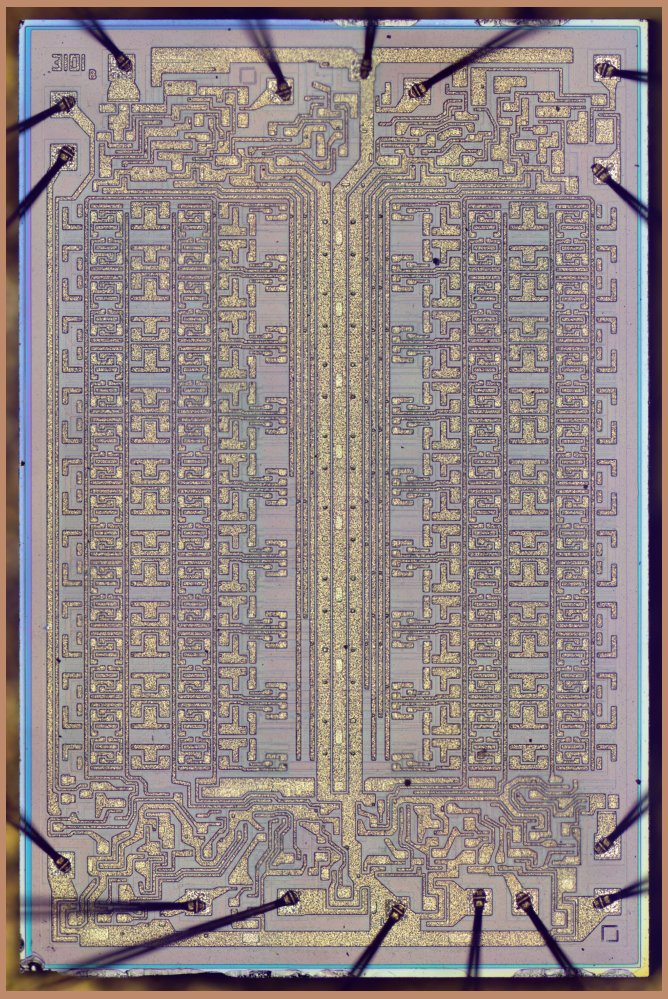
# Digital Elektronikk



Knut Harald Nygaard

3. utgave

Knut Harald Nygaard

post@andiha.no

*Knut Harald Nygaard*

# *Digital Elektronikk*

3. Utgave 2022

ISBN 978-82-690581-4-7



Forsidebilde: Intels første produkt: 3101 RAM chip ved Ken Shirriff, www.righto.com

# Forord

Denne boka er framkommet for å dekke grunnleggende digitalteknikk. Første kapittel er en introduksjon til emnet med litt transistor- og komponent-teori. Neste kapittel omhandler tallsystemer med hovedvekt på det binære tallsystem. De neste kapitler presenterer grunnleggende teori om porter, Boole'sk algebra og kombinatorikk. Det påfølgende kapittel omhandler låsekretser og vipper som er inngangsporten til realisering av skiftregistre og tellere som følger i de neste kapitler.

Neste kapittel omhandler sekvensielle kretser som en utvidelse av hva som er realiserbart med bare skiftregistre og tellere. Påfølgende kapittel er kalt spesialkretser. Dette kapitlet omhandler blant annet multiplikasjonskretser, divisjonskretser og hukommelseskretser. Det er lagt vekt på å gjøre stoffet uavhengig av spesielle integrerte kretser for alle disse kapitlene. Nest siste kapittel gir en kort beskrivelse av mikrokontrollere mens siste kapittel omhandler programmeringsspråket VHDL, som blant annet brukes til programmering av kundespesifiserte integrerte kretser.

Materialet som denne boka utgjør, har vært benyttet i undervisning i faget. Erfaringene fra denne undervisningen har således vært benyttet til å bestemme bokas innhold og struktur. Boka forutsetter ingen spesielle forkunnskaper, og det er med vilje unngått å bruke spesielle matematiske notasjoner som kan vanskeliggjøre stoffet. Det forutsettes at praktiske øvinger og laboratorieoppgaver må komme i tillegg for å få størst mulig utbytte av boka.

Det finnes et utall bøker på engelsk som omhandler digitalteknikk. En norsk bok skal bruke norske ord og uttrykk. Dette er en utfordring siden vi bruker engelskspråklige datablad, tidsskrifter og programvare samtidig som mye av informasjonen som ellers finnes på nettet er engelskspråklig. I denne boka vil du derfor finne både engelsk-norske ord og noen norske «konstruksjoner» som kan diskuteres. Imidlertid bør leseren også kunne alle engelske betegnelser, som tross alt er viktig innenfor dette faget.

Det er lagt vekt på å ikke gjøre stoffet altfor teoretisk. Det finnes en rekke dataverktøy for analyse av digitale kretser, der mange er proprietære. Dette gjelder særlig ved programmering av kundespesifiserte integrerte kretser. Men for de fleste kombinatoriske og sekvensielle funksjoner kan mange simuleringsverktøy brukes. Det oppfordres derfor til bruk av simuleringsverktøy som støtte for læring.

# Innhold

**Kapittel 1 Innledning....1**
1.1. Design av digital elektronikk....1
1.2. Signaler....2
1.3. MOSFET....5
1.4. MOSFET-svitsjer....12
1.5. Digitale komponenter....16
**Kapittel 2 Tallsystemer....21**
2.1. Innledning....21
2.2. Binære tall....21
2.3. Binær Regning....23
2.3.1. Addisjon....23
2.3.2. Subtraksjon....23
2.3.3. Multiplikasjon....24
2.3.4. Divisjon....24
2.4. Ener- og toer-komplement....24
2.5. Negative tall....25
2.6. Flyt-tall....25
2.7. Aritmetikk....27
2.7.1. Innledning....27
2.7.2. Addisjon....27
2.7.3. Subtraksjon....28
2.7.4. Multiplikasjon....29
2.7.5. Divisjon....30
2.8. Det oktale tallsystem....31
2.9. Det heksadesimale tallsystem....31
2.10. BCD-kode....32
2.11. Gray-kode....33
**Kapittel 3 Logiske porter....35**
3.1. Innledning....35
3.2. Inverterer....35
3.3. OG-port....37
3.4. ELLER-port....39
3.5. NAND-port....41
3.6. NOR-port....44
3.7. Buffer....46
3.8. Eksklusiv ELLER-port....48
3.9. Eksklusiv NOR-port....49
3.10. CMOS-Realisering....50
3.10.1. Innledning....50
3.10.2. NAND-port....51
3.10.3. NOR-port....52
3.10.4. OG-port....53
3.10.5. ELLER-port....54
3.10.6. 3-nivå Buffer....54
**Kapittel 4 Boole’sk algebra....57**

4.1. Innledning....57
4.2. Sum og produkt....57
4.3. Lover i boole’sk algebra....58
4.3.1. Kommutative lov....58
4.3.2. Assosiative lov....58
4.3.3. Distributive lov....58
4.3.4. Regler for Boole’sk algebra....59
4.3.5. De Morgan's teoremer....61
4.4. SOP og POS....62
4.4.1. Sum av produkter, SOP....62
4.4.2. Produkt av summer, POS....63
4.5. Realisering med bare NAND- eller NOR-porter....63
4.5.1. Bare NAND-porter....63
4.5.2. Bare NOR-porter....64
4.6. Forenkling av logiske uttrykk....65
4.6.1. Fra logisk krets....65
4.6.2. Fra logisk uttrykk....66
4.7. Karnaugh-diagram....67
4.7.1. Innledning....67
4.7.2. Diagram for tre variable....68
4.7.3. Diagram for fire variable....69
4.7.4. Diagram for fem variable....71
**Kapittel 5 Kombinatoriske funksjoner....73**
5.1. Innledning....73
5.2. OG-ELLER-port....73
5.3. Buss-driver....73
5.4. Eksklusiv ELLER-port....75
5.5. Paritetsbit-generator....75
5.6. Multipleksere....77
5.7. Demultipleksere....79
5.8. Prioritetskodere....80
5.8.1. Innledning....80
5.8.2. 4-2 prioritetskoder....80
5.8.3. 8-3 prioritetskoder....81
5.9. Desimal til BCD koder....82
5.10. 2-4 og 3-8 dekodere....84
5.10.1. Innledning....84
5.10.2. 2-4 dekoder....84
5.10.3. 3-8 dekoder....85
5.11. BCD til desimal dekoder....86
5.12. Binær/Gray dekoder/koder....87
5.13. Adderere....88
5.14. Subtraktorer....90
5.15. Adderer/Subtraktor....92
5.16. Carry Look Ahead....92
5.17. Komparatorer....94
5.18. BCD til 7-segment dekoder....96

5.19. CMOS-Realisering....97
5.19.1. OG-NOR-port....97
5.19.2. Eksklusiv ELLER....97
5.19.3. Eksklusiv NOR....98
5.19.4. Multipleksere....98
5.19.5. 2-4 dekoder....99
5.19.6. Hel-adderer....100
**Kapittel 6 Låsekretser og vipper....103**
6.1. Innledning....103
6.2. SR-Lås (SR Latch)....103
6.3. SR-vippe (SR Flip Flop)....105
6.4. D-vippe (D Flip Flop)....106
6.5. Master Slave D-vippe....107
6.6. JK-vippe (JK Flip Flop)....108
6.7. Master Slave JK-vippe....109
6.8. Generering av klokkesignal....110
6.9. Asynkron setting og resetting....113
6.10. Tidskarakteristika for vipper....114
6.11. CMOS-realisering....115
6.11.1. SR-lås med NOR-porter....115
6.11.2. SR-lås med NAND-porter....116
6.11.3. SR-vippe....117
6.11.4. D-vippe....118
**Kapittel 7 Skiftregistre....121**
7.1. Innledning....121
7.2. Serie inn/serie ut skiftregister....122
7.3. Serie inn/parallell ut skiftregister....123
7.4. Parallell inn /serie ut skiftregister....124
7.5. Parallell inn /parallell ut register....125
7.6. Parallell inn /serie ut skiftregister....127
7.7. Bidireksjonalt skiftregister....129
7.8. Ringteller....131
7.9. Johnson-teller....132
7.10. Mønstergenerator....133
**Kapittel 8 Tellere....135**
8.1. Innledning....135
8.2. Asynkron 3 bit binærteller....135
8.3. Asynkron mod-6 teller....137
8.4. Asynkron dekadeteller....138
8.5. Asynkrone opp/ned-tellere....140
8.6. Kaskadekopling av tellere....141
8.7. Ulemper med asynkrone tellere....141
8.8. Synkron 3 bit binærteller....142
8.9. Gray-teller....143
8.10. Synkron 4 bit binærteller....144
8.11. Synkron dekadeteller....145
8.12. Synkron 4 bit binær nedteller....146

8.13. Synkrone opp/ned-tellere....147
**Kapittel 9 Sekvensielle kretser....149**
9.1. Innledning....149
9.2. Egenskaper for vipper....150
9.3. Tilstandsdiagram....152
9.3.1. Tilstandsdiagram for vipper....152
9.3.2. Skjema, sannhetstabell og tilstandsdiagram....153
9.4. Design av sekvensielle kretser....155
9.5. Design av tellere....156
9.5.1. 3 bit teller med D-vipper....156
9.5.2. 2 bit Gray opp/ned-teller....158
9.5.3. Synkron mod-6 teller....160
9.5.4. Synkron dekadeteller....162
9.6. Design av tilstandsmaskiner....165
9.6.1. Mønstergjenkjenner....165
9.6.2. Trafikklys-styring....168
**Kapittel 10 Spesialkretser....171**
10.1. Innledning....171
10.2. Scrambler og Descrambler....171
10.3. Barrel Shifter....173
10.4. Multiplikasjons-krets....174
10.5. Divisjons-krets....176
10.6. RAM....178
10.6.1. Innledning....178
10.6.2. SRAM....179
10.6.3. DRAM....180
10.7. ROM....181
10.7.1. Innledning....181
10.7.2. ROM-celle....182
10.7.3. PROM....183
10.7.4. EPROM....184
10.8. ALU....187
**Kapittel 11 Mikrokontrollere....191**
11.1. Innledning....191
11.2. Arkitektur....191
11.3. AVR-arkitektur....193
11.4. CPU....195
11.4.1. ALU....195
11.4.2. Generelle registre....195
11.4.3. Kontrollenhet....196
11.4.4. Funksjonsenheter i kontrollenheten....196
11.5. Internhukommelse....197
11.6. Instruksjonsutførelse....200
11.7. Programmering....200
11.7.1. Innledning....200
11.7.2. Maskininstruksjoner....201
11.7.3. Assemblerinstruksjoner....202

**Kapittel 12 VHDL.....205**
12.1. Innledning.....205
12.2. Strukturell beskrivelse.....205
12.3. Forbindelse mellom blokker.....207
12.4. Dataflytbeskrivelse.....209
12.5. Definisjoner.....210
12.6. Datatyper.....210
12.6.1. Bit og std_logic.....210
12.6.2. Bit_vector og std_logic_vector.....212
12.6.3. Integer og natural.....214
12.6.4. Andre operatorer.....215
12.7. Forløpsbeskrivelse.....216
12.7.1. Innledning.....216
12.7.2. Prosess-instruksjonen.....216
12.7.3. Variable.....217
12.7.4. Sekvensielle instruksjoner.....218
12.7.5. Registre.....219
12.7.6. Tellere.....221
12.7.7. Loop-instruksjonen.....225
12.7.8. Tilstandsmaskiner.....228

# *Forfatteren*

***Knut Harald Nygaard*** er utdannet ingeniør i Elektronikk og i Reguleringsteknikk (Automatisering) samt sivilingeniør i Teleteknikk med spesialisering innen Fysikalsk elektronikk og Radioteknikk med senere videreutdannelse (blant annet ettårig fagstudium). Han har arbeidet som vedlikeholds- og service-ingeniør i Sjøforsvaret. Han har som sivilingeniør arbeidet med utvikling i telekommunikasjonsindustrien og som forsker. Fra 1989 har han arbeidet ved ingeniørutdanningene i Østfold og Oslo, ved sistnevnte fra 1996. Han har undervist i analog og digital elektronikk, signalbehandling, programmering, telekommunikasjon, mikrokontrollere, FPGA/CPLD (med VHDL) og dataverktøy for design og analyse av analog og digital elektronikk. Han har lang erfaring innen undervisning og utarbeiding av undervisningsmateriale.

# Kapittel 1
# Innledning

## 1.1. Design av digital elektronikk

Design av elektronikk har tradisjonelt vært delt mellom design av analoge og digitale systemer. Innen disse to områdene er det videre spesialisering innen for eksempel radiofrekvens-design (analogt), kretsdesign (analogt og digitalt) og design av integrerte kretser (analogt og digitalt). Vi har også i de senere år sett spesialisering innen 'mixed signal' design av integrerte kretser. I tillegg has selvfølgelig også spesialisering innen programmering og programvare.

Digital elektronikk spiller en stadig økende rolle innen industriell elektronikk og ikke minst innenfor forbrukerelektronikk, alt fra elektronikk i biler til elektronikk i hjemmet samt all bærbar elektronikk som blir mer og mer vanlig. Produkter som tidligere var analoge som radio, TV og telefon er nå digitale. I tillegg kommer det nye produkter til bruk i det daglige liv. Med disse nye produktene er levetiden samtidig blitt redusert. En levetid på 1-2 år er ikke unormalt nå, mens det «i gamle dager» var snakk om det tidobbelte.

For å holde tritt med denne raske utviklingen, må elektroniske produkter kunne lanseres og produseres i et raskt tempo. Mens design av analog elektronikk i hovedsak fortsatt er en spesialisert profesjon uten særlig automatisering, er design av digital elektronikk blitt stadig mer avhengig av CAD (Computer aided design), også kjent som Electronic Design Automation (EDA).

Verktøy for EDA muliggjør utførelsen av to oppgaver: Syntese, det vil si ta en spesifikasjon til implementering av et design, og simulering, det vil si der spesifikasjonen eller den detaljerte implementeringen kan undersøkes for å verifisere korrekt virkemåte. EDA-verktøy krever at designet overføres fra designer til vektøyet selv. Dette kan gjøres ved å tegne et skjema av designet ved bruk av et grafisk verktøy som opererer med flere aktuelle tilleggsfunksjoner.

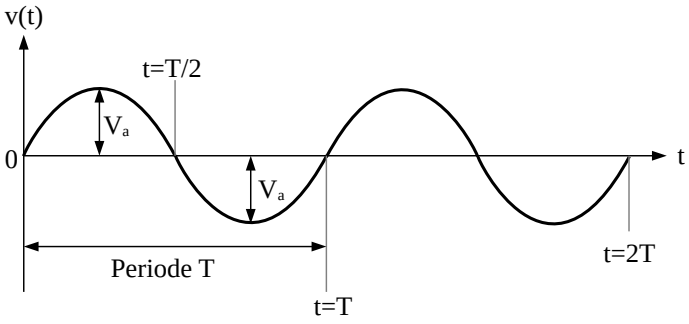
En annen metode kan være å bruke tekstbeskrivelse av designet, mye på samme måte som ved vanlig programmering. Tekstbeskrivelse av digital maskinvare kan gjøres med et modifisert programmeringsspråk som C eller med et maskinvarebeskrivende språk (Hardware Description Language, HDL).

