ved bare å bruke NAND- eller NOR-funksjoner.

$$\overline{A} = \overline{AA} = \overline{A+A}$$

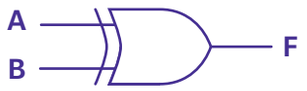
$$AB = \overline{\overline{AB}} = \overline{\overline{A}+\overline{B}}$$

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

XOR er eksklusiv OR. Det betyr at funksjonen er 1 kun når bare en av inngangsvariablene er 1. Sannhetstabellen for XOR blir:

Tabell 5.13 Sannhetstabell for XOR

A	B	$\overline{A}B + A\overline{B}$
0	0	0
0	1	1
1	0	1
1	1	0



Figur 5.7 Symbol for XOR-funksjonen

Det er vanlig å skrive XOR på følgende form: $A \oplus B = \overline{A}B + A\overline{B}$. XNOR er den inverterte av XOR og er gitt ved $\overline{A \oplus B}$.

Figur 5.8 Symbol for XNOR-funksjonen

Tenk deg at du skal lage en port som har mange digitale innganger og en utgang. Hvordan kan den realiseres med færrest mulige logiske grunnfunksjoner? Hensikten med å ha færrest mulige grunnfunksjoner er at det forenkler konstruksjonen, bruker mindre strøm og reagerer raskere (mindre «latency»). La oss starte med et eksempel på en port som har tre innganger A, B og C og en utgang F. Kravet til de logiske operasjoner er gitt i sannhetstabellen, og med den som utgangspunkt skal vi finne mer eller mindre effektive boolske uttrykk for F.

Tabell 5.14 Sannhetstabell for

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

Med første øyekast er det kanskje ikke så lett å se den optimale måten å realisere porten på. La oss starte med å skrive F på den såkalte Standard Sum of Products (SSOP). Da tar man bare rad for rad og bygger opp et uttrykk for F. Kun de radene hvor F er lik er interessante. I vårt tilfelle gir de to første radene og den siste raden bidrag. Fra første rad får en av produktet $\overline{A}\overline{B}\overline{C}$, fra rad 2 får en produktet $\overline{A}\overline{B}C$, og fra siste rad

får en fra ABC. Disse bidragene kalles mintermer. Tar en summen av disse radene, får en et boolsk uttrykk for F som var relativt lett å finne, et godt sted å starte, men som er langt fra optimalt.

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

Er det mulig å finne et enklere uttrykk og dermed en mer effektiv port? Ved å finmyse på uttrykket en har funnet ved hjelp av SSOP og bruke den boolske algebraen for alt den er verdt, er det mulig å redusere uttrykket. Ved å bruke den distributive loven B5 for OG-funksjonen kan F omformes til:

$$F = \overline{A}\,\overline{B}(\overline{C} + C) + ABC$$

Videre kan komplement B8 for ELLER-funksjonen brukes:

$$F = \overline{A}\,\overline{B}(1) + ABC = \overline{A}\,\overline{B} + ABC$$

Har vi nå funnet det enklest mulige uttrykket? Si det. Heldigvis er ikke vi de første som har syslet med denne problemstillingen, og vi kan trekke veksler på det Maurice Karnaugh fant på i 1953 da han arbeidet ved Bell Labs. Han utviklet et såkalt Karnaugh-diagram for å forenkle boolske uttrykk på SSOP formen til den enklest mulige sum av produkter. Det Karnaugh-diagrammet trekker veksler på, er at mennesket er et dyr med en velutviklet evne til å gjenkjenne mønstre.

Hver rute representer en linje i sannhetstabellen, og for tre inngangsvariabler ser diagrammet slik ut:

Tabell 5.15 Karnaugh-diagram for tre variabler

AB\C	0	1
00		
01		
11		
10		

Her er det verdt å observere at ved å flytte seg fra et hvitt felt i tabellen til et annet enten vannrett eller loddrett så er det kun en variabel som endrer verdi. Dette er annerledes enn det vi gjorde da variablenes verdi ble skrevet opp i sannhetstabellen hvor AB verdien 10 kom før 11. Det kalles Gray-koding, som vi har sett på tidligere, og er mye brukt når en skal skille nærliggende verdier fra hverandre og lettere se sammenhenger.

En annen ting er at diagrammet «wrapper» fra høyre til venstre og fra topp til bunn. Egentlig kan Karnaugh-diagrammet tegnes som en smultring; det er noe upraktisk på flatt papir, men er viktig å huske på når en skal sette sammen verdier.

La oss sette inn verdiene for vår funksjon F:

Tabell 5.16 Karnaugh-diagram for

AB\C	0	1
00	1	1
01	0	0
11	0	1
10	0	0

Det neste en gjør er å gruppere 1-ere i grupper. Vi får to grupper. Vi leser ut uttrykket for de enkelte gruppene ved å ta produktet at de variabler som ikke endrer seg. For rød gruppe er det A og B, og for oransje gruppe A, B og C. F blir:

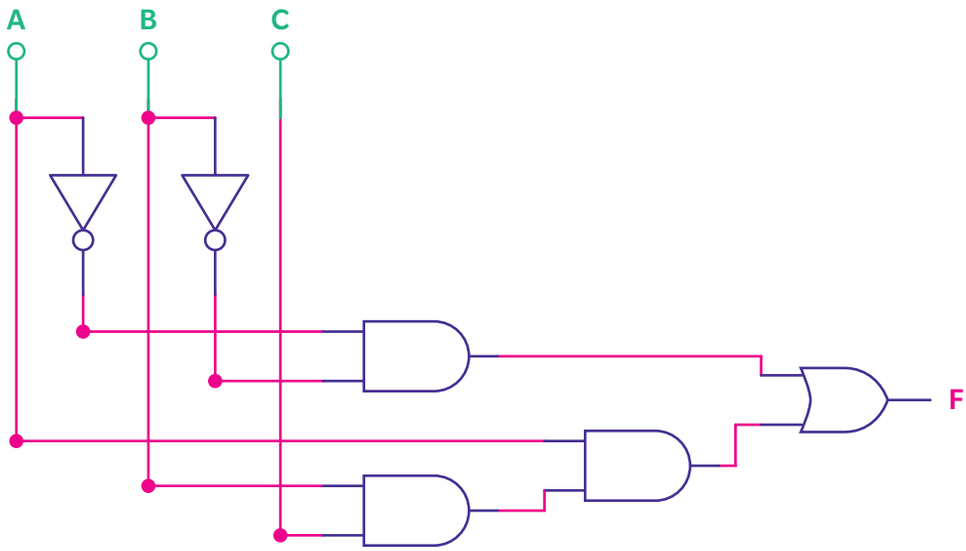
$$F = \overline{A}\,\overline{B} + ABC$$

La oss sjekke resultatet med å sette opp en sannhetstabell på nytt.

Tabell 5.17 Sannhetstabell for ***F*** funnet ved hjelp av Karnaugh-diagram

A	B	C	$\overline{A}\,\overline{B}$	ABC	F
0	0	0	1	0	1
0	0	1	1	0	1
0	1	0	0	0	0
0	1	1	0	0	0
1	0	0	0	0	0
1	0	1	0	0	0
1	1	0	0	0	0
1	1	1	0	1	1

Strålende! Det stemmer med den opprinnelige sannhetstabellen og det uttrykket vi kom frem til ved å foreta boolske forenklinger. I figur 5.9 er kretsen for F vist.



Figur 5.9 Krets for ***F***

Dette var jo rimelig overkommelig. La oss prøve å ta det til neste nivå med fire inngangsvariabler A, B, C og D og utgang F, og følgende sannhetstabell.

Tabell 5.18 Sannhetstabell for

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

Det var straks litt verre. La oss starte med det overkommelige, å finne SSOP.

$$F = \overline{A}\,\overline{B}\,\overline{C}\,\overline{D} + \overline{A}\,\overline{B}\,\overline{C}D + \overline{A}\,\overline{B}C\,\overline{D} + \overline{A}\,\overline{B}CD + A\,\overline{B}\,\overline{C}\,\overline{D} + A\,\overline{B}\,\overline{C}D + A\,\overline{B}C\,\overline{D} + ABCD$$

Det er i møte med slike uttrykk at en priser seg lykkelig for arbeidet Karnaugh og tidligere generasjoner med ham har utført. Igjen er det frem med et Karnaugh-diagram.

Tabell 5.19 Karnaugh-diagram for ***F***

AB\ CD	00	01	11	10
00	1	0	0	1
01	1	0	0	1
11	0	1	1	0
10	0	1	1	0

Vi har to grupper, da den oransje «wrapper» rundt fra høyre til venstre. For rød gruppe er det A og D som ikke forandrer seg, og for oransje gruppe er det også A og D. Uttrykket for blir:

$$F = \overline{A}\,\overline{D} + AD$$

Akkurat dette var det vel ikke så lett å se ved første øyekast på sannhetstabell eller SSOP. Dersom en ønsker å more seg, kan en lage sannhetstabell for det siste uttrykket og se at det stemmer med det opprinnelige.

Dersom en har andre sannhetstabeller og en skulle få overlappende eller delvis overlappende grupper i Karnaugh-diagrammet, er det helt uproblematisk. En legger dem bare sammen under mottoet 1 + 1 = 1 (B1)!

REGLER FOR BRUK AV KARNAUGH-DIAGRAMMER

- En gruppe består av 1, 2, 4, 8 eller 16 1-ere.
- En gruppe må være sammenhengende, og ha kvadratisk eller rektangulær form.
- Hver enkelt 1-er skal inngå i en så stor gruppe som mulig.
- Alle 1-ere må inngå i minst en gruppe. 1-ere kan inngå i flere grupper så lenge gruppene har 1-ere som ikke er felles.

Dersom det skulle være kombinasjoner av inngangsvariabler som ikke oppstår, eller hvor vi ikke bryr oss om hva funksjonsverdien blir, kalles det en «don't care» tilstand og avmerkes med X i Karnaugh-diagrammet. X kan anses som en dersom det fører til en større gruppering i diagrammet.

Tabell 5.20 Karnaugh-diagram for med «don't care» tilstander

AB\ CD	00	01	11	10
00	X	0	0	1
01	1	0	0	1
11	0	1	X	0
10	0	1	1	0

Hadde vi for eksempel hatt et diagram som i tabell 5.20, ville vi fremdeles gruppert som vi gjorde tidligere da vi fant at $F = \overline{A}\,\overline{D} + AD$.

Karnaugh-diagrammet er aldeles utmerket så lenge vi arbeider med få inngangsvariabler, men blir ubrukelig med mer enn fem inngangsvariabler. Har vi en slik utfordring med flere inngangsvariabler enn Karnaugh kan takle, kan vi profittere på samarbeidet mellom filosofen Willard Van Orman Quine og matematikeren Edward J. McCluskey. Sistnevnte arbeidet ved Bell Labs i 1956 da denne Quine-McCluskey-metoden ble utviklet. Bell Labs var i denne perioden et sydende arnested for mange ideer, teorier og metoder som vi bruker den dag i dag.

Quine-McCluskey-metoden er en utvidelse av Karnaugh-metoden og bruker tabeller for å komme frem til sluttresultatet som er den mest effektive måten å skrive et SOP uttrykk på. Quine-McCluskey-metoden er en formell metode hvor den boolske distributive lov brukes til å eliminere inngangsvariabler som opptrer som komplementer i to termer (for eksempel $ABCD + ABC\overline{D} = ABC$).

Den enkleste måten å tilegne seg kunnskap om Quine-McCluskey-metoden på er å ta et eksempel (Digital Fundamentals [2] side 237). Vi må starte med å skrive den ønskede funksjonen i standard minterm SOP. La oss bruke følgende funksjon:

$$F = \overline{A}\,\overline{B}\,\overline{C}D + \overline{A}\,\overline{B}CD + \overline{A}B\overline{C}\,\overline{D} + \overline{A}BC\overline{D} + A\overline{B}\,\overline{C}\,\overline{D} + AB\overline{C}\,\overline{D} + AB\overline{C}D + ABCD$$

Tabell 5.21 Sannhetstabell for *F* med angitte mintermer

A	B	C	D	F	Minterm
0	0	0	0	0	
0	0	0	1	1	m_1
0	0	1	0	0	
0	0	1	1	1	m_3
0	1	0	0	1	m_4
0	1	0	1	1	m_5
0	1	1	0	0	
0	1	1	1	0	
1	0	0	0	0	
1	0	0	1	0	
1	0	1	0	1	m_{10}
1	0	1	1	0	
1	1	0	0	1	m_{12}
1	1	0	1	1	m_{13}
1	1	1	0	0	
1	1	1	1	1	m_{15}

Det neste en gjør er å gruppere mintermene etter antall 1-ere i den gitte termen.

Tabell 5.22 Mintermer gruppert etter antall 1-ere

Antall 1-ere	Minterm	ABCD
1	m_1	0001
	m_4	0100
2	m_3	0011
	m_5	0101
	m_{10}	1010
	m_{12}	1100
3	m_{13}	1101
4	m_{15}	1111

Neste øvelse er å finne mintermer som kun er forskjellig i et bit. En ser for eksempel at m_1 og m_3 kun er forskjellige i posisjon C. Det betyr at en kan redusere $A\bar{B}\bar{C}D + A\bar{B}CD$ i F til $A\bar{B}D$ siden C her er med som både C og $\bar{C}$. La oss lage en ny tabell for å se om mer kan reduseres.

Tabell 5.23 Kombinasjoner av mintermer på første nivå

Antall 1-ere	Minterm	ABCD	Første nivå
1	m_1	0001	$(m_1, m_3)00X1$
	m_4	0100	$(m_1, m_5)0X01$
2	m_3	0011	$(m_4, m_5)010X$
	m_5	0101	$(m_4, m_{12})X100$
	m_{10}	1010	$(m_5, m_{13})X101$
	m_{12}	1100	$(m_{12}, m_{13})110X$
3	m_{13}	1101	$(m_{13}, m_{15})11X1$
4	m_{15}	1111	

Vi ser at mintermen m_{10} er den eneste som ikke har en minterm som er kun forskjellig i et bit. Det betyr at vi ikke kan redusere uttrykket $A\overline{B}C\overline{D}$ som ga denne mintermen. Vi har funnet vår første såkalte prime implicant. Hva med (m_1, m_3) og de andre på første nivå, er ikke de også «prime implicants»? Kanskje, men det vet vi ikke før vi har sett om de kan forenkles et nytt nivå. Igjen leter vi etter grupper som er 1 bit unna hverandre i en gitt posisjon. Etter litt mysing ser en:

Tabell 5.24 Kombinasjoner av mintermer på andre nivå

Første nivå	Antall 1-ere i første nivå	Andre nivå
$(m_1, m_3)00X1$	1	$(m_4, m_5, m_{12}, m_{13})X10X$
$(m_1, m_5)0X01$		$(m_4, m_{12}, m_5, m_{13})X10X$
$(m_4, m_5)010X$		
$(m_4, m_{12})X100$		
$(m_5, m_{13})X101$	2	
$(m_{12}, m_{13})110X$		
$(m_{13}, m_{15})11X1$	3	

På første nivå ser vi nå tre kombinasjoner av mintermer (de fargede) som ikke kan reduseres videre, og derfor er de «prime implicants». Finnes det flere nivåer? Nei, de to uttrykkene i andre nivå er like og kan derfor ikke forenkles. Vi kan derfor nå skrive F på følgende forenklede form:

$$F = B\overline{C} + \overline{A}\,\overline{B}D + \overline{A}\,\overline{C}D + ABD + A\overline{B}C\overline{D}$$

Her kommer første ledd i F fra andre nivå «prime implicant» $(m_4, m_5, m_{12}, m_{13})$, siste ledd fra mintermen som ikke kunne reduseres, og de tre i midten fra første nivå «prime implicants».

Er det mulig å redusere enda mer? La oss skrive opp resultatet i en tabell der vi krysser av alle mintermene som er inkludert i hver «prime implicant»:

Tabell 5.25 Mulige overflødige «prime implicants»

	Mintermer							
Prime implicants	m_1	m_3	m_4	m_5	m_{10}	m_{12}	m_{13}	m_{15}
$B\overline{C}\left(m_4, m_5, m_{12}, m_{13}\right)$			*	*		*	*	
$\overline{A}\ \overline{B}D\left(m_1, m_3\right)$	*	*						
$\overline{A}\ \overline{C}D\left(m_1, m_5\right)$	*			*				
$ABD\left(m_{13}, m_{15}\right)$							*	*
$A\overline{B}C\overline{D}\left(m_{10}\right)$					*			

For å få med de mintermene som kun er krysset ut en gang i sin kolonne, må «prime implicant» som gir krysset være med. I kolonner med flere kryss er det muligheter for forenklinger. En ser her at $\overline{A}\,\overline{C}D$ er dekket av $B\overline{C}$ og $\overline{A}\,\overline{B}D$ og trenger derfor ikke være med i uttrykket for F.

En utfordring med Quine-McCluskey-metoden er at når det blir mange inngangsvariabler, så blir det mye å holde styr på.

Etter hvert som kretsenes kompleksitet vokste, ble det et skrikende behov for å finne en metode som ikke krevde for mye regnekraft, og som samtidig kunne finne optimale forenklede uttrykk. På midten av 1980-tallet utviklet Robert Brayton ved IBM den såkalte Espresso heuristic logic minimizer. Den finnes nå i mange avarter og brukes når store design skal forenkles. Hvorfor skal vi i hele tatt lære om Karnaugh og Quine-McCluskey, når det bare er å kjøre et program og trykke på optimaliseringsknappen? Et godt spørsmål, men det er jo alltid kjekt å kunne utføre logiske øvelser med egen hjerne, forstå hvordan logiske kretser kan bygges opp fra bunnen av, og kjenne litt til den historiske konteksten. Spørsmålet ovenfor minner vel litt om det jeg fikk av mine barn i frustrerte øyeblikk: «Hvorfor skal vi lære å regne når vi har kalkulator?».

Åkkesom, på eksamen blir en spurt om elementær logikk, så det kan være profitabelt å kunne av den grunn også. Det er bare å finne seg en passende oppgave.

Oppgave

c. Finn et enklest mulig uttrykk for funksjonen i tabellen nedenfor ved å bruke K-diagram (Karnaugh)

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

d. Tegn kretsen for den forenklede funksjonen i deloppgaven ovenfor (5c). Hvis du ikke fikk til oppgave 5c, bruk $F = AB + \overline{A}C$

La oss starte med oppgave c. Da er det bare å finne frem et passende Karnaugh-diagram for tre inngangsvariabler, og fylle inn verdiene fra tabellen i oppgaven.

Tabell 5.26 Karnaugh-diagram for **F**

AB\C	0	1
00	0	1
01	X	0
11	1	1
10	1	X

X markerer «don't care» tilstander som kan tas med dersom en får større grupper. X i nederste høyre hjørne passer godt inn i en større gruppe, mens X i rad 2 er rimelig isolert. Dermed har en de fire nederste cellene (oransje) som en gruppe, og 1-er øverst til høyre kan være med i en gruppe (rød) med X nederst til høyre, siden Karnaugh-diagrammet «wrapper». De to gruppene overlapper nederst til høyre uten at det bør bekymre oss.

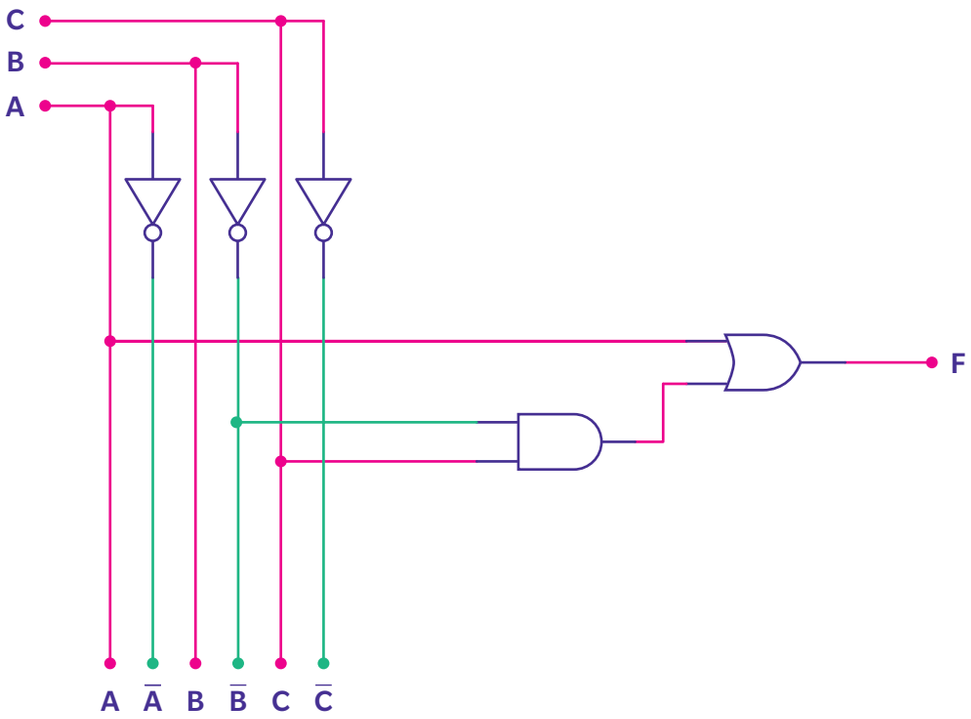
Tabell 5.27 Karnaugh-diagram for **F**

AB\C	0	1
00	0	1
01	X	0
11	1	1
10	1	X

Uttrykket for F finner en ved å se hvilke inngangsvariabler som ikke endrer seg for de to gruppene. For den oransje gruppen er A lik 1, og for den røde gruppen er B lik 0 og C lik 1.

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

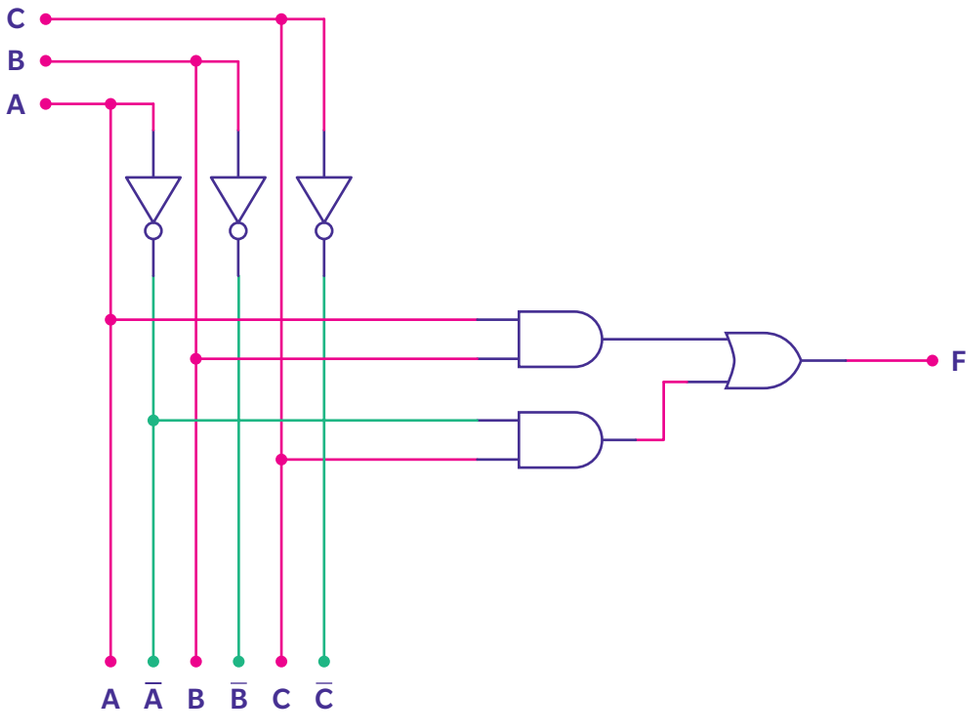
Neste oppgave er d. Vi skal realisere en krets for uttrykket for F. Siden uttrykket er på «Sum Of Products» SOP form, er det naturlig å starte med produktene og så føre dem til en eller funksjon (OR-funksjon). B inverteres i en IKKE-funksjon (NOT-funksjon) og multipliseres i en OG-funksjon (AND-funksjon) med C, og til slutt summerer en med A.



Figur 5.10 Krets for **F**

Figur 5.10 viser et forslag til realisering. Som du ser, har en invertert alle inngangene også, slik at det er lett å hente det en trenger når den logiske strukturen skal bygges opp.

De som har laget oppgaven, har vært hyggelige mot oss. Skulle vi mot formodning feile, får vi lov å prøve oss på $F = AB + \overline{A}C$ isteden. Da blir faktisk oppgaven litt vanskeligere, men det er vel prisen vi må betale. Det blir nå to AND-funksjoner og en OR-funksjon, og resultatet ser du i figur 5.11.



Figur 5.11 Krets for **F**

Det var den oppgaven og det kapittelet. Vi har nå fått det logiske grunnlaget til å bygge videre på. Logikkens DNA er på plass, og den digitale verden ligger for våre føtter. Det er bare å sette bitene sammen på riktig måte, så kan vi skape hva som helst!

6

Kapittel 6

Fra det ene til det andre

«Att og fram er like langt, inn og ut er like trangt.»

Peer Gynt, Henrik Ibsen (1828–1906)

LÆRINGSUTBYTTE: Kombinatoriske logiske funksjoner, «halvadder», «fulladder», parallelle «addere», «ripple carry» forsinkelse, «look-ahead carry», subtraktor, sammenligning, dekoder, enkoder, kodekonverter, multiplekser, demultiplekser, sekvenser, tidsdiagram, «hazard»

Jeg holder min kjære Curta i hånden mens jeg snurrer henne rundt lillefingeren. Hun er resultatorientert og ligger godt i hånden. For hver omdreining adderer hun de tallene jeg ønsker. Hun er mer enn femti år gammel, men er like vakker og perfekt som da hun forlot fabrikken. Curta er en liten, håndholdt mekanisk kalkulator som ble oppfunnet og utviklet av Curt Herzstark mens han var fange i Buchenwald under krigen. Han sørget selvfølgelig for at kalkulatoren akkurat ble ferdig ved krigens slutt. Det reddet hans liv. Curta ble ansett for å være den beste håndholdte kalkulator frem til de elektroniske overtok på 1970-tallet. Mitt velholdte eksemplar av arten er samlere villig til å betale en formue for, men man selger ikke sin kjære for noen pris.

Det viktigste en datamaskin gjør er å regne, og som vi allerede har sett, kan alle regningsartene (+, –, · og /) gjøres om til addisjon. I tillegg ønsker en at denne regningen skal gå så fort som mulig. Vi skal bruke logikkens DNA, nemlig de logiske funksjonene til å bygge mer avanserte kombinatoriske logiske funksjoner som kan addere, sammenligne, kode og multiplekse.

Figur 6.1 Curta i all sin prakt kverner resultater

En kombinatorisk logisk funksjon er beskrevet på følgende vis:

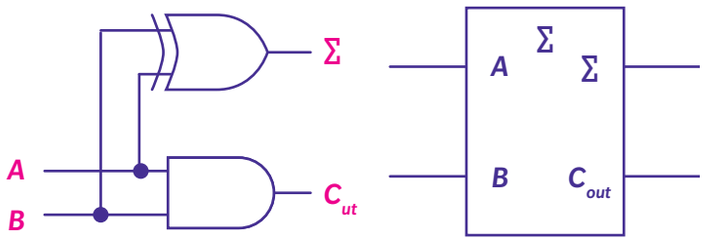
$$f_{boolsk} : \boldsymbol{B}^k \rightarrow \boldsymbol{B}^l \ \ hvor \ \boldsymbol{B} = \{0,1\}$$

Denne funksjonen tar et sammensatt utsagn med boolske variabler og leverer en funksjonsverdi med l boolske verdier. La oss prøve å lage en funksjon som adderer to bit A og B. Sannhetstabellen må da se slik ut:

Tabell 6.1 Sannhetstabell for addisjon av to bit A og B

Inn		Ut	
A	B	C_{ut}	Σ
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

Her er A og B inngangsvariablene som skal adderes, og C_{ut} er mente ved addisjonen og Σ summen. Ved å se på sannhetstabellen oppdager en at $C_{ut} = AB$ og $\Sigma = A \oplus B$. Det gir kretsen i figur 6.2 som kalles en «halvadder»:



Figur 6.2 «Halvadder» krets til venstre og tilhørende symbol til høyre

Utmerket. Da har vi laget en krets som kan addere to bit, men det hadde jo vært kjekt å kunne addere større tall. Da må vi utvide «halvadder» til «fulladder». Det «halvadder» mangler, er evnen til å kunne ta med seg mente fra mindre signifikante posisjoner. La oss utvide den sannhetstabellen vi hadde for «halvadder», til også å ha innkommende mente med som inngangsvariabel:

Tabell 6.2 Sannhetstabell for «fulladder»

Inn			Ut	
A	B	C_{inn}	C_{ut}	Σ
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

Med en «fulladder» har vi laget byggesteinen i en kombinatorisk logisk funksjon som kan addere så store tall vi bare vil. Det er bare å kjede sammen mange «fulle addere».

Oppbyggingen av en «fulladder» kan vi finne ved å se på SSOP for henholdsvis C_{ut} og Σ:

$$C_{ut} = \overline{A}BC_{inn} + A\overline{B}C_{inn} + AB\overline{C_{inn}} + ABC_{inn} = (\overline{A}B + A\overline{B})C_{inn} + AB(\overline{C_{inn}} + C_{inn}) = (A \oplus B)C_{inn} + AB$$

$$\Sigma = \overline{A}\,\overline{B}C_{inn} + \overline{A}B\overline{C_{inn}} + A\overline{B}\,\overline{C_{inn}} + ABC_{inn} = (\overline{A}B + A\overline{B})\overline{C_{inn}} + (\overline{A}\,\overline{B} + AB)C_{inn} =$$
$$(A \oplus B)\overline{C_{inn}} + (A \oplus \overline{B})C_{inn} = (A \oplus B)\overline{C_{inn}} + (\overline{A \oplus B})C_{inn} = (A \oplus B) \oplus C_{inn}$$

Den nest siste overgangen i Σ brukte jeg litt tid på å akseptere, men ved hjelp av De Morgans lover og boolsk OG-distribusjon er det mulig å vise at:

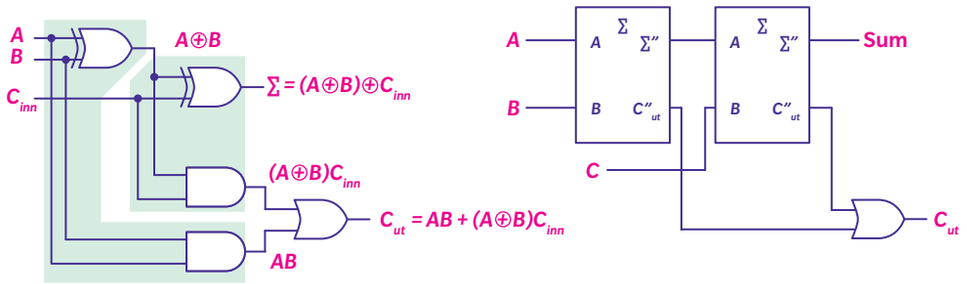
$$\overline{A \oplus B} = \overline{\overline{A}B + A\overline{B}} = \overline{\overline{A}B}\,\overline{A\overline{B}} = (\overline{\overline{A}} + \overline{B})(\overline{A} + \overline{\overline{B}}) = (A + \overline{B})(\overline{A} + B) = (A + \overline{B})\overline{A} + (A + \overline{B})B$$
$$= A\overline{A} + \overline{B}\,\overline{A} + AB + B\overline{B} = \overline{A}\,\overline{B} + AB = A \oplus \overline{B}$$

Dersom du enda ikke er overbevist, er det bare å sette opp en sannhetstabell og sammenligne:

Tabell 6.3 Sannhetstabell for å finne ut at $A \oplus \overline{B} = \overline{A \oplus B}$

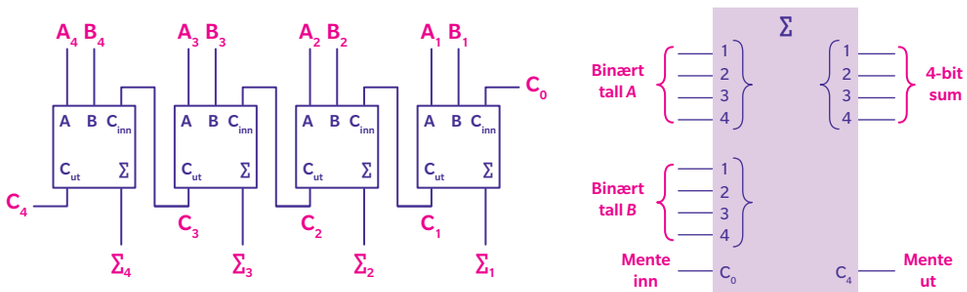
A	B	$A \oplus B = \overline{A}B + A\overline{B}$	$\overline{A \oplus B}$	$A \oplus \overline{B} = \overline{A}\overline{B} + AB$
0	0	0	1	1
0	1	1	0	0
1	0	1	0	0
1	1	0	1	1

Siden en «halvadder» XORer sin sum og ANDer sitt mente, kan en «fulladder» settes sammen av to «halvaddere» på følgende vis:



Figur 6.3 «Fulladder» krets til venstre og tilhørende symbol til høyre

En parallell «adder» er sammensatt av flere «fulladdere», slik at en kan legge sammen binære tall. I figur 6.4 er en 4 bit «adder» vist.



Figur 6.4 4 bit parallell «adder» krets. Oppbygging til venstre og tilhørende symbol til høyre

Dersom en nullstiller C_0 i kretsen ovenfor, utføres følgende regnestykke:

$$A_4A_3A_2A_1$$

$$+B_4B_3B_2B_1$$

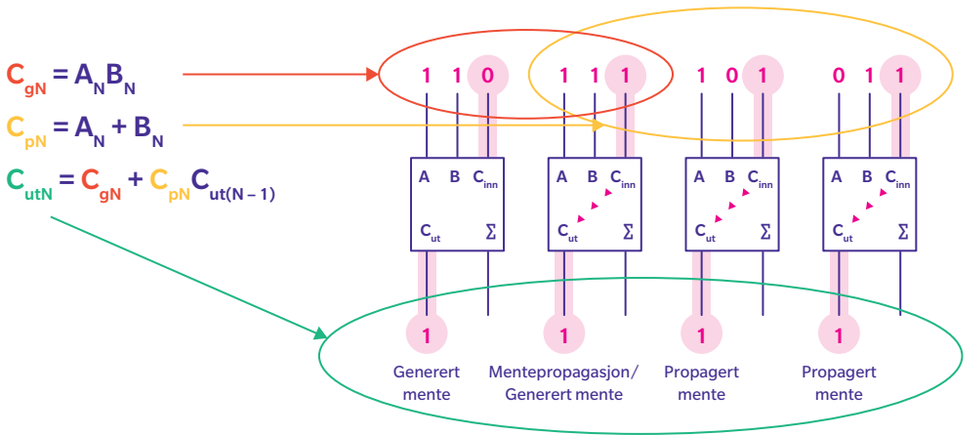
$$C_4\,\Sigma_4\,\Sigma_3\,\Sigma_2\,\Sigma_1$$

Da er det bare å dure i vei og addere så mange fulle «addere» en bare kan tenke seg, for å lage så store tall som en måtte ønske seg. Det er bare en hake. Det tar tid å regne ut mente, og jo større den sammensatte kretsen blir, jo lengre tid tar det før siste mente er blitt regnet ut. Dette problemet har til og med fått sitt eget navn – «ripple carry».

Er det mulig å komme rundt dette problemet? Absolutt! Løsningen ligger i observasjonen at vi har $A_1 \ldots A_n, B_1 \ldots B_n$ tilgjengelig når regningen starter. Utfordringen blir å finne et uttrykk for mente basert direkte på dem uten altfor mange operasjoner.

Hva skal til for at mente dannes? La oss se på en «fulladder». Dersom A og B begge er 1, får vi mente. Det kalles «carry generation» (mentegenerering) $C_g = AB$. Dersom C_{inn} er 1, er det mulig å få «carry propagation» (mentepropagasjon) $C_p = A + B$ når C_p er forskjellig fra 0. Totalt sett vil en få mente gitt ved:

$$C_{ut} = C_g + C_p C_{inn}$$



Figur 6.5 Tilstander som gir «carry generation» og «carry propagation»

Med dette kan en starte å bygge opp en struktur som regner mente mer effektivt:

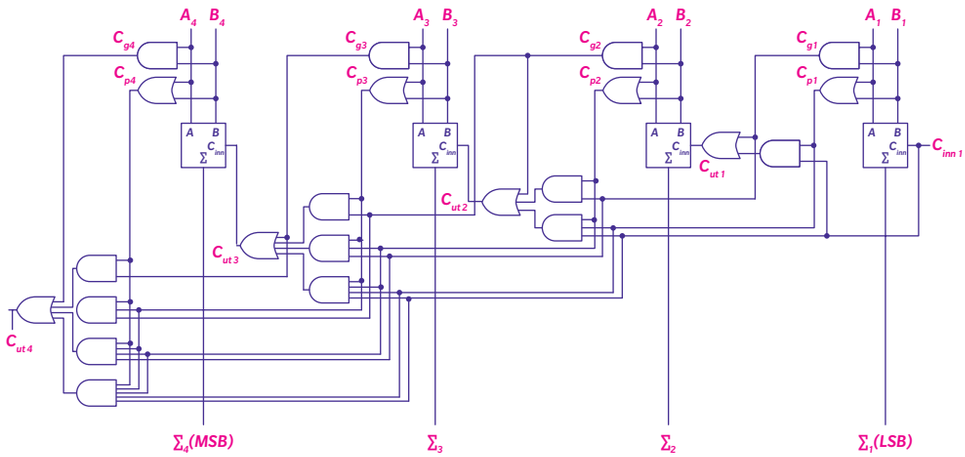
$$C_{ut0} = C_{inn}$$

$$C_{ut1} = C_{g1} + C_{p1}C_{ut0} = A_1B_1 + (A_1 + B_1)C_{inn}$$

$$C_{ut2} = C_{g2} + C_{p2}C_{ut1} = A_2B_2 + (A_2 + B_2)(A_1B_1 + (A_1 + B_1)C_{inn})$$

…

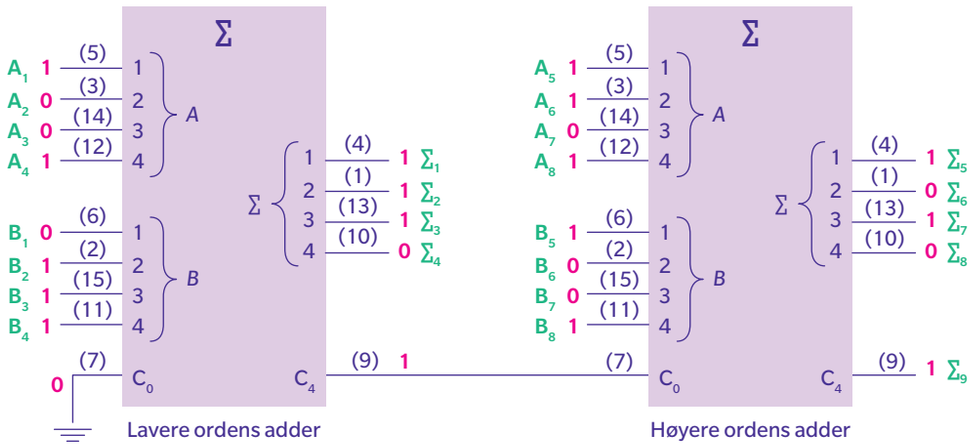
En ser at menter nå kun beregnes av operandverdiene $A_1 \ldots A_n, B_1 \ldots B_n$ og laveste mente. Figur 6.6 viser det logiske diagrammet for en 4 bit «look-ahead carry» (forhåndsmente).



Figur 6.6 4 bit «look-ahead carry»

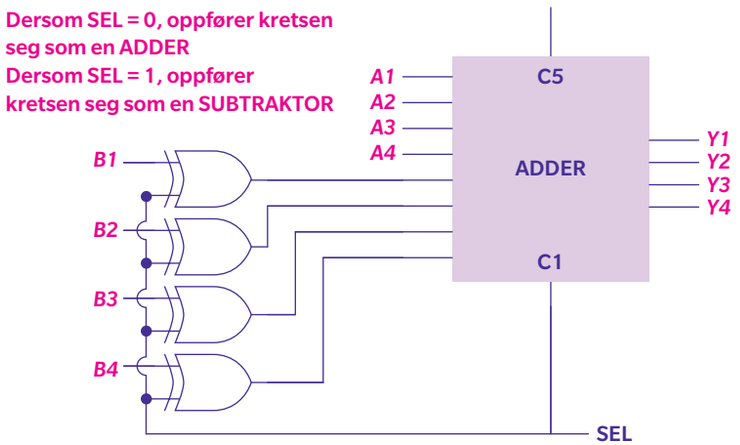
En «look-ahead carry» har liten forplantningsforsinkelse sammenliget med en «ripple carry adder». Forsinkelsen i en 4 bit «look-ahead carry» er 3 porter. Den første når C_p og C_g genereres for alle $A_1 \ldots A_n, B_1 \ldots B_n$, den neste når C_p settes sammen med resultater fra mindre signifikante bit, og den siste når det hele ORes til C_{ut}. Eneste ulempen med «look-ahead carry» er at den bruker flere porter og dermed mer areal.

Dersom en ønsker å lage større «addere», kan de settes sammen av mindre. I figur 6.7 er det laget en 8 bit «adder» av to 4 bit «addere» av typen 74HC283. 74HC283 er navnet på en fysisk krets som implementer 4 bit «adder». Tallene som adderes, er $A_8A_7A_6A_5A_4A_3A_2A_1 = 10111001$ og $B_8B_7B_6B_5B_4B_3B_2B_1 = 10011110$.



Figur 6.7 8 bit «adder» laget av to 4 bit «addere».
Tallene i parentes er pin-konfigurasjon for 74HC283

Så var det kunsten å subtrahere. Vi har tidligere sett at det kan gjøres ved hjelp av addisjon dersom de binære tall skrives på 2ers komplements form. Følgende logiske krets kan brukes både til addisjon og subtraksjon.



Figur 6.8 4 bit «adder» som både kan addere og subtrahere

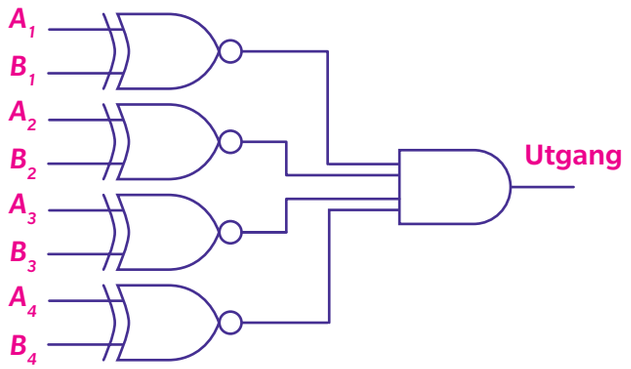
Ved å sette *SEL* til 0 vil kretsen i figur 6.8 utføre normal addisjon. Dersom *SEL* settes lik 1, vil bit $B1$ til $B4$ inverteres (1ers komplement), og så adderes 1 ved å sette $C1$ til 1 ved hjelp av *SEL* som nå er 1. Dermed har en 2ers komplement av tallet $B1$ til $B4$, og kretsen oppfører seg som en subtraktor.

«Min far er sterkere enn din far.» Det har til alle tider vært et ønske om å sammenligne størrelser. Så også binære. For å sammenligne bit er det greit å bruke XNOR funksjonen, for den har den ønskede egenskapen at funksjonsverdien er når inngangsverdier er like.

Tabell 6.4 Sannhetstabell for XNOR

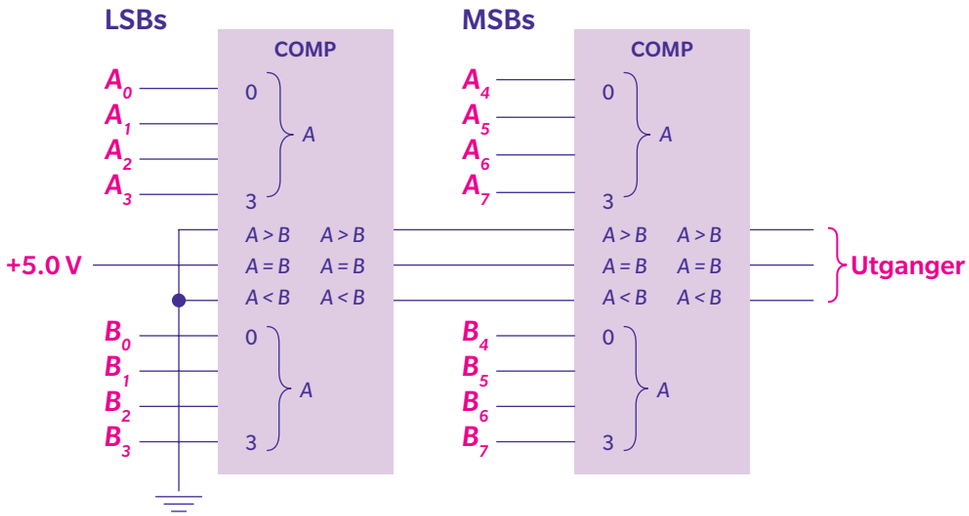
A	B	$\overline{A \oplus B} = \overline{\overline{A}B + A\overline{B}}$
0	0	1
0	1	0
1	0	0
1	1	1

Dersom en ønsker å sammenligne større tall, er det bare å sammenligne bit for bit og ta AND-funksjonen til slutt.



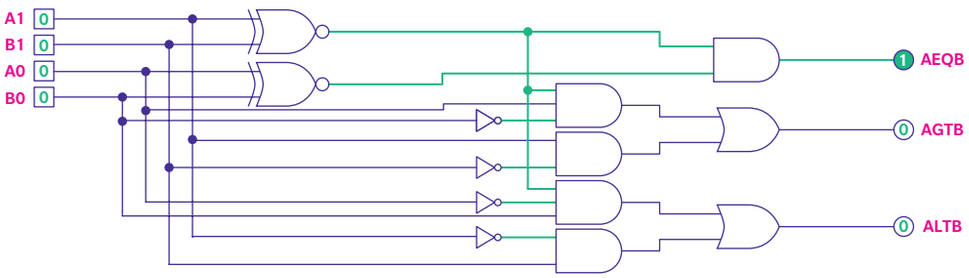
Figur 6.9 Sammenligning av 4 bit

Det finnes dedikerte kretser for sammenligning, såkalte komperatorer. De forteller også hvilket tall som er størst og minst dersom tallene er ulike. Komperatorene kan settes sammen slik at større tall kan sammenlignes. I figur 6.10 er to kretser (74LS85) satt sammen for å sammenligne to byte. LSB har 0 som inngangsverdi for $A = B$ og 0 for $A > B$ og $A < B$.



Figur 6.10 Sammenligning av 8 bit

For å forstå oppbyggingen av en komperator kan det være greit å se på et overkommelig eksempel, nemlig 2 bit komperatoren i figur 6.11. Inngangsverdiene er A_1A_0 og B_1B_0. Tallet A er større enn B dersom $A_1 = 1$ og $B_1 = 0$ eller dersom $A_1 = B_1$ og $A_0 = 1$ og $B_0 = 0$. Figur 6.11 viser hvordan 2 bit komperatoren er bygd med logiske funksjoner. AGTB (A Greater Than B) ser vi er bygd opp av en AND-funksjon som tar imot 1 dersom $A_1 = B_1$, 1 dersom $A_0 = 1$ og 1 dersom $B_0 = 0$. Nedenfor er det enda en AND-funksjon som tar imot 1 dersom $A_1 = 1$ og dersom $B_1 = 0$. De to AND-funksjonen settes sammen med en OR-funksjon, og dermed blir AGTB 1 dersom de to ulike kriteriene for at A er større enn B er oppfylt. For ALTB (A Less Than B) er det bare å bygge en logikk hvor A og B bytter roller. For likhet AEQB (A EQual B) har vi den samme type logikk som vi så i figur 6.9.

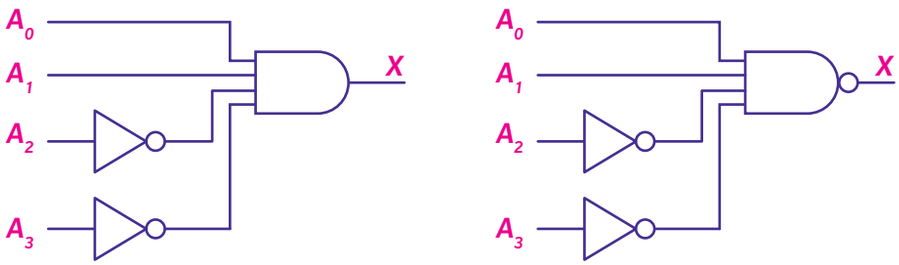


Figur 6.11 2 bit komperator

Vi har nå sett på en del viktige kombinatoriske logiske funksjoner. Finnes det flere? Vel, egentlig er det kun fantasien som setter grenser, men vi skal i det videre konsentrere oss om noen standardiserte funksjoner. Først ut er dekoder og enkoder. De utfører

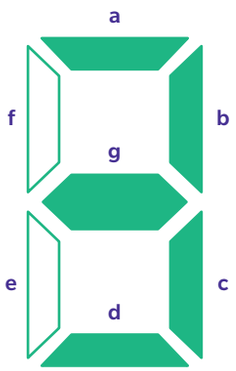
inverse funksjoner av hverandre. En dekoder har innganger og opptil $m = 2^n$ utganger. m er det maksimale antall variasjonsmuligheter en har dersom en bruker n bit. En enkoder får man dersom rollene til innganger og utganger byttes.

En dekoder «detekterer» et gitt bitmønster på inngangene, og det bestemmer utgangenes verdier. En «aktiv Høy utgang»-dekoder sender logisk HØY på utgang ved deteksjon av bitmønsteret som skal detekteres, og «aktiv Lav utgang» gjør det motsatte. I figur 6.12 er «aktiv Høy» og «aktiv Lav» gitt for en 4 til 1 dekoder.



Figur 6.12 «aktiv Høy» og «aktiv Lav» dekodere for $A_3, A_2, A_1, A_0 = 0011$

La oss ta et eksempel på en mer anvendelig dekoder. Den du kanskje har i klokkeradioen på nattbordet, syv segments dekoderen som konverterer BCD kode til desimaltall. Den har fire innganger D_3 til D_0, og når $D_3, D_2, D1, D_0 = 0011 = (3)_{10}$ lyser tretallet mot deg.



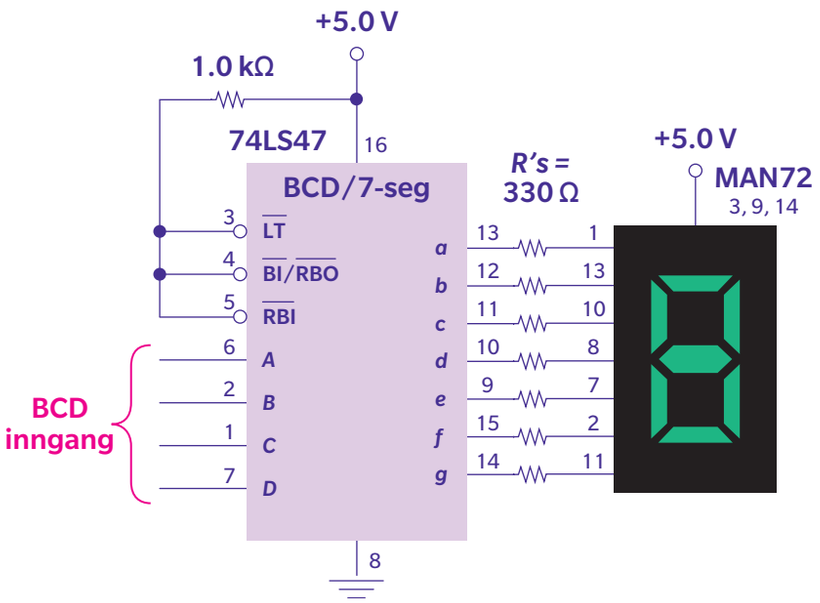
Figur 6.13 Syv segment «display»

For å kunne skape alle tallene fra til trenger dekoderen syv utganger som representerer a til g. Dersom dekoderen er av typen «aktiv Høy», vil $a, b, c, d, e, f, g = 1111001$ vise sifferet 3, men hvis den er «aktiv Lav», vil $a, b, c, d, e, f, g = 0000110$ gjøre det. Sannhetstabell for «aktiv Lav» dekoder:

Tabell 6.5 Sannhetstabell for «aktiv Lav» BCD til desimal dekoder

Desimaltall	D_3	D_2	D_1	D_0	a	b	c	d	e	f	g
0	0	0	0	0	0	0	0	0	0	0	1
1	0	0	0	1	1	0	0	1	1	1	1
2	0	0	1	0	0	0	1	0	0	1	0
3	0	0	1	1	0	0	0	0	1	1	0
4	0	1	0	0	1	0	0	1	1	0	0
5	0	1	0	1	0	1	0	0	1	0	0
6	0	1	1	0	1	1	0	0	0	0	0
7	0	1	1	1	0	0	0	1	1	1	1
8	1	0	0	0	0	0	0	0	0	0	0
9	1	0	0	1	0	0	0	1	1	0	0
Ugyldig	1	0	1	0	X	X	X	X	X	X	X
	1	0	1	1	X	X	X	X	X	X	X
	1	1	0	0	X	X	X	X	X	X	X
	1	1	0	1	X	X	X	X	X	X	X
	1	1	1	0	X	X	X	X	X	X	X
	1	1	1	1	X	X	X	X	X	X	X

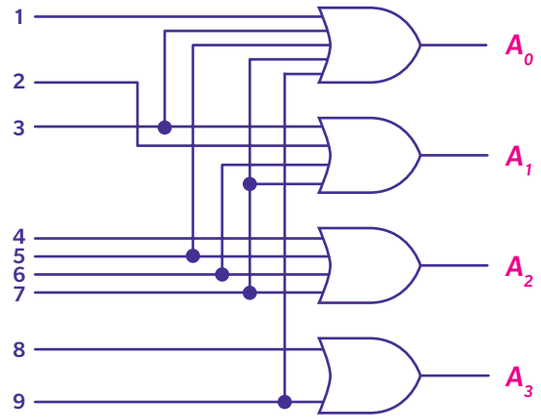
Figur 6.14 viser oppkobling av en syv segments dekoder basert på dekoderkretsen 74LS47.



Figur 6.14 Syv segments dekoder

Hva med inngangene $\overline{LT}$, $\overline{BI}/\overline{RBO}$ og $\overline{RBI}$ på denne dekoderen? Slår en opp i et datablad som beskriver denne kretsen, kan en finne ut at dersom $\overline{LT}$ og $\overline{BI}/\overline{RBO}$ er HØYE, testes «displayet». $\overline{BI}/\overline{RBO}$ sammen med $\overline{RBI}$ brukes til å undertrykke nuller dersom en har mange segmenter. Dersom en har tallet 030,080, vil 30,08 vises med nullundertrykkelse.

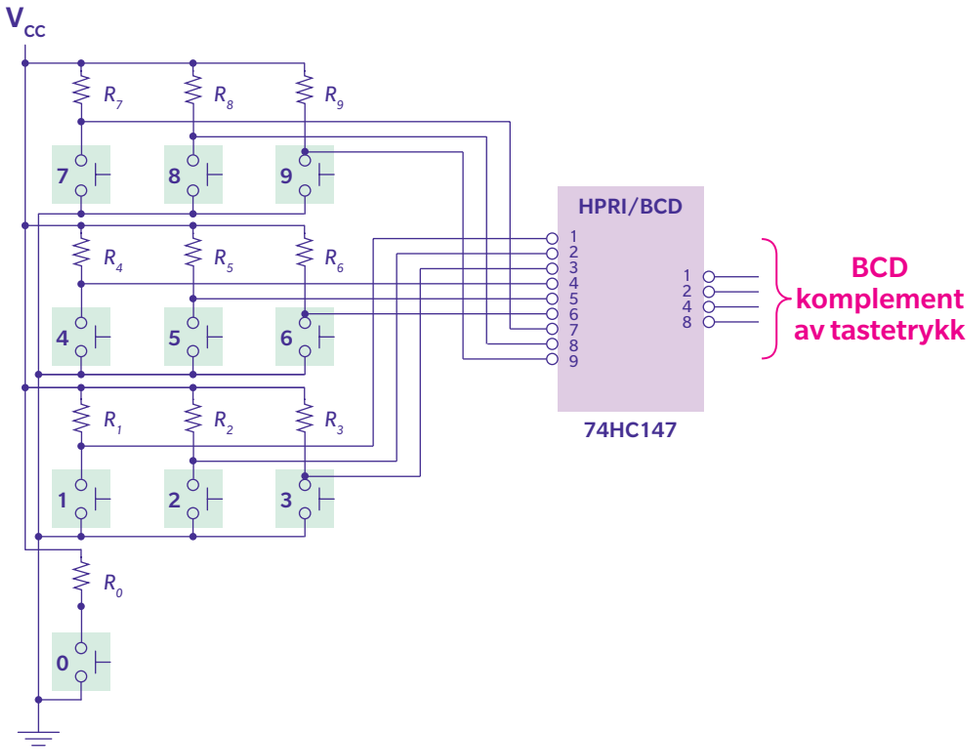
En enkoder er en digital krets som på basis av en eller flere aktive innganger genererer en spesifikk kode på utgangene. Som tidligere nevnt, er en enkoder dekoderens motsats. La oss se på en desimal til BCD enkoder som gjengitt i figur 6.15.



Figur 6.15 Desimal til BCD enkoder

Dersom desimaltallet 7 skal enkodes, settes inngang 7 til HØY, mens alle de andre settes til LAV. Inngang 7 forgreines til de tre OR-funksjonene foran A_2, A_1 og A_0 som alle blir HØY. Resultatet blir $A_3A_2A_1A_0 = 0111$ som forventet av en desimal til BCD dekoder.

Figur 6.16 viser en praktisk implementasjon av et tastatur som bruker en desimal til BCD-komplement enkoder (Digital Fundamentals [2] side 344). Denne enkoderen er i tillegg en såkalt prioritetsenkoder. Den prioriterer en av inngangene dersom flere skulle være HØYE samtidig.



Figur 6.16 Desimal til BCD-komplement enkoder med prioritet

I tabell 6.6 ser en at en har gitt største desimaltall prioritet dersom flere sifre tastes samtidig.

Tabell 6.6 Sannhetstabell for prioritetsenkoder

D_9	D_8	D_7	D_6	D_5	D_4	D_3	D_2	D_1	D_0	BCD_8	BCD_4	BCD_2	BCD_1
0	0	0	0	0	0	0	0	0	1	0	0	0	0
0	0	0	0	0	0	0	0	1	X	0	0	0	1
0	0	0	0	0	0	0	1	X	X	0	0	1	0
0	0	0	0	0	0	1	X	X	X	0	0	1	1
0	0	0	0	0	1	X	X	X	X	0	1	0	0
0	0	0	0	1	X	X	X	X	X	0	1	0	1
0	0	0	1	X	X	X	X	X	X	0	1	1	0
0	0	1	X	X	X	X	X	X	X	0	1	1	1
0	1	X	X	X	X	X	X	X	X	1	0	0	0
1	X	X	X	X	X	X	X	X	X	1	0	0	1

Et annet eksempel på en kombinatorisk krets er kodekonvertere. La oss se på en enkel en som konverterer fra Gray til binært, og en som gjør det motsatte. Vi har tidligere sett i kapittelet «Tenk på et tall» at gray-kode produseres enkelt fra et binært tall med følgende prosedyre:

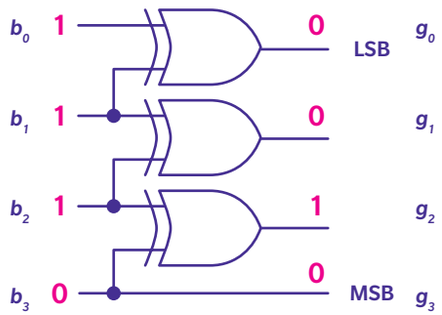
$$g_3 = b_3$$

$$g_2 = b_3 \oplus b_2$$

$$g_1 = b_2 \oplus b_1$$

$$g_0 = b_1 \oplus b_0$$

En kombinatorisk krets som utfører prosedyren, kan se sånn ut:



Figur 6.17 Binær til gray-kode

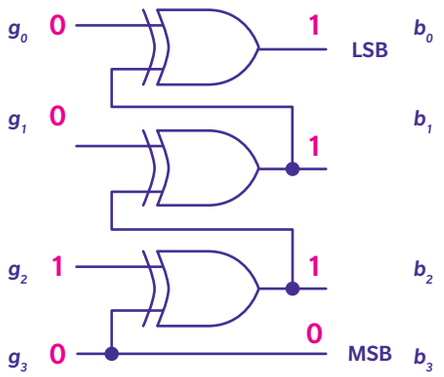
Og tilsvarende har vi for motsatt vei:

$$b_3 = g_3$$

$$b_2 = b_3 \oplus g_2$$

$$b_1 = b_2 \oplus g_1$$

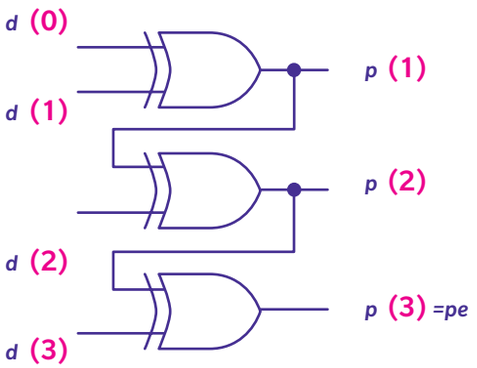
$$b_0 = b_1 \oplus g_0$$



Figur 6.18 Gray-kode til binær

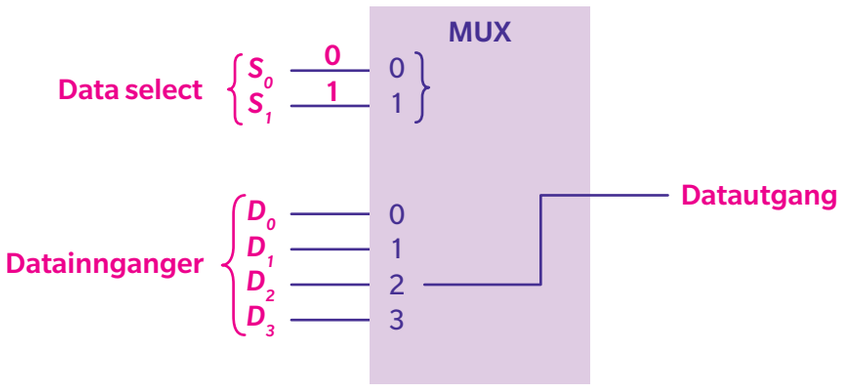
En annen type koder er en som lager paritetssjekk til et ord. Paritetssjekk brukes når data skal sendes fra et sted til et annet. En legger til et ekstra bit til dataordet som sendes, slik at en kan oppdage enkeltfeil ved transmisjon. Du kan lese mer om bruk av paritetssjekk i kapittelet «Fra A til B».

En 2 bit paritetsgenerator kan lages ved å bruke XOR-funksjonen. Når de to databit er henholdsvis 01 eller 10, vil utgangen bli 1. Denne utgangsverdien kan så legges til for å få lik paritet. Dersom det er flere bit som skal ha jevn paritetssjekk, kan en legge til flere XOR-porter.



Figur 6.19 Jevn paritetgenerator for et 4 bit ord

Den neste store gruppen av kombinatoriske kretser er multiplekser/demultiplekser. De utfører som navnene tilsier, motsatte operasjoner. En multiplekser har to sett med innganger «data» og «select». Multiplekseren velger en av datainngangene og sender dens signal på utgangen. Hvilken datainngang som velges, bestemmes av verdiene på «select»-inngangene.



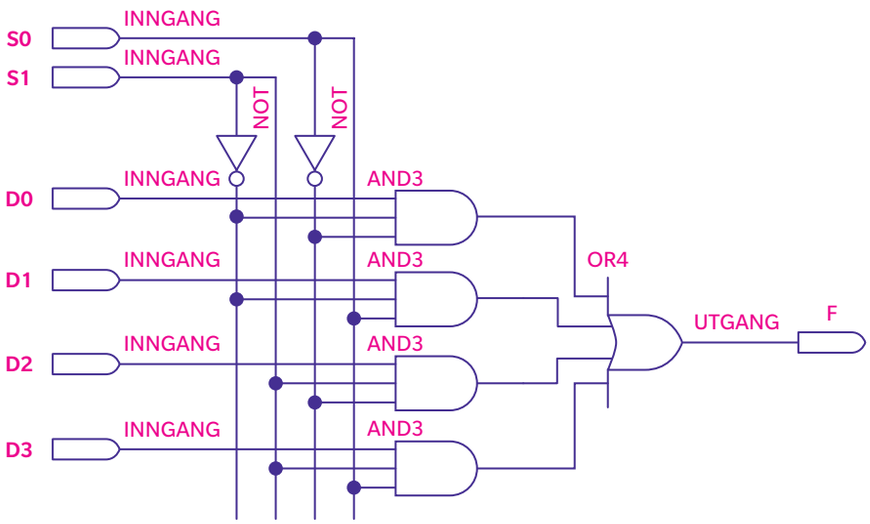
Figur 6.20 Multiplekser

I figur 6.20 ser vi at «select» har verdien $(10)_2 = (2)_{10}$, og derfor velges inngang D_2.

En N-til-1 trenger M «select»-innganger slik at $2^M = N$. For eksempel trenger en 4-til-1 multiplekser to «select»-innganger ($N = 4$ og da blir $M = 2$ fordi $2^2 = 4$) slik som vist i figur 6.20.

Den logiske ligningen for en 4-til-1 multiplekser er gitt ved:

$$F = D_0(\overline{S_0}\,\overline{S_1}) + D_1(S_0\,\overline{S_1}) + D_2(\overline{S_0}S_1) + D_2(S_0S_1)$$



Figur 6.21 4-til-1 multiplekser logikk

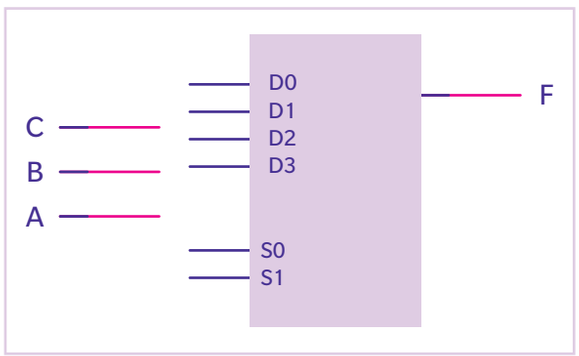
En demultiplekser gjør det motsatte av en multiplekser. Den har en datainngang, en eller flere «select»-innganger og flere utganger. Hvilken datautgang som brukes av inngangsdata, bestemmes av verdien på «select»-inngangene. En 1-til- N demultiplekser trenger M «select»-innganger, slik at $2^M = N$.

En viktig multiplekser/demultiplekser applikasjon er tidsdelt multipleksing (Time Division Multiplexing TDM). I det tilfellet kobles en teller til «select»-inngangene slik at hver strøm av data som kommer mot inngangene, får sin tidsluke for å sende et bit, en byte, et ord eller mer. Her brukes ordet strøm av data. Vår logiske verden er ikke lenger stasjonær, men endres over tid. Vi skal i neste kapittel se nærmere på den sekvensielle logikken og hvilke muligheter den gir oss, men før vi slutter helt, er det på høy tid å bryne seg på en eksamensoppgave. Er det mulig å omsette våre kunnskaper til ferdigheter?

Oppgave

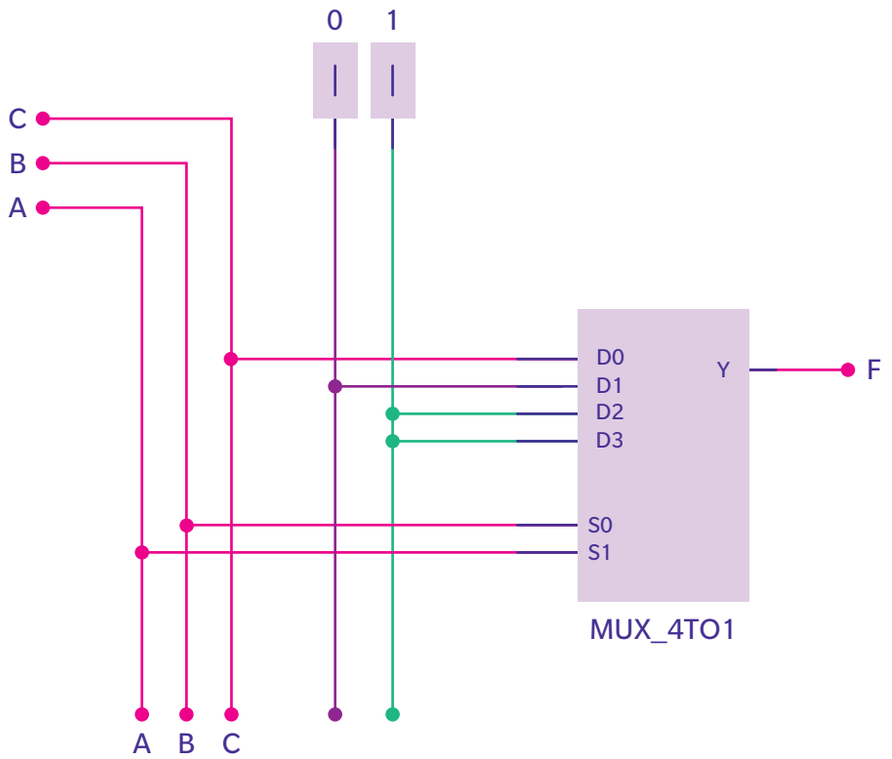
e. Hvordan kan funksjonen nedenfor implementeres ved å benytte en 4-til-1 multiplekser?

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



Ja, det var et godt spørsmål. Tenke, tenke. «Select»-inngangenes verdi bestemmer altså hvilken inngang som velges å sendes til utgangen. Kanskje A og B kunne kobles mot henholdsvis S_1 og S_0? Ved å gå igjennom tabellen vil da alle inngangene bli brukt. Selvfølgelig kunne vi brukt en hvilken som helst annen kombinasjon. For eksempel ville B og C fungert like godt, men A og B sto lagelig til for hogg. Når A og B begge er 0, vil utgang D_0 brukes, og da er F lik C. D_0 kan derfor kobles direkte mot C. Når A er 0 og B er 1 brukes D_1, og da er F lik A. D_1 kan derfor kobles direkte mot A. For de to gjenværende tilfellene, de siste fire rader i tabellen, er $F = A$, og dermed er $D_2 = A$ og $D_3 = A$. Det er på tide med en sniktitt i fasit.

A	B	C	F	
0	0	0	0	For disse to radene er *F = C*
0	0	1	1	
0	1	0	0	For disse to radene er *F = 0*
0	1	1	0	
1	0	0	1	For disse to radene er *F = 1*
1	0	1	1	
1	1	0	1	For disse to radene er *F = 1*
1	1	1	1	

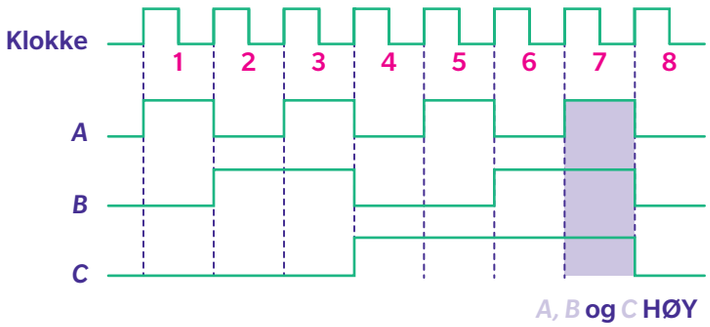


Figur 6.22 Fasit

Det så litt annerledes ut. Vi er enige om å bruke A og B på «select»-inngangene og at D_0 kan hentes fra C. Hvorfor faste verdier er innført istedenfor å bruke A, må du spørre dem som har laget oppgaven om. Åkkesom, resultatet blir det samme.

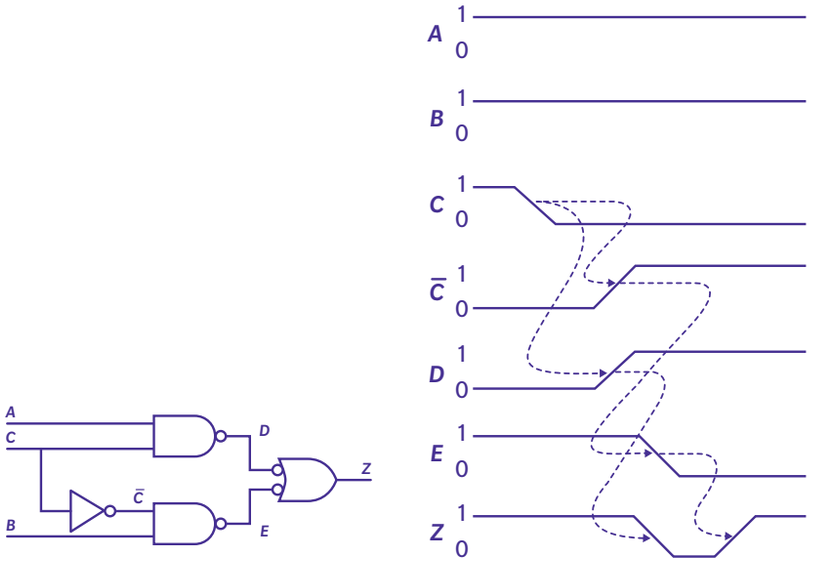
En ting til sånn helt på tampen. Det er vanlig at en utsetter kombinatoriske logiske funksjoner for sekvenser av 0 og 1 på inngangene. De tegnes ofte ved hjelp av såkalte tidsdiagrammer. Et eksempel på et slikt diagram er vist i figur 6.23. Øverst er det et

såkalt klokkesignal, og de andre verdiene kan bare endre seg når klokken gjør det. I dette tilfellet når klokkepulsen stiger.



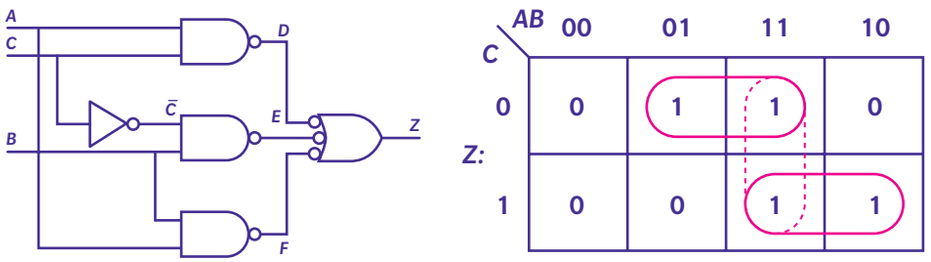
Figur 6.23 Eksempel på tidsdiagram

Hva så? Det skulle jo ikke bety noe fra eller til. Funksjonsverdien er jo bestemt av de til enhver tid gitte inngangsverdier. Hadde det bare vært så vel. Problemet er at ting tar tid. Det er forsinkelse i de logiske kretsene, riktignok i størrelsesorden nanosekunder, men lenge nok til å få uønskede tilstander fordi noen signalveier gjennom en krets er raskere enn andre. I figur 6.24 ser vi et eksempel på en krets med tilhørende tidsdiagram, hvor vi får problemer med funksjonsverdien Z (som skal være 1 hele tiden) i et lite øyeblikk, hvor den blir 0, fordi C og dens inverterte $\overline{C}$ brukes. Dette kalles en «hazard». Legg merke til at overgangene fra 0 til 1 og vice versa er skrå av den enkle grunn at det er vanskelig å gå momentant fra den ene verdien til den andre (figur 6.23 viser en ideell verden).



Figur 6.24 Krets med «hazard» og dens tidsdiagram

Er det mulig å unngå dette? Absolutt. Det en gjør, er å endre kretsen slik at funksjonen har sin ønskede verdi 1 hele tiden. I figur 6.25 ser en den omarbeidede kretsen og tilhørende Karnaugh-diagram hvor en har benyttet seg av den ekstra stiplede termen som gir funksjonsverdi 1 når både A og B er 1.



Figur 6.25 Krets som er uten «hazard» med tilhørende Karnaugh-diagram

7

Kapittel 7

Tingenes tilstand

«Skjebnen leder den villige, men trekker den motvillige etter håret.»

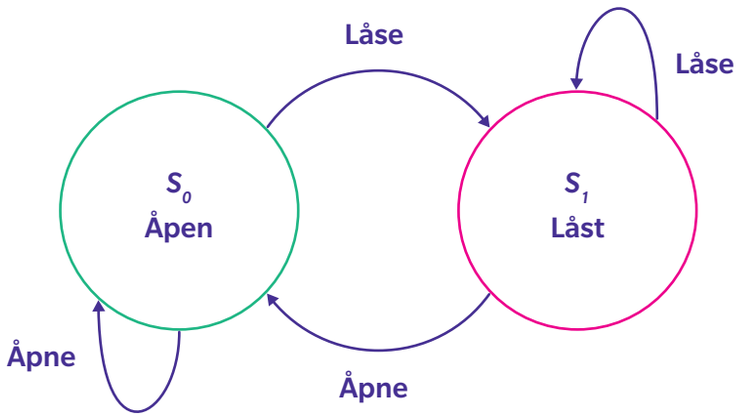
Maskinen, Oswald Spengler (1880–1936)

LÆRINGSUTBYTTE: Endelig tilstandsmaskin FSM (Finite State Machine), sekvensiell logikk, minne, tilbakekobling, låser, vipper, klokke, synkrone og asynkrone innganger, applikasjoner

Det lukkede roms mysterium har fascinert til alle tider. Min mor åpner gatedøren. Utenfor står en politimann og jeg. Sistnevnte i pyjamas. «Er dette din sønn? Han sto og dirigerte morgenrushet på Landåstorget.» Min mor ser på meg. «Hvordan kom du deg ut? Grinden foran sengen din er jo låst.» Den treårige utbryterkongen Houdini velger taushet. Ikke kan hun spørre låsen heller. Den kjenner jo bare sin nåværende tilstand. Hvor lenge den har vært låst eller hvor mange ganger den har blitt åpnet og lukket, ligger utenfor låsens hukommelsesevner. Forklaringen på gåten er banal. Jeg løftet kun litt på madrassen, skjøv sengefjølene til side og krøp ut i frihet. Sekvensiell logikk i praksis. Dette er mitt første minne. Det er muligens falskt.

Hvordan kan vi gi en logisk krets minne? At det er mulig, ser vi jo i all elektronikken vi omgir oss med. I de kretsene vi har sett på så langt, har funksjonsverdien vært gitt av de til enhver tid gitte inngangsverdier – kombinatorisk logikk. Det vi ønsker nå, er at en krets skal klare å huske sin nåværende verdi og kunne endre den til en annen verdi når nye inngangsverdier kommer. La oss starte med det enkleste elektriske minnet. Hva må til for å huske 0 og 1? Dette minner jo mistenkelig om virkemåten til en mekanisk lås.

En lås har også to tilstander. Enten er den åpen, eller så er den låst. I denne omgang er vi rause og ser bort ifra mulig vranglås. Vi kan gå fra en tilstand til en annen ved å bruke nøkkel. Vi kan låse en åpen lås og åpne en lukket lås. Vi kan selvfølgelig også prøve å låse en låst lås eller åpne en åpen lås, men i de tilfellene vil vi bare havne tilbake i den tilstanden vi er i. Vi har laget oss følgende enkle tilstandsmaskin som vist i figur 7.1:

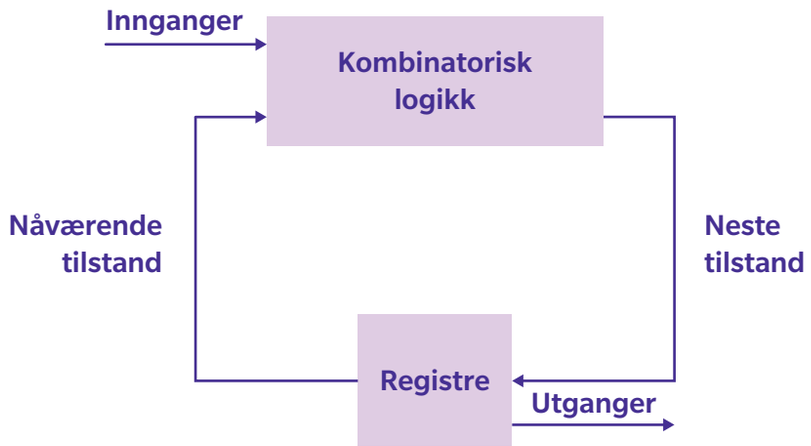


Figur 7.1 Tilstandsmaskin for en lås

Tilstandsmaskinen vi har laget i figur 7.1 er en endelig tilstandsmaskin (Finite State Machine FSM), da den består av et tellbart antall tilstander. To for å være helt presis, nemlig S_0 og S_1 markert med henholdsvis grønn og rød ring. Inne i ringene er også tilstanden gitt. Pilene mellom ringene viser hva som skjer dersom vi prøver å gå fra

tilstand til tilstand med forskjellig inngangsverdi. Dersom tilstandsmaskinen er i S_0 som representerer «Åpen» tilstand, vil en ved å gi inngangsverdi «Låse» gå til tilstand, og «output» vil være tilstandsverdien i S_1 som er «Låst». Dersom inngangsverdi «Åpne» gis når tilstanden er S_0, vil maskinen fortsatt være i tilstand S_0 som er «Åpen». Dersom vi prøver å åpne en åpen lås, vil den fortsatt være åpen. På tilsvarende vis vil en gå fra S_1 til S_0 med inngangsverdi «Åpne» og fra S_1 til S_1 med inngangsverdi «Låse».

Vår lås er et enkelt eksempel på sekvensiell logikk, men FSM kan brukes til å holde orden på store sekvensielle logiske operasjoner. Låsen er en avart av en type endelig tilstandsmaskin (FSM) som kalles en Moore-maskin etter den amerikanske professor Edward F. Moore som introduserte den i artikkelen «Gedanken-experiments on Sequential Machines» som han skrev sammen med Claude Shannon i 1956. Den elektriske varianten av Moore-maskinen er gjengitt i figur 7.2.