Sistnevnte finnes i hovedsak i de to mest brukte variantene: Verilog og VHDL (V står for VHSIC, Very High Speed Integrated Circuit). Standard HDL er viktig fordi Verilog eller VHDL da kan brukes med forskjellig CAD-verktøy fra forskjellige leverandører. Standardiseringen gjør det mulig både å flytte og dele design mellom de forskjellige verktøyene og mellom designere.

## 1.2. Signaler

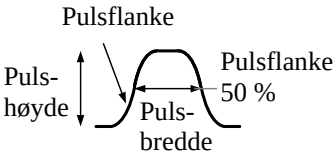
I analogteknikken brukes sinussignaler, særlig i test- og målesituasjoner. Testing med sinussignaler kan brukes for å kartlegge hvordan en krets reagerer på ulike frekvenser, og dermed gi informasjon om hvordan en krets vil håndtere mer kompliserte signaler.

En sinusspenning, se figur 1.1, kan generelt skrives som $v(t) = V_a \sin(2\pi f_0 t)$, der v(t) er momentanverdien [V], $V_a$ amplituden [V] og $f_0$ er frekvensen [Hz]. Perioden er gitt som $T = 1/f_0$.



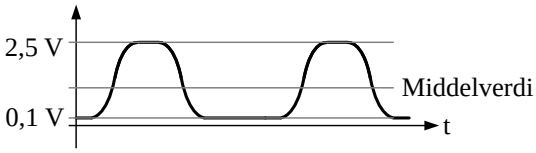

*Figur 1.1. Sinusspenning.*

I digitale elektroniske systemer er det pulstog som er den dominerende signaltypen. Generelt sett er en puls en kortvarig spenningsendring eller et strømstøt. Når pulsene kommer på rekke og rad, kalles det pulstog. Kommer pulsene regelmessig, kan vi også snakke om periode og (repetisjons-)frekvens. Mange forskjellige begreper brukes for å beskrive pulser, og i figurene 1.2 - 1.5 er en del av dem tatt med. Pulser har en viss pulshøyde, som måles i volt, og pulsbredde, varighet, som måles i tid, se figur 1.2.



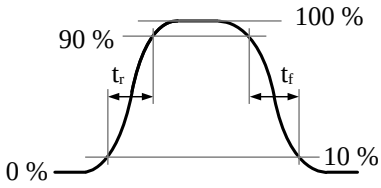

*Figur 1.2. Pulsdefinisjoner (positiv puls).*

Det er vanlig at pulser ikke ligger symmetrisk rundt null. De kan være bare positive, eller bare negative. I for eksempel datautstyr kan spenningen på pulsene ligge mellom +0,1 V og +2,5 V. Vi snakker da om unipolare pulser. Middelverdien til et slikt pulstog er ikke null, se figur 1.3.



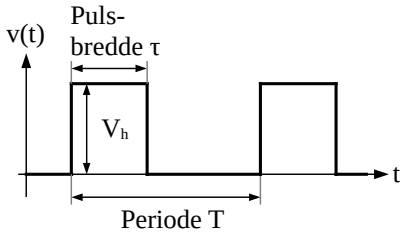

*Figur 1.3. Unipolart pulstog.*

Denne middelverdien kalles ofte for overlagret likespenning eller likespenningskomponenten. Ser vi nærmere på hver enkelt puls, viser det seg at de aldri stiger eller faller momentant. Vi snakker derfor om stigetiden $t_r$ (rise time) og falltiden $t_f$ (fall time). Disse er definert mellom 10% og 90% av det aktuelle spenningsområdet, se figur 1.4.




*Figur 1.4. Definisjoner av stige- og -falltid.*

I mange digitale systemer finner vi pulser med svært liten stige- og falltid i forhold til puls-bredden (= pulsvarigheten). Slike pulser kalles firkantpulser (de har ideelt sett stige- og falltid lik null).




*Figur 1.5. Pulsbredde og periode.*

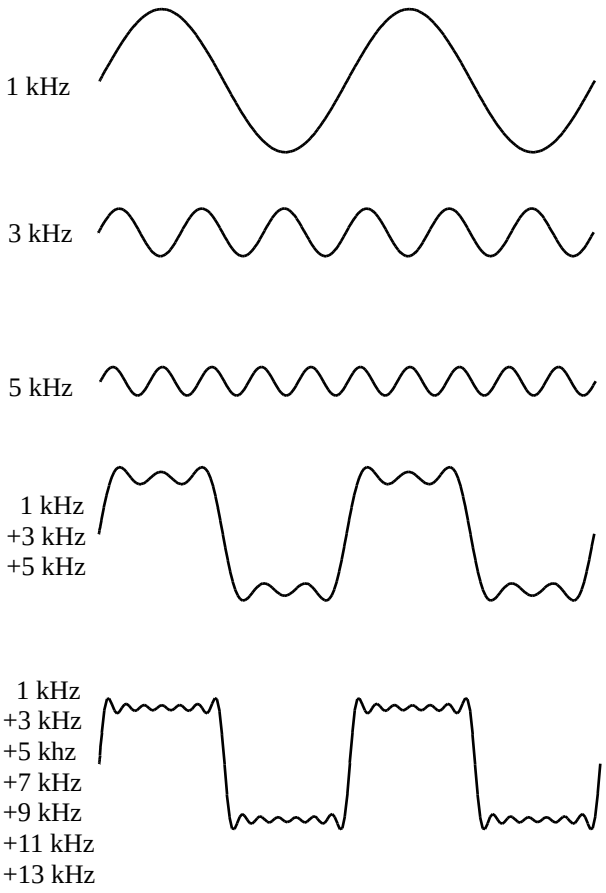
Det er heller ikke sjelden at pulser gjentar seg regelmessig med konstant avstand. Vi definerer da en periode T som vist i figur 1.5, og en repetisjonsfrekvens $f_0 = 1/T$, som for sinussvingninger.

Pulsbredden τ er en annen viktig parameter, og pulsbreddeforholdet τ/T kalles Duty Cycle på engelsk, forkortes DC = τ/T. Dersom τ = ½ T, er DC = 50 %. Dette er typisk for klokkesignaler. Er i tillegg stige- og falltidene mye, mye mindre enn τ, snakker vi om symmetriske firkantpuls-tog. I figur 1.5 foran er DC omtrent 40 %.

I motsetning til et sinussignal inneholder puls-signaler mange forskjellige frekvenser samtidig. Dette gjelder generelt: Alle periodiske signaler som ikke er perfekt sinusformet, har et visst innhold av flere frekvenser samtidig.

For kurveformer med konstant repetisjonsfrekvens kaller vi $f_0 = 1/T$ (periodetiden) for grunnfrekvensen. Da er det også tilstede en viss mengde av den dobbelte frekvensen $f_2 = 2f_0$, den tredoble $f_3 = 3f_0$, og så videre. Dette kan vises matematisk ved å bruke metoden Fourieranalyse. Selve teorien vil vi ikke gå inn på her, men heller ta med et eksempel.

I figur 1.6 ses hva som skjer ved addisjon av tre sinuser; 1 V ved 1 kHz, 0,33 V ved 3 kHz og 0,2 V ved 5 kHz. Tas også med «passe mye» 7, 9, 11 og 13 kHz, fås forløpet nederst i figuren. Om vi fortsetter videre og videre med «passe mye» av stadig høyere frekvenser, ender vi faktisk opp med symmetriske, perfekte firkantpulser. Ja den nederste varianten med 7 ledd ligner ikke så rent lite. Jo flere og høyere frekvenser vi tar med, dess skarpere og jevnere blir resultatet.




*Figur 1.6. Firkantpulstogets oppbygning.*

Omvendt betyr dette at symmetriske firkantpulstog inneholder sinuser med den tredoble, femdoble og så videre av repetisjonsfrekvensen $f_0$ (grunnfrekvensen). Og selvsagt er det slik at alle pulser og kurveformer har et frekvensspekter som er karakteristisk for akkurat den kurveformen. Og felles for alle kurve- og signalformer er dette: Jo raskere signalspenningene endrer verdi, jo kortere stige- og falltider pulsene har, dess høyere frekvenser inneholder de.

## 1.3. MOSFET

Av halvledere i integrerte kretser er MOSFET (Metal Oxide Semiconductor Field Effect Transistor) den transistortypen som er fullstendig dominerende. Denne transistoren er den grunnleggende komponenten i digitale kretser. Dens lille fysiske størrelse gjør at vi kan lage billige og små kretser som for eksempel gigabyte minnebrikker, mikroprosessorer med flere gigahertz klokkefrekvens og mobiltelefoner med mange avanserte funksjoner.

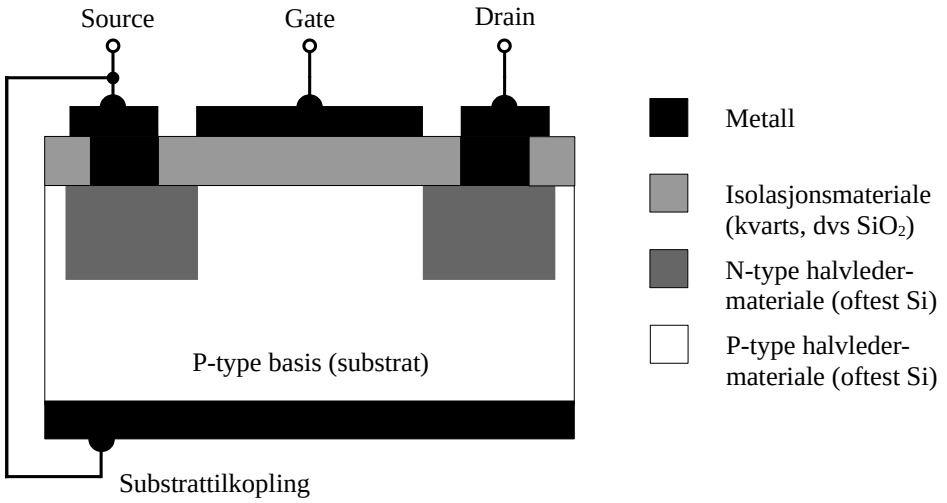
Felteffekttransistorer er av nyere dato enn bipolare transistorer. En felteffekttransistor har en helt annen oppbygning og virkemåte enn dem. Til tross for det er selve grunnprinsippet det samme: Strømmen gjennom komponenten kan styres ved hjelp av en styrespenning på en egen styreelektrode. Som for den bipolare transistoren finnes det komplementære par: N-kanal og P-kanal som svarer til NPN og PNP for den bipolare transistoren.

Selve hovedprinsippet for en felteffekttransistor er at man ved hjelp av doping lager (eller forbereder) en ledende kanal av for eksempel N-type halvledermateriale, der det kan gå strøm mellom to elektroder.

Den ene siden av denne kanalen kalles source, den andre drain. Hvor stor strømmen kan bli, bestemmes av den elektriske ledningsevnen i kanalen, det vil si av kanaltykkelsen og konsentrasjonen av frie ladninger (elektroner i en N-kanal, hull i P-kanal). Poenget er nå at man lett kan påvirke kanaltykkelsen og/eller konsentrasjonen av frie elektroner eller hull. Det gjøres ved å sette spenning på en tredje tilkoplingselektrode, kalt gate, som dermed blir en styreelektrode for drain-source-strømmen.

I det følgende vil vi bruke navnet gate på styre-elektroden selv om navnet grind ses brukt på norsk. Vi bruker også de engelske navnene drain og source selv om det kan ses brukt henholdsvis dren og spring på norsk for disse elektrodene.

For å lage en N-kanals MOSFET, begynner man med et stykke P-type materiale kalt substrat, oftest av silisium, se figur 1.7. Ned i dette doper man inn to N-type fordypninger, "brønner", som blir til drain og source. Oppå der igjen legges et supertynt isolerende sjikt av kvarts (silisiumdioksyd), og det lages metallkontakter ned mot drain og source. Gate legges som en liten metallplate over området mellom drain og source, det er i dette området det vil gå strøm når det settes nok pluss-spenning på gate.



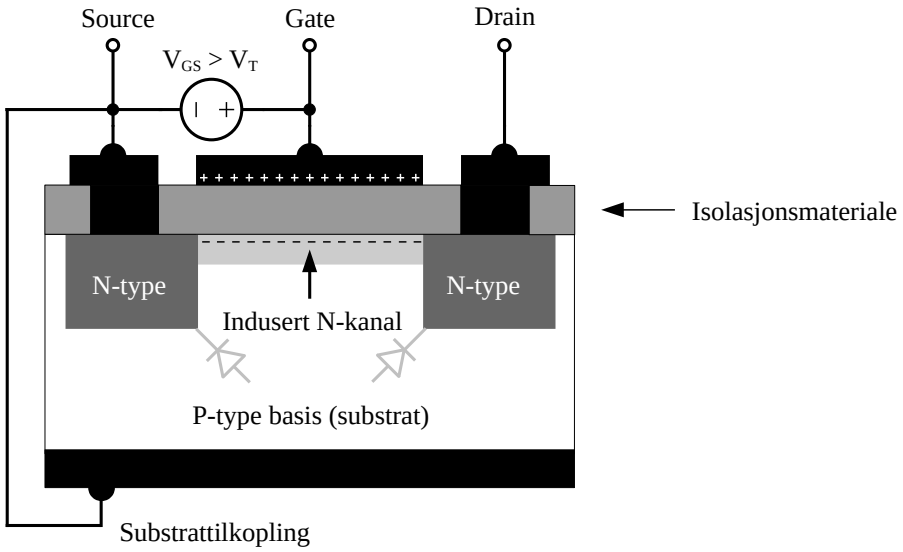

*Figur 1.7. N-kanals lateral utarmings-MOSFET.*

Vi ser nå at substratet, som er P-type, danner en PN-overgang, det vil si en diode, både inn mot source og drain, som begge er N-type. Dette er en lateral MOSFET siden strømmen løper lateralt mellom source og drain. Det finnes også vertikale MOSFET der strømmen løper vertikalt (source over og drain under substatet), men vi vil her holde oss til laterale MOSFET. Og dersom vi prøver å sende strøm fra venstre mot høyre uten spenning på styreelektroden, vil dioden til venstre sperre, og omvendt: Det vil ikke gå strøm noen vei.

Legg merke til at gate og P-substratet til sammen danner en kondensator siden de ligger tett inntil hverandre som to plater med et tynt, isolerende lag mellom seg. Og sender vi positiv spenning på gate i forhold til substratet, lader vi opp denne kondensatoren. Det kommer da positiv ladning inn på den øverste kondensatorplaten, det vil si på gate.

I tillegg vil det også komme negativ ladning inn på den nederste kondensatorplaten, substratet, som er P-type og derfor fra før har et overskudd av frie (positive) hull. Den negative kondensatorladningen tiltrekkes av positiv gate, og blir derfor liggende i sjiktet helt inn mot isolasjonslaget, der overskuddet av positiv ladning (hull) derfor blir mindre.

Etterhvert som gate-spenningen stiger og denne kondensatoren lades mer og mer opp, blir overskuddet av hull i substratet nærmest gate, mindre og mindre, helt til vi ikke lenger har noe overskudd her og bildet skifter: Vi får et elektronoverskudd, slik som vi finner i en N-type halvleder, se figur 1.8.




*Figur 1.8. Dannelse av kanal.*

Substratet helt inntil isolasjonslaget er faktisk blitt forandret fra P-type til N-type. Vi har altså fått et sammenhengende N-område hele veien mellom source og drain, med store konsekvenser, for dette området vil selvsagt lede strøm. Vi sier at vi har fått indusert en N-kanal gjennom substratdelen.

Og jo større gate-spenningen blir, dess større blir ladningstettheten i dette sjiktet, og jo bredere blir det, og dess bedre vil det lede strøm. Terskelspenningen $V_T$ for MOSFET, betingelsen for at det skal gå strøm, for at det skal bli indusert en kanal, er at gatespenningen er tilstrekkelig stor. For små MOSFET er denne spenningen bare noen hundre mV.

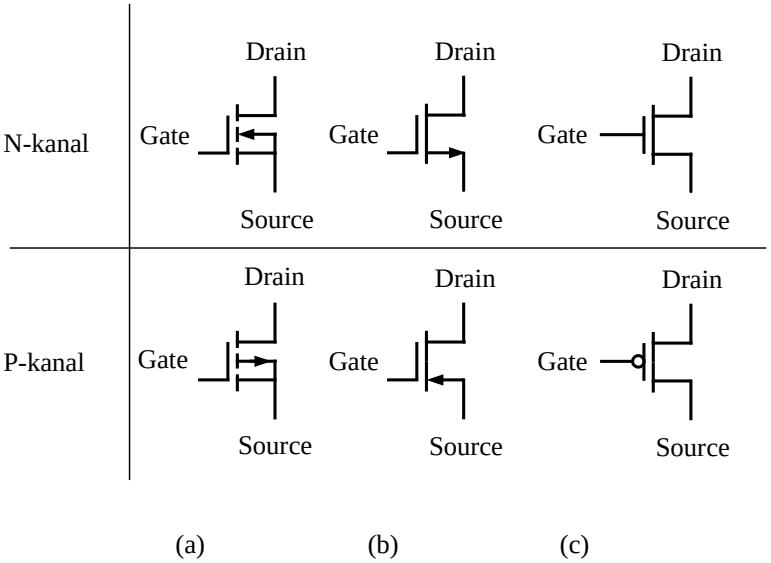
At substratet danner PN-overganger, dioder, inn mot source og drain, har konsekvenser for hvilke spenninger og strømretninger en MOSFET kan operere med. For normalt skal det ikke gå strøm i disse diodene. Substratet må derfor helst ikke bli positivt i forhold til disse to, det koples derfor vanligvis til den ene siden av kanalen, source, som dermed blir referansepunktet for gate-spenningen. Denne sammenkoplingen går igjen i symbolet for komponenten.

Siden gate er svært godt isolert fra kanalen, går det aldri likestrøm i den, bortsett fra en veldig liten lekkstrøm. Ved normal bruk, altså uten diodestrøm, er kanalstrømmen derfor den samme på drainsiden som i source, og $I_D = I_S$ = kanalstrømmen.

Alt som er sagt foran om N-kanals FET, gjelder også for en P-kanals FET, bare at vi må bytte N ut med P, pluss med minus, og omvendt.

For å lage en P-kanals MOSFET starter vi med et N-type substrat. Ned i dette doper man inn to P-type fordypninger, «brønner», som blir til drain og source. For at denne transistoren skal lede strøm, må gate være negativ, og drain skal normalt være negativ i forhold til source.

I figur 1.9 er vist symboler i bruk for MOSFET av type anrikning (Enhancement) som vi vil konsentrere oss om her.




*Figur 1.9. Symboler for anrikning MOSFET.*

I figur 1.9a er vist symbolene som samsvarer best med den fysiske oppbyggingen av transistoren. Substratet er vist koplet til source. Med diskrete transistorer er dette ofte tilfellet. Innenfor integrerte kretser er ofte substratet koplet til 0 V for N-kanal og til positiv forsyningsspenning for P-kanal. I symbolet for P-kanals MOSFET er pilen snudd.

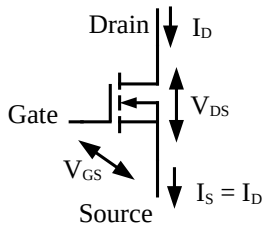
I figur 1.9b er vist symbolene som ofte benyttes i analogteknikk. Her er ofte substrat og source koplet sammen. Legg forøvrig merke til pilretningen for symbolene sammenlignet med symbolene i figur 1.9a.

I figur 1.9c er det ikke indikert noen substrat-tilkopling, og symbolene viser ingen forskjell mellom source og drain. Disse symbolene ses brukt særlig innen CMOS integrerte kretser. For N-kanal transistoren kreves logisk høy for at den skal slås på mens det kreves logisk lav, indikert ved en ring på gate, for å slå på P-kanal transistoren. Substratet er som oftest koplet til 0 V for N-kanal og til positiv forsyningsspenning for P-kanal.

I figur 1.10 er vist betegnelser for en N-kanal MOSFET:

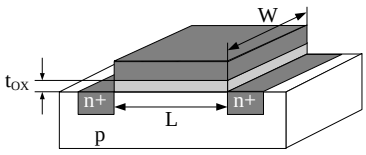
$I_D$: Drainstrøm (= sourcestrømmen).
$I_S$: Sourcestrøm (= drainstrømmen).
$V_{GS}$: Spenning mellom gate og source.
$V_{DS}$: Spenning mellom drain og source.

I tillegg kommer som nevnt $V_T$, terskelspenning (Threshold Voltage) for drain-source- strøm.




*Figur 1.10. Betegnelser for strømmer og spenninger.*

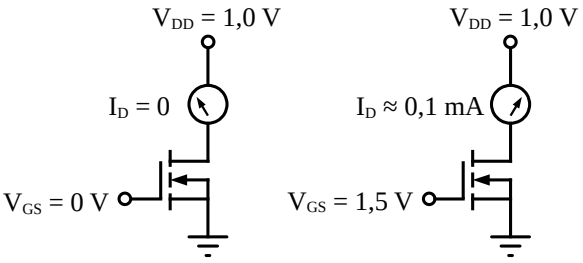
I figur 1.11 er vist en skjematisk fremstilling av geometrien til en N-kanal MOSFET, der L betegner gate-lengden. Når det tales om transistorstørrelse, er det denne størrelsen det henvises til. Tidligere var denne lik flere mikrometer, men er nå kommet ned i nanometer-området, mindre enn 50 nm er nå ikke uvanlig.




*Figur 1.11. MOSFET geometri.*

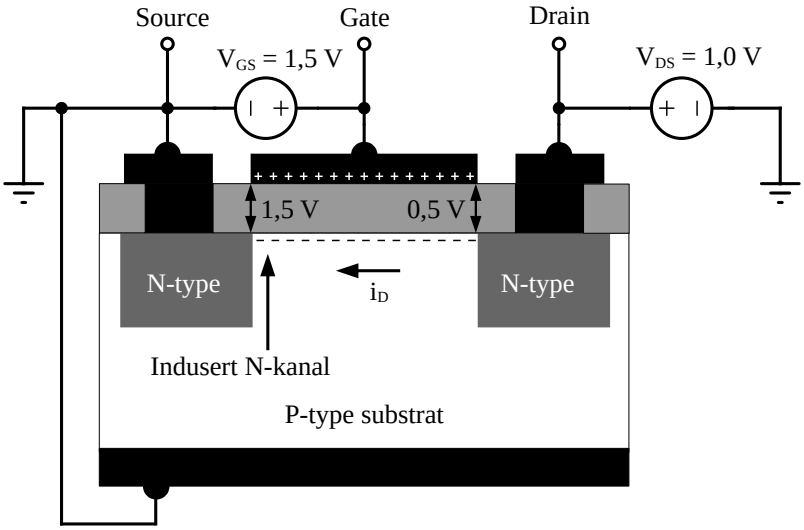
Størrelsen W angir gate-bredden. Størrelsen $t_{OX}$ angir tykkelsen på isolasjonsmaterialet. Dette er meget tynt (mindre enn 10 nm er ikke uvanlig). Imidlertid vil gate-kapasiteten pr arealenhet, $C_{OX}$, øke inverst med tykkelsen. Den totale gate-kapasiteten kan tilnærmes å være lik $C_{GS} = WLC_{OX}$. Drain-strømmen er tilnærmet proporsjonal med gate-bredden W og gate-kapasiteten pr. arealenhet, $C_{OX}$, mens den er omvendt proporsjonal med gate-lengden L.

Til venstre i figur 1.12 er vist en MOSFET der source og gate er koplet til 0 V ($V_S = V_G = 0$ V) og vi legger en spenning på 1,0 V mellom drain og source ($V_{DS}$ = 1,0 V). $V_{DD}$ er forsyningsspenningen, her antatt lik 1,0 V.




*Figur 1.12. Strøm i MOSFET.*

Når $V_{GS}$ = 0 V, vil det ikke gå noe strøm siden $V_{GS}$ er under den minsteverdien $V_T$ som skal til for å lage - indusere - en ledende kanal. Dersom vi øker $V_{GS}$, skjer det ingenting før vi passerer terskelspenningen for transistoren. Og etterhvert som gatespenningen øker, øker også strømmen. Vi stiller inn på $V_{GS}$ = 1,5 V slik som vist i figuren til høyre, og vi får da et bilde av kanalen som vist i figur 1.13.



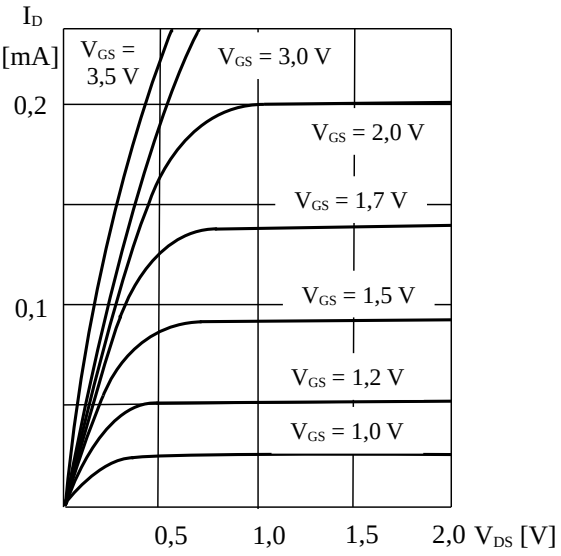

*Figur 1.13. Strøm i MOSFET.*

Kanaltykkelsen avhenger av spenningen mellom gate og kanal: Den øker når $V_{GS}$ øker, og omvendt. Et viktig poeng her er at spenningen mellom gate og kanalen nå ikke er den samme overalt.

Nærmest source er den 1,5 V (siden $V_{GS} = V_G - V_S$), men etterhvert som vi beveger oss mot høyre i kanalen over mot drain, så avtar den jevnt til $V_G - V_D$ = 1,5 V − 1,0 V = 0,5 V helt inn mot drain.

Og det må bety at kanalen smalner av fra source til drain, og at motstanden mellom source og drain er litt større nå enn når det ikke er spenning mellom source og drain.

Økes kanalspenningen $V_{DS}$, går det mer strøm, men samtidig blir det mindre spenning mellom gate og drain. Derfor blir den ledende kanalen smalere når vi nærmer oss drain-siden. Det fører da til at motstanden i kanalen mellom drain og source øker når spenningen øker. Dermed øker ikke lenger strømmen i takt med spenningen, og det ser vi som at $V_{GS}$-$I_D$-kurven flater ut, se karakteristikken for transistoren i figur 1.14.

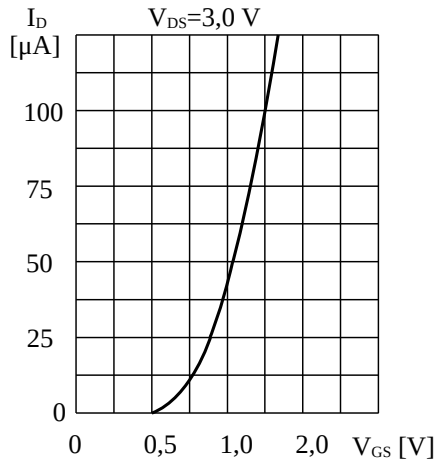



*Figur 1.14. Karakteristikk for N-kanal MOSFET.*

I figuren er vist flere kurver, der hver kurve svarer til en bestemt styrespenning $V_{GS}$. Disse kurvene kalles ofte for utgangskarakteristikken. Høyere styrespenninger $V_{GS}$ gir, naturlig nok, større kanalstrøm (= $I_D$ = $I_S$). Det er også lett å se at når $V_{GS}$ øker, øker kanalstrømmen mer og mer, men det er ikke er en lineær sammenheng her.

Dersom drain-source-spenningen blir stor nok, vil kanalen bli så smal at den er klemt igjen der den når drain. Dette kalles pinch-off (sammenkniping) på engelsk. Øker vi $V_{DS}$ enda mer, blir sammenknipingen enda sterkere, derfor går det ikke særlig mer strøm, og ved enda litt større $V_{DS}$ blir $V_{GS}$-$I_D$-kurven helt horisontal. Spenningen når dette skjer, kalles pinchoff-spenningen. Området til høyre for pinchoff, der $V_{DS}$ er «stor» og $I_D$ ikke lenger er avhengig av $V_{DS}$, kalles for konstantstrømområdet (eller metningsområdet).

I figur 1.15 er vist hvordan strømmen $I_D$ i kanalen avhenger av gate-sourcespenningen $V_{GS}$ i konstantstrømområdet. Legg merke til terskelspenningen $V_T$, den avleses der kurven kommer ned til tilnærmet 0 µA, og i figur 1.15 ser vi at i dette tilfellet er $V_T \approx 0{,}5$ V.

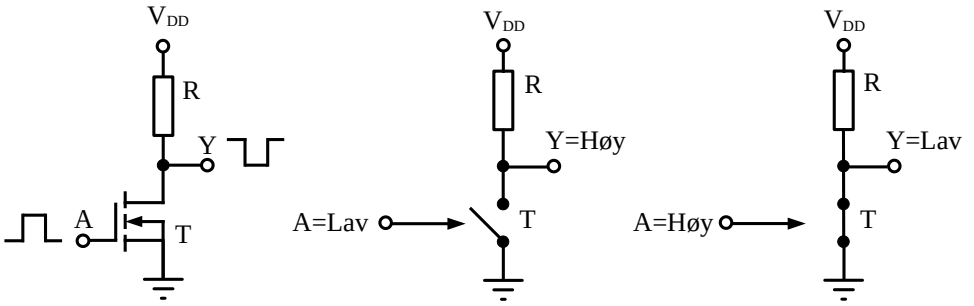



*Figur 1.15. Overførings-karakteristikk for MOSFET.*

## 1.4. MOSFET-svitsjer

I digitalteknikken brukes MOSFET som av/på-brytere og invertereren er en viktig byggekloss. Med en N-kanal MOSFET kan vi tenke oss koplingen til venstre i figur 1.16, der forsyningsspenningen er $V_{DD}$.

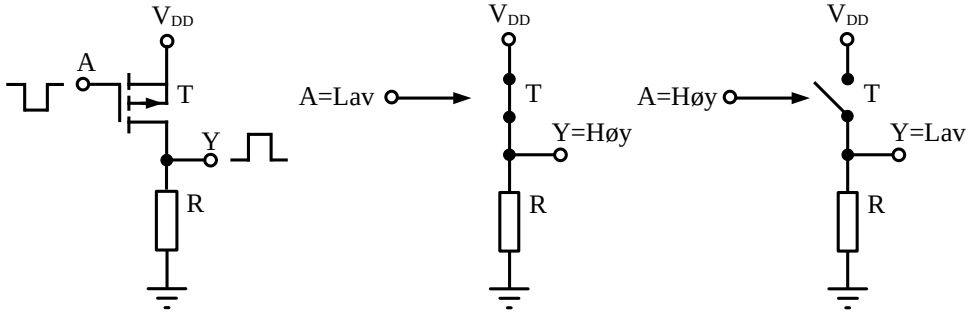
Når gate-spenningen i A er 0 V, vil transistoren være av, det vil si at det går ingen strøm mellom drain og source. Da er det heller ikke noe spenningsfall over motstanden R og utgangen Y ligger på forsyningsspenningen $V_{DD}$. Vi sier at inngangen er lav (logisk 0) og utgangen høy (logisk 1), som vist midt på figuren, der transistoren er erstattet med styrt bryter.




*Figur 1.16. N-kanal MOSFET-svitsj som realiserer inverterer.*

Er gate-spenningen i A lik $V_{DD}$, for eksempel lik 2,5 V, vil transistoren være på, det vil si at det går strøm mellom drain og source. Da er det spenningsfall over motstanden R, og spenningen over transistoren vil være tilnærmet lik 0 V, vi ligger helt til venstre i karakteristikken i figur 1.14. Følgelig ligger utgangen Y på tilnærmet 0 V. Vi sier at inngangen er høy (logisk 1) og utgangen lav (logisk 0), som vist til høyre i figur 1.16, der transistoren igjen er erstattet med styrt bryter.

Vi kan også tenke oss en inverterer realisert ved hjelp av P-kanal MOSFET, som vist til venstre i figur 1.17. Er gate-spenningen i A lik 0 V, vil transistoren være på, det vil si at det går strøm mellom drain og source. Da er det spenningsfall over motstanden R, og spenningen over transistoren vil være tilnærmet lik 0 V. Følgelig ligger utgangen Y på tilnærmet $V_{DD}$. Vi sier igjen at inngangen er lav (logisk 0) og utgangen høy (logisk 1), som vist midt på figuren.