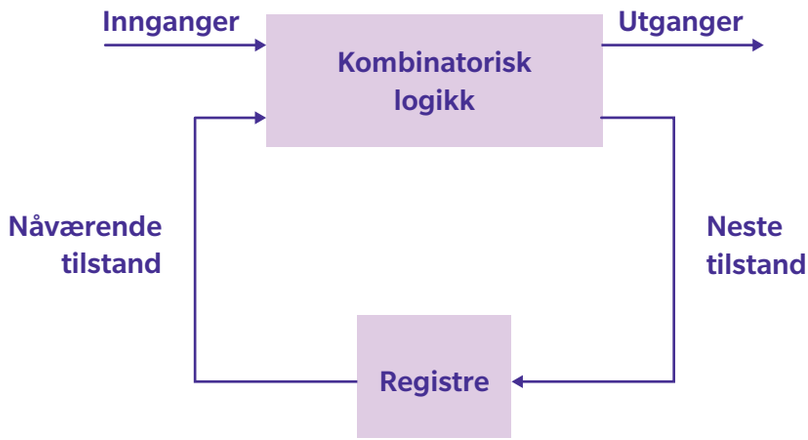
Figur 7.2 Moore-maskin

Moore-maskinen har et registrer/minne som inneholder nåværende tilstand. Nåværende tilstand kombineres med inngangsverdier i den kombinatoriske logikken til en ny tilstand som lagres samtidig som resultatet leveres på utgangen. Dette gir sekvensiell logikk, da det ikke bare er inngangsverdier som bestemmer utgangsverdier, men også den nåværende tilstand. En måte å se det på er at Moore-maskinen er en boolsk funksjon hvor sannhetstabellen endrer seg for hver nye tilstand. Et eksempel kan være en elektrisk kodelås hvor en riktig sekvens av siffer åpner låsen, mens alle andre sekvenser sørger for at låsen forblir låst.

Tilstandsmaskiner brukes i en rekke fagfelt utenfor vårt eget, som for eksempel lingvistikk, datateknologi, filosofi, biologi, matematikk og logikk. Endelige tilstandsmaskiner (FSM) er et godt verktøy for å lage kompakt sekvensiell logikk hvor utfallet av en hendelse er bestemt av nåværende tilstand. FSM gjør det også lettere å oppdage

logiske feil, såkalte deadlocks, hvor en kommer inn i en tilstand og blir værende der til evig tid.

Siden vi har en tilstandsmaskin med navn, betyr det at vi har flere. En annen variant er den såkalte Mealy-maskinen som er gitt i figur 7.3.

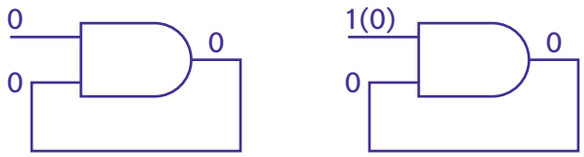


Figur 7.3 Mealy-maskin

I motsetning til Moore-maskinen gir Mealy-maskinen utgangsverdier rett fra den kombinatoriske logikken. Det er Moore-maskinen som er mest brukt. Vi skal senere se på en rekke tilstandsmaskiner når vi skal lage tellere og annen sekvensiell logikk. Vi må bare først lage oss et elektrisk minne, slik at kunnskap om tilstanden kan huskes.

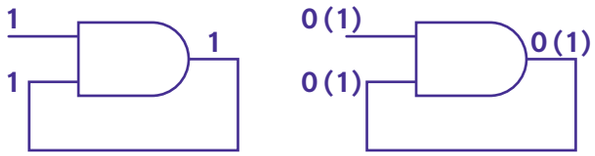
En lås er et eksempel på en enhet med minne. Minnet er riktignok begrenset til kunnskap om nåværende tilstand – «Åpen» eller «Låst». Minnet er binært. Dersom en ønsker å utvide minnet, er det bare å pøse på med flere låser. Ja, en kunne laget rene huskestuen og sette sammen en datamaskin bestående av fysiske låser. En slik datamaskin ville blitt upraktisk, så la oss heller prøve å gjøre låsen elektrisk (kalles «latch») og bruke 0 som «Åpen» og 1 som «Låst». Klarer vi det, så har vi skaffet oss elektrisk minne. Har vi først klart å skaffe oss et binært minne, er det relativt enkelt å utvide det til å være så stort en bare vil. Det skal vi se på senere i kapittelet «Kun i minnet finner hjertet fred». Nå skal vi konsentrere oss om å lage et binært minne basert på elektriske porter.

La oss prøve oss med den enkleste logiske porten vi har – AND-porten. Hvordan kan vi bruke den til å ta vare på sin nåværende tilstand eller forandre den? Nåværende tilstand finnes jo på utgangen, så ved å koble den til en av inngangene kunne det jo være mulig å få til noe. Tilbakekobling finnes jo overalt i naturen, så hvorfor ikke prøve det her også? La oss se:



Figur 7.4 Forsøk på å låse en port som er «Åpen»

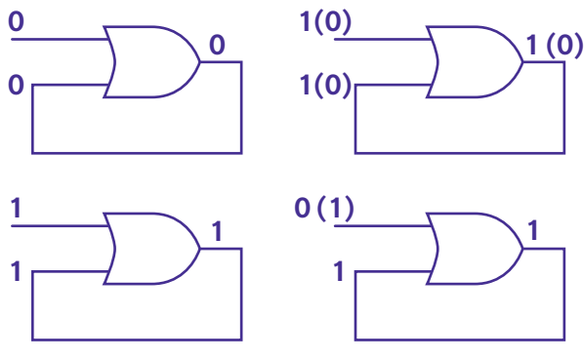
Ved å forandre inngangsverdien fra 0 til 1 skjer det ingenting, som vist i figur 7.4. Tallet i parentes i den høyre figuren i figur 7.4 viser forrige tilstand. Liten suksess med andre ord. Hva med å åpne noe som er «Låst»?



Figur 7.5 Forsøk på å åpne en port som er «Låst»

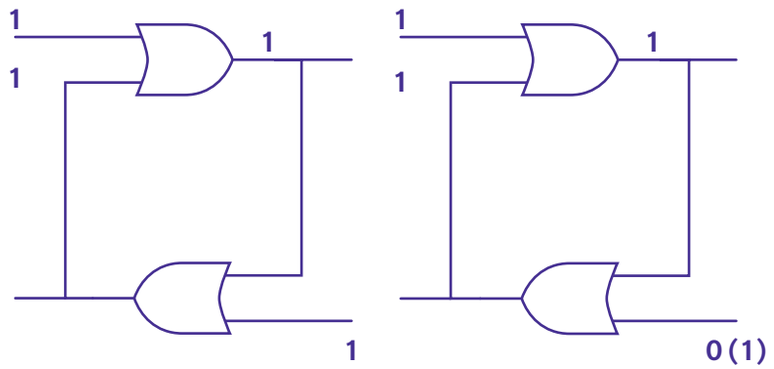
Dette gikk mye bedre. Vi klarte altså å låse opp en «Låst» port ved å endre inngangsverdien fra 1 til 0. Problemet er bare at en slik lås som bare kan låses opp, er lite anvendelig.

Kanskje det går bedre dersom vi bruker en OR-port? La oss sjekke:



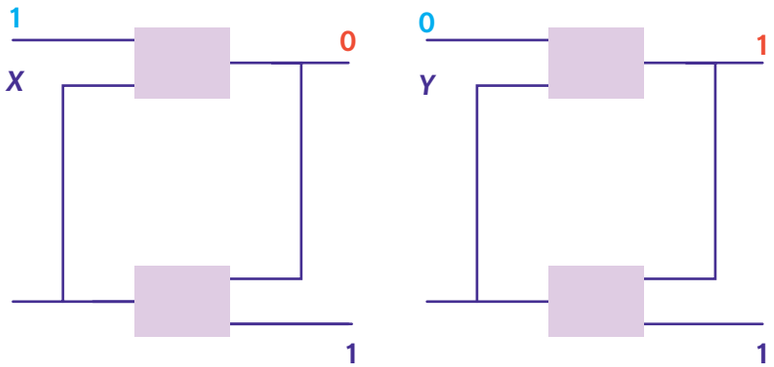
Figur 7.6 Forsøk på å lage lås med OR-port

Ooops! Her får vi det motsatte problemet. Vi klarer å låse, men ikke å åpne. Det trengs nok flere inngangsverdier for å få laget en lås. Det som må til, er at den tilbakekoblede verdien må styres via en port slik at den kan endres dersom en ønsker å låse opp igjen. Hva med å sette inn en OR-port til, slik som i figur 7.7?



Figur 7.7 Forsøk på å lage lås med to OR-porter

Ikke synderlig suksess nå heller. Endringen i inngangsverdi i porten på låse-siden endrer ikke verdien inn på den andre porten. Istedenfor å prøve og feile får vi heller bite i det sure eplet og tenke litt. Vi vet jo hva vi vil ha. Kanskje det kan hjelpe oss til å finne ut hva det vil kreve av portene?



Figur 7.8 Før- og etter tilstand ved låsing

I figur 7.8 har vi skissert hva som kreves for låsing. Inngangsvariabelen som endres er merket med blått og låsetilstanden med rødt. Da vi enda ikke vet hvilken port-type som kan hjelpe oss å løse problemet, er den markert som en firkant. Fra figuren til venstre i figur 7.8 får en følgende betingelser som må oppfylles:

$$0\ op\ 1 \rightarrow X, X\ op\ 1 \rightarrow 0$$

Op er her en eller annen operasjon (port) som vi ikke kjenner enda. Figuren til høyre krever:

$$1\ op\ 1 \rightarrow Y, Y\ op\ 0 \rightarrow 1$$

Vi starter med å gjette at $X = 1$. Da vil NAND-funksjonen tilfredsstille første ligningssett, og hvis vi velger $Y = 0$ vil NAND-funksjonen også kunne brukes på andre ligningssett. Ved å velge $Y = 0$ ser en også at dersom en endrer inngangsverdien fra 0 tilbake til 1 igjen, vil låsen fremdeles være låst. En kan altså låse en port ved å endre inngangsverdien fra 1 til 0 og så tilbake igjen til 1 etter en kort stund.

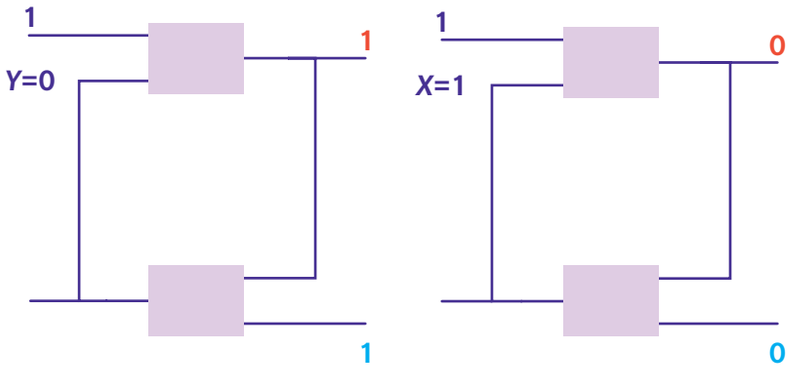
Som en liten repetisjon, er NAND-funksjonen en OG-funksjon med invertert utgang. Sannhetstabellen for NAND blir:

Tabell 7.1 Sannhetstabell for NAND

A	B	$\overline{AB}$
0	0	1
0	1	1
1	0	1
1	1	0

Figur 7.9 Symbol for NAND-funksjonen

Da har vi klart å lage en logisk krets som kan låse, men hva med å låse opp? Da prøver vi å endre på den andre mulige inngangsvariabelen.

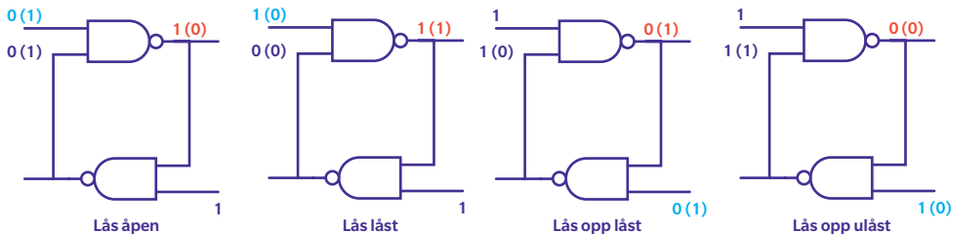


Figur 7.10 Før- og etter tilstand ved åpning

Siden vi er diktert av portvalget vi allerede har gjort ved låsing (NAND-porter), ser vi at følgende konfigurasjon kan brukes til å låse opp. Igjen kan inngangsvariabelen,

merket med blått, settes tilbake til 1 etter at operasjonen er utført, og låsen vil forbli i sin siste tilstand, nemlig «Åpen».

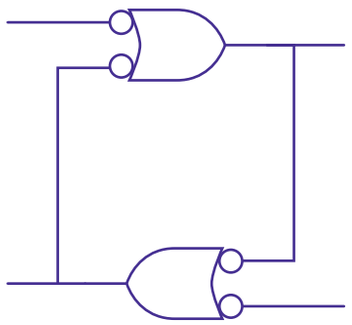
For å sammenfatte det hele så kan vi se på de ulike tilstandsovergangene. Denne gang med NAND-porter inntegnet. Tallene i parentes representerer før-tilstand:



Figur 7.11 Den elektroniske låsen

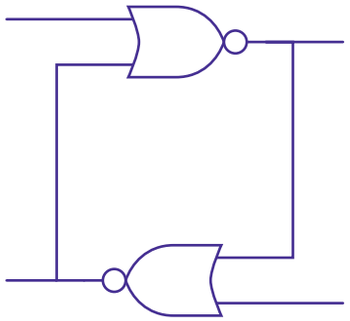
Ved å sammenligne den elektroniske låsen i figur 7.11 med figur 7.1 som viste tilstandsmaskinen for en lås, ser vi at vi har klart å lage det vi er ute etter kun med to NAND-porter og to inngangsvariabler som styrer henholdsvis lukke- og åpne-funksjonen.

Finnes det andre porter som kunne gjort susen? Absolutt. En port hvor inngangene inverteres før det ORes (Negative-OR), gir samme sannhetstabell som NAND-funksjonen og vil derfor fungere like bra.



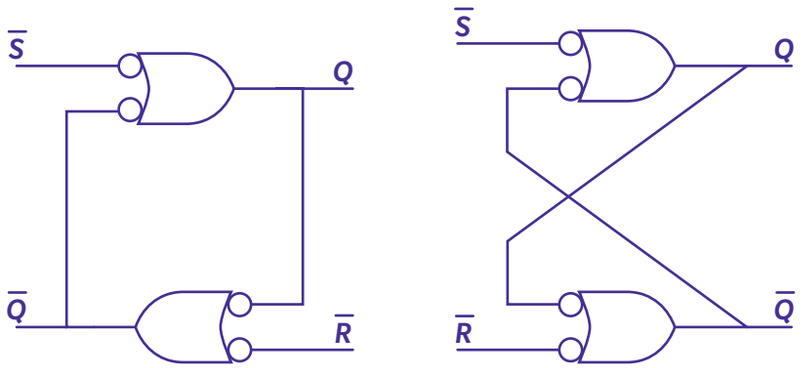
Figur 7.12 Den elektroniske låsen med Negative-OR porter

Dersom en tillater at inngangsverdiene inverteres og bytter rolle, kan en lage en lås med NOR-porter. Det kalles en «aktiv HØY»-lås i motsetning til NAND-låsen som er «aktiv LAV».



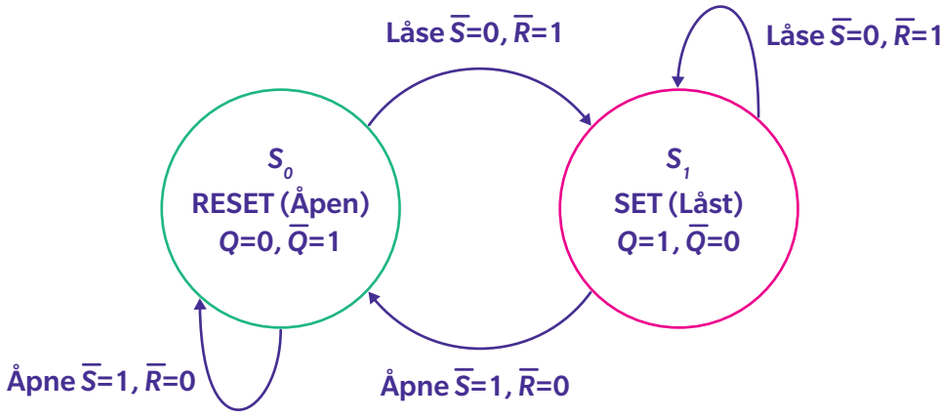
Figur 7.13 Den elektroniske låsen med inverterte innganger og rollebytte

I et forsøk på å forstå den elektroniske låsens oppbygging, har vi beveget oss litt utenfor allfarvei. La oss med nyvunnet kunnskap bli litt mer konvensjonelle. Først vrir vi figuren slik at innganger kommer på ene siden og utganger på andre siden, og så merker vi dem med symbolene allmennheten har blitt enig om.



Figur 7.14 Den elektroniske låsen før og etter vridning

At den elektriske låsen er låst, betegnes med «SET», og at den er åpen med «RESET». Inngangsvariablene kalles $\overline{S}$ og $\overline{R}$ og oppgis som inverterte, siden NAND-låsen er en «aktiv LAV»-lås (NAND Active-Low Latch). Selve låsverdien er gitt av Q og vi har også tilgang til dens inverterte verdi $\overline{Q}$. Denne låstypen kalles en S-R-lås. Tilstandsmaskinen for den elektriske låsen er vist i figur 7.15.



Figur 7.15 Tilstandsmaskin for en S-R-lås

Det er en ting vi ikke har problematisert. Hva om en skulle finne på å sette $\overline{S}=0$ og $\overline{R}=0$ samtidig? Det vil gjøre låsen ustabil, og en vil ikke lenger vite hva Q er. Å gjøre det er en ulovlig operasjon som må unngås.

En kan også analysere oppførselen til en S-R-lås ved hjelp av en tabell.

Tabell 7.2 Oppførsel til en S-R-lås

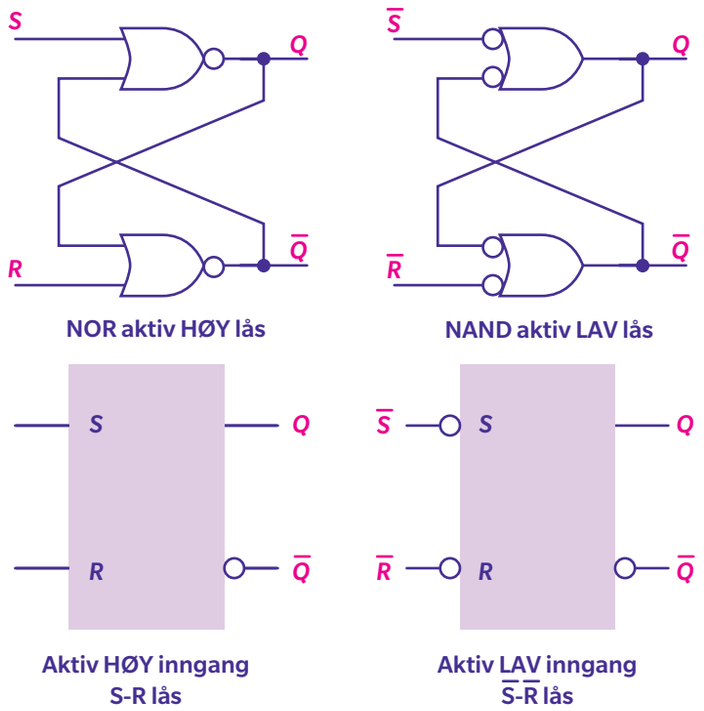
$\overline{S}$	$\overline{R}$	Q_{etter}	**Merknad**
1	1	$Q_{før}$	Ingen endring
0	1	1	SET (uansett)
1	0	0	RESET (uansett)
0	0	(1)	Ulovlig operasjon – setter $Q_{etter}=\overline{Q}_{etter}=1$

For å oppsummere så er en S-R-lås en digital krets med to stabile tilstander:

$$SET(Q=1,\overline{Q}=0)$$

$$RESET(Q=0,\overline{Q}=1)$$

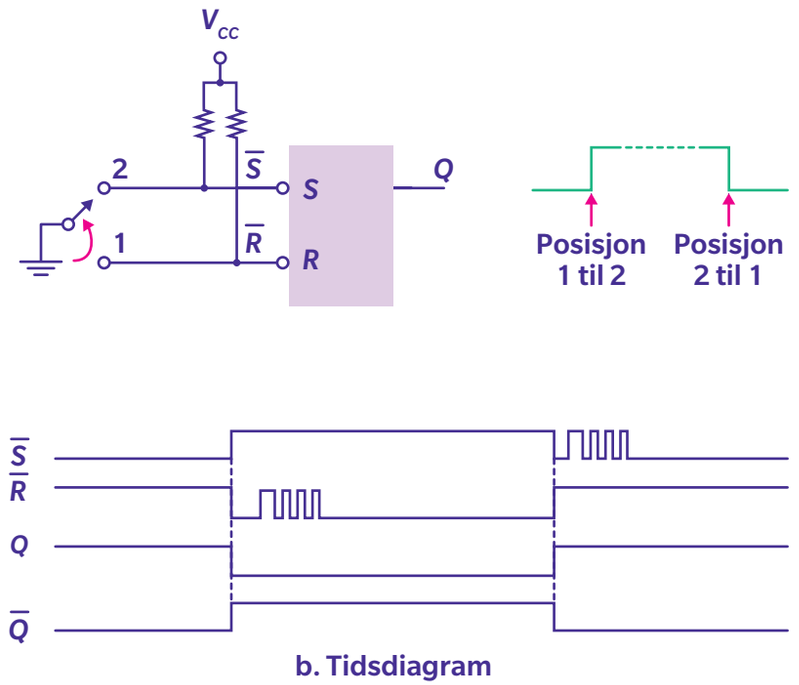
S-R-låsen danner basisformen for et minne. Så lenge en har $\overline{S}=1$ og $\overline{R}=1$, vil kretsen «huske» det som skjedde sist enten det var en SET eller RESET operasjon. En SET (låse) operasjon utføres ved å sende et kortvarig LAV signal til $\overline{S}$. En RESET (åpne) operasjon utføres ved å sende et kortvarig LAV signal til $\overline{R}$. Som vi har vært inne på tidligere, kan S-R-låser implementeres med «aktiv HØY» eller «aktiv LAV». I figur 7.16 er begge typer vist med sine tilhørende logiske symboler.



Figur 7.16 «Aktiv HØY» og «aktiv LAV» S-R-lås

Som et eksempel på bruk av en S-R-lås, kan vi se på hvordan bryter prell som skyldes mekanisk ustabilitet ved lukking av bryteren kan fjernes (Digital Fundamentals [2] side 391). Figur 7.17 viser hvordan bryter prell gjør at det tar tid før bryteren stabiliserer seg.

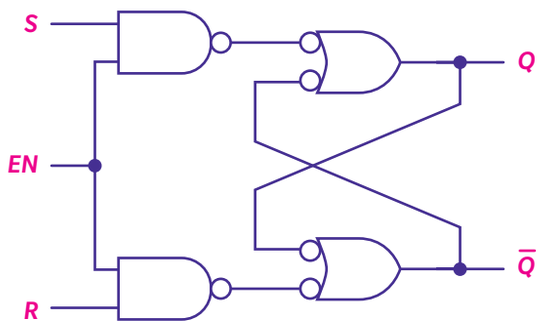
Figur 7.17 Bryter prell



Figur 7.18 Krets som forhindrer bryter prell

Som vi ser i figur 7.18, så påvirker ikke støy fra bryteren når den går fra 1 til 2, på $\overline{S}$ eller $\overline{R}$ utgangsverdien Q for bryteren.

Inngangsvariablene til en S-R-lås kan portstyres ved å lage en «enable» inngang (EN) som vist i figur 7.19.



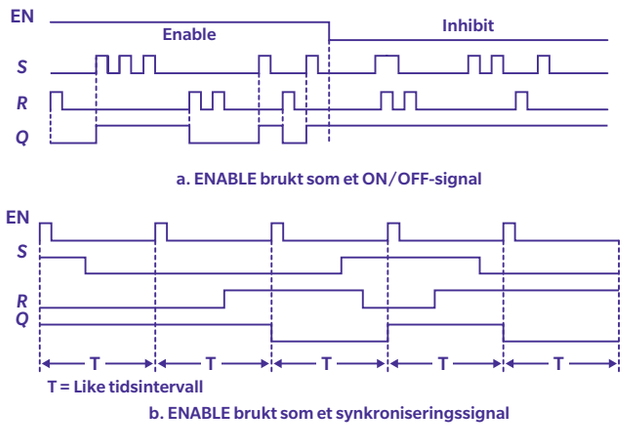
Figur 7.19 S-R-lås med «enable» inngang

Oppførselen til en portstyrt S-R-lås er gitt i tabell 7.3.

Tabell 7.3 Oppførsel til en portstyrt S-R-lås

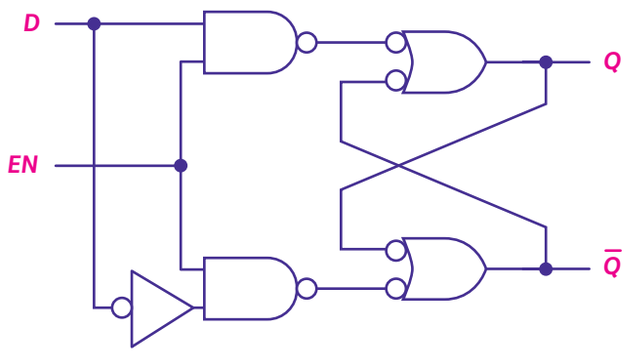
EN	*S*	*R*	Q_{etter}	**Merknad**
1	0	0	$Q_{før}$	Ingen endring
1	1	0	1	SET (uansett)
1	0	1	0	RESET (uansett)
1	1	1	(1)	Ulovlig operasjon – setter $Q_{etter} = \overline{Q}_{etter} = 1$
0	x	x	$Q_{før}$	Ingen endring

Det er to måter en kan anvende «enable» på. Enten som et ON/OFF signal eller som en synkronisator. De to bruksområdene kan ses i figur 7.20.



Figur 7.20 Bruksområder for en portstyrt S-R-lås

Man kan også lage en lås ved å koble sammen de to inngangene S og R slik at det ikke er mulig å fremprovosere ulovlige tilstander. Det gjøres med hjelp av en inverter. En slik lås kalles en D-lås. Figur 7.21 viser en portstyrt D-lås.



Figur 7.21 Portstyrt D-lås

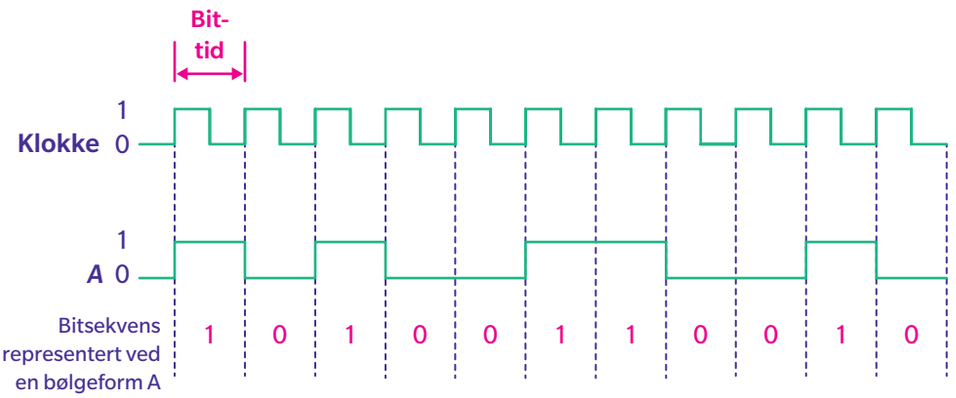
Når «enable» er aktiv, vil utgangen til den portstyrte D-låsen følge data på inngangen, og når «enable» er inaktiv, vil verdien på utgangen bevares slik den ble satt siste gang «enable» var aktiv. På grunn av dette kalles en D-lås ofte en transparent lås.

Tabell 7.4 Oppførsel til en D-lås

Funksjonstabell for en transparent lås					
EN	*D*	Q_{etter}	$\overline{Q}_{etter}$	**Funksjon**	**Kommentar**
0	X	$Q_{før}$	$\overline{Q}_{før}$	Ingen endring	Lagre
1	0	0	1	Reset	Transparent
1	1	1	0	Set	

Noen prinsipper skal en ha. Jeg nekter blant annet å gå i «flip-flop». Du vet disse strandsandalene i plast som hindrer enhver fremgang og får deg til å vippe over i møte med den minste utfordring. Da er det bedre å bruke tiden sin på digitale vipper. En digital vippe kalles «flip-flop» og er en portstyrt lås med en klokkeinngang. Det finnes en rekke ulike typer, så som S-R-vippe, D-vippe og J-K-vippe.

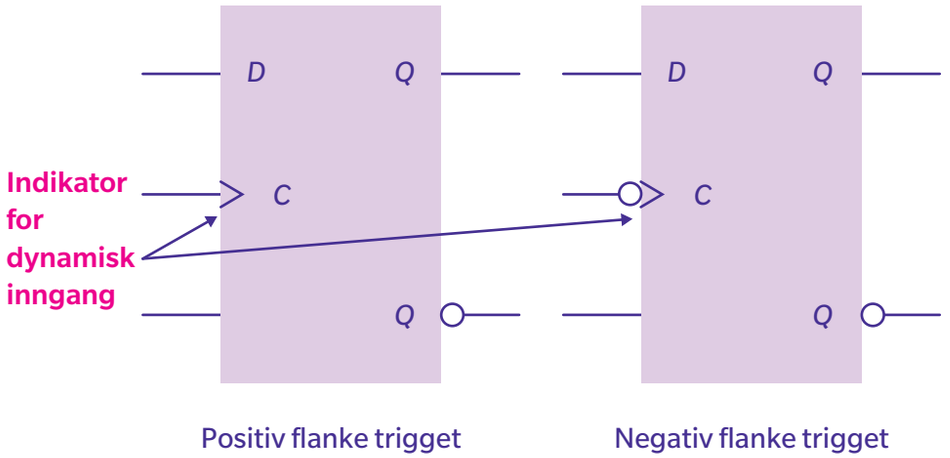
En klokke i digitalteknikk er en sekvens med og som repeteres periodisk. I figur 7.22 er det vist et tidsdiagram for et typisk klokkesignal og en vilkårlig bitsekvens A under.



Figur 7.22 Tidsdiagram for en klokke

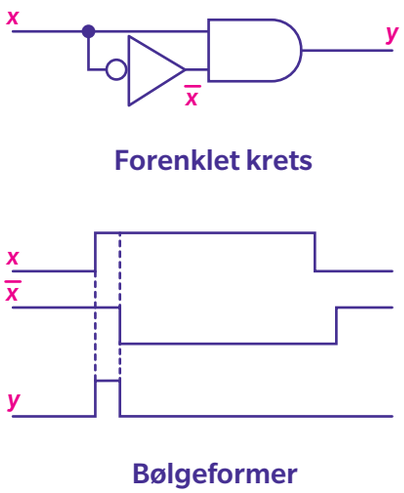
Utgangen til en vippe kan endres når det detekteres en flanke på klokkeinngangen. En positiv flanke har en ved overgang fra logisk 0 til 1 og en har en negativ flanke ved overgang fra logisk 1 til 0. Klokkeinngangen er merket med C i symbolet for en vippe

(figur 7.23) og har en ring som indikerer invertert dersom vippen reagerer på en negativ flanke.



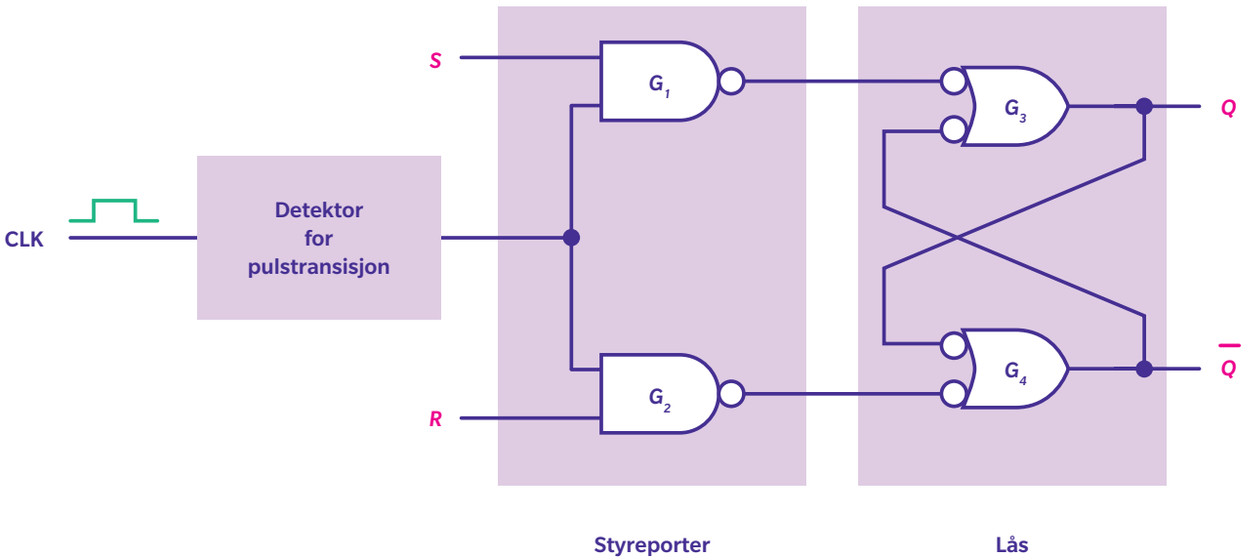
Figur 7.23 Vipper og deres klokkeinnganger

Hvordan oppdager en så en flanke? Vi kan utnytte det som vi tidligere anså som et problem, nemlig forsinkelse i logiske kretser. Kretsen i figur 7.24 kan brukes til å detektere positive flanker, og tilsvarende krets, men litt annerledes, kan detektere negative flanker.



Figur 7.24 Krets for deteksjon av positive flanker

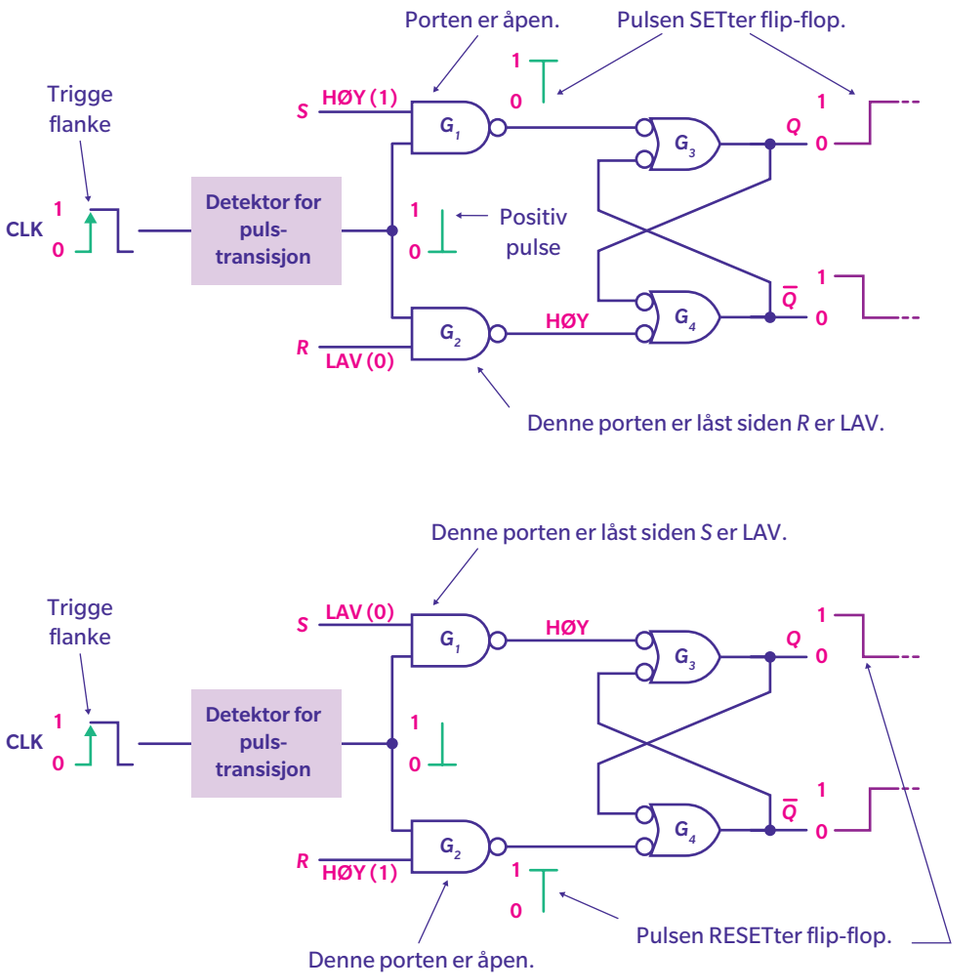
I figur 7.25 er en S-R-vippe vist sammen med sin flankedetektor. Den reagerer på en positiv flanke.



Et forenklet logisk diagram for en positiv flanke trigget S-R flip-flop

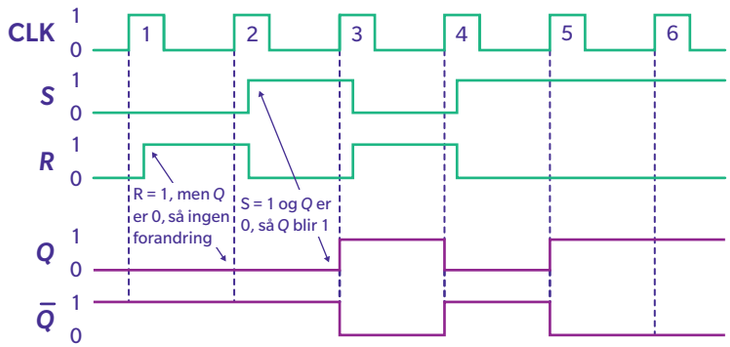
Figur 7.25 S-R-vippe med positiv flankedetektor

La oss analysere virkemåten til en S-R-vippe. Vi setter $S = 1$ for å låse vippen («SET» tilstand). Når den første positive flanken kommer, blir G_1 «enabled», og pulsen som leveres til G_3 gjør at «flip-flop» låses (kommer i «SET» tilstand med $Q = 1$). Dersom en ønsker å låse opp, utføre en «RESET», settes $R = 1$. Figur 7.26 illustrerer dette.



Figur 7.26 «SET» og «RESET» av en S-R-vippe

I figur 7.27 er oppførselen til en S-R-vippe gitt for en sekvens av S og R inngangsverdier. Vær oppmerksom på at det tar tid fra en endring i S eller R medfører en eventuell endring i Q og $\overline{Q}$.



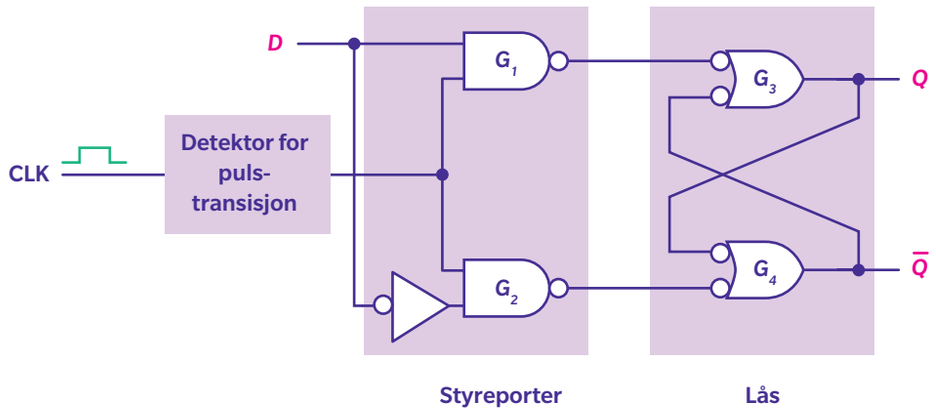
Figur 7.27 Oppførsel til en S-R-vippe

Oppførselen til en S-R-vippe er oppsummert i tabell 7.5. X betyr «don't care», og pil oppover betyr positiv flanke.

Tabell 7.5 Oppførsel til en S-R-vippevippe

S	R	CLK	Q_{etter}	Merknad
0	0	X	$Q_{før}$	Ingen endring
1	0	↑	1	SET (uansett)
0	1	↑	0	RESET (uansett)
1	1	↑	(?)	Ulovlig operasjon

På samme måte som en kan lage en vippe av en S-R-lås, kan det lages en vippe av en D-lås. Fordelen med en slik vippe er at den ikke har noen ulovlige tilstander.



Figur 7.28 D-vippe med positiv flanke detektor

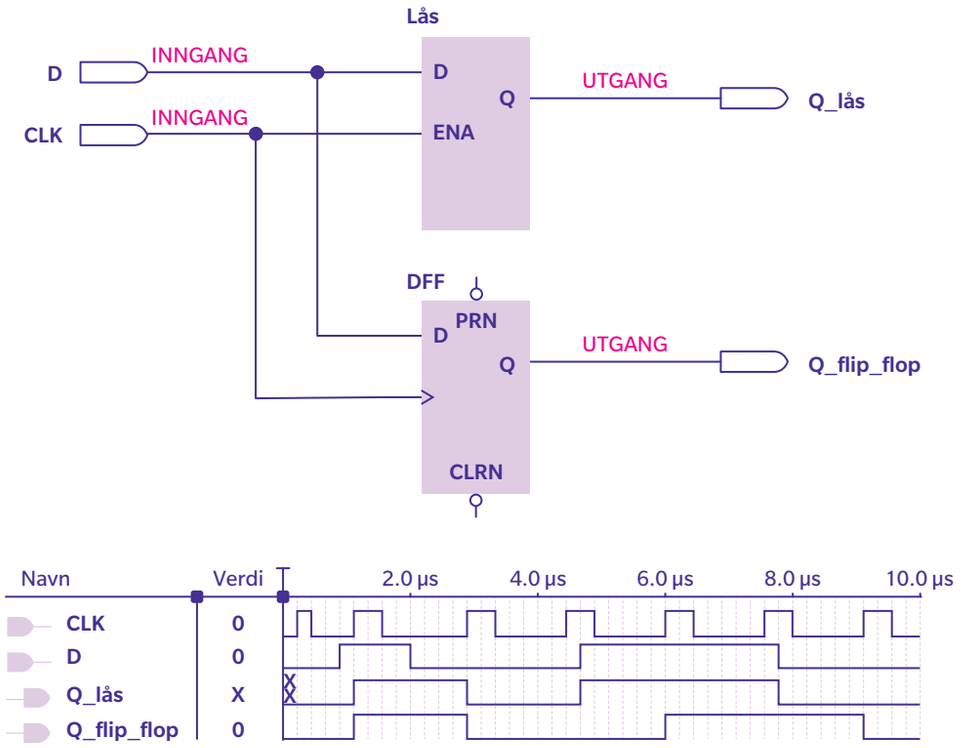
Virkemåten til en D-vippe er lik både for positiv og negativ flanke, som vist i tabell 7.6.

Tabell 7.6 Oppførsel til en D-vippe

Innganger		Utganger		Kommentarer
D	CLK	Q	$\overline{Q}$	
1	↑	1	0	SET
0	↑	0	1	RESET
Positiv flanke trigget				

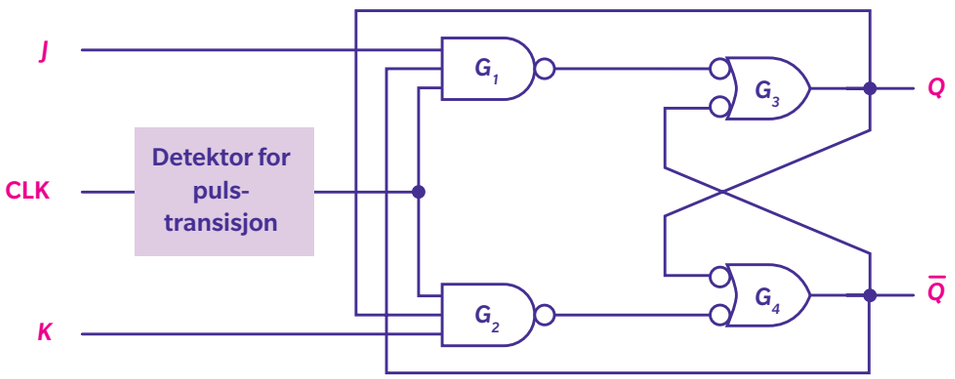
Innganger		Utganger		Kommentarer
D	CLK	Q	$\overline{Q}$	
1	↓	1	0	SET
0	↓	0	1	RESET
Negativ flanke trigget				

La oss for moro skyld se på forskjellen i oppførsel for en D-lås og en D-vippe (figur 7.29). Vi ser at forskjellen ligger i at vippen kun kan endres på den ene flanken, mens D-låsen endrer verdi på utgangen så lenge klokken CLK er høy.



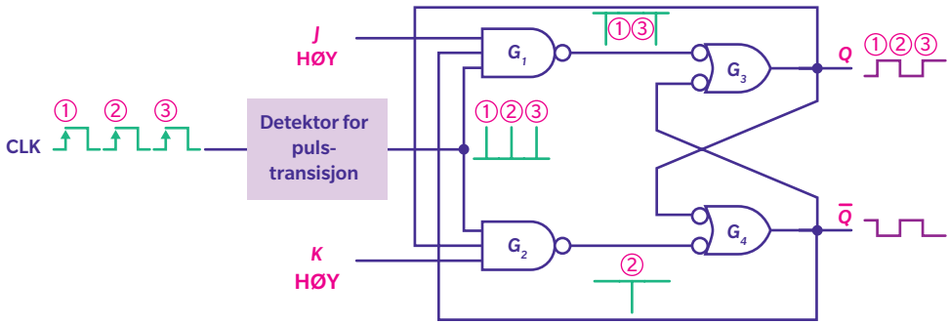
Figur 7.29 D-vippe versus D-lås

Den siste vippetypen vi skal se på, er basert på en S-R-vippe, men med en vesentlig forskjell. Utgangene Q og $\overline{Q}$ er tilbakekoblet til henholdsvis G_2 og G_1. Det forhindrer de ulovlige tilstandene og gjør det mulig å «toggle» dvs. bytte om på verdiene av Q og $\overline{Q}$. Denne vippen kalles en J-K-vippe og er vist i figur 7.30.



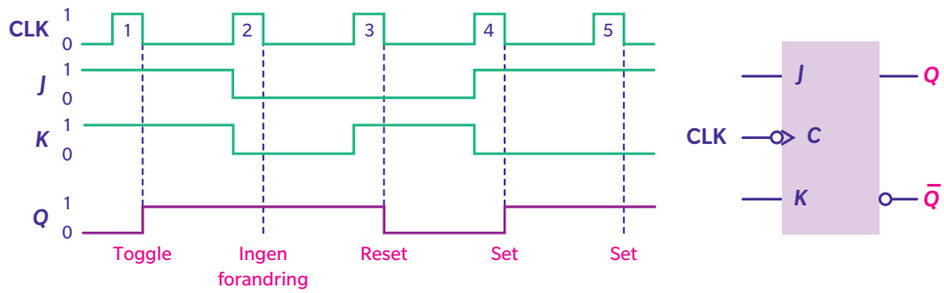
Figur 7.30 J-K-vippe

La oss prøve oss på en analyse av den såkalte «toggle»-tilstanden hvor $J = K = 1$ (figur 7.31). Vi kan velge å starte med $Q = 0$. Da er , og gate G_1 vil slippe signal igjennom ved første flanke, og dermed blir $Q = 1$. Det medfører at gate G_2 vil slippe signal igjennom ved andre flanke og gi $Q = 0$. På denne måten vil J-K-vippen «toggle» Q verdien ettersom flere flanker ankommer.



Figur 7.31 «Toggling» av J-K-vippe

I figur 7.32 er oppførselen til en J-K-vippe som reagerer på negativ flanke vist, og i tabell 7.7 er virkemåten for en J-K-vippe angitt.

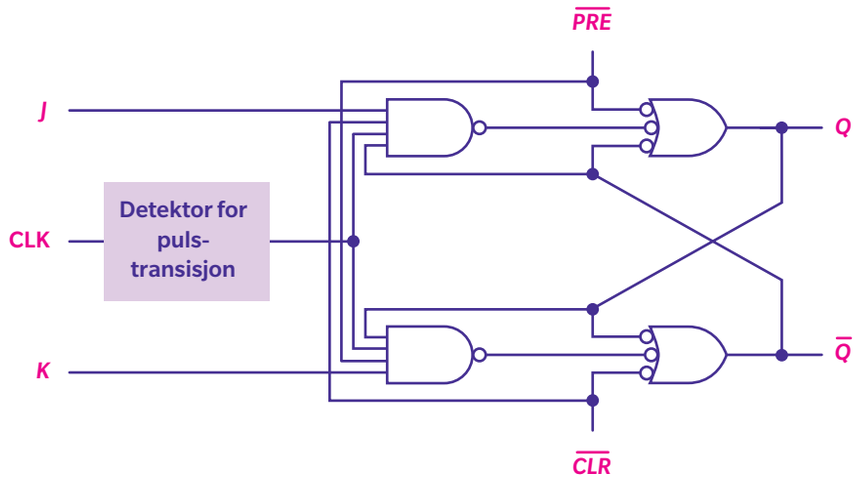


Figur 7.32 Oppførsel til en J-K-vippe

Tabell 7.7 Oppførsel til en J-K-vippe

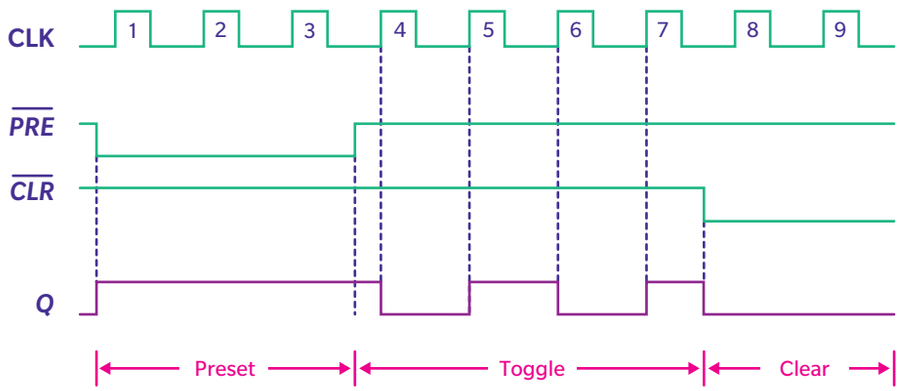
Innganger			Utganger		
J	K	CLK	Q_{etter}	$\overline{Q}_{etter}$	**Kommentarer**
0	0	↑	$Q_{før}$	$\overline{Q}_{før}$	Ingen endring
0	1	↑	0	1	RESET
1	0	↑	1	0	SET
1	1	↑	$\overline{Q}_{før}$	$Q_{før}$	«Toggle»

En vippe kan ha både synkrone og asynkrone innganger. De synkrone inngangene så som D, J og K har kun påvirkning på utgangen av en aktiv klokkeflanke. Asynkrone innganger derimot endrer utgangen med en gang. $\overline{PRE}$ setter utgangen «SET» asynkront, og $\overline{CLR}$ foretar «RESET» asynkront. De asynkrone inngangene har høyere prioritet enn de synkrone. Figur 7.33 viser en J-K-vippe med asynkrone innganger $\overline{PRE}$ og $\overline{CLR}$ i tillegg til de synkrone J og K.



Figur 7.33 J-K-vippe med asynkrone innganger

Figur 7.34 viser oppførselen til en J-K-vippe i «toggle» modus som utsettes for $\overline{PRE}$ og deretter $\overline{CLR}$.



Figur 7.34 J-K-vippe i «toggle» tilstand som utsettes for «PRESET» og «CLEAR»

For å summere det hele opp, er J-K-vippens oppførsel med asynkrone innganger gitt i tabell 7.8.

Tabell 7.8 Oppførsel til en asynkron J-K-vippe

Funksjonstabell for en negativ flanke trigget JK flip-flop med «preset» og nullstill								
Funksjoner								
	$\overline{PRE}$	$\overline{CLR}$	**CLK**	J	K	Q_{etter}	$\overline{Q}_{etter}$	**Funksjon**
Synkrone funksjoner	1	1	↓	0	0	$Q_{før}$	$\overline{Q}_{før}$	Ingen endring
	1	1	↓	0	1	0	1	Reset
	1	1	↓	1	0	1	0	Set
	1	1	↓	1	1	$\overline{Q}_{før}$	$Q_{før}$	Toggle
Asynkrone funksjoner	0	1	X	X	X	1	0	Preset
	1	0	X	X	X	0	1	Nullstill
	0	0	X	X	X	1	1	Ulovlig
	1	1	0	X	X	$Q_{før}$	$\overline{Q}_{før}$	Forhindret
	1	1	1	X	X	$Q_{før}$	$\overline{Q}_{før}$	Forhindret
	1	1	↑	X	X	$Q_{før}$	$\overline{Q}_{før}$	Forhindret

X = «Don't care»

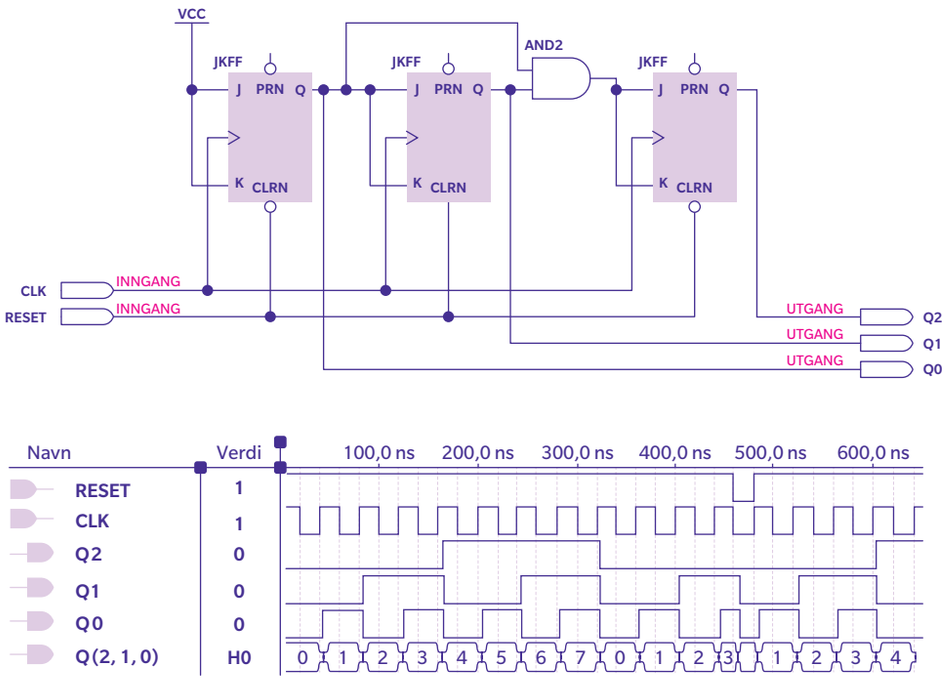
$Q_{før}$ = Nåværende tilstand av Q

Q_{etter} = Neste tilstand av Q

↓ = HØY-til-LAV

↑ = LAV-til-HØY

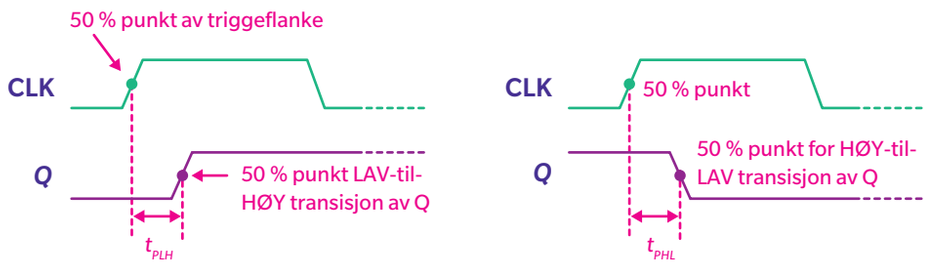
For å ha kontroll på en sekvensiell krets er det vanlig å ha en såkalt «master reset». Det vanlige er å koble et «RESET» signal til $\overline{CLR}$ inngangene på alle vippene. Et eksempel er vist i figur 7.35.



Figur 7.35 Sekvensiell krets med «master reset»

Ved finmysing på tidsdiagrammet til denne kretsen ser det ut til å være en som kan telle til syv, og «RESET» sørger for at tellingen starter på nytt igjen.

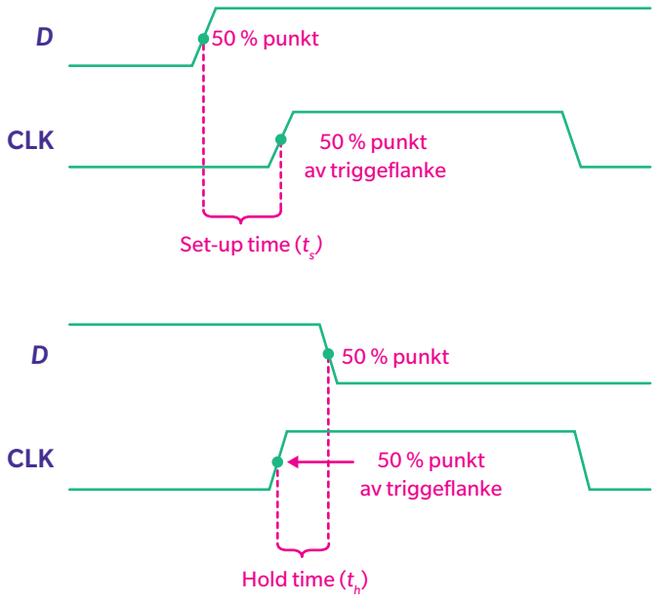
Vi skal nå se på noen generelle egenskaper for vipper. Det viktigste en må tenke på, er at ting tar tid. Som vi har sett tidligere, er det tidsforsinkelse, såkalt forplantningsforsinkelse («propagation time»), i digitale kretser. Forplantningsforsinkelsen for vipper er den tid det tar fra en aktiv klokkeflanke, eventuelt en flanke i et asynkront signal, til utgangen endrer tilstand. I figur 7.36 er det gitt to definisjoner i så henseende.



t_{PLH} = tidsforsinkelsen gjennom kretsen fra klokken trigger til Q går fra LAV til HØY
t_{PHL} = tidsforsinkelsen gjennom kretsen fra klokken trigger til Q går fra HØY til LAV

Figur 7.36 Tidsforsinkelse

De synkrone inngangene må være stabile før aktiv klokkeflanke. «Set-up time» (t_{su}) er tiden de synkrone inngangene til en vippe skal være stabile før aktiv klokkeflanke. «Hold time» (t_h) er tiden de synkrone inngangene til en vippe må være stabile etter aktiv klokkeflanke. I figur 7.37 er dette skissert.



Figur 7.37 «Set-up time» og «hold time»

Pulsbredde (t_w) er minimum lengde i tid som kreves for en puls på klokke CLK, $\overline{PRE}$ eller $\overline{CLR}$ inngangene. Verdien er målt fra midtpunktet på den første flanken til midtpunktet på den avsluttende flanken.

En vippe kan bare fungere opp til en gitt klokkefrekvens. Denne maksimale klokkefrekvensen betegnes f_{max}. Over denne frekvensen vil ikke vippen reagere raskt nok, og det skyldes det enkle faktum at vipper har tilbakekobling, og det tar tid å få signal fra inngang til utgang og tilbake igjen til inngang.

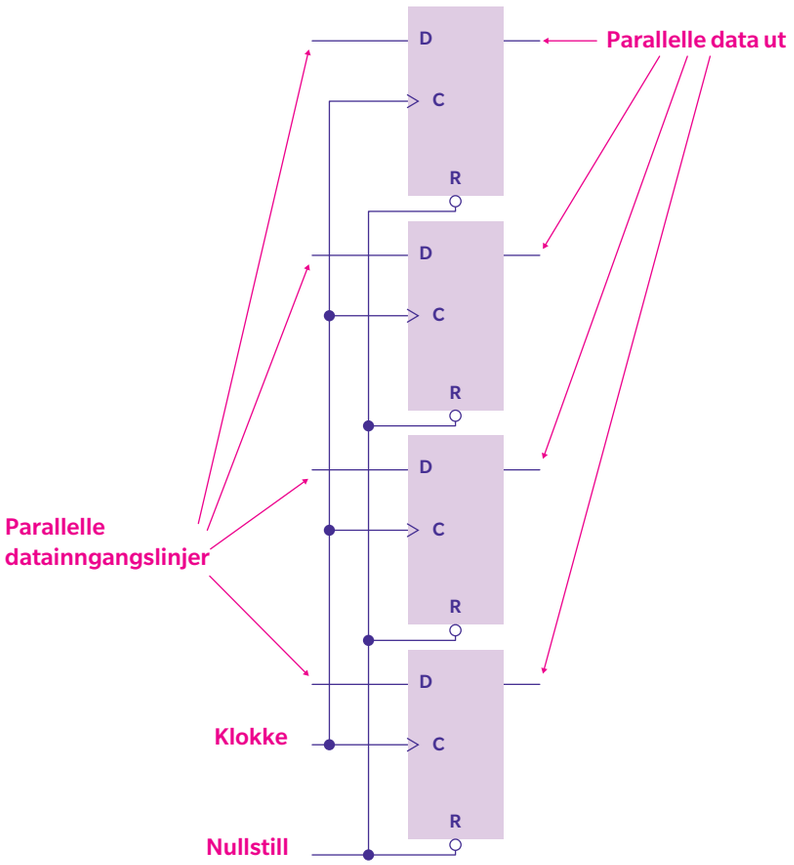
Effektforbruk er en av de viktigste begrensningene ved kretsdesign. Dersom en vippe opererer på 5 V og trekker 5 mA, får vi et effektforbruk på $P = VI = 5\text{ V} \cdot 5\text{ mA} = 25\text{ mW}$. Det ser kanskje ikke så voldsomt ut, men vipper kommer sjelden alene, så totalforbruket kan bli stort med dertil hørende varmeutvikling. Effektforbruket er avhengig av kretsens aktivitet, og da særlig klokkefrekvensen.

I tabell 7.9 er egenskapene for noen vipper oppført. Når en ser på ytelsene, blir en neppe overrasket over at det er CMOS-teknologi som har vunnet frem i digitalteknikk. Blant annet er strømforbruket for CMOS vesentlig mindre enn for bipolar.

Tabell 7.9 Sammenligning av ytelse for CMOS og bipolare vipper

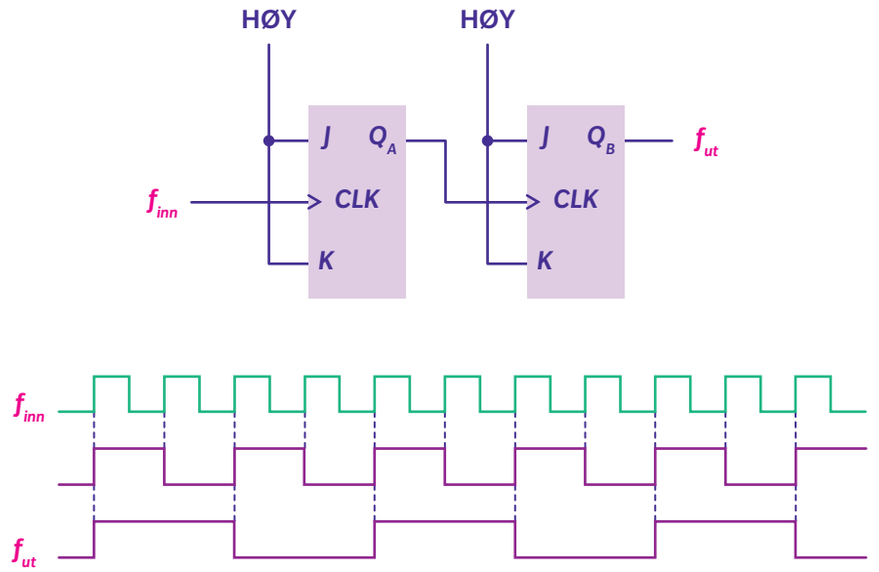
Sammenligning av parametre for fire IC familier av flip-flop av samme type ved 25 grader Celsius				
	CMOS		**BIPOLAR (TTL)**	
PARAMETER	**74HC74A**	**74AHC74**	**74LS74A**	**74F74**
t_{PHL} (CLK til Q)	17 ns	4.6 ns	40 ns	6.8 ns
t_{PLH} (CLK til Q)	17 ns	4.6 ns	25 ns	8.0 ns
t_{PHL} ($\overline{CLR}$ til Q)	18 ns	4.8 ns	40 ns	9.0 ns
t_{PLH} ($\overline{PRE}$ til Q)	18 ns	4.8 ns	25 ns	6.1 ns
t_s (set-up tid)	14 ns	5.0 ns	20 ns	2.0 ns
t_h (hold tid)	3.0 ns	0.5 ns	5 ns	1.0 ns
t_W (CLK HØY)	10 ns	5.0 ns	25 ns	4.0 ns
t_W (CLK LAV)	10 ns	5.0 ns	25 ns	5.0 ns
t_W ($\overline{CLR}$ / $\overline{PRE}$)	10 ns	5.0 ns	25 ns	4.0 ns
f_{max}	35 MHz	170 MHz	25 MHz	100 MHz
Effektforbruk (hvilemodus)	0.012 mW	1.1 mW		
Effektforbruk (50 % av tiden)			44 mW	88 mW

Etter å ha brukt så mye tid på vipper, er det kanskje i seneste laget å spørre hva de kan brukes til. Et område som er veldig viktig, og som fikk oss til å starte med å se på hvordan vi kunne lage låser med minne, er datalagring. I figur 7.38 ser vi fire D-vipper som kan brukes til å lagre 4 bit. Kanskje ikke så mye, men en god begynnelse. D representerer datainnganger for hvert bit, og $\overline{CLR}$ inngangene er koblet sammen for CLEAR operasjon (tømming av minne).



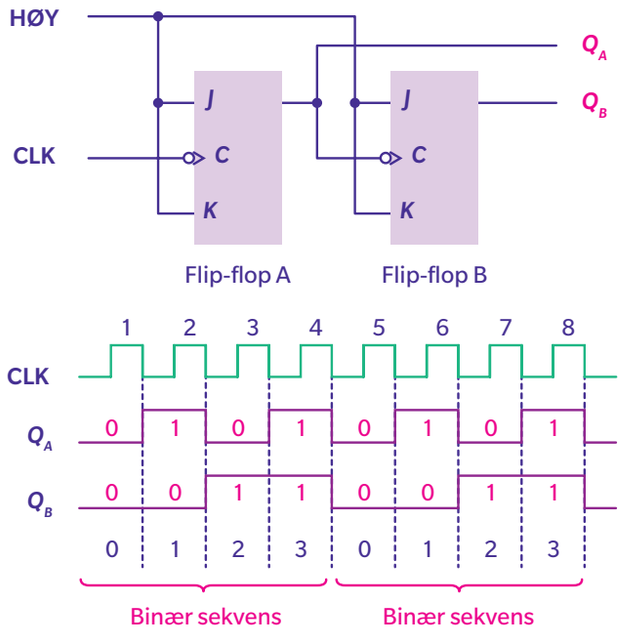
Figur 7.38 D-vipper som lagringsenhet

Vipper kan også brukes til frekvensdeling av klokkefrekvensen. I figur 7.39 har en to J-K-vipper i «toggle» modus som deler klokkefrekvensen til en fjerdedel. Den første vippen mates med klokkefrekvensen, og Q_A vil da «toggle» med halve klokkefrekvensen. Q_A sendes videre til klokkeinngangen på den andre vippen, og vips, så har en en fjerdedel av klokkefrekvensen på utgangen Q_B.



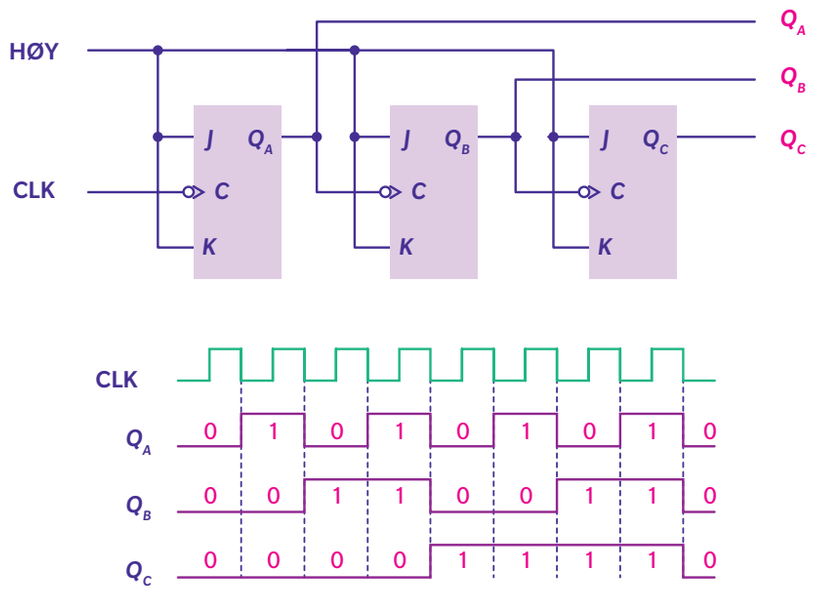
Figur 7.39 J-K-vipper som frekvensdeler

Frekvensdelere kan også brukes til å lage tellere. I figur 7.40 er det vist en teller som teller fra 0 til 3 for så å starte på nytt igjen i en evig runddans. LSB bit hentes fra første vippe (Q_A) og MSB fra andre vippe (Q_B).



Figur 7.40 J-K-vipper som teller

Størrelsen på telleren kan økes med å pøse på med flere vipper. En utfordring er at en da lager større og større problemer med tidsforsinkelse. Det er bare første vippe som har CLK inn på klokkeinngangen, de andre har utgangen fra forrige vippe. Jo flere vipper, jo lengre tid før tellerverdien er riktig og stabil. Hvordan vi kan løse dette problemet, skal vi se på i kapittelet «Evig runddans». Figur 7.41 viser en teller som kan telle fra 0 til 7.



Figur 7.41 Enda en teller

Som vi har sett, er låser og vipper de fundamentale byggeklossene i sekvensiell logikk, og i de neste kapitlene vil vi se nærmere på hvordan de kan nyttes til å lage mer avanserte og anvendelige logiske kretser.

8

Kapittel 8

Gjennomtrekk

«Det tvilsomste klientell, er blitt stamgjester på mitt hotell.»

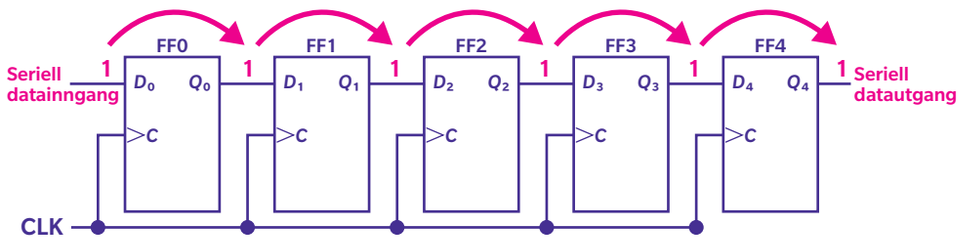
Hotell gjennomtrekk, Anne Grete Preus (1957–)

LÆRINGSUTBYTTE: Skiftregister, seriell-inn/seriell-ut, seriell-inn/parallell-ut, parallell-inn/seriell-ut, parallell-inn/parallell-ut, bidireksjonalt

Forandring fryder! **Det påstås at det kun er spedbarn med full bleie som liker endringer. At det er en sannhet med modifikasjoner, kan observeres når nye koster og organisasjonskonsulenter er i full sving. Ja, det finnes faktisk en rekke kretser som er skapt for endring, som ånder for forandring og som venter med spenning på neste trekk. Deres fellesbetegnelse er skiftregistre. Det første skiftregister så bokstavelig talt dagens lys i noen radiorør i kodeknekker-maskinen Colossus i 1944.**

Kort fortalt er et skiftregister en sekvensiell krets som vil lagre og flytte n bit med data enten serielt eller parallelt i et n-bit register som består av n-vipper. Skiftregistre har mange anvendelser. De kan brukes til å lage forsinkelse, i tellere og være del av asynkrone forbindelser.

Figur 8.1 viser et seriell-inn/seriell-ut skiftregister.



Figur 8.1 Seriell-inn/seriell-ut skiftregister

Som vi ser, er skiftregisteret i figur 8.1 bygd opp av fem D-vipper og kan dermed lagre inntil fem bit. Dersom kretsen mottar sekvensen $1 \rightarrow 0 \rightarrow 1 \rightarrow 1 \rightarrow 0$ med et bit per klokkeslag og høyre bit som sendes først er LSB, vil kretsen fylles opp som angitt i tabell 8.1. Det forutsettes at skiftregisteret er nullstilt før en starter.

Tabell 8.1 Fylling av skiftregisteret

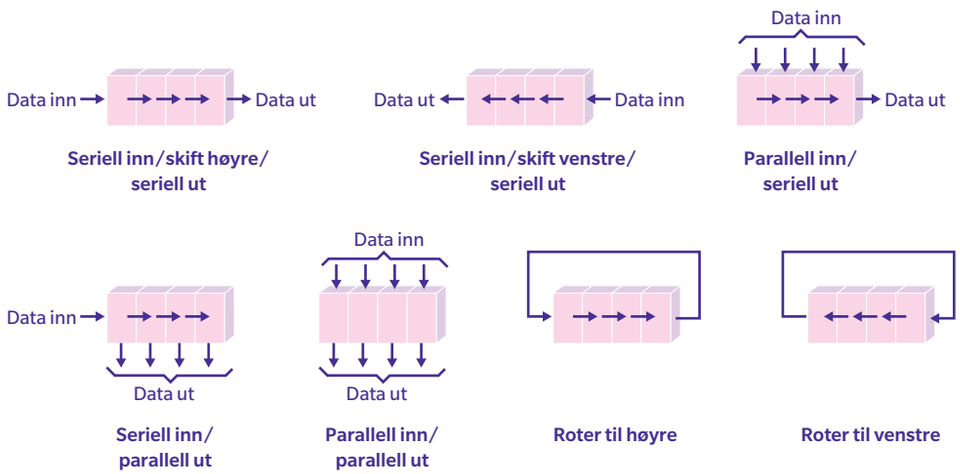
Klokkeslag	Q_1	Q_2	Q_3	Q_4	Q_5
0	0	0	0	0	0
1	0	0	0	0	0
2	1	0	0	0	0
3	1	1	0	0	0
4	0	1	1	0	0
5	1	0	1	1	0

Et skiftregister kan også brukes til å multiplisere med 2 ved et skift mot MSB og dele på 2 ved et skift motsatt vei mot LSB.

Skiftregisteret bruker vippenes evne til å lagre verdier. Når en D-vippe mottar 1 på D, vil Q bli 1 ved neste klokkeflanke, og dermed blir verdien tatt vare på. Når D mottar en 0, vil Q bli 0, eller forbli 0 dersom den allerede er det, ved neste klokkeflanke, og verdien 0 blir lagret.

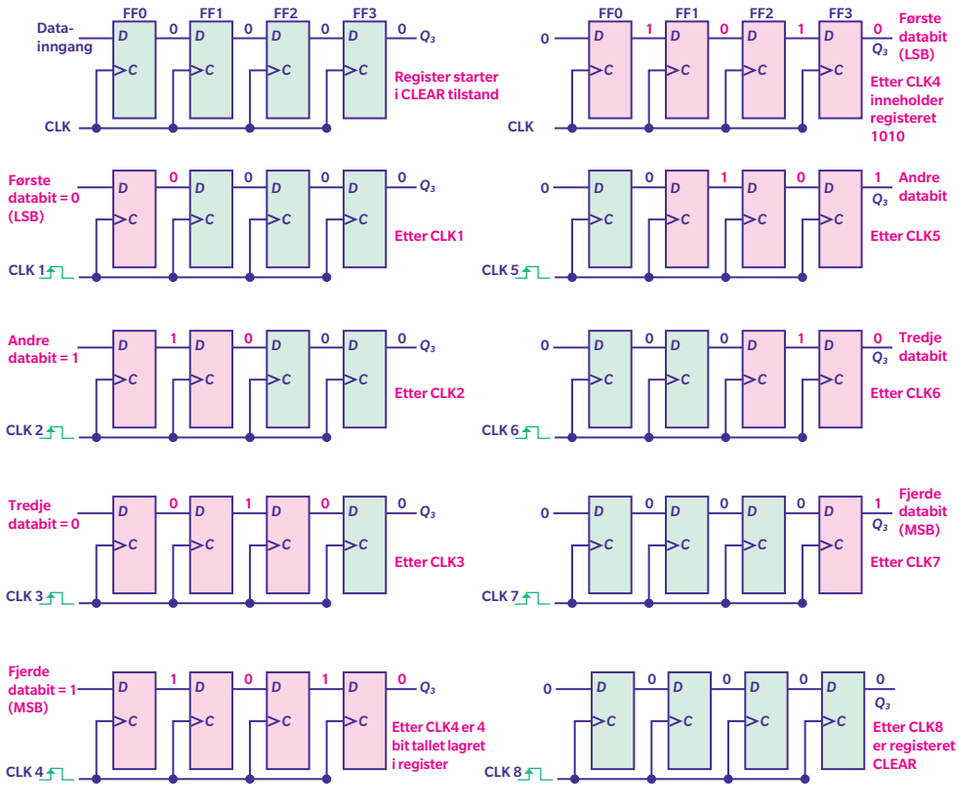
Figur 8.2 Vippe som lagringselement

Ved å bruke vipper som byggeklosser kan mange ulike typer skiftregistre konstrueres. I figur 8.3 er noen basis skiftregisteroperasjoner gitt.



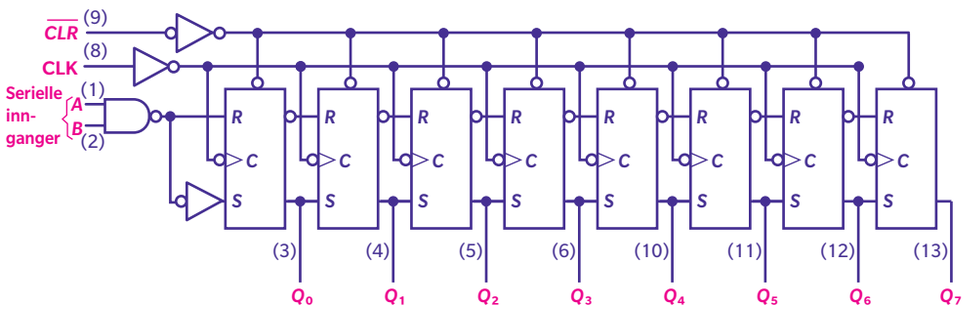
Figur 8.3 Basis skiftregisteroperasjoner

Figur 8.4 viser fylling og tømming av et seriell-inn/seriell-ut skiftregister. På venstre side starter vi med et tomt register. Ved første klokkepuls fylles 0 inn i registerets første D-vippe. Slik fortsetter det til hele registeret er fylt med sekvensen 1010. Til høyre ser vi tømming av det samme registeret.



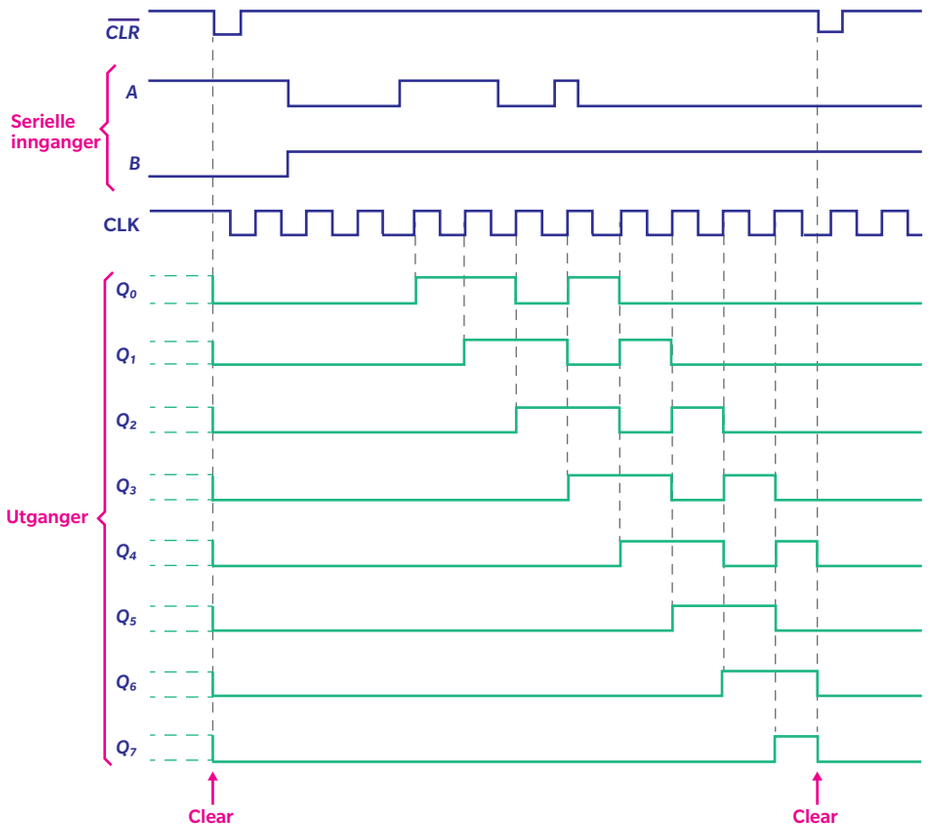
Figur 8.4 Fylling og tømming av et serielt skiftregister

Det neste vi skal se på, er et seriell-inn/parallell-ut skiftregister. Figur 8.5 viser et slikt register med to serielle innganger A og B og åtte parallelle utganger Q_0 til Q_7. En av de serielle inngangene kan brukes som «enable»-inngang eller kobles til V_{cc} som er det samme som logisk HØY.



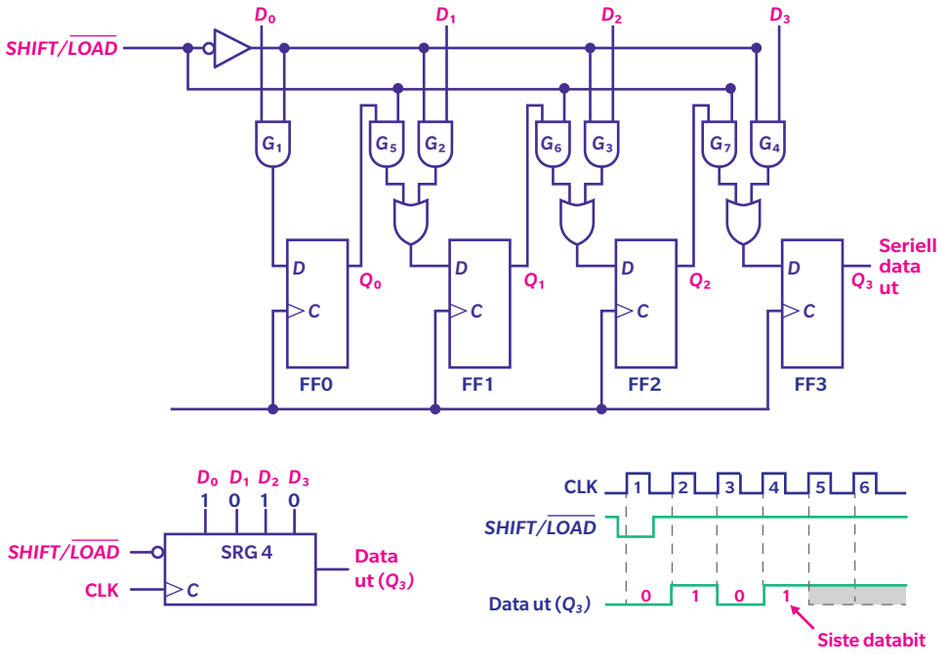
Figur 8.5 Seriell-inn/parallell-ut skiftregister

Dersom vi fyller dette registeret med seriell data på inngang A og bruker B som «enable»-inngang, vil vi få en oppførsel som vist i figur 8.6. I dette tilfellet har vi anvendt S-R-vipper. Vi ser at etter hvert som tiden går etter at B er «enablet», vil det serielle bitmønsteret fra A konverteres til et parallelt bitmønster som kan hentes ut på utgangene Q_0 til Q_7. Et slikt skiftregister er nyttig dersom en ønsker å konvertere fra serielt til parallelt.