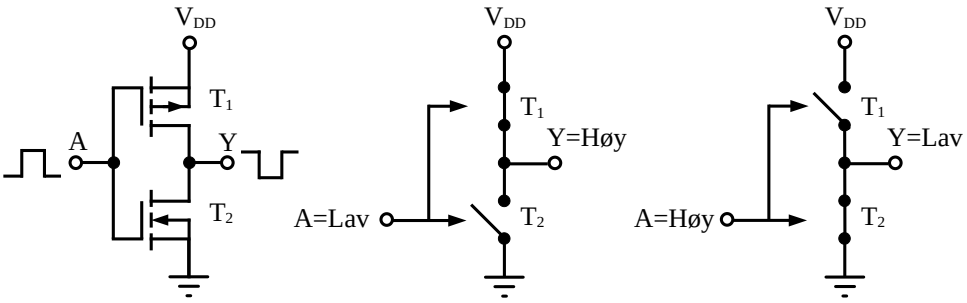



*Figur 1.17. P-kanal MOSFET-svitsj som realiserer inverterer.*

Når gate-spenningen i A er $V_{DD}$, vil nå transistoren være av, det vil si at det går ingen strøm mellom drain og source. Da er det heller ikke noe spenningsfall over motstanden R og utgangen Y ligger på 0 V. Vi sier igjen at inngangen er høy (logisk 1) og utgangen lav (logisk 0), som vist til høyre i figur 1.17.

En fordel MOSFET har i forhold til bipolare transistorer, er at med MOSFET i integrerte kretser kan man langt på vei unngå motstander og dermed både bruke liten plass og lite strøm. Den komplementære egenskapen til N-kanal og P-kanal MOSFET er utnyttet i logisk design med CMOS, Complementary MOS. Integrerte kretser med CMOS kan da bare inneholde N- og P-kanals vanlig MOSFET.

Den grunnleggende kretsen i CMOS er invertereren vist til venstre i figur 1.18. Den øverste transistoren er P-kanal, PMOS, mens den nederste er N-kanal, NMOS. Det ses at vi har kombinert svitsjene i figur 1.16 og 1.17.




*Figur 1.18. CMOS inverterer.*

Når inngangsspenningen i A er 0 V, er N-kanalen $T_2$ av, mens P-kanalen $T_1$ er på, som vist midt i figuren, der transistorene er erstattet med styrte brytere. Og når inngangsspenningen er $V_{DD}$, er N-kanalen $T_2$ på, mens P-kanalen $T_1$ er av, som vist til høyre i figuren. Med andre ord: Høy inn gir lav ut, og omvendt, noe som karakteriserer en inverterer. Transistorene trekker kun strøm når de svitsjer fra høy til lav og motsatt. Er inngangen permanent lav eller høy, trekker ikke transistorene strøm siden de er seriekoplet og den ene transistoren alltid er av. Og inngangene trekker normalt ingen strøm.

Men jo høyere svitsjehastighet, jo mer strøm vil CMOS trekke og jo større blir effektforbruket. Dette skyldes blant annet at kapasitetene i transistorene vil få større betydning. En annen viktig egenskap å merke seg er at jo mindre fysisk størrelse transistorene har, jo mindre strøm vil de da trekke. Dette skyldes at mindre transistorer har mindre kapasitet som må lades opp og ut. Men det finnes imidlertid en nedre grense hva størrelse angår, vi får da sekundære effekter som påvirker effekttapet i negativ retning.

Selv om det er vanlig å si at logisk 1 er lik $V_{DD}$, for eksempel lik 2,5 V, og at logisk 0 er lik 0 V, kan et visst spenningsområde representere hva som tolkes som logisk 1 og 0. Med en forsyningsspenning $V_{DD}$ = 2,5 V, kan vi operere med verdiene vist i tabell 1.1. Fra tabellen ses at maksimal utgangsspenning som representerer logisk 0 er lik $V_{OLmaks}$ = 0,75 V. Alle spenninger opp til $V_{ILmaks}$ = 1,05 V er imidlertid tolket som logisk 0. Forskjellen på 0,3 V representerer støymarginen. Dette kommer av at utgangsspenningen tåler å få opp til $V_{ILmaks} - V_{OLmaks}$= 0,3 V addert støy uten at det oppstår feil.

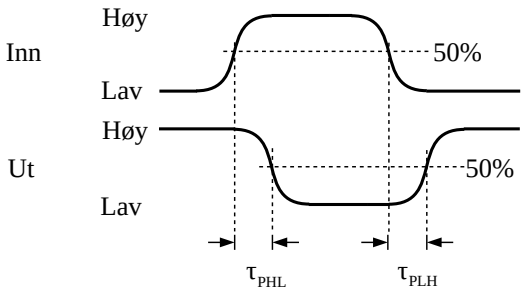
Tilsvarende ses at minimum spenning som tolkes som logisk 1 er $V_{IHmin}$ = 1,35 V mens minimum utgangsspenning er $V_{OHmin}$ = 1,75 V. Dette gir en støymargin på $V_{OHmin} - V_{IHmin}$ = 0,4 V. Generelt gjelder at jo større støymargin, jo bedre er systemet rustet mot feil. Selv om vi her bare har brukt invertereren som eksempel, vil de data som er presentert også gjelde andre typer porter. Invertereren er en type port som har en inngang og en utgang. En nær beslektet port er NAND-porten. Den vil ha to eller flere innganger og en utgang. Logiske porter kan generelt ha flere innganger, men normalt bare ha en utgang.

| Parameter | Beskrivelse | Verdi |
|---|---|---|
| $V_{IHmaks}$ | Maksimal spenning tolket som logisk 1 | 2,50 V |
| $V_{IHmin}$ | Minimum spenning tolket som logisk 1 | 1,35 V |
| $V_{ILmaks}$ | Maksimum spenning tolket som logisk 0 | 1,05 V |
| $V_{ILmin}$ | Minimum spenning tolket som logisk 0 | 0,00 V |
| $V_{OHmaks}$ | Maksimum spenning generert som logisk 1 | 2,50 V |
| $V_{OHmin}$ | Minimum spenning generert som logisk 1 | 1,75 V |
| $V_{OLmaks}$ | Maksimum spenning generert som logisk 0 | 0,75 V |
| $V_{IHmaks}$ | Minimum spenning generert som logisk 0 | 0,00 V |

*Tabell 1.1. Spenningsnivåer for CMOS med $V_{DD}$ = 2,5 V.*

Den engelske betegnelsen fan-out brukes for å gi et tall på hvor mange ekvivalente portinnganger en gitt logisk utgang kan drive. Siden inngangen på portene i CMOS er kapasitive, trenges en viss tid for å lade opp og ut kondensatoren på inngangen. Det betyr at portens svitsjehastighet bestemmer størrelsen på fan-out. I tillegg vil forbindelsen mellom portene være kapasitive, og denne kapasiteten kommer i tillegg til inngangskapasiteten. Det er faktisk slik at denne forbindelseskapasiteten i integrerte kretser kan være dominerende. Og da kan ikke denne kapasiteten finnes før utlegget av kretsen er utført.

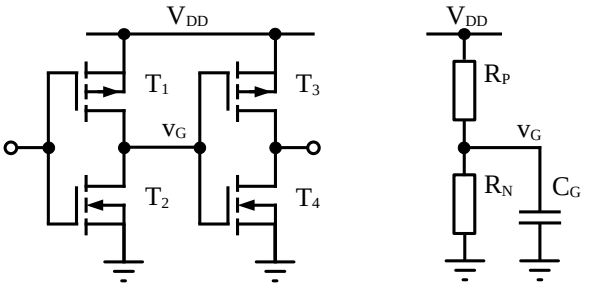
Transportforsinkelsen, latensen, (Propagation delay, Latency) mellom inngang og utgang bestemmer hvor raskt vi kan operere svitsjene. Når vi påtrykker en puls på inngangen av for eksempel en CMOS inverterer, vil det gå en viss tid til utgangen responderer, se figur 1.19.




*Figur 1.19. Transportforsinkelse.*

Vi måler da tiden mellom 50 % av pulshøyden på inngangsignalet til 50 % av pulshøyden på utgangsignalet. Vi får da tidene $t_{PHL}$ og $t_{PLH}$ for henholdsvis høy-lav-transisjon og lav-høy-transisjon. Det er åpenbart at klokkeperioden må være større enn disse tidene for korrekt operasjon.

Figur 1.20a viser en inverterer som driver en annen inverterer. I figur 1.20b er vist ekvivalenten for denne koplingen. Hvis den første invertereren svitsjer fra logisk 1 ($V_{DD}$) til logisk 0 (0 V) ved tiden t = 0, og vi antar at den ekvivalente motstanden $R_N$ er mye mindre enn $R_P$, er gatespenningen $v_G$ gitt som: $v_G = V_{DD}\, e^{-t/R_N C_G}$




*(a)* *(b)*

*Figur 1.20. Inverterere i kaskade (a) med ekvivalent (b).*

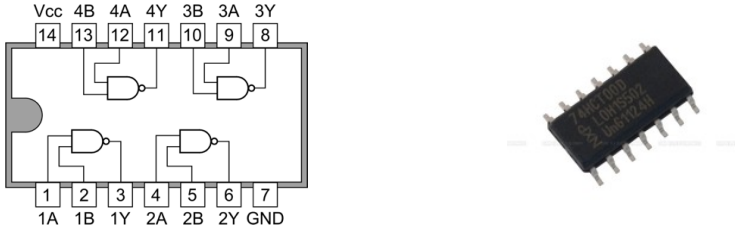
Dersom vi bruker verdiene for $V_{DD}$ = 2,5 V, $V_{IHmin}$ = 1,35 V og $V_{ILmaks}$ = 1,05 V fra tabell 1.1 og antar at $R_N$ = 40 ohm og $C_G$ = 50 pF, kan vi bruke uttrykket ovenfor. Da får vi at tiden det tar for spenningen $v_G$ å falle fra 1,35 V til 1,05 V er:

$$t = -R_N C_G \ln\left(V_{ILmaks}/V_{IHmin}\right) = -40 \cdot 50 \cdot 10^{-12} \ln\left(1{,}05/1{,}35\right) = 0{,}5\ \text{ns}$$

Må den første invertereren drive to innganger, vil tiden dobles. Og som nevnt vil også forbindelsen mellom portene ha betydning. Med andre ord, er som nevnt, fan-out avhengig av hvor raskt vi ønsker at kretsen skal svitsje.

## 1.5. Digitale komponenter

Foruten porter er det nødvendig med minne-elementer for å kunne lage digitale kretser der utgangssignaler kan være en funksjon av både nåværende inngangssignal og tidligere inngangssignaler. Både porter og vipper, der sistnevnte utgjør minne-elementene, kan fås som separate komponenter. Således er det mulig å kjøpe for eksempel 6 inverterere i en pakke, fire OG-porter i en pakke eller to datavipper i en pakke. I figur 1.21 er vist som eksempel fire NAND-porter i en pakke.




*Figur 1.21. Fire NAND-porter i en pakke.*

Et digitalt system kunne bygges ved å binde sammen for eksempel hundre slike elementer. Selv om hver av disse komponentene ikke koster så mye hver for seg, ville en slik løsning være lite kostnadseffektiv, også fordi alt fra design til utlegg av kretskort vil være tidkrevende og kostbart. I tillegg ville en slik løsning ta stor plass og bruke større effekt enn nødvendig.

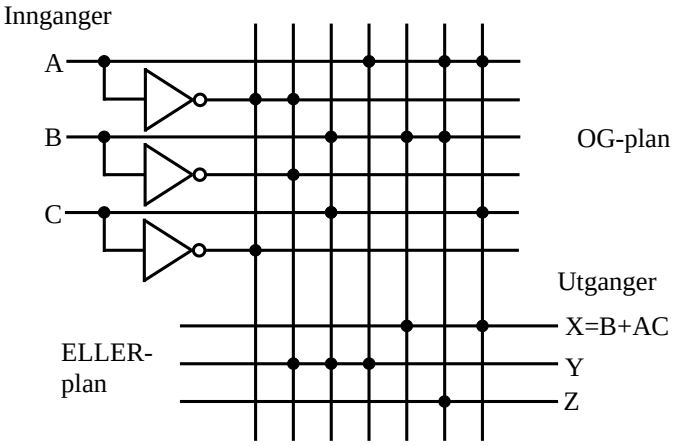
Adskillig mer kompliserte funksjoner er tilgjengelig i masseproduserte integrerte kretser, for eksempel mikrokontrollere og mikroprosessorer. Med økende kompleksitet kommer fleksibilitet. En mikroprosessor kan programmeres til å utføre en uendelighet av oppgaver og en mikrokontroller kan brukes til alt fra automatisering til helt trivielle oppgaver. Imidlertid kan mangel på hurtighet begrense anvendelsesområdet.

Design av digital elektronikk kan derfor i noen tilfeller gjøres ved å benytte standard (avanserte) komponenter og binde dem sammen, ofte med programmering som en ekstra oppgave i tillegg. Det er imidlertid uunngåelig at sider av designet ikke kan realiseres ved hjelp av standardkomponenter. Designeren står da overfor valget å bruke diskrete porter eller å designe en spesialisert integrert krets for å utføre oppgaven. Sistnevnte kan i mange tilfeller også være kostnadseffektivt, siden kostnader som nevnt også kan knyttes til at kretskortutlegg med mange komponenter kan være kostbart.

Design av en integrert krets helt fra bunnen av, med det menes helt på transistornivå, er selvfølgelig ingen enkel oppgave. Det har imidlertid vært mulig i lang tid å kunne designe såkalte semi-custom integrerte kretser kalt gate-arrays, som navnet sier er dette en integrert krets som inneholder port-matriser.

Design med applikasjons-spesifikke integrerte kretser (Application-specific Integrated Circuit, ASIC), der port-matriser brukes, går derfor ut på å bestemme hvordan matrisene skal forbindes (for eksempel med metall-ledere) for å oppnå det ønskede designet. En ASIC kan således lages slik at all prosessering av den integrerte kretsen er gjort på forhånd med unntak av den siste biten med metallforbindelser som designeren legger inn helt til slutt. Dermed kan port-matriser produseres i store antall, og prisen er ikke avskrekkende høy.

En annen måte å lage små kundespesifiserte integrerte kretser på, er å bruke programmerbar logikk. Første generasjon av disse var programmerbare logiske matriser (Programmable Logic Arrays, PLAs) som bare inneholdt kombinatorikk i form av OG-porter og ELLER-porter ferdige på en brikke som så kunne forbindes av designeren. Figur 1.22 viser prinsippet. En PLA har flere innganger og utganger, et OG-plan og et ELLER-plan. I figuren er vist noen forbindelser mellom inngangene og produktleddene samt mellom sumleddene og utgangene.




*Figur 1.22. Prinsipp for PLA.*

Mange programmerbare kretser bruker dobbel-gate MOSFET. Forbindelser er realisert med disse transistorene. Én gate på transistoren er forbundet med inngangen mens den andre gate'en er flytende. Programmeringen består i å tilføre ladning til sistnevnte gate. Da slås transistoren av, og en allerede for-programmert forbindelse brytes. Dersom ladning ikke tilføres, beholdes forbindelsen. Med denne teknikken kan PLA reprogrammeres. PLA kan også programmeres på andre måter, men metoden beskrevet er sannsynligvis den mest brukte.

En annen familie av programmerbar logikk er programmerbar matriselogikk (Programmable Array Logic, PAL) der PLA utvides med noen minne-elementer, vipper, slik at også sekvensielle kretser kan designes. I praksis kan dette gjøres ved at utgangene i figur 1.22 føres til vippene.

Flere PAL-lignende makroceller realiserer programmerbare logiske kretser (Programmable Logical Devices, PLDs) og er tilgjengelig med stadig flere transistorer for å kunne realisere mer komplekse kombinatoriske og sekvensielle funksjoner. Disse går under navnet CPLD (Complex PLD).

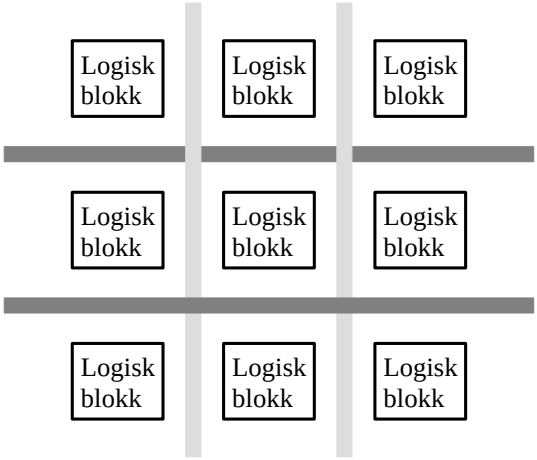
En CPLD er en kombinasjon av programmerbare kretser og en rekke makroceller. Hver PLD kan reprogrammeres og kan utføre en rekke logiske funksjoner. Makroceller er funksjonelle blokker som realiserer reprogrammerbar kombinatorikk eller sekvensiell logikk. De er også meget fleksible ved å ha tillegg for inverterte funksjoner og flere forskjellige tilbakekoplinger.

CPLD er et godt alternativ for realisering av ganske avansert maskinvare ved ikke altfor store design. De er også relativt rimelige i pris.

Feltprogrammerbare portmatriser (Field Programmable Gate Arrays, FPGAs) er de mest avanserte programmerbare kretsene som etterhvert har gjort det mulig å lage avanserte design av digitale systemer. Teknologien er imidlertid ulik den som brukes for PAL, PLA og CPLD. De forskjellige produsentene opererer også med forskjellig teknologi.