Figur 8.6 Virkemåte til et seriell-inn/parallell-ut skiftregister

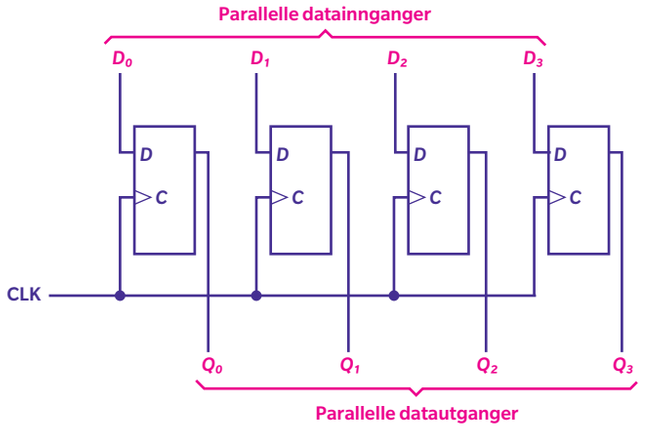
Et parallell-inn/seriell-ut skiftregister kan realiseres som angitt i figur 8.7.



Figur 8.7 Parallell-inn/seriell-ut skiftregister

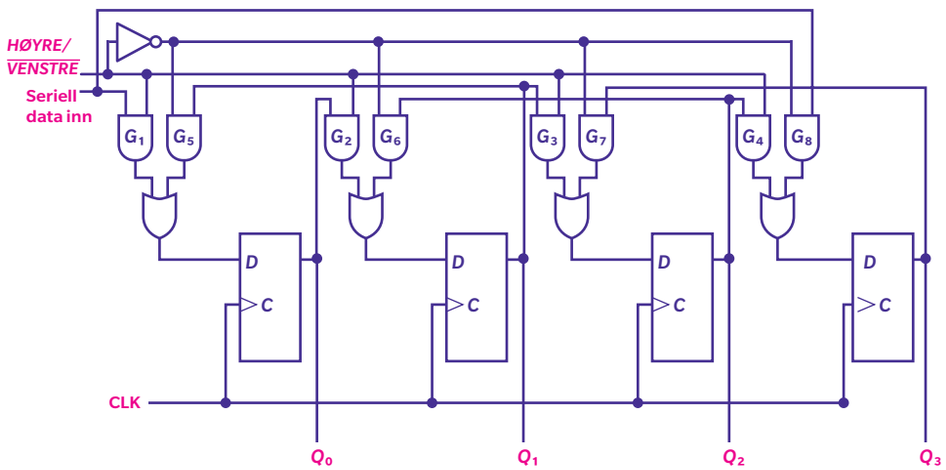
Hvordan virker så denne kretsen? Vel, når $SHIFT / \overline{LOAD}$ settes til 0, fylles Q_0 til Q_3 med data fra D_0 til D_3, da portene G_1, G_2, G_3 og G_4 vil være åpnet av det inverterte $SHIFT / \overline{LOAD}$ signalet. D_0 er det minst signifikante bit (LSB). Ved utlesning settes $SHIFT / \overline{LOAD}$ til 1, og portene G_5, G_6 og G_7 vil åpnes og gjøre seriell utlesning via Q_3 mulig.

Flere byer i Belgia? Absolutt! Hva med et parallell-inn/parallell-ut register? Som vist i figur 8.8, er oppbyggingen rimelig enkel, i hvert fall logisk sett.



Figur 8.8 Parallell-inn/parallell-ut skiftregister

Den siste typen skiftregister vi skal se på, er et bidireksjonalt skiftregister. Det er serielt inn og parallelt ut og laget slik at utgangsdata kan flyttes frem og tilbake mellom Q_0 til Q_n basert på verdien av styresignalet $RIGHT / \overline{LEFT}$. Figur 8.9 viser et fire bit bidireksjonalt skiftregister.



Figur 8.9 Fire bit bidireksjonalt skiftregister

9

Kapittel 9

Evig runddans

«Men etter ti runder øl og en jæger, da hadde
æ mista tellinga, mått start på nytt.»

Øl & jæger, Holmsve

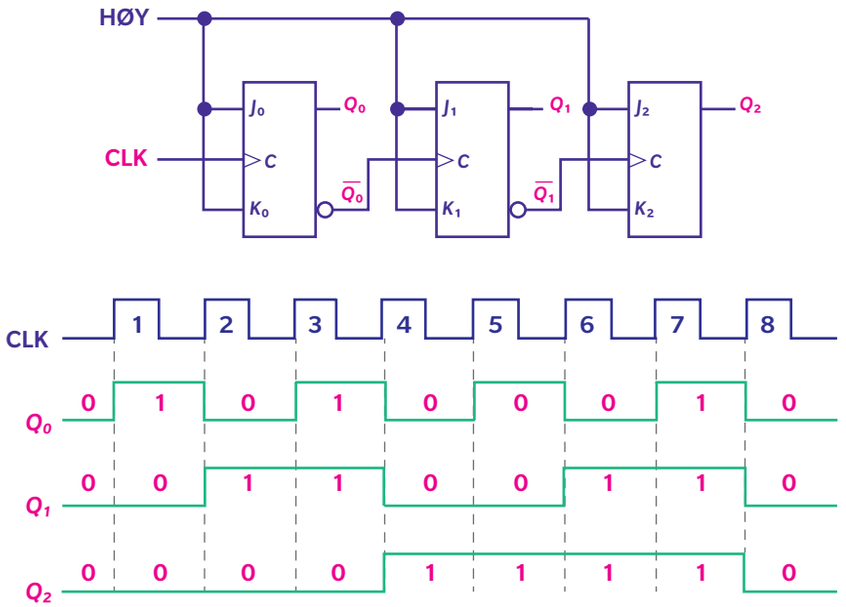
LÆRINGSUTBYTTE: Tellere, asynkrone tellere, modulus til teller, synkrone tellere, opp-ned tellere, generelle tellere, større tellere, frekvensmanipulasjon med tellere

«Hva er det som går og går og aldri kommer til døra?» Denne barnegåten, hvis svar er tiden, ledet en ung mann til Greenwich utenfor London. Bredbent over nullmeridianen for lengdegrader med en fot i hver tidssone prøvde han å få endene til å møtes. Senere i livet lærte han at nullpunkt ofte er politisk bestemt, og at også fordums stormakter som Portugal og Spania tidligere hadde sine nullmeridianer gjennom henholdsvis Lisboa og Madrid. Hvor de binære tellerne «wrapper» er, er derimot kun et spørsmål om tilgjengelig antall bit.

Hva gjør en teller? Det ligger nærmest i navnet – den teller. En teller er en sekvensiell krets hvor utgangen utvikler seg med et forutsigbart mønster. Telleren avanserer en tilstand per klokkepuls. Tellere har en rekke bruksområder, som blant annet telling av hendelser, frekvensdeling og «timing»- og kontrolloperasjoner. I figur 9.1 ser en hvordan en teller med binære tall. Det er et repeterende mønster, og vi så jo allerede i figurene 7.40 og 7.41 i kapittel 7 hvordan vipper kunne brukes til å lage tellere.

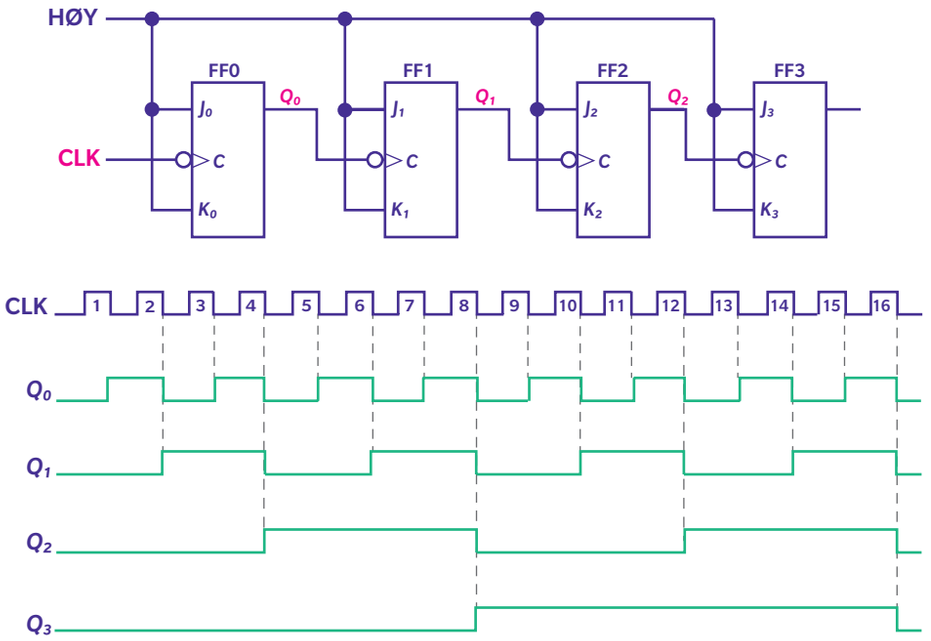
Figur 9.1 Å telle med binære tall

Et eksempel på en binær teller er gitt i figur 9.2. Den består av tre J-K-vipper som er satt i «toggle» modus ved å sette alle J og K til HIGH. Q_0 endrer seg hver gang CLK har en ny positiv flanke, og kan derfor brukes som LSB bit. Klokkesignalet den midtre vippen får, er $\overline{Q_0}$ og gjør at Q_1 «toggler» med halve raten av Q_0. MSB kan vi hente ut fra Q_2, da høyre vippe er matet med klokkesignalet $\overline{Q_1}$ som gjør at Q_2 «toggler» med halve raten av Q_1. På den måten har vi fått en teller som kan utføre det vi så i figur 9.1.



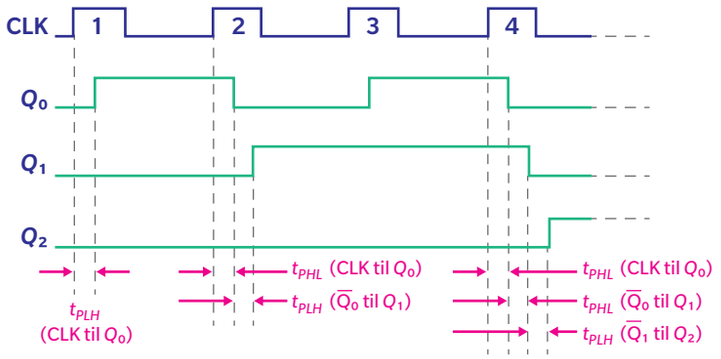
Figur 9.2 Binær teller med tidsdiagram

Dersom en ønsker å telle lenger, er det bare å utvide kretsen. I figur 9.3 har vi en 4 bit teller som teller fra 0 til 15 ved hjelp av fire vipper. Legg merke til at vippene i dette eksempelet er negativ kant «trigget». Det er symbolisert med den lille rundingen på vippenes C inngang.



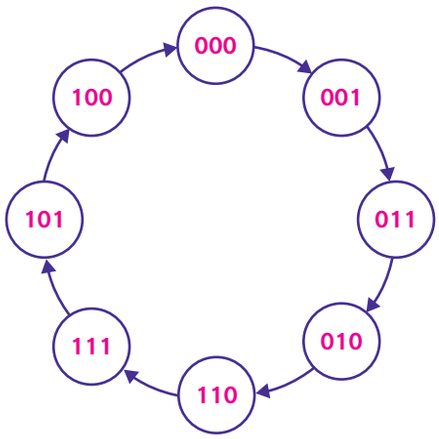
Figur 9.3 4 bit binær teller med tidsdiagram

Felles for de tellerne vi har sett på til nå, er at de er asynkrone. Det betyr at en aktiv klokkekant ikke vil endre verdiene på alle Q-utgangene samtidig. For eksempel ville en ved overgang fra 111 til 000 i en 3 bit teller oppleve at Q_0 endrer verdien først, så Q_1 og til slutt Q_2. Årsaken til det er forsinkelse i kretsene, og de kalles derfor «ripple counters». Figur 9.4 viser et eksempel på slik forsinkelse.



Figur 9.4 Forsinkelse i asynkron teller

For tellere, som for det meste annet, er det to angrepsmetoder – enten analyse eller syntese. Ved analyse tar et utgangspunkt i et kretsdiagram og forsøker så å finne ut kretsens virkemåte. For hver mulig kretstilstand prøver en å finne ut hva som blir kretsens neste tilstand. Når en har samlet all informasjon, dvs. analysert alle tilstander, kan en bygge et tilstandsdiagram.

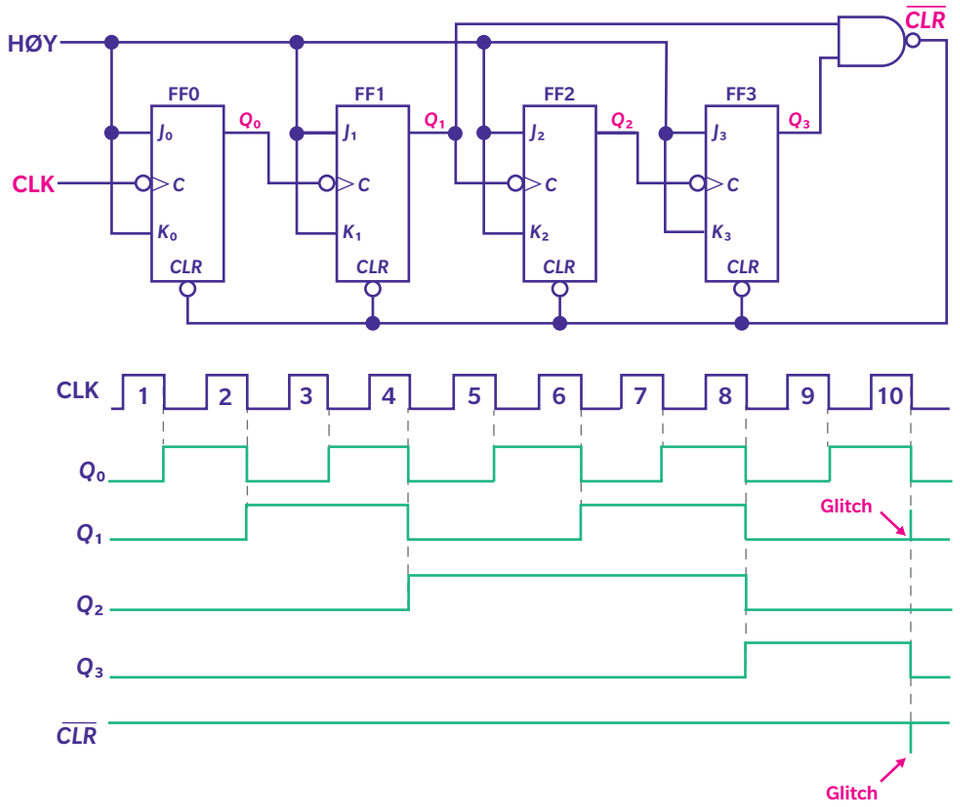


Figur 9.5 Tilstandsdiagram for en 3 bit Gray-kode teller

Ved syntese, for eksempel at en skal lage n tellerkrets, bruker en et oppgitt tilstandsdiagram av typen gitt i figur 9.5, eller informasjon som kan definere et tilstandsdiagram, til å lage kretsen en ønsker. Tilstandsdiagrammet er en avart av Moore tilstandsmaskinen som vi tidligere har vært borti (7.2). De enkelte tilstander er symbolisert med sine utgangsverdier, og «input» verdien er en klokkepuls angitt med pilen mellom to gitte tilstander. Tellesekvensen er den spesifikke serien av utgangstilstander som en teller går igjennom.

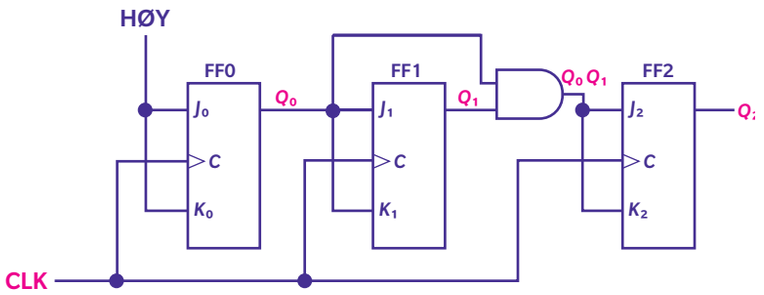
Modulus til en teller er antallet tilstander en sekvens går gjennom før den repeterer seg selv, og kalles Modulo-N eller mod-N. Modulus til telleren i figur 9.5 er 8. En teller kan enten telle opp eller ned. Høyere til lavere indikeres med Ned, mens det motsatte angis med Opp.

Figur 9.6 viser et eksempel på en mod-10 asynkron teller som teller fra 0 til 9. Når Q_3 og Q_1 begge er 1 (kombinasjonen 1010), vil telleren hoppe til 0000 ved å aktivere logisk lav CLR på alle fire vippene. Etter $(1001)_2 = (9)_{10}$ kommer $(0000)_2 = (0)_{10}$. Det er «glitch» på Q_1 ved overgangen fra 9 til 0 som aktiverer CLR.



Figur 9.6 Asynkron 0 til 9 teller med tidsdiagram

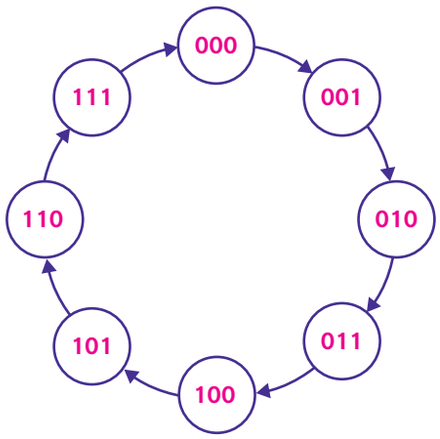
Etter å ha brukt kvalitetstid på asynkrone tellere, er tiden nå kommet til å se på de synkrone tellere. En av utfordringene med asynkrone tellere er tidsforsinkelse, og det unngås i synkrone tellere fordi det er like lang forplantningsforsinkelse uansett antallet vipper. Riktignok blir kretsen for en synkron teller litt mer komplisert, men det er absolutt verdt det. I figur 9.7 ser vi en 3 bit synkron teller hvor utgangsverdier hentes fra henholdsvis Q_0 (LSB), Q_1 og Q_2 (MSB). Vi ser at klokkepulsen tilføres C på alle vippene, og dermed endres de synkront. Vippen til venstre er satt i «toggle» modus med HIGH, og dermed vil Q_0 endre verdi ved hver klokkepuls, og det er jo akkurat det vi ønsker av LSB. For at Q_1 skal endre verdi, må Q_0 være HØY, og dermed endrer Q_1 verdi med halve frekvensen av Q_0 som ønsket. Videre må Q_0 og Q_1 begge være HØY skal Q_2 endre verdi. Denne analysen av virkemåte er også angitt i tabell 9.1, og det resulterende tilstandsdiagrammet er vist i figur 9.8.



Figur 9.7 3 bit synkron teller

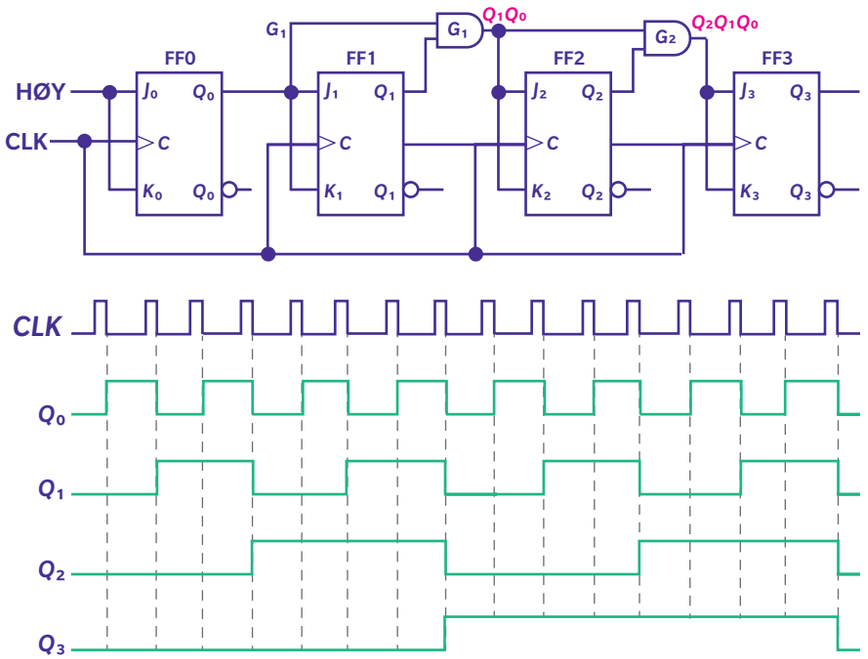
Tabell 9.1 Analyse av 3 bit synkron teller

Q_2	Q_1	Q_0	$J_2 = Q_0 Q_1$	$K_2 = Q_0 Q_1$	$J_1 = Q_0$	$K_1 = Q_0$	$J_0 = 1$	$K_0 = 1$	Q_2^{neste}	Q_1^{neste}	Q_0^{neste}
0	0	0	0	0	0	0	1	1	0	0	1
0	0	1	0	0	1	1	1	1	0	1	0
0	1	0	0	0	0	0	1	1	0	1	1
0	1	1	1	1	1	1	1	1	1	0	0
1	0	0	0	0	0	0	1	1	1	0	1
1	0	1	0	0	1	1	1	1	1	1	0
1	1	0	0	0	0	0	1	1	1	1	1
1	1	1	1	1	1	1	1	1	0	0	0



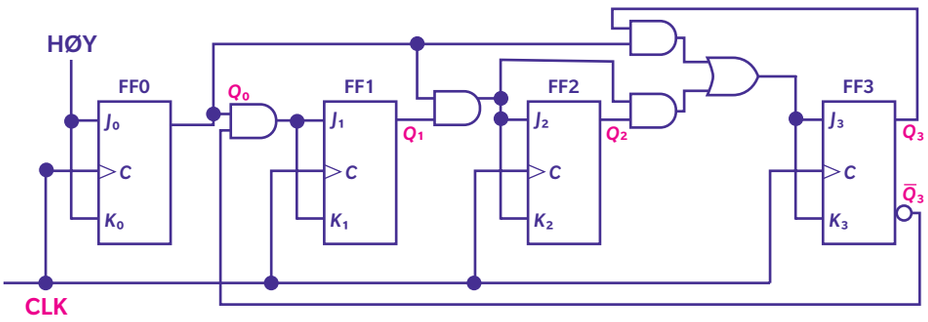
Figur 9.8 Tilstandsdiagram for 3 bit synkron teller

Dersom vi ønsker å lage større synkron tellere, så er det bare å pøse på med vipper og ANDer. Figur 9.9 viser en 4 bit synkron teller.



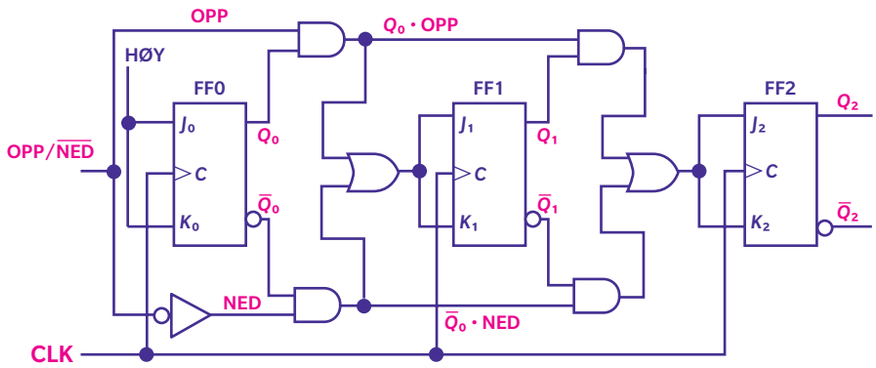
Figur 9.9 4 bit synkron teller med tidsdiagram

Vi kan også lage synkrone mod-N tellere. Figur 9.10 viser en synkron mod-10 teller som teller fra 0 til 9. Vi kjenner igjen brorparten av kretsdesignet fra 4 bits telleren i figur 9.9, men det er lagt til litt ekstra logikk for å få kretsen til å «wrappe» fra 1001 til 0000. Når Q_3 og Q_0 begge er 1, vil logikken øverst foran høyre vippe «toggle» Q_3 til 0 som ønsket. Tilbakeføringen fra Q_3 til AND-porten lengst til venstre vil sørge for å stenge telling fra LSB+1 når $Q_3 = 1$, og dermed har vi fått det som vi vil.



Figur 9.10 mod-10 4 bit synkron teller

Opp eller ned? Det er et ofte stilt spørsmål i heisen. Du skal ikke se bort ifra at det er tellerlogikk à la det som er vist i figur 9.11, som holder orden på hvilken etasje en er i.



Figur 9.11 3 bit opp-ned teller

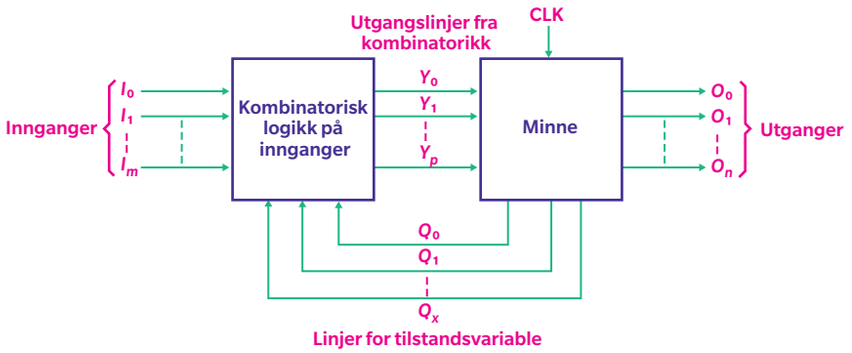
Det første en legger merke til i figur 9.11, er symmetrien. Den øverste delen kjenner vi igjen fra tidligere synkrone tellere som har hatt evnen til å telle oppover. I tillegg har vi fått et styresignal $UP / \overline{DOWN}$ som bestemmer om det skal telles opp eller ned. Når $UP / \overline{DOWN}$ er HØY, telles det oppover på vanlig måte, men når $UP / \overline{DOWN}$ er LAV, hentes tellerverdiene ut fra $\overline{Q_0}$, $\overline{Q_1}$ og $\overline{Q_2}$ isteden, og en teller dermed nedover. En måte å se at dette fungerer aldeles utmerket på er å lage seg en tabell som gitt i tabell 9.2.

Tabell 9.2 Opp og ned teller hver sin vei

Opp			**Ned**		
Q_2	Q_1	Q_0	$\overline{Q_2}$	$\overline{Q_1}$	$\overline{Q_0}$
0	0	0	1	1	1
0	0	1	1	1	0
0	1	0	1	0	1
0	1	1	1	0	0
1	0	0	0	1	1
1	0	1	0	1	0
1	1	0	0	0	1
1	1	1	0	0	0

Til nå har vi sett på tellere som teller i stigende eller synkende rekkefølge. En generell teller må kunne telle i mer avanserte mønstre. For å få til det kan kunnskap om nåværende tilstand samt annen input sammen med kombinatorisk logikk brukes til

å generere neste tilstand. Figur 9.12 viser en generell teller. Dersom du ser nøyere etter og sammenligner med kapittel 7, figur 7.2, ser du at det er en Moore-maskin.



Figur 9.12 En generell teller

En generell teller består av kombinatorisk inngangslogikk som kombinerer kjennskap om nåværende tilstand med inngangslinjer av typen kommando- og statuslinjer. I tillegg er det en minnedel som husker nåværende tilstand og forteller omverdenen om denne tilstanden via utgangsverdier. Disse utgangene er ofte koblet til en eller annen dekoder.

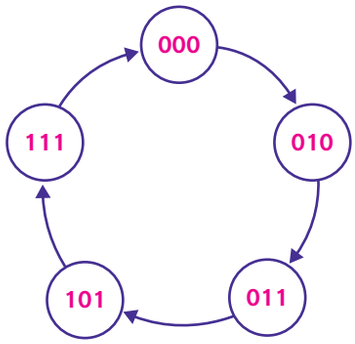
Oppskriften for å lage en teller:

1) Velg vippetype (som oftest brukes D-vippe) og bestem antallet. Årsaken til at en velger D-vippe, er at den har kun en inngang og ingen ulovlige tilstander.
2) Fyll inn tilstandsovergangsinformasjon (nåværende og neste tilstand).
3) For hver tilstand: bestem hvilke verdier en må ha på innganger for å gå over i neste tilstand.
4) For hver vippeinngang: bruk Karnaugh-diagram hvor nåværende tilstander er innganger og vippeinngang er utgang, til å bestemme hvordan de skal kobles til.
5) Tegn/konstruer kretsen.

Utlært? Det er på tide å sjekke kunnskapsnivået med en meget typisk (hint) eksamensoppgave.

Oppgave

Figuren nedenfor viser tilstandsdiagrammet for en teller.



Vi skal konstruere denne telleren ved å benytte tre synkrone D-vipper.

a. Sett opp overgangstabellen for telleren.
b. Finn uttrykkene for D-inngangene til vippene.
c. Tegn kretsskjema.

Kretsen implementerer også tilstandene 001, 100 og 110, som ikke er vist i diagrammet ovenfor.

Vis et fullstendig tilstandsdiagram med disse tilstandene inntegnet.

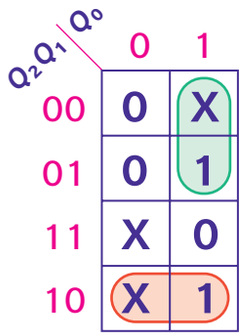
La oss starte med a. – overgangstabellen. Den finner vi ved å se på tilstandsdiagrammet som er gitt i figuren i oppgaven.

Tabell 9.3 Overgangstabell

Nåværende tilstand			**Neste tilstand**		
Q_2	Q_1	Q_0	Q_2	Q_1	Q_0
0	0	0	0	1	0
0	0	1	*X*	*X*	*X*
0	1	0	0	1	1
0	1	1	1	0	1
1	0	0	*X*	*X*	*X*
1	0	1	1	1	1
1	1	0	*X*	*X*	*X*
1	1	1	0	0	0

For de tilstandene vi ikke har i den evige runddansen, setter vi inn «don't care» X-er i neste tilstand. Deres verdi spiller jo ingen rolle når den kombinatoriske logikken skal bestemmes.

Det neste som skal gjøres (b.), er å bestemme den kombinatoriske logikken ved å finne uttrykkene for D-inngangene til vippene. De vil gi opphav til neste tilstand. Vippe D_2 tar seg av neste Q_2, D_1 hører til neste Q_1 og D_0 gir neste Q_0. La oss starte med D_2 og lage et Karnaugh-diagram basert på overgangstabellen i tabell 9.3. Vær obs på at Karnaugh-diagrammet er Gray-kodet!

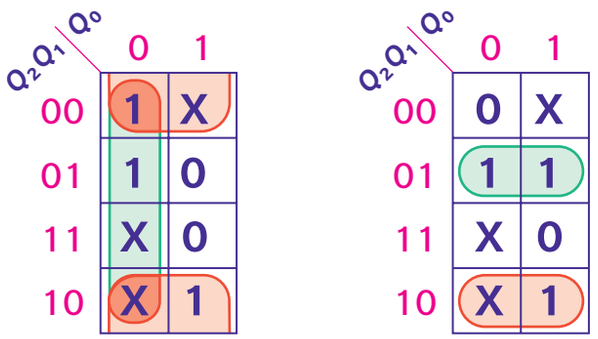


Figur 9.13 Karnaugh-diagram for D_2 (som er neste Q_2)

Vi ser at vi kan gruppere to grupper, og den øverste er gitt ved $\overline{Q_2}Q_0$ og den nederste med $Q_2\overline{Q_1}$. Uttrykket for D_2 er dermed gitt ved:

$$D_2 = Q_2\overline{Q_1} + \overline{Q_2}Q_0$$

På samme måte kan vi finne uttrykk for D_1 og D_0. Figur 9.14 viser Karnaugh-diagram for D_1 og D_0.



Figur 9.14 Karnaugh-diagram for D_1 og D_0. D_1 til venstre.

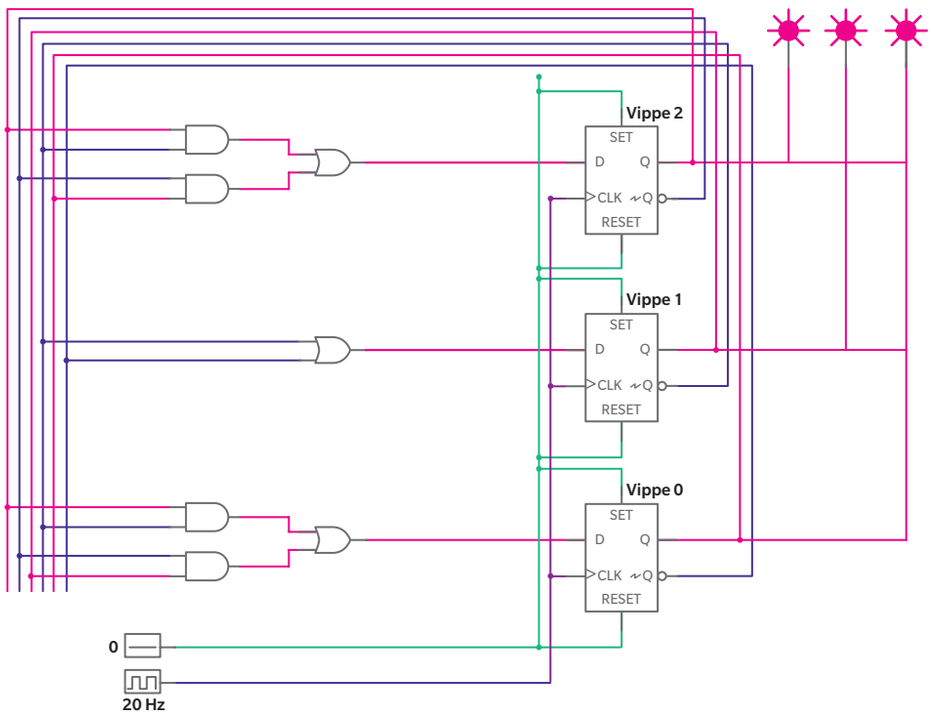
Vi ser at D_1 har store grupper, og den ene «wrapper» til alt overmål rundt øvre og nedre kant. Uttrykket for D_1 blir:

$$D_1 = \overline{Q_0} + \overline{Q_1}$$

D_0 har to små grupper. Uttrykket for D_0 blir:

$$D_0 = Q_2\overline{Q_1} + \overline{Q_2}Q_1$$

Da er det bare å gyve løs på neste utfordring (c.), nemlig å tegne kretsskjema. Vi har tre D-vipper og fører utgangene Q og $\overline{Q}$ (og dermed minnet) tilbake til inngangene gjennom den kombinatoriske logikken vi nettopp har funnet at kretsen må tilfredsstille for å telle på ønskelig måte. Som prikken over i-en sender vi utgangene Q_2, Q_1 og Q_0 videre til noen pærer som kan lyse opp tilværelsen. Herligheten er gjengitt i figur 9.15. Årsaken til at klokken er satt til 20 Hz, er at det skal blinke og lyse i bedagelig tempo.



Figur 9.15 Kretsskjema for telleren

Da er det kun en deloppgave igjen, nemlig d. Påstanden der er at kretsen også implementerer tilstandene 001, 100 og 110, og vi blir bedt om å tegne et fullstendig tilstandsdiagram. La oss starte med å fylle ut overgangstabellen som vi lagde i tabell 9.3,

og bestemme X basert på logikken vi har laget. For ikke å ta munnen for full kan vi starte med $D_2 = Q_2\overline{Q_1} + \overline{Q_2}Q_0$.

Tabell 9.4 Overgangstabell med D_2 informasjon

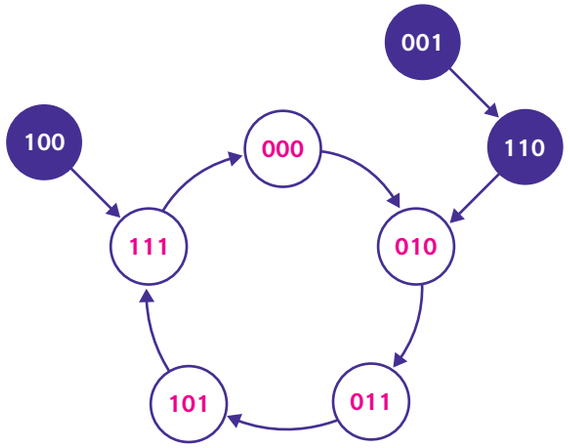
Nåværende tilstand			**Neste tilstand**		
Q_2	Q_1	Q_0	Q_2	Q_1	Q_0
0	0	0	0	1	0
0	0	1	1	X	X
0	1	0	0	1	1
0	1	1	1	0	1
1	0	0	1	X	X
1	0	1	1	1	1
1	1	0	0	X	X
1	1	1	0	0	0

Det samme kan gjøres for $D_1 = \overline{Q_0} + \overline{Q_1}$ og $D_0 = Q_2\overline{Q_1} + \overline{Q_2}Q_1$, og den endelige tabellen er vist i tabell 9.5.

Tabell 9.5 Endelig overgangstabell

Nåværende tilstand			**Neste tilstand**		
Q_2	Q_1	Q_0	Q_2	Q_1	Q_0
0	0	0	0	1	0
0	0	1	1	1	0
0	1	0	0	1	1
0	1	1	1	0	1
1	0	0	1	1	1
1	0	1	1	1	1
1	1	0	0	1	0
1	1	1	0	0	0

Dermed har vi overganger for alle åtte mulige tilstander, og tilstandsdiagrammet er vist i figur 9.16.

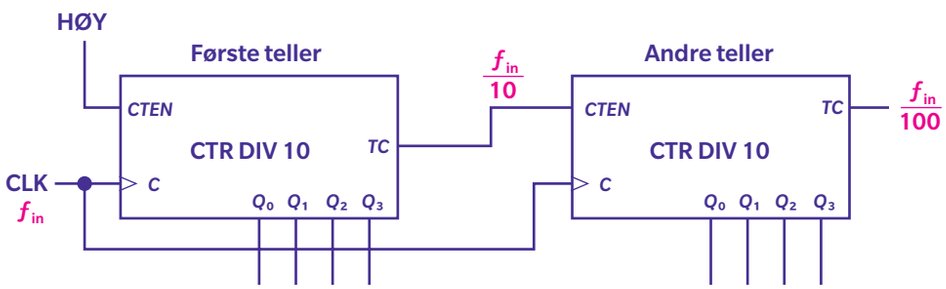


Figur 9.16 Det endelige tilstandsdiagrammet

Som en ser av tilstandsdiagrammet i figur 9.16, kan en starte i tilstandene 001, 100 og 110, men en kommer aldri tilbake igjen og forsvinner fort inn i den digitale malstrømmen.

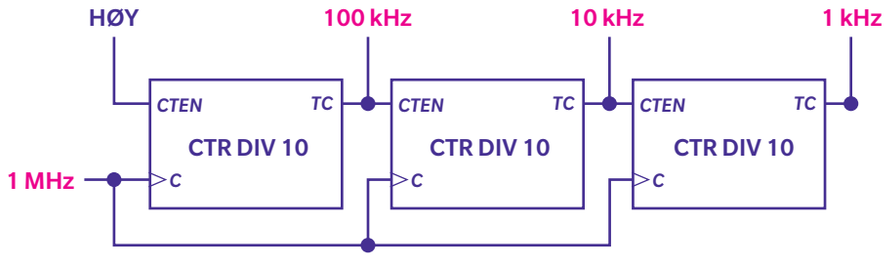
Det var det, og ofte er denne oppgavetypen den siste i oppgavesettet, så det er bare å se seg om i eksamenslokalet og smile til de andre i trygg forvissning om at dette vil gå bra. Før vi avslutter kapittelet, skal vi se på to ting til som kan være av interesse.

Større tellere kan en konstruere ved å sette sammen tellere. Figur 9.17 viser to tellere fra 0 til 9 som er satt sammen. *CTEN* er «enable» og *TC* blir logisk HØY når teller kommer til 1001.



Figur 9.17 Sammensatt teller

Sammensatte tellere kan også brukes til frekvensmanipulasjon, og figur 9.18 viser hvordan tre påfølgende tellere reduserer 1 MHz til henholdsvis 100 kHz, 10 kHz og 1 kHz.



Figur 9.18 Frekvensmanipulasjon

10

Kapittel 10

Å sette ord på det

«I believe the right combination of words strung together, like computer binary code, can unlock the secrets of the Universe.»

Alexander Bentley

LÆRINGSUTBYTTE: VHDL (VHSIC (Very High Speed Integrated Circuit) Hardware Description Language), VHDL struktur, dataflyt metode, strukturell metode, components, signals, biblioteker, testbenk, eksempler 4-til-1 multiplekser, D-vippe og teller

Etter hvert som de logiske kretsene vokste seg uhåndterlige på 1980-tallet, ønsket kunden, les det amerikanske forsvarsdepartementet, på en enkel måte å dokumentere oppførselen til kretsene som leverandørene brukte i utstyret. Da logikk som kan produseres med logiske kretser også kan lages ved hjelp av programmeringsspråk, ble det naturlig å bruke et programmeringsspråk for å beskrive kretsene. Programmeringsspråk er jo som kjent nettopp laget for å kjøre på datamaskiner konstruert med logiske kretser. Da forsvarsdepartementet allerede hadde satset på programmeringsspråket Ada, ble det brukt som basis for det nye språket VHDL (VHSIC (Very High Speed Integrated Circuit) Hardware Description Language) som beskriver logiske kretser.

Da kretsene var redusert til kodesnutter, tok det ikke lang tid før man laget simulatorer som analyserte og testet oppførselen til kretsene. Neste trinn ble selvfølgelig å lage, syntetisere, kretser ved hjelp av VHDL. Moderne digital kretsdesign er på mange måter blitt en øvelse i programmering.

Her er det på sin plass med en kraftig advarsel. VHDL er laget for å beskrive simultane «concurrent» prosesser som skal realiseres i «hardware», mens konvensjonell programmering er sekvensiell hvor koden utføres linje for linje. I VHDL-verdenen er klokken alfa og omega og styrer når hendelsene skjer.

Hvordan skal en lære seg et nytt skriftspråk? Ett alternativ er å studere grammatikken og språkets struktur til det kjedsommelige. Et annet er å prøve å lese større stykker tekst og på den måten bokstavelig talt knekke koden. La oss prøve litt av begge deler.

Noe av det første vi så på, var OG-funksjonen. Symbolet som beskriver den er gitt i figur 10.1.

Figur 10.1 OG-funksjon

Hvordan kan så oppførselen til en OG-funksjon dokumenters i kode? La oss se på følgende beskrivelse:

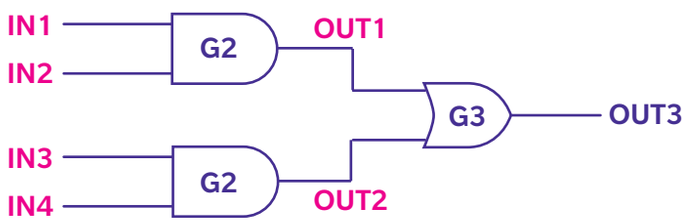
```
entity OG_port is
   port (A, B: in bit; X: out bit);
end entity OG_port;

architecture OG_funksjon of OG_port is
begin
   X <= A and B;
end architecture OG_funksjon;
```

Det første som slår en, er at dette var da en omstendelig måte å uttrykke den enkle OG-funksjonen på. Trøsten er at mengden kode blir vesentlig mindre etter hvert som de logiske kretsene blir større. I tillegg har denne måten å beskrive kretser på den fordel at den kan brukes både til analyse, simulering og syntese av kretser.

Koden er delt i to med en **entity** enhet og en **architecture** del. **Entity** beskriver forholdet til omverdenen, som i dette tilfellet er en port med to **in** bit A og B og en **out** bit X. Tekst med fet skrift er reserverte ord som er forbeholdt språket VHDL. Vær oppmerksom på at VHDL ikke bryr seg om forskjellen på små og store bokstaver. **Architecture** delen beskriver selve funksjonen, som i dette tilfellet er en OG-funksjon mellom A og B og gitt med tilordningen X<=A **and** B. **Entity** OG_port og **architecture** OG-funksjon knyttes sammen i «architectures» første linje **architecture** OG_funksjon **of** OG_port. Resten av koden er tekst bestemt av programmereren. Vel, bortsett fra bit som er en forhåndsdefinert variabel (identifier) med 0 og 1 som gyldige verdier. For å få frem det som var kodet av meg, tillot jeg meg å skrive den delen på norsk. Det er ikke så vanlig, så fra nå av vil all koden bli skrevet på engelsk. Det vi allerede har gjort, vil da se slik ut:

```
entity AND_gate is
   port (A, B: in bit; X: out bit);
end entity AND_gate;

architecture AND_function of AND_gate is
begin
   X <= A and B;
end architecture AND_function;
```

La oss prøve oss med et annet og kanskje litt mer utfordrende eksempel. I figur 10.2 er to OG-funksjoner satt sammen med en OR-funksjon.



Figur 10.2 Sammensatt funksjon

En mulig måte å skrive kode for denne sammensatte funksjonen på er:

```
-- (This is a VHDL comment)

-- This is a code example using dataflow approach

-- This is the IEEE library
library IEEE;

-- This is the entity for the function given in Figure 10.2
entity AND_OR_Logic is
   port (IN1, IN2, IN3, IN4: in bit; OUT3: out bit);
end AND_OR_Logic;

architecture LogicOperation of AND_OR_Logic is
begin
   OUT3<= (IN1 and IN2) or (IN3 and IN4);
end architecture LogicOperation;
```

Denne måten å skrive kode på kalles dataflyt, og i dette tilfellet har en nøstet de tre funksjonene i figur 10.2 sammen i en større logisk operasjon i **architecture**. Denne måten å gjøre det på gir kompakt kode, men kan bli uoversiktlig dersom koden blir stor. I dette eksempelet er også kommentarer gitt etter VHDL syntaksen -- introdusert. I tillegg er det generelle biblioteket IEEE blitt spesifisert.

En annen angrepsmåte er å strukturere koden. Fordelen med den metoden er at en kan bygge seg kodekomponenter som kan gjenbrukes. Det en taper ved å være litt mer omstendelig til å begynne med, vinner en på i det lange løp. Det må riktignok gjøres en avveining. Er det snakk om korte kodebiter, er nok dataflytmetoden å foretrekke. En strukturell kode for den logiske kretsen i figur 10.2 kan se slik ut:

```
-- This is a code example using structural approach

-- This is the IEEE library
library IEEE;

-- This is the entity for AND_gate
entity AND_gate is
   port (A, B: in bit; X: out bit);
end entity AND_gate;
```

```
architecture AND_function of AND_gate is
begin
   X<=A and B;
end architecture AND_function;

-- This is the entity for OR_gate
entity OR_gate is
   port (A, B: in bit; X: out bit);
end entity OR_gate;

architecture OR_function of OR_gate is
begin
   X<=A or B;
end architecture OR_function;

-- This is the entity for the function given in Figure 10.2
entity AND_OR_Logic is
   port (IN1, IN2, IN3, IN4: in bit; OUT3: out bit);
end AND_OR_Logic;

architecture LogicOperation of AND_OR_Logic is
   component AND_gate is
      port (A, B: in bit; X: out bit);
   end component AND_gate;

   component OR_gate is
      port (A, B: in bit; X: out bit);
   end component OR_gate;
   signal OUT1, OUT2: bit;

begin
   G1: AND_gate port map (A => IN1, B => IN2, X => OUT1);
   G2: AND_gate port map (A => IN3, B => IN4, X => OUT2);
   G3: AND_gate port map (A => OUT1, B => OUT2, X => OUT3);
end architecture LogicOperation;
```

Denne siste koden er vesentlig lengre. Til å begynne med har vi også tatt med en **entitity** for OR_gate, da figur 10.2 inneholder både OG- og OR-funksjoner. Inne i **architecture** LogicOperation **of** AND_OR_Logic **is** er det lagt til to **component** strukturer for henholdsvis AND_gate og OR_gate. Disse **component** gjør det mulig å bruke ferdiglagde strukturer slik som AND_gate og OR_gate uten å skrive dem om

og om igjen. Mellom **begin** og **end** i **architecture** LogicOperation **of** AND_OR_Logic is opprettes instanser av de enkelte gatene, og **port map** sørger for at gatenes terminal knyttes til de riktige forbindelseslinjer. G1, G2 og G3 er labler. Det eneste som ikke er nevnt, er deklarasjonen **signal** OUT1, OUT2: bit;. **Signal** er et reservert ord for indre forbindelseslinjer i den gitte krets. Dersom du har vært borte i programmering, kan du tenke på **signal** deklarasjon som deklarasjon av lokale variabler.

«Det som ikke er testet, virker ikke.»

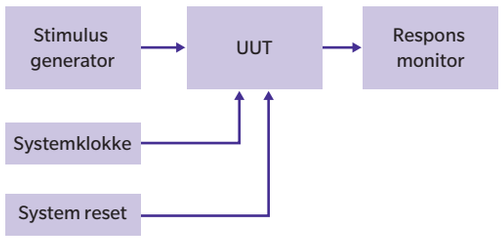
Dette utsagnet fra en ukjent programmerer er så sant, så sant. Heldigvis er det lett å lage såkalte testbenker i VHDL for å teste ut kode. La oss ta et enkelt eksempel med utgangspunkt i AND_gate:

```
entity AND_gate is
   port (A, B: in bit; X: out bit);
end entity AND_gate;

architecture AND_function of AND_gate is
begin
   X <= A and B;
end architecture AND_function;
```

En testbenk for AND_gate kan se slik ut:

```
entity Test_AND_gate is
end entity Test_AND_gate;

architecture IO of Test_AND_gate is
   signal K, L, M: bit;
begin
   G1: AND_gate port map (A => K, B => L, X => M);
   K <= '0', '1' after 100 ns;
   L <= '0', '1' after 150 ns;
end architecture IO;
```

Det første som en kan observere, er at **entity** Test_AND_gate ikke har noen forbindelser til verden utenfor. Alt foregår i testmiljøet. I architecture til Test_AND_gate opprettes en instans av AND_gate som testes med varierende K og L, og resultat leveres til M. **After** er et reservert ord som forteller at verdiene av K og L skal settes til 1 etter henholdsvis 100 og 150 ns. Testbenker kan lages vesentlig mer avansert enn dette, hvor en sender klokkesignal, tidsserier med variabler og observerer resultater og finner feil ved hjelp av tidsdiagram. Figur 10.3 viser en generell testbenk. UUT er «Unit Under Test».



Figur 10.3 Typisk testbenk

Flere byer i Belgia? Vel, med VHDL ligger den digitale verden for våre føtter. La oss prøve å lage 4-til-1 multiplekser samtidig som det innføres litt mer VHDL terminologi.

```
-- This is a 4-to-1 MUX

-- This is the IEEE library
library IEEE;
-- To get some useful identifiers
use IEEE.std_logic_1164.all;

entity MUX is
   port (A, B, C, D: in std_logic; S: in std_logic_vector(1 downto 0); X:
   out std_logic);
end entity MUX;

architecture MUXLogic of MUX is
begin
   process (A, B, C, D, S)
   begin
      case S is
         when "00" => X <= A;
         when "01" => X <= B;
         when "10" => X <= C;
         when others => X <= D;
      end case;
   end process;
end architecture MUXLogic;
```

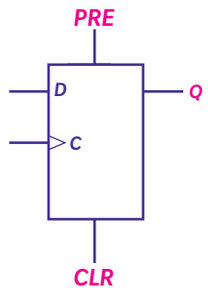
Vi har utvidet repertoaret med variabler (instanser) ved å bruke **use** IEEE.std_logic_1164;. Da får vi tilgang til std_logic som kan brukes istedenfor bit. Std_logic har syv nye mulige verdier i tillegg til 0 og 1. Disse andre verdiene med blant annet 'Z' for høy impedans, les brudd, er nyttige ved mer komplekse kretser og når testbenker

skal lages. Std_logic_vector er en vektor som kan inneholde std_logic verdier, og i vårt tilfelle er den definert til å ha lengde to med utsagnet std_logic_vector(1 **downto** 0).

I selve **architecture** har vi brukt en ny konstruksjon kalt **process,** som er hendig å bruke når en har sekvensielle prosesser. Når en anvender **process,** utfører en såkalt oppførsel metode («behavioral style») VHDL programmering. Bruken av **process** minner om konvensjonell programmering. Process har en sensitivitetsliste som her er (A, B, C, D, S). Det er bare hver gang en av disse variablene (signalene) endrer seg, at **process** blir utført. Du bør lage dine **process** snutter små. Det skal tross alt realiseres i kretser til slutt.

Vi har altså tre ulike angrepsmåter i programmeringen – dataflyt, strukturelt eller oppførselsbasert. Hva skal en velge? Det som til enhver tid passer best. Typisk blir koden en salig blanding av de tre ulike stilene.

Dette var jo på grensen til morsomt. Er det mer som kan lages? Hva med en D-vippe (Flip-flop) som vist i figur 10.4?



Figur 10.4 D-vippe

```
-- This is a D Flip-flop

-- This is the IEEE library
library IEEE;
-- To get some useful identifiers
use IEEE.std_logic_1164.all;

entity DFfl is
   port (D, Clock, Pre, Clr: in std_logic; S: Q: out std_logic);
end entity DFfl;
```

```
architecture DFflLogic of DFfl is
begin
   process
   begin
      wait until rising_edge(Clock);
-- Check for Preset and Clear conditions
      if Clr = '1' then
         if Pre = '1' then
            if D = '1' then
-- Q input follows D input when Clr and Pre inputs are High
               Q <= '1';
            else
               Q <= '0';
            end if;
         else
-- Q is set HIGH when Pre input is LOW
            Q <= '1';
         end if;
      else
         Q <= '0';
      end if;
-- Q is set LOW when Clr input is LOW
   end process;
end architecture DFflLogic;
```

Her møter vi en del nye ting. **Wait until rising edge**(Clock) som er brukt i **process,** venter på stigende klokkeflanke før innsignal D blir prosessert for å gi en mulig ny Q verdi. Det er synkron CLR og PRE siden dette står inni struktur som trigger på positiv klokkeflanke.

Da har vi vært igjennom de fleste digitale kretskonstruksjonene vi har sett på så langt. Er det noen som er glemt? Ja, før vi gir oss, må vi ta med et tellereksempel. Nedenfor er det gitt et eksempel på en slik teller, og forklaringen på dens virkemåte er gitt i VHDL kommentarer underveis.

```
-- This is a 4 bit counter

-- This is the IEEE library
library IEEE;
-- To get some useful identifiers
use IEEE.std_logic_1164.all;
```

```
-- To get the unsigned type and the + operator
use IEEE.numeric_std.all;

entity Counter is
-- The generic statement defines here the width of the counter
generic (Width : integer := 4);
-- Unsigned is a vector of bit
   port (D, Clock, Reset, Load: in std_logic;
         Data: in unsigned (Width-1 downto  0);
      Q: out unsigned (Width-1 downto 0));
end entity Counter;

architecture CounterLogic of Counter is
-- cnt holds the current counter value and wraps when it is full
signal cnt : unsigned (Width-1 downto  0);
begin
   process (Reset, Clock)
   begin
      if Reset = '1' then
-- All the bits in the cnt vector is set to zero
         cnt <= (others => '0');
      elsif rising_edge(Clock) then
         if Load = '1' then
-- The counter cnt is loaded with counter value from Load
            cnt <= Data;
         else
-- The counter is incremented with one and wraps if full
            cnt <= cnt + 1;
         end if;
      end if;
   end process;
-- Counter value cnt is given to Q
   Q <= cnt;
end architecture CounterLogic;
```

Er vi så utlært i VHDL? Er vi kommet til veis ende? Nei, det er en god del mer å lære, men vi har fått med oss grunnprinsippene. Når man skal lage store programmerbare logiske kretser, er en nødt til å ta VHDL i bruk. Faktisk er digital kretskonstruksjon blitt ren programmering. En kan derfor spørre seg om hva hensikten var med de innledende kapitlene hvor en lærte digitalteknikk logisk og skjematisk. Fortvil ikke! Ingenting er som den indre tilfredsstillelsen av å vite hvordan tingene egentlig henger sammen. I tillegg kommer den grunnleggende og generelle kunnskapen om digitale kretser

godt med når man utfører VHDL programmering. Ting kan virke fint i simulering på programmeringsnivå, men etter syntese er ikke alltid det tilfellet. Når ting ikke lenger fungerer på grunn av timingproblemer, er det nyttig med kunnskap om hvordan alt henger sammen, slik at man kan gjøre fornuftige designvalg. Det er nyttig når man lager testbenk også, å vite at verdiene på signalene bruker litt tid på å forplante seg, og at ikke alle verdier er tilgjengelige overalt samtidig, eller at utganger ikke er stabile og klare akkurat med en gang.

Lykke til!

11

Kapittel 11

Logikk for viderekommende

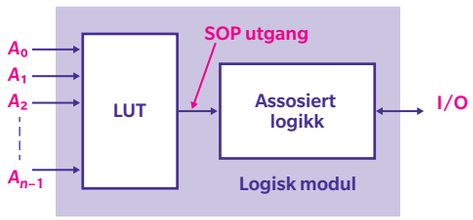
«If we will attentively consider new born children, we shall have little reason to think that they bring many ideas into the world with them.»

An Enquiry Concerning Human Understanding, John Locke (1632–1704)

LÆRINGSUTBYTTE: Programmerbar logikk, FPGA (Field Programmable Gate Arrays), logisk modul, oppslagstabell (LUT Look-Up Table), CLB (Configurable Logic Blocks), I/O blokker, flyktig eller permanent, konfigurasjonsminne, hard-core, soft-core, plattform FPGA, designflyt, skjematisk versus VHDL, simulering, syntese, implementering, timing, Boundary scan test, JTAG (Joint Test Action Group), intest, extest

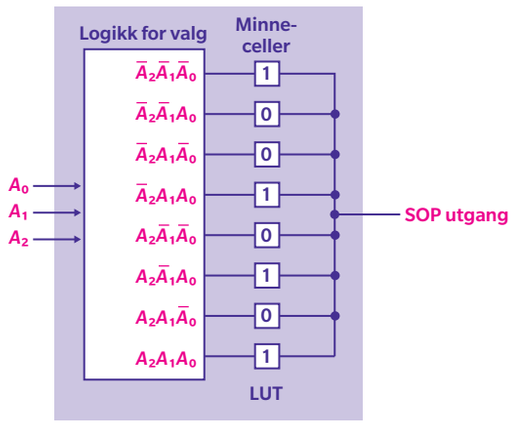
Winston Smith bøyer hodet ned i den kalde vinden på vei til en ny dag på jobb i SANNHETSMINISTERIET. Der sitter han i sin kontorcelle og omskriver små fliker av fortiden slik at den passer med den til enhver tid gjeldende SANNHET. Rundt ham sitter tusenvis av andre og gjør det tilsvarende. Sammen kan de produsere den virkelighet som STORE BROR til enhver tid måtte ønske seg. Det er kun ett problem. Winston har minne og kan derfor huske hvordan ting var før.

Å sammenligne SANNHETSMINISTERIET i George Orwells roman 1984 med slik den programmerbare logikken i Field Programmable Gate Arrays (FPGA) virker, er kanskje ikke så tussete som det først kan synes. FPGA er i bunn og grunn en enorm matrise av sannhetstabeller som er knyttet sammen slik at den kan programmeres til å gjøre tilnærmet hva som helst. FPGA er logikkens tabula rasa (rene bord). Der hvor Winston satt i sitt cellekontor, er det i FPGA en såkalt logisk modul.



Figur 11.1 FPGA logisk modul

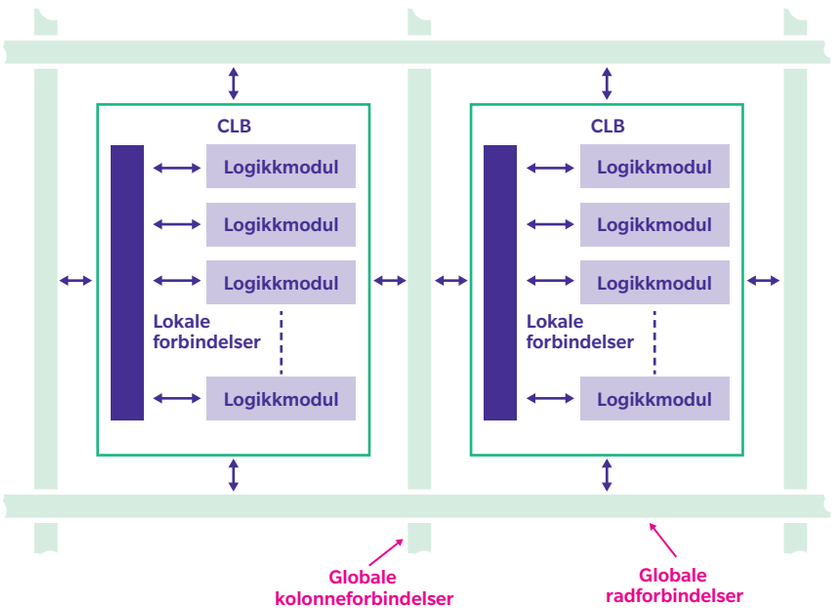
Den logiske modulen som er angitt i figur 11.1 består av to deler, henholdsvis en oppslagstabell LUT (Look-Up Table) og assosiert logikk. LUT-en består av en rekke minneceller (2^n) hvor n er antall inngangsvariabler. LUT-en brukes til å kunne generere et hvilket som helst SOP-uttrykk (altså en hvilken som helst sannhetstabell). I figur 11.2 (Digital Fundamentals [2] side 579) er det gitt et eksempel på en LUT som produserer uttrykket $\overline{A}_2\overline{A}_1\overline{A}_0 + \overline{A}_2A_1A_0 + A_2\overline{A}_1A_0 + A_2A_1A_0$.



Figur 11.2 Programmert LUT

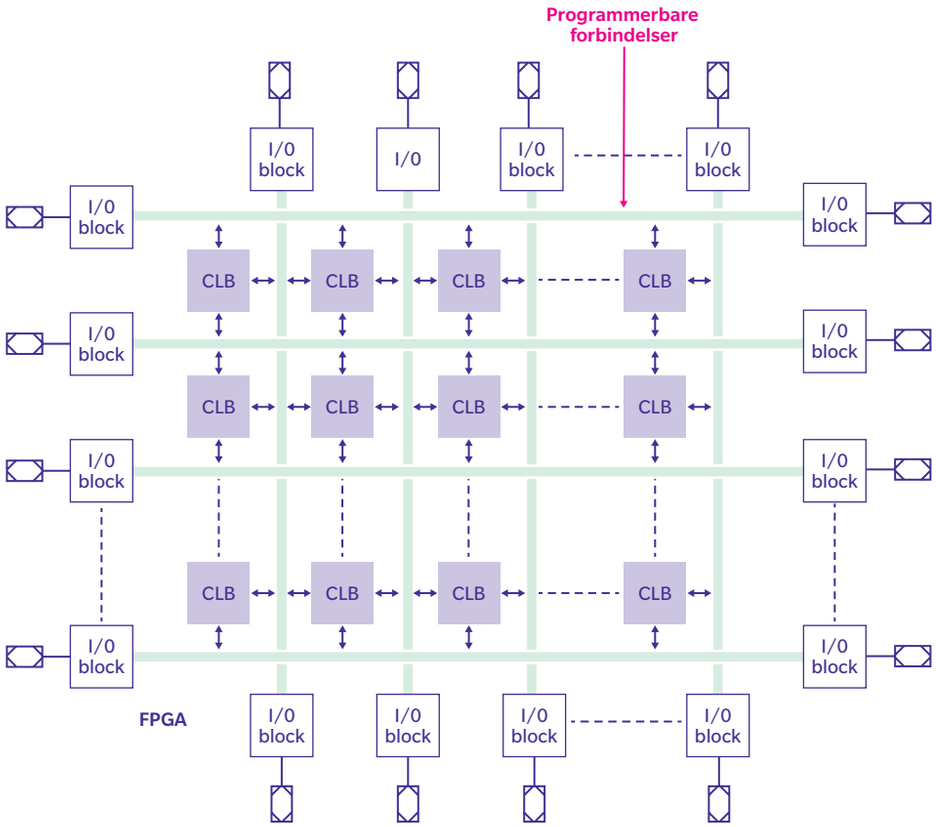
For å kunne huske uttrykket trengs det en flip-flop, og den befinner seg i den assosierte logikken. Å huske er essensielt i sekvensiell logikk, hvor kommende tilstander er bestemt av foregående.

De logiske modulene i en FPGA er satt sammen i såkalte CLB-er (Configurable Logic Blocks) som kan kommunisere med hverandre i en matrisestruktur.



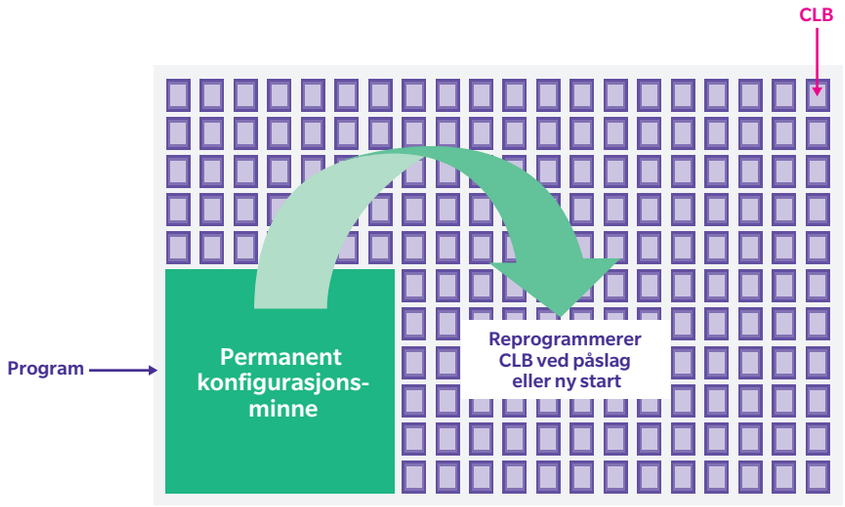
Figur 11.3 CLB

En FPGA må jo kunne kommunisere med utenverdenen og har derfor en rekke programmerbare tilkoblingspunkter, I/O (In/Out) blokker, langs randen av matrisen (Figur 11.4).



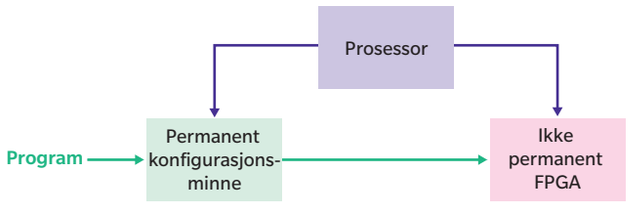
Figur 11.4 FPGA struktur

I utgangspunktet er en FPGA en enorm matrise av tomme sannhetstabeller som i prinsippet kan brukes til å lage nær sagt hva som helst. En FPGA kan enten være flyktig, dvs. at den mister innholdet i sine LUT-er når strømmen blir borte, permanent eller en salig blanding. I det siste tilfellet, som vist i figur 11.5, vil FPGA-en ha et permanent konfigurasjonsminne som sørger for riktig innhold i oppslagstabellene (LUT-ene) ved oppstart.



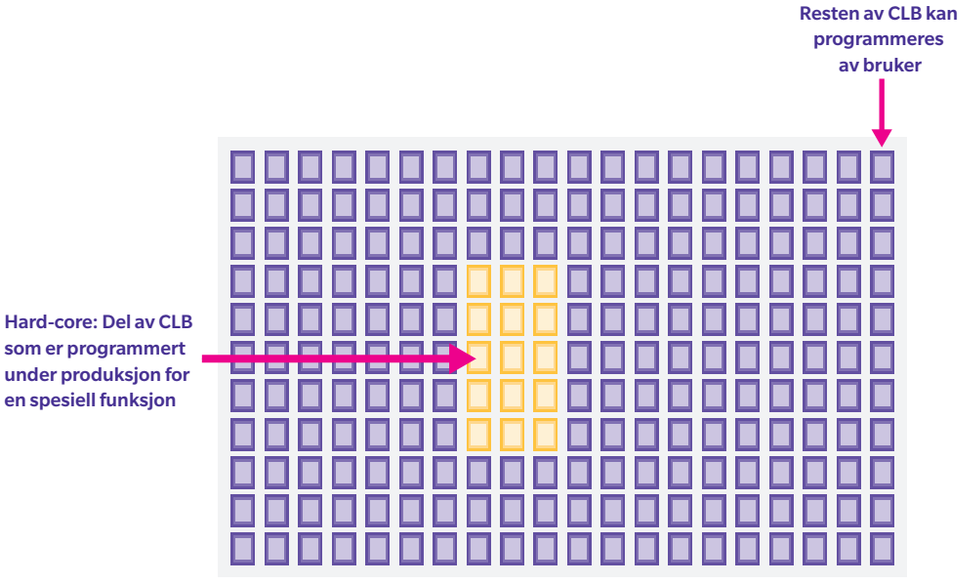
Figur 11.5 FPGA med konfigurasjonsminne

Dersom FPGA-en er fullstendig flyktig, må den programmeres hver gang den starter på nytt, som angitt i figur 11.6.



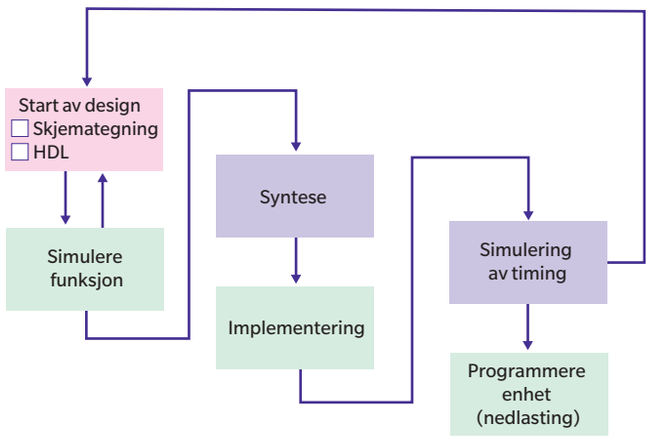
Figur 11.6 Flyktig FPGA

For at brukerne ikke skal finne opp hjulet på nytt hver gang, leverer produsentene av FPGA ferdigprogrammerte deler, såkalte hard-cores. De består av funksjoner som er mye brukt i digitale systemer, slik som mikroprosessorer, standard I/O grensesnitt og signalprosessering. Dersom de innebygde funksjonene har programmerbare funksjoner, kalles de «soft-core». I figur 11.7 er det en FPGA med hard-core. FPGA-er som inneholder enten «hard-core», «soft-core» eller en blanding, kalles plattform FPGA-er, da de kan implementere hele systemer.



Figur 11.7 FPGA med «hard-core»

En ting er å ha alle verdens muligheter som en FPGA gir, men hvordan skal man utnytte det? Hvor skal en begynne? Nede i høyre hjørne? Spøk til side! Her må en gå systematisk til verks. Figur 11.8 viser en naturlig designflyt.



Figur 11.8 Designflyt for FPGA

Jeg antar du har en PC, så det første du må skaffe deg, er programvare til designfasen. Det er to måter å starte et design på – enten skjematisk eller ved å bruke VHDL. I begge tilfellene er det lurt å arbeide strukturert og lage et blokkbasert design.

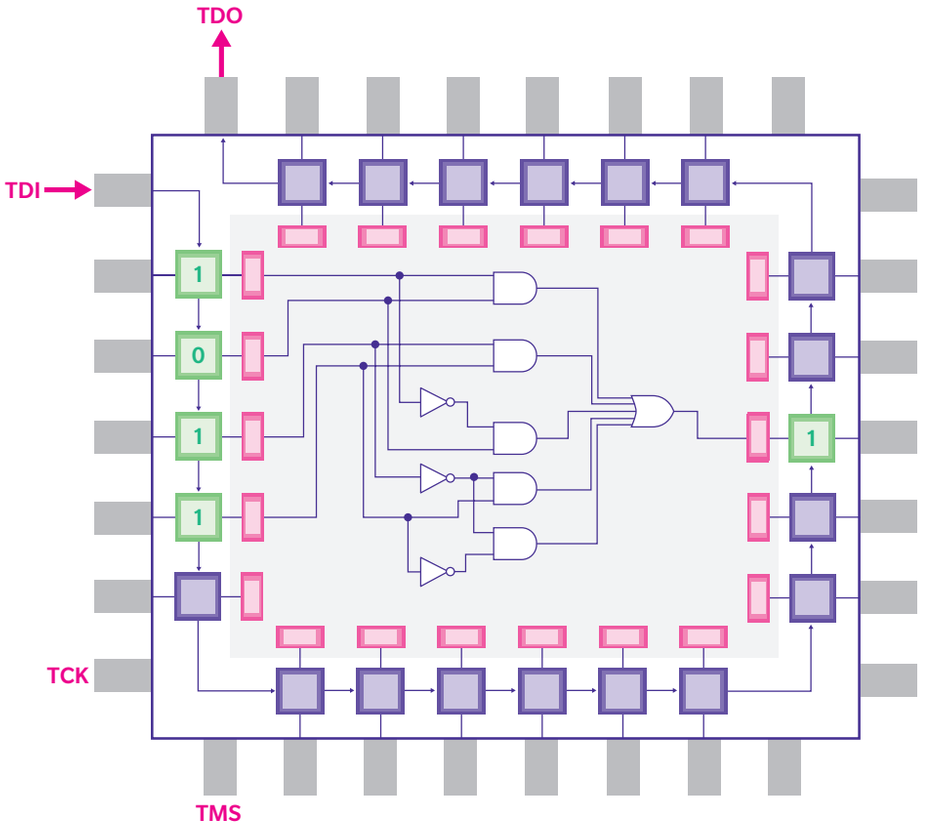
Etter å ha laget et første utkast foretas det en funksjonell simulering, for å teste at kretsen oppfører seg som forventet. Det er to ulike måter å teste på, enten grafisk ved å sende pulstog på inngangene eller ved å lage en testbenk i VHDL. En tester i begge tilfeller at kretsen som skal lages, følger logikken som var ønsket. Hvis ikke det skjer, er det tilbake til start.

Etter at en har fått logikken på plass, utføres det en syntese av designet. I den fasen optimaliseres og effektiviseres designet slik at det bruker færrest mulig porter og eventuell overflødig logikk fjernes. Syntesen produserer en såkalt nettliste som beskriver den optimaliserte logiske kretsen.

Etter syntesen er det en såkalt implementeringsfase hvor designet tilpasses den FPGA som skal brukes. Før designet lastes ned på den ønskede FPGA, foretas det en tidssimulering for å se at det vil fungere på en gitt frekvens, og at en ikke har problemer med «timing». Som en siste økt sendes designet som en strøm av 0 og 1 til den aktuelle krets.

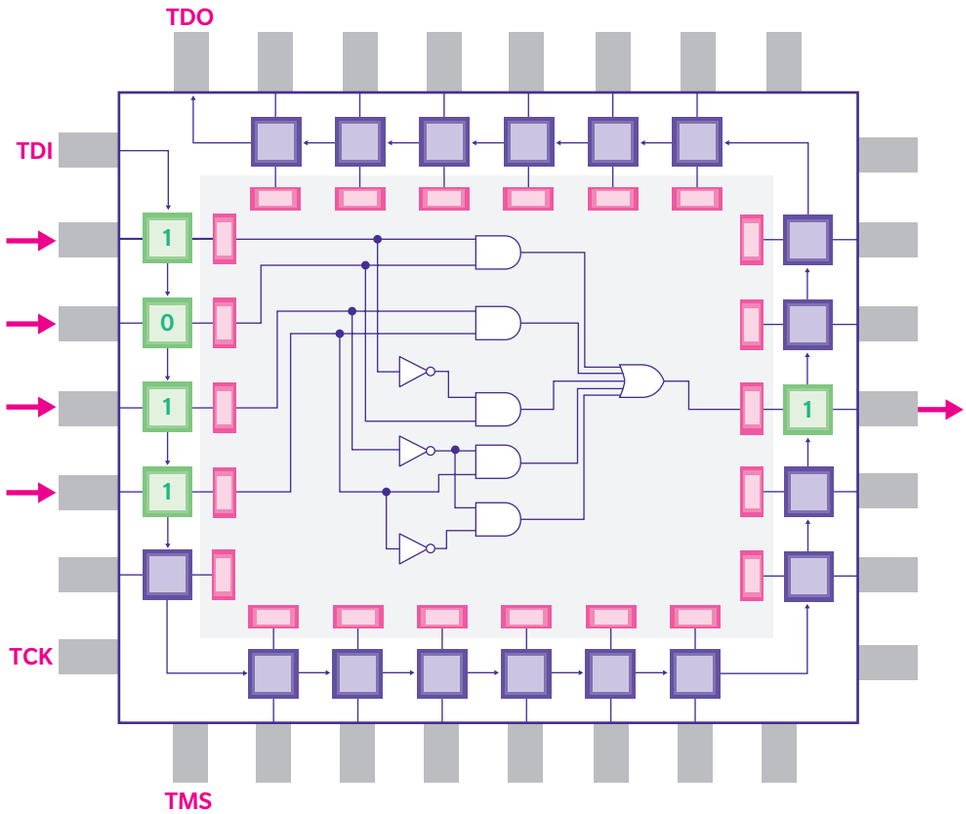
De første FPGA-er startet som små 8x8 matriser på 1980-tallet. Siden den gang har utviklingen av disse kretsene fulgt Moores lov, antallet transistorer på et areal dobles hver 24. måned, som om den skulle vært en naturlov. Etter hvert som kretsene ble enorme, dukket behovet for å teste enkeltdeler og funksjoner opp. Da disse kan befinne seg langt inne i rutenettet, ble det laget en standard, IEEE 1149.1, av JTAG (Joint Test Action Group) som spesifiserte interne testpunkter og hvordan de skulle rutes til FPGA-ens I/O blokker. Når en tester, kan derfor I/O blokkenes tilkoblinger redefineres midlertidig slik at en får tilgang til ønsket indre logikk. Denne testmetoden kalles for «boundary scan test», da en bruker I/O blokkene for å få tilgang til indre testpunkter.

Når deler av en FPGAs indre logikk skal testes, brukes såkalt Intest. Et bitmønster av 1 og 0 sendes inn på pinnen testdata inn (TDI Test Data In), og resultatet leses av på TDO (Test Data Out). Ved å sende inn alle mulige ønskede bitmønstre, kan oppførselen til en ønsket del av den indre logikken bli undersøkt og feilsøkt. TCK (Test ClocK) er tilkoblingspunkt for testklokke, og TMS (Test Mode Select) angir testmodus. Et eksempel på «Intest» er vist i figur 11.9.



Figur 11.9 «Intest» av FPGA

Det er også mulig å utføre «Extest», som vist i figur 11.10, og da blir i tillegg den eksterne logikk og I/O blokkene testet. Testsignalene sendes rett på I/O blokkene, og resultatet hentes ut på ønsket I/O blokk.



Figur 11.10 «Extest» av FPGA

12

Kapittel 12

UARTig?

«Teori er når man vet alt, og ingenting stemmer. Praksis er når alt stemmer, og ingen vet hvorfor. Her blir teori og praksis forent, ingenting stemmer og ingen vet hvorfor.»

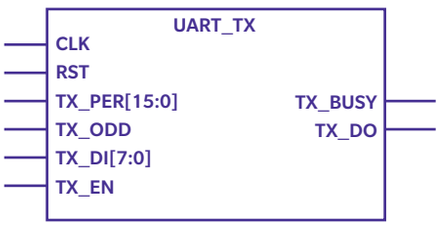
Livsvisdom

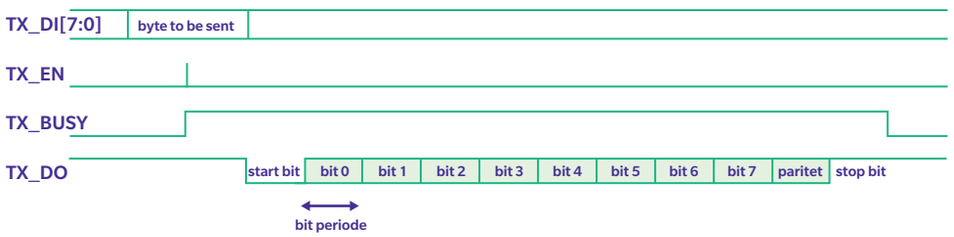
LÆRINGSUTBYTTE: UART_TX utvikling med porter og vipper og VHDL, UART anvendelser, syntese, systemtenkning, arkitektur og blokkskjema, strukturert hierarkisk design, testing, synkront sekvensielt design, hensikten med synkront design, utfordringer i den digitale hverdag, synkron D-vippe, synkron teller, testbenk, logisk konverter, VHDL utviklingsmiljø, testbenk og UUT (Unit Under Test), synkront VHDL design.

Hva er nytten av kunnskap når en ikke kan bruke den? Designoppgaven har ligget på pulten i månedsvis uten at jeg har tatt fatt på den. For å være ærlig, så har jeg prøvd å skjule den så godt jeg har kunnet under andre og mer presserende oppgaver. Hvorfor? Fordi den utfordrer meg. Den krever at jeg bruker tilegnet kunnskap for å løse nye problemer. Den tar meg ut av analysens komfortsone og inn i syntesens verden med et vell av mulige løsninger. Prokrastinering var et av de første nye ordene jeg lærte da jeg begynte å arbeide i akademia. Jeg måtte faktisk slå det opp: «Prokrastinering, kronisk utsettelsesadferd, som preges av å være ufornuftig eller irrasjonell og til ulempe for en selv.» Både studenter og lærere gjør det, men i dag har jeg bestemt meg for å ta den «sure silden» først. Og som så ofte før, viste den seg å være den beste.

DESIGNOPPGAVE UART_TX

En serieport/UART (Universal Asynchronous Receiver/Transmitter) benyttes for å sende og motta data på seriell form. Vi skal fokusere på senderdelen UART_TX. Se blokkskjema og timingdiagram og beskrivelse av IO under.





Signal Navn	Retning	Forklaring
CLK	IN	System klokke
RST	IN	Synkron reset
TX_PER[15:0]	IN	Konfigurasjon av bit periode. Eksempel: verdi på 100 (desimalt) betyr at lengden av et bit på seriell utgang skal være 100 perioder av system klokken.
TX_ODD	IN	0: Even paritet 1: Odd paritet
TX_DI[7:0]	IN	Data Byte som skal sendes ut på seriell utgang TX_DO

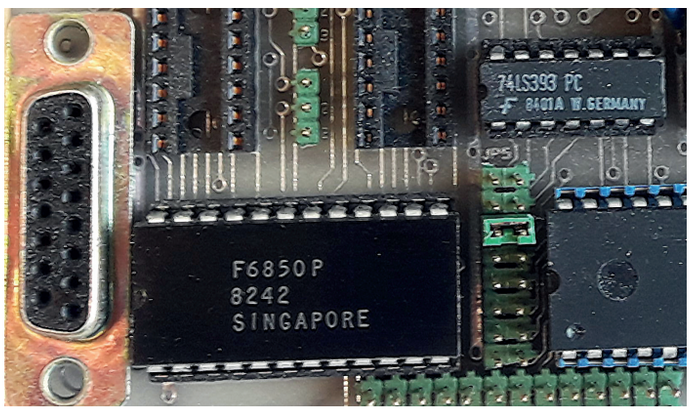
Signal Navn	Retning	Forklaring
TX_EN	IN	Høy puls med varighet på en klokke periode starter sending av data fra TX_DI[7:0] ut på TX_DO
TX_BUSY	OUT	0: UART_TX modul er ledig, 1: UART_TX modul er opptatt med å sende en byte.
TX_DO	OUT	Seriell utgang

Designoppgave:

1) Forslå en grov arkitektur basert på funksjonelle blokker som: register, skiftregister, tellere, tilstandsmaskin, mux, dekoder … (ikke nødvendigvis alle).
2) Realiser arkitektur detaljert med vipper og porter i Multisim og simuler.
3) Realiser arkitektur detaljert i VHDL og simuler i ModelSim.

Vi skal altså lage en UART (Universal Asynchronous Receiver/Transmitter). Ifølge min kollega Svein, som har gitt oss designoppgaven, er vi heldige, da vi kun blir bedt om å lage senderdelen. Mottakeren er visst verre å lage. Vårt design skal være en liten del av et større FPGA design for en IP-modul. Det skal foreslås en grov arkitektur basert på funksjonelle blokker som vi har vært borti før. Designet vårt skal realiseres i både vipper og porter og VHDL kode.

Et naturlig spørsmål å stille seg er om UART er mye i bruk. Absolutt! Den finnes i mikroprosessorer og haugevis av andre enheter som trenger å få overført data serielt. Du kan for eksempel bare ta en tur ut i sikringsskapet ditt. I den automatiske strømmåleren finner du en UART. Skulle du ønske å monitorere strømforbruket i sanntid, er det bare å koble seg på der. I figur 12.1 ser du en 1980-tallsvariant av en UART realisert i hardware.



Figur 12.1 F 6850 P er et eksempel på en UART. Den kommuniserer via RS232 porten til venstre

Hvem oppfant UART-en? Igjen kommer Wikipedia oss til unnsetning. Det var Gordon Bell som ikke jobbet på Bell Labs, men i DEC. Under utviklingen av datamaskinen PDP-1 i 1959 så den første UART dagens lys.

Tilbake til designet. Spesifikasjonen viser en UART_TX blokk med innganger og utganger. Data inn TX_DI[7:0] er tilgjengelig på 8 innganger. TX_PER[15:0] gir bitperioden på 16 innganger. Hvilken paritet en skal ha, gis med TX_ODD. Dersom du er nysgjerrig på hva en skal med paritet, kan det være

lurt å ta en liten titt i kapittelet «Fra A til B». Systemklokken CLK er tilgjengelig på inngangen. Synkron reset RST nullstiller hele kretsen dersom den «enables». TX_EN starter sending av data. På sendersiden er det to utganger, TX_DO og TX_BUSY. På den serielle utgangen TX_DO sendes data, og TX_BUSY er 1 så lenge data sendes.

I tidsdiagram ser en at byten som skal sendes TX_DI[7:0] skal ligge klar når sendingen starter med en TX_EN puls. Så går TX_BUSY til 1 og blir liggende slik til byten på TX_DI[7:0] er sendt ut på TX_DO.

Oi sann, hvordan skal vi ta tak i denne designutfordringen? Her gjelder det å rydde tankene slik at det kan lages en god struktur. Hva med å la reisen starte med det kjente og så seile inn i det mer ukjente etter hvert? Ifølge Svein er det nye ting på gang innenfor digitalt design, nemlig såkalt «behavioral- and high level synthesis». Tanken er at en i en ikke for fjern fremtid kan skrive kode i høynivåspråk som C++ og lignende for å designe kretser istedenfor VHDL eller enda mer basalt vipper og porter.

Kanskje vi skal foregripe begivenhetens gang ved å skrive en kode i høynivåspråk for UART_TX designet og bruke koden til å få orden på tanker og arkitektur. Bjørnstjerne Bjørnson valgte seg april, da i den det gamle faller og det nye får feste. Vi bruker høynivåspråket C++ av gammel vane og antar at CLK, RST, TX_PER, TX_ODD, TX_DI, TX_EN, TX_BUSY og TX_DO er globale variabler som vi kan få tak i.

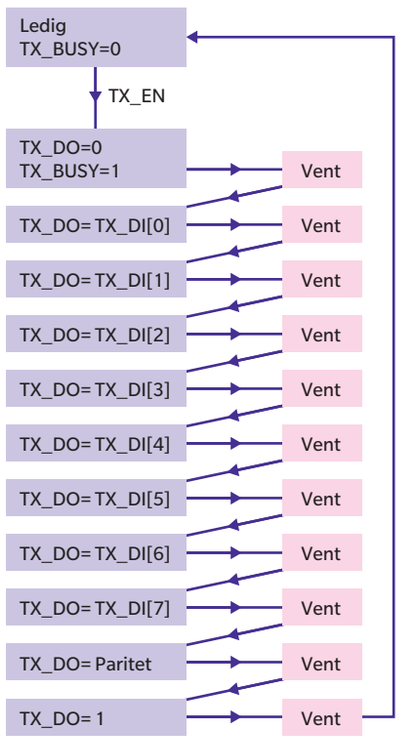
```
void main()
{
    int tx_out[11]; // [0 ... 10]
    int i, j, startbit, stopbit, par;
    startbit = 0;
    stopbit = 1;
    while (true)
    {
        if (TX_EN == 1)
        {
            // Gi beskjed til mottaker ved å sette TX_BUSY HØY
            TX_BUSY = 1;
            // Regn ut paritetsbit
            par = Parity(TX_DI, TX_ODD);
            tx_out[0] = startbit; // Fyll inn startbit
            // Fyll inn de parallelle bit som skal sendes
            // Dette kan realiseres i en multiplekser eller
            // skiftregister blokk
            tx_out[1] = TX_DI[0];
            tx_out[2] = TX_DI[1];
```

```
            tx_out[3] = TX_DI[2];
            tx_out[4] = TX_DI[3];
            tx_out[5] = TX_DI[4];
            tx_out[6] = TX_DI[5];
            tx_out[7] = TX_DI[6];
            tx_out[8] = TX_DI[7];
            tx_out[9] = par; // Fyll inn paritetsbit
            tx_out[10] = stopbit; // Fyll inn stoppbit
            // Send ut bitene en etter en serielt.
            // Her trengs en mod 10 teller
            // eventuelt en 4 bit teller og en komparator som
            // får telleren til å «wrappe» ved n = 10
            // for å tømme multiplekser / skiftregister
            for (int i = 0; i <= 10; i++)
            {
                TX_DO = tx_out[i]; // det i'te bit sendes
                for (int j = 0;
                    j <=Convert_to_integer(TX_PER) – 1; j++)
                    // vent(TX_PER - 1) klokkeperioder
                    // Her trengs en 16 bit teller. Vi må da
                    // også ha en komparator for å få
                    // telleren til å «wrappe» ved(TX_PER - 1)
                    Wait(CLK); // vent en klokkeperiode.
                                // OBS! systemavhengig
            }
            TX_BUSY = 0; // Sending er over
        } // av if
    } // av while
}   // av main

    // Bestemmer paritet
int Parity(int DI[], int ODD)
{
    int i, par, even;
    int svar = 0;
    even = 1; // Starter med å anta jevn paritet
              // Bestem paritet opp til og med i'te bit
    for (int i = 0; i <= 7; i++)
    {
        if (DI[i] == 1)
        {
            if (even == 1) even = 0;
```

```
            else even = 1;
        }
        else
        {
            if (even == 1) even = 1;
            else even = 0;
        }
    }
    if (ODD == 1) // ODD bestemmer paritet i paritetsbit
    {
        if (even == 1) svar = 0;
        else svar = 1;
    }
    else
    {
        if (even == 1) svar = 1;
        else svar = 0;
    }
    return svar;
}

// Denne funksjonen trenger vi kun når vi driver på i
// høynivåspråk
int Convert_to_integer(int PER[])
{
    int svar = 0;
    int i, j, k;
    k = 1;
    j = 0;
    for (int i = 0; i <= 15; i++)
    {
        j = j + k*PER[i];
        k = 2 * k;
    }
    svar = j;
    return svar;
}
```

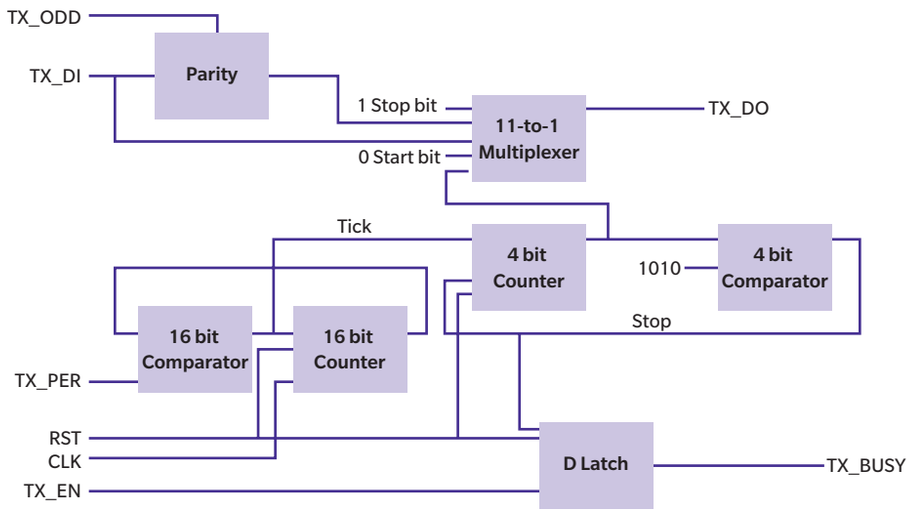
Ved å lage denne kodesnutten har vi funnet ut at vi trenger en paritetsblokk, en multiplekser, to tellere og to komparatorer. Den eneste variabelen vi ikke har brukt, er RST, men så er det heller ikke så vanlig å drive å resette sekvensiell programvare. Programvare startes jo som kjent ved å eksekvere den. Den ene telleren skal kun telle til 10, så da trenger den ikke være lenger enn 4 bit, mens den andre telleren må være 16 bit, da den styres av tallet gitt av TX_PER. Det er ikke nødvendig å lage en mod 10 teller av 4 bit telleren, da vi også trenger teller og komparator for 4 bit for å lage en 16 bit teller og tilhørende komparator. Komparatorene, i dette tilfellet, trenger heller ikke å gi oss informasjon om hvorvidt tallene er større eller mindre enn hverandre, men bare å gi oss et hint når de er like. Multiplekseren må ha minst 11 datainnganger og fire «select»-innganger for å velge. De to komparatorene må være henholdsvis 4 bit og 16 bit.

Dersom programmering eller C++ skulle være ukjent for deg, er en alternativ måte å rydde tankene på før en starter selve designet, å lage et tilstandsdiagram. Figur 12.2 viser et tilstandsdiagram for vår designutfordring.



Figur 12.2 Tilstandsdiagram for UART_TX

Figur 12.3 gir en grovskisse av UART_TX arkitekturen.



Figur 12.3 Det første blokkskjemaet for UART_TX

Dermed er vi ferdig med designoppgave 1. Det er kanskje på sin plass å fortelle litt om hvordan det er tenkt i blokkskjemaet? Øverst i venstre hjørne har vi enheten «Parity». Den styres av signalet TX_ODD, og pariteten baseres på innholdet i signalene (data) TX_DI som skal sendes. I blokkskjemaet kan strekene representere henholdsvis 1, 4, 8 og 16 linjer. «11-to-1 Multiplexer» er den laveste inngangen satt til 0 (startbit), deretter følger TX_DI signalene, paritetsbit og så til slutt et stoppbit lik 1. Så har vi to sammenhørende blokker med komparator og teller. I de til venstre får «16 bit Comparator» signal (tall) fra TX_PER, og til høyre for den er det en «16 bit Counter». Når telleren blir lik TX_PER-1, nullstilles telleren med et signal Tick, og samtidig økes «4 bit Counter» telleren. Den sistnevnte telleren vil «wrappe» når den blir 10 (1010 binært) som er hardkodet til «4 bit Comparator». «4 bit Counter» telleren sørger for «select» signal til «11-to-1 Multiplekser», slik at den suksessivt sender ut data til TX_DO. Nederst til høyre har vi en liten «D latch» med muligens litt tilbehør som sørger for at TX_BUSY blir HØY når TX_EN slår til, og LAV igjen når sending er slutt.

Da har vi basisarkitekturen på plass. Designoppgave 2 utfordrer oss til å realisere arkitekturen i figur 12.3 i vipper og porter i Multisim og deretter se om det virker. Hva har du lyst til å starte med å lage? Det er naturlig å tenke seg at her kan en gå hierarkisk til verks. Har en først laget en 4 bit komparator, så er det grunn til å tro at den kan brukes som en byggestein i en 16 bit komparator. Det samme gjelder nok for

de to tellerne også. Jeg foreslår, i prokrastineringens ånd, at vi starter med det enkleste først. I tillegg kan det være greit å begynne litt strukturert og ta de kombinatorisk logiske funksjonene først og så den sekvensielle logikken etterpå. Paritetssjekkeren, multiplekseren og komparatorene er kombinatorisk logiske funksjoner, mens D-låsen og tellerne er sekvensiell logikk.

La oss begynne med paritetssjekkeren. La oss starte enkelt med bare å sjekke paritet for to bit, $D[0]$ og $D[1]$. Hva kreves av en sannhetstabell som finner paritet?

Tabell 12.1 Sannhetstabell for paritet

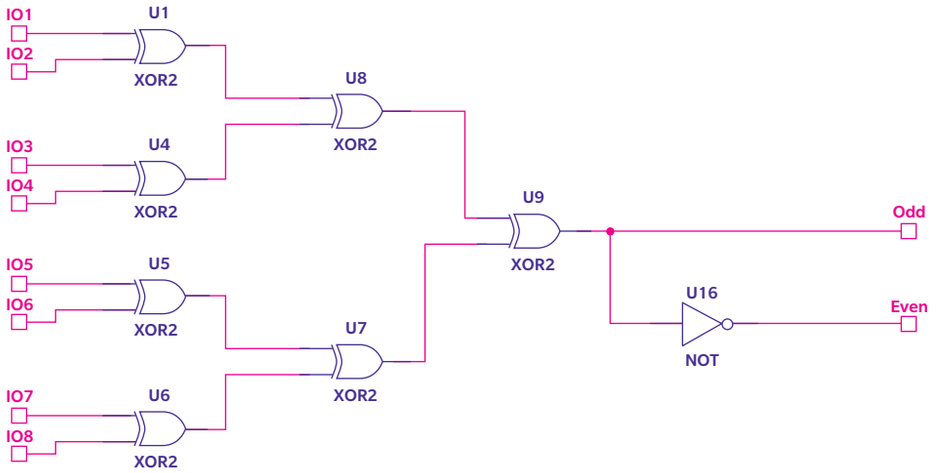
$D[0]$	$D[1]$	F
0	0	0
0	1	1
1	0	1
1	1	0

Tabell 12.1 har de ønskede egenskaper. Ved odde paritet har en funksjonsverdien 1, og ved jevn paritet funksjonsverdien 0. Denne sannhetstabellen har vi sett før. Det er sannhetstabellen for XOR.

Tabell 12.2 Sannhetstabell for XOR

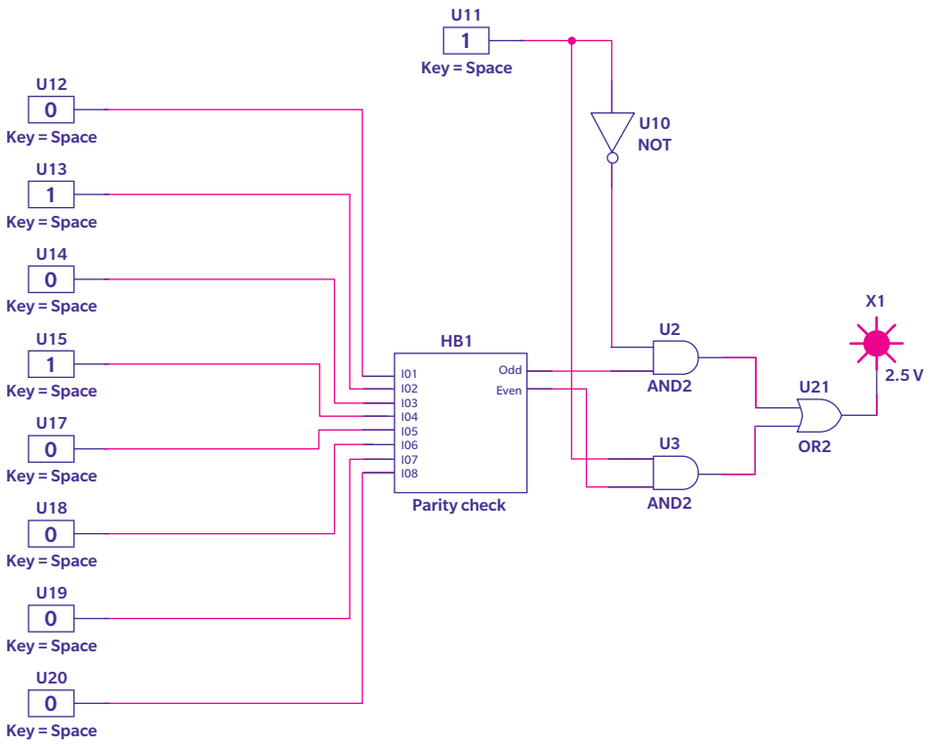
$D[0]$	$D[1]$	$D[0] \oplus D[1] = \overline{D[0]}D[1] + D[0]\overline{D[1]}$
0	0	0
0	1	1
1	0	1
1	1	0

Vi kan altså bruke en XOR-port for å sjekke paritet for to bit. Dersom vi ønsker å sjekke paritet for henholdsvis 4 og 8 bit, er det bare å stable sammen XOR-porter enten i en trekant- eller en vertikal struktur. I figur 12.4 ser du en modell av en 8 bit paritetssjekker laget med simuleringsverktøyet Multisim. Det verktøyet kan virke overveldende med alle sine knapper ved første gangs bruk. Bare sørg for å ha en opplyst venn i nærheten når du går i gang. Jeg fikk hjelp av Mathias.



Figur 12.4 Første utgave av 8 bit paritetssjekker

Stopp en hal! Vi har jo glemt at vi skal styre om vi skal ha odde eller jevn paritet med TX_ODD. Det må legges til litt logikk som gjør at den rette utgangen, enten Odd eller Even i kretsen i figur 12.4, blir valgt. Etter å ha vært innom ekstremsporten tenkning en stund, materialiserer et forslag seg, som vist i figur 12.5.

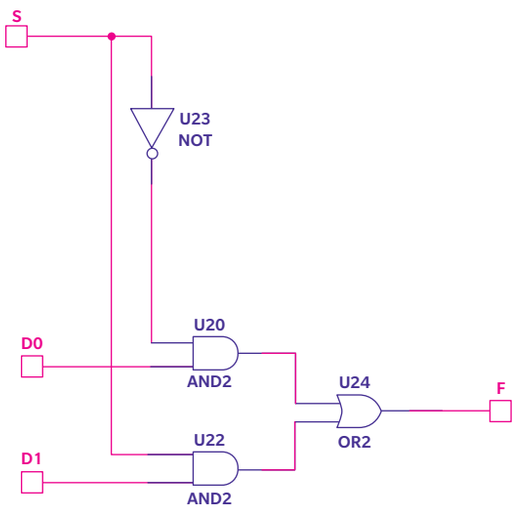


Figur 12.5 Andre utgave av 8 bit paritetssjekker

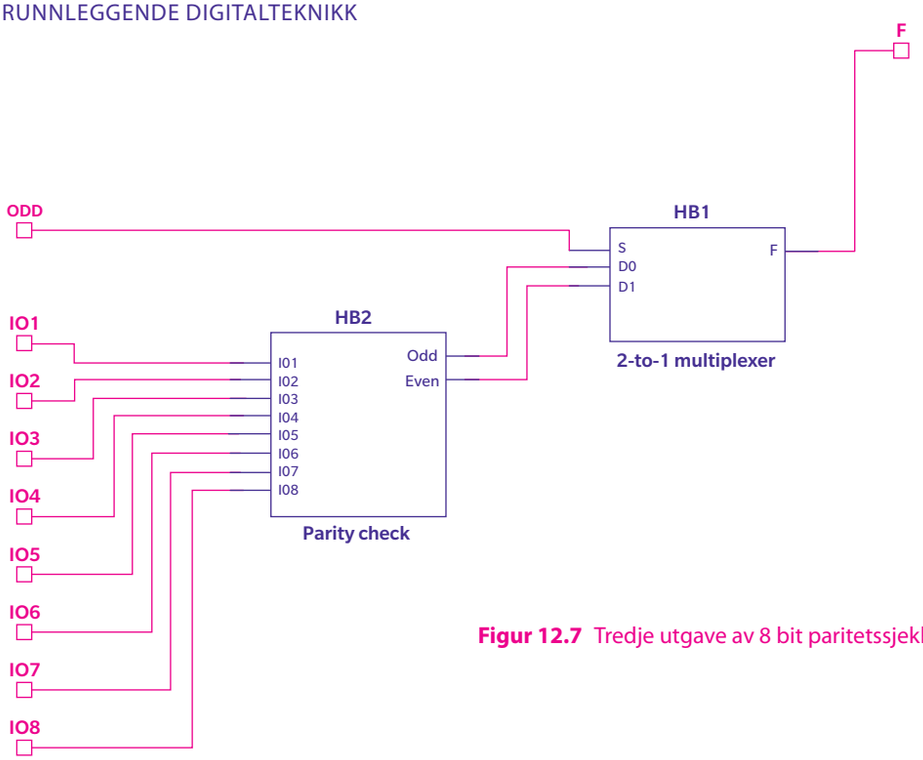
I figur 12.5 er det som ble vist i figur 12.4, samlet i en hierarkisk blokk kalt «Parity check». Her må det rettes en kraftig advarsel. Som du ser i figur 12.5, er det skrevet HB1 over «Parity check». Det betyr at blokken er laget med «Replace by hierarchical block». Det må unngås å bruke «Replace by subcircuit», for da kan ikke blokken brukes i andre design enn det du holder på med. Du må bruke «Replace by hierarchical block» som tillater gjenbruk. Da vil du få HB? over blokken. Hvor langt jeg var kommet i UART designet før jeg fant ut dette, ønsker ikke du å vite. Min eneste trøst var at det er mye god læring i repetisjon.

Blokken har innganger for databit og utganger for «Odd» og «Even» paritet. «Odd» og «Even» er koblet til noe kombinatorisk logikk. Valget av om en bruker «Odd» eller «Even», foretas ved å sette et bit i U11. U11 er vår TX_ODD. Ved å telle antallet 1-ere i data som fores til IO1 til IO8, ser vi i dette tilfellet at jevn paritet leveres til «Parity check». Styringsbit i U11 (TX_ODD) er satt til 1, som ifølge designoppgaven (spesifikasjonen) betyr at vi skal sende odde paritet, og det betyr at paritetsbit må være1. Byte vi skal sende pluss paritetsbit skal altså til sammen ha odde paritet i denne omgang. Vi ser til vår glede at lampen X2 lyser, noe som indikerer at det som kommer ut av U21 er 1. I figur 12.5 kjører Multisim en simulering, og en kan derfor endre på innkommende bitverdier ved enten å bruke mellomromstasten eller å klikke på en verdi og «toggle» den mellom 0 og1.

Dersom vi ser litt nærmere på den logikken vi har utenfor «Parity check», så ser vi at den velger enten «Odd» eller «Even» basert på verdien i U11 (TX_ODD). Vi har faktisk laget oss en liten multiplekser med to datainnganger, «Odd» og «Even», og en «select» inngang U11(TX_ODD). I figur 12.6 har vi laget en liten hierarkisk blokk av 2-til-1 multiplekseren også.



Figur 12.6 2-til-1 multiplekser

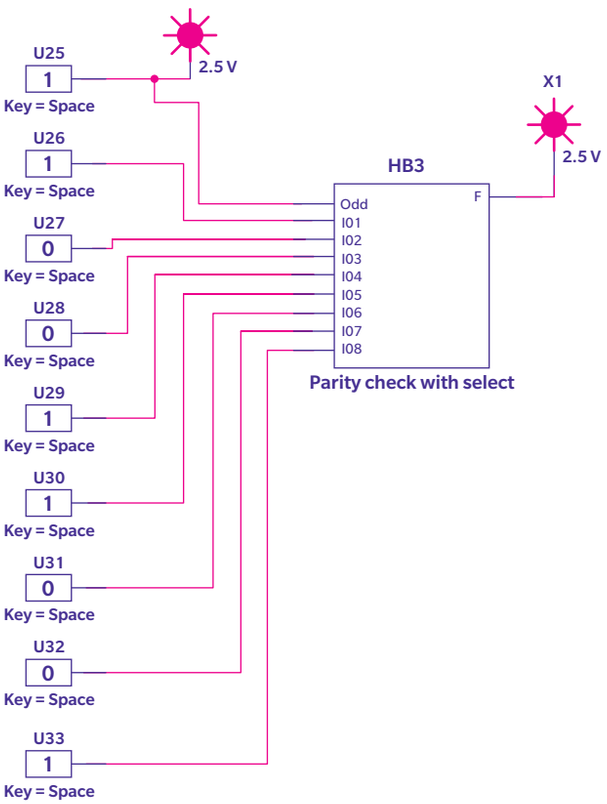


Figur 12.7 Tredje utgave av 8 bit paritetssjekker

Denne blokken bruker vi så sammen med den opprinnelige paritetssjekkeren og får kretsen som vist i figur 12.7.

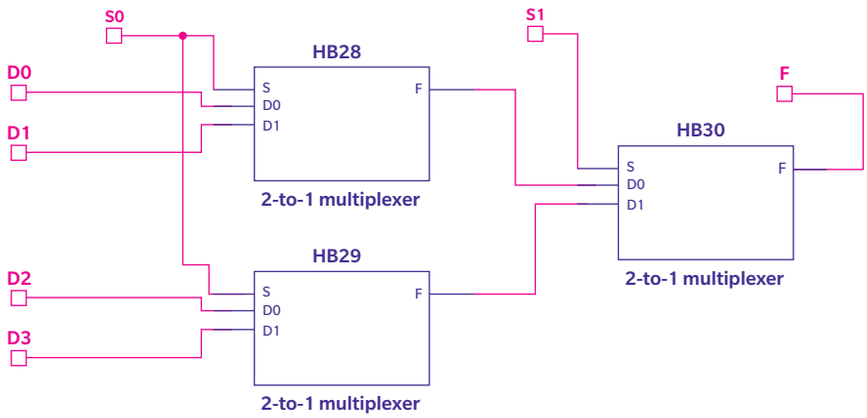
Da gjenstår det bare å smelte de to blokkene i figur 12.7 sammen til én blokk, og vi har den endelige utgaven av vår paritetssjekker i figur 12.8. Vi utsetter den endelige utgaven for enda en test, og vi ser til stor tilfredshet at den fungerer i henhold til våre ønsker.

For en resultatorientert person av «happy go lucky»-typen kan denne hierarkiske tilnærmingsmåten virke omstendelig, tungvint og treg. Det er allikevel mange gode grunner til å velge en strukturert fremgangsmåte. En kan bygge opp basisblokker som lett kan utvides til store strukturer ved hjelp av kopier og lim, det hjelper en med å holde oversikten i store, komplekse systemer, og ikke minst kan en teste og feilsøke underveis før det hele blir fullstendig uhåndterlig.



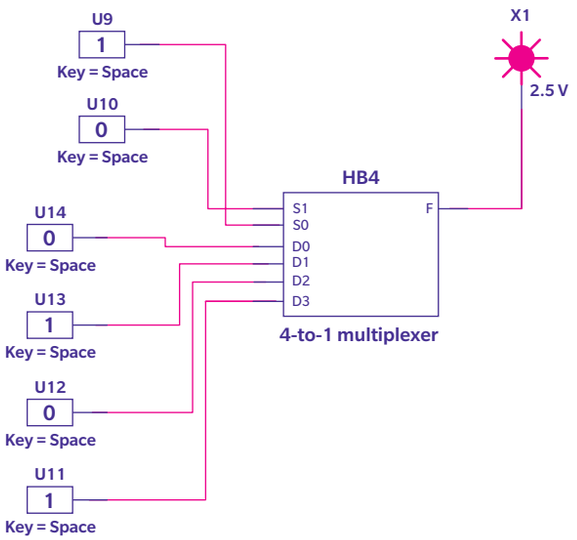
Figur 12.8 Den endelige utgaven av 8 bit paritetssjekker med ODD som «select»

Som et umiddelbart eksempel kan vi se på det neste vi skal lage, nemlig multiplekseren. Vi har jo allerede laget en 2-til-1 multiplekser. Kan vi trekke veksler på det designet når vi skal lage en 11-til-1 multiplekser? Absolutt! Det er bare å gå hierarkisk til verks. Som vist i figur 12.9, kan en 4-til-1 multiplekser bygges ved å sette sammen to 2-til-1 multipleksere.

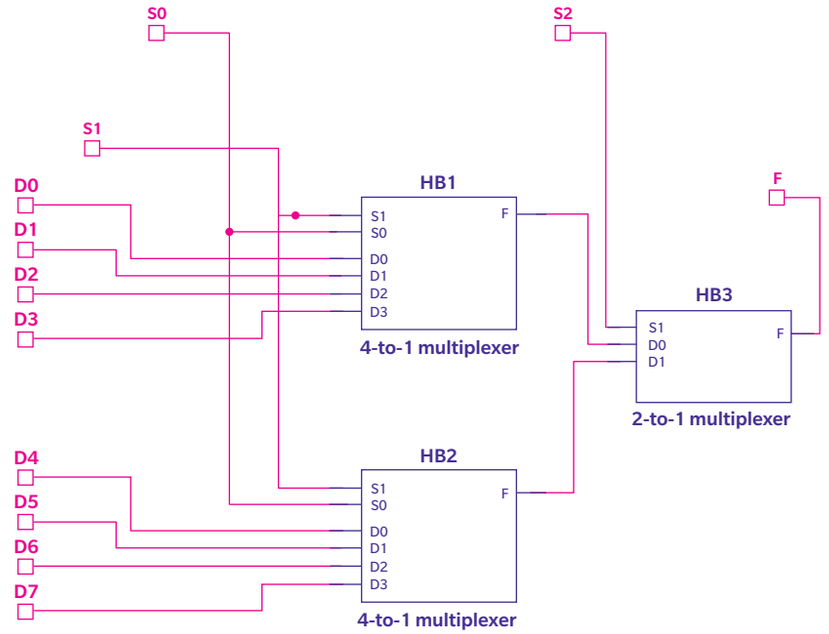


Figur 12.9 4-til-1 multiplekser bygd av tre 2-til-1 multipleksere

Bare for å være sikker på at vårt forslag til hierarkisk struktur virker som vi håper, er det fornuftig å teste 4-til-1 multiplekseren slik som vist i figur 12.10. Du ser at når S0 er 1, så velges D1 slik som den skal. Ved å prøve alle andre mulige kombinasjoner, vil du se at denne kretsen gjør susen.

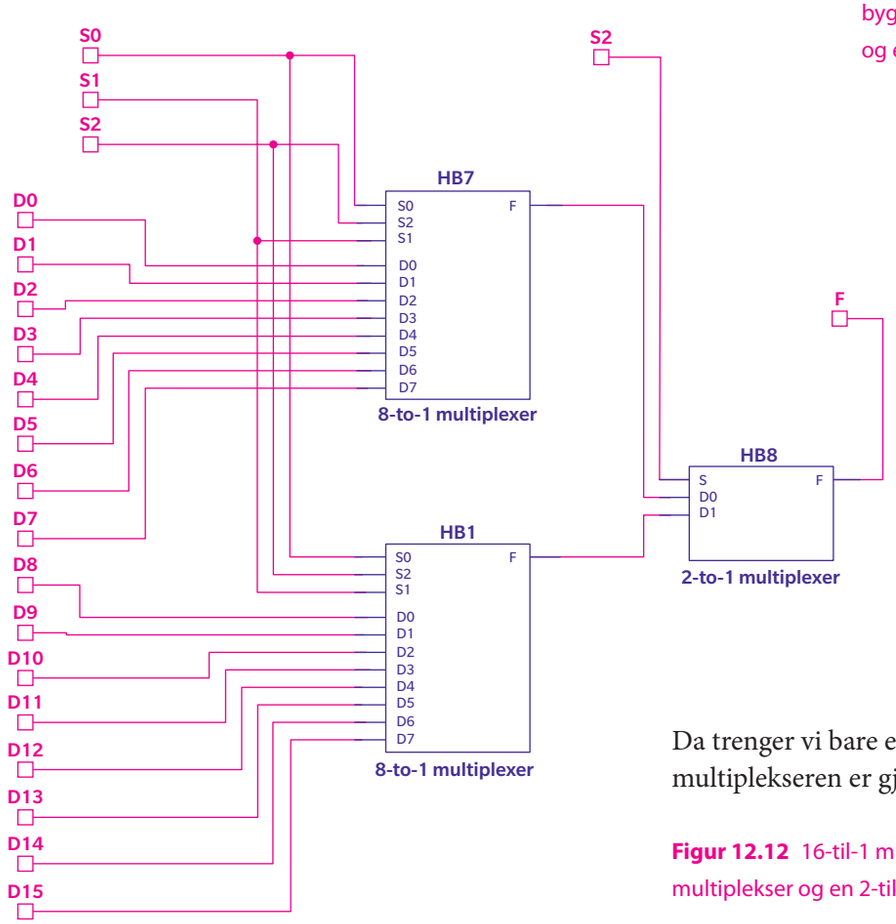


Figur 12.10 Testing av 4-til-1 multiplekser



Og sånn går nå dagene. I figur 12.11 har vi doblet igjen for å få en 8-til-1 multiplekser. Med hierarkisk design er vi rimelig trygge på at den vil virke i henhold til spesifikasjonen, og det er nærmest unødvendig å teste den.

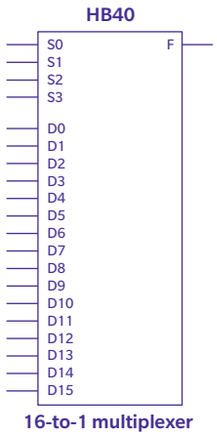
Figur 12.11 8-til-1 multiplekser bygd av to 4-til-1 multiplekser og en 2-til-1 multiplekser



Da trenger vi bare en runde til før vi er i mål. 16-til-1 multiplekseren er gjengitt i all sin prakt i figur 12.12.

Figur 12.12 16-til-1 multiplekser bygd av to 8-til-1 multiplekser og en 2-til-1 multiplekser

Da er det bare å ramme det hele inn, se figur 12.13, og vår 16-til-1 multiplekser er klar til bruk i UART_TX.



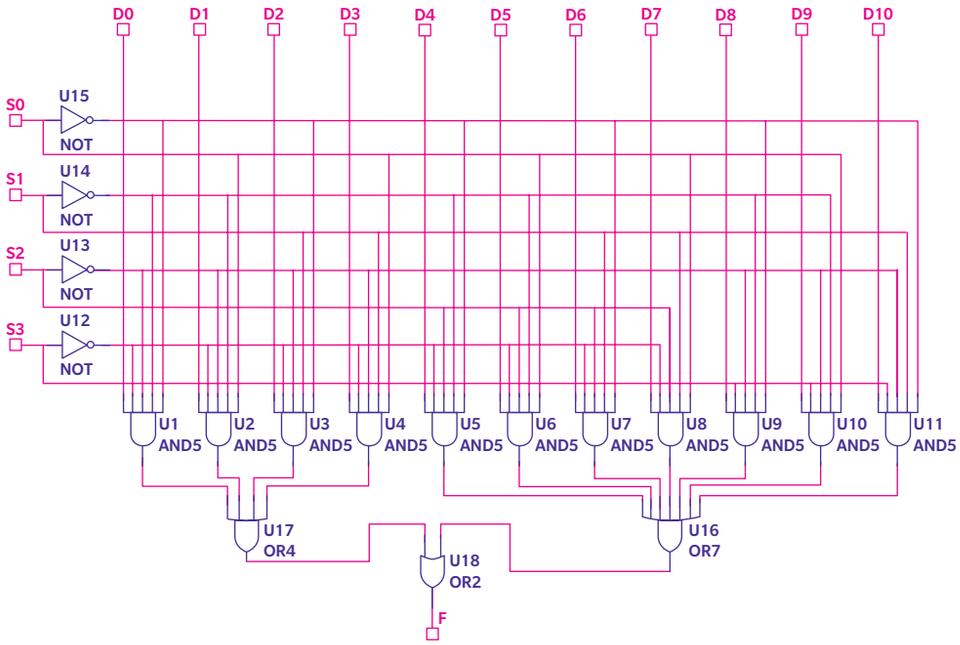
Figur 12.13 16-til-1 multiplekser

Multisims første forslag til pinnekonfigurasjon var litt hulter til bulter og uestetisk, men ved å klikke med høyre mustast over symbolet får en blant mange alternativer muligheten til «Edit symbol/title block» som gir deg sjansen til å flytte pinnene rundt omkring der du synes de passer best. En god regel er å ha inngangssignaler til venstre og utgangssignaler til høyre.

Nå har vi laget en stor multiplekser som vi vet virker, og det er kanskje den største sjarmen med å velge en hierarkisk angrepsmetode. En bruker basalblokker som er lette å teste, og setter dem sammen på en kontrollert måte. Dermed er resultatet rimelig forutsigbart så lenge en ikke har foretatt logiske kortslutninger underveis.

Det er en pris å betale. I vårt tilfelle trengte vi en 11-til-1 multiplekser, men fordoblingen har gitt oss en 16-til-1 multiplekser. Den er riktignok litt for stor for vårt formål, men det er jo bare å sørge for at «select» aldri blir større enn 10, og jorde $D11$ til $D15$.

Er det mulig å lage en 11-til-1 multiplekser? Ja, det var det første jeg gjorde, og den er vist i figur 12.14. Den skapte mye frustrasjon og #@!! ved unnfangelsen, og Svein var lite fornøyd med en blokk med tilnærmet null gjenbruksverdi.



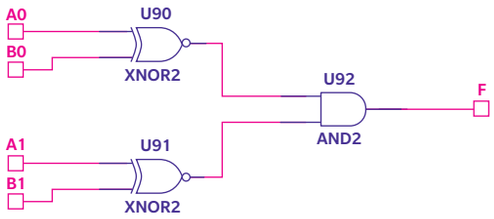
Figur 12.14 11-til-1 multiplekser til skrekk og advarsel

Det går jo så det griner. Kun en kombinatorisk logisk krets igjen, nemlig komparatoren. Husker du vår matematikervenn som mente at barn burde studere mengdelære før tall? Om to tall var like, kunne en jo bare sjekke med å holde fingrene mot hverandre. Samme metode kan vi bruke med å sjekke en finger, i vårt tilfelle et binært tall, av gangen. La oss starte med å sammenligne to bit. Sannhetstabellen vil bli som i tabell 12.3.

Tabell 12.3 Sjekk for likhet av to bit

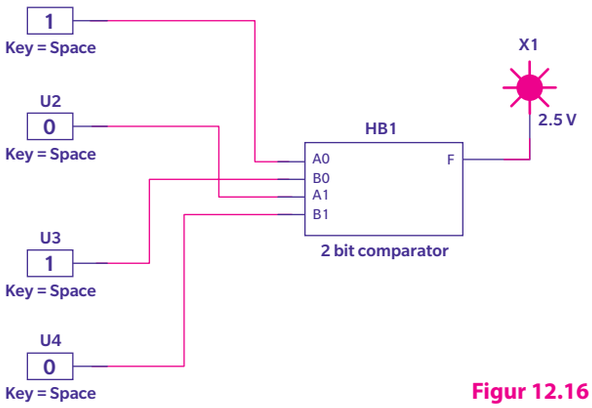
$A[0]$	$B[0]$	$F = \overline{A[0] \oplus B[0]}$
0	0	1
0	1	0
1	0	0
1	1	1

Negativ XOR gjør susen. Skulle vi være så heldige å ha flere fingre (bit), er det bare å bruke OG-funksjonen. I figur 12.15 ser du en komparator for tall med lengde 2 bit.

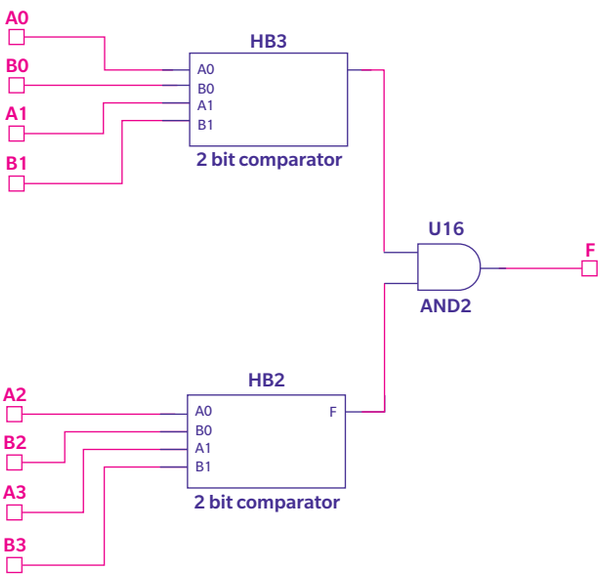


Figur 12.15 2 bit komparator

Det er kanskje på tide med en liten test selv om logikken er enkel. Selv små tuer kan velte store lass. I figur 12.16 ser du testen av vår 2 bit komparator. Tallene er like, og F lyser som den skal.



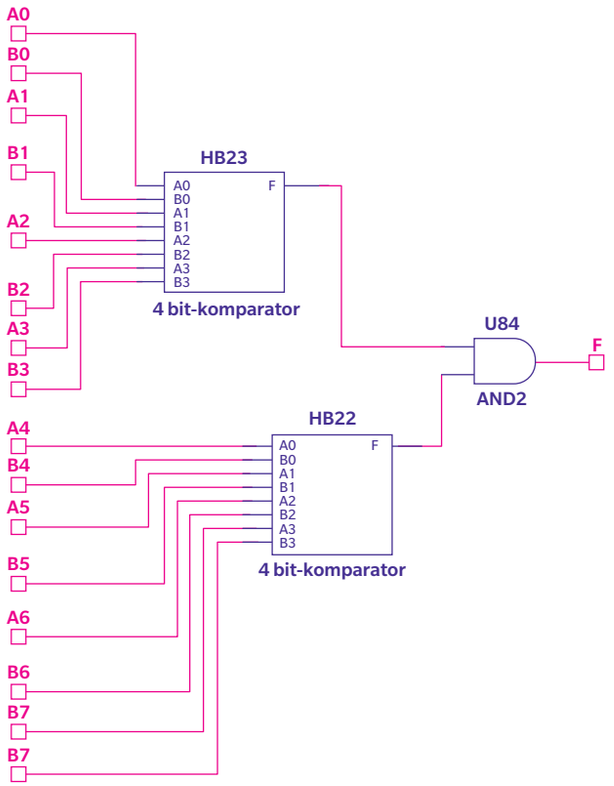
Figur 12.16 Test av 2 bit komparator



Da er byggesteinen på plass, og det er bare å hengi seg til fordoblingens edle kunst. Figur 12.17 viser en 4 bit komparator.

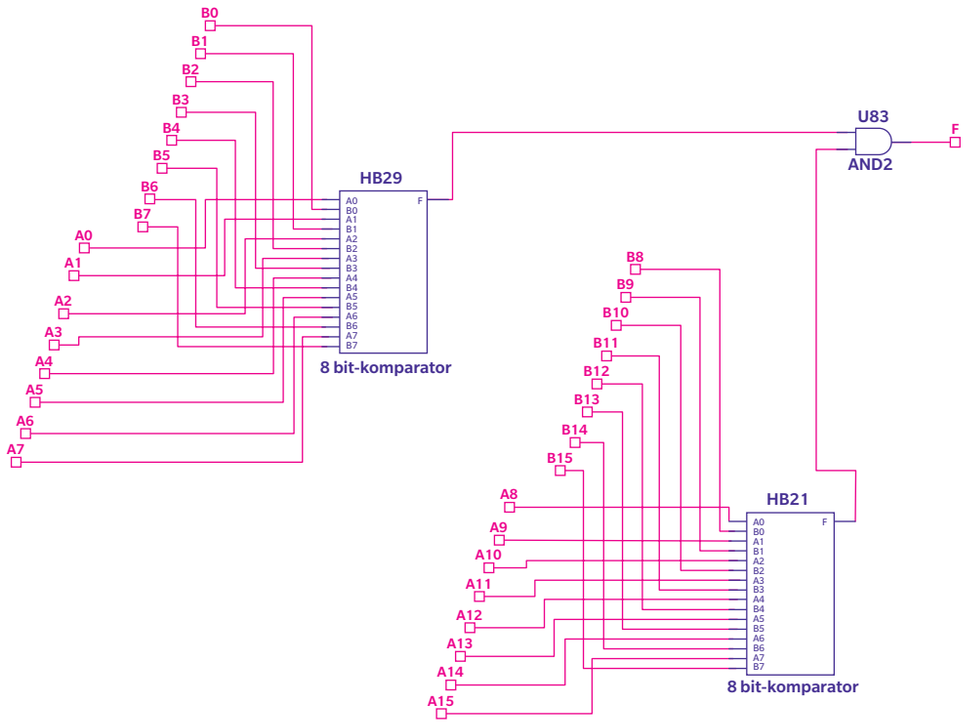
Figur 12.17 4 bit komparator

Hva tror du kommer nå? Riktig, i neste figur har vi en 8 bit komparator.

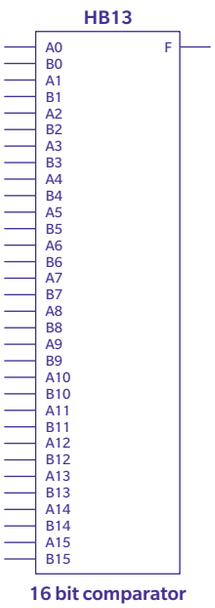


Figur 12.18 8 bit komparator

Det baler på seg, og i figur 12.19 som viser en 16 bit komparator, har det blitt så mange innganger at inntrykket er blitt litt rotete. Tro meg, det virker. Jeg har trykket på alle terminalene, og Multisim kvadrater viser at det er riktig koblet.



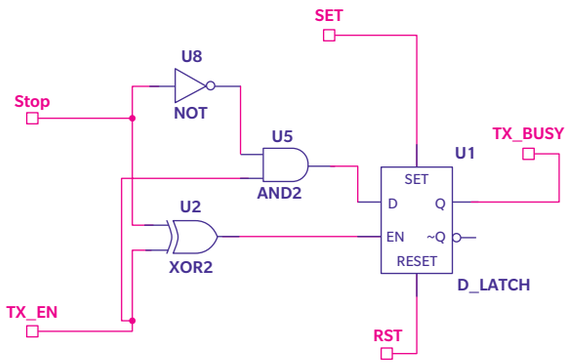
Figur 12.19 16 bit komparator



Til slutt er det bare å putte komparatoren i boks som vist i figur 12.20. Til vår glede ser vi at denne gangen ble det rimelig ryddig selv om det så en smule kaotisk ut i forrige figur.

Figur 12.20 16 bit komparator

Dermed er vi ferdig med blokkene av de kombinatorisk logiske kretsene. Da er det bare å hive seg over den sekvensielle logikken. Det enkleste er nok å starte med D-låsen. Den skal «huske» at TX_EN signalpulsen setter den til 1 og dermed TX_BUSY til HØY. TX_BUSY skal være HØY så lenge sending pågår, og settes til LAV når stopbit er ferdigsendt. Det vil skje når 4 bit komparatoren som er hardkodet til 10, merker at 4 bit telleren er kommet til 10. Da sender 4 bit komparatoren et Stop til D-låsen som setter TX_BUSY til LAV. Det trengs altså en D-lås med litt garnityr som vist i figur 12.21.

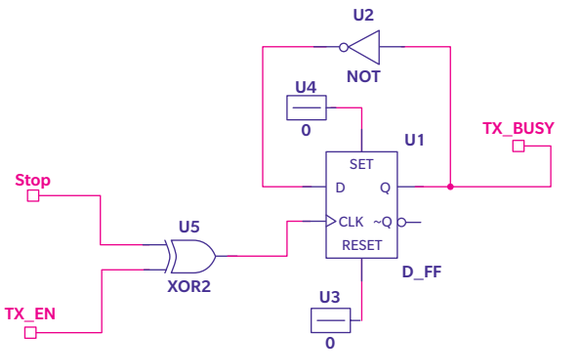


Figur 12.21 Vår utvidede D-lås

Av figur 12.21 ser vi at TX_EN eller Stop «enabler» D_LATCH på EN inngangen. I tillegg rutes TX_EN og Stop til D inngangen via en OG-port. Ved start settes TX_EN HØY et lite øyeblikk, og Stop er da LAV. Siden Stop inverteres, settes D HØY, og dermed blir også TX_BUSY HØY. Når Stop blir HØY, skal en stoppe. TX_EN er da LAV. Stop og TX_EN setter da D til LAV, og dermed blir også TX_BUSY LAV. Alt skulle dermed være i sin skjønneste orden.

Det må innrømmes at Svein ikke ble særlig glad da han så dette designet. «Det står jo i spesifikasjonen (oppgaveteksten) at det skal brukes vipper.» Det hjalp lite med argumenter av typen «det virker jo». Her trengs det en ny runde med Multisim og en blokk basert på D-vippe isteden. D-vippe er så basal at den kan vi bare hente fra det ferdige biblioteket. Skulle du mot formodning ikke huske dens virkemåte, er det bare å lese litt rundt tabellen 7.5 og figuren 7.28 i kapittelet «Tingenes tilstand».

Hvordan skal en få en vippe til å vippe mellom to tilstander? Når TX_EN går HØY, skal TX_BUSY bli HØY, og når stopbit er ferdigsendt, skal TX_BUSY gå LAV. Etter litt prøving og feiling – husk å jorde ubrukte innganger, ellers flagrer de og gir rare resultater – i Multisim samt litt tenkning materialiserer følgende krets, figur 12.22, seg.

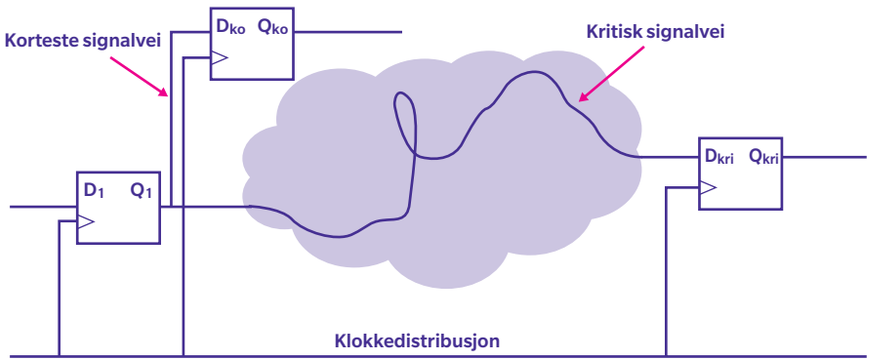


Figur 12.22 Styringslogikk for TX_BUSY med D-vippe

Her ser du at vippingen er løst med en tilbakekobling fra Q til D hvor signalet fra Q blir invertert. Når TX_EN blir HØY, «enables» CLK. D som allerede er 1 på grunn av tilbakekoblingen av Q som er 0 før «enablingen», vil når CLK «enables», sørge for at Q blir 1 som ønsket og selv bli lik 0 på grunn av invertering. Når sending er ferdig og Stop slår til, vil Q gå tilbake til 0. Dermed har TX_BUSY fått den ønskede oppførsel.

«Du er på riktig vei, men du har gjort en kardinalfeil, nærmest en stor synd mot det synkrone evangeliet. Du skal ikke bruke D-vippens CLK inngang til noe annet enn klokkesignalet.» Jeg ser spørrende på min mentor Svein. Han forklarer at klokkedistribusjon er alfa og omega i digitalt design. Det er viktig å få distribuert klokken så langt inn i grunnelementene (blokkene) som mulig. Da unngår en problemer senere med at ting går i utakt når systemene blir store.

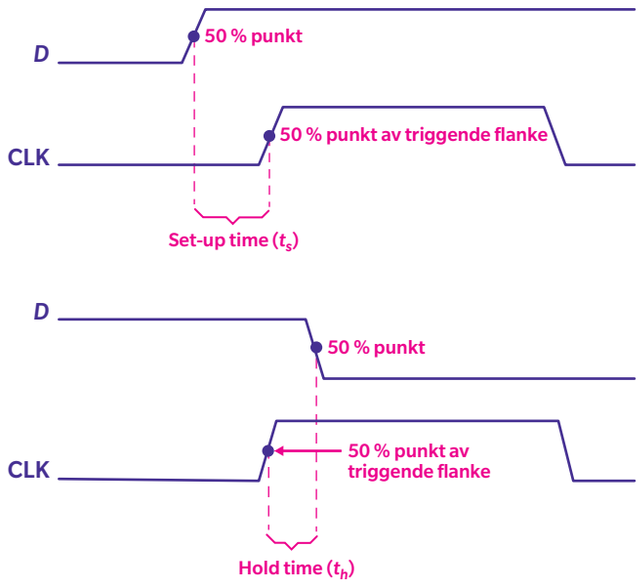
Svein finner frem et A4 papir og tegner figuren som er gjengitt i figur 12.23.



Figur 12.23 Et digitalt synkront design

«Her ser du et digitalt synkront design hvor klokken er godt distribuert. Inne i skyen er det en såkalt kritisk signalvei som er den lengste stien i designet.» «Hvor lang tidsforsinkelse kan den ha før det går galt?». Jeg svarer spontant: «En periode,» «Aldeles korriktig!» «Voksenopplæringen går jo så det suser.»

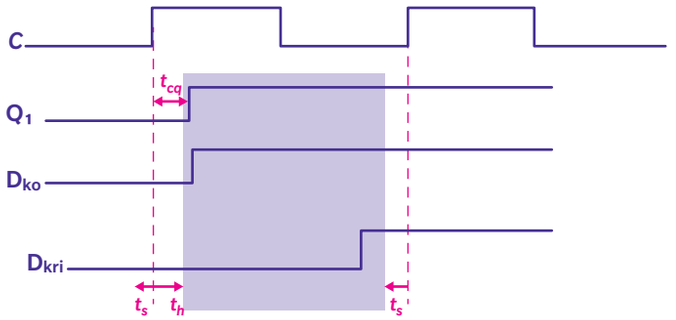
«Husker du det med 'set-up time' og 'hold time'?» «Ja, det ble gjennomgått i kapittelet 'Tingenes tilstand', og figuren 7.37 kan jeg gjerne reprodusere. «'Set-up time' har en da D trenger å stabilisere seg før klokken trigger den, og 'hold time' for at klokken skal være stabil før D endrer seg.»



Figur 12.24 «Set-up time» og «hold time»

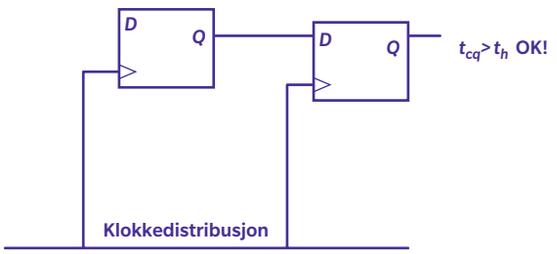
Svein ser på meg. «Det er en annen forsinkelse du også må tenke på. For det første at det tar tid for et signal å gå gjennom en vippe fra D til Q. Denne forsinkelsen betegnes med t_{cq}.»

Svein snur arket og tegner en nytt diagram, figur 12.25, og forklarer: «Øverst ser du klokken og nedenfor Q_1 for første vippe, D_{ko} for vippen med 'korteste signalvei' og D_{kri} for vippen med 'kritisk signalvei'. D_{kri} må være satt minst t_s før neste klokkeperiode, som er tilnærmet det samme som du svarte for litt siden. Det vil altså være 'kritisk signalvei' som bestemmer klokkehastigheten som kan brukes.»



Figur 12.25 Utfordringer i den digitale hverdag

Vi har en utfordring til. Det er «korteste signalvei» hvor to påfølgende vipper får klokken samtidig. I figur 12.25 er det vist som «korteste signalvei». Dersom signalforsinkelsen t_{cq} gjennom vippen med D_1 og Q_1 er kortere enn holdetiden t_h, vil Q_1 i første vippe og dermed D_{ko} i andre vippe i «korteste signalvei» være endret før klokken kommer, og andre vippe, med D_{ko}, vil kunne levere feil verdi videre. For at vi ikke skal få problemer med verken «kritisk signalvei» eller «korteste signalvei», må alle transisjoner skje innen det skraverte området i tid i figur 12.25. Figur 12.26 viser et eksempel på «korteste signalvei» i et lite skiftregister.

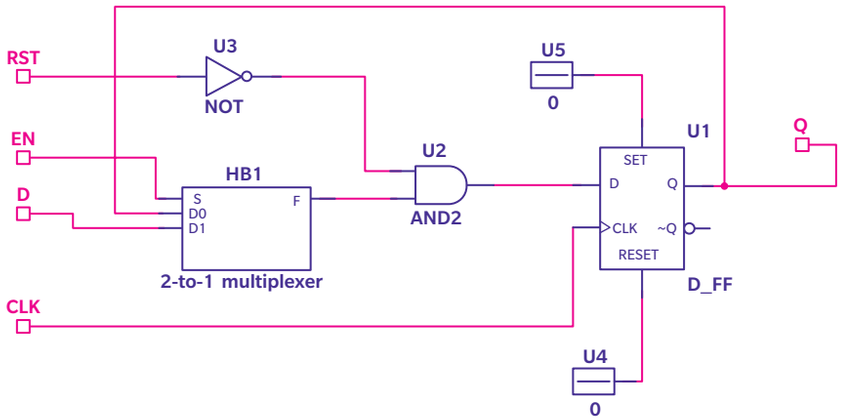


Figur 12.26 Eksempel på «korteste signalvei»

«Bortsett fra disse småproblemene, er det ingen ulemper med synkront design?». «Jo, det resulterer i at alle kretser og blokker 'rister' i takt med klokken, noe som utvikler uønsket varme og energitap. Det løses som regel med å gi muligheter for å 'enable/disable' klokken ned til blokker som ikke er i umiddelbar bruk.»

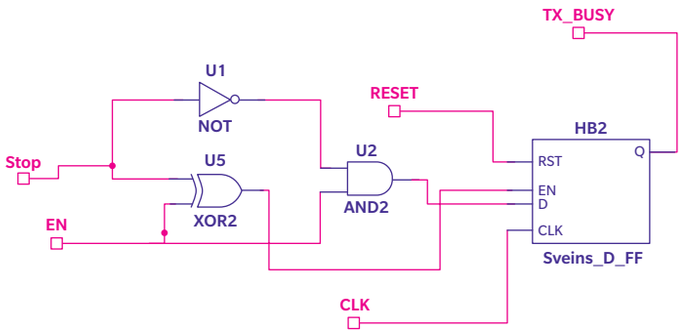
«Det får være nok belæring. La oss gå tilbake til sakens kjerne – den synkrone D-vippe.» Svein finner et ledig hjørne på A4 papiret og utvider D-vippen, gjengitt i Multisimutgave i figur 12.27, slik at den blir i henhold til kanonisk synkront design.

Klokkesignalet dras inn i vippen på CLK inngangen, og i tillegg til D inngangen er det koblet en «enable» EN inngang. Det er denne EN inngangen som så kan brukes til å «klokke» vippen med en annen frekvens enn den opprinnelige klokken.



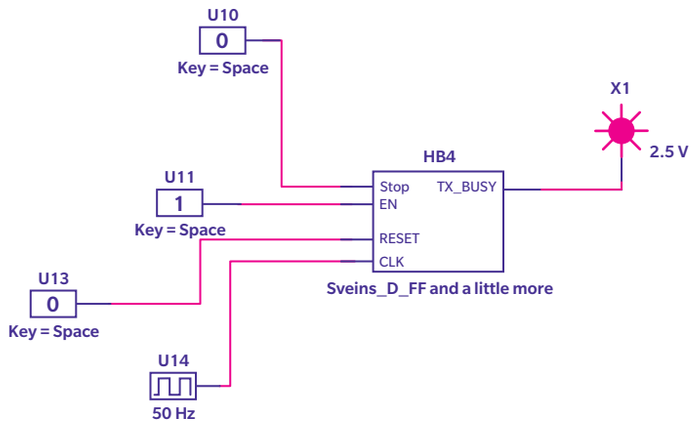
Figur 12.27 Sveins synkrone D-vippe design

Da er det bare å ta det første utkastet, figur 12.21, for styringslogikk av TX_BUSY og erstatte den opprinnelige D-vippen med Sveins synkrone utgave vist i figur 12.28. Vi er med ett blitt medlemmer i synkronklubben og må bare huske å dra denne kunnskapen med oss videre i større og mer kompliserte design.



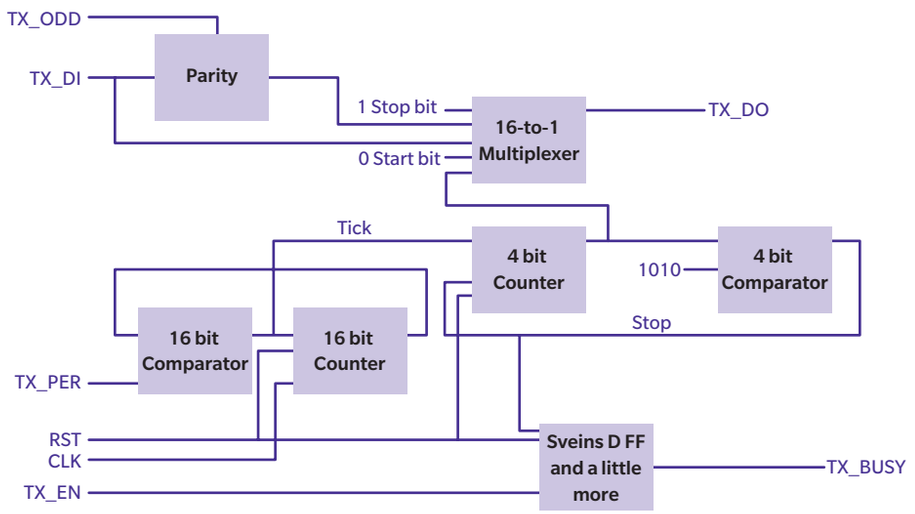
Figur 12.28 Styringslogikk for TX_BUSY med D-vippe

Endelig er vi i mål med styringslogikken, og til slutt er det bare å kapsle inn hele herligheten i en hierarkisk blokk som vist i figur 12.29, og teste at den virker.



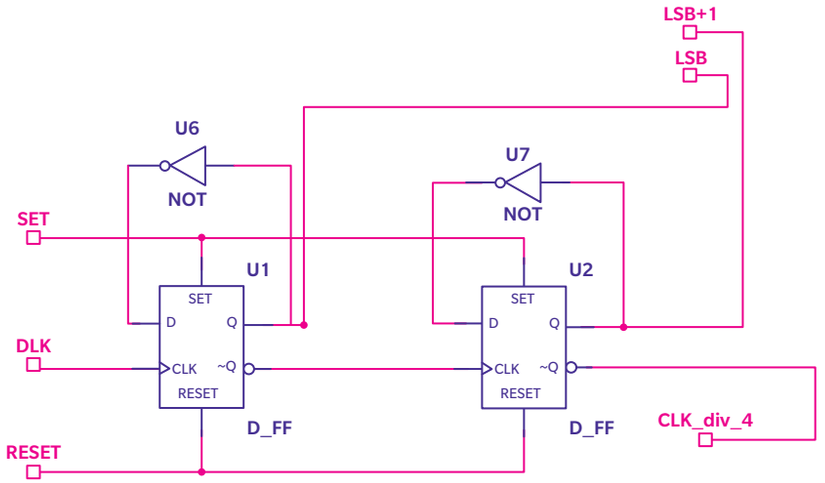
Figur 12.29 Styringslogikkblokk for TX_BUSY

Siden vi nå bruker Sveins D-vippe med litt ekstra logikk for å styre TX_BUSY og en 16-til-1 multiplekser, bør også blokkskjemaet vårt oppdateres. Det er gjort i figur 12.30.



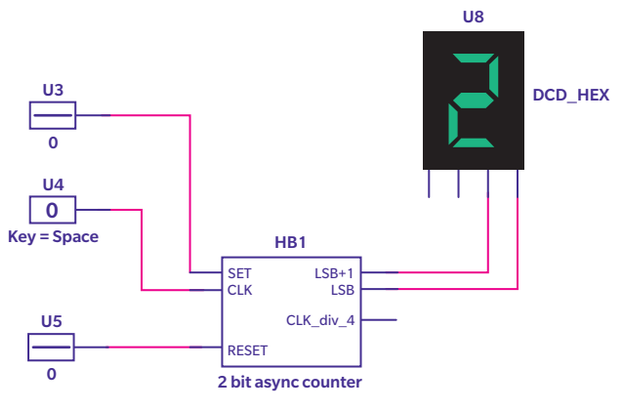
Figur 12.30 Siste utgave av blokkskjema for UART_TX?

Da gjenstår det bare å lage en ting til, nemlig tellere. I ren latskap kunne vi jo bladd tilbake til J-K-vippe telleren gitt i figuren 9.9 i kapittelet «Evig runddans» og kopiert, men la oss heller prøve å lage en ny basert på D-vipper. Vi har jo allerede observert at å tilbakekoble en D-vippe med en inverter (NOT) gjør den til en vippe som «toggler», og det kan vi utnytte til vår fordel. La oss begynne enkelt og hierarkisk med en 2 bit teller ved å koble sammen to D-vipper. Et første utkast ser du i figur 12.31.



Figur 12.31 Første forsøk på en 2 bit teller

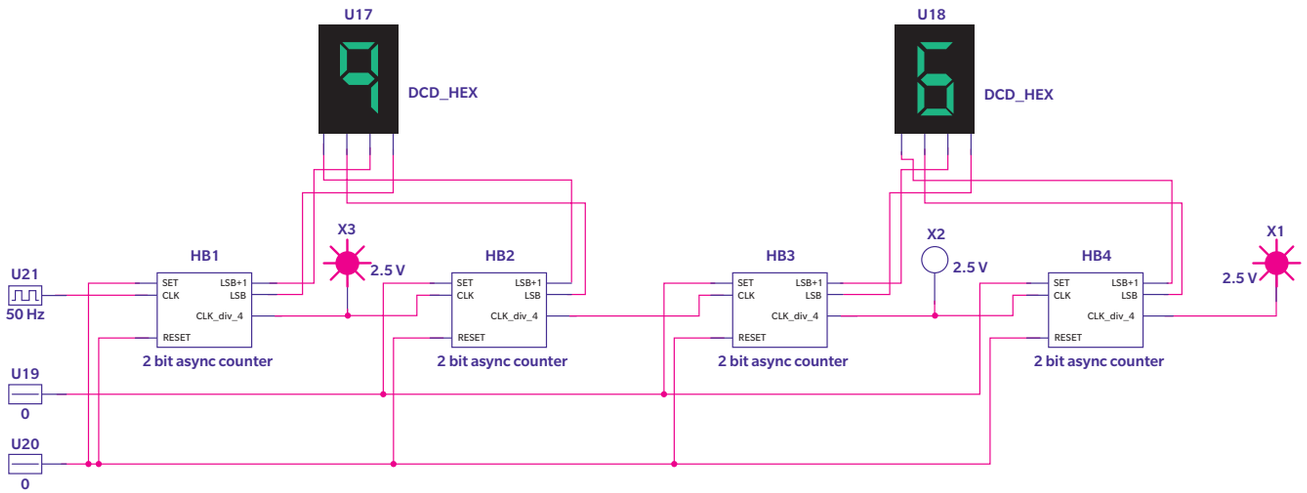
Vi ser at vi har klart å lage D-vippene slik at de «toggler» ved hjelp av tilbakekoblinger. LSB «toggles» av CLK inn på den første D-vippens klokkeinngang, og LSB+1 «toggles» ved at ~Q fra første D-vippe fores inn i den andre D-vippens klokkeinngang. Den andre vippens klokkeinngang klokkes dermed med en frekvens som er den halve av den opprinnelige klokkefrekvensen. Når 2 bit telleren «wrapper», sendes det ut signal CLK_div_4. Er vi så sikre på at det virker? Vel, det er bare å lage seg en hierarkisk blokk av det hele og teste. Figur 12.32 viser 2 bit telleren under uttesting.



Figur 12.32 Uttesting av 2 bit teller

Det er bare å hamre løs på musen med klikk og leke klokkesignal, og til vår glede oppfører telleren seg som forventet. Den starter på 0 og teller opp til 3 før den fortsetter i en evig runddans. Strålende!

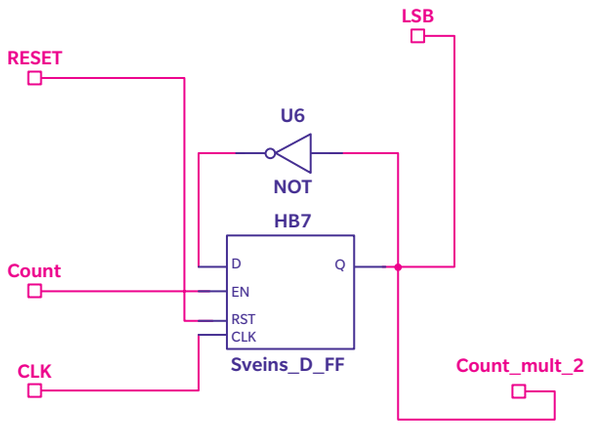
For å lage større tellere er det bare å kaskadekoble 2 bit telleren til ønsket størrelse. I figur 12.33 ser vi en 8 bit teller sammensatt av fire 2 bit tellere.



Figur 12.33 Uttesting av 8 bit teller

Det er bare ett problem. Vi har syndet mot synkronismens første bud som krever at vi skal koble alle vipper til klokken. I dette tilfellet 50 Hz kilden gitt i U21. Vi har faktisk laget en asynkron teller. For hver gang vi går gjennom en 2 bit teller i figur 12.33, får vi en forsinkelse. Riktignok i størrelsesorden nanosekunder, men blir de mange nok, får vi et problem med synkronisering. For å sitere salige Robert Noyce: «After you become reconciled to the nanosecond, computer operations are conceptually fairly simple.» Trøsten er at det er rimelig lett å reparere designet. Det er bare å bytte ut D-vippene med Sveins synkrone D-vipper, og vi er på rett kjøl igjen. Jeg sletter min asynkrone 2 bit teller fra Multisim, slik at jeg aldri skal falle for fristelsen å bruke den igjen.

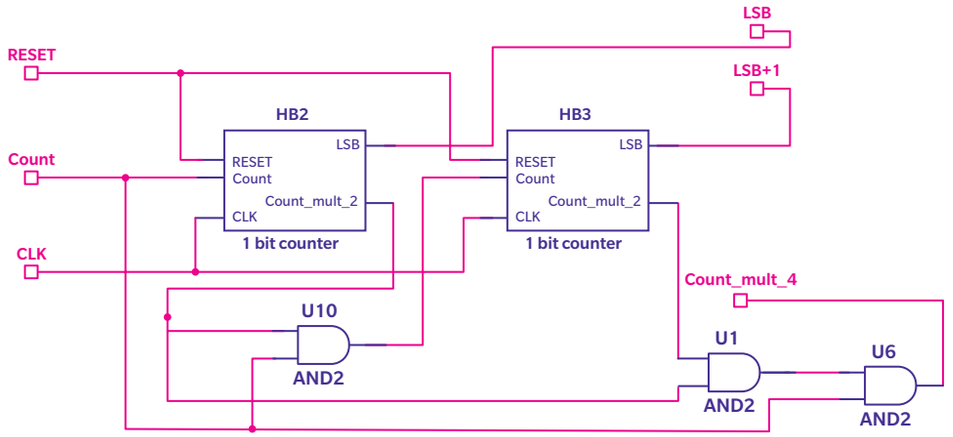
La oss gå stille, rolig og hierarkisk til verks. Det første vi utfordrer oss selv med, er en synkron 1 bit teller – figur 12.34. Den lager vi ved å tilbakekoble Q til D med en invertering (NOT) i Sveins_D_FF. Da får vi en «toggling» hver gang Sveins_D_FF blir «enablet» på 1 bit tellerens Count inngang. Tellerens verdi hentes ut på LSB, og når telleren «wrapper», sendes det en HØY puls ut på Count_mult_2.



Figur 12.34 Synkron 1 bit teller

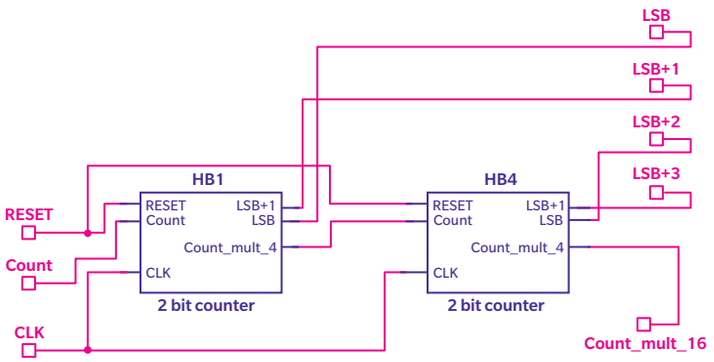
Hva er det neste naturlige steget etter at en har laget en synkron 1 bit teller? Du har sikkert allerede gjettet det. En synkron 2 bit teller. La oss starte naivt med bare å kaskadekoble to synkrone 1 bit tellere. Figur 12.35 viser den synkrone 2 bit telleren. Du kan jo se på figuren og prøve å forstå hva som skjer, basert på mottoet at et bilde sier mer enn tusen ord. Hvis det skulle bli vanskelig, kan du lese ordene som kommer nå. Den første telleren vil gi oss det minst signifikante bit LSB, mens den andre vil gi LSB+1 som i dette tilfellet er det mest signifikante bit. Vi ser at klokken distribueres inn i vippene, og dermed er vi ekte medlemmer av synkronlauget. RESET for 1 bit tellerne er samlet sammen i den synkrone 2 bit telleren. Hva sender vi så inn på Count inngangen til den synkrone 2 bit telleren? Det er to muligheter. Står denne synkrone 2 bit telleren inne i en kaskadekobling, så sendes det et HØY signal når tellerne før denne telleren «wrapper». Er denne telleren den første, settes Count til HØY for å «enable» telleren slik at den gjør det den skal gjøre, nemlig telle.

Sammenkoblingen av de synkrone 1 bit tellerne er litt mer utfordrende enn da vi laget den asynkrone telleren i figur 12.31. Vi kan ikke øke den siste 1 bit telleren før vi vet at alle tellerne tidligere i kjeden «wrapper». Hvor kan vi så hente den informasjonen fra? Vel, når Count, som er inngangen til den synkrone 2 bit telleren, er HØY og Count_mult_2, fra første synkrone 1 bit teller i den synkrone 2 bit telleren, er HØY samtidig, vet vi at alle bit før den siste synkrone 1 bit telleren i vår synkrone 2 bit teller er HØYe. Derfor ANDer vi sammen Count og Count_mult_2 og forer resultatet til Count i den siste synkrone 1 bit telleren i vår synkrone 2 bit teller. Når vår synkrone 2 bit teller «wrapper» sammen med alle eventuelle tidligere tellere, blir Count_mult_4 HØY. Som du ser av figur 12.35, er Count_mult_4 et resultat av en AND prosess som blir HØY når alle tidligere tellere «wrapper».



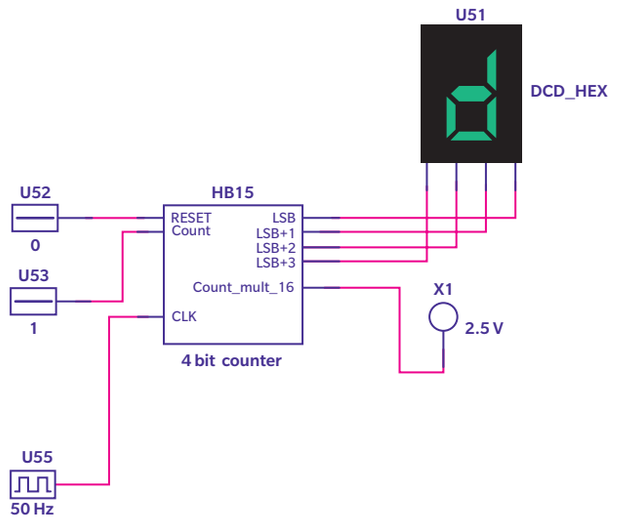
Figur 12.35 Synkron 2 bit teller

Så er det bare å ta det hele til neste nivå som er en 4 bit synkron teller. Det gledelige nå er at vi, fordi vi har arbeidet strukturert og hierarkisk, er ferdig med det som voldte hodebry. Det er bare å stable sammen av hjertens lyst, da utfordringen med «wrapping» er innbakt i den synkrone 2 bit telleren. I figur 12.36 ser vi hvordan to synkrone 2 bit tellere er slått sammen til en synkron 4 bit teller. Designet er så enkelt at ord er overflødige.



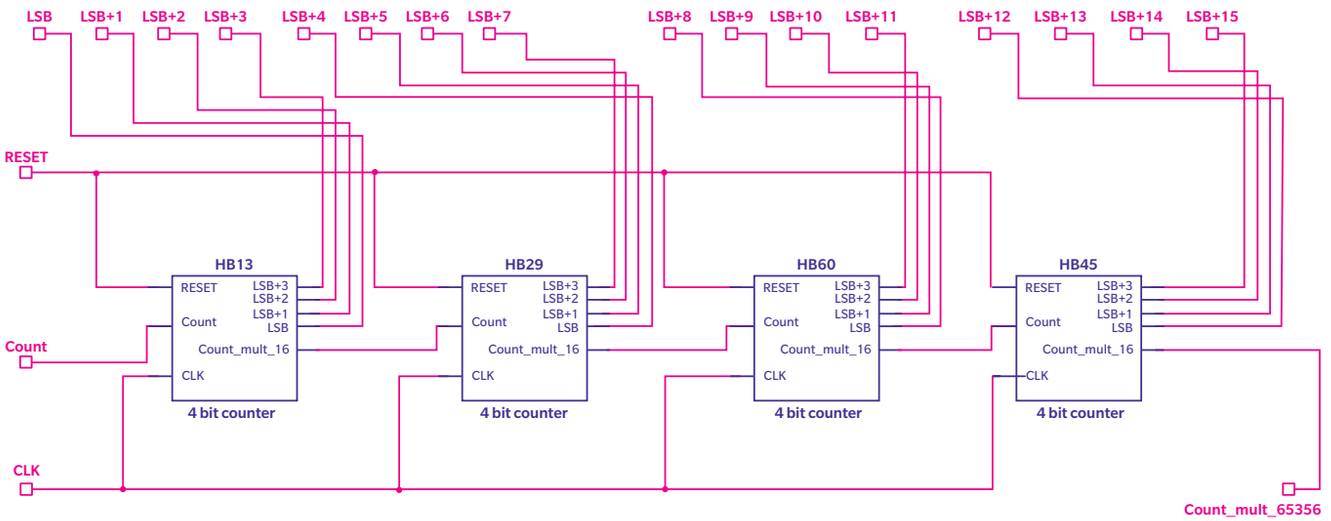
Figur 12.36 Synkron 4 bit teller

Strålende! Da har vi den første telleren vi trengte, nemlig den synkrone 4 bit telleren. Virker den? Jeg har nok underslått det faktum at testing underveis har vært en medvirkende årsak til suksessen. Figur 12.37 viser uttesting av den synkrone 4 bit telleren, og tro meg, den virker faktisk som forventet. Litt av sjarmen med simulatorbasert digitalt kretsdesign er jo å trykke på «play» knappen og sette seg tilbake og nyte synet av ting som gjør det de skal.



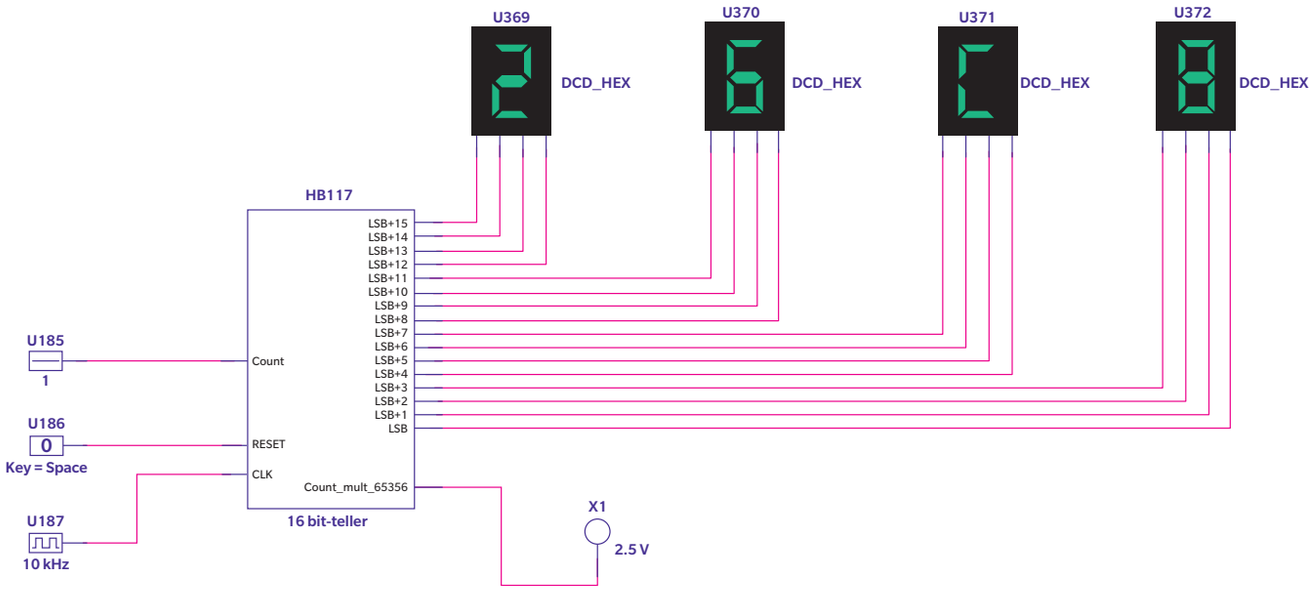
Figur 12.37 Uttesting av synkron 4 bit teller

Da gjenstår kun en blokk, og det er den synkrone 16 bit telleren. Her kunne vi gått veien om en 8 bit teller, men da vi ikke trenger den, kan vi synde litt mot fordoblingens gleder og gå direkte fra synkron 4 bit teller til synkron 16 bit teller som vist i figur 12.38.