En FPGA kan sies å være bygd opp av mange logiske blokker i en matrise, se figur 1.23. Det er koplinger mellom de forskjellige blokkene, vist som vertikale og horisontale busser. Disse kan være av forskjellig lengde og håndtere forskjellige hastigheter. Noen kan være lokale mens andre kan forbinde de logiske blokkene med for eksempel inn/ut-porter eller minneblokker. I hver logiske blokk ligger styringen av klokkene.

Logiske blokker kan bestå av adaptive logiske moduler som kan konfigureres til å implementere logiske og aritmetiske funksjoner samt registre. De kan også brukes som minne. Adaptive logiske moduler kan for eksempel være bygd opp av oppslagstabeller (’Look Up Tables, LUTs), vipper eller RAM (’Random Access Memory’)-blokker.




*Figur 1.23. Logisk blokkmatrise i FPGA.*

Til forskjell fra en ASIC kan en FPGA omprogrammeres istedenfor at kretsen må produseres på nytt hver gang designet skal modifiseres. Men det ses at FPGA brukes til prototyping mens ASIC brukes til å implementere det endelige designet.

Det brukes ulike former for programmeringsteknologi for FPGA, som Antifuse-basert, SRAM-basert og Flash-basert. SRAM (’Static RAM’) har i motsetning til Antifuse og Flash den ulempe at den må omprogrammeres ved hvert spenningspåslag siden informasjonen mistes når spenningen slås av.

Funksjonen til en FPGA blir ofte definert i et maskinvarebeskrivende språk slik som Verilog eller VHDL (se kapittel 12). Koden blir så oversatt til teknologispesifikk konfigureringskode (kompilert) og overført til FPGA. Vanlige funksjonsblokker som RAM, FIFO (’First In, First Out Memory’), PCI (’Peripheral Component Interconnect’)-grensesnitt og Ethernet-grensesnitt er som regel tilgjengelig fra FPGA-produsenten.

FPGA med hele prosessorer kan være tilgjengelige, herunder også dedikerte digitale signalprosessorer (’Digital Signal Processor, DSP’). I tillegg er det også mulig å få FPGA med noen analoge funksjoner som Analog/Digital-og Digital/Analog-omformere. Dette muliggjør såkalt ’Mixed Signal’-design.

Den høye graden av integrering hjelper til med å redusere effektforbruk til digitale systemer. I tillegg kommer kostreduksjon på grunn av færre komponenter. Siden systemomfanget blir mindre, leder dette også til større pålitelighet siden de fleste feil har en tendens til å skyldes feil på kretskortene istedenfor i selve halvlederbrikkene.

# Kapittel 2
# Tallsystemer

## 2.1. Innledning

Daglig benytter vi det desimale tallsystem uten å tenke noe særlig over det. Når vi snakker om datamaskiner, er vi derimot nødt til å finne en måte å representere tall på og hvordan beregninger skal foretas.

Det er vanskelig å lage en elektronisk krets som kan innta ti forskjellige tilstander for å representere tallene 0 til 9. I stedet benyttes transistoren som svitsj, den er enten på eller av, og da kan vi representere tallet 0 og 1. Dette er et binært siffer, kalt bit.

Bit er engelsk akronym for binary digit (etter datapioneren John W. Tukey). Ordet bit betyr forøvrig (også på norsk) lite stykke.

En gruppe på 4 bit kalles en nibbel (engelsk: nibble) og en gruppe på 8 bit kalles en byte. Betegnelsen byte brukes også mest på norsk, men vi kan også bruke betegnelsen oktett. En gruppe på 16 bit er 2 byte, 24 bit er 3 byte og så videre.

I tillegg brukes kilobyte (KB), som er 1024 ($=2^{10}$) byte, megabyte (MB) som er 1024 kilobyte, gigabyte (GB) som er 1024 megabyte, terabyte (TB) som er 1024 gigabyte, petabyte (PB) som er 1024 terabyte, exabyte (EB) som er 1024 petabyte, zettabyte (ZB) som er 1024 exabyte og yottabyte (YB) som er 1024 zettabyte.

## 2.2. Binære tall

Som en introduksjon til binære tall, kan vi se på tallrekken 0-15 desimalt. Denne er gjengitt i tabellen nedenfor sammen med den tilsvarende binære tallrekken.

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

Når vi teller desimalt, teller vi fra 0 og opp til 9. Så må vi legge til et nytt siffer til venstre (lik 1) før vi starter tellingen på nytt med sifrene 0, 1, 2 etc. Et binært siffer, ett bit, kan bare ha to verdier, 0 og 1. Når vi teller binært, må vi telle 0, 1, 10, 11, 100 og så videre som vist i tabellen, siden vi har bare to siffer istedenfor ti. Som vi ser, trenger vi to  bit for å telle til 3, tre bit for å telle til 7 og fire bit for å telle til 15. Generelt kan vi telle opp til $2^n - 1$ når vi har n bit tilgjengelig. Er for eksempel n = 8, svarer dette desimalt opp til 255.

I det følgende vil vi benytte indeks for å skille mellom de forskjellige grunntallene, slik at $15_{10} = 1111_2$ som vi ser av tabellen over.

Binærtallet 1110,0110 svarer tydeligvis til $14_{10}$ foran komma, men hva svarer så $0110_2$ bak komma til? Svaret er at vi benytter samme fremgangsmåte for binære tall som for desimaltall. Foran komma dobles verdien for hvert bit. Etter komma må følgelig verdien halveres for hvert bit. I vårt eksempel kan vi følgelig skrive: $1110{,}0110_2 = 1\cdot2^3 + 1\cdot2^2 + 1\cdot2^1 + 0\cdot2^0 + 0\cdot2^{-1} + 1\cdot2^{-2} + 1\cdot2^{-3} + 0\cdot2^{-4}$. Følgelig er $1110{,}0110_2 = 14{,}375_{10}$ (siden $0{,}010_2 = 0{,}250_{10}$ og $0{,}001_2 = 0{,}125_{10}$).

I tabellen nedenfor er gjengitt desimalverdiene for spennet $2^8$ til $2^{-6}$.

| $2^8$ | $2^7$ | $2^6$ | $2^5$ | $2^4$ | $2^3$ | $2^2$ | $2^1$ | $2^0$ | $2^{-1}$ | $2^{-2}$ | $2^{-3}$ | $2^{-4}$ | $2^{-5}$ | $2^{-6}$ |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 256 | 128 | 64 | 32 | 16 | 8 | 4 | 2 | 1 | 1/2<br>0,5 | 1/4<br>0,25 | 1/8<br>0,125 | 1/16<br>0,0625 | 1/32<br>0,03125 | 1/64<br>0,015625 |

Tabellen kan brukes for å konvertere fra desimaltall til binærtall. La oss si at vi ønsker å finne hva 74 er binært. Vi ser at $2^6 = 64$. Da gjenstår $74-64$ som er $2^3 = 8$ pluss $2^1 = 2$. Vi kan da følgelig skrive $74_{10} = 1001010_2$. Samme fremgangsmåte kan brukes for tall etter komma.

En annen metode for å konvertere heltall fra desimalt til binært er gjentatt divisjon med to. Et eksempel forklarer metoden enklest. La oss si at vi ønsker å finne 74 binært:

| | | |
|---|---|---|
| 74:2 = 37 | Rest 0 | Dette er minst signifikante bit, LSB |
| 37:2 = 18 | Rest 1 | |
| 18:2 = 9 | Rest 0 | |
| 9:2 = 4 | Rest 1 | |
| 4:2 = 2 | Rest 0 | |
| 2:2 = 1 | Rest 0 | |
| 1:2 = 0 | Rest 1 | Dette er mest signifikante bit, MSB |

Svaret er følgelig 1001010 som vi fant tidligere. Legg forøvrig merke til at vi stopper divisjonen når kvotienten er lik 0.

Når vi skal finne desimalene binært, kan vi bruke en metode med gjentatt multiplikasjon med to. La oss ta som eksempel 0,375:

| | | |
|---|---|---|
| $0{,}375\cdot2 = 0{,}75$ | Mente 0 | Dette er mest signifikante bit, MSB |
| $0{,}75\cdot2 = 1{,}50$ | Mente 1 | |
| $0{,}50\cdot2 = 1{,}00$ | Mente 1 | |

Svaret er følgelig 0,011 som vi har sett tidligere. Her tar vi vare på menten, som gir oss våre binære siffer. Når vi fortsetter multiplikasjonen, er det med menten fratrukket forrige multiplikasjonsresultat. I vårt eksempel her var det naturlig å slutte fordi vi stod igjen med 0 til slutt. Men det trenger ikke bestandig å være slik. Ta for eksempel 0,367:

0,367·2 = 0,734 Mente 0 Dette er mest signifikante bit, MSB
0,734·2 = 1,468 Mente 1
0,468·2 = 0,936 Mente 0
0,936·2 = 1,872 Mente 1
0,872·2 = 1,744 Mente 1
0,744·2 = 1,488 Mente 1
0,488·2 osv

Når skal vi stoppe? Svaret er faktisk gitt av hvor mange bit vi har tilgjengelig. Har vi for eksempel tre bit tilgjengelig, kan svaret være 0,010 eller 0,011, der vi i det siste tilfellet har foretatt en forhøyning (avrunding).

# 2.3. Binær Regning

## 2.3.1. Addisjon

Det er ikke problematisk å se at 0+0 = 0, 0+1 = 1, 1+0 = 1 og 1+1 = 10 (= 2 desimalt). Mente genereres som for desimaltallene. Anta at vi har to tall, A og B, som skal summeres, slik at C = A+B. I tabellen nedenfor er det vist et par eksempler (også med desimalverdier).

| Mente | | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | | | | | | | | | | 1 | | | | | |
| A | | | | | 1 | | | 1 | | | | 0 | 1 | | | 1 |
| B | + | 1 | 0 | 0 | + | 4 | + | | | | 1 | 1 | | + | | 3 |
| C=A+B | = | 1 | 0 | 1 | = | 5 | = | 1 | 0 | 0 | | | | = | | 4 |

## 2.3.2. Subtraksjon

Det er ikke vanskelig å se at 0−0 = 0, 1−1 = 0, 1−0 = 1 og 10−1 = 1. I sistnevnte tilfelle illustreres hva som skjer dersom vi prøver å utføre 0−1. Vi må da låne fra mer signifikante bit. Anta at vi skal subtrahere tallet B fra A, slik at C = A−B. I tabellen nedenfor er det vist et par eksempler (også med desimalverdier).

| Låne | | | | | | | | | | | 10 | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| A | | 1 | 0 | 1 | | 5 | | | 1 | 0 | 1 | | | 5 |
| B | − | 1 | 0 | 0 | − | 4 | − | | | 1 | 1 | | − | | 3 |
| C=A−B | = | 0 | 0 | 1 | = | 1 | = | 0 | 1 | 0 | | | = | | 2 |

### 2.3.3. Multiplikasjon

Det er ikke problematisk å se at 0·0 = 0, 0·1 = 0, 1·0 = 0 og 1·1 = 1. Anta at vi har to tall, A = 111 (desimalt 7) og B = 101 (desimalt 5), som skal multipliseres, slik at C = A·B. I tabellen nedenfor er dette vist. Resultatet er 100011 (desimalt 35).

| | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| | | | | 1 | 1 | 1 | · | 1 | 0 | 1 | |
| | | | | 1 | 1 | 1 | | | | | |
| | | | 0 | 0 | 0 | | | | | | |
| | | 1 | 1 | 1 | | | | | | | |
| = | 1 | 0 | 0 | 0 | 1 | 1 | | | | | |

### 2.3.4. Divisjon

Divisjon kan også utføres på samme måte som for desimaltall. Anta at vi skal dele A = 10011 (desimalt 35) med B = 101, slik at C = A:B. I tabellen nedenfor er dette vist. Resultatet er 111 (desimalt 7).

| | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 0 | 0 | 0 | 1 | 1 | : | 1 | 0 | 1 | = | 1 | 1 | 1 |
| | 1 | 0 | 1 | | | | | | | | | | |
| | | 1 | 1 | 1 | | | | | | | | | |
| | | 1 | 0 | 1 | | | | | | | | | |
| | | | 1 | 0 | 1 | | | | | | | | |
| | | | 1 | 0 | 1 | | | | | | | | |

## 2.4. Ener- og toer-komplement

Enerkomplement finnes ved å erstatte alle 0-ere med 1 og motsatt. Denne operasjonen kalles invertering. For et 8 bit ord A = 00101101 er enerkomplementet B = 11010010.

Toerkomplementet finnes ved først å ta enerkomplementet og så legge til 1 (i minst signifikante bit). For et 8 bit ord A = 00101101 er enerkomplementet B = 11010010 og toerkomplementet følgelig lik C = 11010011. Også her er det viktig å tenke på antall bit vi har til rådighet, noe vi skal komme tilbake til.

Ønsker vi å komme tilbake til den opprinnelige binære formen fra ener-komplementet, er det bare å invertere alle bit. Vi ser at B ovenfor (= 11010010) blir til A = 00101101 ved en slik invertering.

Ønsker vi å komme tilbake til den opprinnelige binære formen fra toer-komplementet, tas først ener-komplementet av toer-komplement-tallet og så legge til 1 (i LSB). Vi ser at ener-komplementet av C ovenfor blir til 00101100, og legger vi 1 til dette, er vi tilbake til A.

## 2.5. Negative tall

### *Fortegnsbit*

Vanligvis benyttes et bit for å angi fortegn. I hovedsak benyttes mest signifikante bit til dette formålet. Vanligvis angir 0 at det er et positivt tall, mens 1 angir et negativt tall.

### *Fortegn og tallverdi*

En kan tenke seg å bruke fortegnsbit (MSB) og de resterende bit til å angi tallverdien. Hvis vi holder oss til 8 bit, vil A = 00100010 og B = 10100010 være henholdsvis +34 og −34, der vi bruker 7 bit til å angi tallverdien. Dette er ingen lur måte å representere negative tall på, som vi skal se senere.

### *Ener-komplement*

Her representeres positive tall ved sin tallverdi som foran, A = 00100010 er +34 når vi bruker åtte bit. Skal vi ha −34, tas ener-komplementet av 34, som gir B = 11011101. Dette er heller ingen lur måte å representere negative tall på.

### *Toer-komplement*

Her representeres positive tall ved sin tallverdi som foran, A = 00100010 er +34 når vi bruker åtte bit. Skal vi ha −34, tas toer-komplementet av 34, som gir B = 11011110. Så kan man jo spørre seg hvorfor denne metoden å representere negative tall på, foretrekkes.

### *Tall-område*

Hvis vi holder oss til 8 bit og toer-komplement (der MSB angir fortegnet), blir det største positive tallet vi kan ha, lik $2^7-1 = 127$. Vi skal også kunne representere 0. Tilsvarende blir det minste negative tallet lik $-2^7 = -128$. Skal vi kunne øke verdien, må vi opp med antall bit. Bruker vi 16 bit kan vi dekke området −32768 til 32767. Generelt får vi da området fra $-2^{n-1}$ til $2^{n-1}-1$, der n er antall bit.

## 2.6. Flyt-tall

Som nevnt gir åtte bit, en byte, mulighet for å lagre tall (uten fortegnsbit) opp til en størrelse desimalt lik 256. Med 16 bit, 2 byte, kan vi da lagre opp til desimalt 65536. For å lagre store tall, må vi ta i bruk flere byte. Skal vi håndtere også deltall, må antall byte økes ytterligere.

Dersom vi tar i bruk flyt-tall, har vi mulighet til å lagre store tall uten at antall byte blir uoverkommelig. Flyt-tallet består av fortegn pluss to deler, kalt mantisse og eksponent. Mantissen er et tall mellom 0 og 1 mens eksponenten angir antall plasser kommaet (binærpunktet) skal forskyves. Desimalt kan tallet 361.508.600 dermed dekomponeres til en mantisse på 0,3615086 og en eksponent på 9. Tallet kan med andre ord skrives som $0{,}3615086 \cdot 10^9$.

For binære flyt-tall finnes det standardiserte formater. Vi skal her se nærmere på Single precision som er på 32 bit, vist nedenfor.

| S | E | F |
|---|---|---|
| 1 bit | 8 bit | 23 bit |
| 0 | 10001011 | 00010010110100000000000 |

S: Fortegnsbit (Sign Bit)

E: Eksponent

F: Mantisse (Fraction)

Komma (binærpunktet) er antatt å være til venstre for de 23 bit. I realiteten er det 24 bit siden mest signifikante bit alltid er lik 1. Denne opptar ikke noen plass, men er underforstått å være der.

Normalt må en eksponent ha et fortegnsbit. For å unngå dette, benyttes en metode med å forskyve eksponenten ved å legge 127 til den virkelige eksponenten. Dette muliggjør at vi kan representere eksponent-verdier fra −126 til +128.