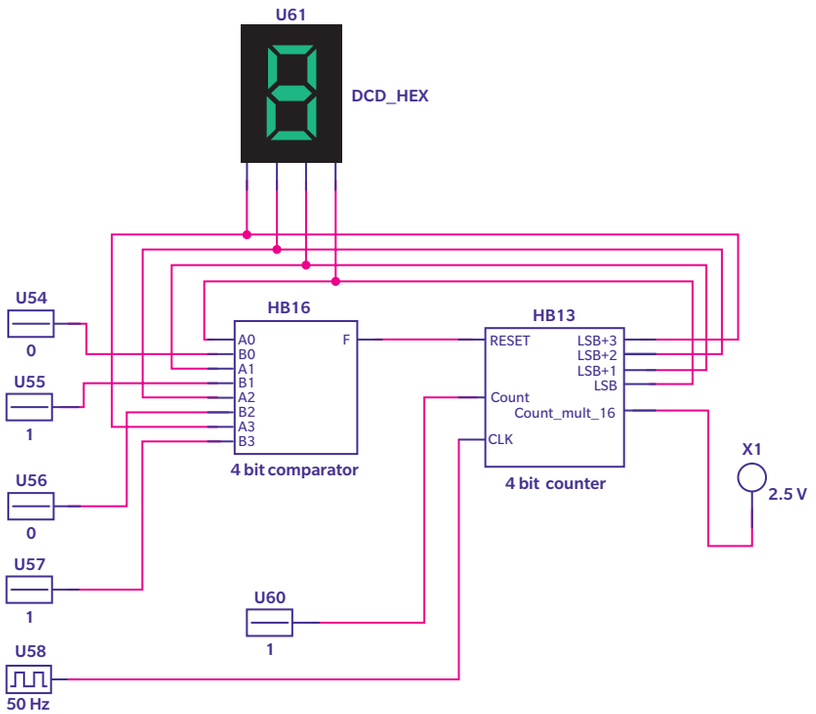
Figur 12.38 Synkron 16 bit teller

Det påstås at det er resultatene som teller, og for å være på den sikre siden er det alltid lurt å teste at blokken virker. Figur 12.39 viser den synkrone 16 bit telleren under uttesting. Som du ser, er Count satt til HØY, og RESET kan «toggles» for å teste at den virker. Idet skjermdumpen ble foretatt, hadde telleren kommet til 26C8 heksadesimalt, og hadde vi ventet lenge nok, hadde et blått lys blinket i X1 ved overgangen fra *FFFF* til 0000.



Figur 12.39 Uttesting av synkron 16 bit teller

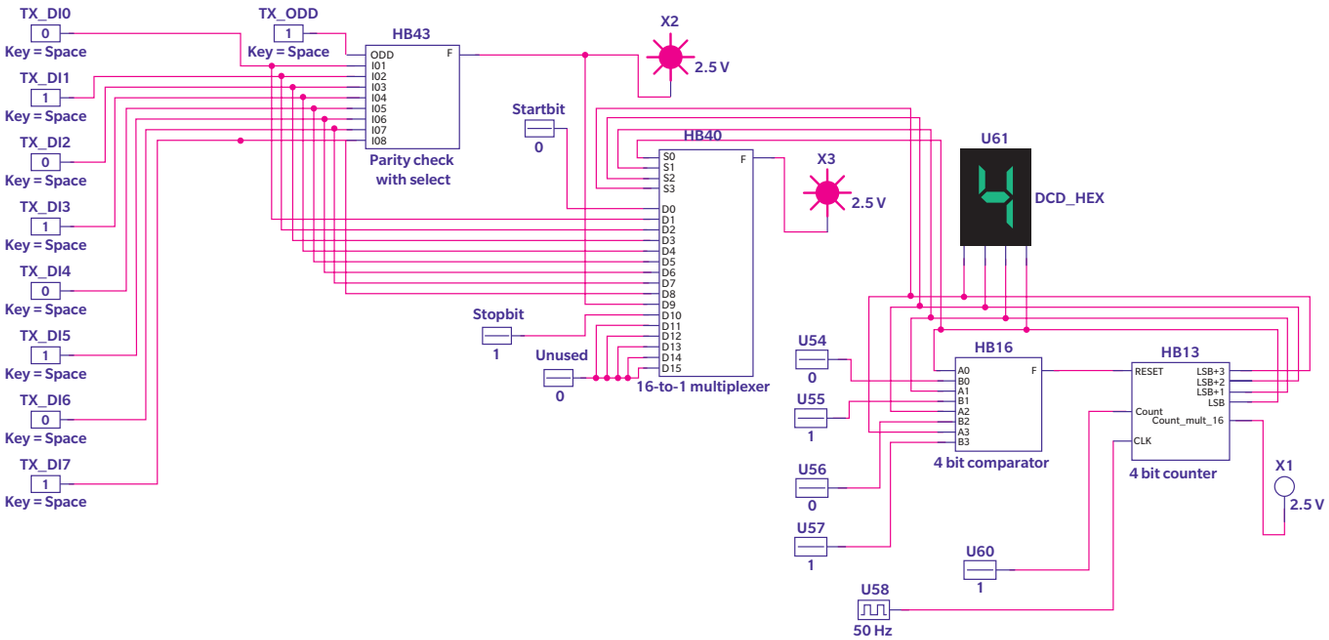
Med dette er alle blokkene vi skal ha med i UART_TX laget, og det gjenstår å sette dem sammen. Det kan være fornuftig å gå skrittvis til verks. Først setter vi sammen telleren og komparatoren og ser om deres samspill fungerer. Figur 12.40 viser testen av 4 bit utgavene.



Figur 12.40 Uttesting av 4 bit teller og komparator

Her er komparatoren hardkodet med det binære tallet 1010 som desimalt er lik 10. Når telleren blir lik 10, «wrapper» den. På samme måte kan en teste 16 bit utgavene, og en ser at de også virker sammen.

Det går jo som en lek. Da er det bare å sette sammen flere blokker etter oppskriften gitt i vårt blokkskjema i figur 12.30. I første omgang kan det være fornuftig å utelate 16 bit teller og komparator for ikke å få for mange variabler og ting å ta hensyn til. Figur 12.41 viser neste steg i uttestingen.



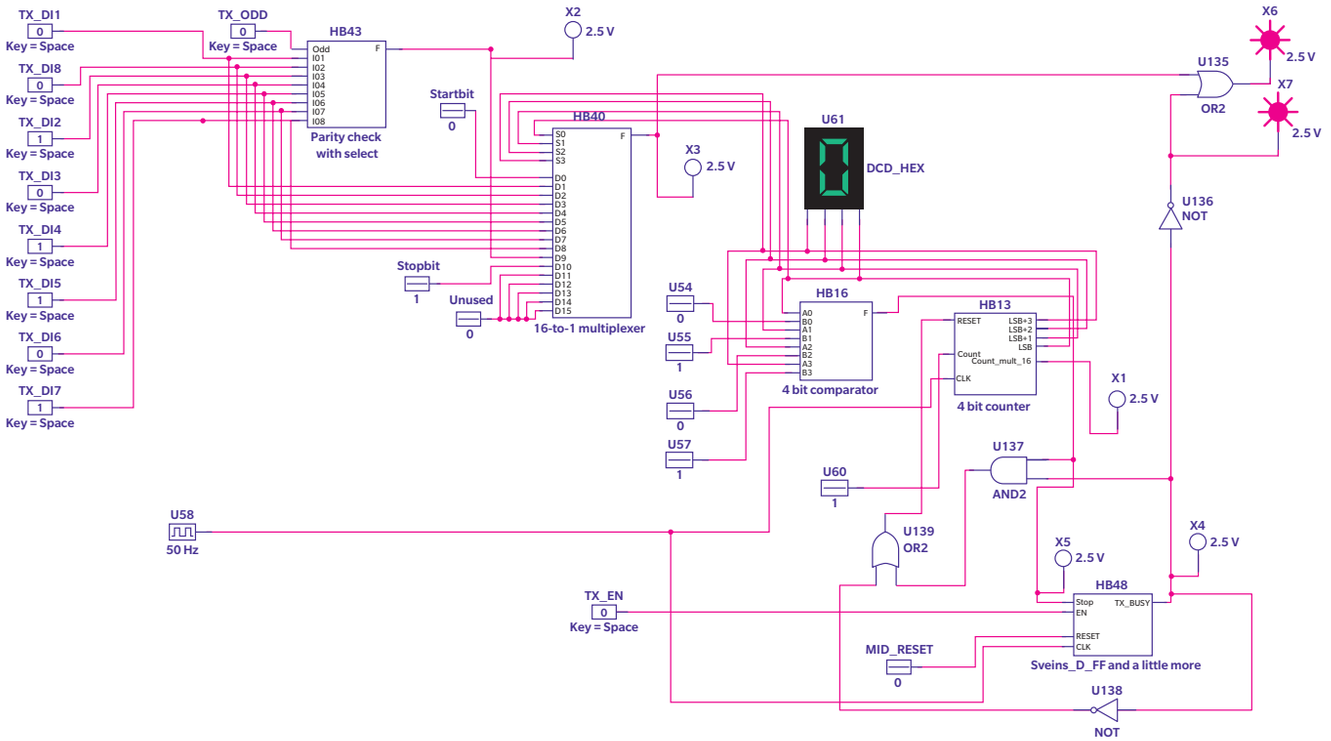
Figur 12.41 Uttesting av UART_TX uten 16 bit teller og komparator

I figur 12.41 har vi fått på plass «Parity check select» som leverer paritetsbit til D9 inngangen på 16-til-1 multiplekseren. TX_ODD velger hvilken type paritet. TXDI0 til TXDI7 sendes til «Parity check select» og multiplekserens D1 til D8 innganger. Startbit er hardkodet til 0 på multiplekserens D0 inngang, mens stoppbit er hardkodet til 1 på D10. De ubrukte inngangene D11 til D15 er satt til 0.

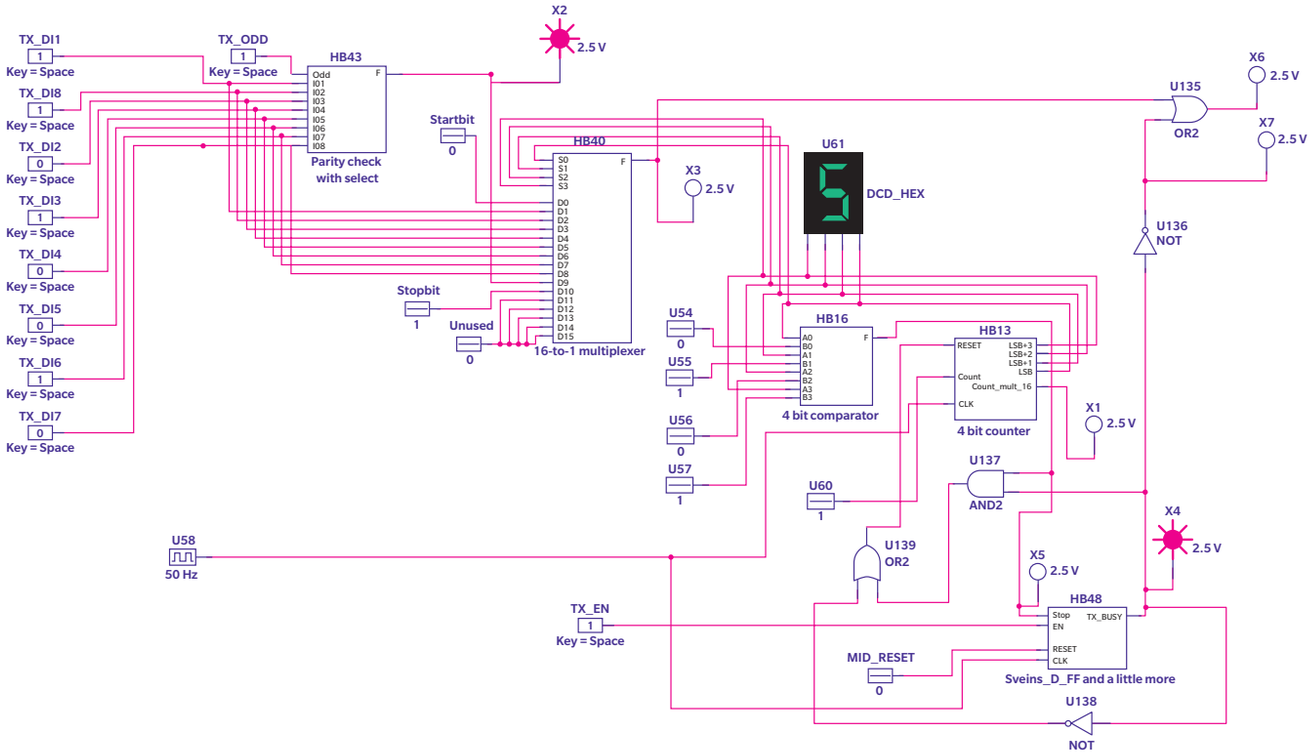
Hvilken av inngangene D0 til D11 på multiplekseren som sendes ut på porten F, bestemmes av S0 til S3 som får sine verdier av 4 bit tellereren. 4 bit komparatoren er hardkodet med binært 1010 (10 desimalt) som gjør at telleren «wrapper» ved 10 desimalt. Det som kommer ut på F porten på multiplekseren, skal vi senere sende ut på UART_TX sin TX_DO. For å få 4 bit telleren til å telle har vi satt på et klokkesignal på CLK inngangen.

Da er det bare å starte en simulering. Til vår glede ser vi at kretsen sender ut de rette bit på multiplekserens F port. Det er bare en liten hake. Det går i en evig runddans. Det trengs mer logikk for å få UART_TX til å starte og stoppe riktig. Neste blokk som må på plass, er «Sveins_D_FF and a little more» med litt garnityr rundt. Figur 12.42 viser den utvidede kretsen før oppstart. «Sveins_D_FF and a little more» er kommet på plass, og dens EN inngangen er koblet til TX_EN som kan «toggles» ved simulering. Når TX_EN er LAV, vil TX_BUSY være LAV, og tilbakekoblingen gjennom inverteren (NOT) U138 vil sørge for at telleren «resettes» hele tiden. Idet vi starter med en

TX_EN HØY puls, vil TX_BUSY bli HØY, og dermed begynner 4 bit telleren å telle. «Sveins_D_FF and a little more» sin Stop inngang er koblet til 4 bit komparatorens F utgang. Når F blir HØY (4 bit teller lik 1010 binært), vil TX_BUSY bli LAV. Så «wrapper» 4 bit telleren til 0000, og komparatorens F port blir LAV. F og TX_BUSY som nå begge er LAV OGes i U137, og 4 bit telleren «resettes» og utsending av bit fra multiplekserens F port stopper. Vi har klart å sende en og bare en byte. Signaler fra multiplekserens F port er ORet i U135 med det inverterte TX_BUSY signalet og skal i neste omgang sendes til TX_DO. Når TX_BUSY er HØY, er det bit fra multiplekseren som avgjør hva som kommer etter U135, men før og etter start vil TX_BUSY LAV sørge for at TX_DO vil være HØY som spesifikasjonen krever. «Sveins_D_FF and a little more» har en inngang til. RESET er midlertidig satt til 0 i påvente av en global «reset» i det endelig designet. Figur 12.43 viser kretsen under kjøring. Legg merke til at TX_EN fremdeles er HØY. Jeg er nok ikke den raskeste til å lage pulser, men så lenge TX_EN går til 0 før første «wrapping», går det greit.

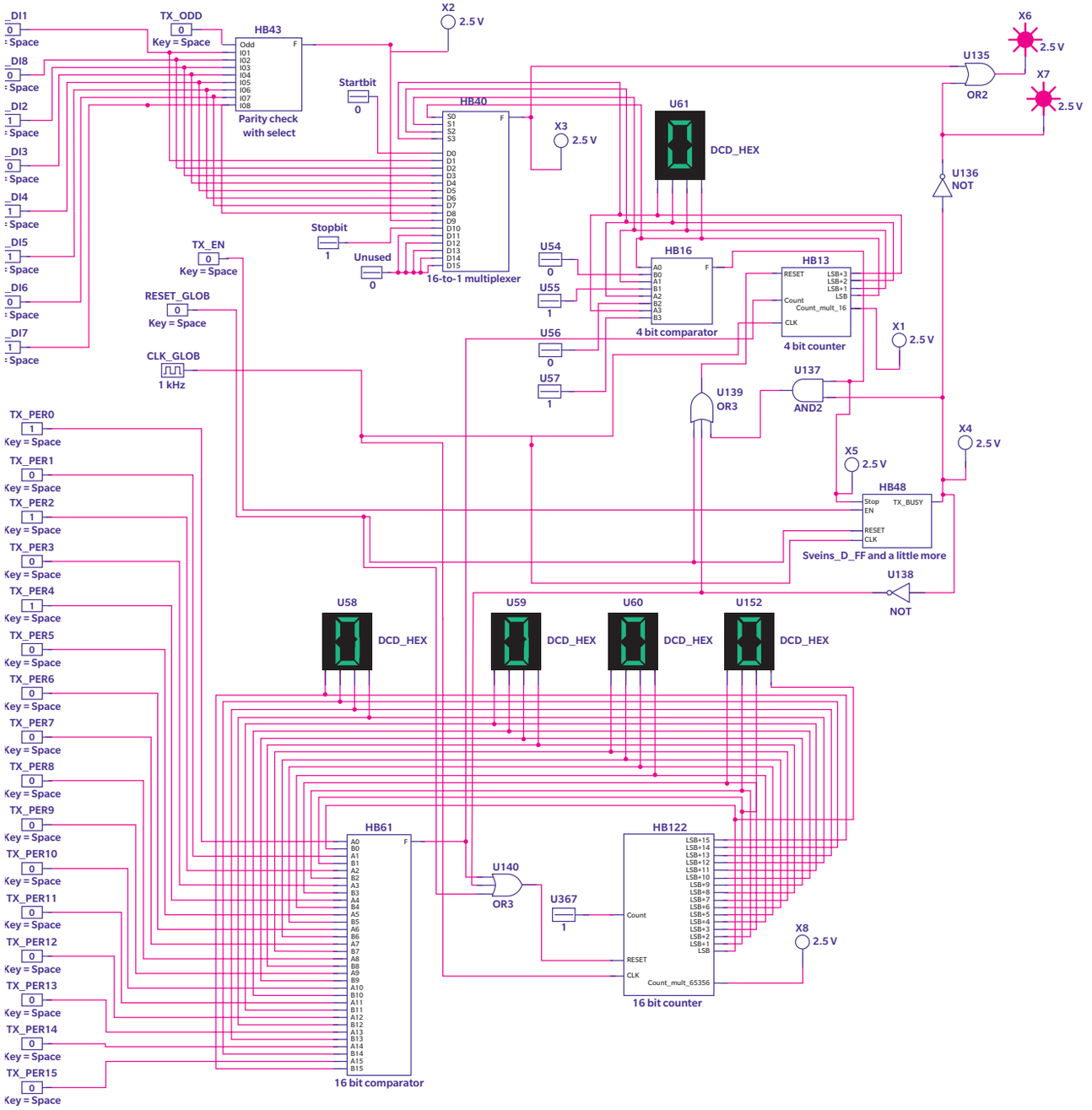


Figur 12.42 Uttesting av UART_TX start og stopp funksjon uten 16 bit teller og komparator. Her før oppstart



Figur 12.43 Uttesting av UART_TX start og stopp funksjon uten 16 bit teller og komparator. Her under kjøring

Da gjenstår det bare å få 16 bit teller og komparator på plass og koble dem sammen med 4 bit telleren. Kretsen er etter hvert blitt så stor at en trenger enten A2 ark eller lupe for å få med seg detaljene. Du får finne frem en lupe, så du får med deg detaljene i figur 12.44.

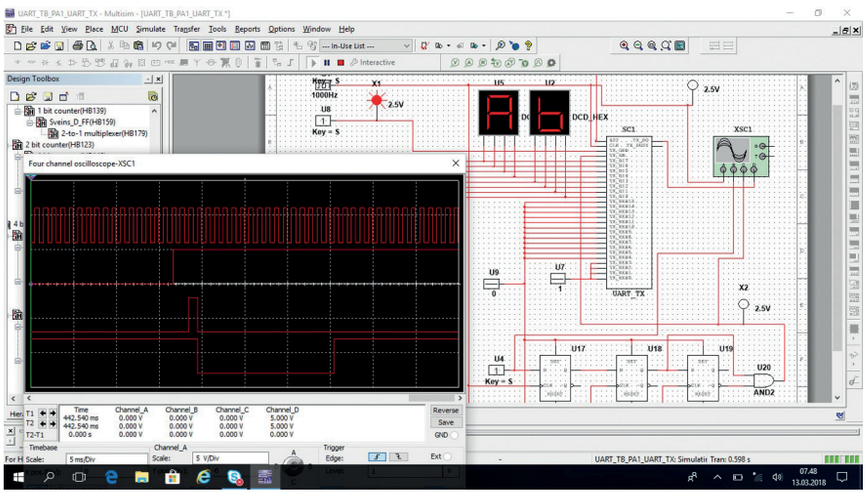


Figur 12.44 Den endelige utgaven av UART_TX før start

Som du ser av figur 12.44, så brukes det inverterte TX_BUSY til å «resette» 16 bit telleren før start og etter stopp av sending av en byte. Når TX_BUSY blir HØY vil 16 bit telleren starte å telle. I tillegg har vi nå fått en RESET_GLOB (UART_TX sin RST funksjon kobles på her) som vi kan bruke til å «resette» tellere og Sveins D-vippe. F utgangen på 16 bit komparatoren sendes til inngangen til 4 bit telleren slik at sist-nevnte øker med en hver gang 16 bit telleren «wrapper» og sender et Tick. CLK_GLOB (UART_TX sin CLK) er distribuert inn i alle UART_TX sine vipper.

Hildrende du! Vi er nesten i mål med designoppgave 2. Kretsen er testet med ulike påtrykk og oppfører seg etter alle solemerker som spesifisert. Eller? Underveis opp-daget jeg en liten hake. Vi skal med 16 bit telleren telle fra 0 til TX_PER-1. Nå teller vi fra 0 til TX_PER. En liten detalj kanskje, men rett skal være rett. Hvordan skal vi fikse det? Subtrahere med 1 i enda en krets? Heldigvis ble jeg reddet av gongongen da Svein sendte meg en Multisim testbenk han hadde laget for å tyne kretsen. I e-posten sto det: «Ser at det gir litt enklere kode hvis en spesifiserer at TX_PER skal settes til Ønsket_periode −1 (dvs. en setting på 99 gir sekvensen 0,1,2,…99)».

Da er vi klar for den ultimate testen, Sveins pinebenk. Jeg kopierer UART_TX inn i Sveins testbenk og starter enda en simulering. Figur 12.45 viser testmiljøet rett etter oppstart.



Figur 12.45 UART_TX i Sveins testbenk

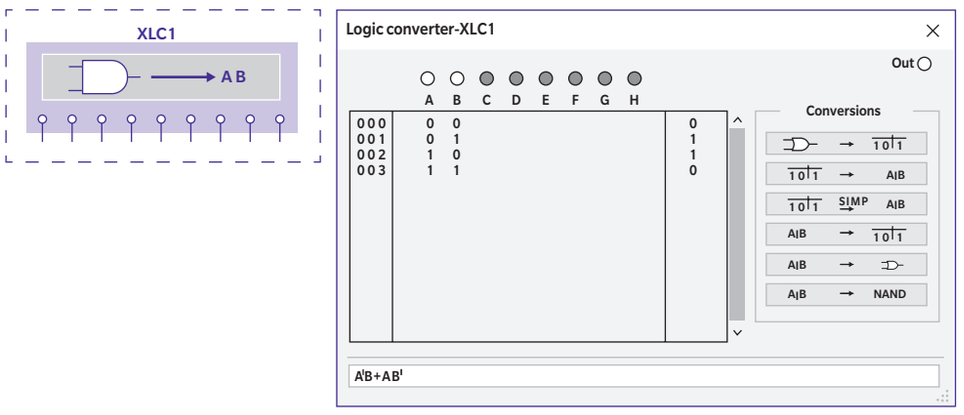
Vi ser vår UART_TX plassert litt til høyre i figur 12.45. Svein har hardkodet PER_TX til 1111 binært som er 15 desimalt. TX_DI er satt til AB heksadesimalt. Oscilloskopet i venstre hjørne viser tidsforløpet med et tidsdiagram. Øverst i oscilloskopets skjerm ser vi klokken. Den nest nederste grafen viser «enable» pulsen, og den nederste grafen

viser det som blir sendt ut på TX_DO. Vi ser at det går 15 klokkepulser for hver gang en ny TX_DI verdi sendes ut. Det første som sendes ut er startbit som er LAV, og så kommer TX_DI0 som er HØY. Etter mye kvalitetstid sammen med testbenken er jeg kommet til den erkjennelse at vår UART_TX virker i henhold til spesifikasjonen. Designoppgave 2 er utført.

Før vi gyver løs på neste designoppgave, kan det kanskje være fornuftig å reflektere litt over hva vi har lært så langt. For egen del har jeg gjenoppdaget den evige sannhet at der hvor vegringen er størst før en begynner, og mye positiv frustrasjonen og motstand er til stede underveis, er som regel mestringsgleden og læringsutbyttet størst. En annen viktig erkjennelse er at det som ikke er testet, virker ikke. Det har vært gjort feil underveis. Den verste flausen skjedde da jeg en sen kveldstime koblet utgangene fra to porter direkte sammen og ikke via en ny port. Ifølge Svein syndet jeg da mot digitalteknikkens først bud, men simulatoren tillot det med medfølgende underlig oppførsel. Du må gjerne splitte en utgang på en port så mye du vil, men for guds skyld, ikke koble to eller flere utganger direkte sammen. Du har kanskje også observert fraværet av Karnaugh-diagrammer, tilstandsmaskiner og andre utfordrende ting. Det skyldes det enkle faktum at vi har gått hierarkisk til verks. Det mest kompliserte vi har blitt utsatt for, er noen små og enkle sannhetstabeller og en D-vippe.

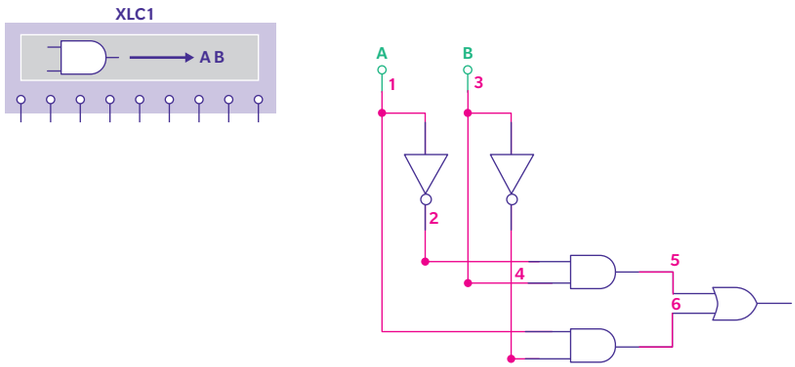
Dersom vi hadde hatt mer komplekse ting enn tellere i den sekvensielle logikken, hadde vi nok ikke sluppet unna tilstandsmaskiner. Våre tellere i dette tilfellet er jo tilstandsmaskiner, men såpass enkle at det ikke er nødvendig sette i gang hele apparatet rundt slike maskiner.

I kombinatorisk logikk er sannhetstabellen det grunnleggende. Sannheten ligger i tabellen, og det er vi som definerer den. For å understreke det faktum finnes det i Multisim en «Logic converter» hvor en oppgir sannhetstabellen, og så lages kretsen automagisk. I figur 12.46 har vi tatt vår sannhetstabell i tabell 12.1 for XOR-funksjonen.



Figur 12.46 Beskrivelse av sannhetstabell i «Logic converter»

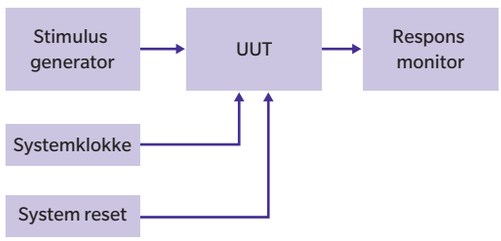
Så er det bare å trykke på den nest nederste knappen som omformer det hele til porter, og en får resultatet som vist i figur 12.47.



Figur 12.47 Resultat fra «Logic converter»

Etter å ha hvilt på laurbær en stund, er det bare å kaste seg over designoppgave 3. Der skal vi realisere arkitekturen i figur 12.30 i VHDL og simulere i ModelSim. Alle veier fører som kjent til Rom, men er den kortere med VHDL enn med vipper og porter? Tiden vil vise.

En ting er å lage en UART_TX i VHDL, men den må testes også. Som du kanskje husker fra kapittelet «Å sette ord på det», må en ha en testbenk for å kontrollere oppførselen til det vi lager. I figur 12.48 er det vist en generell testbenk. UUT er «Unit Under Test», altså vår UART_TX.



Figur 12.48 Generell testbenk

Igjen kommer Svein oss til unnsetning. «Jeg skal lage testbenk og UART_TX skallet til deg. Du må bare fylle UART_TX med innhold.» Jeg åpner Sveins testbenkfil. Er det mulig å forstå det som står skrevet?

```
-- library declarations
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;

-- entity declaration of testbench
entity TB_UART_TX is
end TB_UART_TX;

-- behavioral decription of testbench
architecture BEHAV of TB_UART_TX is

-- UART_TX Component Declaration
component UART_TX is
    port
    (
        CLK    : in std_logic;
        RST    : in std_logic;
        TX_EN     : in std_logic;
        TX_PER : in std_logic_vector(15 downto 0);
        TX_DI     : in std_logic_vector(7 downto 0);
        TX_ODD : in std_logic;
        TX_BUSY   : out std_logic;
        TX_DO     : out std_logic
    );
end component;

-- Signal Declarations
shared variable T   : time := 20 ns;
shared variable TSU : time := 5 ns;
signal VectorClk    : std_logic := '0';
signal StrobeClk    : std_logic := '0';

-- UART_TX Input Signals
signal CLK    : std_logic := '0';
signal RST    : std_logic := '0';
signal TX_EN  : std_logic := '0';
signal TX_ODD : std_logic := '1';
signal TX_PER : std_logic_vector(15 downto 0) := (others => '0');
signal TX_DI  : std_logic_vector(7 downto 0) := (others => '0');

-- UART_TX Output Signals
signal TX_BUSY: std_logic;
```

```
signal TX_DO  : std_logic;

-- UART_TX InOut Signals
begin
-- UART_TX Port Map
UART_TX0 : UART_TX
port map
(
    CLK    => CLK,
    RST    => RST,
    TX_EN     => TX_EN,
    TX_PER => TX_PER,
    TX_DI     => TX_DI,
    TX_ODD => TX_ODD,
    TX_BUSY   => TX_BUSY,
    TX_DO     => TX_DO
);

-- infinite clock generator
VectorClk <= not VectorClk after T/2;
CLK <= not VectorClk after TSU;

-- init process
INIT : process
begin
-- insert your stimuli here !
    wait for T;
    RST <= '1';

    wait for T;
    RST <= '0';
    TX_PER <= X"000A";

    wait for 10*T;
    TX_DI <= X"5B";
    TX_EN <= '1';

    wait for T;
    TX_EN <= '0';

    wait for 3 us;
    TX_DI <= X"00";
```

```
    TX_EN <= '1';

    wait for T;
    TX_EN <= '0';

    wait for 3 us;
    TX_DI <= X"FF";
    TX_EN <= '1';

    wait for T;
    TX_EN <= '0';

    wait for 1 sec;

    end process;
end BEHAV of TB_UART_TX;
```

Svein starter koden med å ta med de bibliotekene han trenger. Deretter deklarerer han testbenken «**entity** TB_UART_TX **is**» og beskriver dens innhold. Legg merke til at han bruker frasen «**architecture** BEHAV of TB_UART_TX **is**». BEHAV signaliserer at det i testbenken brukes språklige konstruksjoner som **wait** og **after**, som ikke kan brukes til syntese av kretser, men til uttesting. Så følger komponentdeklareringen av UART_TX «**component** UART_TX is» slik at min kode skal bli tilgjengelig for testbenken. Inne i selve «**architecture** BEHAV of TB_UART_TX is» starter det hele med en «port map» mot UART_TX, før en uendelig klokke lages med følgende konstruksjon «VectorClk <= not VectorClk after T/2; CLK <= not VectorClk after TSU; ». Så følger selve prosessen som skal utsette vårt design for juling. Først kommer «-- insert your stimuli here ! **wait for** T; RST <= '1';» som setter «reset» til HØY. Etter T settes reset til LAV, og TX_PER gis verdi «**wait for** T; RST <= '0'; TX_PER <= X"000A";». X signaliserer her at verdien er gitt heksadesimalt. Så følger TX_DI verdiene, og TX_EN settes HØY «**wait for** 10*T; TX_DI <= X"5B"; TX_EN <= '1';», og ballet kan begynne. Etter T slutter første runde av festen med «**wait for** T; TX_EN <= '0';». Deretter følger to nye runder hvor ekstremverdiene til TX_DI blir testet. Vi kunne selvfølgelig ønsket oss mer av «response monitor» til Sveins testbenk. Den er nærmest fraværende. Testbenken viser kun utgangsverdiene TX_DO og TX_BUSY. En testbenk inneholder som regel mer kode for å teste at det som kommer ut, er i tråd med det som kommer inn. I vårt tilfelle får vi bruke ModelSim for å sjekke hva som skjer.

Hvordan ser så min kodebit ut idet Svein gir meg den?

```
library ieee;
use ieee.std_logic_1164.all;
```

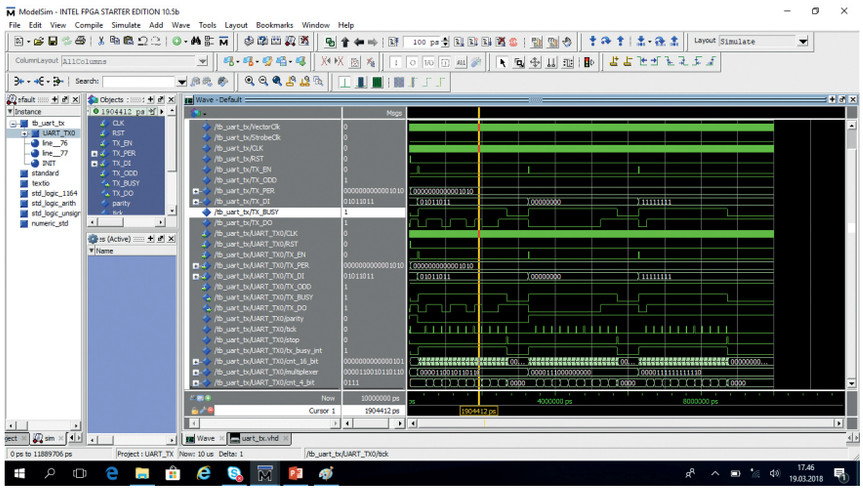
```
use ieee.std_logic_unsigned.all;

entity UART_TX is
port(
-- inputs
   CLK :in std_logic; -- System Clock
   RST :in std_logic; -- Sync reset active high
   TX_EN :in std_logic; -- TX Enable
   -- Rate configuration
   TX_PER :in std_logic_vector(15 downto 0);
   TX_DI :in std_logic_vector(7 downto 0); -- write byte
   TX_ODD :in std_logic;
   TX_BUSY :out std_logic; -- Busy
   TX_DO :out std_logic -- Serial output
);
end UART_TX;

architecture RTL of UART_TX is
begin
-- Fyll inn ;-)
end RTL of UART_TX;
```

Den har en port som skal sørge for tilgang til signaler fra utenverdenen, og RTL, som er en forkortelse for «Register Transfer Level», gir beskjed om at det som fylles inn i arkitekturen skal kunne realiseres i «hardware». I selve blokken har Svein fylt inn en emoji for å motivere oss til videre arbeid.

Hvordan skal vi gripe det an? Svein har indirekte gitt oss et hint ved å si at vi skal realisere arkitekturen i figur 12.30. Det kan være fornuftig å lage de blokkene som der er gitt. Hvilken programmeringsstil skal vi velge – strukturell, dataflyt eller oppførselsbasert? Mine tanker går umiddelbart mot strukturelt, men Svein tar umiddelbart til motmæle. «Strukturelt er vel og bra, og vi har allerede brukt metoden mellom testbenken jeg har laget og UART_TX som du skal lage. Du vil oppdage at mengden kode inne i UART_TX vil bli liten, og dermed blir strukturell metode fullstendig «overkill». Den metoden vil i det tilfellet gjøre koden vanskeligere å lese og forstå». La oss gi oss for overmakten og satse på en blanding av dataflyt og oppførselsbasert kode.

Hvor skal en begynne? Kombinatorisk logikk først og så sekvensiell? Igjen har Svein tips på lager. «Start fra inngangen og jobb deg mot utgangen. Riktignok skjer alt samtidig, men en slik angrepsmåte gjør koden enklere å lese.»

Da er det bare å starte ModelSim og lage et prosjekt som inneholder Sveins testfil og den filen som vi skal lage. Uvisst av hvilken grunn ble prosjektet kalt UART_TX. I ModelSim har man en fargeglad «editor» som hjelper en å følge VHDL språkets syntaks. Ifølge Svein har ikke ModelSims «look and feel» endret seg i løpet av de siste tjue år, og det er godt å vite når en går i gang og lærer seg et nytt verktøy. Når en har kodebiter ferdig til testing, er det bare å kompilere filene i prosjektet og simulere det en har laget. Ved å bruke ModelSims «wave» vindu kan en finmyse på resultatet av simuleringen og til slutt forsikre seg om at en har laget det en ønsket. Figur 12.49 viser en skjermdump av simuleringen av vår krets foretatt i et euforisk øyeblikk da alt endelig virket.



Figur 12.49 Simulering av ferdig UART_TX

I den grå delen av «wave» vinduet ser du signalene. Øverst er signalene fra Svein, og våre signaler starter med /tb_uart_tx/UART_TX0/CLK og nedover. Signalene med små bokstaver er de vi bruker lokalt i vår UART_TX. I det grå Msgs felt rett til høyre ser du verdiene av signalene ved den vertikale streken i den svarte delen av «wave» vinduet. I figur 12.49 er streken satt på 1904212 picosekunder etter start. I det svarte feltet finner du tidsdiagrammer for de enkelte signaler, og tro meg, de er helt i henhold til spesifikasjonen som var gitt.

Hvor mange tusen linjer med kode ble så det? Vel, jeg skal ikke la deg vente i spenning, men heller avsløre den straks. Etterpå skal vi ta en gjennomgang av de enkelte bitene i et forsøk på å forstå.

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;
```

```
entity UART_TX is
port(
-- inputs
    CLK    :in std_logic; -- System Clock
    RST    :in std_logic; -- Sync reset active high
    TX_EN     :in std_logic; -- TX Enable
 -- Rate configuration
    TX_PER :in std_logic_vector(15 downto 0);
    TX_DI     :in std_logic_vector(7 downto 0); -- write byte
    TX_ODD :in std_logic;
    TX_BUSY   :out std_logic; -- Busy
    TX_DO     :out std_logic -- Serial output
);
end UART_TX;

architecture RTL of UART_TX is
signal parity, tick, stop, tx_busy_int : std_logic;
-- 16 bit counter value and multiplexer
signal cnt_16_bit, multiplexer: std_logic_vector(15 downto 0);
-- 4 bit counter value
signal cnt_4_bit : std_logic_vector(3 downto 0);
begin
-- Start and stop. Starting up when TX_EN pulse is HIGH
-- The simplest FSM one can imagine.
-- A sort of S R flip flop with priority
process (CLK) -- Starting up
begin
   if rising_edge(CLK) then
      if RST = '1' then
         tx_busy_int<='0';
      else
         if TX_EN='1' then
            tx_busy_int <='1';
         else
            if stop='1' then
              tx_busy_int<='0';
            end if; -- stop
         end if; -- TX_EN
      end if; -- RST
   end if; -- rising_edge
end process; -- process Start and stop
```

```
TX_BUSY<=tx_busy_int;

-- Calculating parity bit
parity<=((((TX_DI(0) xor TX_DI(1)) xor (TX_DI(2) xor TX_DI(3)))
    xor ((TX_DI(4) xor TX_DI(5)) xor (TX_DI(6) xor TX_DI(7))))
    xor TX_ODD);

-- Filling the multiplexer with values.
multiplexer(0)<='0'; -- Startbit
multiplexer(1)<=TX_DI(0);
multiplexer(2)<=TX_DI(1);
multiplexer(3)<=TX_DI(2);
multiplexer(4)<=TX_DI(3);
multiplexer(5)<=TX_DI(4);
multiplexer(6)<=TX_DI(5);
multiplexer(7)<=TX_DI(6);
multiplexer(8)<=TX_DI(7);
multiplexer(9)<=parity;
multiplexer(10)<='1'; -- Stopbit
-- Keeping TX_DO high while waiting for tx_busy_int='0'
multiplexer(11)<='1';
multiplexer(12)<='0'; -- Unused
multiplexer(13)<='0'; -- Unused
multiplexer(14)<='0'; -- Unused
multiplexer(15)<='0'; -- Unused

-- Another way of doing the same thing
-- multiplexer<="00001";'1'&parity&TX_DI&'0';

process (CLK) -- 16 bit counter
begin
   if rising_edge(CLK) then
      if RST = '1' or tick = '1' or
         tx_busy_int='0' then
      -- All the bits in the cnt_16_bit vector
      -- is set to zero
         cnt_16_bit <= (others => '0');
      else
      -- The counter is incremented with one
      -- and wraps if full
         cnt_16_bit <= cnt_16_bit + 1;
      end if; -- RST or tick or tx_busy_int
```

```
      end if; -- rising_edge
   end process; -- process 16 bit counter

   -- miniprocess with implicit sensitivity
   tick <= '1' when cnt_16_bit = TX_PER else '0';

   process (CLK) -- 4 bit counter
   begin
      if rising_edge(CLK) then
         if RST = '1' or tx_busy_int='0' then
         -- All the bits in the cnt
         -- vector is set to zero
            cnt_4_bit <= (others => '0');
         else
         -- The counter is incremented with one
         -- and wraps if full
            if tick = '1' then
               cnt_4_bit <= cnt_4_bit + 1;
            end if;
         end if; -- RST
      end if; -- rising edge
   end process; -- process 4 bit counter

   -- miniprocess with implicit sensitivity
   stop <= '1' when cnt_4_bit = "1011" else '0';

   -- Sending multiplexer values to TX_DO
   TX_DO <= '1' when tx_busy_int='0' else
      multiplexer(to_integer(unsigned(cnt_4_bit)));

end RTL of UART_TX;
```

Det er nesten for godt til å være sant. Det vi brukte 25 sider på å konstruere med porter og tellere, er kokt ned til 2 A4 sider med VHDL kode. Det føles nesten som juksing. Ikke rart at produktiviteten skjøt i været med VHDL.

Før vi starter med en gjennomgang, kan det være fornuftig nok en gang å minne seg selv på at VHDL er et språk som beskriver «hardware», og at alt skjer samtidig overalt. For å si det med matematikerne: Faktorenes orden er likegyldig. Hvilke kodesnutter i UART_TX som kommer først eller sist, er totalt uinteressant, men Sveins råd om å gå fra inngang til utgang er forsøkt fulgt for å gjøre det hele mer lesbart. I tillegg kan

det nok spores snev av sekvensiell tankegang hos meg i kodens struktur. Jeg prøver å bli kvitt den uvanen.

La oss starte øverst. Der har vi definert noen biblioteker som er høyst standard og nok for vår bruk. Prøv å bruke standardbiblioteker, da slipper du overraskelser senere. Så kommer UART_TX sin **entity** blokk med portmappingen som gjør det mulig for oss å få tak i de signaler som testbenken til Svein serverer.

Øverst i **architecture** er det definert signaler som trengs i UART_TX. Det første som møter oss inne i kroppen til **architecture** for UART_TX, er en enkel vippe som skal holde orden på når TX_BUSY er HØY eller LAV. Nedenfor er kodesnutten for dette gjengitt, og vi ser at det er brukt en sekvensiell sekvens styrt av klokken. Vi holder oss i den synkrone verden også her i VHDL. Dersom det kommer en RST eller sendingen er ferdig, indikeres med stop='1', blir tx_busy_int satt til LAV ellers når en får en TX_EN='1' blir tx_busy_int satt til HØY. Hvorfor bruker vi ikke TX_BUSY direkte istedenfor den interne tx_busy_int? Det skyldes det enkle faktum at VHDL av en eller annen grunn nekter bruk av utgangssignaler til test av betingelser, noe vi trenger andre steder i koden. Det interne signalet overføres til TX_BUSY med tilordningen under prosessen.

```
-- Start and stop. Starting up when TX_EN pulse is HIGH
-- The simplest FSM one can imagine.
-- A sort of S R flip flop with priority
process (CLK) -- Starting up
begin
   if rising_edge(CLK) then
      if RST = '1' then
         tx_busy_int<='0';
      else
         if TX_EN='1' then
            tx_busy_int <='1';
         else
            if stop='1' then
               tx_busy_int<='0';
            end if; -- stop
         end if; -- TX_EN
      end if; -- RST
   end if; -- rising_edge
end process; -- process Start and stop

TX_BUSY<=tx_busy_int;
```

Paritetssjekkeren kan enkelt lages med dataflytmetode. Følgende linje fra UART_TX arkitekturen gjør susen.

```
parity<=((((TX_DI(0) xor TX_DI(1)) xor (TX_DI(2) xor TX_DI(3)))
   xor ((TX_DI(4) xor TX_DI(5)) xor (TX_DI(6) xor TX_DI(7))))
   xor TX_ODD);
```

Dette hierarkiet av eksklusiv OR gjør det samme som paritetssjekkeren i figur 12.6 basert på porter. Her har vi gjort en liten vri og bare XORet TX_ODD med TX_DI. Det samme kunne vi ha gjort da vi brukte porter, men da trengte vi allikevel en 2-til-1 multiplekser. Til slutt tilordnes det hele et signal parity som vi har definert lokalt i UART_TX arkitekturen.

Det neste vi støter på, er multiplekseren. Egentlig kunne vi ha kopiert og utvidet den basert på 4-til-1 multiplekseren fra kapittelet «Å sette ord på det», men jeg har valgt en annen vri hvor vi bruker en vektor som indekseres. Innsignalene til multiplekseren tilordnes med følgende kode:

```
-- Filling the multiplexer with values.
multiplexer(0)<='0'; -- Startbit
multiplexer(1)<=TX_DI(0);
multiplexer(2)<=TX_DI(1);
multiplexer(3)<=TX_DI(2);
multiplexer(4)<=TX_DI(3);
multiplexer(5)<=TX_DI(4);
multiplexer(6)<=TX_DI(5);
multiplexer(7)<=TX_DI(6);
multiplexer(8)<=TX_DI(7);
multiplexer(9)<=parity;
multiplexer(10)<='1'; -- Stopbit
-- Keeping TX_DO high while waiting for tx_busy_int='0'
multiplexer(11)<='1';
multiplexer(12)<='0'; -- Unused
multiplexer(13)<='0'; -- Unused
multiplexer(14)<='0'; -- Unused
multiplexer(15)<='0'; -- Unused

-- Another way of doing the same thing
-- multiplexer<="00001";'1'&parity&TX_DI&'0';
```

Den nederste linjen som er kommentert ut, er kun en annen måte å gjøre det samme på. Hvordan vi bruker denne vektoren til å fore TX_DO, skal vi se på om en stund.

Det neste vi finner i koden, er 16 bit telleren og dens komparator. Da en teller er sekvensiell av natur, er det naturlig å bruke oppførselsbasert metode basert på en prosess. I **process** sin sensitivitetsliste har vi kun klokken CLK. Vår 16 bit counter skal altså reagere hver gang klokken har en ny positiv flanke. Vi driver altså med synkront design hvor det er klokken som styrer. Dersom en har «reset» RST HØY, tick, dens rolle skal vi straks komme til, HØY eller tx_busy_int LAV, nullstilles tellervektoren cnt_16_bit med **others** konstruksjonen, ellers øker cnt_16_bit med 1. Etter vår prosess har vi koden «tick <= '1' **when** cnt_16_bit = TX_PER **else** '0';». Her ser vi at det interne signalet tick settes til HØY når telleren er lik TX_PER, og ellers til LAV. I VHDL tok det faktisk kun en linje å skrive komparator. Denne linjen er en slags miniprosess med implisitt sensitivitet.

```
process (CLK) -- 16 bit counter
begin
   if rising_edge(CLK) then
      if RST = '1' or tick = '1' or
         tx_busy_int='0' then
      -- All the bits in the cnt_16_bit vector
      -- is set to zero
         cnt_16_bit <= (others => '0');
      else
      -- The counter is incremented with one
      -- and wraps if full
         cnt_16_bit <= cnt_16_bit + 1;
      end if; -- RST or tick or tx_busy_int
   end if; -- rising_edge
end process; -- process 16 bit counter

-- miniprocess with implicit sensitivity
tick <= '1' when cnt_16_bit = TX_PER else '0';
```

Miniprosessen ovenfor kunne vært skrevet på følgende måte:

```
process (cnt_16_bit, TX_PER) – the same with explicit sensitivity
begin
   if cnt_16_bit = TX_PER then
      tick<='1';
   else
      tick<='0';
   end if;
end process;
```

Det ville gitt samme resultat, men vesentlig mer kode.

Så er det 4 bit telleren med dens prosess.

```
process (CLK) -- 4 bit counter
begin
   if rising_edge(CLK) then
      if RST = '1' or tx_busy_int='0' then
      -- All the bits in the cnt
      -- vector is set to zero
         cnt_4_bit <= (others => '0');
      else
      -- The counter is incremented with one
      -- and wraps if full
         if tick = '1' then
            cnt_4_bit <= cnt_4_bit + 1;
         end if;
      end if; -- RST
   end if; -- rising edge
end process; -- process 4 bit counter

-- miniprocess with implicit sensitivity
stop <= '1' when cnt_4_bit = "1011" else '0';
```

Den nullstilles når RST er HØY eller tx_busy_int er LAV, og økes med 1 hver gang betingelsen tick = '1' er oppfylt. Det var dette tick signalet som ble brukt i vår 16 bit teller til nullstilling når den «wrapper». Under selve 4 bit tellerprosessen er det en miniprosess som sørger for at vår UART_TX går i pausemodus og venter på neste TX_EN pulse. Stop settes til 1 når cnt_4_bit blir desimalt 11. Når stop settes til 1, vil tx_busy_int settes til 0 i første delen av koden, og dermed blir TX_BUSY også lik 0.

Da er det bare en linje igjen å kommentere.

```
-- Sending multiplexer values to TX_DO
TX_DO <= '1' when tx_busy_int='0' else
   multiplexer(to_integer(unsigned(cnt_4_bit)));
```

Denne miniprosessen sørger for at innhold i multiplekseren sendes til TX_DO når betingelsen tx_busy_int='1', ellers sendes ut HØY som er kravet til TX_DO i den såkalte «idle», tomgangstilstanden. Vi bruker cnt_4_bit sin verdi til å indeksere multiplekservektoren. Da cnt_4_bit er en vektor, må den gjøres om til tall med konstruksjonen unsigned(cnt_4_bit).

Da er vi ferdig med designoppgave 3 også. Finnes det andre som har syslet med denne designoppgaven? Ja, på nettet ligger det diverse utgaver av VHDL kode som påstås å gjøre det samme som vår. Derimot fant jeg ikke så mange forslag på løsninger med Multisim. Kun et nødrop fra en fortvilt student: «I'd pay someone to do this project for me as I'm really desperate. Contact me if you're interested. Deadline is soon and I am swamped with work.»

Hva nå? Kanskje kan vi lage en UART_RX? Den skal jo være mer utfordrende, og etterpå kan en sette UART_TX og UART_RX sammen i Sveins testbenk og se de to spille sammen i harmoni.

Lykke til!

13

Kapittel 13

Kun i minne finner hjertet fred

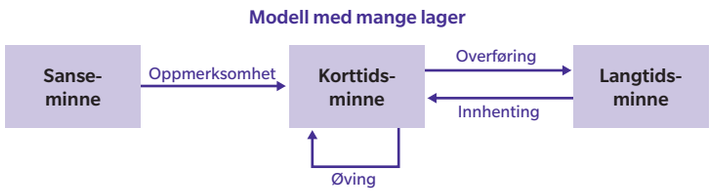
«Sjel, vær trofast til det siste: Seirens seier er alt å miste.
Tapets alt din vinning skapte – Evig eies kun det tapte!»

Brand, Henrik Ibsen (1828–1906)

LÆRINGSUTBYTTE: Minne, flyktig minne, permanent minne, cache, asynkront og synkront minne, ordlengde, fra lås til minne, Random Access Memory (RAM), statisk og dynamisk RAM, Read Only Memory (ROM), programmerbar og ikkeprogrammerbar ROM, Flash, minne i VHDL, minnehierarki

I've seen things you people wouldn't believe. Attack ships on fire off the shoulder of Orion. I watched C-beams glitter in the dark near the Tannhauser Gate. All those moments will be lost in time… like tears in rain… Time to die.

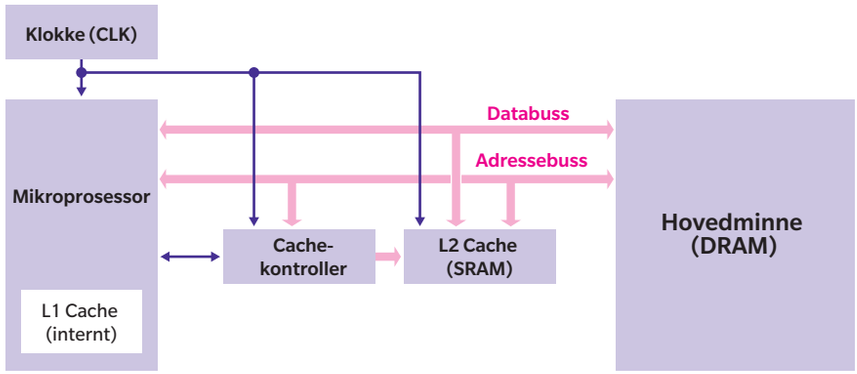
Humanoiden Battys sluttreplikk i filmen «Blade Runner» ved nedkoblingstidspunktet viser med all ønskelig tydelighet verdien av minne både for mennesker og maskiner. Er det noen likhet mellom menneskets minne og digitale minner? For å kunne svar på det spørsmålet er det naturlig å ta en titt på en mye brukt, men begrenset, modell for menneskets minne.



Figur 13.1 Atkinson-Shiffrins modell for det menneskelige minne

Denne modellen i figur 13.1 deler opp minnet i et kortidsminne og et langtidsminne. Kortidsminnet er flyktig (volatilt), mens langtidsminnet er permanent (ikke volatilt) av natur. Kortidsminnet må hele tiden holdes i ånde ved oppfriskning og nye impulser fra utenverdenen. Informasjon fra kortidsminnet kan lagres i langtidsminnet, og gamle minner kan hentes derfra til videre bearbeidelse i korttidsminnet. Det er kanskje et eller annet adresseringssystem for å lagre og hente minner fra langtidsminnet, selv om det av og til føles som det er tilfeldigheter og ville assosiasjoner som bestemmer hva som kommer frem til pannebrasken. Minner som er lagret, er alt fra korte impulser til lange hendelser.

Minner så dette om hvordan digitale minner er bygd opp? Figur 13.2 viser en typisk oppbygging av en minnestruktur rundt en prosessor. Internt i prosessoren har en et hurtigminne (cache), og utenfor prosessoren har en to minner som kan adresseres. Det ene minnet er et hurtigminne bygd med vipper som minneceller (SRAM), og det andre, litt tregere hovedminnet er bygd opp med kondensatorer som minneceller (DRAM). SRAM står for Static Random Access Memory. Static betyr at det er brukt vipper, og RAM betyr at en kan lese eller skrive til minnet med en vilkårlig valgt adresse. DRAM krever jevnlig oppfriskning av ladning til kondensatorene (D for Dynamic).



Figur 13.2 Digital minneoppbygging i et datamaskinsystem

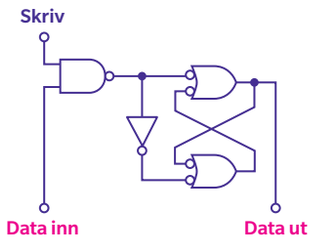
Som en ser, er det mange likheter mellom menneskelig og maskinelt minne. Skal man lære, noe så må en ha minne enten man er menneske eller kunstig intelligens. Vårt korttidsminne er slående likt det digitale hurtigminnet, og begge brukes til å hente data som nylig har vært brukt på en rask måte, slik at en unngår å lete det frem fra langtidsminnet eller hovedminnet. Noen forskjeller er det riktignok. Hos mennesket er det korttidsminnet som hele tiden må oppfriskes, mens det i datamaskinen er hovedminnet som må få jevnlig påfyll av nye ladninger. Årsaken til at en har brukt kondensatorer i DRAM, er at det er billigere og mer kompakt. Prisen en betaler er at det er tregere, og derfor brukes det ikke nært prosessoren. I motsetningen til mennesket, er det digitale minnet enkelt adresserbart, og alle delene av minne har samme lengde – såkalt ordlengde. Denne ordlengden varierer med ulike systemer, men er typisk et multippel av 8 bit (1 byte). Av figuren ser en også at hurtigminnene er styrt av en klokke. Data inn eller ut av hurtigminnene kan bare skje ved en klokkepuls (synkront minne), mens hovedminnet som er vesentlig tregere, er typisk asynkront.

Verken hovedminnet (DRAM) eller hurtigminnet (SRAM) er permanent. I en datamaskin lagres permanente data i Read Only Memory (ROM). Som navnet tilsier, så kan data bare leses, og ikke skrives, til dette minnet. ROM brukes blant annet til å lagre data som skal være permanente og blant annet brukes til oppstart av digitale enheter. En avart av ROM er såkalt PROM (Programmable ROM), og den mest benyttede typen er EEPROM som kan slettes og programmeres ved elektriske pulser. Det finnes enda et slag ikke-flyktig minne, og det er Flash. Dette transistorbaserte minnet kan det både skrives til og leses fra. Flash er populært i periferienheter som minnepinner og lagringsmedium i kamera og mobiltelefoner. Tabell 13.1 gir en oversikt over minnetyper og deres egenskaper.

Tabell 13.1 Sammenligninger av minnetyper

Type minne	Ikke-flyktig	Kompakt	En transistor celle	Skrivbar
Flash	Ja	Ja	Ja	Ja
SRAM	Nei	Nei	Nei	Ja
DRAM	Nei	Ja	Ja	Ja
ROM	Ja	Ja	Ja	Nei
EEPROM	Ja	Nei	Nei	Ja

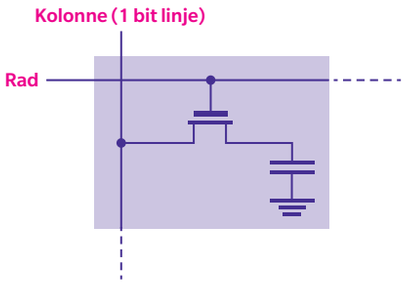
La oss starte med å se på det enkleste minnet en kan tenke seg. En minnecelle som kan lagre ett bit. Vi har allerede vært borti et slikt minne i kapittelet «Tingenes Tilstand». I figuren 7.21 så vi hvordan en kunne lage et 1 bits minne med en portstyrt D-lås. Figur 13.3 viser en avart av denne D-låsen som brukes som SRAM minnecelle.



Figur 13.3 SRAM minnecelle

Cellen blir valgt og kan skrives til ved å sette Skriv til HØY og gi et data bit (1 eller 0) på Data inn linjen. Et data bit kan leses på Data ut linjen. Logikken i figur 13.3 kan bygges med CMOS teknologi som vist i appendikskapittelet «Store tanker gir små kretser». Totalt vil denne konstruksjonen kreve 13 transistorer, og vi skal snart se at andre minnetyper enn SRAM kan klare seg med vesentlig færre transistorer og dermed bygges mer kompakt. SRAM-minnet er statisk, og så lenge det har strøm, kan det holde på 1 eller 0 i det uendelige. Dersom strømmen slås av, vil det lagrede bit forsvinne.

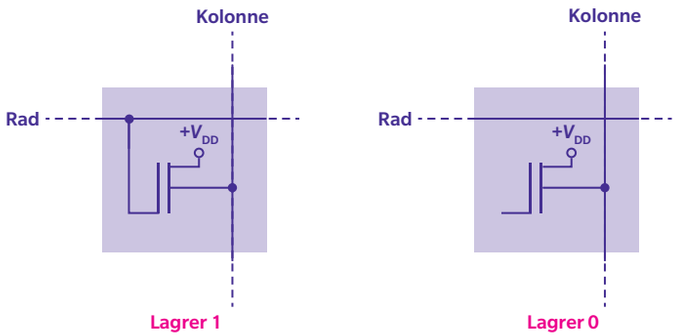
Minnecellen for DRAM er vesentlig enklere i sin oppbygging og består kun av en kondensator og en transistor. Dersom kondensatoren er oppladet, representerer det 1 bit, og er den utladet, betyr det et 0 bit. Lese- og skrivetilgangen til kondensatoren styres av en MOSFET transistor. Virkemåten til MOSFET transistoren er forklart i appendikskapittelet «Store tanker gir små kretser». Figur 13.4 viser en DRAM minnecelle.



Figur 13.4 DRAM minnecelle

Digitalt minne organiseres som regel i rader og kolonner. For å få tilgang til kondensatoren må raden settes til HØY. Ved også å sette kolonnen til HØY vil kondensatoren fylles med ladning, og en skriver dermed et 1 bit til minnecellen. Ved å sette kolonnen til LAV vil kondensatoren lades ut, og en skriver dermed et 0 bit til minnecellen. I begge tilfeller settes raden til LAV etter skriveoperasjonen. For å lese ut data settes rad til HØY, slik at en kan måle spenningen over kondensatoren. Er den HØY, inneholder minnecellen et 1 bit, og er den LAV, leser en ut et 0 bit. Riktignok er DRAM mye mer kompakt enn SRAM, men da kondensatorer lekker ladning, må minnecellene oppfriskes med jevne mellomrom. Hvor ofte? Med noen millisekunders mellomrom. På samme måte som for SRAM forsvinner DRAM minnet med strømmen.

Etter å ha sett på de flyktige minnecellene er det på tide med en titt på de permanente som er ROM, EEPROM og Flash. La oss starte med ROM. ROM står som sagt for Read Only Memory og brukes til å lagre data som ønskes permanent tilgjengelig, og wsom ikke skal endres. De fleste ROM bruker fravær eller tilstedeværelse av en transistorforbindelse for å lagre henholdsvis et 0 bit og et 1 bit. Figur 13.5 viser to ROM minneceller som lagrer 1 og 0.

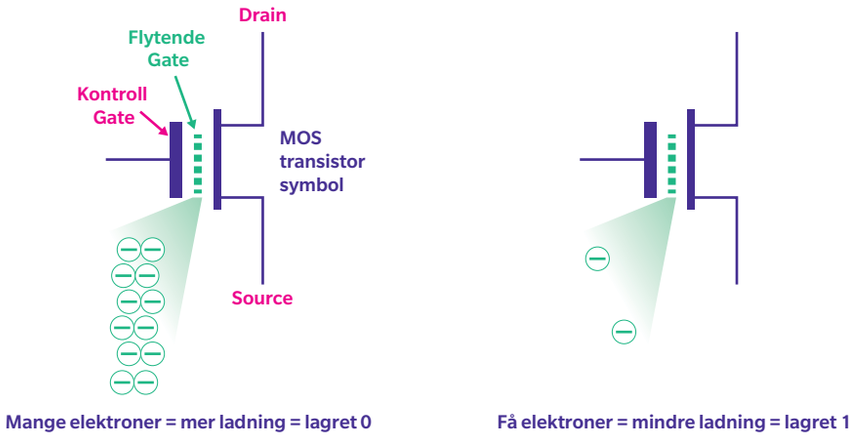


Figur 13.5 ROM minneceller

Raden settes HØY for å velge minnecellen, og verdien leses ut i kolonnen. Siden det ikke er noen forbindelse mellom rad og kolonne for et 0 bit, vil kolonnen der være LAV (=0 bit), i motsetning til for 1 bit hvor kolonnen vil bli HØY (=1 bit) når raden er satt HØY.

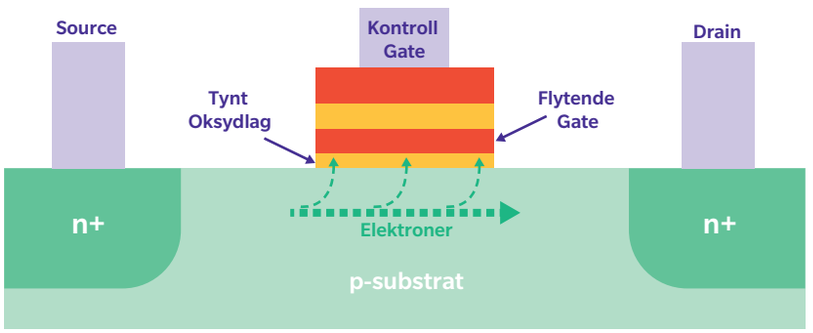
EEPROM og Flash har permanente minneceller som også kan skrives til og dermed endres. Disse minnecellene er laget med en såkalt Floating Gate MOSFET transistor. Denne MOSFET transistoren har i tillegg til den vanlige strukturen en ekstra «flytende» Gate som er elektrisk isolert. Når denne flytende Gaten er fylt med elektroner,

lagrer transistoren et 0 bit. I motsatt fall, når det er få elektroner i den flytende Gaten, lagrer transistoren et 1 bit. Figur 13.6 viser Floating Gate MOSFET som lagrer henholdsvis 0 og 1.



Figur 13.6 EEPROM og Flash minneceller

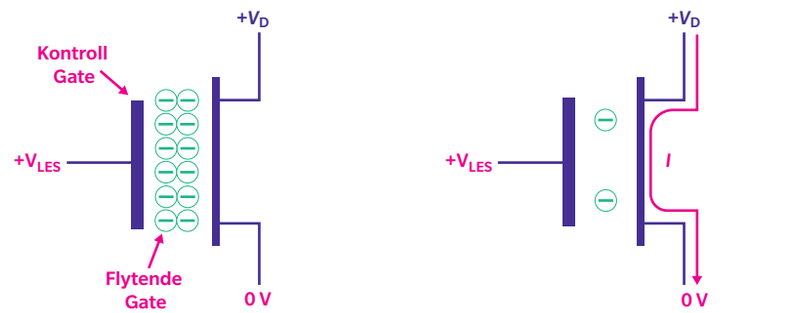
Spørsmålet som melder seg, er hvordan en klarer å variere mengden elektroner i den flytende Gaten når den er isolert. For å kunne svare på det må en ta kvantemekanikkens tunneleffekt i bruk. Frykt ikke! Du vil ikke bli utsatt for Schrödingers ligning, men en mer folkelig forklaring. I en vanlig MOSFET er det et oksydlag mellom Gate og substratet som skiller Source og Drain. I Floating Gate varianten er den flytende Gaten lagt i det isolerende oksydlaget, men veldig tett mot substratet, som vist i figur 13.7.



Figur 13.7 Floating Gate MOSFET

De tre basisoperasjoner for Floating Gate MOSFET minneceller er programmering, lesing og sletting. I utgangspunktet før programmering av en minnecelle antar en at den er i 1 bit tilstanden, da elektroner er blitt fjernet i en sletteoperasjon. Ved å ha en tilstrekkelig høy positiv Gatespenning relativt til Source vil en del av elektronene som egentlig skulle gått til Drain som har en spenning $+V_D$, tunellere til den flytende Gaten. Mange elektroner i den flytende Gaten gir et 0 bit. For å programmere 1 bit trenger man ikke å gjøre noe som helst.

For å lese av en gitt bit-verdi tilfører man en positiv spenning på Gaten som er vesentlig mindre enn når en programmerer Floating Gate MOSFET minnecellen. Dersom det er et 0 bit som er programmert, vil elektronene i Floating Gaten skjerme for lesespenningen på Gate, slik at det ikke går noen strøm fra Source til Drain. I motsatt tilfelle, når har et 1 bit, er det få elektroner i Floating Gate, og det vil gå strøm fra Source til Drain. På denne måten kan en altså lese av minnecellens lagrede verdi.



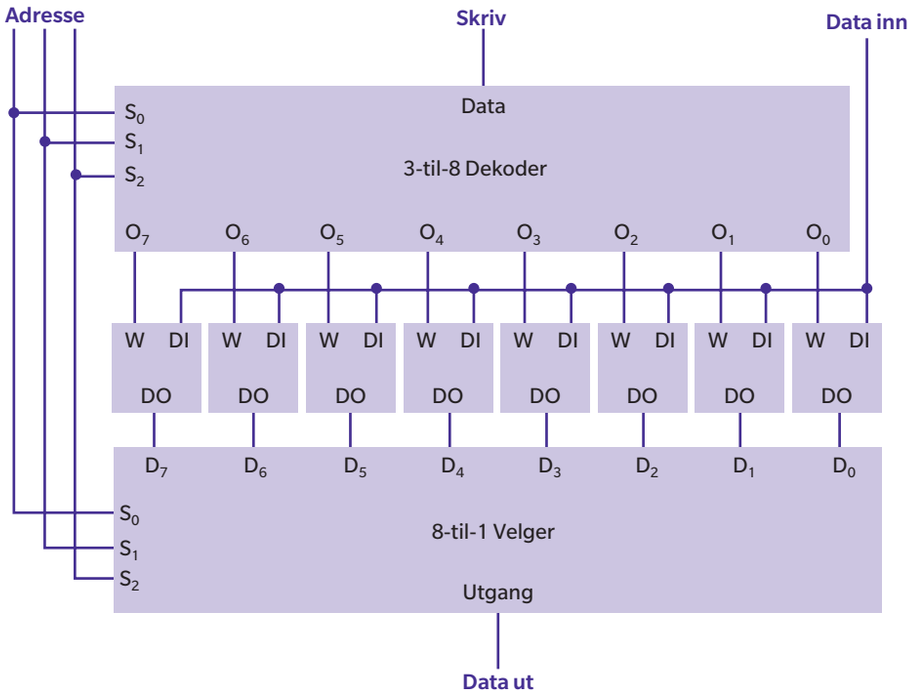
Når 0 leses, skrus transistoren av, fordi ladningene på den flytende Gate forhindrer lesespenningen fra å bli større enn skru på terskel.

Når 1 leses, skrus transistoren på, fordi mangel på ladninger på den flytende Gate tillater lesespenningen å bli større enn skru på terskel.

Figur 13.8 Lesing Floating Gate MOSFET minnecelle

For å slette innholdet fra en Floating Gate MOSFET minnecelle gis Drain en positiv spenning mens Gate settes til 0 volt. Elektronene vil dermed forlate den flytende Gaten, og minnecellen er klar til ny bruk.

Nå har vi sett på hvordan et bit kan lagres. Hva må til for å lagre flere bit? Vel, vi må sette sammen flere 1 bit minneceller. Først må en bestemme ordlengden som gir hvor mange bit minneceller som skal til for å lagre ordet. Ønsker en for eksempel å bruke en ordlengde på 1 byte som er åtte bit, så trengs det åtte 1 bit minneceller for å ta vare på ordet. I tillegg ønsker en som regel å ta vare på mange ord, og derfor må det enkelte ord kunne adresseres ved skriving og lesing. I figur 13.9 er det gitt et eksempel på et adresserbart minne med åtte minneceller hvor ordlengden er 1 bit.

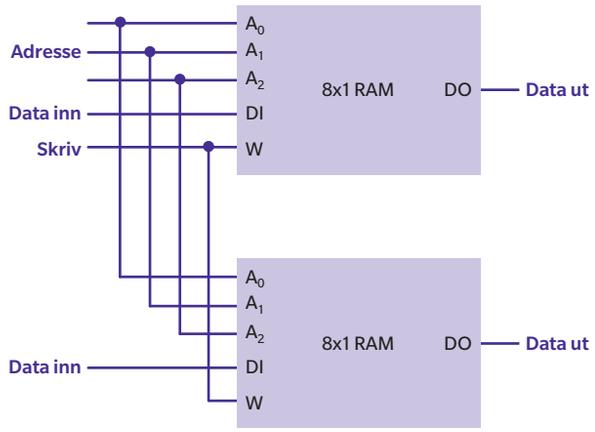


Figur 13.9 Adresserbart 1 bit minne

Dersom en ønsker å skrive til dette minnet, angir en adressen ved å sette en kombinasjon av S_0, S_1 og S_2 HØY eller LAV. I tillegg må Skriv være HØY. Er for eksempel $S_0 = 1,\ S_1 = 0,\ S_2 = 1,$ så vil O_5 bli valgt av 3-til-8 dekoderen og settes HØY, og dermed kan Data inn skrives til celle nummer 5. Dersom data skal hentes fra minnet, gis adressen til den ønskede celle, og 8-til-1 selektoren vil velge den rette cellen og data vil bli tilgjengelig på Data ut porten. På denne måten har vi laget oss et $8x1$ RAM minne som forenklet kan tegnes som vist i figur 13.10.

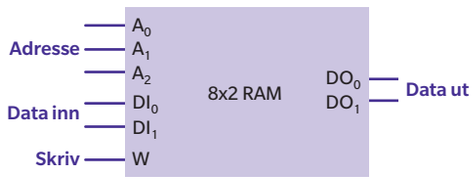
Figur 13.10 8x1 RAM minne

Nå har vi laget oss en ny byggekloss som kan settes for enten å utvide ordlengde eller for å skaffe oss flere ord. I figur 13.11 økes ordlengden til 2 bit.



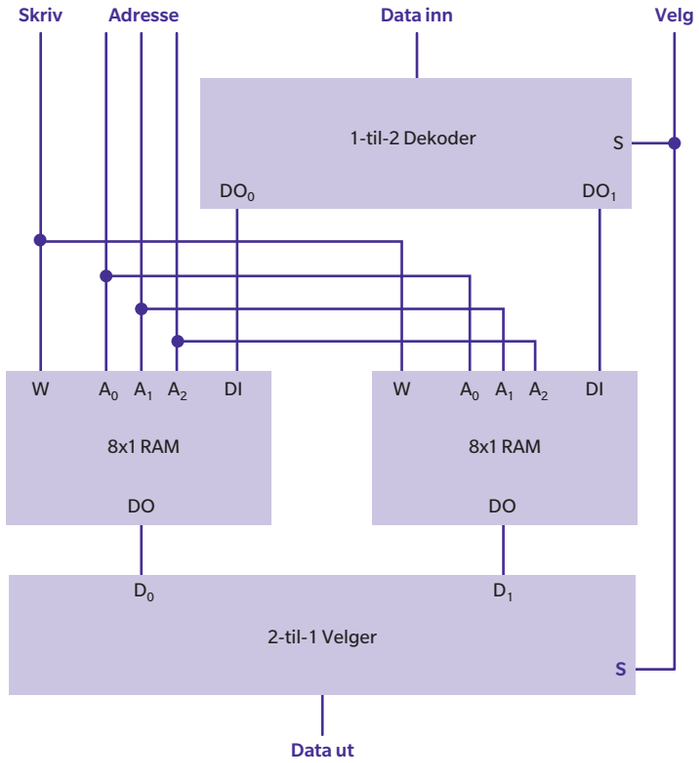
Figur 13.11 Adresserbart 2 bit minne

Figur 13.11 kan forenklet fremstilles som et $8x2$ RAM minne som vist i figur 13.12.

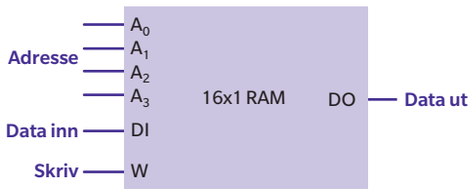


Figur 13.12 8x2 RAM minne

For å øke antallet minneceller kan $8x1$ RAM minne kobles sammen som vist i figur 13.13. 1-til-2 dekoderen blir brukt til å velge hvilken $8x1$ RAM det skrives til, og 2-til-1 velgeren hvilken 8x1 RAM det skal leses fra. Velg blir dermed å anse som en ny adresseringsmulighet. I $16x1$ RAM som er vist i figur 13.14 er Velg satt lik A_3, og en har dermed doblet adresseringsmuligheten.

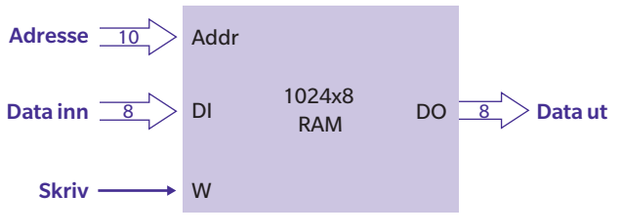


Figur 13.13 Dobling av minne



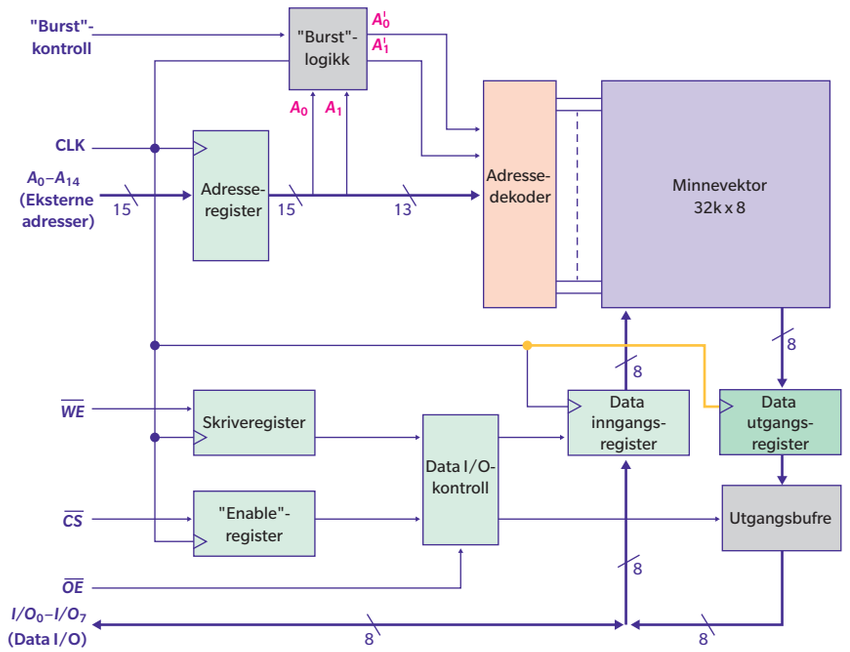
Figur 13.14 16 *x*1 RAM minne

Det skal ikke mye fantasi til for å tenke seg at disse to prosessene kan en fortsette så lenge en vil for å skaffe seg den nødvendige ordlengde og ønsket antall ord. Figur 13.15 viser for eksempel et minne som inneholder 1024 ord (ti adresselinjer $2^{10} = 1024$) hvor ordene er 1 byte lange. Dette minnet har en totalkapasitet på en kilobyte under mottoet at $1024 \approx 1000$.



Figur 13.15 1024*x*8 RAM minne

Den minneoppbyggingen vi har gjort så langt, har vært såkalt asynkront minne. Dersom en ønsker raskt minne som skal operere sammen med for eksempel en mikroprosessor, trengs det synkront minne styrt av systemets klokkepuls. Synkront minne er som regel raskere enn asynkront på grunn av arkitekturen bak. Figur 13.16 viser et eksempel på et slikt synkront minne.

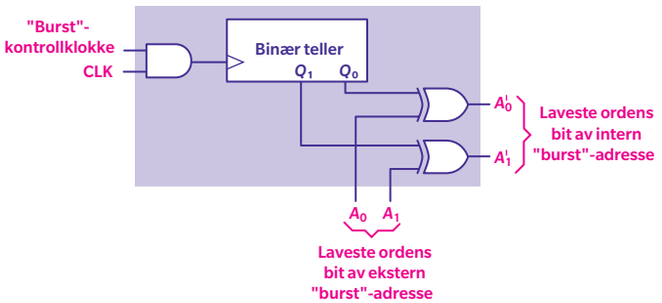


Figur 13.16 32*kx*8 synkront SRAM minne

Dette minnet er på 32 kilobyte, og de litt fetere linjene med skråstrek og et tall representerer et sett med parallelle linjer hvor antallet er bestemt av det gitte tallet. Vi ser

at alle registrene er styrt av klokken. Adressen $A_0,\ldots,A_{14}$ blir lastet inn i adresseregisteret ved den positive flanken til klokkepulsen. På samme klokkepuls lastes «Write Enable» ($\overline{WE}$) og «Chip Select» ($\overline{CS}$). $\overline{WE}$ og $\overline{CS}$ sammen med «Output Enable» ($\overline{OE}$) styrer om det skal leses eller skrives fra minnet via Data I/O linjen etter at data er gjort tilgjengelig i Data input eller Data output registeret.

En rekke SRAM-minner har «burst» logikk som gjør det mulig å hente ut data fra adresser rundt den oppgitte uten å adressere dem spesifikt. Rasjonalet bak dette er at en ofte ønsker å hente data som ligger på samme sted i minnet. Hvordan virker så «burst» adresselogikken? Figur 13.17 gir en skjematisk fremstilling.



Figur 13.17 «Burst» adressering

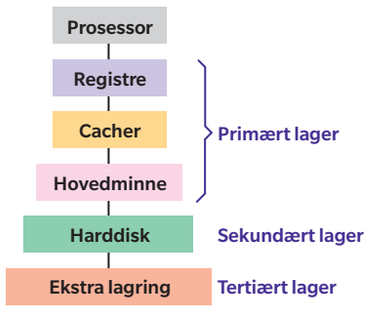
Som en ser, brukes det to XOR-funksjoner som henter bidrag fra de to minst signifikante adressebitene A_0 og A_1 og resultatet fra en binær teller som starter fra 00 og ender med 11. På den måten kan en hente ut i dette tilfellet de fire nærmeste adressene til den gitte adressen uten å oppgi suksessive adresser på nytt.

Er det mulig å beskrive minne ved hjelp av VHDL? Selvfølgelig! La oss se på en liten synkron SRAM.