Anta at vi ønsker å representere tallet 1000100101101 som flyt-tall. Dette kan skrives som $1{,}000100101101 \cdot 2^{12}$. Eksponenten blir 12+127 = 139 = 10001011. Mantissen blir 000100101101 (1 foran komma sløyfes). Antas tallet positivt, blir fortegnsbit lik 0. Tallet representeres da som vist over.

Anta at vi har et flyt-tall som vi ønsker å finne binærtallet til. Generelt kan dette finnes som:
Binærtall = $(-1)^S(1+F)(2^{E-127})$
La oss ta tallet nedenfor som eksempel:

| S | E | F |
|---|---|---|
| 1 | 10010110 | 11010000000000000000000 |

Verdien av eksponenten er $10110110 = 150_{10}$. Innsatt i formelen fås:
Binærtall = $(-1)^1(1{,}1101)(2^{150-127}) = -1{,}1101 \cdot 2^{23} = -11101000000000000000000000$.
Dette svarer til desimaltallet −15.204.352.

Siden vi kan ha eksponent-verdier fra −126 til +128, kan meget små og meget store tall representeres. Ved hjelp av disse 32 bit kan vi representere opp til 129 bit for heltall. Til slutt bør det bemerkes at 0 representeres ved bare 0 i flyt-tall-formatet mens uendelig representeres som bare 1 i eksponenten og bare 0 i mantissen.

# 2.7. Aritmetikk

## 2.7.1. Innledning

Vi skal her se på binær aritmetikk. Vi begrenser oversikten til å operere kun med toer-komplement. For enkelhets skyld vil vi også begrense oss til å se på tall med fast ordlengde. Prinsipielt er imidlertid fremgangsmåten den samme med flyt-tall.

## 2.7.2. Addisjon

Anta at vi har to positive tall, A og B, hvert på 8 bit (en byte) som skal summeres, slik at C = A+B, der også C er på åtte bit. I det første eksemplet summerer vi A = 00101101 (= $+45_{10}$) og B = 01000110 (= $+70_{10}$):

| Mente | | | | 1 | 1 | | | |
|---|---|---|---|---|---|---|---|---|
| A | 0 | 0 | 1 | 0 | 1 | 1 | 0 | 1 |
| B | 0 | 1 | 0 | 0 | 0 | 1 | 1 | 0 |
| C | 0 | 1 | 1 | 1 | 0 | 0 | 1 | 1 |

Svaret blir C = 01110011 (= $115_{10}$) som forventet. I neste eksempel summerer vi A = 01101101 (= $109_{10}$) med B = 01000110 (= $+70_{10}$):

| Mente | 1 | | | 1 | 1 | | | |
|---|---|---|---|---|---|---|---|---|
| A | 0 | 1 | 1 | 0 | 1 | 1 | 0 | 1 |
| B | 0 | 1 | 0 | 0 | 0 | 1 | 1 | 0 |
| C | 1 | 0 | 1 | 1 | 0 | 0 | 1 | 1 |

Nå blir svaret C = 10110011 (= $-77_{10}$), og det er ikke summen av 109 og 70 som jo er 179. Det vi ser et eksempel på her, er at ordlengden er for kort for å kunne romme resultatet. Vi ser at fortegnsbit for sluttresultatet C er forskjellig fra fortegnsbit for A og B.

I mikrokontrollere er det en mekanisme for å håndtere slike tilfeller som dette ved at det ukorrekte fortegnsbit som genereres, setter et flagg (som det heter), kalt Overflow. Imidlertid vil vi også få generert et Mente-flagg. Det betyr at det er mulig å få tak i det korrekte resultatet ved å kombinere disse flaggene med verdien i C (uten fortegnsbit). Legg forøvrig merke til at sluttverdien i C er korrekt dersom vi ikke opererte med fortegnsbit ( $10110011_2 = 179_{10}$). For å kunne legge sammen større tall, kan vi selvfølgelig øke ordlengden: Å bruke to byte istedenfor ett er ganske vanlig.

## 2.7.3. Subtraksjon

Når vi bruker toer-komplement, kan subtraksjon utføres som addisjon siden C = A–B kan beregnes som C = A+(–B). Dette gjelder også om begge tallene er negative, idet C = –A–B kan beregnes som C = (–A)+(–B). Bruk av toer-komplement er dermed overlegen enerkomplement og metoden med tallrepresentasjon der det brukes fortegn og tallverdi.

### *Positivt tall A og negativt tall B, A ≥ \|B\|*

Anta at vi har et positivt tall A og et negativt tall B med en tallverdi som er mindre enn A. Vi antar at begge tallene har en byte ordlengde, og at vi bruker toer-komplement. Sluttresultatet er C, også med en ordlengde på åtte bit. Vi summerer A = 01000111 (= $+71_{10}$) og B = 11010011 (= $-45_{10}$):

| Mente | 1 | 1 | | | | 1 | 1 | 1 | |
|---|---|---|---|---|---|---|---|---|---|
| A | | 0 | 1 | 0 | 0 | 0 | 1 | 1 | 1 |
| B | | 1 | 1 | 0 | 1 | 0 | 0 | 1 | 1 |
| C | | 0 | 0 | 0 | 1 | 1 | 0 | 1 | 0 |

Svaret blir C = 00011010 (= $26_{10}$) som forventet. Legg merke til at den siste menten som genereres, forkastes. I en mikrokontroller vil det heller ikke settes noe flagg.

### *Positivt tall og negativt tall, A < \|B\|*

Anta at vi har et positivt tall A og et negativt tall B med en tallverdi som er større enn A. Vi antar at begge tallene har en byte ordlengde, og at vi bruker toer-komplement. Sluttresultatet er C, også med en ordlengde på åtte bit. Vi summerer A = 00101101 (= $+45_{10}$) og B = 10111001 (= $-71_{10}$):

| Mente | | | 1 | 1 | 1 | | | 1 | |
|---|---|---|---|---|---|---|---|---|---|
| A | | 0 | 0 | 1 | 0 | 1 | 1 | 0 | 1 |
| B | | 1 | 0 | 1 | 1 | 1 | 0 | 0 | 1 |
| C | | 1 | 1 | 1 | 0 | 0 | 1 | 1 | 0 |

Svaret blir C = 11100110 (= $-26_{10}$) som forventet.

### *To negative tall*

Anta at vi har to negative tall A og B. Vi antar at begge tallene har en byte ordlengde og at vi bruker toer-komplement. Sluttresultatet er C, også med en ordlengde på åtte bit. Vi summerer A = 11010011 (= $-45_{10}$) og B = 10111001 (= $-71_{10}$):

| Mente | 1 | 1 | 1 | 1 | | | 1 | 1 | |
|---|---|---|---|---|---|---|---|---|---|
| A | | 1 | 1 | 0 | 1 | 0 | 0 | 1 | 1 |
| B | | 1 | 0 | 1 | 1 | 1 | 0 | 0 | 1 |
| C | | 1 | 0 | 0 | 0 | 1 | 1 | 0 | 0 |

Svaret blir C = 10001100 (= $-116_{10}$) som forventet. Legg merke til at den siste menten blir forkastet siden fortegnsbit i A, B og C er det samme.

I neste eksempel summerer vi A = 10010011 (= $-109_{10}$) med B = 10111010 (= $-70_{10}$):

| Mente | 1 | 0 | 1 | 1 | | | 1 | | |
|---|---|---|---|---|---|---|---|---|---|
| A | | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 1 |
| B | | 1 | 0 | 1 | 1 | 1 | 0 | 1 | 0 |
| C | | 0 | 1 | 0 | 0 | 1 | 1 | 0 | 1 |

Svaret blir C = 01001101 (= $+77_{10}$) og det er ikke summen av −109 og −70 som jo er −179. Det vi igjen ser, er et eksempel på at ordlengden er for kort for å kunne romme resultatet. Vi ser at fortegnsbit for sluttresultatet C er forskjellig fra fortegnsbit for A og B. Dette er samme situasjon vi hadde med to positive tall som førte til Overflow, og vi kan også her få tak i det korrekte resultatet.

### 2.7.4. Multiplikasjon

Når vi multipliserer for eksempel multiplikanden 21 med multiplikatoren 18, betyr dette at vi skal addere multiplikanden 18 ganger. Vi bruker normalt å utføre denne prosessen ved å multiplisere med et siffer av gangen samtidig som vi flytter en plass til venstre. Vårt regnestykke ser da slik ut:

| 2 1 x 1 8 | | |
|---|---|---|
| 1 6 8 | 21 addert 8 ganger | |
| 2 1 | 21 addert 1 gang | Skiftet en plass til venstre |
| 3 7 9 | Produkt = Sum | |

Det er med andre ord to prosesser involvert: Addisjon og skifting. Innenfor digitalteknikken benyttes en adderer til addisjon mens det benytter et skiftregister til skifting-operasjonen.

Når vi multipliserer multiplikanden $35_{10} = 100011_2$ med multiplikatoren $5_{10} = 101_2$, ser dette regnestykket slik ut:

| | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| | | 1 | 0 | 0 | 0 | 1 | 1 | x | 1 | 0 | 1 |
| | | 1 | 0 | 0 | 0 | 1 | 1 | | | | Multiplikand |
| | 0 | 0 | 0 | 0 | 0 | 0 | | | | | Skift (ingen addisjon) |
| 1 | 0 | 0 | 0 | 1 | 1 | | | | | | Skift (addisjon) |
| 1 | 0 | 1 | 0 | 1 | 1 | 1 | 1 | | | | Produkt (=175) |

Vi kan nevne at når multiplikatoren har 0 som siffer, foretas bare en skift-operasjon og ingen addisjon. Har multiplikatoren 1 som siffer, foretas først en skift-operasjon og så en addisjon. Det bør også bemerkes at addisjon foretas forløpende (såfremt multiplikatorens siffer er lik 1), som sum mellom bare to tall.

Fortegnet til produktet er positivt såfremt multiplikand og multiplikator har samme fortegn og negativt dersom de har ulikt fortegn. Multiplikasjonen foretas med tallenes sanne verdi, det vil si uten (toer-)komplement.

## 2.7.5. Divisjon

Ved divisjon kaller vi resultatet for kvotient når vi dividerer dividend med en divisor. Er for eksempel dividenden lik $23_{10}$ og divisor lik $7_{10}$, blir kvotienten lik heltallsdelen 3 med en rest på 2. Kvotienten forteller med andre ord antall ganger divisoren går opp i dividenden. La oss først se hvordan operasjonen kan gjøres i titalls-systemet. Regnestykket er da:

| | | | |
|---|---|---|---|
| | 23 | | Dividend |
| − | 7 | Nr 1 | Subtraher divisor |
| | 16 | | Foreløpig rest |
| − | 7 | Nr 2 | Subtraher divisor |
| | 9 | | Foreløpig rest |
| − | 7 | Nr 3 | Subtraher divisor |
| | 2 | | Heltallsrest |

Som vi ser, har vi utført subtraksjonen 3 ganger, som gir oss kvotienten = 3. Skal vi gjøre operasjonen binært, benyttes toer-komplementet av divisoren. Som med multiplikasjon, vil samme fortegn gi positiv kvotient mens ulikt fortegn gir negativ kvotient. Når vi dividerer dividenden $23_{10} = 00010111_2$ med divisoren $7_{10} = 00000111_2$, ser dette regnestykket slik ut:

| | | |
|---|---|---|
| 0 0 0 1 0 1 1 1 | | Dividend |
| 1 1 1 1 1 0 0 1 | Nr. 1 | Addér toer-komplement av divisor |
| 0 0 0 1 0 0 0 0 | | Foreløpig rest |
| 1 1 1 1 1 0 0 1 | Nr. 2 | Addér toer-komplement av divisor |
| 0 0 0 0 1 0 0 1 | | Foreløpig rest |
| 1 1 1 1 1 0 0 1 | Nr. 3 | Addér toer-komplement av divisor |
| 0 0 0 0 0 0 1 0 | | (Foreløpig) rest |
| 1 1 1 1 1 0 0 1 | Nr. 4 | Addér toer-komplement av divisor |
| 1 1 1 1 1 0 0 1 | | Rest ≤ 0 |

Prosessen stoppes når resultatet er Rest ≤ 0. Dette får vi etter nr. 4. Følgelig ser vi at svaret er kvotient lik $3_{10} = 00000011_2$ med rest lik $2_{10} = 00000010_2$ (som ventet).

Det skal ikke stor fantasi til for å se at denne algoritmen er meget ineffektiv. I praksis vil en derfor benytte metoder som gir færre trinn for å komme fram til resultatet. Ikke overraskende baserer disse seg på skifting og addisjon (med toer-komplement).

## 2.8. Det oktale tallsystem

Det oktale tallsystem har to færre symboler enn det desimale. Som navnet sier, består det av åtte siffer: 0, 1, 2, 3, 4, 5, 6 og 7. Binært svarer dette til 000, 001, 010, 011, 100, 101, 110 og 111. Dette innebærer at for eksempel $6_8 = 110_2$, mens $10_8 = 1000_2$, $11_8 = 1001_2$, $12_8 = 1010_2$, etc.

Eksempelvis vil $375_8$ desimalt være lik $3 \cdot 8^2 + 7 \cdot 8^1 + 5 \cdot 8^0 = 253_{10}$. Men legg merke til at binært kan dette tallet skrives 011 111 101, altså 375 siffer for siffer. Det oktale tallsystem er ikke særlig utbredt siden det svarer til bare tre binære siffer.

## 2.9. Det heksadesimale tallsystem

I motsetning til det oktale tallsystem, er det heksadesimale tallsystem i utstrakt bruk, særlig innen mikrokontrollere. Det heksadesimale tallsystem har 16 symboler, de ti laveste er de samme som for det desimale tallsystem, mens de seks neste symbolene er bokstavene A, B, C, D, E og F. I tabellen er vist sammenhengen mellom de desimale, binære og heksadesimale (nederst i tabellen) tallene opp til desimalt 15.

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

Når vi teller heksadesimalt, teller vi fra 0 og opp til 9, så videre til A, B, C, D, E, F, 10, 11, 12,...19, 1A, 1B....1F, 20, 21, etc. Legg merke til at hvert heksadesimale symbol svarer til 4 bit. Dette er grunnen til populariteten.

Eksempelvis vil $1A75_{16}$ desimalt være lik $1 \cdot 16^3 + 10 \cdot 16^2 + 7 \cdot 16^1 + 5 \cdot 16^0 = 6773_{10}$. Men legg merke til at binært kan dette tallet skrives 1 1010 0111 0101, altså 1A75 siffer for siffer. Tilsvarende vil for eksempel $1011\ 0100\ 1110\ 1001_2 = B4E9_{16}$ siden vi kan slå sammen fire og fire bit.

Dersom en ønsker å konvertere et desimaltall (heltall) til et heksadesimalt tall, kan det gjøres ved å først konvertere til et binærtall for så å slå sammen 4 og 4 binære siffer til et heksadesimalt tall. Binært svarer $675_{10}$ til $10\ 1010\ 0011_2$. Heksadesimalt er dette lik $2A3_{16}$. Vi kan selvfølgelig også foreta konverteringen direkte. Med vårt eksempel $675_{10}$ får vi:

675:16 = 42,1875 ⇒ 0,1875·16 = 3 Dette er minst signifikante siffer, LSD
42:16 = 2,625 ⇒ 0,625·16 = $10_{10}$ = A
2:16 = 0,125 ⇒ 0,125·16 = 2 Dette er mest signifikante siffer, MSD

Operasjonen stoppes når heltallskvotienten er lik 0. Vi har også benyttet de engelske betegnelsene LSD (’Least Significant Digit’) for minst signifikante siffer og MSD (’Most Significant Digit’) for mest signifikante siffer. Svaret er da at $675_{10} = 2A3_{16}$ som funnet tidligere.

Dersom en ønsker å konvertere desimaldelen i et desimaltall til et heksadesimalt tall, kan det gjøres ved å først konvertere til et binærtall for så å slå sammen 4 og 4 binære siffer til et heksadesimalt tall. Binært svarer $0,675_{10}$ til $0,1010\ 1100\ 1100\ ..._2$. Heksadesimalt er dette lik $0,ACC..._{16}$. Vi kan selvfølgelig også foreta konverteringen direkte. Med vårt eksempel $0,675_{10}$ får vi:

0,675·16 = 10,8 ⇒ $10_{10}$ = A Dette er mest signifikante siffer etter komma
0,8·16 = 12,8 ⇒ $12_{10}$ = C
0,8·16 = 12,8 ⇒ $12_{10}$ = C

Operasjonen stoppes etter hvor mange siffer vi har tilgjengelig (også bestemt av nøyaktigheten). Svaret er her $0,675_{10} = ACCCCCC..._{16}$. Avrundet til 2 byte (16 bit) blir svaret $ACCD_{16}$.