```
-- This is a SRAM memory

-- This is the IEEE library
library IEEE;
-- To get some useful identifiers
use IEEE.std_logic_1164.all;
-- To get the unsigned type and the + operator
use IEEE.numeric_std.all;
```

```
entity SRAM is
-- The generic statement defines here
-- address length and size of memory
-- Natural is subttype of integer and contains
-- the natural numbers 0,1,2,…
generic (M : natural := 4; N : integer := 4;);
   port (Address: in std_logic_vector(M-1 downto 0);
         Data: in std_logic_vector(N-1 downto 0);
         WE, Clock: in std_logic;
         Quot: out std_logic_vector(N-1 downto 0);
end entity SRAM;

architecture SRAMLogic of SRAM is
-- Use type to describe memory
-- 2**M means 2*2*… M times
type ram_array is array (0 to 2**M-1) of
   std_logic_vector(N-1 downto 0);
-- Mem is the signal holding the memory
signal mem: ram_array;
begin
   process (Clock)
   begin
      if rising_edge(clock) then
         if WE = '0' then
         -- Filling memory mem with address Address
         -- with Data when write is enabled (WE).
            mem(to_integer(unsigned(Address))) <= Data;
         end if;
      end if;
   end process;
   -- Content of memory mem with address
   -- Address to Qout.
   Qout <= mem(to_integer(unsigned(Address)));
end architecture SRAMLogic;
```

Som vi så innledningsvis, er både vårt eget minne og det digitale bygd opp hierarkisk. Figur 13.2 viste en typisk oppbygging av minne rundt en prosessor. Figur 13.18 viser hvordan et generelt hierarkisk digitalt minne ser ut. Inne i selve prosessoren har man en rekke raske registre som inneholder små mengder av data som er mest brukt. På neste nivå finner en «cache»-minnet, som er hurtigminnet til prosessoren. Det minnet brukes til å holde på så mye av program-minnet som mulig. Hovedminnet består av en RAM del som inneholder data som ikke blir brukt like ofte som det man finner i «cache»-minnet. I tillegg har hovedminnet en ROM del for permanent lagring av data som brukes fra tid til annen. Etter hovedminnet har man som regel en harddisk for oppbevaring av store mengder raskt tilgjengelige data. Videre har man ekstralagring enten på servere lokalt eller i den såkalte skyen, hvor den enn måtte befinne seg.



Figur 13.18 Typisk minnehierarki

14

Kapittel 14

Følg instruksen!

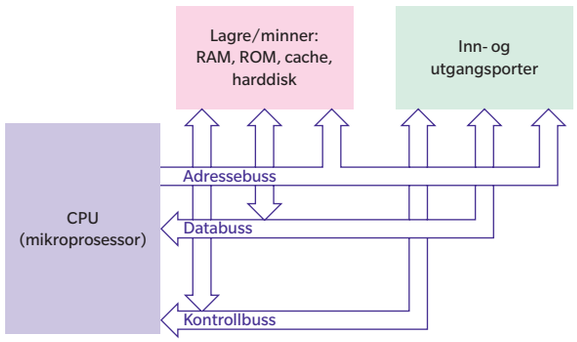
«Ved to anledninger har jeg blitt spurt; 'Unnskyld, Mr. Babbage, hvis du legger inn gale tall i maskinen vil de riktige svarene komme ut?' … Jeg er ikke i stand til å forstå hva slags forvirrede idéer som kan fremprovosere slike spørsmål.»

Passages from the Life of a Philosopher, Charles Babbage (1791–1871)

LÆRINGSUTBYTTE: Datamaskin, mikroprosessor (CPU Central Processing Unit), minne, I/O porter, buss, ALU (Arithmetic Logic Unit), kontrollenhet, registre, statusflagg, BIOS (Basic Input/Output system), operativsystem, maskinspråk, instruksjonskode, programteller, instruksjonsteller, datateller, akkumulator, «fetch» - «execute», instruksjonssyklus, avbrudd, avbruddsvektor, DMA (Direct Memory Access), «assembly», label, mnemonics, operand, «assembler», høynivåspråk, kompilator, mikrokontroller, «embedded» systemer, SoC (System on Chip)

Jeg sitter med føttene godt plantet i det grå slapset som dekker linoleum på Sognsvannstrikken. Ved den høyre foten ligger det en liten lapp. En handleliste? En kulinarisk beskrivelse av kommende helg? Jeg bøyer meg ned og tar den opp. Jeg leser – 9C 0A 30 40 9C 0A 31 80 60. Ikke akkurat mat for Mons. Hvem skriver sine ønsker i kode? En vordende programmerer, kanskje? Ukeslutt under mottoet «Garbage in, garbage out».

Datamaskiner er seg selv lik. Enten det er snakk om bygningstore tallknusere eller smarttelefoner, er den grunnleggende oppbygging den samme. I figur 14.1 er det vist et generelt blokkskjema for en datamaskin.

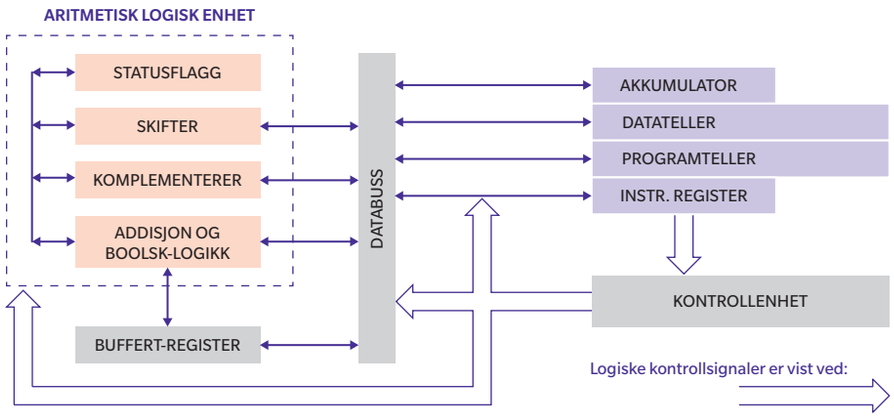


Figur 14.1 Blokkskjema for en datamaskin

Mikroprosessoren (CPU «Central Processing Unit») er selve hjernen. I tillegg finnes det minne i form av RAM, ROM, cache og harddisker og mulighet for å kommunisere med omverdenen via I/O porter. Alle enhetene er knyttet sammen ved hjelp av såkalte busser. Størrelsen, antall ledere, på disse bussene er bestemt av behovet. Har en for eksempel 16 linjer for adressering, kan en adressere opptil 2^{16} lokasjoner. Databussen er bestemt av ordlengde og er typisk 64 linjer for moderne mikroprosessorer. Kontrollbussen brukes til å formidle kontrollsignaler som skriv, les, avbrudd (interrupt) med mer.

Her er det på plass med et lite avbrudd før vi fortsetter med CPU-en. I den videre fremstillingen skal vi gå i Adam Osbornes fotspor (Innføring i mikrodatamaskiner [8] side 45 og utover). Osborne var ingeniør og forretningsmann. Han var far til den første bærbare datamaskin Osborne 1 og har den noe tvilsomme æren av å bli opphavet til den såkalte Osborne-effekten. Osborne-effekten er en moderne variant av å selge skinnet før bjørnen er skutt. Osborne gikk ut og fortalte at de hadde en ny, mindre og bedre bærbar PC enn Osborne 1 måneder før de hadde den klar. Resultatet ble at salget av Osborne 1 bråstoppet, og uten inntekter var konkursen et faktum.

I CPU-en utføres det instruksjoner, såkalte programmer. For å kunne utføre det har CPU-en fire ulike enheter. Et funksjonsdiagram for en CPU er gitt i figur 14.2. Den aritmetisk logiske enhet (ALU «Arithmetic Logic Unit») utfører beregninger og logiske funksjoner. Instruksjonsdekoderen tolker kodene gitt i instruksjonene, slik at ALU utfører de riktige operasjoner. Det finnes også en «timing» og kontrollenhet og et sett med registre for å ta vare på data og adresser som ALU trenger. Hvis vi ser nærmere på den aritmetisk logiske enheten, har den typisk fire enheter. Den viktigste er den som kan utføre addisjon og boolsk logikk. I tillegg er det en komplementerer som er kjekk å ha ved subtraksjon, en skifter som gjør det enkelt å multiplisere eller dele med et multippel av 2, og til slutt et statusflagg som brukes til å signalisere til omverdenen. Dersom en for eksempel får overflyt, kan det angis med et bit i statusflagget.



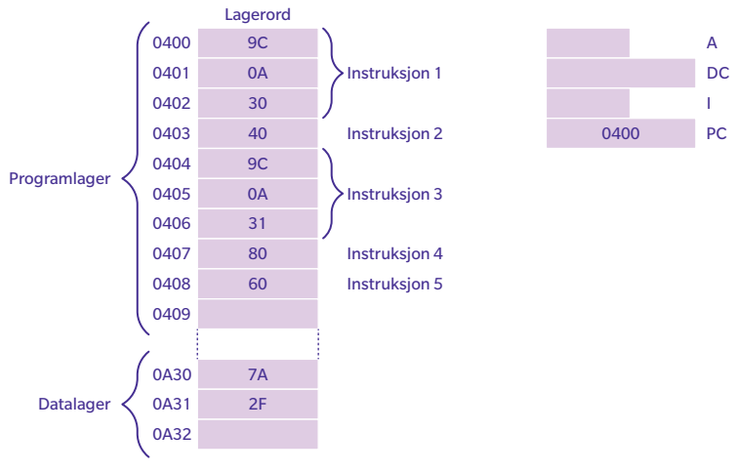
Figur 14.2 CPU funksjonsdiagram

Minne har vi jo vært borti før. Vi har sågar kjempet oss igjennom et helt kapittel: «Kun i minne finner hjertet fred». Det eneste nye her er at datamaskiner som regel har et minne for grunnleggende inn- og utsystem (BIOS «Basic Input/Output System»). BIOS inneholder lavnivåkode. Det brukes ved oppstart for selvtest, initalisering av drivere og start av operativsystemet. Datamaskinens operativsystem (OS «Operating System») er et program som tilrettelegger for andre programmer slik at de får tilgang til datamaskinens ressurser via funksjonskall/prosesser i operativsystemet.

I/O portene gjør det mulig for datamaskinen å transportere data inn og ut. Det er to ulike måter CPU-en kan bruke I/O porter på. Enten kan data som skal inn og ut skrives inn («mappes») i minnet, slik at en I/O port for en CPU ser ut som en hvilken som helst minnelokasjon, eller så har CPU-en egne pinner (porter) og programkoder for I/O.

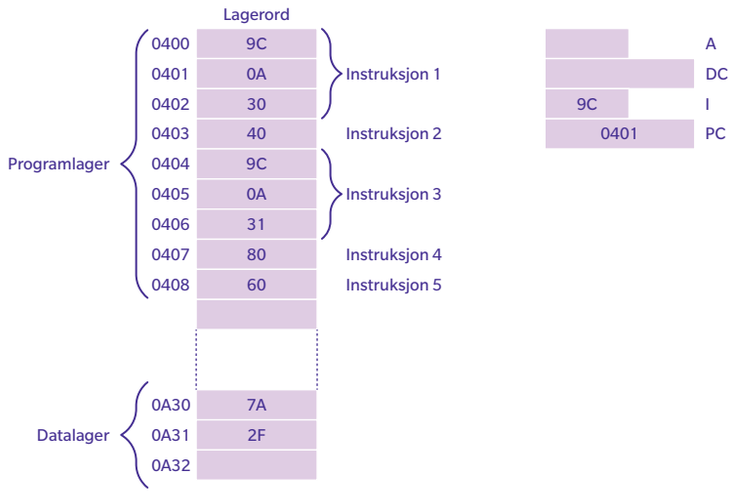
La oss gå tilbake til handlelappen jeg fant på Sognsvannstrikken. Den har jeg hatt som et minne om studietiden i alle år. Grunnen til at jeg antok at den var et

mikroprosessorprogram skrevet i maskinspråk, var lengden og bruken av heksadesimale tall. I figur 14.3 ser du deler av lappen til venstre og et CPU register til høyre.



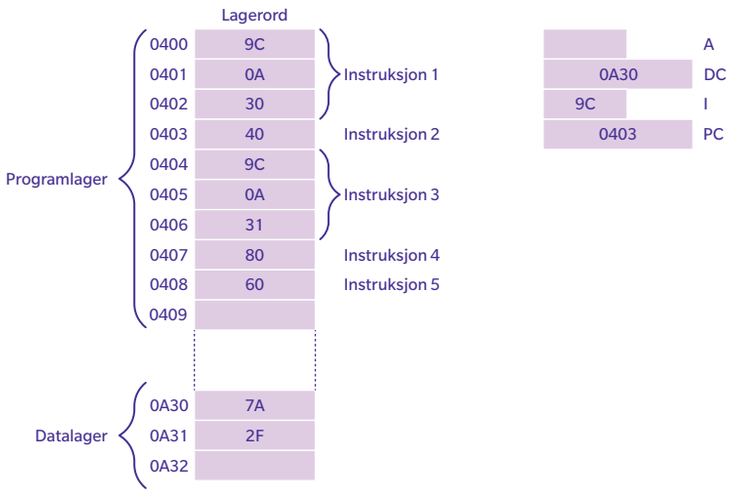
Figur 14.3 CPU programstart

Programtelleren (PC «Program Counter») er fylt med adressen til lagerordet (minnecellen) hvor programmet skal starte. I dette tilfellet 0400. Hvordan kan CPU-en vite at dette lagerordet inneholder en instruksjonskode? Egentlig inneholder jo alle lagerordene kun data. Vel, siden vi begynner her, så må CPU-en tolke det som instruksjonskode. Hadde vi derimot kommet i skade for å starte med lagerord 0401, ville det ha gått hakka gale, da data hadde blitt tolket som instruksjonskode. CPU-en henter innholdet i dette lagerordet 0400 og putter det, en kopi altså, i instruksjonstelleren I samtidig som programtelleren økes med en til 0401. Se figur 14.4.



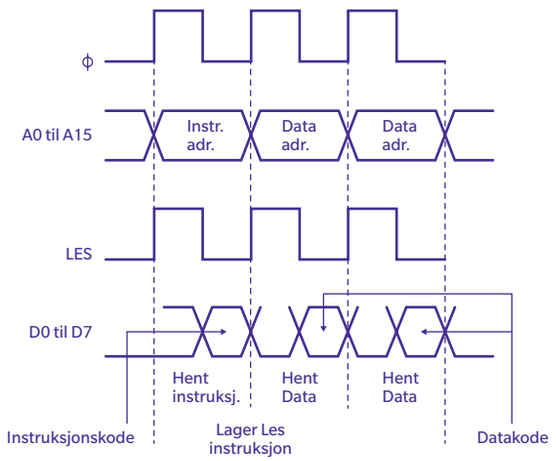
Figur 14.4 CPU innlesning av instruksjonskode 9***C***

Instruksjonskoden *9C* sørger for at adressen til lagerordene som er gitt av de to neste bytene i programlageret 0401 og 0402, blir lest inn i datatelleren (DC «Data Counter») i en totrinnsprosess. Hvorfor? Vel, fordi vi har definert at *9C* betyr å gjøre akkurat det. Dermed er instruksjon 1 gjennomført, og registrene til CPU-en er fylt slik som figur 14.5 viser.



Figur 14.5 CPU etter instruks 1 er utført

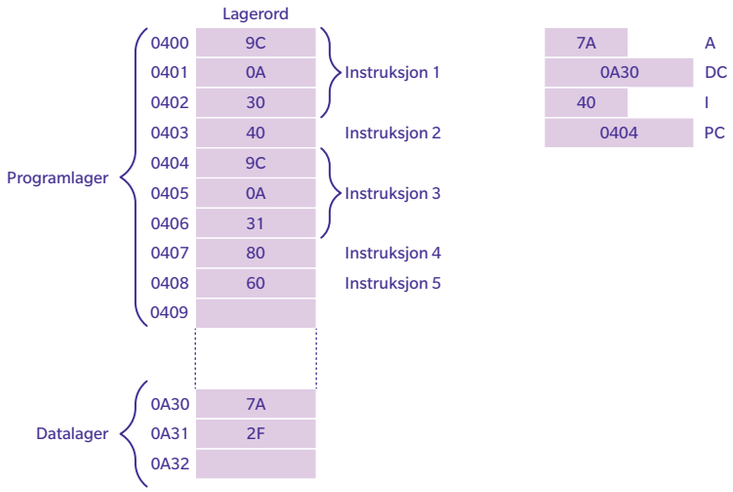
Hva er det som styrer gjennomføringen av en instruksjon? Det er klokken. Det brukes en klokkeperiode til å hente, «fetch», en instruksjon og en eller flere påfølgende perioder for å utføre instruksjonen «execute». For *9C* instruksen ville det se ut som i figur 14.6.



Figur 14.6 Instruksjonssyklus

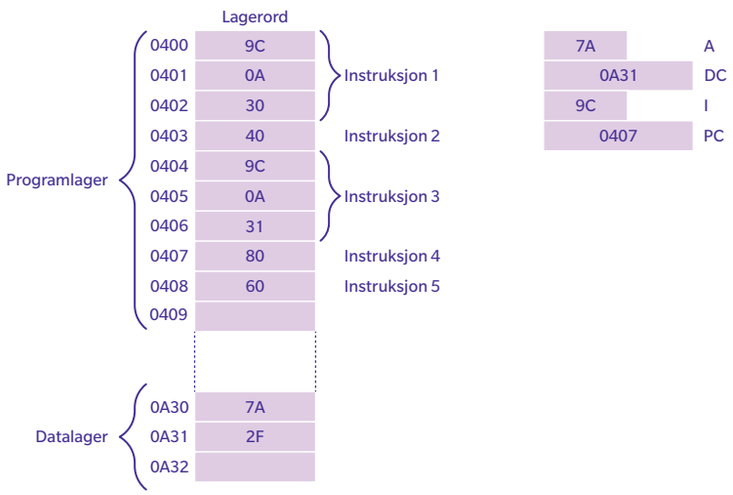
Her angir ϕ klokken, A0 til A15 adresselinjer og D0 til D7 datalinjer. Avlesing av data fra D0 til D7 skjer først etter at adressene A0 til A15 har stabilisert seg etter en viss tid.

Så er det på'an igjen. Neste er instruks 2. Den leses inn i samtidig som programteller-leren økes med en. Instruksjonskoden 40 sørger for at innholdet av datalageret som datatelleren peker på, leses inn i akkumulatoren. Når instruksen er utført, ser det ut som vist i figur 14.7.



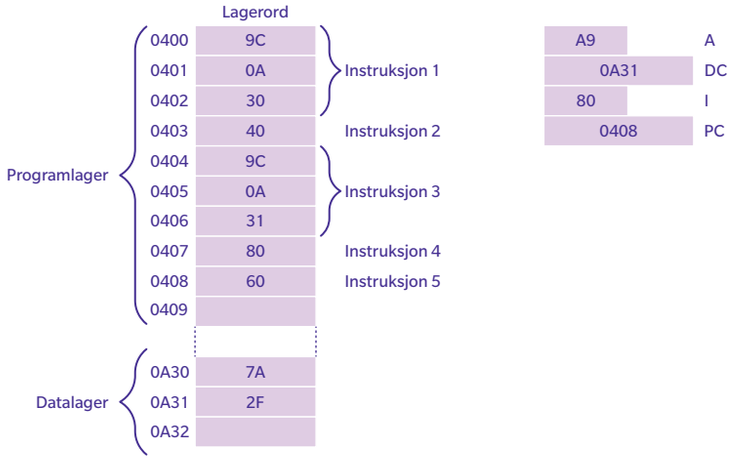
Figur 14.7 CPU etter instruks 2

Instruks 3 som vi finner i programlager 0404, er jo en gammel kjenning, nemlig 9C. Da er det bare å repetere allerede gammel kunnskap, og etter noen programtrinn ser det ut som vist i figur 14.8.



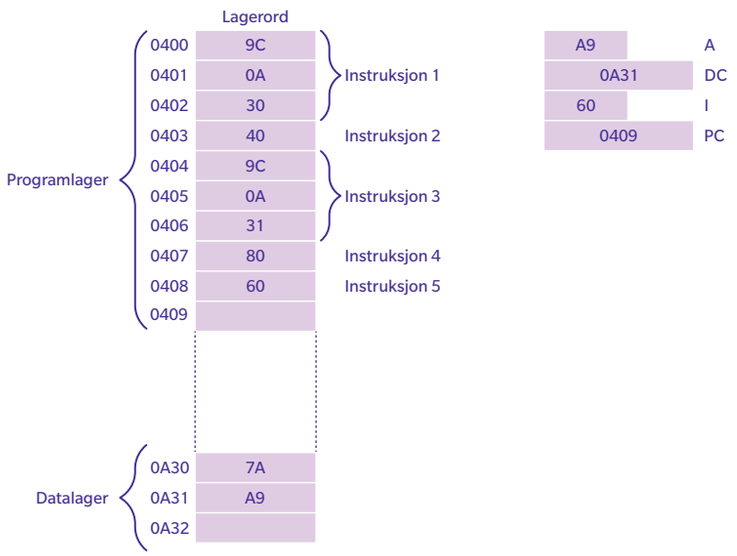
Figur 14.8 CPU etter instruks 3

Så er det instruks 80 fra programlager 0407. Denne instruksen sier at data fra datalageret med adresse gitt av datatelleren skal adderes til akkumulatoren. I akkumulatoren hadde vi $7A$, og så hentes $2F$ fra $0A31$, og det hele adderes $7A + 2F = A9$. Figur 14.9 viser hvordan det ser ut etter at instruks 4 er utført.



Figur 14.9 CPU etter instruks 4

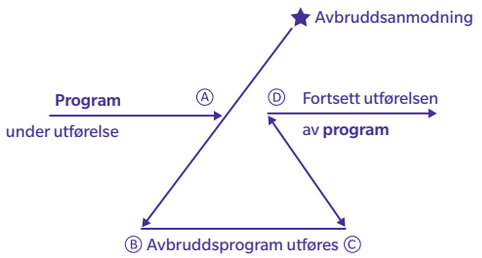
Da er det bare en instruks igjen. Instruks 5 har koden 60 som betyr at det som er i akkumulatoren skal skrives til adressen gitt i datatelleren. I vårt tilfelle vil vi overskrive den verdien vi har der, $2F$, med resultat i akkumulatoren som er $A9$. Når programmet er ferdig kjørt, vil registre og datalager se ut som gitt i figur 14.10. For ordens skyld må det nevnes at de instruksjonskodene vi har brukt i dette eksempelet er litt tilfeldig valgt. De varierer mellom ulike mikroprosessorfamilier.



Figur 14.10 CPU og datalager etter endt program

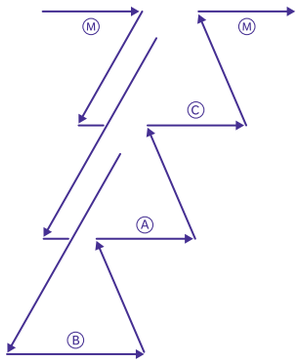
Dette var da omstendelig, kan du kanskje si. Masse data og instruksjoner for kun å legge sammen to tall. Det ville jo være en umenneskelig oppgave å lage et håndskrevet binært program for å styre en hel datamaskin. Heldigvis finnes det hjelp å få. Det finnes programmeringsspråk på ulike abstraksjonsnivåer som kan brukes for å styre datamaskiner. Disse språkene omformes til sekvenser av binære tall som så kan kjøres på CPU-en. Vi skal se nærmere på disse språkene senere. Vi må bare gjøre oss ferdig med «hardwaren» først.

Liker du å bli avbrutt? Det kan jo være en hyggelig avveksling i hverdagen, men så er det å huske hvor en slapp, når en skal begynne igjen. En mikroprosessor blir ofte utsatt for avbrudd fra sine I/O porter. Det er vanskelig å forutsi når disse avbruddene kommer, og de er som regel asynkrone av natur. Et typisk forløp er som følger. Først sender en I/O port en anmodning om avbrudd (IREQ Interrupt REQuest) på en kontroll-linje for avbrudd. CPU-en kan enten ignorere forespørselen eller akseptere den ved å sende en avbruddskvittering (IACK Interrupt ACKnowledge) til gjeldende I/O port. Så må mikroprosessoren ta vare på det den holder på med. Det vil si å oppbevare registre og statusflagg for programmet som kjøres. Deretter hentes en såkalt avbruddsvektor fra I/O porten. Den spesifiserer adressen til starten av avbruddsprogrammet som skal kjøres. Når avbruddsprogrammet er ferdigkjørt, fylles registre og statusflagg med verdiene som ble tatt vare på fra programmet som kjørte før avbrudd.



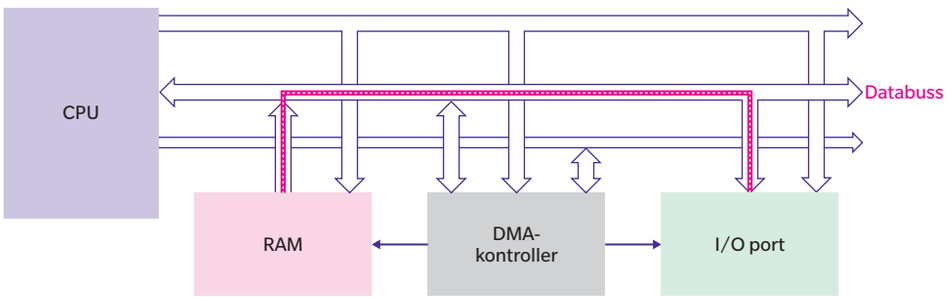
Figur 14.11 Avbruddshåndtering

Hva vil skje dersom det kommer flere påfølgende avbrudd? Dersom CPU-en har flere kontroll-linjer for avbrudd, kan det lages et system for avbruddsprioritering, slik at avbruddene kan kjøres i sekvens slik som vist i figur 14.12. M er hovedprogrammet som kjøres, og så blir avbruddsprogrammene B, A og C kjørt i en prioritert orden før man vender tilbake til hovedprogrammet.



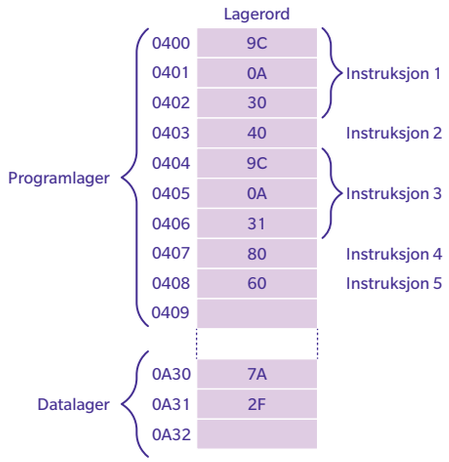
Figur 14.12 En rekke avbrudd

Det er vanlig i mikroprosessorsystemer å ha en såkalt DMA (Direct Memory Access) kontroller for rask skyfling av data mellom minneenheter og innlesning av data fra I/O porter når det trengs. Dette avlaster mikroprosessoren betraktelig, og avleste verdier fra I/O portene kan mikroprosessoren hente fra minnet når det passer. Figur 14.13 viser et system med DMA kontroller.



Figur 14.13 System med DMA kontroller

Hittil har vi programmert mikroprosessoren ved å bruke maskinspråk. For å gjøre livet vesentlig enklere for programmerere, ble det såkalte «assembly» språket utviklet. Istedenfor å fikle med heksadesimale tall ble det innført engelsklignende ord som var lettere å huske.



Figur 14.14 Program i maskinspråk

Vårt programmeringseksempel som er vist på nytt i figur 14.14 kunne ha for eksempel følgende kode som gitt i tabell 14.1:

Tabell 14.1 Program i «assembly»

Label	MNEMONICS	Operand	Kommentar
	ORG	H'0400'	; Definerer startadresse for program
ADDR1	DA	H'0A30'	; ADDR1 tilordnes lagerord 0A30
ADDR2	DA	H'0A31'	; ADDR2 tilordnes lagerord 0A31
	LIM	DC, ADDR1	; Les adresse 1 inn i DC (Instruks 1)
	LMA		; Les dataord inn i akkumulator (Instruks 2)
	LIM	DC, ADDR2	; Les adresse 2 inn i DC (Instruks 3)
	LAA		; Adder dataord til innhold i akkumulator (Instruks 4)
	SRA		; Lagre akkumulator i datalager (Instruks 5)

Labelfeltet brukes til å angi posisjoner, adresser, og i vårt tilfelle har vi brukt label to ganger for å spesifisere innholdet i adresse 1, ADDR1, og adresse 2, ADDR2. Labelfeltet kan også brukes til å spesifisere hvor enkeltlinjer er i programmet. La oss se på et litt omskrevet program som er vist i tabell 14.2.

Tabell 14.2 Lett revidert program i «assembly»

Label	MNEMONICS	Operand	Kommentar
	ORG	H'0400'	; Definerer startadresse for program
ADDR1	DA	H'0*A*30'	; ADDR1 tilordnes lagerord 0*A*30
ADDR2	DA	H'0A31'	; ADDR2 tilordnes lagerord 0*A*31
HER	LIM	DC, ADDR1	; Les adresse 1 inn i DC (Instruks 1)
	LMA		; Les dataord inn i akkumulator (Instruks 2)
	LIM	DC, ADDR2	; Les adresse 2 inn i DC (Instruks 3)
	LAA		; Adder dataord til innhold i akkumulator (Instruks 4)
	SRA		; Lagre akkumulator i datalager (Instruks 5)
	JMP	HER	; Hopp til begynnelsen av programmet (Instruks 6)

Her har vi brukt labelen HER for å identifisere hvor programmet startet (startadressen). Siste linjen i programmet (Instruks 6) ber oss hoppe (JMP) til HER. Programmet vil på den måten gå i evig sløyfe og legge til tallet i ADDR1 til det stadig voksende innhold i lagerord ADDR2, bortsett fra litt overflyt dann og vann.

MNEMONICS, som uttales nemonics, kommer fra det greske ordet for minne. MNEMONICS er forkortelser som skal hjelpe oss å huske ulike instruksjonskoder. LIM betyr i vår assembler Last Inn Minne, og SRA er forkortelse for SkRiv til Akkumulator. Hvordan disse forkortelsene ser ut, er bestemt av hvem som har laget oversetterprogrammet («Assembler»). Det finnes ulike ferdigskrevne assemblere for ulike mikroprosessorer, men det er fullt mulig å skrive sin egen dersom en har lyst til det. Det er en 1 til 1 sammenheng mellom «Assembly» og maskinspråk. Fordelen med å skrive i «assembly» er at det går fortere, og at risikoen for å gjøre feil er mindre.

Operand feltet angir de verdier eller adresse som instruksjonen gitt i MNEMONICS skal bruke. La oss se på en tilfeldig linje i assemblerkoden vår. LIM DC, ADDR1 forteller at LIM (Last Inn Minne) skal hente inn verdier med ADDR1 og legge det i datatelleren (DC Data Counter). Til slutt har en et kommentarfelt som starter med; og som det oppfordres til å bruke flittig. Ofte kan det være vanskelig å forstå egen ukommentert kode etter en stund, og andres kan fort bli rene hodepinen.

De fleste mikroprosessorer har en rekke instruksjonskoder med tilhørende MNEMONICS. Instruksjonene kan grupperes etter funksjonalitet. Noen står for datatransport, andre for beregning (aritmetikk), bitmanipulering, løkker og hopp, avbrudd og prosesskontroll.

I datamaskinenes spede barndom var «assembly» mye brukt, da det resulterte i rask og minne-effektiv kode. Etter hvert som det ble kapasiteten til programmererne istedenfor datamaskinenes begrensninger som ble problemet, dukket de såkalte høynivåspråk opp. Først ute var FORTRAN («FORmula TRANslation»), og senere kom ALGOL («ALGOritmic Language») og en rekke språk med utgangspunkt i sistnevnte. Morsmålet mitt og dermed den første og største kjærlighet var det norskutviklede SIMULA som var verdens første objektorienterte programmeringsspråk. I dag er tilnærmet alle høynivåspråk objektorienterte. Hva fremtiden vil bringe, skal bli spennende å se.

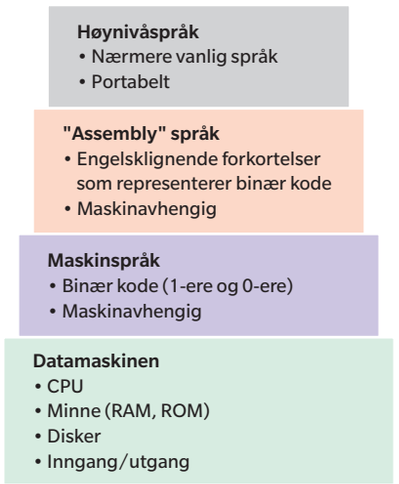
Fordelen med høynivåspråk er at en kan konsentrere seg om algoritmer og objekter til oppgaven en ønsker å løse. Hvordan det blir seende ut i maskinspråk og hvor det havner i maskinen, blir revnende likegyldig. Hvordan ville så vår maskinkode se ut i et høynivåspråk? Vi kan egentlig bruke et hvilket som helst høynivåspråk. Likheten mellom dem er slående. Størst forskjell er det i syntaksen. La oss velge C++ da det er et ekteføtt barn av ALGOL og SIMULA. Vårt eksempel ville da kunne se slik ut:

```
#include<iostream>
using namespace std;
// Dette er en kommentar og void main()
// forteller at her kommer det et program.
// ; avslutter en linje med kode
void main()
{ // { angir begynnelsen på en blokk
    int a; // Deklarasjon av variabelen a som er
           // av type integer
    a = 0x7A; // a tilordnes verdien 7A.0x forteller
              // at verdien er angitt heksadesimalt
    a = a + 0x2F; // a tilordnes verdien som
                  // allerede er i a pluss 2F
} // } angir enden på en blokk
```

Det var det. Det burde kanskje vært med en cout << a << endl; på slutten for at vi skulle fått gleden av å se sluttresultatet, men da hadde vi jo gjort mer enn vår opprinnelige maskinkode. For å oversette denne programsnutten til maskinspråk

trengs en kompilator. I motsetning til «assembly» er det ikke en 1 til 1 sammenheng med maskinspråk, og koden vår må leses av et kompilatorprogram som oversetter til kjørbar maskinkode.

For å oppsummere de ulike nivåer og egenskaper for de ulike språk er det gjengitt et hierarki i figur 14.15.

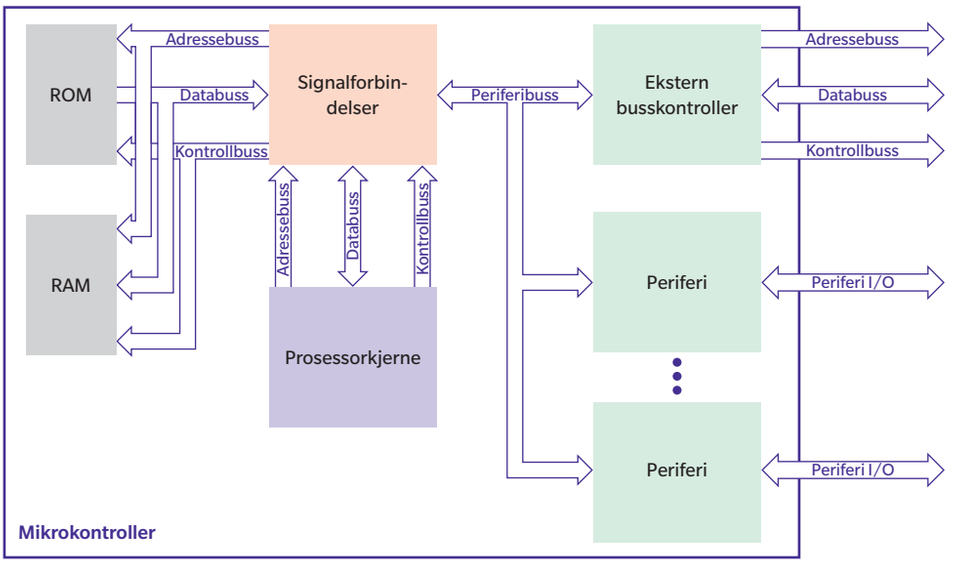


Figur 14.15 Hierarki av programmeringsspråk

Inntil nå har vi sett på en generell datamaskin og hvordan den kan programmeres. Det finnes såkalte mikrokontrollere, µC blant venner, som kombinerer mikroprosessor, minne og ulike periferienheter i en pakke. Disse mikrokontrollerne brukes ofte i såkalte embedded systemer som utfører et sett med spesialiserte funksjoner. Noen blant mange eksempler på sånne systemer er mobiltelefoner, kalkulatorer, automatiserte systemer og roboter. En typisk mikrokontroller inneholder følgende:

- Mikroprosessor (prosessorkjerne)
- Permanent minne for programkode og konfigurasjonsdata
- RAM for programdata, interne registre og andre data
- Periferienheter som analog-til-digital konvertere (ADCs Analog to Digital Converters), digital-til-analog konvertere (DACs Digital to Analog Converters), kommunikasjonskontrollere og I/O porter
- Interne busser
- Grensesnitt mot omverdenen

Figur 14.16 viser et forenklet blokkdiagram av en mikrokontroller.



Figur 14.16 Mikrokontroller blokkdiagram

I jakten etter så små, et par kvadratcentimeter, og kraftfulle systemer som mulig, har en gått et hakk videre og laget systemer på en «chip» (SoC «System on Chip»). SoCs har følgende funksjonelle elementer alt etter bruksområdet:

- Enkel- eller multiprosessorkjerne
- Digital Signal Prosessor (DSP)
- Grafisk prosessor (GPU Graphical Prosessing Unit)
- Minne (ROM, RAM, EEPROM, Flash)
- Analoge funksjoneer som ADC og DAC
- I/O grensesnitt
- Klokkekilder som oscillatorer eller faselåste sløyfer
- Programmerbar logikk

I figur 14.17 vises Raspberry Pi 2 som er en populær SoC.

Det er fascinerende! I løpet av få kapitler og noen mentale anstrengelser har vi beveget oss fra den boolske mengde, som er den enkleste en kan tenke seg, til ekstremt kompakte og kraftfulle logiske systemer som en SoC representerer. Neste skritt er å la den digitale verden få møte virkeligheten, hva den nå er, som er analog av natur.

Figur 14.17 Raspberry Pi 2

15

Kapittel 15

Fremskritt og tilbakesteg

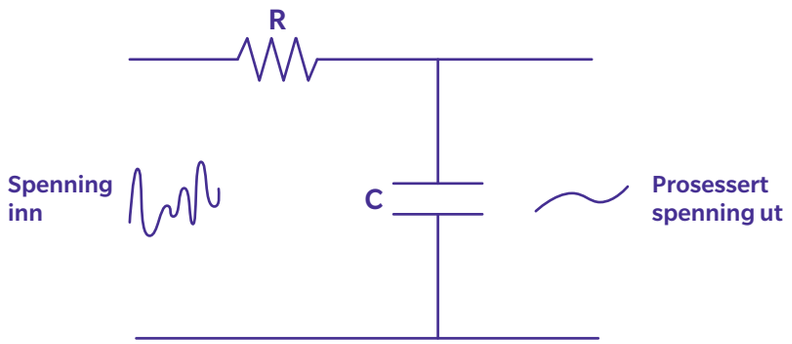
«We are analog beings living in a digital world, facing a quantum future.»

Neil Turok (1958–)

LÆRINGSUTBYTTE: Digital signalbehandling (DSP Digital Signal Processing), fordeler og ulemper med digital og analog signalbehandling, DSP system, «aliasing», Nyquistfrekvensen, «sample and hold», kvantisering, kvantiseringsstøy, SNR (Signal to Noise Ratio), ADC (Analog Digital Converter), ADC ytelse, operasjonsforsterker, ulike ADC-er som FLASH, SAR (Suksessivt Approksimasjons Register), sigma-delta (Σ-Δ), DAC (Digital to Analog Converter), binær-vektet DAC, R/2R-stige DAC, DAC karakteristikker og ytelse, rekonstruksjonsfilter, DSP, Harvardarkitektur, sanntid, digitalt lavpassfilter, DSP anvendelser

Dette er historien om modellen som trådde inn i virkeligheten. Det hele startet rundt 1940 med at en brukte datamaskiner til å beregne, analysere og simulere oppførselen til modeller av elektriske kretser. Etter hvert ble beregningene utført så raskt at lyse hoder fikk ideen om å erstatte de opprinnelig analoge kretsene med digitale. Den digitale signalbehandlingen var født.

La oss starte med et overkommelig eksempel på analog signalprosessering. I figur 15.1 er det vist et filter som slipper igjennom signaler med lave frekvenser, mens det effektivt stopper høyfrekvente signaler. Hvordan? Vel, det er kondensatoren *C* som kommer oss til hjelp. Ved lave frekvenser vil ikke ladningene over kondensatoren variere mye, og spenning ut av filter vil derfor være omtrent lik spenning inn. Ved høye frekvenser derimot vil en få spenningsvariasjoner over kondensatoren slik at den blir tilnærmet en kortslutning, og spenningen ut av filteret blir mye mindre enn spenning inn. Vi har laget oss et lavpassfilter.



Figur 15.1 Analogt lavpassfilter med motstand *R* og kondensator *C*

Hvorfor tar en seg bryet med å digitalisere analog signalprosessering? Som du skal se, krever det en hel del å digitalisere selv det enkle analoge filteret som er gitt i figur 15.1. Ulemper med analog signalprosessering kan oppsummeres slik:

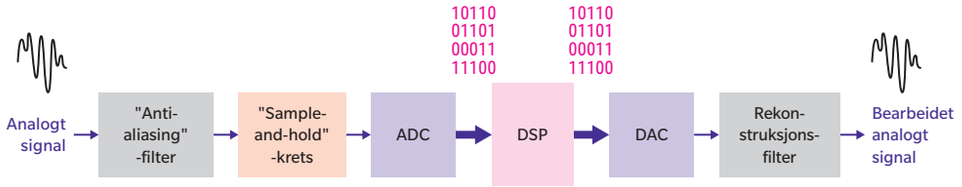
- Komponenter må endres for å forandre signalbehandlingen
- Utfordringer med repeterbarhet på grunn av komponentvariasjoner
- Aldring som endrer kretsen
- Følsomhet for temperaturforandringer

Hva er så fordelene med digital signalbehandling?

- Fleksibelt
- Programmerbart
- Repeterbart
- Fysisk stabilt

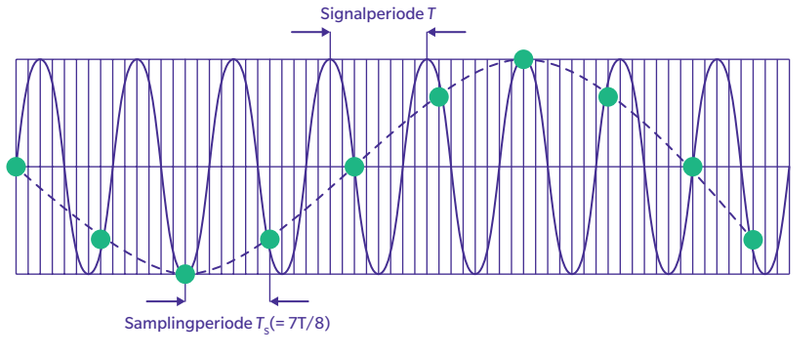
Er det verdt prisen å drive digital signalbehandling? Vel, svaret er allerede gitt i alt du omgir deg med. Digital signalbehandling finnes overalt.

Hvordan skal vi gå frem for å få laget en digital variant av dette filteret? Vi må lage oss et digitalt signalprosesseringssystem. Figur 15.2 viser hvordan dette kan realiseres, og de enkelte blokkene er svar på utfordringene vi får ved å gjøre prosessen digital.



Figur 15.2 Blokkdiagram for et typisk digitalt signalprosesseringssystem

Det ligger i ordet digitalt at vi må «oversette» de analoge verdiene til tall før systemet kan viderebehandle signalet. I den forbindelse er det tre spørsmål som reiser seg. Hvor ofte skal vi ta punktprøver, «samples», av signalet, skal det skje med regulære intervaller og hvilken oppløsning ønsker vi? Svar på spørsmål to om regulære intervaller er som regel et rungende ja, men det finnes anvendelsesområder hvor samplingshastigheten varierer over tid. Første spørsmål kan vi få svar på ved å se på punktprøving av et sinussignal med en gitt frekvens som vist i figur 15.3.



Figur 15.3 Punktprøving av et sinussignal

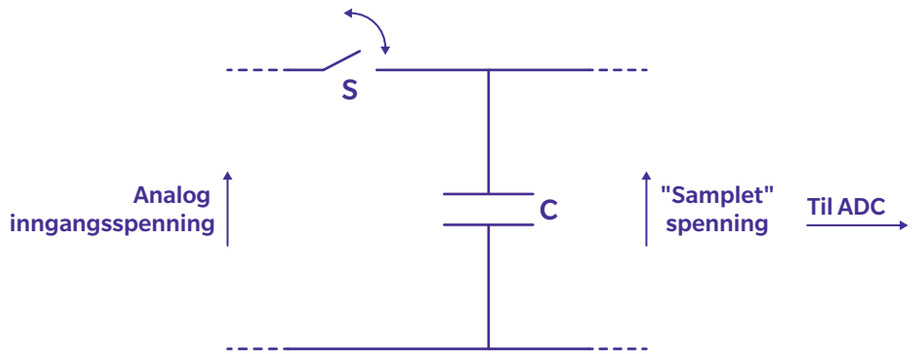
Her skjer det noe underlig. Ved å trekke en stiplet linje mellom punktprøvene vil en tolke det opprinnelige signalet som et signal med vesentlig lavere frekvens. Dette kalles «aliasing», da det ser ut som det opprinnelige signalet ser ut som et annet. Hvordan kan en reparere det? Ved å ta mye oftere punktprøver. Hadde vi for eksempel tatt punktprøver med samplingsperiode $T_S = \frac{T}{8}$ istedenfor $T_S = \frac{7T}{8}$ ville vi fått mange flere punkter langs sinussignalet og kunnet gjenskape det. Hvor går så grensen? Hvor ofte må et signal med periode T samples? Det har også folk før oss lurt på, og den svenske

ingeniøren Harry Nyquist som først fant det ut, har fått den såkalte Nyquistfrekvensen oppkalt etter seg. Skal en kunne gjenskape et sinussignal, må en ha minst to punktprøver innenfor en periode av signalet; $T_s < \frac{T}{2}$. Da sammenhengen mellom frekvens og periode for et signal er gitt ved $f = \frac{1}{T}$, vil kravet til samplingsfrekvensen bli:

$$f_s > 2f_T = 2f_{Nyquist}$$

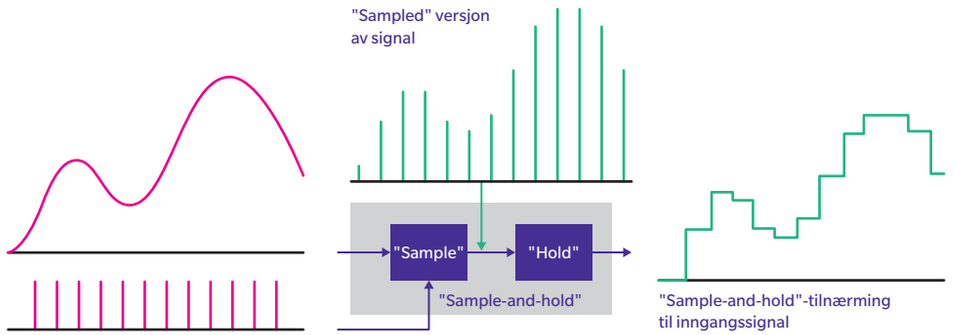
For å kunne få med oss alt signalinnholdet må det altså samples med den doble frekvensen av den høyeste analoge frekvensen. Før et signal samples, går det gjennom et anti-aliasing filter som er et lavpassfilter som fjerner frekvenser over den høyeste som vi ønsker å undersøke. På den måten unngår vi feiltolkning av uønsket høyfrekvent innhold.

Det neste som må utføres, er å avlese den analoge inngangsverdien. Det foretas i en «sample and hold» krets som i prinsippet er et lite nettverk med en kondensator og en bryter hvor inngangen har lav impedans (hurtig ladning), og utgangen har høy impedans (rolig utladning). Figur 15.4 viser en slik krets. Når bryteren lukkes, fylles kondensatoren med ladning slik at spenningen blir lik inngangsspenningen. Deretter åpnes bryteren igjen, og spenningen avleses på utgangen og leveres videre til en analog til digital konverter (ADC «Analog to Digital Converter»).



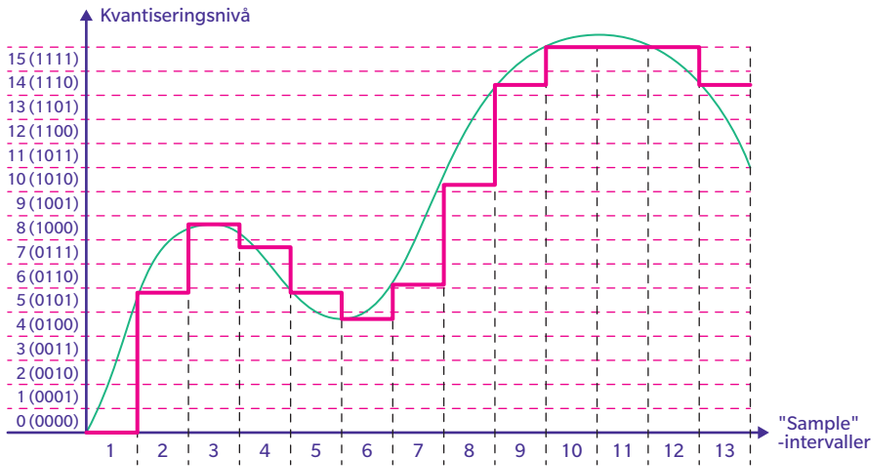
Figur 15.4 «Sample and hold» krets

Figur 15.5 viser signalomforming som foregår i en «sample and hold» krets.



Figur 15.5 «Sample and hold» operasjon

Valget av ADC er diktert av to ting: hvilken oppløsning vi ønsker, og hvor fort prosesseringen skal foregå. La oss ta det første først. I figur 15.6 er kvantiseringsnivåene for en 4 bit ADC vist sammen med det analoge signalet og «sample and hold» utgaven. Det analoge signalnivået i starten av «sample and hold» perioden vil bli tilordnet den digitale verdien som er nærmest det gitte signalnivå.



Figur 15.6 Kvantiseringsnivå for en 4-bit ADC

Siden den digitale verdien ikke vil være eksakt lik den opprinnelige analoge signalverdien, kan forskjellen mellom dem anses som støy. Jo bedre oppløsning, flere bit, en ADC har, jo mindre vil denne kvantiseringsstøyen bli. Dersom en ser på et sinussignal, vil den midlere signal til støy effekt (SNR «Signal to Noise Ratio») målt i desibel være gitt ved:

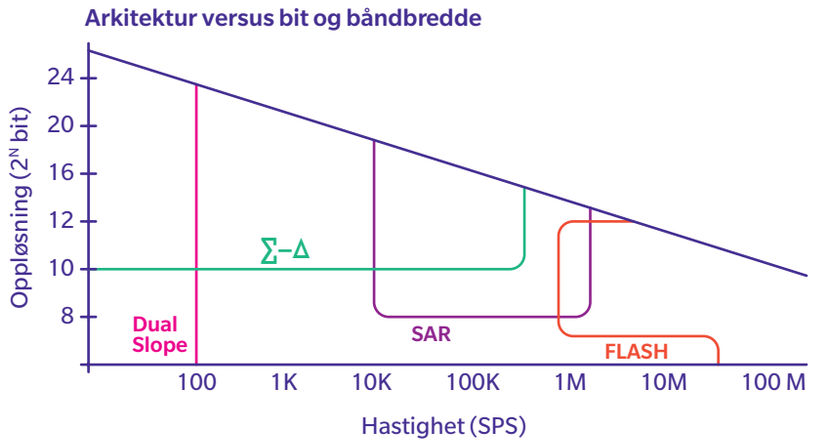
$$SNR_{\mathrm{dB}} = \left(6{,}02N + 1{,}76\right)\ \mathrm{dB}$$

Her er N antall bit i ADC-en. I vårt tilfelle var $N=4$ og $SNR=(6{,}02\cdot 4+1{,}76)\,\text{dB}=25{,}8\,\text{dB}$. Det er jo ikke rare greiene, men trøsten får være at for hvert ekstra bit vi er villige til å spandere, så øker SNR med 6 dB. Desibel (dB) angir et forholdstall, og når verdien er 25,8, betyr det at det gjennomsnittlige spenningsnivå på et signal vil være 19,5 ganger større enn støyen.

$$SNR_{dB}=10log_{10}\left(\left(\frac{V_{signal}}{V_{støy}}\right)^2\right)\text{dB}$$

$$\left(\frac{V_{signal}}{V_{støy}}\right)=10^{\left(\frac{SNR_{dB}}{20}\right)}=10^{\left(\frac{25{,}8}{20}\right)}=19{,}5$$

Når en har valgt ønsket oppløsning på sin ADC, må man finne ut hvilken frekvens den skal gå med. Minimum vil jo være Nyquistfrekvensen som var den doble av det høyeste frekvensinnholdet i det analoge signalet. Det finnes en rekke ulike analog til digital-konvertere som har sine fordeler og ulemper. I figur 15.7 ser en ulike bruksområder for ADC som «Dual slope», Σ-Δ, SAR og FLASH. Noen har mulighet for høy oppløsning, mens andre særpreges av at de er raske. Valg av ADC vil altså være diktert av anvendelsen.

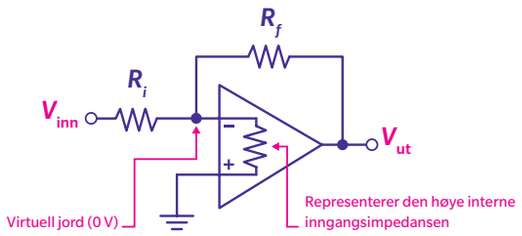


Figur 15.7 Ulike ADC-er og deres ytelse

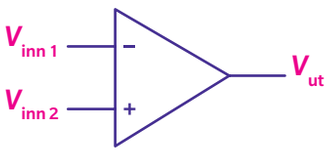
Da alle ADC-ene bruker operasjonsforsterker, op-amp, er det naturlig å starte med en rask oppsummering av operasjonsforsterkerens egenskaper. En op-amp er en lineær forsterker med to innganger og en utgang. Den har høy forsterkning og

inngangsimpedans. I figur 15.8 er operasjonsforsterkeren koblet slik at den er en inverterende forsterker hvor forholdet mellom inngangsspenning og utgangsspenning er gitt ved:

$$\frac{V_{ut}}{V_{inn}} = -\frac{R_f}{R_i}$$

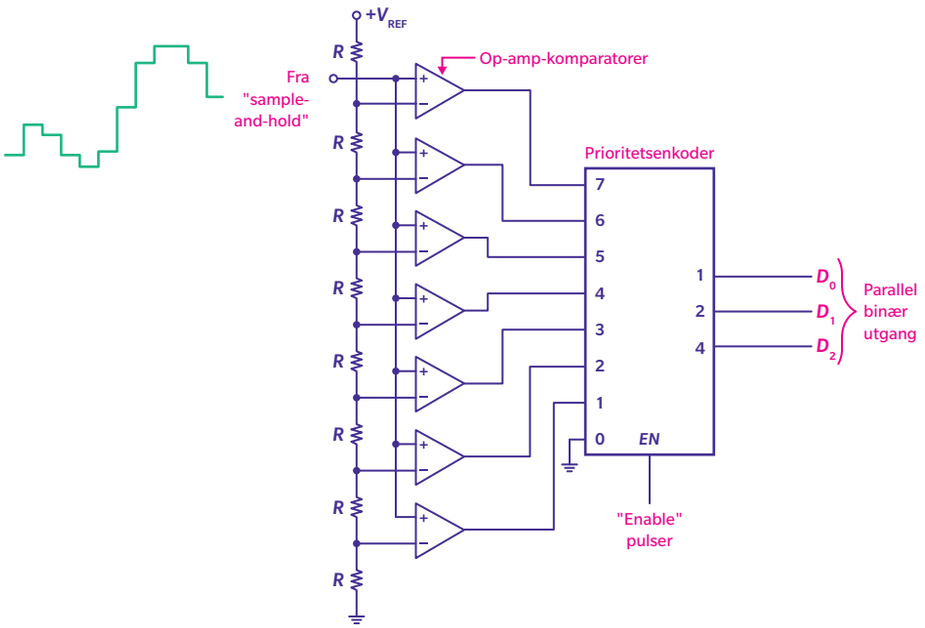


Figur 15.8 Op-amp som inverterende forsterker



Figur 15.9 Op-amp som komparator

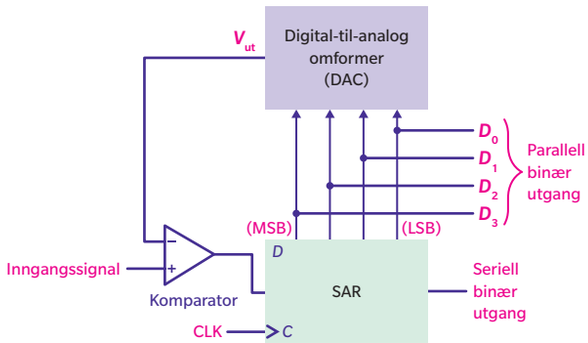
Når operasjonsforsterkeren brukes som komparator, figur 15.9, vil en liten forskjell på de to inngangssignalene drive den i metning enten som HØY eller LAV alt etter hvilket signal som er det største.
Hvilken ADC skal vi begynne med? Kanskje den med enklest oppbygging? Da står FLASH ADC-en lagelig til for hogg. Figur 15.10 viser en 3-bit FLASH.



Figur 15.10 3-bit FLASH ADC

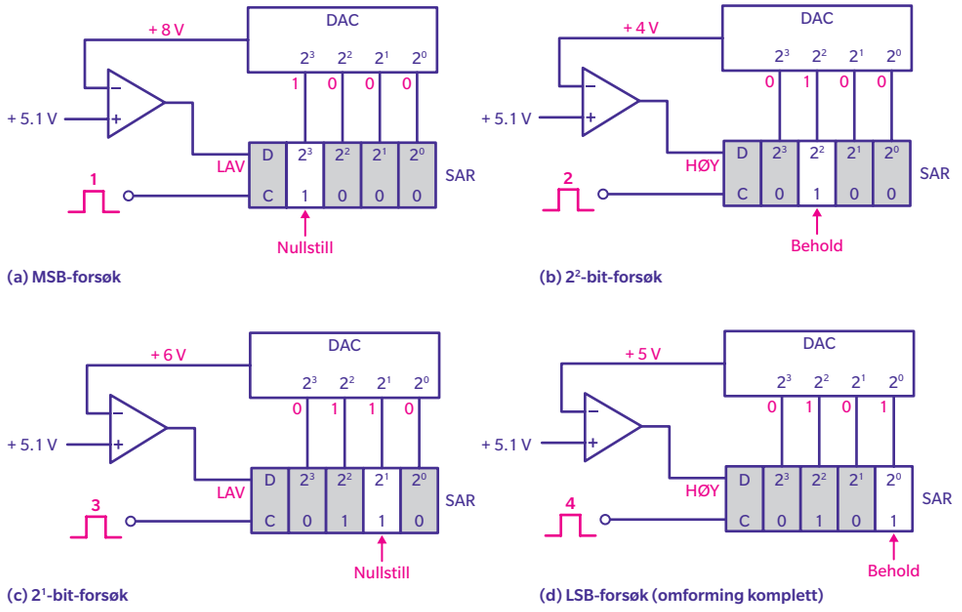
Her sammenlignes inngangssignalet med en referansespenning som er delt ned i ulike spenningsnivå mot jord av like motstander R. Sammenligningen gjøres i 7 komparatorer. Dersom signalspenningen i en gitt komparator er større enn den neddelte referansespenning, blir komparatoren HØY på utgangen. I motsatt tilfelle vil den bli lav. I prioritetsenkoderen velges den høyeste inngangen av 0 til 7 som er HØY når «enable» settes HØY. Den høyeste inngangen med HØY sendes så til prioritetsenkoderens utgang. Fordelen med FLASH ADC er at den er rask, men dessverre trenger den mange komparatorer dersom en ønsker høy oppløsning.

En mye brukt ADC er SAR (Suksessivt Approksimasjons Register). Den er rimelig rask, og man kan tillate seg større oppløsning uten å sprenge budsjettet. Figur 15.11 viser en 4-bit SAR.



Figur 15.11 4-bit SAR ADC

Som du ser, klarer en seg her med kun en komparator, og den sammenligner suksessivt inngangssignalet med den verdien som bygger seg opp i registeret. For å få det til må det som er i registeret konverteres tilbake til et analogt signal igjen i en DAC (Digital to Analog Converter). Hvordan en slik DAC virker, skal vi komme tilbake til senere. Vi må bare gjøre oss ferdig med det motsatte, les ADC, først. For å forstå hvordan SAR virker, kan det være naturlig å bruke metoden som alltid virker – å forklare ved hjelp av et eksempel (Digital Fundamentals [2] side 710). Figur 15.12 viser hvordan vår 4-bit SAR leser en inngangsspenning på 5,1 volt.

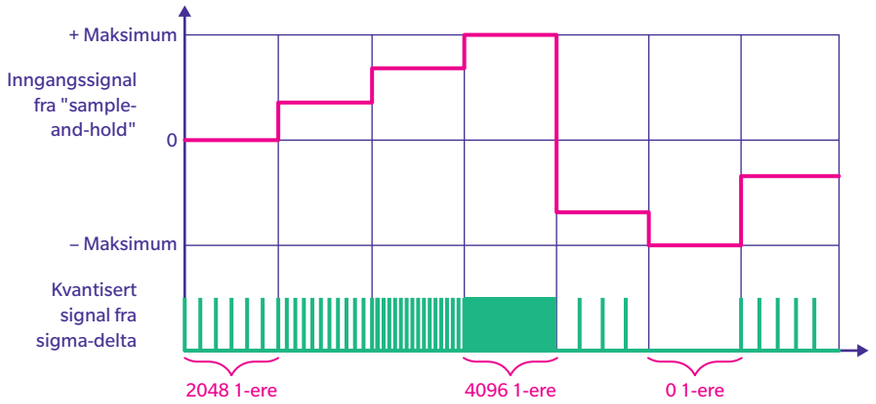


Figur 15.12 4-bit SAR ADC

Registeret inneholder 4 bit og MSB representerer 8 volt, og så deler vi på 2 helt til vi kommer til LSB som representer 1 volt.

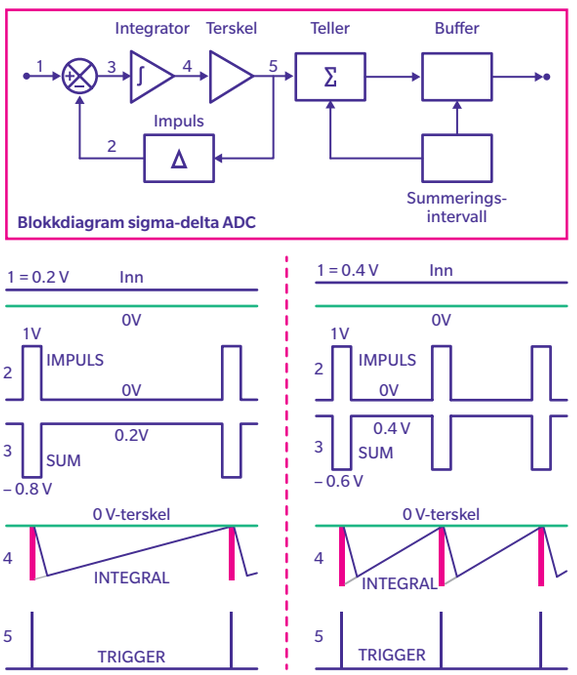
Vi begynner med å sette MSB til 1, og da vil 8 volt bli sammenlignet med inngangssignalet som er 5,1 volt ved første klokkesyklus. Det vil resultere i LAV fra komparatoren, og MSB vil bli satt til null. Ved neste klokkepuls blir 2^2 bit til 1, og det gir 4 volt ut av DAC. 4 volt er mindre enn 5,1 volt, og komparatoren sender ut HØY og 2^2 blir beholdt som 1. Dette fortsetter ufortrødent til LSB, og vårt inngangssignal på 5,1 volt vil bli representert med tallet 5 og sendt til videre behandling i signalprosessoren.

Neste ADC ut er sigma-delta (Σ-Δ) konverteren. Da den kan realiseres med stor oppløsning samtidig som den er rask, er den mye brukt. Før vi finner ut hvordan den fungerer, kan det være nyttig å se hva den produserer. I figur 15.13 ser vi hvordan et «sample and hold» innsignal blir omgjort til en sekvens av 1 bit. Maksimum innsignal gir 4096 påfølgende bit, 0 gir 2048 bit, og maksimum negativ verdi 0 gir 1 bit. Denne sigma-delta har en oppløsning på 12 bit, da $2^{12} = 4096$. Det siste sigma-delta ADC-en gjør, er å summere strømmen av 1 bit til et tall som leveres videre.



Figur 15.13 Sigma-delta ADC konvertering

Hvordan er så en sigma-delta konverter realisert? I figur 15.14 er det vist et blokkskjema for en sigma-delta konverter sammen med noen grafer som forklarer virkemåten.



Figur 15.14 Sigma-delta ADC konverter

La oss prøve å forstå basert hva som skjer. I eksempelet nederst til venstre i figur 15.14 kommer det et analogt signal på $0,2$ volt (ved 1 i blokkdiagrammet for sigma-delta ADC). Det blir sammenlignet med en puls (2) på 1 volt. Deres differanse summerer seg sammen til $+0,2-1=-0,8$ volt (3). Integratoren integrerer så fra $-0,8$ volt med

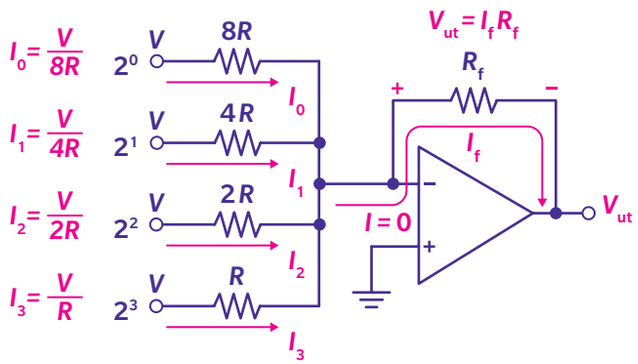
inngangssignalet $0,2$ volt som stigningstall til terskelen, 0 volt, den sammenlignes med i komparatoren mellom (4) og (5). Når terskelen er nådd, vil det sendes ut en ny puls (5) på 1 volt som vil sendes til både impuls- og tellerenhet. På denne måten lages det en strøm av pulser som representerer $0,2$ volt. De blir telt opp i telleren, og via en buffer sendt videre. I eksempelet nederst til høyre i figur 15.14 ser vi at det analoge signalet er $0,4$ volt. Da vil stigningstallet bli 0,4 volt, og det trigges pulser dobbelt så ofte som da stigningstallet var $0,2$ volt for et analogt signal på $0,2$ volt.

Hva så med «dual-slope» ADC-en som det også er referert til i figur 15.7? Vel, den hadde jeg tenkt å forbigå i stillhet. Den har riktignok høy oppløsning, men er lite brukt i digital signalbehandling da den er relativt treg sammenlignet med de andre. Et viktig bruksområde for «dual-slope» er målesystemer, for eksempel multimetre, som trenger høy nøyaktighet, men hvor tiden kommer til en.

Så er det rosinen i pølsen, kjernen i det hele, nemlig den digitale signalprosessoren (DSP Digital Signal Prosessor). Det som kjennetegner en DSP, er at den kan utføre en begrenset mengde oppgaver, men gjør det raskt. Da studiet av digital signalprosessering er et langt lerret å bleke, så la oss heller gjøre oss ferdig med det som kommer etter DSP-en. Etter at vi har gjort oss ferdige med DSP-ens omgivelser både før og etter den, kan vi med glede studere de muligheter som DSP-en gir oss.

Etter at DSP-en har drevet med signalbehandling basert på tallknusing, må tallene gjøres om til et analogt signal igjen. Det skjer i digital til analog konverteren (DAC «Digital to Analog Converter»).

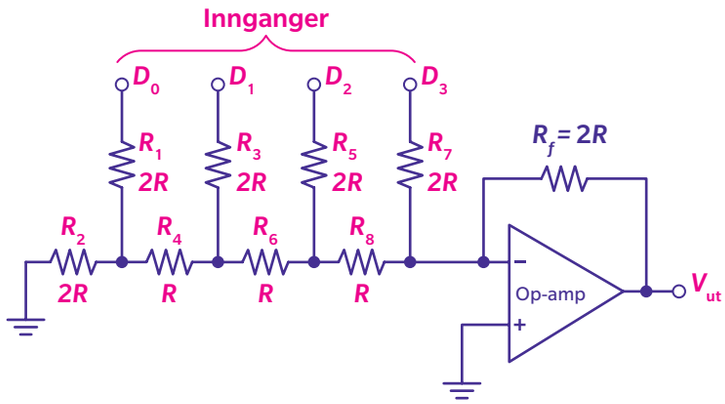
Vi skal se på to mye brukte digital til analog konvertere. De bruker begge motstandsnettverk og operasjonsforsterker for skaffe til veie et analogt utgangssignal. La oss starte med den binær-vektede DAC. Figur 15.15 viser en utgave med 4-bit.



Figur 15.15 Binær-vektet DAC

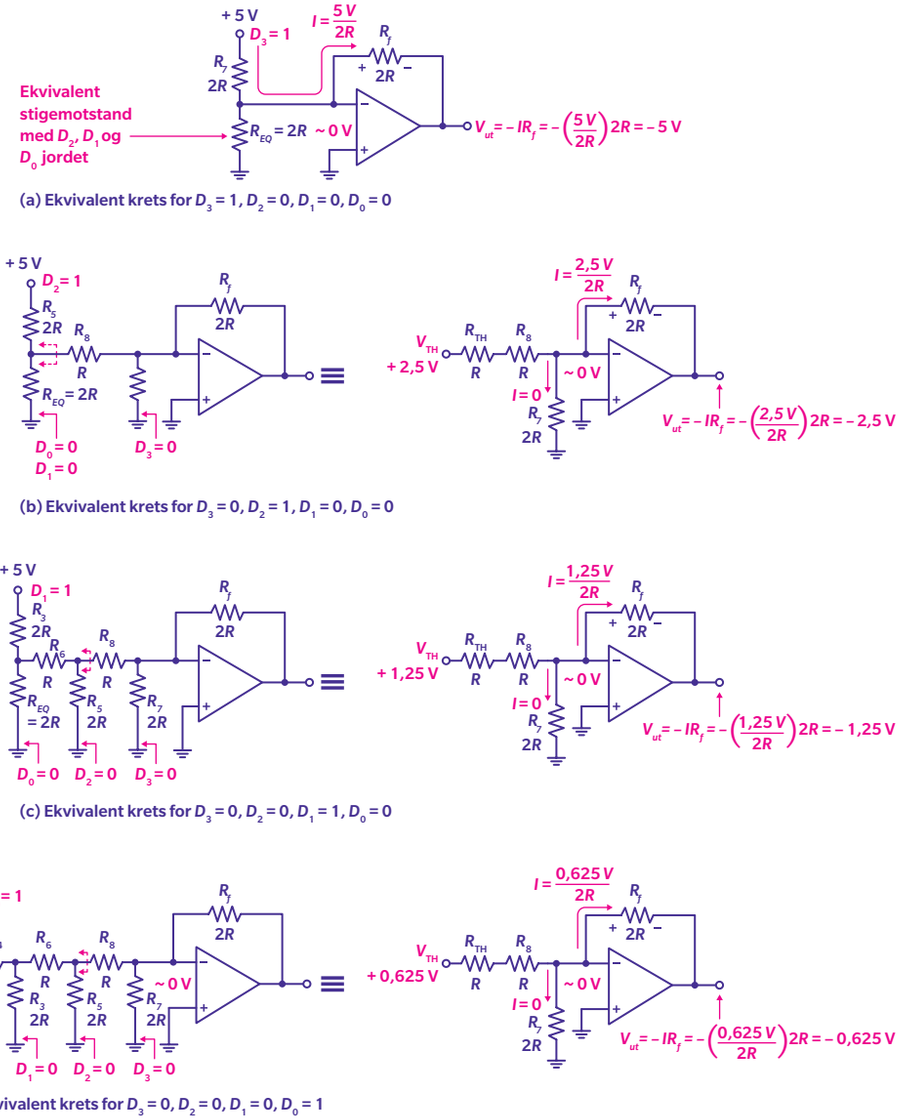
Den binære vektingen foretas ved å la MSB ha den laveste motstandsverdien og de påfølgende et multippel av 2 av forrige verdi. I vårt eksempel blir det da R, $2R$, $4R$ og $8R$. Når et bit er HØY, vil det gi en strøm som er avhengig av motstandsverdien for det gitte bit. Er flere bit HØY, vil strømmene addere seg, og utgangen av operasjonsforsterkeren vil gi en utgangsspenning som er proporsjonal med det binære tallet som leveres på inngangen.

En annen digital til analog konverter type er den såkalte R/2R-stige DAC-en som er vist i figur 15.16. Sammenlignet med binær-vektet DAC har den den fordel at det kun brukes to motstandsverdier.



Figur 15.16 4-bit R/2R-stige DAC

I figur 15.17 (utvidet eksempel basert på Digital Fundamentals [2] side 718) ser vi hvordan HØY på ulike binære innganger gir en tilhørende utgangsspenning. Da vi har en inverterende forsterker, vil MSB ha den laveste spenning, og den divideres med 2 for hvert påfølgende bit.



Figur 15.17 Analyse av 4-bit R/2R-stige DAC

For deg som er en kløpper med Thevenin og sånt, er dette muligens lett å forstå, men for oss som ikke husker fra tolv til middag eller ikke har vært borti dette før, kan det muligens vært greit med en rask repetisjon.

Lineære kretser kan analyseres ved å bruke såkalt Theveninekvivalenter. Da blir hele kretsen redusert til en Theveninspenning og en Theveninmotstand i serie med Theveninspenningskilden og den ytre kretsen. For å finne de to ukjente, nemlig

Theveninspenningen og Theveninmotstanden, tar en det ytre påtrykk til de to ytterkanter: åpen krets, uendelig motstand, og kortslutning.

La oss se på figur 15.17 c). Alle bit unntatt det som kommer på D_1 er 0 og er dermed jordet. Nederst i venstre hjørne ser en at $R_{EQ} = 2R$. For å se hvordan det kommer frem, kan det være fornuftig å ta en titt på figur 15.16. Når D_0 jordes, vil $R_1 = 2R$ og $R_2 = 2R$ være i parallell, og til sammen blir motstanden etter R_4 lik R_{EQ}.

$$R_{EQ} = \frac{1}{\left(\frac{1}{R_1} + \frac{1}{R_2}\right)} + R_4 = \frac{1}{\left(\frac{1}{2R} + \frac{1}{2R}\right)} + R = 2R$$

Så finner vi Theveninekvivalenten til kretsen til venstre for R_8. Theveninspenningen er gitt ved spenningen mellom R_5 og jord når vi kobler bort kretsen til høyre for R_5. Først må vi finne motstanden i parallellkoblingen som består av R_{EQ} og $\left(R_6 + R_5\right)$.

$$R_{parallell} = \frac{1}{\left(\frac{1}{R_{EQ}} + \frac{1}{R_6 + R_5}\right)} = \frac{1}{\left(\frac{1}{2R} + \frac{1}{3R}\right)} = \frac{6}{5}R$$

Så kan vi finne strømmen I_{Total} fra D_1 til jord.

$$I_{Total} = \frac{5\ V}{\left(R_3 + R_{parallell}\right)\ \Omega} = \frac{5\ V}{\left(2 + \frac{6}{5}\right)R\ \Omega}$$

Deretter må vi finne hvor mye strøm det går i greinen med R_6 og R_5. Motstanden i den greinen er totalt $3R$, og i den andre greinen med R_{EQ} er motstanden $2R$. Strømmen vil altså være minst i greinen med R_6 og R_5 og være gitt ved:

$$I_{grein} = \frac{5\ V}{\left(2 + \frac{6}{5}\right)R\ \Omega} \cdot \frac{2}{\left(3 + 2\right)} = \frac{2\ V}{\left(2 + \frac{6}{5}\right)R\ \Omega}$$

For å finne spenningen (Theveninspenningen) i punktet over R_5 er det bare å multiplisere I_{grein} med R_5:

$$V_{TH} = \frac{2\ V}{\left(2 + \frac{6}{5}\right)R\ \Omega} \cdot R_5 = \frac{2\ V}{\left(2 + \frac{6}{5}\right)R\ \Omega} \cdot 2R\ \Omega = \frac{4\ V}{\left(\frac{10}{5} + \frac{6}{5}\right)} = \frac{5\ V}{4} = 1{,}25\ \ V$$

Så er det å finne Theveninmotstanden. Det gjøres ved å kortslutte kretsen til venstre før R_8. Først finner vi strømmen i greinen med kortslutning. Da må vi først finne parallelmotstanden for de to greinene R_{EQ} og R_6. Husk at vi nå har kortsluttet R_5.

$$R_{parallell} = \frac{1}{\left(\frac{1}{R_{EQ}} + \frac{1}{R_6}\right)} = \frac{1}{\left(\frac{1}{2R} + \frac{1}{R}\right)} = \frac{2}{3}R$$

Så kan vi finne strømmen I_{Total} fra D_1 til jord. Denne gang med R_5 kortsluttet.

$$I_{Total} = \frac{5\ V}{\left(R_3 + R_{parallell}\right)\ \Omega} = \frac{5\ V}{\left(2 + \frac{2}{3}\right)R\ \Omega} = \frac{5\ V}{\frac{8}{3}R\ \Omega}$$

Igjen må finne hvor mye strøm det går i greinen med R_6 . Motstanden i den greinen er R , og i den andre greinen med R_{EQ} er motstanden $2R$. Strømmen vil altså være størst i greinen med R_6 og være gitt ved:

$$I_{grein} = \frac{5\ V}{\frac{8}{3}R\ \Omega} \cdot \frac{2}{(1+2)} = \frac{5\ V}{\frac{8}{3}R\ \Omega} \cdot \frac{2}{3} = \frac{5\ V}{4R\ \Omega}$$

Siden vi vet V_{TH} , kan vi finne R_{TH} siden den er det eneste som påvirker strømmengden ved kortslutning.

$$R_{TH} = \frac{V_{TH}}{I_{grein}} = \frac{1{,}25\ V}{\frac{5\ V}{4R\ \Omega}} = R\ \Omega$$

Det tok sin tid, og til vår fascinasjon så stemmer det med resultatene i figur 15.17 c) også. Var dette litt gresk, kan jeg anbefale å lese kapittelet «Stille! Eksamen pågår» i den tilhørende boken *Grunnleggende elektroteknikk – kort og godt fortalt* [13]. Der blir Thevenin studert i detalj. Med V_{TH} og R_{TH} på plass er det rimelig lett å analysere resten av kretsen. Da + (ikke inverterende) inngang er jordet, vil på grunn av den store inngangsmotstanden i operasjonsforsterkeren også – (den inverterende) inngangen tilnærmet være jordet. All strømmen som er bestemt av spenningsfallet fra D_1 til den inverterende inngangen, vil gå gjennom R_f, og da den også har verdien $2R$, ender vi ut med $-1{,}25$ volt.

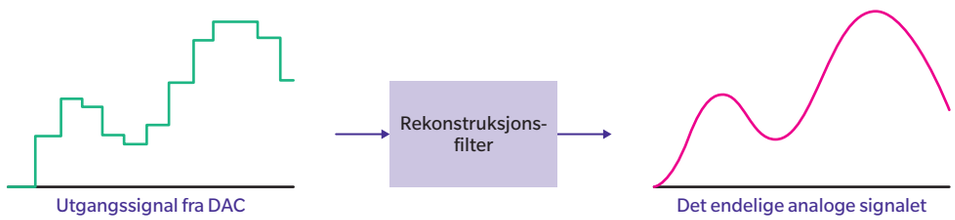
Før vi forlater DAC-ene, kan det være på sin plass å se litt på karakteristikkene som beskriver deres ytelse (Digital Fundamentals [2] side 719).

- Oppløsning er det omvendte av hvor mange trinn DAC-en har, og oppgis gjerne i %. For en 4-bit DAC blir det .

$$\left(\frac{1}{2^4-1}\right)\cdot 100 = \left(\frac{1}{15}\right)\cdot 100 = 6{,}67\ \%$$

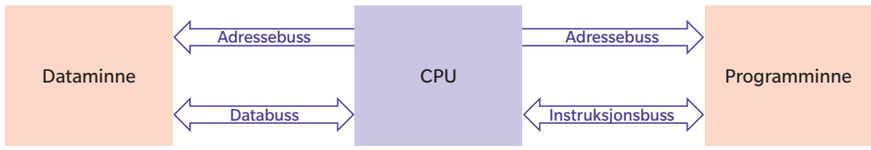
- Nøyaktighet finner en ved å sammenligne aktuelt utgangssignal fra en DAC med det forventede. Ideelt sett bør det ikke være mer enn ±½ av det minst signifikante bit. For en 4-bit DAC er LSB $6{,}67\ \%$ av full skala, og kravet til nøyaktighet vil da være $\pm 6{,}67\ \%$.
- Linearitet er avvik fra en ideell rett linje på utgangsverdiene når DAC-en fores med stadig økende digitale inngangsverdier.
- En DAC er monoton når utgangsverdiene øker med stadig økende digitale inngangsverdier.
- Stabiliseringstid er den tid det tar før utverdien er stabil, innenfor en verdi på ±½ LSB ved en endring av inngangsverdi.

Det siste som må gjøres med signalet før det slippes ut i den analoge verden, er å sende det igjennom et rekonstruksjonsfilter. Det er et lavpassfilter, med cut-off frekvens på halve Nyquistfrekvensen, som jevner signalet ved å fjerne trappetrinnene som kun er en konsekvens av digitaliseringsprosessen, og ikke har noe med signalet vi ønsker å gjøre. De raske endringene i trappetrinnene gir høyfrekvent innhold i utsignalet dersom de ikke fjernes. Cut-off frekvensen til et filter er den frekvens hvor et filter demper signalet til det halve (3 dB). For et lavpassfilter vil frekvenser over cut-off frekvensen dempes enda mer, og typisk øker dempningen med 20 dB for hver dobling av frekvens. Du kan lese mer om filtre i *Grunnleggende elektroteknikk – kort og godt fortalt* [13] i kapittelet «Gleden av å ha en impedans å snakke inn i». Figur 15.18 viser resultatet av denne prosessen.



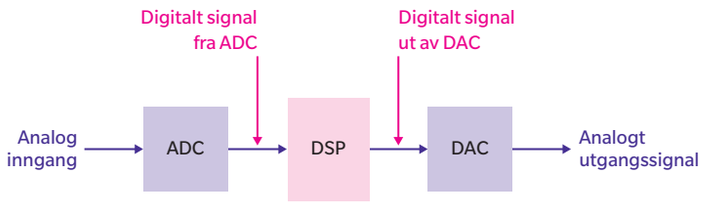
Figur 15.18 Rekonstruksjonsfilter

Så var det tilbake til selve sakens kjerne, nemlig DSP-en. Som tidligere nevnt, er den en spesiell type mikroprosessor som prosesserer data i sanntid. DSP-en har som mikroprosessoren en CPU og minner. Mange DSP-er har såkalt Harvard arkitektur, vist i figur 15.19, hvor det er egne minner for henholdsvis data og program. Hensikten med det er å få prosessene til å gå raskere.



Figur 15.19 DSP med Harvard arkitektur

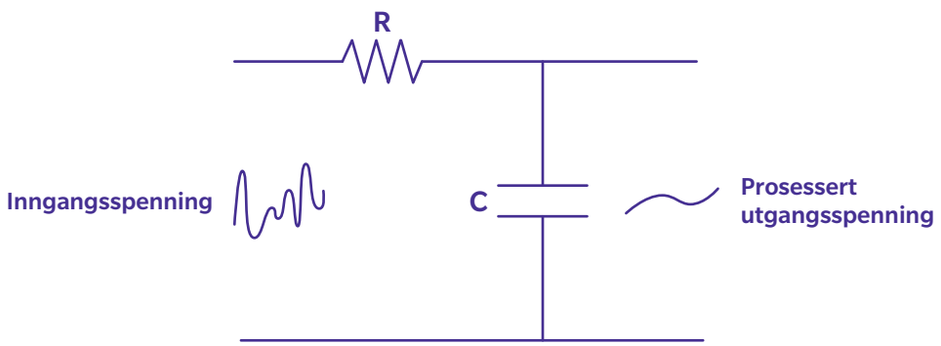
Anvendelsene til en DSP fokuserer på behandling av analoge signaler som har blitt digitalisert. DSP-en henter data fra analog til digitalkonverteren og leverer videre etter prosessering til digital til analog konverteren (DAC). Figur 15.20 viser kjeden.



Figur 15.20 DSP forholder seg digitale verdier

For å få ting til å gå fort, programmeres DSP-en i maskinære språk som assembler eller C. DSP programmer er også vesentlig mindre enn tradisjonelle mikroprosessorprogrammer. I mange tilfeller har DSP-en et redusert instruksjonssett.

La oss returnere til det analoge lavpassfilteret, figur 15.21, som vi presenterte innledningsvis. Hvordan skal vi lage en digital utgave av det som DSP-en kan programmeres med? Dette er en relativt liten jobb for en DSP, men et sted må vi jo begynne.



Figur 15.21 Analogt lavpassfilter med motstand *R* og kondensator *C*

Det første vi må gjøre, er å lage en modell som beskriver virkemåten. Da må vi returnere til den analoge kretsteknikken, for å finne en sammenheng mellom inn- og utspenning. Spenningsfallet over motstanden R ved et gitt tidspunkt t er:

$$V_{inn}(t) - V_{ut}(t) = R \cdot I(t)$$

Strømmen i kretsen er diktert av ladningsvariasjonen i kondensatoren:

$$I(t) = \frac{dQ_c}{dt} = C\frac{d\left(V_{ut}(t)\right)}{dt}$$

C er kondensatorens kapasitans. Vi setter så de to ligninger sammen og får:

$$V_{inn}(t) - V_{ut}(t) = R \cdot C\frac{d\left(V_{ut}(t)\right)}{dt}$$

$$R \cdot C\frac{d\left(V_{ut}(t)\right)}{dt} + V_{ut}(t) = V_{inn}(t)$$

Modellen vår er det matematikerne kaller en ikke homogen differensialligning av første orden. Den er ikke homogen, da vi har et påtrykk $V_{inn}(t)$ på høyre siden av ligningen. Ligningen er av første orden, da vi ikke har deriverte av høyere orden enn første.

Vi må digitalisere denne ligningen. Vi mottar nye inngangsverdier for hver gang det samples. Den n'te verdien kan vi benevne $V_{inn}[n]$. Hakeparentesen signaliserer at vi nå er i den digitale verden, og at $V_{inn}[n]$ er verdien etter nT samples. Vi kan nå ved å omforme vår formel til en differensligning, gi DSP-en noe å tygge på.

$$R \cdot C\frac{\left(V_{ut}[n] - V_{ut}[n-1]\right)}{T} + V_{ut}[n] = V_{inn}[n]$$

Vi ser at den deriverte blir forskjellen mellom to påfølgende utsignal delt på samplingstiden.

Når innsignalet varierer langsomt, vil $\left(V_{ut}[n] - V_{ut}[n-1]\right) \approx 0$, og dermed blir $V_{ut}[n] \approx V_{inn}[n]$. Lave frekvenser slipper rett igjennom filteret. Ved høye frekvenser blir den deriverte stor og $V_{ut}[n] \approx 0$. Vi har laget et digitalt lavpassfilter.

For å gjøre formelen litt mer beregningsvennlig samles $V_{ut}[n]$ på venstresiden:

$$V_{ut}[n] = \frac{V_{inn}[n] + \frac{R \cdot C}{T}\left(V_{ut}[n-1]\right)}{\left(1 + \frac{R \cdot C}{T}\right)}$$

$V_{ut}[n]$ finner en når en kjenner $V_{inn}[n]$ og $V_{ut}[n-1]$. Dersom en ønsker å endre på filteret, er det bare å manipulere R og C som nå ikke lenger er fysiske komponenter, men tall i DSP-en. Den lille formelen vi har kommet frem til for det digitale lavpassfilteret, kan enkelt beskrives med en kodesnutt i C eller assembler. En har oppnådd de fordelene som digital signalbehandling gir, som ble lovet innledningsvis.

- Fleksibelt
- Programmerbart
- Repeterbart
- Fysisk stabilt

Var det alt? Nei, knapt nok begynnelsen. Det er skrevet tykke bøker og heldigvis noen tynne om emnet digital signalbehandling. I denne omgang får vi nøye oss med å se på noen av de mange anvendelsene av DSP.

DSP er dedikert til prosesser som foregår i sanntid. DSP-er finnes i mobiltelefoner, måleinstrumenter, digitale radioer og video- og musikkavspillere. De brukes blant annet for å bedre signalkvaliteten i kildekodere og til komprimering og kryptering. I telekommunikasjon anvendes DSP til multipleksing, filtrering, protokollhåndtering, ekkokanselering, feildeteksjon og feilkorreksjon. Mobiltelefonen du har i lommen er en DSP med garnityr.

I musikk brukes DSP til filtrering, signalredigering, kunstig ekko, komprimering og simulering av naturlige lytteforhold. DSP spiller også en aktiv rolle i systemer for talekonstruksjon og -gjenkjenning. Å gjenkjenne tale er vesentlig vanskeligere enn å konstruere tale.

I radar- og sonarteknologi brukes DSP til å prosessere data for bedre å kunne bestemme avstand, fjerne støy og gjenkjenne objekter basert på deres ekko. I medisinsk billedbehandling brukes DSP til å prosessere data fra CT («Computed Tomography») og MRI («Magnetic Resonance Imaging»).

Dersom du åpner en bok i digital signalbehandling, vil du se at den primært handler om filtre og digital Fourieranalyse. Årsaken er enkel. Med det verktøyet kan en utføre separasjon av signaler, gjenskape signaler som har vært utsatt for støy og forvrengning, komprimere signaler, utføre frekvensanalyse og skape signaler med signalsyntese. Verden formelig ligger for våre føtter.

16

Kapittel 16

Fra A til B

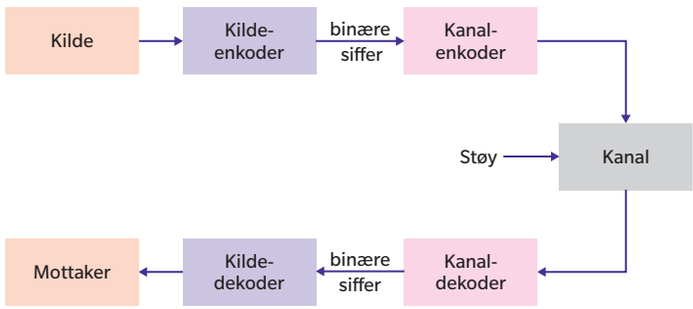
«The signal is the truth. The noise is what distracts us from the truth.»

The signal and the noise, Nate Silver (1978–)

LÆRINGSUTBYTTE: Digital kommunikasjon, enkodere og dekodere, støy, komprimering, kryptering, modulasjon, buss, parallell og seriell buss, synkron eller asynkron kommunikasjon, vridde parkabler, koaksialkabel, fiberkabel, trådløs kommunikasjon, simpleks, halv-dupleks, full-dupleks, amplitudeskiftmodulasjon, frekvensmodulasjon, fasemodulasjon, kvadratur-amplitude modulasjon, pulskode modulasjon, tidsdelt og frekvensdelt multipleksing, kildekoder, informasjonsteori, informasjonsmengde, entropi, Shannon-Fano-koding, Huffman-koding, redundans, Lempel-Ziv, kryptering, AES (Advanced Encryption Standard), offentlig nøkkel, RSA (Rivest Shamir Alderman), Shannon kanalkapasitet, Shannon-grensen, kanalkoder, paritetssjekk, CRC-kode, feilkorrigerende koder, perfekt kode, Hamming-avstand, systemkvalitet, konvolusjonskoder, trellis, Viterbi

Dette er ikke en moralpreken, men en observasjon. Mange velger sin livsledsager ved et øyeblikks innskytelse klokken halv fire om natten på en mørk bar iført ølbriller. Er det fornuftig? Vel, dersom en skal sikre seg et godt valg, bør en enten observere over lang tid for å fjerne støyen i målingene eller sørge for at signalene en mottar er mye sterkere enn støyen, slik at selv en kjapp titt er nok. Det beste er selvfølgelig å gjøre begge deler. Tilfeldighetene sørget for at min kone og jeg brukte et halvt år på tilnærmingsprosessen med signaler så sterke at andre måtte fortelle oss det. Resultatet ble 42 års samliv.

Hva skal et digitalt kommunikasjonssystem gjøre? Det skal sørge for effektiv og sikker kommunikasjon mellom sender og mottaker. I figur 16.1 er det vist en prinsippskisse av et digitalt kommunikasjonssystem fra kilde til mottaker.

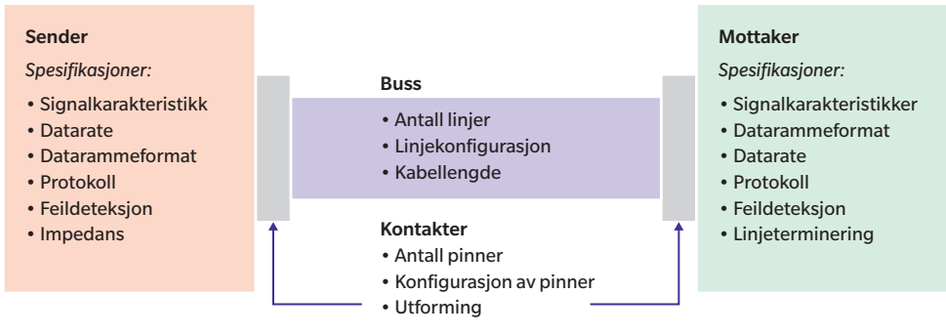


Figur 16.1 Digitalt kommunikasjonssystem

På kildesiden komprimeres data som skal sendes. Hvor mye en kan komprimere uten å miste meningsinnhold, skal vi se på litt senere. Dersom man ikke ønsker at andre enn mottaker skal forstå budskapet som blir sendt, krypteres data som sendes. Disse to prosessene, komprimering og kryptering, foregår i kildeenkoderen. Deretter tilpasses data som skal sendes til transmisjonsmediet (kanalen, overføringsmediet) i kanalenkoderen. Er kanalen analog, må det digitale signalet fra kildeenkoderen omgjøres til et analogt signal som kan moduleres på ulike måter. Hvilken modulasjonsmetode en velger, er bestemt av ønsket transmisjonshastighet og støynivå i kanalen. Jo mer støy, desto lavere blir overføringskapasiteten. På mottakersiden dekodes signalet fra kanalen i kanaldekoderen og digitaliseres. I kildekoderen foretas dekryptering og dekomprimering av data. Dersom ting går vår vei, vil det mottatte signal være en eksakt kopi av det sendte. For å forvisse seg om at så er tilfellet, legges det ofte til noen ekstra bit i kanalenkoderen som gjør det mulig for kommunikasjonssystemet enten å oppdage feil eller både å oppdage og rette feil. Hvilken metode som brukes, kommer an på hva slags data som overføres. Med tidskritiske data prøver en å rette feil, men for data som ikke er tidskritiske, er det nok å oppdage feil og be om å få sendt data som inneholdt feil på nytt.

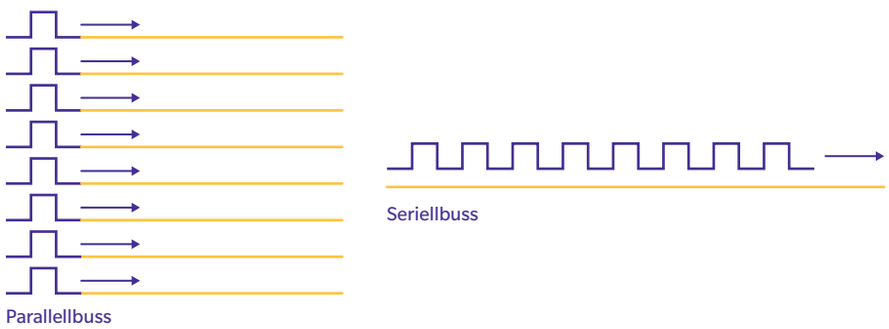
Etter denne noe summariske gjennomgangen av et kommunikasjonssystem, la oss se litt nærmere på de enkelte delene og prøve å forstå hva de gjør, og hvordan de virker. La oss starte med kanalen (transmisjonsmediet). Valg av medium er som oftest diktert av omgivelsene til kommunikasjonssystemet. Skal data sendes internt i en datamaskin, brukes en buss. Skal en derimot kommunisere med en satellitt på vei ut av solsystemet, er det mikrobølgeradio som må brukes. Det sier seg selv at det å kommunisere i det sistnevnte tilfellet er mest utfordrende med tanke på støy fordi signalnivåene er så lave.

En buss lar sender og mottaker kommunisere med hverandre. En buss kan enten være intern eller ekstern i et system. Figur 16.2 viser en prinsippskisse av et buss-system.



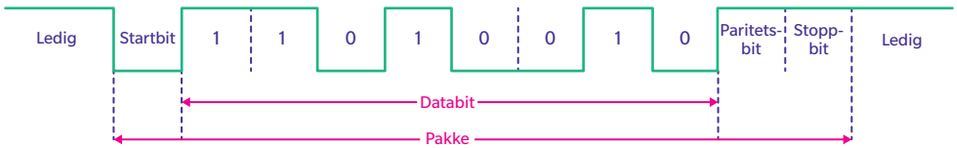
Figur 16.2 Buss-system

En buss har en rekke ledninger (kan være ledningsbaner internt i et system) som kan overføre data og kontrollsignaler. Data kan overføres enten parallelt eller serielt. En parallell buss transporterer data parallelt på ulike ledninger, mens en seriell buss overfører data etter hverandre på en ledning. Figur 16.3 viser forskjellen mellom parallell og seriell buss.



Figur 16.3 Parallell versus seriell buss

Data kan enten sendes asynkront eller synkront. Asynkront betyr at pakken kan sendes når som helst, mens synkront betyr at en må sende suksessivt styrt av en klokke. Velger en å overføre data asynkront, slik som vi gjorde i kapittelet «UARTig?», må en lage datapakker hvor en angir hvor datapakken starter og stopper, i tillegg til selve datainnholdet. Et eksempel på en slik asynkron datapakke er gitt i figur 16.4.



Figur 16.4 Asynkron datapakke

Når den asynkrone bussen er ledig, vises det med en ledig-tilstand. Startbit for en asynkron datapakke er det inverterte av ledig-tilstanden. Så kommer data, kontrollbit og et stopp bit. Kontrollsignal i pakken i figur 16.4 er et såkalt paritetsbit som sjekker overføringskvaliteten. Hvordan det settes, skal vi komme tilbake til.

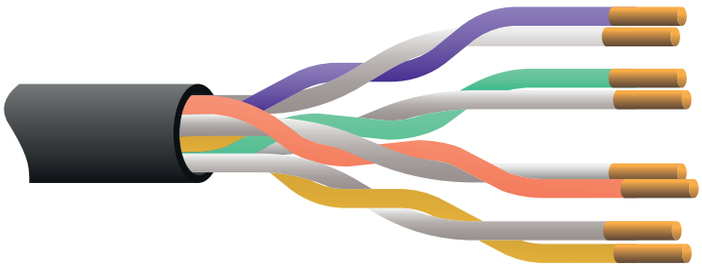
Med synkron dataoverføring har sender og mottaker samme klokke. Data kan sendes kontinuerlig, og det skjer generelt raskere enn ved asynkron overføring. Ulempen er at alle enheter på bussen må ha samme klokke og gå med samme hastighet. Data sendes i rammer med struktur som vist i figur 16.5.

Figur 16.5 Synkron datapakke

Innledning er et forvarsel til mottaker om at en ny pakke er på vei, og at mottaker må synkronisere sin klokke. Adressefeltet inneholder sender- og mottakeradresse, slik at riktig mottaker får pakken og vet hvem som har sendt den. Kontrollfeltet brukes til å angi datatype (oppkoblingsdata, filtransport og størrelsen på data). Datafeltet inneholder det egentlige datainnhold i pakken. Rammesjekken inneholder informasjon som brukes for å kontrollere om dataoverføringen har vært vellykket. Sluttrammen er et sett med bit som angir enden på hele den synkrone datapakken.

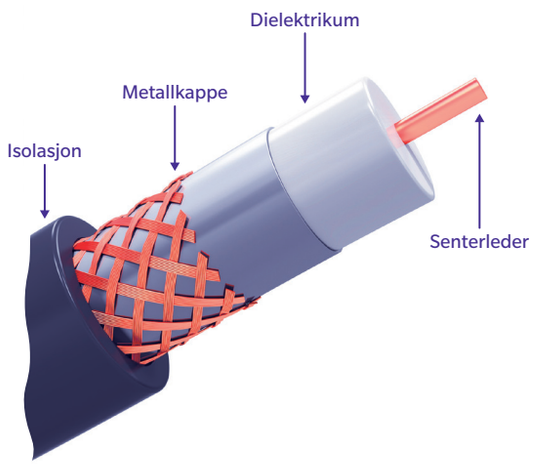
Skal man kommunisere på lengre avstander, har man en rekke valg av transmisjonsmedium. Da telefoni ble utbygd, valgte en å koble opp abonnentene med parledninger, figur 16.6, hvor lederne er vridd rundt hverandre. Å vri lederne sammen gjør

parkabelen mindre utsatt for støy. En parkabel er allikevel sårbar for støy og kan ikke brukes for signaler med høy frekvens (>1 MHz). Bruksområdet er derfor begrenset til lavrate dataoverføring i støysvake omgivelser.



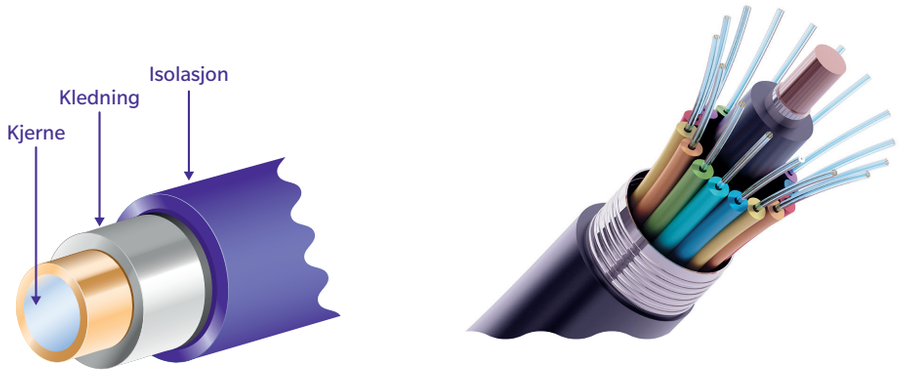
Figur 16.6 Vridde parkabler (kilde: jeffhobrath/Shutterstock)

Dersom en ønsker høyere datarate og signaler med høyere frekvensinnhold, kan koaksialkabel, figur 16.7, brukes. Signalet transporteres langs kabelen mellom innerleder og den metalliske kappen. Kappen sørger for effektiv støyskjerming, og en kan oppnå datarater opp mot 1 GHz.



Figur 16.7 Koaksialkabel (kilde: dny3d/Shutterstock)

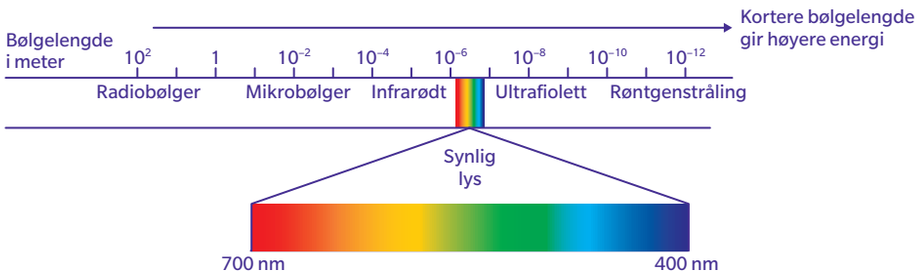
Skal en oppnå enda høyere datarater, må en bruke fiberoptisk kabel. Figur 16.8 viser til venstre en enkelt fiberkabel og til høyre en samling fiberkabler i en større kabel.



Figur 16.8 Fiberkabler (kilde: (a) Designua/Shutterstock, (b) cigdem/Shutterstock)

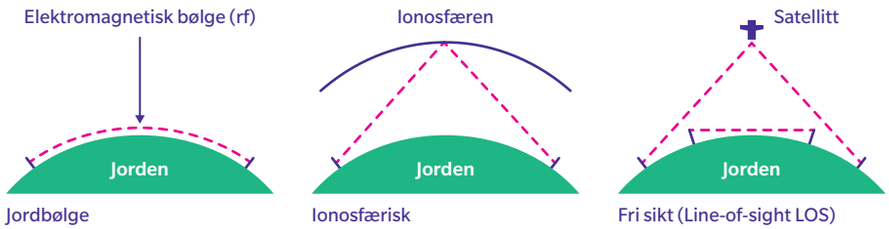
I en fiberkabel bruker man lyspulser for å overføre data. En fiberoptisk kabel har lavt tap og kan brukes over lange strekninger. Ryggraden i dagens verdensomspennende Internett er et tettmasket nett bestående av fiberkabler. For å øke datahastigheten i fiberkabler brukes såkalt fargemultipleksing, hvor en bruker lys med ulike bølgelengder samtidig. Det er mulig å oppnå datahastigheter på mer enn 1 Terabit/s = 1000 Gigabit/s.

Med fremveksten av mobiltelefoni og trådløst Internett, har bruken av radioteknologi som bærer av digital informasjon blitt verdensomspennende. For å transportere data i luften eller i verdensrommet brukes elektromagnetiske bølger. Da elektromagnetiske bølger er analoge av natur, må den digitale informasjonen tilpasses det analoge mediet i kanalenkoderen og gjøres om til et digitalt signal igjen i kanaldekoderen (se figur 16.1). Radio bruker den lavfrekvente delen av det elektromagnetiske spektrum, som vist i figur 16.9.



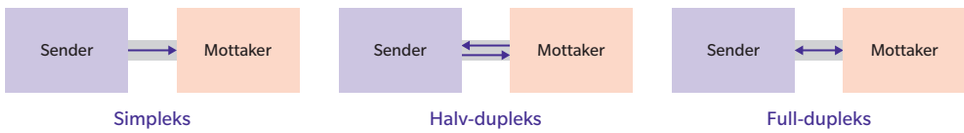
Figur 16.9 Det elektromagnetiske spektrum

På lave frekvenser (opptil 2 MHz) følger radiosignaler jordens krumning og kan derfor brukes til å kommunisere på lange avstander. Utfordringen er den begrensede båndbredden som gir lav datarate. Radiofrekvenser mellom 30 MHz og 85 MHz kan bruke ionosfæren, da den er ladet, som speil til å kommunisere over lange avstander. Utfordringen er at ionosfærens ladningsfordeling varierer med døgnet og i tillegg er temmelig uforutsigbar. Dette frekvensområdet er brukt av radioamatører og militære til opportunistisk kommunikasjon. Det finnes såkalte kognitive radioer til bruk i dette frekvensområdet som til enhver tid søker den beste frekvensen å sende på og på den måten kan sikre kommunikasjon i store prosentdeler av tiden. Mellom 85 MHz og 300 MHz er det kringkasting som råder grunnen. Mobiltelefoni har beslaglagt frekvensene fra 800 MHz til 3 GHz. Jo høyere frekvens, jo større båndbredde kan en få, og dermed en kanal med større overføringskapasitet. WiFi for innendørsbruk er i 2,4 GHz og 5 GHz. Frekvenser over 3 GHz er brukt til radiolinjer og satellittsamband. Radiolinjer som trenger fri sikt mellom sender og mottaker, brukes primært til mating av basestasjoner for mobiltelefoni. Satellitter brukes til kringkasting og lavrate-datanettverk. Figur 16.10 viser de ulike måtene radiobølger kan utbre seg på.



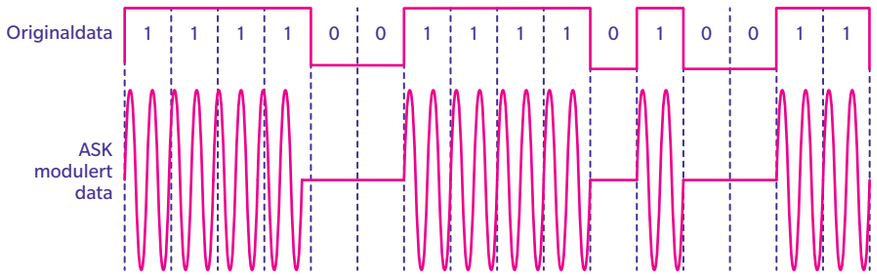
Figur 16.10 Utbredelse av radiobølger

I figur 16.1 så vi et kommunikasjonssystem med en kilde og en mottaker. Et slikt enveiskommunikasjonssystem kalles simpleks, da informasjonen kun går én vei som på en ensformig forelesning. Heldigvis finnes det andre kommunikasjonsformer. Med halv-dupleks går det kommunikasjon i begge retninger, men ikke samtidig. Dersom en har full-dupleks, går det informasjon i begge retninger samtidig. Det blir som å prate i munnen på hverandre, men disse systemene har heldigvis evnen til å gjøre to ting på en gang. Figur 16.11 viser de ulike kommunikasjonsformene skjematisk.



Figur 16.11 Ulike kommunikasjonsmoder

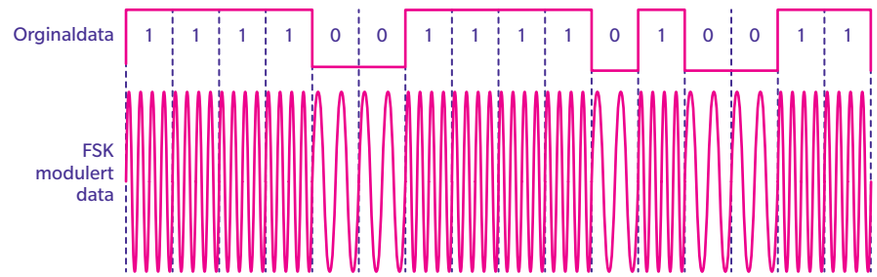
Som tidligere nevnt, må digitale signaler som skal overføres ved hjelp av analoge kanaler som for eksempel radiobølger, tilpasses transmisjonsmediet. Denne tilpasningen kalles modulasjon og skjer i kanalenkoderen. Ved modulasjon endrer man egenskaper til signalet i transmisjonsmediet med varierende digitale data. En enkel modulasjonsmetode er såkalt amplitudeskiftmodulasjon (ASK «Amplitude Shift Keying»). I sin enkleste form er det et sinussignal (bærebølge) som slås av eller på som vist i figur 16.12.



Figur 16.12 Amplitudeskiftmodulasjon

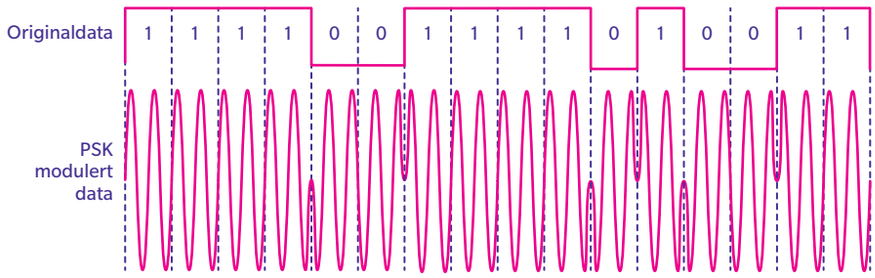
Amplitudeskiftmodulasjon er sårbar for plutselige endringer i amplitude på grunn av dempning, men er mye brukt i fiberkabler hvor det er lite støy.

For å eliminere utfordringer med amplitudestøy kan en bruke frekvensmodulasjon (FSK, «Frequency Shift Keying») som vist i figur 16.13.



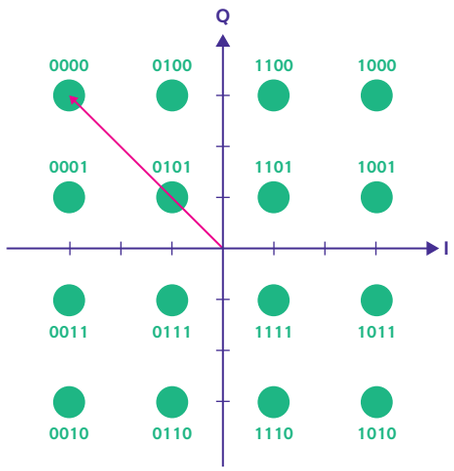
Figur 16.13 Frekvensmodulasjon

I dette tilfellet er 1 bit gitt ved en bærebølge med høy frekvens og 0 bit med en lavere bærebølgefrekvens. En annen metode for enkel modulasjon er fasemodulasjon (PSK «Phase Shift Keying») hvor bærebølgens fase endres med 180 grader hver gang bit endres fra 1 til 0 eller omvendt. Figur 16.14 viser et eksempel på fasemodulasjon.



Figur 16.14 Fasemodulasjon

Til nå har vi tilordnet et bit en modulasjonstilstand. For å effektivisere overføring av data kan man gruppere bit og tilegne gruppen en modulasjonstilstand (symbol). Ved å bruke både størrelse på amplitude og fase kan symboler representere flere bit. Det er nettopp dette som utnyttes i kvadratur-amplitude modulasjon (QAM, Quadrature Amplitude Modulation). Figur 16.15 viser et 16 QAM system hvor de ulike symbolene har forskjellig fase og amplitude.

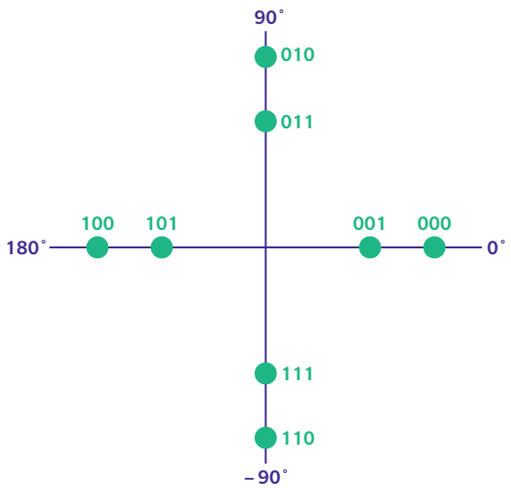


Figur 16.15 Gray-kodet 16 QAM konstellasjonsdiagram

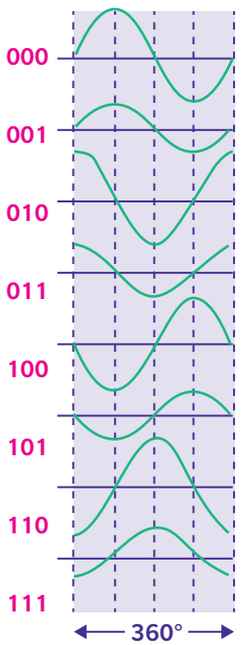
I figur 16.15 er det 16 ulike symboler, og hvert symbol kan derfor tilordnes 4 bit. Den røde vektoren viser at symbolet 0000 er representert med et sinussignal med amplitude lik vektorens lengde og fase på 135 grader. Konstellasjonsdiagrammet er Gray-kodet, da det er mest sannsynlig at en ved støy leser av et nabosymbol. Hvor mange symboler i et QAM symbolet kan man tørre å bruke? Det kommer an på kanalen. Er det lite støy, kan symbolene ligge tett som hagl, og i radiosystemer brukes det 2048 QAM

i slike tilfeller, og så reduserer en antallet symboler når støyen øker. Modulasjonen blir på denne måten adaptiv og tilpasset støynivået i kanalen til enhver tid.

For å forstå sammenhengen mellom konstellasjonsdiagram og hvordan symboltransporten ser ut, så la oss ta et enklere eksempel (Digital Fundamentals [2] side 753). Figur 16.16 viser et 8 QAM system, figur 16.17 viser hvordan sinussignalet for de enkelte symbolene ser ut, og til slutt ser en 8 QAM transmisjon for en binær sekvens i figur 16.18.



Figur 16.16 8 QAM konstallasjonsdiagram



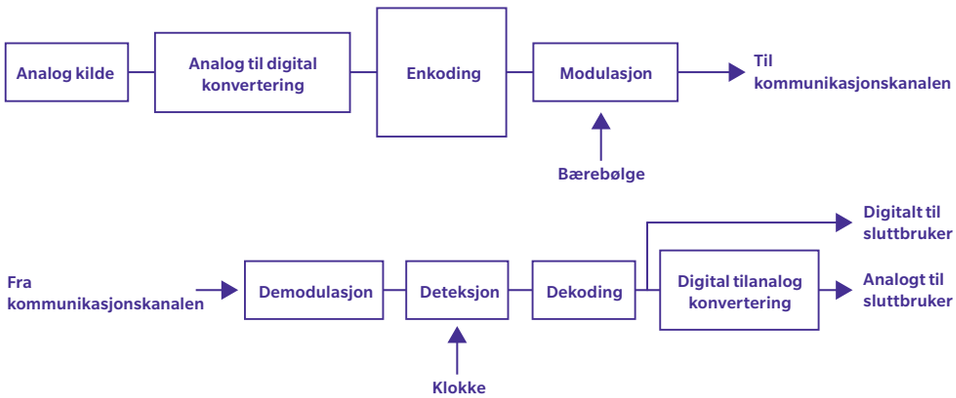
Figur 16.17 Amplitude og fase for 8 symboler

Figur 16.18 8 QAM transmisjon av en binær sekvens

Dersom kilden til data er analog som for eksempel ved telefoni, er det vanlig å modulere det analoge signalet, talen, digitalt. Etter digitalisering kan tale sendes i digital form til mottaker. Fordelene med digitalisering av tale er at den kan komprimeres og overføres med mindre støy. I telenettet, fasttelefonidelen, er såkalt pulskode-modulasjon (PCM, Pulse Code Modulation) mye anvendt. Blokkdiagrammet i figur 16.19 viser prosessen med å digitalisere, og de to første blokkene kjenner vi igjen fra DSP-systemene vi så på i kapittelet «Fremskritt og tilbakesteg».

Figur 16.19 Blokkdiagram av et PCM system

I figur 16.19 ser vi sammenhengen mellom det analoge signalet og PCM-koden som blir generert. PCM brukes for digital lyd i datamaskiner, CD, DVD, Blu-ray og digitale telefonisystemer. I telefoni punktprøves det analoge signalet 8000 ganger med 8 bit oppløsning. Det gir en datarate på 64 kbit/s. Denne dataraten kan komprimeres, og det gjøres i mobiltelefoni, ned mot 10 kbit/s før det går vesentlig utover kvaliteten. I figur 16.20 er det vist et funksjonelt blokkdiagram for et datatransmisjonssystem.

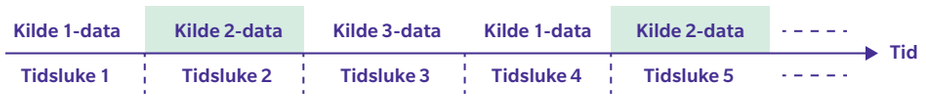


Figur 16.20 Blokkdiagram av et datatransmisjonssystem

I figur 16.21 ser en at det modulerte signalet blandes med en såkalt bærebølge. Denne bærebølgen har som regel vesentlig høyere frekvens enn det modulerte signalet. Hensikten med å bruke en bærebølge er å tilpasse det vi ønsker å sende til det transmisjonsmediet (kabler, luft, …) vi har.

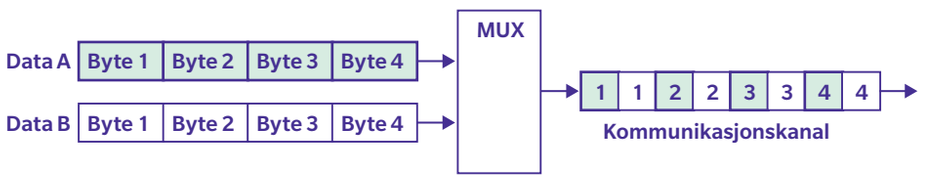
Hva gjør en så dersom en har flere datakilder og kun ett transmisjonsmedium? Da må mediet deles mellom de ulike kildene med multipleksing. To viktige mutipleksingteknikker er tidsdelt og frekvensdelt multipleksing. I tidsdelt multipleksing (TDM, Time Division Multiplexing) tildeles de ulike kilder forskjellige tidsluker. Hastigheten til det multipleksede signalet må være N ganger raskere enn det enkeltkildene leverer data med. N er antall kilder. Ved frekvensdelt multipleksing bruker de enkelte kildene kanaler med ulike bærebølgefrekvenser.

Figur 16.21 viser det grunnleggende konseptet bak TDM. Her er det tre kilder som deler en felles kanal.



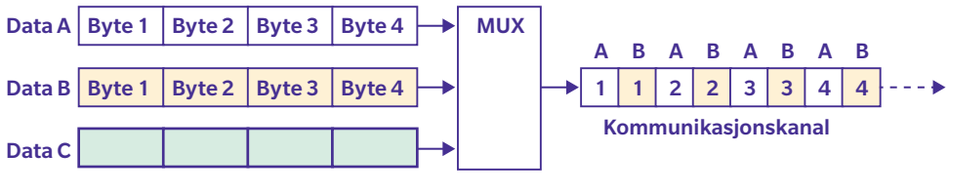
Figur 16.21 TDM konsept

Når en bruker TDM på byte nivå, må multiplekseren inneholde en buffer som sørger for å samle opp en hel byte fra en kilde før det sendes ut på kanalen. Figur 16.22 gir en prinsippskisse.



Figur 16.22 TDM bytefletting

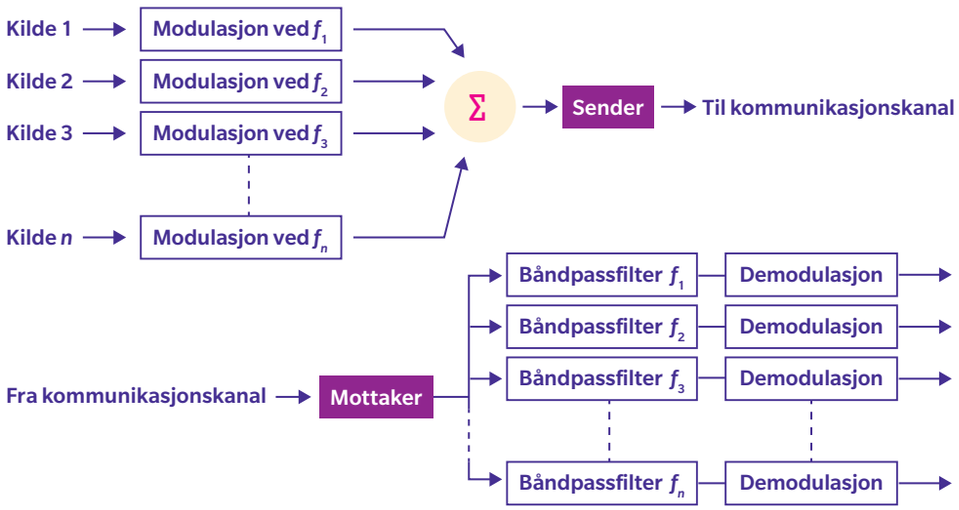
Det vi har sett på av TDM til nå, er såkalt synkron TDM hvor det kontinuerlig kommer data fra alle kildene. I en del anvendelser hender det fra tid til annen at en eller flere kilder ikke har data å transportere. Det utnyttes i statistiske multipleksere til å øke mengden data overført for de kildene som har data å sende. I figur 16.23 ser vi et eksempel på en multiplekser med tre kilder hvor kilde C har tatt en midlertidig pause som de to andre kildene kan utnytte til å øke datahastigheten.



Figur 16.23 Statistisk TDM

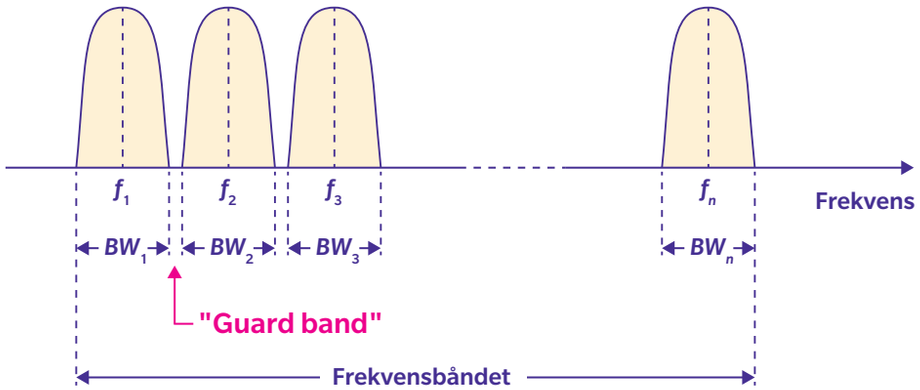
TDM brukes i telefoni for tale og data. En talekanal på 64 kbit/s multiplekses sammen med 29 andre samt to 64 kbit/s kanaler som angir rammestruktur, har feildeteksjon og data for signalering. Denne 32 kanals TDM strukturen kalles E1 og har en størrelse på 2,048 Mbit/s. E1 strukturen kan multiplekses videre til 8, 34 og 140 Mbit/s før det sendes over en ønsket kanal.

Frekvensdelt multipleksing (FDM, Frequency Division Multiplexing) er en bredbåndsteknikk, hvor den totale tilgjengelige båndbredde deles opp i sub-bånd og sendes samtidig analogt over kanalen som nå er delt opp i mange sub-kanaler. I figur 16.24 er det vist hvordan et FDM system er bygd opp.



Figur 16.24 Et FDM system

Figur 16.25 viser et generelt spektrum for FDM hvor kildene har ulike frekvenser med bærebølgefrekvenser $f_1, \ldots f_n$ og en båndbredde (BW, Band Width) rundt. Båndbredden er bestemt av mengden data som overføres og modulasjonsteknikken som er brukt. I tillegg er det et lite «guard band» mellom de ulike sub-kanalene for å unngå forstyrrelser mellom de ulike sendingene.



Figur 16.25 Spektrum til et FDM system

Jeg vet ikke om Alfred Vail var spesielt glad i å lese aviser, men det var det han brukte tiden sin på da han sammen med Samuel Morse utviklet morsekoden i 1840. Hva han var på jakt etter? Han telte forekomsten av de enkelte bokstaver for å finne deres frekvens i engelsk tekst. Han fant ut at bokstaven e var den som forekom oftest, og den ble tilordnet det korteste symbolet, som var kun en prikk. Jo sjeldnere en bokstav ble brukt, jo lengre sekvens av prikker og streker ble den representert ved. På denne måten komprimerte morsekoden det som skulle sendes over telegrafen. Var det mulig å komprimere det som ble sendt enda mer? Så absolutt, og kundene til telegrafen, blant annet børsmeglere, som syntes at bruken av den var dyr, utviklet koder på toppen av morsekoden for å komprimere meldingene mer. Det er cirka 75 prosent redundans (overflødighet) i det engelske språk. Mye kan altså fjernes/omarbeides uten at meningsinnholdet forsvinner. Dette ble utnyttet til å komprimere enda mer. Problemet var bare at når all redundans var fjernet, ble systemet sårbart for feil. Kun én feil resulterte i at hele meldingen ble uforståelig.

Er det mulig å finne ut om en kode gir maksimal komprimering? Dette spørsmålet stilte Shannon seg i sin artikkel «A Mathematical Theory of Communication» som ble publisert i 1948. Hans svar forandret måten vi tenker om informasjon på.

Anta at vi har en kilde med $m_1, m_2, \ldots, m_M$ tillatte meldinger med sannsynlighet $p_1, p_2, \ldots, p_M$ hvor $p_1 + p_2 + \ldots p_M = 1$. Informasjonsmengden i en gitt melding p_k er definert som:

$$I_k \equiv log_2\left(\frac{1}{p_k}\right)$$

Å definere informasjonsmengde i en melding som det inverse av sannsynlighet for samme melding virker naturlig. Jo mindre sannsynlig en melding er, jo mer informasjon inneholder den. Å få vite at utenomjordiske vesener har landet, er absolutt mer

sensasjonelt enn at USAs president Trump har sendt en ny Twitter-melding. I tillegg brukte Shannon logaritme med base 2, slik at informasjonsmengden kunne uttrykkes i bit. Dersom det kun er en melding m_1 i kilden, er sannsynligheten for den meldingen $p_1 = 1$, og informasjonsinnholdet er 0.

$$I_1 = log_2\left(\frac{1}{1}\right) = 0$$

Dersom alle M meldinger er like sannsynlige og $M = 2^N$, er informasjonen i hver melding:

$$I = log_2 M = log_2\left(2^N\right) = N \text{ bit}$$

Dersom meldinger er uavhengige, er informasjonsmengden i den sammensatte melding lik summen av informasjonsmengden i enkeltmeldingene.

$$I_{k,l} = log_2\left(\frac{1}{p_k p_l}\right) = log_2\left(\frac{1}{p_k}\right) + log_2\left(\frac{1}{p_l}\right) = I_k + I_l$$

La oss igjen se på en kilde med M ulike meldinger m_1, $m_2, \ldots, m_M$ med sannsynlighet p_1, $p_2, \ldots, p_M$. En lang sekvens L av disse meldingene fra kilden vil gi $p_1 L$ meldinger av m_1 og $p_2 L$ meldinger av m_2 etc. Den totale informasjonen i meldingssekvensen vil være:

$$I_{total} = p_1 L\ log_2\left(\frac{1}{p_1}\right) + p_2 L\ log_2\left(\frac{1}{p_2}\right) + \ldots + p_M L\ log_2\left(\frac{1}{p_M}\right)$$

Den gjennomsnittlige informasjon per meldingsintervall L kalt entropi H blir:

$$H \equiv \frac{I_{total}}{L} = p_1 log_2\left(\frac{1}{p_1}\right) + p_2 log_2\left(\frac{1}{p_2}\right) + \ldots + p_M log_2\left(\frac{1}{p_M}\right) = \sum_{k=1}^{M} p_k log_2\left(\frac{1}{p_k}\right)$$

Shannon hentet begrepet entropi fra statistisk fysikk. Brukt i informasjonsteori kan entropi ses på som et mål på uforutsigbarhet/usikkerhet. Når en har kun en melding ($p_1 = 1$), blir den gjennomsnittlige informasjonen $H = 0$, og en har fullstendig forutsigbarhet. Dersom en har en meget usannsynlig melding ($p_k \to 0$ og $I_k \to \infty$), vil den gjennomsnittlige informasjon også være 0, da:

$$\lim_{p_k \to 0} p_k log_2\left(\frac{1}{p_k}\right) = 0$$

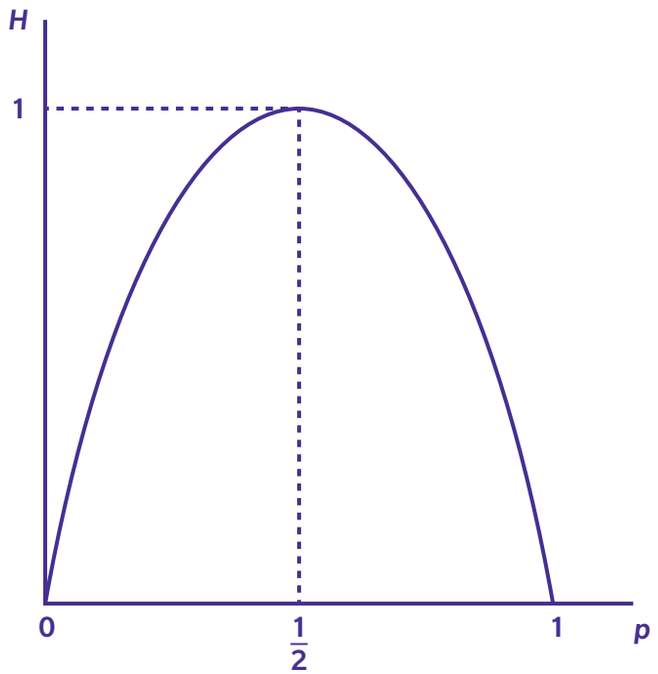
Det er lite gjennomsnittsinformasjon i en melding som er så sjelden at vi aldri ser den, men skulle meldingen mot formodning dukke opp, ville det være mye informasjon i den. Mitt favorittord, floccinaucinihilipilifikasjon, som er langt som et vondt

år, dukker ikke opp så ofte, og da kan det være godt å vite at dets betydning er å anse noe som verdiløst.

Av uttrykket for entropien H ser vi at når den vokser, stiger også usikkerheten om hva som kommer med neste melding. Når er H maksimal og forutsigbarheten minst? La oss ta et overkommelig eksempel med to meldinger med sannsynlighet p og $(1-p)$.

$$H = p\ log_2\left(\frac{1}{p}\right)+(1-p)\ log_2\left(\frac{1}{(1-p)}\right)$$

I figur 16.26 ser vi hvordan H varierer med p.



Figur 16.26 Entropien *H* som funksjon av sannsynlighet *p*

H er maksimal når $\frac{dH}{dp}=0$ ved $p=\frac{1}{2}$.

$$H_{Maks}=\frac{1}{2}log_2(2)+\frac{1}{2}log_2(2)=1 \text{ bit/melding}$$

I dette tilfellet kunne vi ha brukt symbolet 0 for den ene meldingen og 1 for den andre meldingen.

Fra dette eksempelet kan en trekke slutningen, som vi vil ha glede av om ikke så lenge, at når en har M meldinger, blir H maksimal når alle meldinger er like sannsynlige ($p=\frac{1}{M}$).

$$H_{Maks} = \sum \frac{1}{M} log_2(M) = log_2(M)$$

Alt dette er vel og bra, men hvordan skal det brukes til å finne koder med maksimal komprimering? Vel, entropien gir oss et mål, basert på hyppighet av ulike meldinger, på hvor mye en kan komprimere. Jo nærmere en komprimeringskode kommer entropien, jo mer optimal er den. Finnes det gode metoder for å komprimere? Absolutt! Siden Shannon ga ut sin artikkel, har mange brukt livet sitt på å nærme seg H.

La oss starte med Shannon-Fano-koding ved hjelp av et eksempel.

Tabell 16.1 Shannon-Fano-koding

M	$p(m_k)$			
m_1	0,25	0	0	
m_2	0,25	0	1	
m_3	0,2	1	0	
m_4	0,15	1	1	0
m_5	0,15	1	1	1

Tabell 16.1 viser fem meldinger m_1 til m_5 med tilhørende sannsynligheter og tilordnet bitmønster. Tabellen er sortert etter sannsynlighet. En starter med å gruppere meldingene i cirka 2 deler, og det fortsetter en med til alle meldingene har fått tildelt et bitmønster. En ser at de meldingene som opptrer sjeldnest, m_4 og m_5, har de lengste symbolene, 3 bit, mens m_1 til m_3 som er hyppigst, er tildelt 2 bit symbol. Den midlere kodelengden for Shannon-Fano i dette tilfellet blir:

$$l = \sum_{k=1}^{5} l_k p(m_k) = 2 \cdot 0,25 + 2 \cdot 0,25 + 2 \cdot 0,2 + 3 \cdot 0,15 + 3 \cdot 0,15 = 2,3 \text{ bit/melding}$$

l_k representerer lengden på de ulike symbolene. Er denne koden optimal med henhold på komprimering? For å finne ut det er det bare å sammenligne med entropien H.

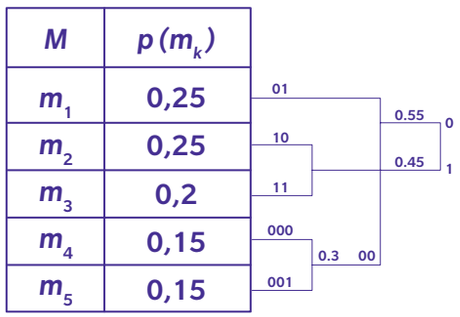
$$H = \sum_{k=1}^{5} p_k log_2\left(\frac{1}{p_k}\right) = 0,25log_2\left(\frac{1}{0,25}\right) + 0,25log_2\left(\frac{1}{0,25}\right)$$
$$+0,2log_2\left(\frac{1}{0,2}\right) + 0,15log_2\left(\frac{1}{0,15}\right) + 0,15log_2\left(\frac{1}{0,15}\right) = 2,2855 \text{ bit/melding}$$

Ikke særlig langt unna, og vi kan finne kodens redundans ved å sammenligne den midlere kodelengden og H.

$$\eta = \frac{l - H}{H} = 0,0064 = 0,64\ \%$$

Koden vi har laget, $m_1 = 00$, $m_2 = 01$, $m_3 = 10$, $m_4 = 110$ og $m_5 = 111$, har i tillegg en annen sjarmerende egenskap. Den kan dekodes instantant. Mottar vi 00, 01 eller 10, så vet vi at symbolet er avsluttet, og mottar vi 11, så vet vi at det kommer et bit til, enten 0 eller 1, før det er slutt på symbolet. Vi trenger ingen opplysning om når et symbol ender, da det er bakt inn i koden.

En annen viktig kodemetode er Huffman-koding. Den er optimal – å vise det er et langt lerret å bleke, og det overlater jeg til andre – og dermed det nærmeste man kan komme den hellige gral H. La oss se på det samme eksempelet som for Shannon-Fano.



Figur 16.27 Huffman-koding

I Huffman-koding starter en med å se på symbolene med minst sannsynlighet og slå dem sammen. Det fortsetter en med til en får sannsynlighet 1 og trestrukturen vist til høyre i figur 16.27. Etterpå tildeler en 0 og 1 i hvert forgreiningspunkt fra rot til løvet. Vi ser at vi også i dette tilfellet får en instantan kode med tre symboler med lengde 2 og to symboler med lengde 3 . Redundans for Huffman-koden blir i dette tilfellet lik den vi fikk for Shannon-Fano-koden.

De to eksemplene på koding har forutsatt at vi kjenner frekvensen til meldingene i kilden. Hva om så ikke er tilfellet? Er det mulig å lage seg en kodebok mens en koder som nærmer seg entropien asymptotisk? Det hadde vært kjekt, for da hadde en ikke trengt å kjenne alle kildemeldingene før en begynte å sende. Lempel-Ziv algoritmene som blant annet brukes for å «zippe» filer, er eksempler på slik koding. Se for deg at du ønsker å kode følgende bitmønster:

$$010110111010|011001000\ldots\ldots\ldots$$

Her har en kodet frem til |. For å kode bitmønsteret til høyre for | finner vi det lengste bitmønsteret til venstre for | som vi allerede har kodet. I dette tilfellet er det gitt med understrekningen:

$$01\underline{0110}111010|\underline{0110}01000\ldots\ldots\ldots$$

De fire bitene som er understreket til høyre, blir kodet som en referanse $(k,l)=(10,4)$ hvor k er avstanden til det tidligere like bitmønsteret, og 4 er lengden på det. Så er det bare å finne neste maksimale segment:

$$01011011\underline{0100}110|\underline{0100}0\ldots\ldots\ldots$$

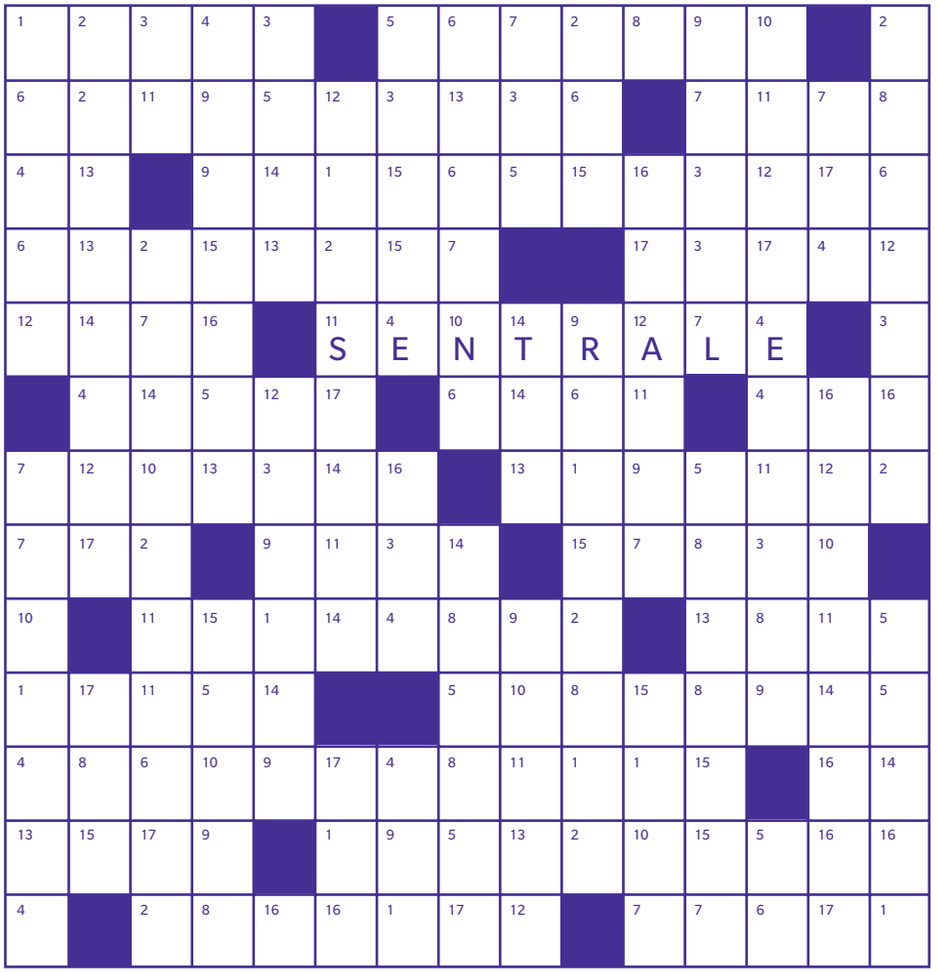
Referansen for dette segmentet blir $(k,l)=(7,4)$. Ved dekoding brukes allerede dekodet bitmønster sammen med de påfølgende referanser til å gjenskape det opprinnelige bitmønster. Når det blir mange meldinger, går denne algoritmen mot entropien som grense, og er dermed en effektiv måte å kode data på.

Har du noen gang prøvd å løse et kodekryssord? I figur 16.28 er det et du kan prøve deg på. Tallene i rutene representerer bokstaver, og i tillegg får du et nøkkelord som hjelp med på veien.

1	2	3	4	3	■	4	3	5	2	6	7	8	■	2
9	6	5	5	2	10	1	11	12	13	■	14	7	2	13
3	9	■	15	4	12	11	■	4	7	5	15	11	13	12
5	6	16	17	12	17	18	12	■	18	7	11	19	3	17
12	5	5	12	■	16	12	1	8	12	13	12	11	■	16
■	5	3	11	9	12	■	2	7	17	13	■	3	2	12
2	1	2	6	3	5	13	■	12	13	12	11	17	6	13
4	9	8	■	11	1	12	2	■	19	11	12	2	13	■
3	■	3	9	2	13	12	8	13	12	■	4	20	3	2
8	7	2	12	12	13	■	21	11	9	21	4	12	17	13
2	13	20	12	5	12	11	■	12	6	8	12	■	13	7
5	12	1	17	■	11 R	6 I	18 D	12 E	2 S	12 E	17 N	13 T	12 E	11 R
1	■	17	12	18	6	2	12	13	■	17	12	9	17	12

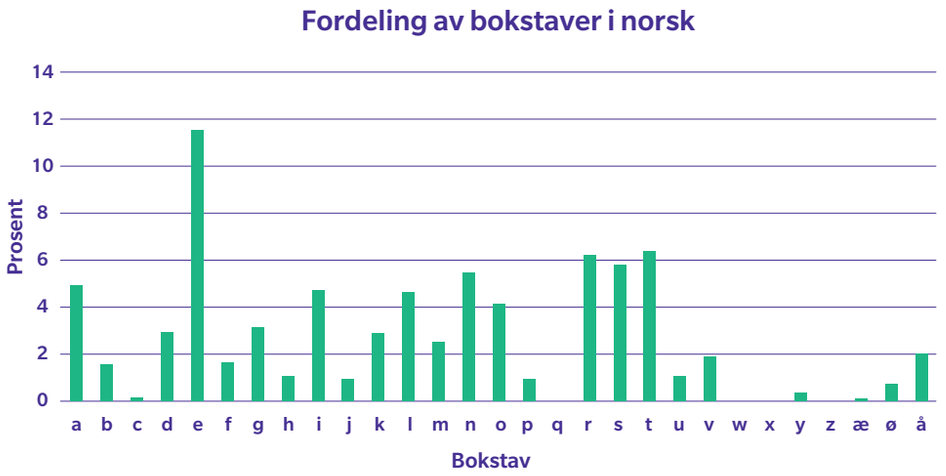
Figur 16.28 Kodekryssord

Gikk det greit? Hvilken løsningsstrategi brukte du? Fasit finner du i appendiks. Lyst på et til? Jeg foreslår at du prøver det i figur 16.29 som er en smule mer utfordrende.



Figur 16.29 Enda et kodekryssord

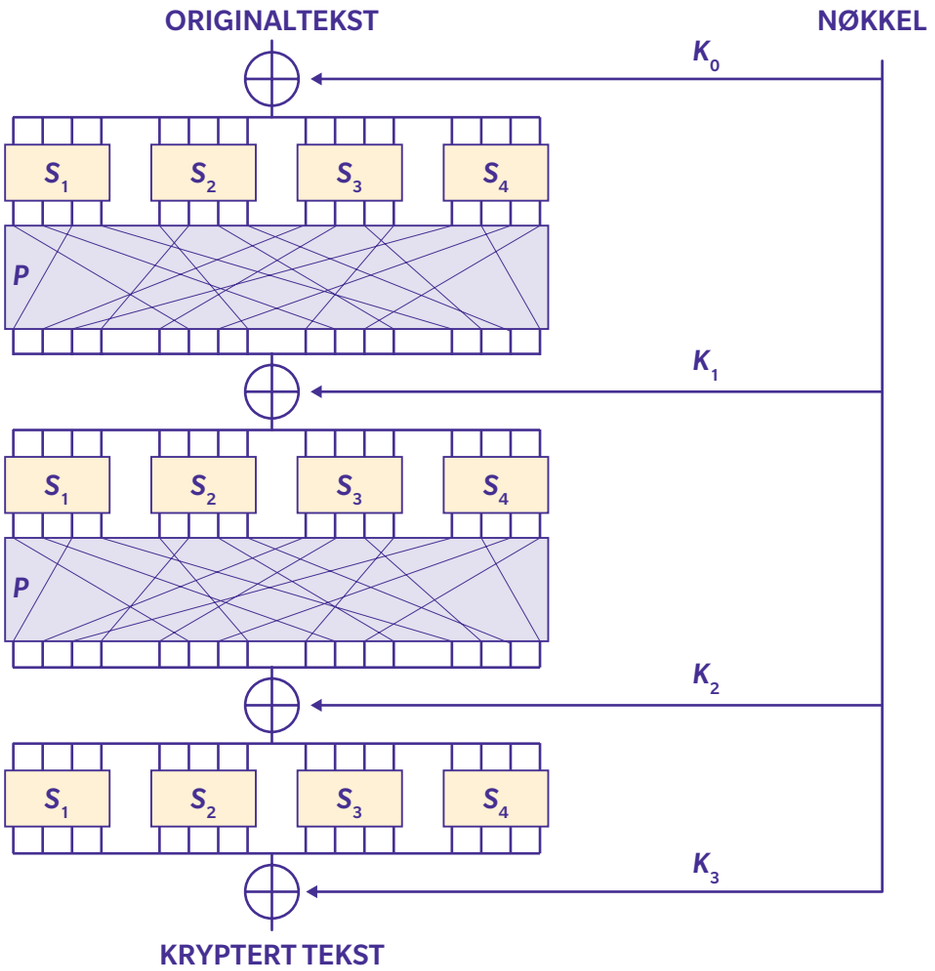
Som liten gutt elsket jeg kodekryssord og oppdaget etter hvert at frekvensen av de ulike bokstavene i det norske språk varierte. Ved bare å telle antall forekomster av tall samt foreta en enkel analyse av deres posisjon og sammensetning, kunne en finne en rekke bokstaver, og da ga ofte resten seg selv, og mestringsgleden var et faktum. Jo større kryssordet var, jo mer tekst var til rådighet, og analysen ble enda enklere. Figur 16.30 viser fordelingen av bokstaver i norsk.



Figur 16.30 Omtrentlig fordeling av bokstaver i norsk

Fikk du problemer med det siste kodekryssordet? Det var meningen. Der er all frekvensinformasjon fjernet, og innholdet er utsatt for en tilfeldighetsprosess. Koden er nærmest uknekkelig. På 1930-tallet fikk den engelske etterretningen, uten at noen fikk vite det, overta kryssordspalten i The Daily Telegraph og laget stadig vanskeligere kryssord og koder. Leserne sendte inn sine svar, og de flinkeste fikk plutselig en ny jobb.

Dersom vi ønsker å sende informasjon fra A til B uten at andre enn sender og mottaker skal forstå budskapet, må informasjonen (data) krypteres. I tillegg til konfidensialitet må vi være sikre på at data som mottas ikke er korrumpert og virkelig kommer fra senderen vi kommuniserer med. Som vi så med eksempelet med kodekryssord, er det en god strategi å fjerne all frekvensinformasjon og umiddelbar sammenheng i det som vi ønsker å overføre. Det skjer som oftest i en randomiseringsprosess basert på permutasjoner og substitusjoner. I figur 16.31 er det vist et slikt nettverk som tar 16 bit originaltekst som substitueres og permuteres i fire runder med ulike nøkler. Nøklene er et bitmønster som XORes med den stadig mer randomiserte originalteksten.



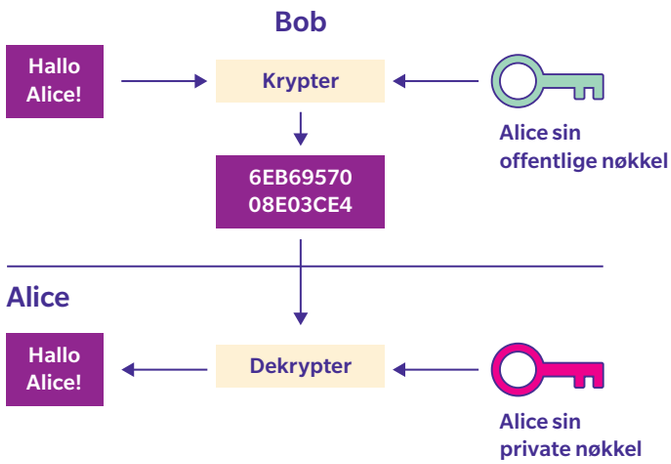
Figur 16.31 Substitusjon og permutasjonsnettverk

I blokkene merket med S foretas det substitusjon, og i blokkene P foretas det permutasjoner som angitt med strekene gjennom blokkene. Den mest brukte standarden som bruker denne metoden for å randomisere, er AES (Advanced Encryption Standard). Jo lengre nøkler, jo vanskeligere blir koden å knekke.

Hvorfor? Vel, la oss returnere til informasjonsteorien for et øyeblikk. Dersom nøkkelen er 8 bit, vil entropien i den krypterte informasjonen være $H = log_2(8) = 3$ siden vi antar at alle 8 bit meldinger er like sannsynlige på grunn av randomiseringsprosessen. Jo lengre nøkkel, jo større vil entropien og usikkerheten bli, da antallet forskjellige meldinger øker med toerpotensen av antall bit i nøkkelen. Entropien i den krypterte informasjonen er altså større enn i originalinformasjonen. Dersom en velger en nøkkel like lang som informasjonen som sendes, har man en såkalt perfekt kode som det ikke

er mulig å knekke. Da ser en en fullstendig randomisert melding bare én gang og kan ikke gjenskape originalteksten. Dersom nøkkelen er vesentlig kortere enn originalteksten, har man derimot mange krypterte meldinger som kan analyseres i håp om å gjenskape originalteksten. Den perfekte kode ble først foreslått av den amerikanske ingeniøren Gilbert S. Vernam i 1917 og ble brukt på den såkalte «hot-line» mellom USA og Sovjetunionen under den kalde krigen. En nøkkel som er like lang som den opprinnelige informasjon er upraktisk i bruk, og AES anses som rimelig sikker med nøkler på 192 eller 256 bit.

«Ja, tenke det; ønske det, *ville* det med; – men *gjøre* det! Nej; det skjønner jeg ikke!» Jeg vet ikke om Diffie og Hellman hadde lest sin Peer Gynt, men det kan virke sånn når man leser hva de tenkte og ønsket seg, i artikkelen «Privacy and Authentication» fra 1976. Der foreslo de en såkalt asymmetrisk nøkkel. Én nøkkel skulle brukes til kryptering (den offentlige) og én til dekryptering (den private). Selv om en kjente krypteringsnøkkelen, skulle den ikke kunne brukes til å finne dekrypteringsnøkkelen. Denne tanken var revolusjonerende og gjorde det mulig å ha offentlige nøkler for kryptering. Krypteringsnøkkelen kunne en gi til gud og hvermann og allikevel sikre seg privat kommunikasjon med de enkelte, da ingen av dem en kommuniserte med kjente den private dekrypteringsnøkkelen. Figur 16.32 viser en prinsippskisse for bruk av offentlig og privat nøkkel for kryptering. Diffie og Hellman visste ikke hvordan dette skulle løses i praksis, men de trengte ikke å vente lenge før det dukket opp løsninger.



Figur 16.32 Kryptering med bruk av offentlig og privat nøkkel

Året etter ble RSA (Rivest Shamir Alderman) algoritmen publisert. De tre herrene fra MIT (Massachusetts Institute of Technology) foreslo å bruke store primtall og det faktum at det er mye lettere å multiplisere enn å faktorisere. Metoden går som følger (Computer Networks [11] side 412):

1) Velg to store primtall, p og q, som begge er større enn 10^{100}
2) Regn ut $n = pq$ og $z = (p-1)(q-1)$
3) Velg et tall som er et godt primtall i forhold til z, og kall det d
4) Finn e slik at $ed = 1 \, mod \, z$

For å kryptere en melding P (i dette tilfellet en bokstav) regner man ut $C = P^e \, (mod \, n)$, og for å dekryptere må en foreta $P = C^d \, (mod \, n)$. For å kryptere trenger en paret e og n, og for å dekryptere d og n.

La oss ta et trivielt eksempel, med små primtall, for å forstå algoritmen. Vi velger $p = 3$ og $q = 11$ som gir $n = 3 \cdot 11 = 33$ og $z = (3-1)(11-1) = 20$. Et fornuftig valg av d blir 7, da 7 og 20 ikke har noen felles faktorer. Med disse valgene finner en at $e = 3$, da $3 \cdot 7 = 21 = 1 \, mod \, 20$. Figur 16.33 viser hvordan originalteksten SUZANNE krypteres med den offentlige nøkkel og dekrypteres med den private.

Originaltekst (P)		Kryptert tekst (C)			Etter dekryptering	
Symbolsk	tallverdi	P^3	P^3 (mod 33)	C^7	C^7 (mod 33)	Symbolsk
S	19	6859	28	13492928512	19	S
U	21	9261	21	1801088541	21	U
Z	26	17576	20	1280000000	26	Z
A	01	1	1	1	1	A
N	14	2744	5	78125	14	N
N	14	2744	5	78125	14	N
E	05	125	26	8031810176	5	E
		Senders beregning		Mottakers beregning		

Figur 16.33 Krypteringseksempel med RSA algoritmen

Siden primtallene som er valgt i eksempelet er så små, må alle meldingene som kodes gis en tallverdi, symbol, mindre enn 33 for at den krypterte teksten skal være en entydig. Resultatet i dette tilfellet blir en substitusjonskode (melding blir byttet ut mot en annen) på bokstavnivå. Hadde vi derimot valgt p og $q \cong 10^{100}$, ville $n \cong 10^{200}$, og meldingene kunne hatt symbollengde på opptil 664 bit ($2^{664} \cong 10^{200}$).

En utfordring med RSA algoritmen er at den er rimelig treg, så i praksis anvender en ofte RSA eller lignende algoritmer for distribusjon av nøkler til raskere symmetriske krypteringsalgoritmer som for eksempel AES.

Vi har nå sett på hvordan en komprimerer og krypterer data i kilden og tilpasser data til transmisjonsmediet. Det neste spørsmålet er hvor fort vi kan få overført data fra sender til mottaker. Dersom kilden genererer meldinger med en rate r meldinger per sekund, defineres informasjonsraten som:

$$R \equiv rH$$

R er da gjennomsnittlig antall bit informasjon per sekund.

Eksempel: Et signal begrenset av B Hz og samplet med Nyquistrate ($2B$) til 4 nivåer. m_1 til m_4 er uavhengige med sannsynligheter $p_1 = p_4 = 1/8$ og $p_2 = p_3 = 3/8$. Gjennomsnittlig informasjon (entropi) H:

$$H = \frac{1}{8}log_2\left(8\right) + \frac{3}{8}log_2\left(\frac{8}{3}\right) + \frac{3}{8}log_2\left(\frac{8}{3}\right) + \frac{1}{8}log_2\left(8\right) = 1{,}8 \text{ bit/melding}$$

Informasjonsraten blir:

$$R = rH = 2B\left(1{,}8\right) = 3{,}6B \text{ bit/sekund}$$

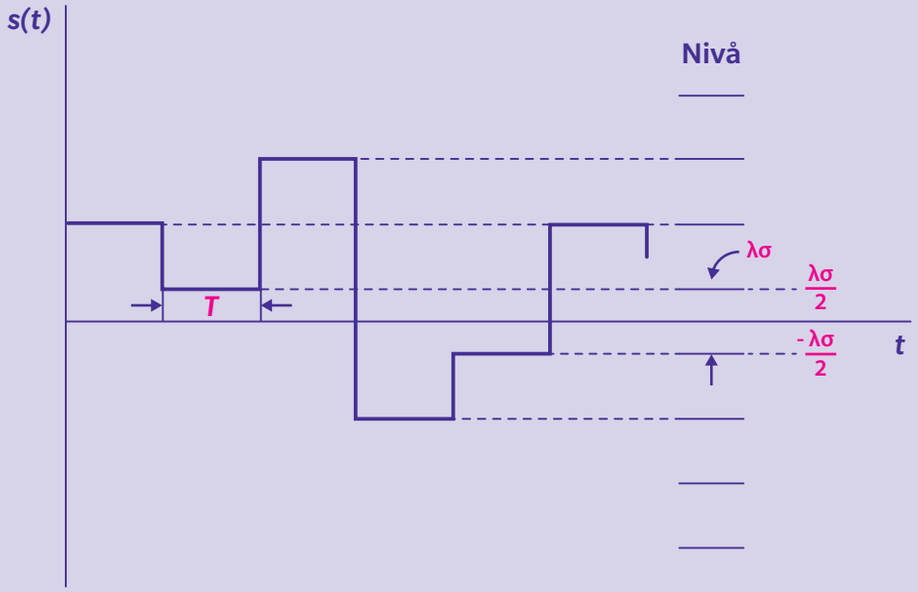
Hvor mye av denne informasjonsstrømmen klarer kanalen å svelge unna før det går i stå? Dette var også en av de tingene Shannon fant svar på i sin artikkel «A Mathematical Theory of Communication». Det han fant ut, var at en kunne overføre informasjon med en vilkårlig liten sannsynlighet for feil såfremt informasjonsraten R var mindre eller lik C som kalles kanalkapasiteten. Kanalkapasiteten i en kanal med begrenset båndbredde og gaussisk støy (gaussisk fordelt med middelverdi 0) som er typisk for kommunikasjonssystemer, fant Shannon ut var:

$$C = Blog_2\left(1 + \frac{S}{N}\right) \text{ bit/sekund}$$

Hvor B er kanalbåndbredden, S signalstyrken og N den totale støy innenfor kanalbåndbredden.

For å nærme seg C i en kanal med støy må det brukes kanalkodingsteknikker som vi skal se på etter vi har prøvd å rettferdiggjøre formelen for kanalkapasitet C (Principles of Communication Systems [12] side 421).

Anta at du har et system som sender signaler $s\left(t\right)$ med faste spenningsverdier. Det mottatte signal er belemret med støy som har en RMS-verdi σ. RMS («Root Mean Square») er et mål for den midlere støy. Intervallavstandene er valgt slik at en får en akseptabel feilrate. Figur 16.34 gir en skisse av signalnivåer med tilhørende støygrenser.



Figur 16.34 Signalnivåer og støygrenser

Anta videre et jevnt antall nivåer:

$$\pm\lambda\sigma/2,\ \pm 3\lambda\sigma/2,\ \pm 5\lambda\sigma/2,\ \ldots$$

Med M mulige meldinger må det være M nivåer, og vi antar at alle er like sannsynlige. Den gjennomsnittlige signalstyrken (signaleffekten) blir:

$$S=\frac{2}{M}\left\{\left(\frac{\lambda\sigma}{2}\right)^2+\left(\frac{3\lambda\sigma}{2}\right)^2+\ldots+\left(\frac{(M-1)\lambda\sigma}{2}\right)^2\right\}$$

Her er det multiplisert med 2 siden vi har symmetri av nivåer rundt 0, og delt på M da vi ønsker å finne det gjennomsnittlige nivået. Er det mulig å skrive dette mer kompakt? Absolutt, med litt hjelp av et triks Gauss viste sin lærer på barneskolen.

$$S=\frac{2}{M}\left(\frac{\lambda\sigma}{2}\right)^2\left\{1^2+3^2+\ldots+(M-1)^2\right\}$$

$$S=\frac{1}{2M}\left\{\frac{M\left(M^2-1\right)}{6}\right\}(\lambda\sigma)^2=\frac{\left(M^2-1\right)}{12}(\lambda\sigma)^2$$

Snur vi rundt på flisen, finner vi antall nivåer for en gitt signalstyrke:

$$M = \left(1 + \frac{12S}{\lambda^2 \sigma^2}\right)^{1/2} = \left(1 + \frac{12S}{\lambda^2 N}\right)^{1/2} \quad hvor\ N = \sigma^2$$

Støyeffekt N er altså gitt som RMS-støyspenning σ kvadrert.

Siden alle meldinger M var like sannsynlige, overfører hver melding en gjennomsnittlig informasjonsmengde:

$$H = log_2\left(M\right) = log_2\left(1 + \frac{12S}{\lambda^2 N}\right)^{1/2} = \frac{1}{2}\log_2\left(1 + \frac{12}{\lambda^2}\frac{S}{N}\right) \text{ bit/melding}$$

Ifølge Nyquist er meldingsraten:

$$r = \frac{1}{T} = 2B \text{ meldinger/sekund}$$

Her er T et gitt tidsintervall som det tar for å lese av et nivå med ønsket sikkerhet.

Den gjennomsnittlige informasjonsraten er $R = rH$, og siden vi har laget det slik at vi kan tåle en akseptabel mengde feil, blir kanalkapasiteten:

$$C \approx R = rH = Blog_2\left(1 + \frac{12}{\lambda^2}\frac{S}{N}\right)$$

Velger vi $\lambda = \sqrt{12}$, blir $12/\lambda^2 = 1$, og vi har funnet det vi er på jakt etter. Dersom du ønsker å finne ut hvordan en kommer frem til dette resultatet for en vilkårlig liten feil, må du lese Shannons originalpapirer.

Hvor går så grensen? Vel, den fant jo Shannon ut at gikk ved $R = C$. R kan ikke bli større enn C. Dersom det skulle være tilfellet, vil vi ende i det digitale Mordor hvor det går feil hele tiden. La oss sette $R = C$ og finne et uttrykk for grensen, men før det må det gjøres noen forpostfektninger.

$$\frac{C}{B} = log_2\left(1+\frac{S}{N}\right)$$

$$2^{\frac{C}{B}} = \left(1+\frac{S}{N}\right)$$

$$\frac{S}{N} = \left(2^{\frac{C}{B}} - 1\right)$$

La oss omforme S/N til energi per bit (E_b) og støy per Hertz (N_0) for å følge en gyllen tradisjon.

$$\frac{S}{N} = \frac{S \cdot T}{N \cdot T} = \frac{E_b}{N \cdot T} = \frac{E_b}{N_0 BT} = \frac{E_b R}{N_0 B}$$

Her er $E_b = S \cdot T$ siden S er effekten og T er tiden per bit. Støyen i hele båndbredden er N, og da er støy per Hertz lik N_0.

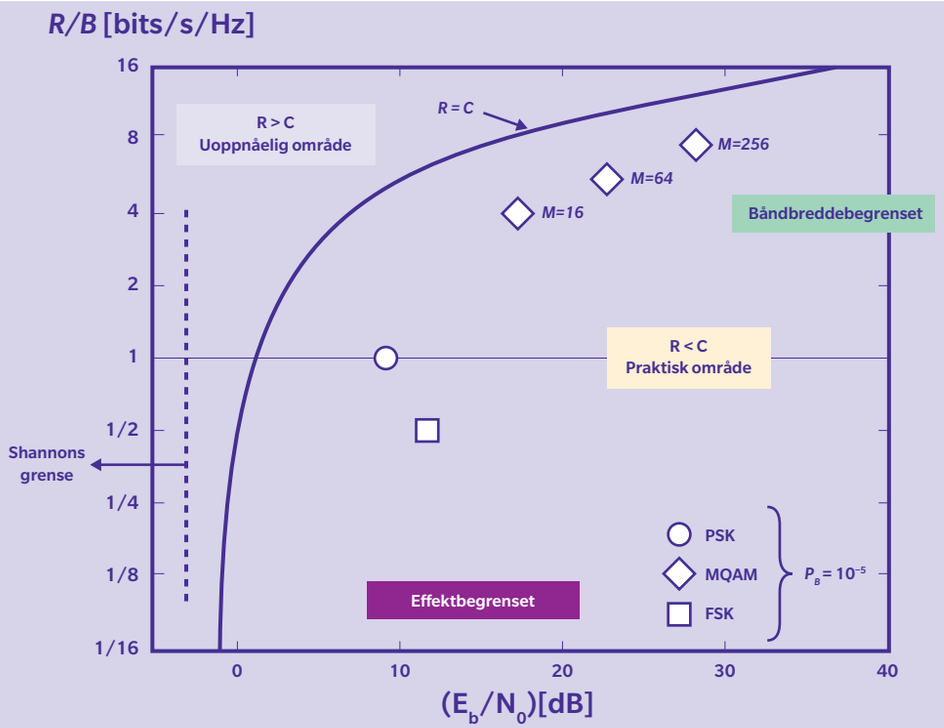
Så setter vi inn grensen $R = C$.

$$\frac{S}{N} = \left(2^{\frac{C}{B}} - 1\right)$$

$$\frac{E_b R}{N_0 B} = \left(2^{\frac{C}{B}} - 1\right)$$

$$\frac{E_b}{N_0} = \frac{B}{C}\left(2^{\frac{C}{B}} - 1\right) = \frac{2^{\frac{C}{B}} - 1}{\frac{C}{B}}$$

Figur 16.35 viser R/B som funksjon av E_b / N_o. Grensen $R = C$ er tegnet inn, og i tillegg er noen ulike modulasjonsmetoder plottet inn ved feilrate på $P_B = 10^{-5}$. Jo høyere modulasjonsrate, $R/B \frac{\text{bit}}{\text{sekund}}/Hz$, jo større må E_b / N_o dB være. For eksempel vil et system med seksten tilstander, 16 QAM, som har fire bit per symbol som overføres, trenge $E_b / N_o \geq 15\,\text{dB}$ for å gi feilfri transmisjon.



Figur 16.35 *R/B* som funksjon av E_b/N_o

Hvordan går det med grensen når båndbredden går mot uendelig? Hva blir da forholdet E_b / N_o?

$$B \to \infty \quad \frac{C}{B} \to 0$$

$$\frac{E_b}{N_0} = \lim_{\frac{C}{B} \to 0} \left(\frac{2^{\frac{C}{B}} - 1}{\frac{C}{B}} \right)$$

Ooops! Da deles det på null, og vi får et problem. Heldigvis for oss, fant matematikeren L'Hôpital løsningen på dette problemet ved isteden å se hvor grensen for de deriverte går. Det gir nemlig samme resultat. Vi deriverer derfor med hensyn på $\frac{C}{B}$ over og under brøkstreken:

$$\frac{E_b}{N_0} = \lim_{\frac{C}{B} \to 0} \left(\frac{2^{\frac{C}{B}} ln2}{1} \right) = \left(\frac{ln2}{1} \right) = -1{,}6 \text{ dB}$$

Det er det den svarte streken for $R = C$ i figur 16.35 nærmer seg mot i nederste venstre hjørne.

Formelen til Shannon indikerer at en støyfri gaussisk kanal har uendelig kapasitet fordi $S/N = \infty$. Problemet er bare at når båndbredden øker, så øker også støyen. La oss trylle litt med formelen.

$$C = Blog_2\left(1+\frac{S}{N_0B}\right) = \frac{S}{N_0}\frac{N_0B}{S}log_2\left(1+\frac{S}{N_0B}\right) = \frac{S}{N_0}log_2\left(1+\frac{S}{N_0B}\right)^{\frac{N_0B}{S}}$$

For å komme videre henter vi litt hjelp fra matematikken: $\lim_{x\to 0}(1+x)^{1/x} = e$.

$$\lim_{B\to\infty} C = \frac{S}{N_0}log_2(e) = 1,44\frac{S}{N_0}$$

Vi ser at det vi kan håpe på å få til med stor båndbredde, er gitt av tilgjengelig signalstyrke og støyen per Hertz.

Vi har til nå sett på kanaler med gaussisk støy. Er det en stor begrensning? Nei, de fleste kanaler er til første orden gaussiske av natur, og folk som har brukt vesentlig mer tid enn oss på dette emnet, har vist at dersom en kan få en gitt feilrate i et gaussisk system, så kan det oppnås en enda lavere feilrate i et ikke-gaussisk system.

Kanalkoder

Alt som kan gå galt, går galt. ~~Peters~~ Murphys universelle erkjennelse/lov gjelder også for lagring og sending av binære tall. Derfor utviklet en tidlig koder som kunne oppdage og eventuelt også rette feil. For tidskritisk data som telefoni, tale og video brukes det feilkorrigerende koder, mens det for ikke-tidskritisk data som for eksempel filoverføring brukes feildetektering med tilhørende retransmisjonsmekanismer. Ved å legge tilleggsinformasjon til det binære tallet kan en øke feiltoleransen. Jo mer ekstra informasjon (redundans), jo mer feil tåler det resulterende kodeordet uten at det blir feil i det opprinnelige binære tallet.

Paritetssjekk

Den enkleste formen for feildeteksjon er å legge et ekstra bit til det opprinnelige binære tallet. Det gjøres på en slik måte at den resulterende koden enten har jevn eller odde paritet. At en kode har jevn paritet, betyr at antallet 1 bit er jevnt, og ved odde paritet er antallet 1 bit odde. En må selvfølgelig velge om en vil gå for jevn eller odde paritet. I tabell 16.2 er paritet for BCD koden vist.

Tabell 16.2 BCD kode med paritetsbit

Jevn paritet		Odde paritet	
P	BCD	Desimal	BCD
0	0000	1	0000
1	0001	0	0001
1	0010	0	0010
0	0011	1	0011
1	0100	0	0100
0	0101	1	0101
0	0110	1	0110
1	0111	0	0111
1	1000	0	1000
0	1001	1	1001

Hvis en mottar et kodeord og det har feil paritet, så vet en at det er feil i et eller flere bit, men man kan ikke gjenskape det opprinnelige binære tallet. Når en bruker paritetssjekk, må en i tillegg ha en mekanisme som spør om å få tilsendt det binære tallet på nytt når noe går feil.

Dersom en mottar et kodeord med riktig paritet, kan en dessverre ikke være fullstendig sikker på feilfrihet. Paritetssjekk oppdager kun et odde antall feil.

La oss se for oss at vi har et odde paritetssystem og mottar følgende kode: **1**0110, **1**1010, **1**1011. Kodens ekstra bit er fetet, og om det ekstra bit settes først eller sist, varierer med ulike design. De to første tallene har tre 1 bit og dermed odde paritet, mens det siste har fire 1 bit og jevn paritet. Vi kan derfor være sikre på at det har skjedd en feil med siste kode.

CRC

En mye brukt kode for å sende binærdata over en kommunikasjonslinje er «Cyclic Redundancy Check». Den kan oppdage 1 eller 2 bit feil, og dersom den er velkonstruert, kan den også oppdage såkalte «burst»-feil hvor det er mange bitfeil etter hverandre. I CRC adderes en sjekksum etter databitene, og på mottakersiden sjekkes så databiter samt «checksum» for å se om det har oppstått feil underveis. Sjekksummen genereres ved å dele koeffisientene fra et generatorpolynom mod 2 på databitene. På mottakersiden divideres den mottatte kode mod 2 med sjekksummen, og blir det ingen

rest, så vet en at overføringen har vært vellykket. Det finnes en rekke polynomer som kan brukes med ulike ønskede egenskaper. La oss ta et eksempel med generatorpolynom $1 \cdot x^3 + 0 \cdot x^2 + 1 \cdot x^1 + 0 \cdot x^0$. Dette polynomet kan beskrives ved et 4 bit binært tall $G = 1010$. La databitene være *D*=11010011. Adder så fire 0ere til *D*. Ved å dividere *D*0000 mod 2 (XOR) på G får man en rest som adderes til D for å få kodeordet D' som sendes. Siden G er 4 bit, så må D' være 4 bit lengre enn D.