## 2.10. BCD-kode

Ved koden binærkodet desimal, Binary Coded Desimal, BCD, kodes hvert desimale siffer for seg. Det betyr at for eksempel 81 kodes som 1000 0001, og ikke binært som $1010001_2$.

| Desimal | 8421 BCD | 2421 BCD | Excess-3 |
|---|---|---|---|
| 0 | 0000 | 0000 | 0011 |
| 1 | 0001 | 0001 | 0100 |
| 2 | 0010 | 0010 | 0101 |
| 3 | 0011 | 0011 | 0110 |
| 4 | 0100 | 0100 | 0111 |
| 5 | 0101 | 1011 | 1000 |
| 6 | 0110 | 1100 | 1001 |
| 7 | 0111 | 1101 | 1010 |
| 8 | 1000 | 1110 | 1011 |
| 9 | 1001 | 1111 | 1100 |

Ved BCD-kode benyttes 4 bit, og dermed brukes ikke binærtallene 1010, 1011, 1100, 1101, 1110 og 1111. Koden er dermed redundant.

Koden går også under navnet 8421 BCD-kode siden dette angir vekten til de forskjellige sifre ($8 = 2^3$, $4 = 2^2$, $2 = 2^1$, $1 = 2^0$). Grunnen til at koden brukes, er at vi med digitale klokker, termometre, måleinstrumenter og så videre bruker 7-segment skjerm (display) der vi ønsker presentasjonen gitt med desimale siffer.

Det finnes også en 2421 BCD-kode der vekten er $2 = 2^1$, $4 = 2^2$, $2 = 2^1$, $1 = 2^0$, se tabellen.

Koden Excess-3 framkommer ved å legge $3_{10} = 11_2$ til 8421-koden. Excess-3 benyttes ved BCD-aritmetikk.

## 2.11. Gray-kode

Gray-koden karakteriseres ved at kun ett bit skifter når tallverdien øker med 1. Denne koden hører til typen koder som kalles minimumskift-koder, se tabellen nedenfor.

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

Legg forøvrig merke til at med bare to bit, er tellesekvensen 00, 01, 11, 10, 00 etc. Med tre bit er tellesekvensen 000, 001, 011, 010, 110, 111, 101, 100, 000 etc.

Denne koden brukes for eksempel innen modulasjon (for data- og telekommunikasjons-systemer).

# Kapittel 3
# Logiske porter

## 3.1. Innledning

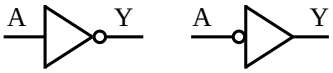
Digitale kretser er oppbygd av logiske byggeklosser som sies å være kombinatoriske og sekvensielle enheter. En kombinatorisk logisk enhet tar et sett av inngangsvariable og omformer til et sett av utgangsvariable, gitt av hvilken funksjon som skal utføres. Utgangene er funksjon kun av inngangene og foreligger straks (med en liten forsinkelse) på utgangen. Det er ingen tilbakekopling fra utgangen til inngangen for kombinatoriske funksjoner.

Innganger og utganger har to distinkte nivåer: høy og lav. Normalt vil verdier på omlag 0 V tolkes som lav, som logisk 0. Logisk 1, høyt nivå, kan sies å ligge på ca 3,3 V (eller nærmere 5 V). Imidlertid vil moderne raske digitale kretser operere med lavere spenningsnivåer (helt ned mot 1 V).

Her skal vi se på de minste byggeklossene som kalles porter (gates). Vi tar for oss symboler og funksjonsbeskrivelser. Til slutt ser vi på praktisk realisering av noen av portene.

## 3.2. Inverterer

Som nevnt i kapittel 1, vil en inverterer som påtrykkes et høyt signal på inngangen, produsere et lavt signal på utgangen og vice versa. Symbolet er vist til venstre i figur 3.1. Inverteringen er representert med en ring på utgangen. Til høyre i figuren er ringen flyttet til inngangen (notasjonen aktiv lav).




*Figur 3.1. Symbol for inverterer.*

I engelskspråklig litteratur kalles ofte invertererens funksjon for NOT. Vi kommer ikke til å benytte denne betegnelsen her.

Algebraisk kan vi uttrykke funksjonen som:

$$Y = \overline{A} \tag{3.1}$$

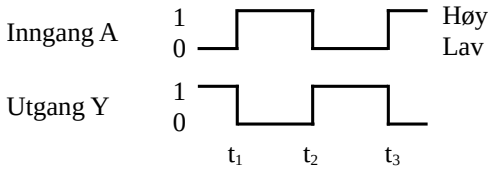
Vi kan sette opp en sannhetstabell som vist i tabell 3.1 for å beskrive invertereren. Vi bruker 0 for lavt og 1 for høyt nivå.

| Inngang | Utgang |
|---|---|
| A | Y |
| 0 | 1 |
| 1 | 0 |

*Tabell 3.1. Sannhetstabell for inverterer.*

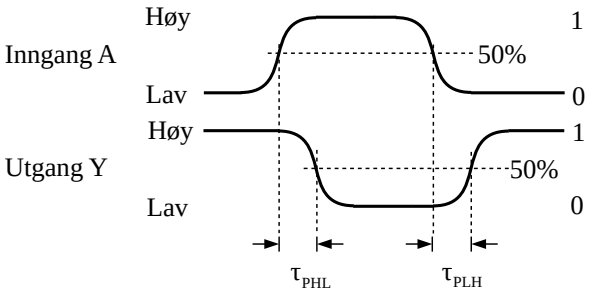
I sannhetstabellen tas med alle inngangskombinasjoner. For invertereren er det kun to mulige kombinasjoner.

Det kan også være nyttig å tegne et tidsdiagram for å se på forløpet for inn- og ut-gangssignalene. Dette er gjort i figur 3.2. Ved tidspunktet $t_1$ ser vi at inngangen går høy, til logisk 1. Da går utgangen lav, til logisk 0. Ved tidspunktet $t_2$ skjer det motsatte.




*Figur 3.2. Tidsdiagram for inverterer.*

I virkeligheten vil det gå en viss tid fra inngangen endrer seg til utgangen gjør det. Dette er ganske naturlig: Endringen kan ikke skje momentant. Dette er illustrert i figur 3.3.

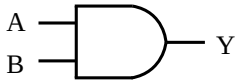



*Figur 3.3. Transportforsinkelse i inverterer.*

Transportforsinkelsen (’Propagation Delay’) $\tau_{PHL}$ og $\tau_{PLH}$ betegner tiden fra inngangen til utgangen når 50 % fra henholdsvis høy (H) til lav (L) og lav til høy. For CMOS er det ikke uvanlig å anta at $\tau_{PHL}$ og $\tau_{PLH}$ er like lange.

## 3.3. OG-port

Logisk OG er den norske betegnelsen som på engelsk heter AND. Symbolet for en OG-port med to innganger er vist i figur 3.4.




*Figur 3.4. Symbol for OG-port.*

OG-porten kan ha flere enn to innganger, men har alltid bare en utgang. Algebraisk kan vi uttrykke funksjonen som:

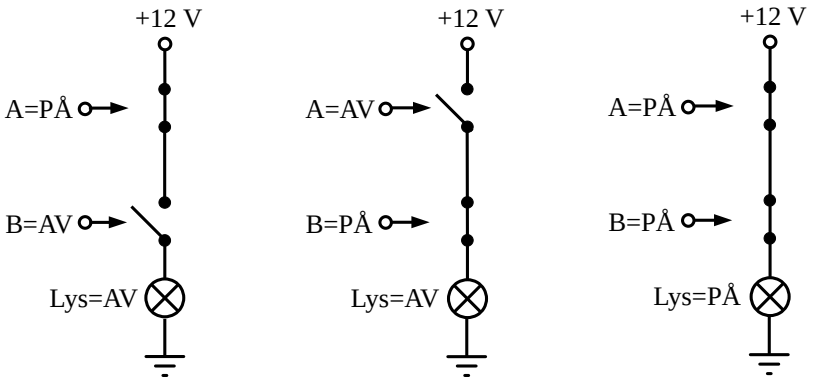
$$Y = A \cdot B \tag{3.2}$$

Utgangen Y er høy, logisk 1, bare når alle inngangene er høye, logisk 1. Dersom en eller flere av inngangene er logisk 0, er utgangen logisk 0. Dette er vist i sannhetstabellen i tabell 3.2. Siden vi her har to innganger, vil vi ha $2^2 = 4$ kombinasjoner som vist i tabellen.

| Innganger | | Utgang |
|---|---|---|
| A | B | Y |
| 0 | 0 | 0 |
| 0 | 1 | 0 |
| 1 | 0 | 0 |
| 1 | 1 | 1 |

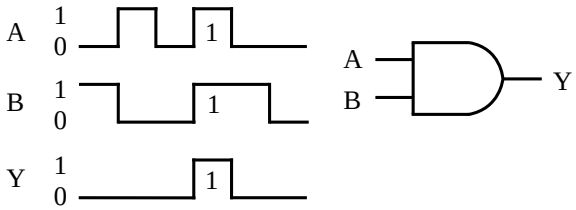
*Tabell 3.2. Sannhetstabell for OG-port.*

Funksjonen kan illustreres som i figur 3.5. Vi tenker oss at vi har to brytere, A og B. Bryter på svarer til logisk 1 og bryter av svarer til logisk 0. Som det ses av figuren må bryterne koples i serie for at vi skal ha en OG-funksjon. Bare i det tilfellet at både bryter A er på OG bryter B er på vil vi få lampa til å lyse. Dersom en eller begge bryterne er av, vil lampa være slukket.




*Figur 3.5. OG-funksjonen illustrert ved brytere og lampe.*

Som for invertereren kan det være nyttig å tegne et tidsdiagram for å se på forløpet for inn- og ut-ganger. Dette er gjort i figur 3.6. Vi ser at utgangen bare er høy, logisk 1, når begge inngangene er høye.

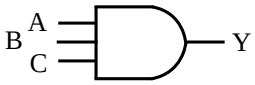



*Figur 3.6. Tidsdiagram for OG-port.*

Som for invertereren vil det gå en viss tid fra inngangen endrer seg til utgangen gjør det. Også her opereres med transportforsinkelse $\tau_{PHL}$ og $\tau_{PLH}$ når utgangen går fra henholdsvis høy til lav og fra lav til høy.

En OG-port med tre innganger vil ha $2^3 = 8$ forskjellige inngangskombinasjoner, se figur 3.7. Men det er fortsatt slik at alle inngangene må være logisk 1 for at utgangen skal være lik 1. Alle de andre inngangskombinasjonene gir logisk 0 på utgangen av porten.

| A | B | C | Y |
|---|---|---|---|
| 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 0 |
| 0 | 1 | 0 | 0 |
| 0 | 1 | 1 | 0 |
| 1 | 0 | 0 | 0 |
| 1 | 0 | 1 | 0 |
| 1 | 1 | 0 | 0 |
| 1 | 1 | 1 | 1 |

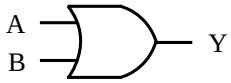



*Figur 3.7. Sannhetstabell og symbol for OG-port med 3 innganger.*

OG-port med flere innganger enn tre er også mulig. Imidlertid er mer enn 8 innganger uvanlig å se. Dette skyldes at antall innganger som ses fra den drivende kretsen (fan-in) øker med to for hver ekstra inngang og transportforsinkelsen øker.

## 3.4. ELLER-port

Logisk ELLER er den norske betegnelsen som på engelsk heter OR. Symbolet for en ELLER-port med to innganger er vist i figur 3.8.




*Figur 3.8. Symbol for ELLER-port.*

ELLER-porten kan ha flere enn to innganger, men har alltid bare en utgang. Algebraisk kan vi uttrykke funksjonen som:

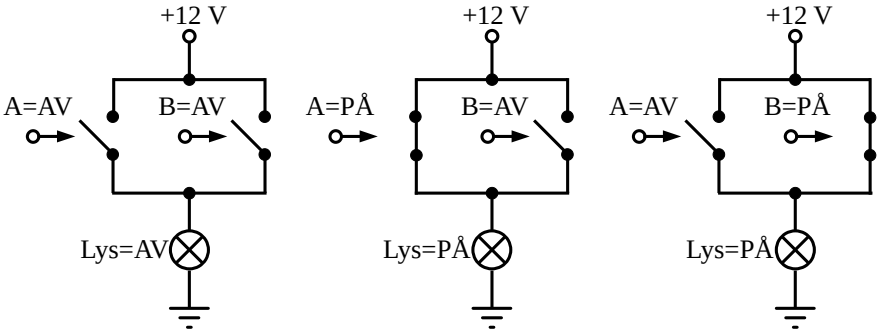
$$Y = A + B \tag{3.3}$$

Utgangen Y er høy, logisk 1, når minst en av inngangene er høy, logisk 1. Dersom alle inngangene er logisk 0, er utgangen logisk 0. Dette er vist i sannhetstabellen i tabell 3.3. Siden vi her har to innganger, vil vi igjen ha $2^2 = 4$ kombinasjoner som vist i tabellen.

| Innganger | | Utgang |
|---|---|---|
| A | B | Y |
| 0 | 0 | 0 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 1 |

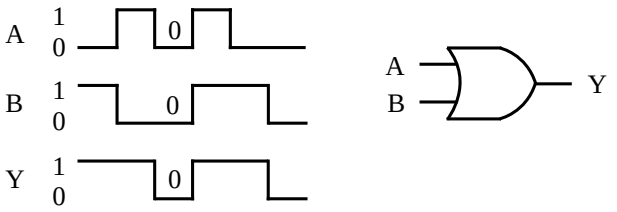
*Tabell 3.3. Sannhetstabell for ELLER-port.*

Funksjonen kan illustreres som i figur 3.9. Vi tenker oss igjen at vi har to brytere, A og B, der bryteren på svarer til logisk 1 og bryteren av svarer til logisk 0. Som det ses av figuren må bryterne koples i parallell for at vi skal ha en ELLER-funksjon. Bare i det tilfellet at begge bryterne er av vil lampa være slukket. I alle de tre andre tilfellene vil lampa lyse.




*Figur 3.9. Brytere og lampe for å illustrere funksjonen til ELLER-porten.*

Et tidsdiagram som illustrerer forløpet for inn- og ut-ganger, er vist i figur 3.10. Vi ser at utgangen bare er lav, logisk 0, når begge inngangene er lave.

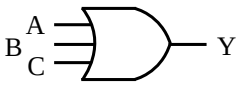



*Figur 3.10. Tidsdiagram for ELLER-port.*

Også her vil det i virkeligheten gå en viss tid før en endring på inngangen forplanter seg til utgangen. Det opereres følgelig også her med transportforsinkelse når utgangen går fra høy til lav og fra lav til høy.

En ELLER-port med tre innganger vil ha $2^3 = 8$ forskjellige inngangskombinasjoner, se figur 3.11. Men det er fortsatt slik at bare alle inngangene lik logisk 0 fører til at utgangen er lik logisk 0. Alle de andre inngangskombinasjonene gir logisk 1 på utgangen av porten.

| A | B | C | Y |
|---|---|---|---|
| 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 1 |
| 0 | 1 | 0 | 1 |
| 0 | 1 | 1 | 1 |
| 1 | 0 | 0 | 1 |
| 1 | 0 | 1 | 1 |
| 1 | 1 | 0 | 1 |
| 1 | 1 | 1 | 1 |

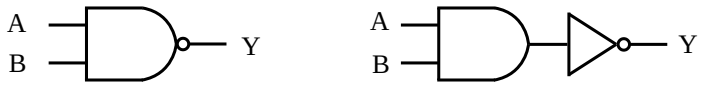



*Figur 3.11. Sannhetstabell og symbol for ELLER-port med 3 innganger.*

ELLER-porter med flere innganger enn tre er også mulig. Imidlertid er det som for OG-porter at mer enn 8 innganger er uvanlig å se. Igjen skyldes dette at antall innganger som ses fra den drivende kretsen øker med to for hver ekstra inngang og transportforsinkelsen øker.

## 3.5. NAND-port

Vi bruker her den engelske betegnelsen NAND for en OG-port med invertering istedenfor den norske betegnelsen NOG. Symbolet er vist i figur 3.12a. I figur 3.12b er vist den funksjonelle ekvivalenten, som består av en OG-port fulgt av en inverterer.




*(a)* *(b)*

*Figur 3.12. NAND-port (a) og ekvivalent (b).*

NAND-porten kan brukes som en universell port. Med det menes at den kan brukes som inverterer, som OG-port og som ELLER-port når henholdsvis en, to eller tre slike porter settes sammen.

NAND-porten kan ha flere enn to innganger, men har alltid bare en utgang. Algebraisk kan vi uttrykke funksjonen som:

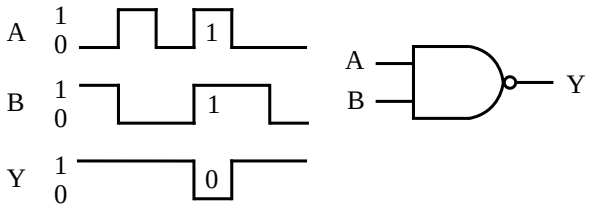
$$Y = \overline{A \cdot B} \tag{3.4}$$

Siden en NAND-port er en OG-port med invertering, er utgangen Y lav, logisk 0, bare når alle inngangene er høye, logisk 1. Dersom en eller flere av inngangene er logisk 0, er utgangen logisk 1. Dette er vist i sannhetstabellen i tabell 3.4. Siden vi her har to innganger, vil vi ha $2^2 = 4$ kombinasjoner som vist i tabellen.

| Innganger | | Utgang |
|---|---|---|
| A | B | Y |
| 0 | 0 | 1 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 0 |

*Tabell 3.4. Sannhetstabell for NAND-port.*

Et tidsdiagram som viser forløpet for inn- og ut-gangene er vist i figur 3.13. Vi ser at utgangen bare er lav, logisk 0, når begge inngangene er høye.