```
11010011 0000 : 1010
1010
----
 1110
 1010
----
  1000
  1010
----
   1011
   1010
----
    1000
    1010
----
     100
```

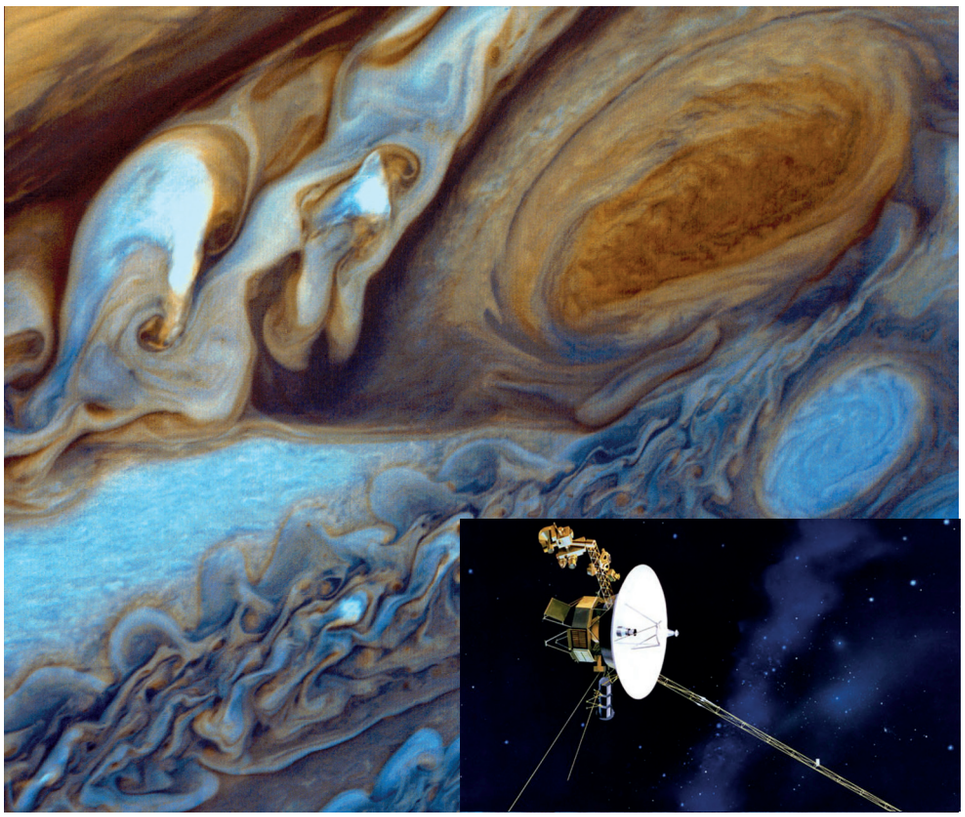
Resten er 100, og den adderes til D'= 11010011 0100. På den måten vet en at en ikke får noen rest ved divisjon mod 2 (XOR) på mottakersiden dersom overføringen er feilfri. Skulle en derimot være så uheldig at det for eksempel oppsto feil i andre bit slik at en mottok D'= 10010011 0100, vil en få rest ved divisjon og dermed ane ugler i mosen.

```
10010011 0100 : 1010
1010
----
 1100
 1010
----
  1101
  1010
----
   1111
   1010
----
    1010
    1010
----
     100
```

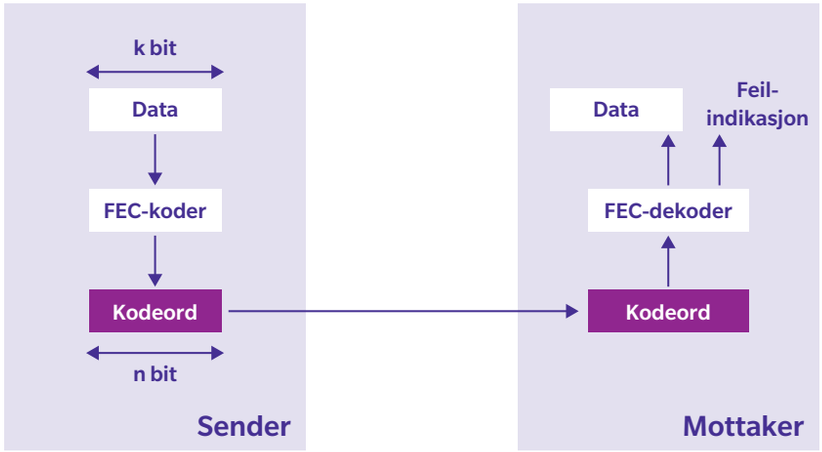
Resten i dette tilfellet er 100. Aldeles utmerket! Det bør bemerkes at jeg regnet stykket fire ganger før resultatet falt på plass, men fortvil ikke – øvelse gjør mester!

Feilkorrigerende koder

Dersom en ønsker å overføre data og ikke har mulighet til å be om å få data sendt på nytt, kan ulike strategier brukes. Mange velger for eksempel sin livsledsager med metoder som nevnt i innledningen til dette kapittelet. Et det fornuftig? Nei, da var nok metodene som kommunikasjonsingeniørene til Voyagersatellitten, figur 16.36, brukte, bedre fundert. De valgte først og fremst en lav datarate (160 bit/s) slik at de kunne bruke god tid til å skille informasjonen (0 og 1) fra støy. Siden kommunikasjonskanalen mellom satellitt og jordstasjon er belemret med lave signalnivåer og muligheten for å sende data på nytt er begrenset, ble det også valgt en feilkorrigerende kode med like mange kodebit som databit. Til alt overmål utstyrte de satellitten med enda en feilkorrigerende kode, en såkalt Reed-Solomon kode, hvor 20 % var kodebit. Sistnevnte feilkorrigerende kode ble brukt for å overføre data når støyen ikke var altfor plagsom.



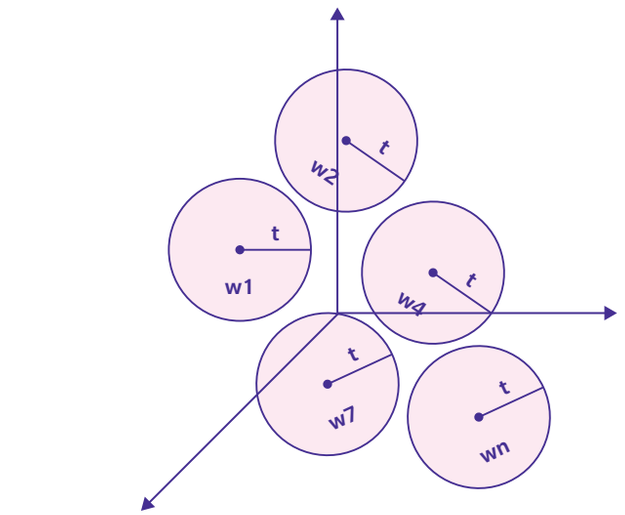
Figur 16.36 Voyagers bilde av Jupiters røde flekk. Vel verdt å vente på



Figur 16.37 Skjematisk fremstilling av en feilkorrigerende FEC («Forward Error Correction») koder

Figur 16.37 viser den prinsipielle oppbygging av en FEC («Forward Error Correction»). k bit med data gjøres om til et kodeord med n bit som sendes over til mottaker. Mottaker beregner en feilsum som forteller hvor korrumpert det sendte kodeordet er, og korrigerer dersom det er mulig.

Oppgaven for enhver kodemaker er å prøve å gjøre den perfekt. En perfekt kode er en kode hvor alle kodeord er like langt fra hverandre. Det høres kanskje enkelt ut, men generasjoner av matematikere har slitt med problemet.



Figur 16.38 En perfekt kode

Figur 16.38 viser kodemakernes hellige gral. Her er kodeordene $w_1, \ldots, w_n$ alle i like stor avstand $2t+1$ fra hverandre. Hva menes med avstand (metrikk)? Det finnes mange måter å definere et avstandsmål mellom kodeord på, og et som er mye brukt, er Hamming-avstand.

Hamming-avstanden mellom to n-bit binære sekvenser er definert som summen av posisjoner hvor sifrene er ulike. Et eksempel:

$$w_1 = 011011$$

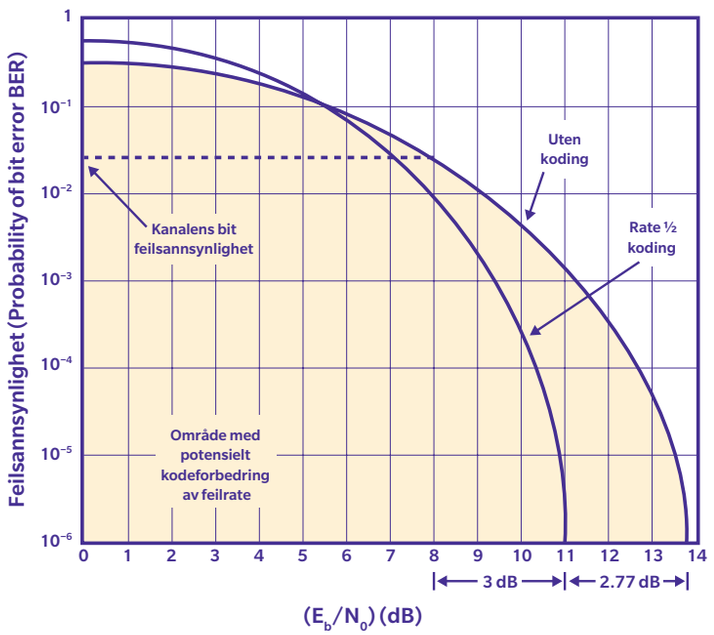
$$w_2 = 110001$$

$$d(w_1, w_2) = 3$$

Det kan vises at en kode med Hamming-avstand $2t+1$ kan rette opptil t bitfeil og oppdage feil på $2t$ bit.

Med en (n,k) kode er det 2^k lovlige kodeord ut av 2^n. Andelen redundante bit $\frac{(n-k)}{k}$ kalles redundansen til koden og $\frac{k}{n}$ for koderaten. For Voyager er redundansen $(2-1)/1 = 1$ og koderaten ½.

Figur 16.39 viser hvordan koding forbedrer feilraten til et system. For en feilrate på 10^{-6} kan en med en koderate på $1/2$ klare seg med en E_b/N_o som er 2,77 dB lavere enn for et ukodet system. Først ved meget lav $E_b/N_o \approx 5{,}5\,\text{dB}$ svikter kodingen oss.

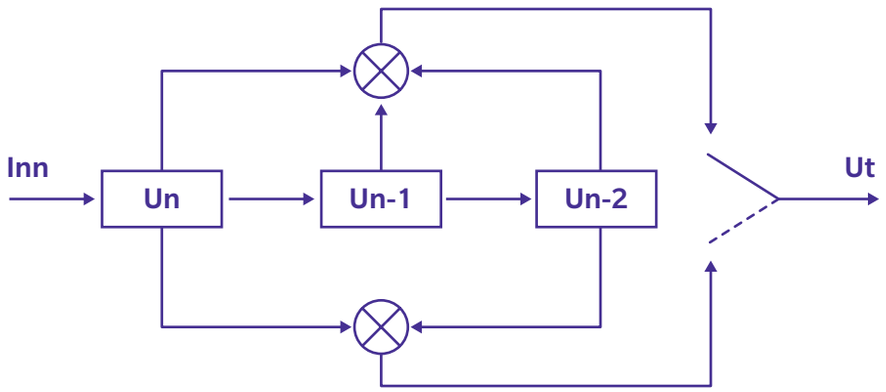


Figur 16.39 Koding forbedrer systemkvaliteten

Hvordan kodeord genereres og tilhørende koder lages, er en hel liten vitenskap i seg selv. Vi skal ikke gå dypt inn i detaljene, men den som er interessert i mer kunnskap på dette feltet, kan søke på Golay koder, algebraiske koder, BCH koder, Reed-Solomon, konvolusjonskoder, Viterbi og turbokoder. Sistnevnte er i skrivende stund en meget populær variant.

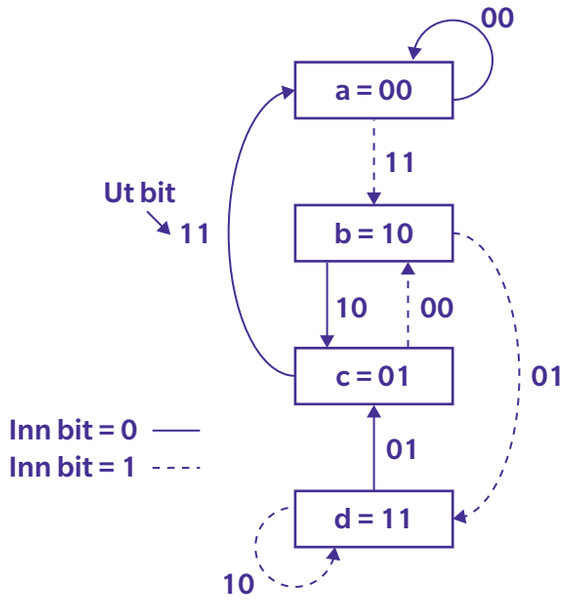
Forresten, la oss ta en liten titt på konvolusjonskoder, da de er et eksempel på bruk av tilstandsmaskiner og den fascinerende Viterbi måten å dekode på.

En konvolusjonskode (n,k,K) behandler k bit av gangen og produserer n bit for hver k innkommende bit. En konvolusjonskode har minnelengde som er gitt av K som kalles «constraint factor». n er bestemt av de siste $K \cdot k$ innkommende bit. Figur 16.40 viser skjematisk en $(1,2,3)$ konvolusjonkoder. I figur 16.40 er det vist en koder med lagre U_n, U_{n-1} og U_{n-2}.



Figur 16.40 Konvolusjonskoder

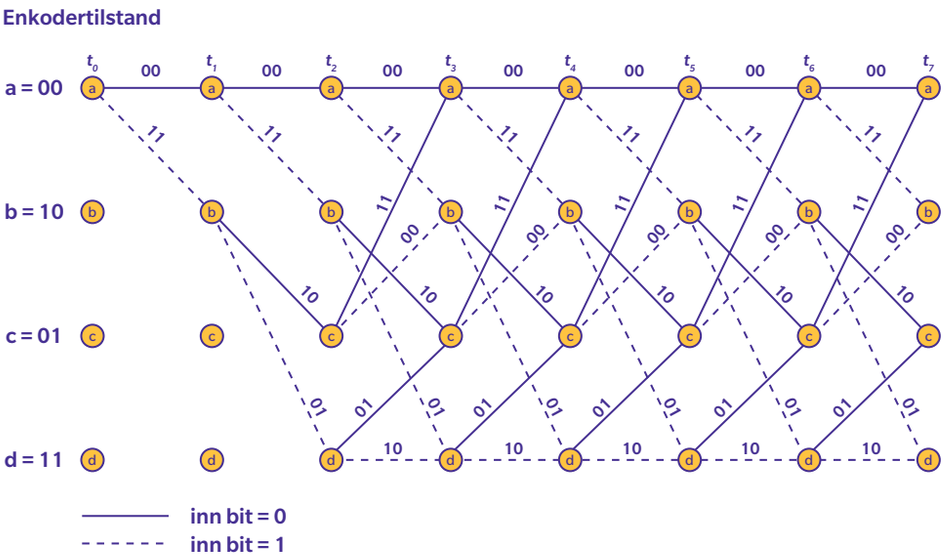
For koderen i figur 16.40 kan en lage en tilstandsmaskin med fire tilstander a, b, c og d som vist i figur 16.41.



I figur 16.41 ser en hvordan en hopper fra tilstand til tilstand i en evig runddans basert på verdien av inn bit og hvilken tilstand en er i. De tilhørende ut bit er vist ved tilstandsovergangene.

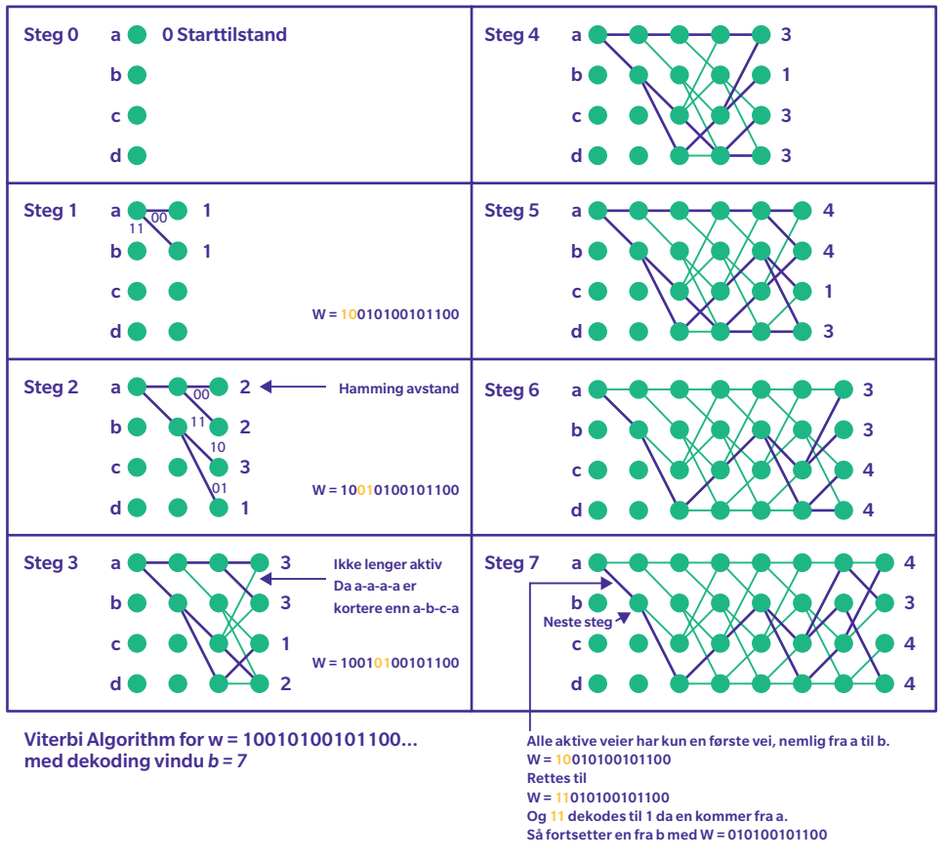
På mottakersiden utnytter vi det faktum at vi har gitt systemet minne. Det er en sammenheng mellom det en ønsker å dekode, og det som kommer etterpå. Vi lager oss en såkalt trellis som er tilstandsmaskinen ved påfølgende tidspunkt, og så ser vi hvordan en mottatt sekvens av bit avtegner seg i trellisen, og måler feil med for eksempel Hamming distanse. Figur 16.42 viser trellisen for vår tilstandsmaskin.

Figur 16.41 Tilstandsmaskin for vår konvolusjonskoder



Figur 16.42 Trellis med tilhørende overganger

Anta at vi mottar bitsekvensen $w = 10010100101100$. Viterbi dekoding går ut på å finne den korteste vei gjennom trellisen basert på mottatt bitsekvens. I figur 16.43 viser Viterbi dekoding med tilhørende forklaringer. I dette tilfellet er det valgt et såkalt vindu med lengde 7. Det betyr at en stopper etter 7 steg i dekoding og så fortsetter med å dekode de to neste bit.



Figur 16.43 Viterbi dekoding

Da har en lang dags ferd brakt oss fra A(lice) til B(ob) uten at Eave(sdropper) får så mye med seg av det som blir kommunisert. Heldigvis er det full-dupleks, slik at vi kan leve opp til utsagnet om at alle reiser er en omvei hjem.

APPENDIKS

Store tanker gir små kretser

«En sten kan ikke flyve. Mor Nille kan heller
ikke flyve. Ergo er Mor Lille en sten.»

Erasmus Montanus, Ludvig Holberg (1684–1754)

LÆRINGSUTBYTTE: Fra boolsk algebra til logiske kretser, logikk med brytere, logikk basert på releer, transistorlogikk

Mye av barndommen ble tilbrakt over det tynne, blå ozonlaget. Mens min far sto bøyd over maskinen i heisrommet, satt jeg på råbetongen og nøt lukten av O_3 og klapringen av releer som fylte veggen bak meg. Releene var styringslogikken til heisen. Tar du en tur inn i et heisrom i dag, vil du se, dersom du finner den, at det meste av styringen er blitt transistorisert med unntak av noen få releer for store strømmer til maskinen. Miniatyrising er tidens melodi, ikke klikk-klakk.

«Alle mennesker er dødelige. Sokrates er et menneske. Altså er Sokrates dødelig.» Matematikere prøvde i mange hundre år å beskrive logikk ved hjelp av matematikk. Den naive, eller geniale, tanken var at dersom en kunne beskrive hvordan man tenkte, så kunne man også vite hvordan hjernen fungerte. Vår venn Gottfried Wilhelm von Leibniz gjorde i sine unge år et hederlig forsøk på å beskrive logikk matematisk, men ga det opp og gjorde heller en karriere ved å finne opp kalkulus. Det var først med George Boole og hans tynne bok *The Mathematical Analysis of Logic, Being an Essay Towards a Calculus of Deductive Reasoning* fra 1847 at gjennombruddet for matematisk logikk kom. Boole var ikke på den tiden alene om å sysle med matematisk logikk, og han fikk blant annet impulser fra Augustus De Morgan som er far til De Morgans lover.

Selv om elektrisitet var tilgjengelig på 1800-tallet, var det ingen som hadde fantasi eller så muligheten til å realisere boolsk logikk ved hjelp av elektriske kretser. Det ble på denne tiden laget mange mekaniske regnemaskiner. Selv ikke mesteren innen det faget, Charles Babbage som hadde korrespondert med Boole, så muligheten for å bruke releer istedenfor gir og spaker. Det var først på 1930-tallet at en oppdaget denne muligheten. Det viktigste bidraget kom fra den unge studenten Claude Elwood Shannon i hans masteroppgave «A Symbolic Analysis of Relay and Switching Circuits» fra 1938. I denne oppgaven viste Shannon hvordan boolsk algebra kunne brukes til å analysere, designe og forenkle svitsjenettverk. Masteroppgaven, som det er en fest å lese, ligger lett tilgjengelig på Internett og viser på slutten blant annet et eksempel på en binær elektrisk regnemaskin (adder). Shannon gjorde senere en strålende karriere i teleselskapet AT&Ts laboratorium Bell Labs. I 1948 publiserte Shannon «The Mathematical Theory of Information», og dermed var informasjonsteorien født som fagfelt og Shannon ble ikonisk.

Shannon brukte releer til å lage logiske kretser. La oss starte litt enklere med bare å bruke manuelle brytere. I kapittelet «En del elementære logiske emner» så vi på en rekke sammensatte logiske utsagn. Blant annet:

5 er et primtall og et partall, eller så er ikke 5 et primtall eller et partall.

I det utsagnet er det to grunnutsagn:

$$P: 5 \textit{ er et primtall}$$

$$Q: 5 \textit{ er et partall}$$

Det sammensatte utsagnet uttrykt ved P og Q blir:

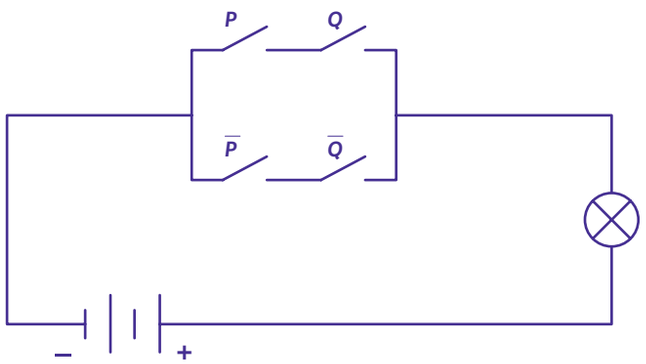
$$(P \wedge Q) \vee \overline{(P \vee Q)}$$

Hvordan kan dette gjøres om til en logisk krets? Våre forfedre brukte 90 år på å finne det ut, men vi bør komme i mål noe raskere. Her er det naturlig å la utsagnene P og Q være brytere. I kretsen kan et usant utsagn presenteres med en åpen bryter og et sant utsagn med en lukket bryter. $P \wedge Q$ finner en ved å ta to brytere i serie, mens $P \vee Q$ får en ved å la to brytere være i parallell. Brytere i serie blir altså en AND-funksjon, og brytere i parallell blir en OR-funksjon. Hva så med $\overline{(P \vee Q)}$?

Den var straks litt verre. En kommer ingen vei ved først å ta bryterne P og Q i parallell og så prøve å ta negasjonen. Heldigvis kommer De Morgan oss til unnsetning. $\overline{(P \vee Q)} = \overline{P} \wedge \overline{Q}$. Ved også å bruke bryterne $\overline{P}$ og $\overline{Q}$ kan utsagnet

$$(P \wedge Q) \vee \overline{(P \vee Q)} = (P \wedge Q) \vee (\overline{P} \wedge \overline{Q})$$

realiseres i kretsen som er gitt i figur A.1.



Figur A.1 En logisk krets for utsagnet $(P \wedge Q) \vee (\overline{P} \wedge \overline{Q})$

Da er det bare å lukke de bryterne som har utsagn som er sanne. Dersom lyspæren lyser, er hele utsagnet sant. En av fordelene med logiske kretser er at en rimelig raskt

kan kontrollere mer eller mindre kompliserte logiske utsagn. La oss sjekke våre utsagn og se om det sammensatte utsagnet er sant og får lyspæren til å lyse opp vår tilværelse:

$$P: 5 \textit{ er et primtall}$$

$$Q: 5 \textit{ er et partall}$$

$$\overline{P}: 5 \textit{ er et ikke primtall}$$

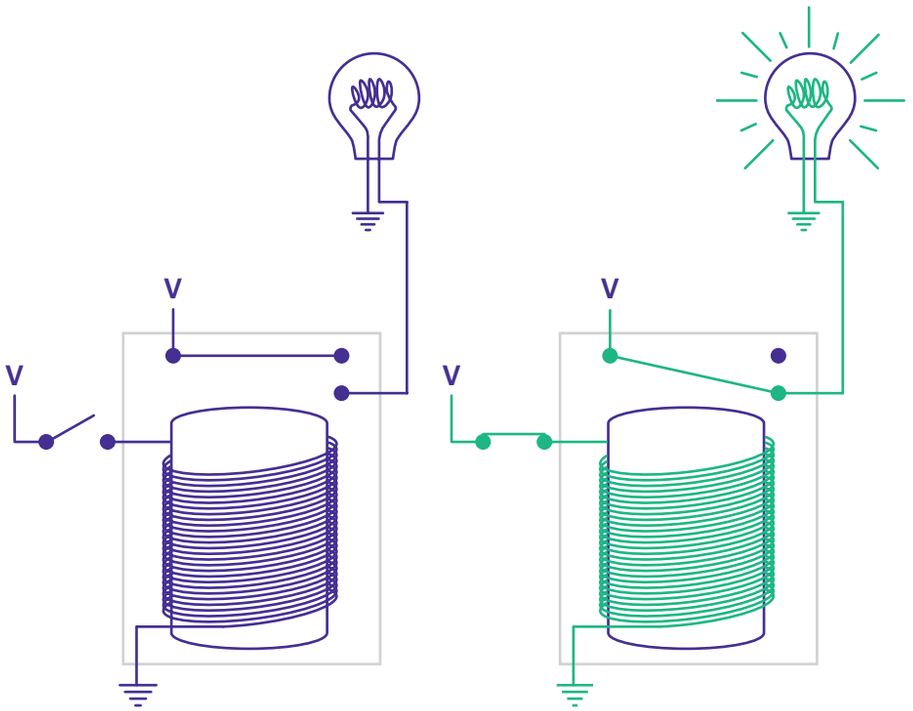
$$\overline{Q}: 5 \textit{ er et ikke partall}$$

Her ser en at Q og $\overline{P}$ er usanne, og de bryterne vil være åpne. Det vil ikke gå strøm i noen av parallelgreinene i kretsen, og lyspæren vil ikke lyse. Det sammensatte utsagnet er altså usant. Ser du noen utsagn som kunne fått lyspæren til å lyse? Stopp og tenk litt før du leser videre.

Dersom en hadde valgt tallet 2 istedenfor 5, ville P og Q vært sanne. Strømmen ville gått igjennom den øverste delen av parallellkretsen, og lyspæren ville lyse. Med 2 ville $\overline{P}$ og $\overline{Q}$ vært usanne, og ingen strøm ville gått gjennom nedre del av parallellkretsen. Hvordan skulle en fått strøm igjennom nedre del av parallellkretsen? Da måtte man valgt et oddetall som ikke er et primtall, og dem er det uendelig mange av.

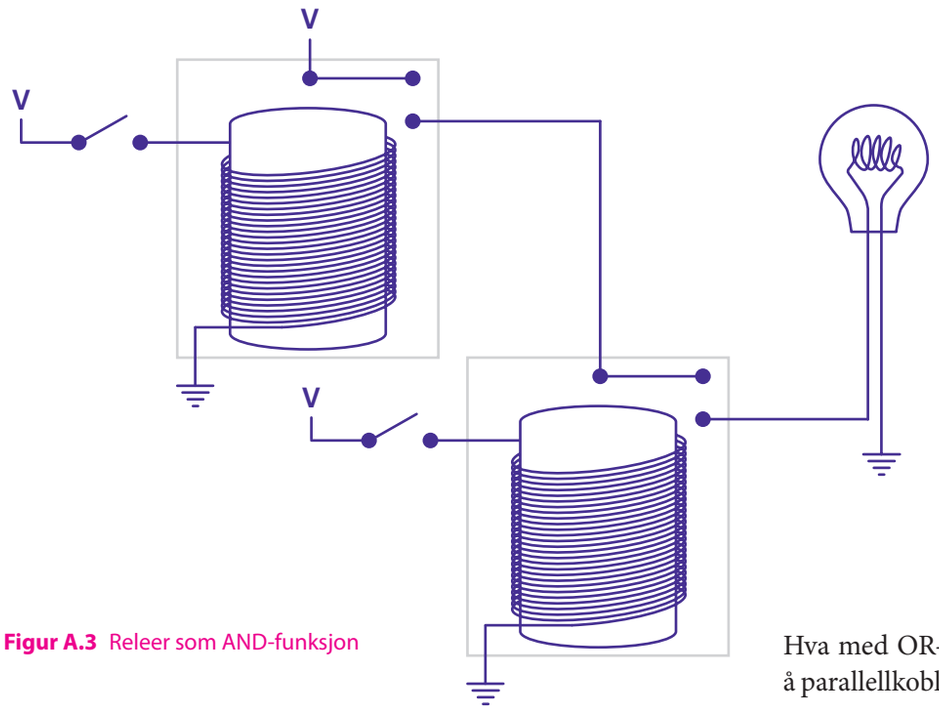
Hvorfor brukte unge Shannon releer istedenfor manuelle brytere? Vel, releene har den fordel at de kan styres elektrisk uten menneskelig medvirkning. Et relé er en bryter styrt av en elektromagnet. Releer var allerede en hundre år gammel oppfinnelse da Shannon begynte å interessere seg for dem. Den amerikanske forskeren Joseph Henry oppfant releet i 1835, og hans hensikt var å forbedre den elektriske telegrafen. Shannon derimot hadde en annen agenda. Han ønsket å lage logiske kretser.

I figur A.2 ser vi en skjematisk fremstilling av et åpent og et lukket relé. Når bryteren er åpen, går det ikke strøm gjennom spolen, og lyspæren vil ikke lyse. Dersom bryteren lukkes, vil det gå strøm gjennom spolen. Det resulterer i et magnetfelt rundt spolen som trekker til seg metallkontakten over spolen, og dermed lyser lyspæren.



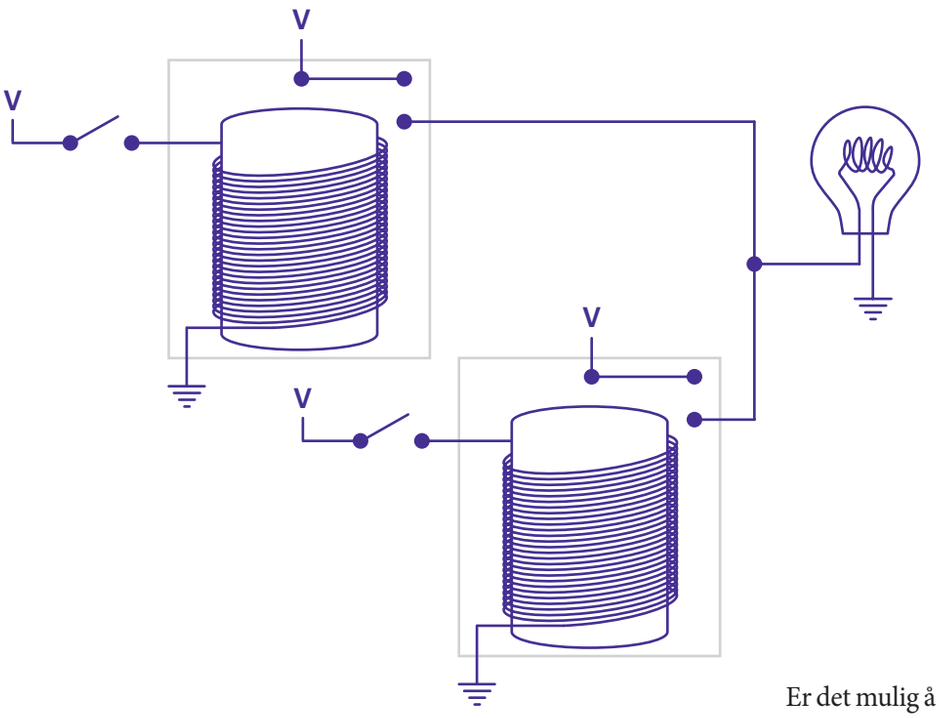
Figur A.2 Relé som bryter

Med releet som grunnstamme er det bare å starte med å lage logiske kretser. Ved å sette to releer i serie får man en AND-funksjon som vist i figur A.3. For å få lyspæren til å lyse må begge bryterne være lukket. En spenning på V volt betyr binært 1 og ingen spenning binært 0.



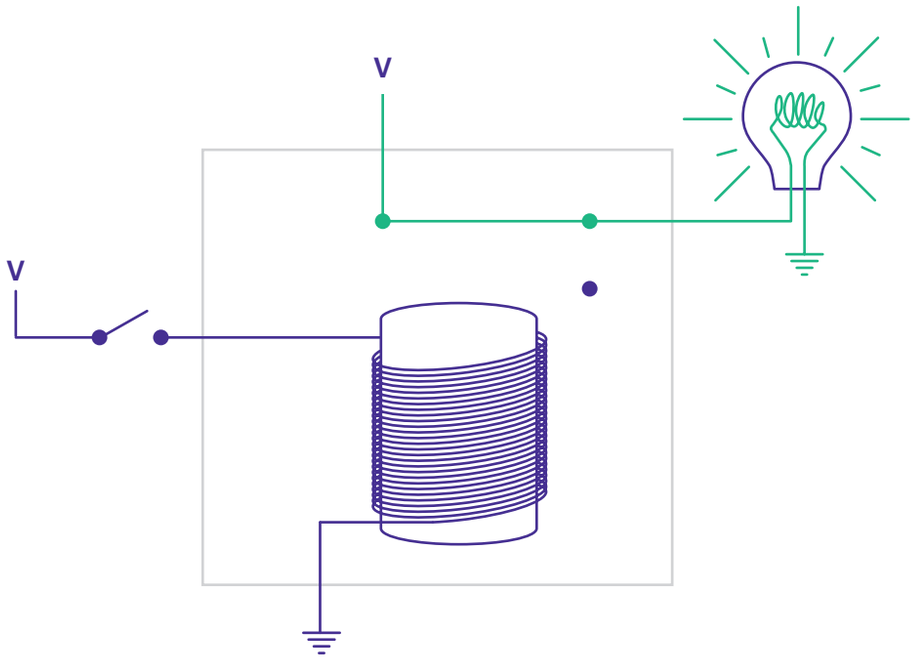
Figur A.3 Releer som AND-funksjon

Hva med OR-funksjonen? Da er det bare å parallellkoble releene som vist i figur A.4.



Figur A.4 Releer som OR-funksjon

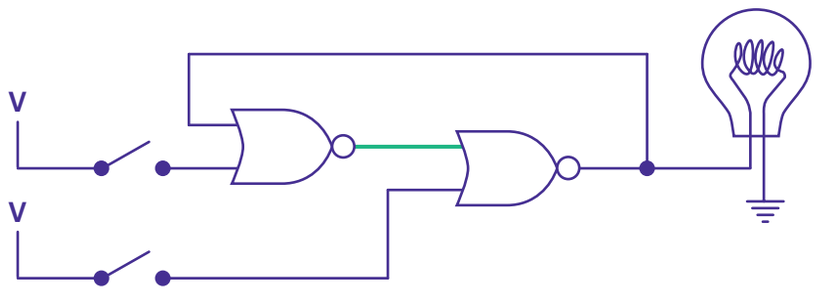
Er det mulig å lage en NOT-funksjon (inverter)? Ja, det er bare å koble lyspæren til den andre utgangen som angitt i figur A.5.



Figur A.5 Relé som NOT-funksjon

Dersom vi ønsker å lage NAND- og NOR-funksjoner, er det bare å settes lyspæren på den andre utkontakten i de opprinnelige AND- og OR-funksjoner.

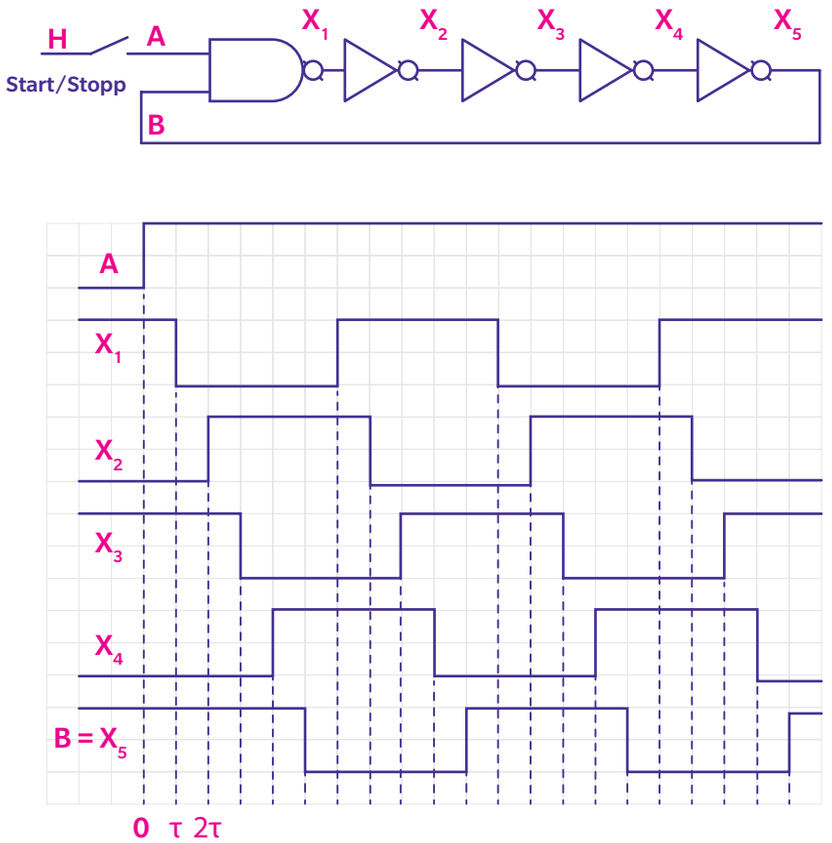
Da har vi med hjelp av releer skaffet oss alt det grunnleggende logiske verktøyet vi trenger. Dersom en skulle ønske å lage en vippe, kan den realiseres som angitt i figur A.6.



Figur A.6 Vippe laget av releer

Er det mulig å lage et klokkesignal med releer? Absolutt, da må vi på samme måte som med vippene sørge for tilbakekobling på en slik måte at spenningen skifter mellom høy og lav. Det kan realiseres med et odde antall invertere, det må jo variere

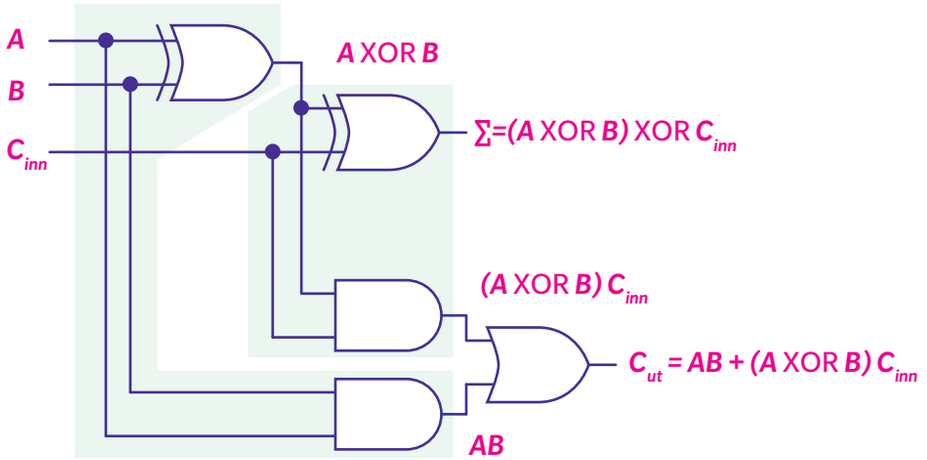
mellom 0 og 1, i serie og så tilbakekobling. Antall invertere vil bestemme klokkeperioden, da de alle har en gitt forsinkelse. Jo flere invertere, jo lengre blir klokkeperioden. I figur A.7 er det vist et eksempel på en klokke med fem invertere og tilhørende tidsdiagram.



Figur A.7 Klokke laget av releer med tilhørende tidsdiagram

Det må innrømmes at dette er en noe naiv måte å lage klokke på som skaper forsinkelse for hver port signalet går igjennom. En mer fornuftig metode er å bruke en krystalloscillator til formålet.

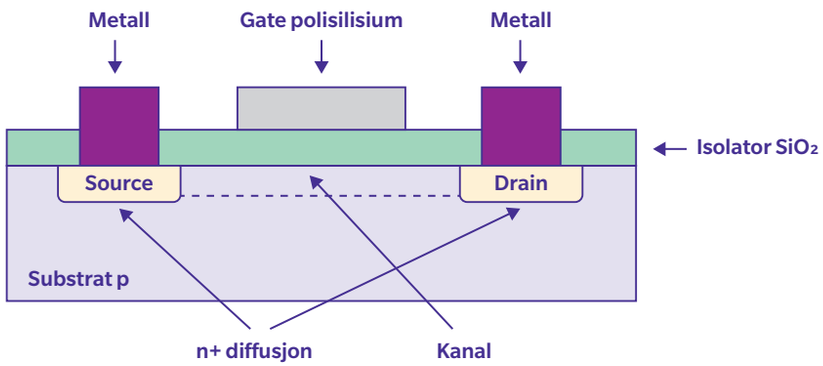
Noe av det siste Claude Shannon viste i sin masteroppgave, var at det var mulig å lage en elektrisk regnemaskin (adder) basert på relélogikk. Vi har tidligere sett på addere av typen gitt i figur A.8. Prinsipielt var Shannons adder lik, men utførelsen var en smule annerledes.



Figur A.8 Adder basert på releer

En fordel med releer er at de er meget robuste, men de er trege brytere (~1 millisekund). De første elektriske datamaskiner ble derfor bygd med radiorør som brytere (svitsjetid 1 mikrosekund) istedenfor releer. Etter hvert som datamaskinene ble store, fikk en et annet problem, nemlig radiorørenes relativt korte levetid. Mang en beregning ble avbrutt av at man måtte skifte radiorør underveis. I 1947 kom løsningen på problemet da William Shockleys forskergruppe ved Bell labs oppfant transistoren. Den kan brukes som en meget rask og robust bryter, og siden den er laget av halvledermateriale, kan den lages så liten som teknologien til enhver tid tillater. Det betyr at en kan få komplekse logiske kretser på meget små arealer.

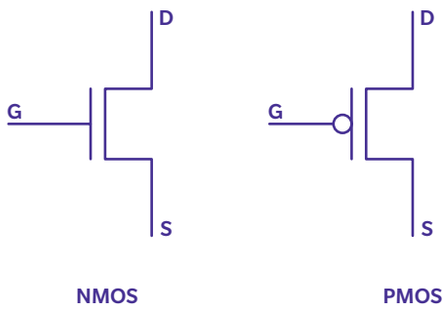
Det finnes to typer transistorer. Den bipolare transistor brukes til å lage såkalt transistor-transistor logikk (TTL) sammen med motstander. TTL blir stadig mindre brukt i logiske kretser fordi den andre typen transistor, felt effekt transistoren, utvikler vesentlig mindre varme når den brukes som en bryter. Det finnes to avarter av denne felt effekt transistoren til bruk i logiske kretser, nemlig PMOS og NMOS. MOS står for metalloksyd felt effekt transistor. Navnet MOS er ikke lenger helt beskrivende for hvilke materialer transistoren er bygd opp av, men henger igjen fra gammelt av. P og N står for hvilke dopingegenskaper substratet har. Dersom en del av disse begrepene er ukjente, kan det være nyttig å lese kapittelet «Elektronikkens DNA» i boken *Grunnleggende elektroteknikk – kort og godt fortalt* [13].



NMOS transistorstruktur.

Figur A.9 Skjematisk fremstilling av en NMOS transistor

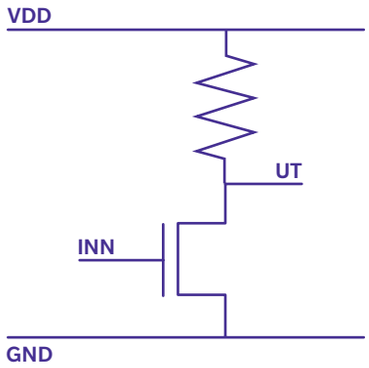
Hvordan kan felt effekt transistoren brukes som en bryter (relé)? Figur A.9 viser en NMOS hvor substratet er p-dopet. Gatespenningen brukes som en bryter mellom source og drain. Når gatespenningen er den samme som sourcespenningen, er source og drain isolert fra hverandre, og bryteren er derfor åpen. Når gatespenningen økes, vil det dannes en kanal mellom source og drain, og dermed er bryteren lukket. I en PMOS hvor substratet er n-dopet, er oppførselen omvendt. Den leder når gatespenningen er lik source spenningen, og bryter når gatespenningen er høyere.



Figur A.10 Symboler for henholdsvis NMOS og PMOS

Legg merke til den lille ringen på gatesiden av PMOS transistoren i figur A.10. Den indikerer den inverterende egenskapen til PMOS-bryteren.

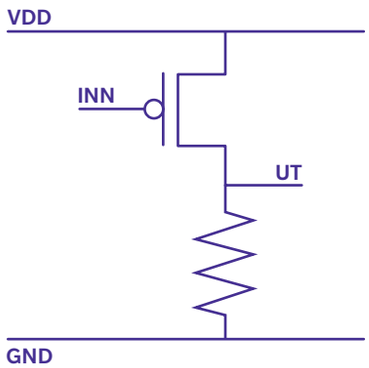
Hvordan skal vi på en effektiv måte lage logiske porter av disse byggeklossene? La oss starte forsiktig med en IKKE-funksjon (inverter). Dersom vi bruker NMOS, kunne vi ha kommet i land med følgende krets.



Figur A.11 NMOS Ikke-funksjon

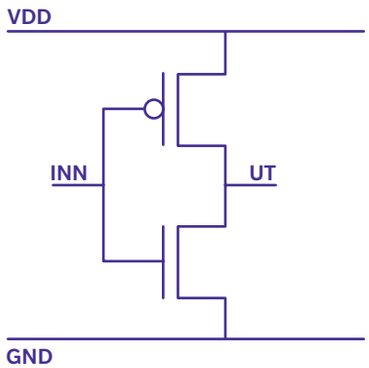
Her angir VDD forsyningsspenning, og over transistoren har vi en liten motstand. Når gatespenningen (inn) er 0 volt, leder ikke transistoren, og sourcespenningen (ut) er lik VDD. Lav inn gir høy ut. Dersom gatespenningen er høy, vil transistoren lede og sourcespenningen (ut) bli null. Igjen har en invertering.

Hvordan skal en få til det samme med PMOS? Da PMOS oppfører seg motsatt av NMOS, er det bare å la motstand og transistor bytte plass.



Figur A.12 PMOS Ikke-funksjon

Er det mulig å forbedre denne logiske kretsen? Motstander brenner av unødvendig mye energi. Hva med å bruke transistorene som motstander? Når bryteren er åpen, er motstanden uendelig, og når den er lukket, er den null. Akkurat hva vi trenger. Ved å legge figur A.11 og figur A.12 oppå hverandre og erstatte motstandene med transistorer får vi følgende logisk krets for IKKE-funksjonen.



Figur A.13 CMOS Ikke-funksjon

Denne logiske kretsen har den ønskede funksjon. Når innspenningen er lav, leder ikke den nederste transistoren, men den øverste, og dermed blir utspenningen høy. Dersom innspenningen er høy, leder den nederste transistoren, men ikke den øverste, og utspenningen blir lav. Vi har klart å lage en logisk krets som kun forbruker energi i det korte svitsjeøyeblikket. Denne bruken av NMOS og PMOS hvor de utfyller hverandre, kalles CMOS, som står for komplementær MOS.

Da er det bare å prøve å lage de andre grunnleggende portene med CMOS teknologi. Det har andre prøvd før oss og funnet ut at det er lettest å realisere NAND og NOR porter. Som vi husker fra kapittelet «En del elementære logiske emner», kan en utlede de tre basisfunksjonene IKKE, OG og ELLER fra enten NAND eller NOR ved å bruke den boolske algebra. En kan altså bygge opp en logisk krets ved bare å bruke NAND eller NOR funksjoner.

$$\overline{A} = \overline{AA} = \overline{A + A}$$

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

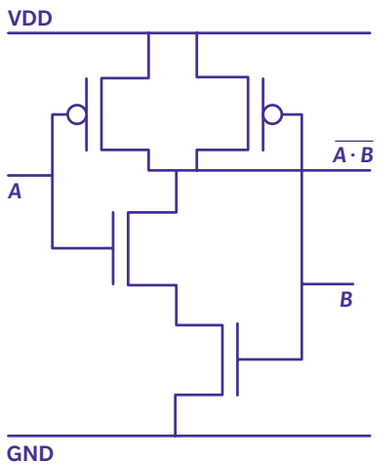
$$A + B = \overline{\overline{A}\,\overline{B}} = \overline{\overline{A + B}}$$

Dermed har vi det vi trenger. La oss starte med å prøve å lage NAND. Hvilke egenskaper skal vi dyrke frem? Vel, de er gitt av sannhetstabellen for NAND.

Tabell A.1 Sannhetstabell for NAND

A	B	$\overline{AB}$
0	0	1
0	1	1
1	0	1
1	1	0

Av den ser vi at når enten A eller B er 0, så er $\overline{AB} = 1$. Ved å sette A og B sin NMOS i serie fra jord og opp vil vi sørge for vi er isolert fra jord når enten A eller B er lik 0. I tillegg må vi sørge for at VDD kommer frem til der hvor vi henter ut $\overline{AB}$. Det kan vi gjøre ved å sette A og B sin PMOS i parallell. Denne logiske kretsen vil også gi det riktige resultatet når A og B begge er 1. Da leder NMOS-ene som er seriekoblet, og $\overline{AB} = 0$. Det var kretsen beskrevet med ord. Figur A.14 gir den skjematiske fremstilling.



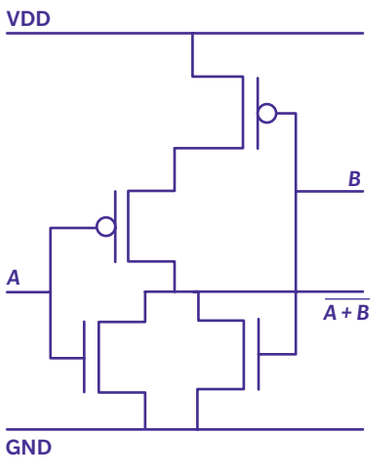
Figur A.14 CMOS NAND

Hva så med NOR-funksjonen? La oss starte med sannhetstabellen:

Tabell A.2 Sannhetstabell for NOR

A	B	$\overline{A+B}$
0	0	1
0	1	0
1	0	0
1	1	0

Her ser vi at når enten A eller B er lik 1, så blir $\overline{A+B}=0$, og når begge er 0, så blir $\overline{A+B}=1$. For NAND var det stikk motsatt. Dermed kan NOR realiseres med to NMOS i parallell mot jord og to PMOS i serie mot VDD. Dersom du ikke er overbevist, så prøv alle kombinasjoner av A og B og se hvor du havner.



Figur A.15 CMOS NOR

Det må medgis at dette var en relativt enkel gjennomgang av virkemåten til grunnleggende logiske kretser. Dersom du ønsker å dykke dypere ned i teknologien, finner du forslag til videre lesing i «Litteraturliste og kilder» gitt i appendiks.

Konvensjoner m.m.

Konvensjoner og enheter

Tabell B.1 Konvensjoner og enheter

Utsagn	Bitverdi	Beskrivelse	Spenningsnivå
Sant	1	HØY	5, 3,3 eller 1,8 volt
Usant	0	LAV	0 volt

Notasjon

Tabell B.2 Logiske tegn

Funksjon	Tegn – Matematisk logikk	Tegn – Digitalteknikk
IKKE	$^-$ eller $\neg$	$^-$
OG	$\wedge$	$\cdot$ eller ingen prikk
ELLER	$\vee$	+

Definisjoner

Tabell B.3 Definisjoner

Navn	Definisjon	Kommentar
Bit	Basis enhet for informasjon	Bit har to mulige verdier 1 og 0
Byte	Åtte sammenhengende bit	
Ord	N sammenhengende byte	
Informasjonsmengde	$I_k \equiv log_2\left(\frac{1}{p_k}\right)$	Mengde informasjon i en melding med sannsynlighet p_k målt i bit
Entropi	$H \equiv \frac{I_{total}}{L} = \sum_{k=1}^{M} p_k log_2\left(\frac{1}{p_k}\right)$	Den gjennomsnittlige informasjon per meldingsintervall målt i bit
Informasjonsrate	$R \equiv rH$	R er da gjennomsnittlig antall bit informasjon per sekund

Forkortelser

Tabell B.4 Forkortelser

Forkortelse	Forklaring
ACK	ACKnowledge
ADC	Analog to Digital Converter
AEQB	A EQual B
AES	Advanced Encryption Standard
AGTB	A Greater Than B
ALGOL	ALGOrithmic Language
ALTB	A Less Than B
ALU	Arithmetic Logic Unit
AND	AND (OG funksjon/port)
BCD	Binary Coded Decimal
BIOS	Basic Input/Output System
CLB	Configurable Logic Blocks
CMOS	Complementary Metal-Oxide-Semiconductor
CPU	Central Processing Unit
CRC	Cyclic Redunancy Check
DAC	Digital to Analog Converter
DC	Data Counter
DMA	Direct Memory Access
DRAM	Dynamic RAM
DSP	Digital Signal Processing
EEPROM	Electrically Erasable Programmable ROM
EPROM	Electrically Programmable ROM
FDM	Frequency Division Multiplexing
FET	Field Effect Transistor
FORTRAN	FORmula TRANslation
FPGA	Field Programmable Gate Arrays
FSK	Frequency Shift Keying

FSM	Finite State Machine
GPU	Graphical Processing Unit
IACK	Interrupt ACKnowledge
I/O	Input/Output
IREQ	Interrupt REQuest
JTAG	Joint Test Action Group
LUT	Look Up Table
µC	Micro Computer
NAND	Negative AND
NMOS	N-type Metal-Oxide-Semiconductor
NOR	Negative OR
NOT	NOT
OR	OR
PC	Program Counter
PCM	Pulse Code Modulation
PMOS	P-type Metal-Oxide-Semiconductor
PSK	Phase Shift Keying
QAM	Qadrature Amplitude Modulation
RAM	Random Access Memory
ROM	Read Only Memory
RMS	Root Mean Square
RSA	Rivest Shamir Alderman
RTL	Register Transfer Level
SAR	Successive-Approximation Register
SoC	System on Chip
SRAM	Static RAM
SSOP	Standard Sum of Products
TDM	Time Division Multiplexing
UART	Universal Asynchronous Receiver Transmitter
VHDL	VHSIC Hardware Description Language. VHSIC Very High Speed Integrated Circuit
XOR	eXclusive OR

Tabeller

Tabell B.5 ASCII kode

Binært	Desimal	Hex	Grafisk	Binært	Desimal	Hex	Grafisk	Binært	Desimal	Hex	Grafisk
010 0000	32	20	Mellomrom	100 0000	64	40	@	110 0000	96	60	`
010 0001	33	21	!	100 0001	65	41	A	110 0001	97	61	a
010 0010	34	22	“	100 0010	66	42	B	110 0010	98	62	b
010 0011	35	23	#	100 0011	67	43	C	110 0011	99	63	c
010 0100	36	24	$	100 0100	68	44	D	110 0100	100	64	d
010 0101	37	25	%	100 0101	69	45	E	110 0101	101	65	e
010 0110	38	26	&	100 0110	70	46	F	110 0110	102	66	f
010 0111	39	27	‘	100 0111	71	47	G	110 0111	103	67	g
010 1000	40	28	(	100 1000	72	48	H	110 1000	104	68	h
010 1001	41	29	)	100 1001	73	49	I	110 1001	105	69	i
010 1010	42	2A	*	100 1010	74	4A	J	110 1010	106	6A	j
010 1011	43	2B	+	100 1011	75	4B	K	110 1011	107	6B	k
010 1100	44	2C	,	100 1100	76	4C	L	110 1100	108	6C	l
010 1101	45	2D	-	100 1101	77	4D	M	110 1101	109	6D	m
010 1110	46	2E	.	100 1110	78	4E	N	110 1110	110	6E	n
010 1111	47	2F	/	100 1111	79	4F	O	110 1111	111	6F	o
011 0000	48	30	0	101 0000	80	50	P	111 0000	112	70	p
011 0001	49	31	1	101 0001	81	51	Q	111 0001	113	71	q
011 0010	50	32	2	101 0010	82	52	R	111 0010	114	72	r
011 0011	51	33	3	101 0011	83	53	S	111 0011	115	73	s
011 0100	52	34	4	101 0100	84	54	T	111 0100	116	74	t
011 0101	53	35	5	101 0101	85	55	U	111 0101	117	75	u
011 0110	54	36	6	101 0110	86	56	V	111 0110	118	76	v
011 0111	55	37	7	101 0111	87	57	W	111 0111	119	77	w
011 1000	56	38	8	101 1000	88	58	X	111 1000	120	78	x
011 1001	57	39	9	101 1001	89	59	Y	111 1001	121	79	y
011 1010	58	3A	:	101 1010	90	5A	Z	111 1010	122	7A	z
011 1011	59	3B	;	101 1011	91	5B	[	111 1011	123	7B	{
011 1100	60	3C	<	101 1100	92	5C	\	111 1100	124	7C	|
011 1101	61	3D	=	101 1101	93	5D	]	111 1101	125	7D	}
011 1110	62	3E	>	101 1110	94	5E	^	111 1110	126	7E	~
011 1111	63	3F	?	101 1111	95	5F	_				

En liten digitalteknisk formelsamling

Tabell B.6 Formler

Navn	Formel	Kommentar
Nyquistfrekvensen	$f_s > 2f_T = 2f_{Nyquist}$	Samplingsfrekvensen må være mer enn dobbel av signalets høyeste frekvensinnhold
Analog til digital kvantiseringsstøy	$SNR_{dB} = (6{,}02N + 1{,}76)$ dB	N er antall bit i analog til digitalkonverter
Kanalkapasitet	$C = Blog_2\left(1 + \frac{S}{N}\right)$ bit/s	Hvor B er kanalbåndbredden, S signalstyrken og N den totale støy innenfor kanalbåndbredden

Tallsystemer

Tabell B.7 Sammenheng mellom ulike tallsystemer

Heksadesimalt	Desimalt	Oktalt	Binært
0	0	0	0000
1	1	1	0001
2	2	2	0010
3	3	3	0011
4	4	4	0100
5	5	5	0101
6	6	6	0110
7	7	7	0111
8	8	10	1000
9	9	11	1001
A	10	12	1010
B	11	13	1011
C	12	14	1100

Heksadesimalt	Desimalt	Oktalt	Binært
D	13	15	1101
E	14	16	1110
F	15	17	1111

Tabell B.8 Binære tall med fortegn

Binær	Desimal	Heksadesimal
10000000	−128	80
10000001	−127	81
10000010	−126	82
10000011	−125	83
…	…	…
…	…	…
…	…	…
11111110	−2	*FE*
11111111	−1	*FF*
00000000	0	0
00000001	1	1
00000010	2	2
00000011	3	3
…	…	…
…	…	…
…	…	…
01111101	125	7*D*
01111110	126	7*E*
01111111	127	7*F*

Tabell B.9 Desimalt/BCD

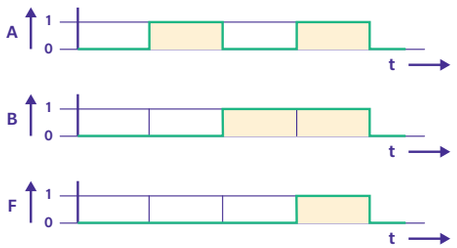
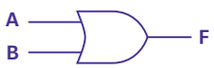
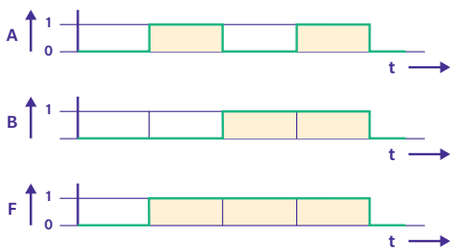
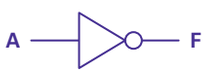
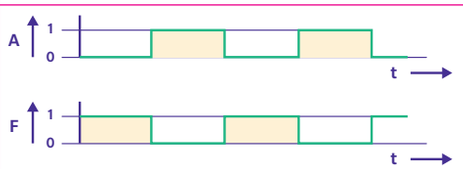
Konvertering mellom desimalt og BCD										
Desimalt	0	1	2	3	4	5	6	7	8	9
BCD	0000	0001	0010	0011	0100	0101	0110	0111	1000	1001

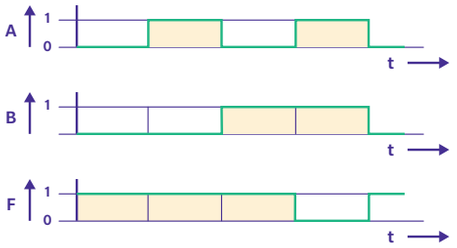
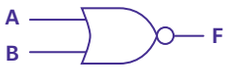
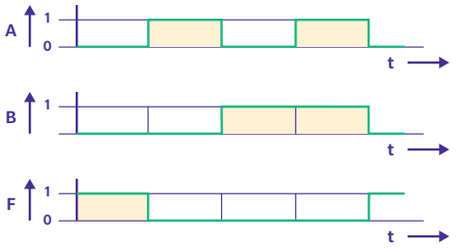
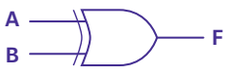
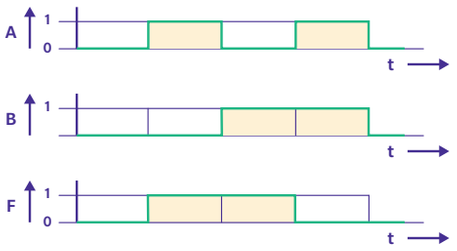
Tabell B.10 Fire bit Gray-kode

Desimal	Binær	Gray-kode	Desimal	Binær	Gray-kode
0	0000	0000	8	1000	1100
1	0001	0001	9	1001	1101
2	0010	0011	10	1010	1111
3	0011	0010	11	1011	1110
4	0100	0110	12	1100	1010
5	0101	0111	13	1101	1011
6	0110	0101	14	1110	1001
7	0111	0100	15	1111	1000

Boolsk algebra med tilhørende symboler

Tabell B.11 Boolske grunnfunksjoner

Funksjon	Portsymbol	Sannhetstabell	Tidsdiagram	Logisk funksjon
OG (AND)		A B F 0 0 1 0 1 0 1 0 0 1 1 0		$F = AB$
ELLER (OR)		A B F 0 0 0 0 1 1 1 0 1 1 1 1		$F = A + B$
IKKE (NOT)		A F 0 1 1 0		$F = \overline{A}$

Negativ OG (NAND)		A B F 0 0 1 0 1 1 1 0 1 1 1 0		$F = \overline{AB}$
Negativ ELLER (NOR)		A B F 0 0 1 0 1 0 1 0 0 1 1 0		$F = \overline{A + B}$
Eksklusiv ELLER (XOR)		A B F 0 0 0 0 1 1 1 0 1 1 1 0		$F = \left(A\overline{B}\right) + \left(\overline{A}B\right)$

Tabell B.12 De boolske lovene

Nummer	ELLER	OG	Navn
B1	$A+A=A$	$AA=A$	Idempotent
B2	$A+(B+C)=(A+B)+C$	$A(BC)=(AB)C$	Assosiativ
B3	$A+B=B+A$	$AB=BA$	Kommutativ
B4	$A+(AB)=A$	$A(A+B)=A$	Absorpsjon
B5	$A+(BC)=(A+B)(A+C)$	$A(B+C)=(AB)+(AC)$	Distribusjon
B6	$A+1=1$	$A0=0$	Bundet
B7	$A+0=A$	$A1=A$	Identitet
B8	$A+\overline{A}=1$	$A\overline{A}=0$	Komplement
B9	$\overline{0}=1$	$\overline{1}=0$	0 og 1
B10	$\overline{(A+B)}=\overline{A}\ \overline{B}$	$\overline{(AB)}=\overline{A}+\overline{B}$	De Morgan

Det finnes totalt ti par med boolske lover, og de er gitt i tabell B.12. De lovene som ikke gjelder i ordinær algebra, men i boolsk algebra, er angitt med farge.

Litteraturliste og kilder

Litteraturliste

1. Erstad, G. og I. Bjørnsgård (1972). *Matematikk 1A Algebra*. Oslo: H. Aschehoug & Co.
2. Floyd, T. (2015). *Digital Fundamentals* (11. utg.). Upper Saddle River, New Jersey: Pearson.
3. Gulbrandsen, M.G., J. Kleppe, T.A. Kro og J.E. Vatne (2013). *Matematikk for ingeniørfag*. Oslo: Gyldendal Akademisk.
4. Haustveit, S. (2016). Forelesningsnotater.
5. Lia, H. (2005). *Teleteknikk*. Kompendium.
6. Mealy, B. og F. Tappero (2017). *Free Range VHDL*. http://www.freerangefactory.org
7. Meddins, B. (2000). *Introduction to Digital Signal Processing*. Oxford: Newnes.
8. Osborne, A. (1978). *Innføring i mikrodatamaskiner*. Oslo: Universitetsforlaget.
9. Petzold, C. (1999). *Code*. Redmond, WA: Microsoft Press.
10. Pierce, J.R. (1980). *An Introduction to Information Theory: Symbols, Signals and Noise* (2. reviderte utg.). New York: Dover publications.
11. Tanenbaum, A. (1981). *Computer Networks*. Englewood Cliffs: Prentice-Hall.
12. Taub, H. og D.L. Schilling (1971). *Principles of Communication Systems*. New York: McGraw-Hill.
13. Thorvaldsen, P. (2017). *Grunnleggende elektroteknikk*. Bergen: Fagbokforlaget.
14. Weste, N. og D. Harris (2010). *CMOS VLSI Design: A Circuits and Systems Perspective* (4. utg.). Boston: Addision Wesley.
15. Zwolinski, M. (2004). Digital System design with VHDL (2. utg.). Harlow: Pearson education.

Kilder

Møt familien
Dette kapittelet er inspirert av min kollega høyskolelektor Svein Haustveits forelesningsnotat *Where are the NAND gates?*

Det er mengden det kommer an på
Her hadde jeg gleden av en gammel gymnasbok, Erstad og Bjørnsgård *Matematikk 1A Algebra*. En nyere variant som tar for seg samme emne på en fin måte, er *Matematikk for ingeniørfag* av Martin G. Gulbrandsen, Johannes Kleppe, Tore A. Kro og Jon Eivind Vatne.

Tenk på et tall
I dette kapittelet er to kilder brukt. *Innføring i mikrodatamaskiner* av Adam Osborne har en meget god, kort og konsis innføring i tallsystemer. Thomas Floyd *Digital Fundamentals* har en noe lengre utredning.

En del elementære logiske emner
Også i dette kapittelet er *Innføring i mikrodatamaskiner* av Adam Osborne og *Digital Fundamentals* av Thomas Floyd hovedkilder.

Fra det ene til det andre
Her er Thomas Floyd *Digital Fundamentals* faglig hovedkilde.

Tingenes tilstand
Deler av dette kapittelet er basert på egne notater, mens noe har Thomas Floyd *Digital Fundamentals* som kilde.

Gjennomtrekk
Dette kapittelet er primært basert på høyskolelektor Svein Haustveits forelesningsnotater.

Evig runddans
Her er Thomas Floyd *Digital Fundamentals* faglig hovedkilde.

Å sette ord på det
Dette kapittelet er basert på lesning av Zwolinski Mark *Digital System design with VHDL* og Mealy og Tapperos *Free Range VHDL*. De er begge meget lesverdige dersom du ønsker å dykke dypere ned i denne materien.

Logikk for viderekommende

Her er Thomas Floyd *Digital Fundamentals* faglig hovedkilde.

UARTig?

Dette kapittelet er basert på UART_TX oppgave gitt av høyskolelektor Svein Haustveit. For å utføre oppgaven ble simuleringsverktøyene Multisim fra National Instruments og ModelSim fra Intel brukt.

Kun i minne finner hjertet fred

Hovedkilde til dette kapittelet er *Code* av Charles Petzold. Den er en meget god og underholdende innføring i digitalteknikk. I tillegg er Thomas Floyd *Digital Fundamentals* brukt som kilde.

Følg instruksen!

Dette kapittelet er basert på et eksempel gitt i *Innføring i mikrodatamaskiner* av Adam Osborne.

Fremskritt og tilbakesteg

Thomas Floyd *Digital Fundamentals* og Bob Meddins *Introduction to Digital Signal Processing* er brukt som kilder. Bob Meddins er en meget god innføringsbok i digital signalbehandling som det er vel verdt å lese.

Fra A til B

Her er det brukt en rekke kilder: Thomas Floyd *Digital Fundamentals*, Hermann Lia *Teleteknikk*, Taub and Schilling *Principles of Communication Systems*, Andrew S. Tanenbaum *Computer Networks* og William Stallings *Wireless Communication & Networks*. Dersom du skulle ønske å lese mer om informasjonsteori, så finnes det en fantastisk bok av John R. Pierce *An Introduction to Information Theory: Symbols, Signals and Noise*.

Appendiks

Store tanker gir små kretser

Dette kapittelet er inspirert av Charles Petzolds Code. Ønsker du å studere «hardware» mer nøye, anbefales *CMOS VLSI Design: A Circuits and Systems Perspective (4th Edition)* av Weste og Harris.

Løsningsforslag til kodekryssord

1 O	2 S	3 A	4 K	3 A		4 K	3 A	5 L	2 S	6 I	7 U	8 M		2 O
9 V	6 I	5 L	5 L	2 S	10 P	1 O	11 R	12 E	13 T		14 F	7 U	2 S	13 T
3 A	9 V		15 Ø	4 K	12 E	11 R		4 K	7 U	5 L	15 Ø	11 R	13 T	12 E
5 L	6 I	16 G	17 N	12 E	17 N	18 D	12 E		18 D	7 U	11 R	19 B	3 A	17 N
12 E	5 L	5 L	12 E		16 G	12 E	1 O	8 M	12 E	13 T	12 E	11 R		16 G
	5 L	3 A	11 R	9 V	12 E		2 S	7 U	17 N	13 T		3 A	2 S	12 E
2 S	1 O	2 S	6 I	3 A	5 L	13 T		12 E	13 T	12 E	11 R	17 N	6 I	13 T
4 K	9 V	8 M		11 R	1 O	12 E	2 S		19 B	11 R	12 E	2 S	13 T	
3 A		3 A	9 V	2 S	13 T	12 E	8 M	13 T	12 E		4 K	20 J	3 A	2 S
8 M	7 U	2 S	12 E	12 E	13 T		21 Å	11 R	9 V	21 Å	4 K	12 E	17 N	13 T
2 S	13 T	20 J	12 E	5 L	12 E	11 R		12 E	6 I	8 M	12 E		13 T	7 U
5 L	12 E	1 O	17 N		11 R	6 I	18 D	12 E	2 S	12 E	17 N	13 T	12 E	11 R
1 O		17 N	12 E	18 D	6 I	2 S	12 E	13 T		17 N	12 E	9 V	17 N	12 E

1 B	2 V	3 U	4 E	3 U	■	5 O	6 I	7 L	2 V	8 K	9 R	10 N	■	2 V
6 I	2 V	11 S	9 R	5 O	12 A	3 U	13 M	3 U	6 I	■	7 L	11 S	7 L	8 K
4 E	13 M	■	9 R	14 T	1 B	15 Å	6 I	5 O	15 Å	16 C	3 U	12 A	17 P	6 I
6 I	13 M	2 V	15 Å	13 M	2 V	15 Å	7 L	■	■	17 P	3 U	17 P	4 E	12 A
12 A	14 T	7 L	16 C	■	11 S	4 E	10 N	14 T	9 R	12 A	7 L	4 E	■	3 U
■	4 E	14 T	5 O	12 A	17 P	■	6 I	14 T	6 I	11 S	■	4 E	16 C	16 C
7 L	12 A	10 N	13 M	3 U	14 T	16 C	■	13 M	1 B	9 R	5 O	11 S	12 A	2 V
7 L	17 Ø	2 V	■	9 R	11 S	3 U	14 T	■	15 Å	7 L	8 K	3 U	10 N	■
10 N	■	11 S	15 Å	1 B	14 T	4 E	8 K	9 R	2 V	■	13 M	8 K	11 S	5 O
1 B	17 P	11 S	5 O	14 T	■	■	5 O	10 N	8 K	15 Å	8 K	9 R	14 T	5 O
4 E	8 K	6 I	10 N	9 R	17 P	4 E	8 K	11 S	1 B	1 B	15 Å	■	16 C	14 T
13 M	15 Å	17 P	9 R	■	1 B	9 R	5 O	13 M	2 V	10 N	15 Å	5 O	16 C	16 C
4 E	■	2 V	8 K	16 C	16 C	1 B	17 P	12 A	■	7 L	7 L	6 I	17 P	1 B

Løsningsforslag til QUIZ: Melding utenfra

0	0	1	1	0	0	0
1	0	1	1	0	0	0
1	1	1	1	1	1	1
0	0	1	1	0	0	1
0	0	1	1	0	0	1
0	0	1	1	0	0	1
0	0	1	1	0	0	0
0	1	1	1	1	0	0
0	1	0	0	1	0	0
0	1	0	0	1	0	0
1	1	0	0	1	1	0

Figur Vår venn utenfra

Faktisk har et lignende bilde blitt sendt fra jorden til det ytre rom. Det ble brukt to vesentlig større primtall for å få bedre oppløsning i bildet. En antar at en rimelig oppegående mottaker vil kunne rekonstruere bildet, da primtall er uavhengig av språk og notasjon.

Stikkord

I

K

L

M

N

P

S

T