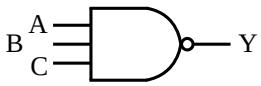



*Figur 3.13. Tidsdiagram for NAND-port.*

Også her has transportforsinkelse. Det spesielle for CMOS er at OG-porten realiseres som en NAND-port med etterfølgende invertering. Følgelig har NAND-porten kortere transportforsinkelse enn OG-porten.

En NAND-port med tre innganger vil ha $2^3 = 8$ forskjellige inngangskombinasjoner, se figur 3.14. Men det er fortsatt slik at alle inngangene må være logisk 1 for at utgangen skal være lik logisk 0. Alle de andre inngangskombinasjonene gir logisk 1 på utgangen av porten.

| A | B | C | Y |
|---|---|---|---|
| 0 | 0 | 0 | 1 |
| 0 | 0 | 1 | 1 |
| 0 | 1 | 0 | 1 |
| 0 | 1 | 1 | 1 |
| 1 | 0 | 0 | 1 |
| 1 | 0 | 1 | 1 |
| 1 | 1 | 0 | 1 |
| 1 | 1 | 1 | 0 |

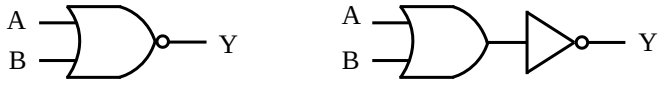



*Figur 3.14. Sannhetstabell og symbol for NAND-port med 3 innganger.*

Det finnes også NAND-porter med flere enn tre innganger. Imidlertid er mer enn 8 innganger uvanlig å se av samme grunn som nevnt tidligere.

## 3.6. NOR-port

Vi bruker her den engelske betegnelsen NOR for en ELLER-port med invertering istedenfor den norske betegnelsen NELLER. Symbolet er vist i figur 3.15a. I figur 3.15b er vist den funksjonelle ekvivalenten, som består av en ELLER-port fulgt av en inverterer.




*(a)* *(b)*

*Figur 3.15. NOR-port (a) og ekvivalent (b).*

NOR-porten kan som NAND-porten brukes som en universell port. Med det menes at den kan brukes som inverterer, som ELLER-port og som OG-port når henholdsvis en, to eller tre slike porter settes sammen.

NOR-porten kan ha flere enn to innganger, men har alltid bare en utgang. Algebraisk kan vi uttrykke funksjonen som:

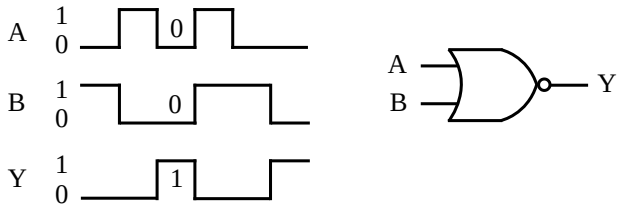
$$Y = \overline{A + B} \tag{3.5}$$

Siden en NOR-port er en ELLER-port med invertering, er utgangen Y lav, logisk 0, når minst en av inngangene er høy, logisk 1. Dersom alle inngangene er logisk 0, er utgangen logisk 1. Dette er vist i sannhetstabellen i tabell 3.5. Siden vi her har to innganger, vil vi ha $2^2 = 4$ kombinasjoner som vist i tabellen.

| Innganger | | Utgang |
|---|---|---|
| A | B | Y |
| 0 | 0 | 1 |
| 0 | 1 | 0 |
| 1 | 0 | 0 |
| 1 | 1 | 0 |

*Tabell 3.5. Sannhetstabell for NOR-port.*

Et tidsdiagram som viser forløpet for inn- og ut-gangene er vist i figur 3.16. Vi ser at utgangen bare er høy, logisk 1, når begge inngangene er lave.

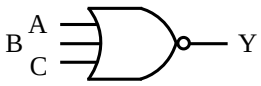



*Figur 3.16. Tidsdiagram for NOR-port.*

Også her has transportforsinkelse. Som for NAND- og OG-porten er det spesielt for CMOS at ELLER-porten realiseres som en NOR-port med etterfølgende invertering. Følgelig har NOR-porten kortere transportforsinkelse enn ELLER-porten.

En NOR-port med tre innganger vil ha $2^3 = 8$ forskjellige inngangskombinasjoner, se figur 3.17. Men det er fortsatt slik at med alle inngangene lik logisk 0, vil utgangen være lik logisk 1. Alle de andre inngangskombinasjonene gir logisk 0 på utgangen av porten.

| A | B | C | Y |
|---|---|---|---|
| 0 | 0 | 0 | 1 |
| 0 | 0 | 1 | 0 |
| 0 | 1 | 0 | 0 |
| 0 | 1 | 1 | 0 |
| 1 | 0 | 0 | 0 |
| 1 | 0 | 1 | 0 |
| 1 | 1 | 0 | 0 |
| 1 | 1 | 1 | 0 |




*Figur 3.17. Sannhetstabell og symbol for NOR-port med 3 innganger.*

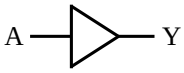
Det finnes også NOR-porter med flere enn tre innganger. Imidlertid er mer enn 8 innganger uvanlig å se av samme grunn som nevnt tidligere.

## 3.7. Buffer

Vi benytter den engelske betegnelsen på denne porten som kopierer inngangen til utgangen (med en viss tidsforsinkelse). Dette kan høres ut som bortkastet, men denne porten kan i praksis ha flere anvendelser.

### *Enkel buffer*

Symbolet for en enkel buffer er vist i figur 3.18. Denne har som oppgave å være forsterker slik at flere logiske porter kan drives av et enkelt signal (på inngangen av bufferen).




*Figur 3.18. Symbol for buffer.*

Algebraisk kan vi uttrykke funksjonen som:

$$Y = A \tag{3.6}$$

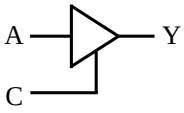
Vi kan sette opp en sannhetstabell som vist i tabell 3.6 for å beskrive bufferen. I sannhetstabellen tas med alle inngangskombinasjoner. For bufferen er det kun to mulige kombinasjoner.

| Inngang | Utgang |
|---|---|
| A | Y |
| 0 | 0 |
| 1 | 1 |

*Tabell 3.6. Sannhetstabell for buffer.*

### *3-nivå (3-state) buffer*

Dette er en buffer der utgangen foruten å kunne være lav (logisk 0) og høy (logisk 1) også kan frakoples, det vil si kan representere brudd (ha meget høy impedans). Symbolet for en slik buffer er vist i figur 3.19.




*Figur 3.19. Symbol for 3-nivå buffer.*

Vi kan sette opp en sannhetstabell som vist i tabell 3.7 for å beskrive bufferen. Her betegner Z at utgangen er frakoplet (har høy impedans).

| Inngang | | Utgang |
|---|---|---|
| C | A | Y |
| 0 | 0 | Z |
| 0 | 1 | Z |
| 1 | 0 | 0 |
| 1 | 1 | 1 |

*Tabell 3.7. Sannhetstabell for 3-nivå buffer.*

Inngangen C = 0 benyttes her for å frakople bufferen. Navnet 3-nivå henspeiler på at utgangen kan innta tre nivåer, tilstander. Denne typen buffer finnes overalt i buss-systemer og kan være innebygd i vipper, registre, minner og så videre. Det medfører at flere utganger kan være koplet til samme leder i en buss, se forøvrig kapittel 5.3.

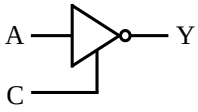
Alle de utgangene som ved et gitt tidspunkt ikke skal tilkoples lederen, legges høyimpedante, mens den ønskede utgangen legges til lederen. En buss betegner en samling av flere ledere som har samme funksjon.

Algebraisk kan vi uttrykke funksjonen som:

$$Y = A \cdot C + \overline{C} \cdot Z \tag{3.7}$$

Legg merke til at vi her har et OG-ELLER-uttrykk (der Y = A dersom C = 1 mens Y = Z dersom C = 0), se forøvrig kapittel 5.2.

Det kan også lages 3-nivå buffer som er inverterende. Symbolet for en slik buffer er vist i figur 3.20. Også styresignalet kan forøvrig lages inverterende.




*Figur 3.20. Symbol for inverterende 3-nivå buffer.*

Vi kan sette opp en sannhetstabell for den inverterende bufferen i figur 3.20 som vist i tabell 3.8. Her betegner igjen Z at utgangen er frakoplet (har høy impedans).

| Inngang | | Utgang |
|---|---|---|
| C | A | Y |
| 0 | 0 | Z |
| 0 | 1 | Z |
| 1 | 0 | 1 |
| 1 | 1 | 0 |

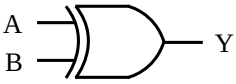
*Tabell 3.8. Sannhetstabell for inverterende 3-nivå buffer.*

Algebraisk kan vi da uttrykke funksjonen som:

$$Y = \overline{A} \cdot C + \overline{C} \cdot Z \tag{3.8}$$

## 3.8. Eksklusiv ELLER-port

Funksjonen Eksklusiv ELLER (Exclusive OR) kan realiseres ved hjelp av portene allerede beskrevet, men siden den kan anvendes i flere sammenhenger, finnes den som en egen port. Symbolet for denne er vist i figur 3.21.




*Figur 3.21. Symbol for Eksklusiv ELLER-port.*

Denne porten finnes normalt bare med to innganger. Funksjonen kan uttrykkes som:

$$Y = \overline{A} \cdot B + A \cdot \overline{B} = A \oplus B \tag{3.9}$$

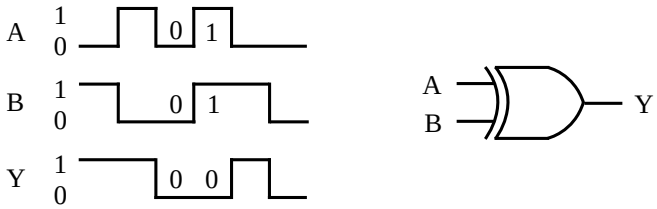
Legg merke til at denne funksjonen har fått sin egen operator. Eksempel på realisering av Eksklusiv ELLER-porten er vist i kapittel 5.

Utgangen Y er høy, logisk 1, når kun én av inngangene er høy, logisk 1. Dersom begge inngangene er logisk 0, er utgangen logisk 0. Utgangen er også logisk 0 når begge inngangene er lik logisk 1. Det siste skiller den fra ELLER-porten, derav navnet Eksklusiv ELLER. Dette er vist i sannhetstabellen i tabell 3.9. Siden vi her har to innganger, vil vi igjen ha $2^2 = 4$ kombinasjoner som vist i tabellen.

| Innganger | | Utgang |
|---|---|---|
| A | B | Y |
| 0 | 0 | 0 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 0 |

*Tabell 3.9. Sannhetstabell for Eksklusiv ELLER-port.*

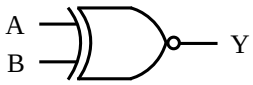
Et tidsdiagram som viser forløpet for inn- og ut-ganger er vist i figur 3.22. Vi ser at utgangen er lav, logisk 0, når begge inngangene samtidig er lik logisk 0 eller 1. Transportforsinkelsen kan være større for Eksklusiv ELLER-porten enn for ELLER-porten med to innganger.




*Figur 3.22. Tidsdiagram for Eksklusiv ELLER-port.*

## 3.9. Eksklusiv NOR-port

Funksjonen Eksklusiv NOR (Exclusive NOR) kan også realiseres ved hjelp av portene allerede beskrevet, men siden den som Eksklusiv ELLER kan anvendes i flere sammenhenger, finnes den realisert som egen port. Funksjonsmessig er dette en invertering av Eksklusiv ELLER-porten. Symbolet for Eksklusiv NOR-porten er vist i figur 3.23.




*Figur 3.23. Symbol for Eksklusiv NOR-port.*

Denne porten finnes normalt bare med to innganger. Algebraisk kan funksjonen uttrykkes som:

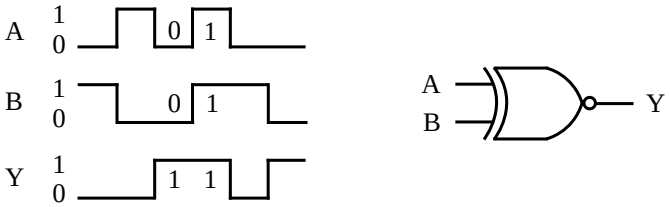
$$Y = A \cdot B + \overline{A} \cdot \overline{B} = \overline{A \oplus B} \tag{3.10}$$

Utgangen Y er høy, logisk 1, når begge inngangene er logisk 0 eller 1 samtidig. Derfor kan denne porten kalles binær komparator. Dette kan ses i sannhetstabellen i tabell 3.10.

| Innganger | | Utgang |
|---|---|---|
| A | B | Y |
| 0 | 0 | 1 |
| 0 | 1 | 0 |
| 1 | 0 | 0 |
| 1 | 1 | 1 |

*Tabell 3.10. Sannhetstabell for Eksklusiv NOR-port.*

Et tidsdiagram som viser forløpet for inn- og ut-ganger er vist i figur 3.24. Vi ser at utgangen er høy, logisk 1, når begge inngangene er like.




*Figur 3.24. Tidsdiagram for Eksklusiv NOR-port.*

Transportforsinkelsen kan være større for Eksklusiv NOR-porten enn for NOR-porten med to innganger. Eksempel på realisering av Eksklusiv NOR-porten er vist i kapittel 5. Siden Eksklusiv NOR-porten er den inverterte av Eksklusiv ELLER-porten, ses følgelig:

$$\overline{\overline{A} \cdot B + A \cdot \overline{B}} = \overline{A \oplus B} = A \cdot B + \overline{A} \cdot \overline{B} \tag{3.11}$$

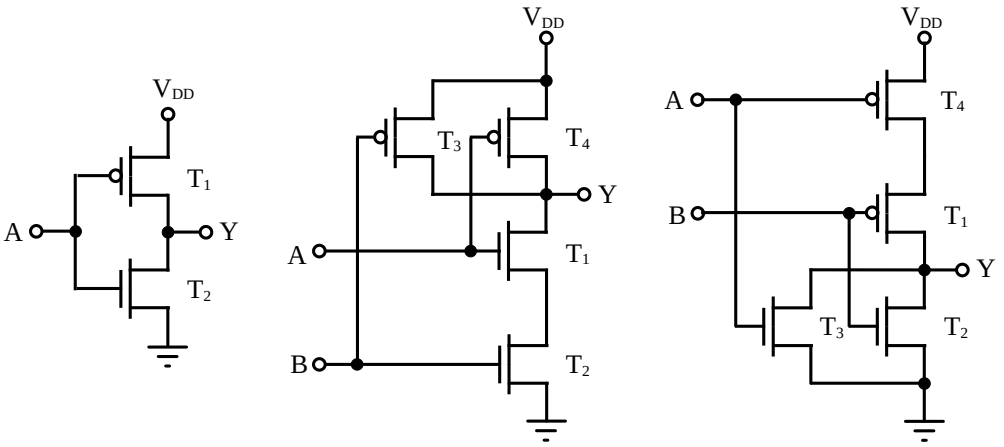
# 3.10. CMOS-Realisering

## 3.10.1. Innledning

Vi skal her kort se på hvordan noen porter kan realiseres i CMOS-teknologi. Denne teknologien baserer seg på bruk av N- og P-kanal Anrikning (Enhancement) MOSFET-transistorer, derav navnet CMOS, Complementary MOSFET. Teknologien er dominerende innen digital elektronikk og kjennetegnes ved høy svitsjehastighet og forholdsvis lavt effektforbruk. Effektforbruket er proporsjonalt med svitsjehastigheten, slik at det er meget lavt for moderate hastigheter.

Den grunnleggende kretsen i CMOS er det komplementære trinnet som realiserer en inverterer, vist på nytt i figur 3.25a. Den øverste transistoren er P-kanal, PMOS, mens den nederste er N-kanal, NMOS. Det er benyttet symbolene vist i kapittel 1. Virkemåten for invertereren ble beskrevet i kapittel 1.

Forsyningsspenningen er $V_{DD}$, der typiske verdier kan ligge på 3,3 V eller 5 V. Noen storskalakretser kan ha en forsyningsspenning ned mot 1 V. Jordsymbolet angir 0 V (ofte kalt $V_{SS}$).




*(a)* *(b)* *(c)*

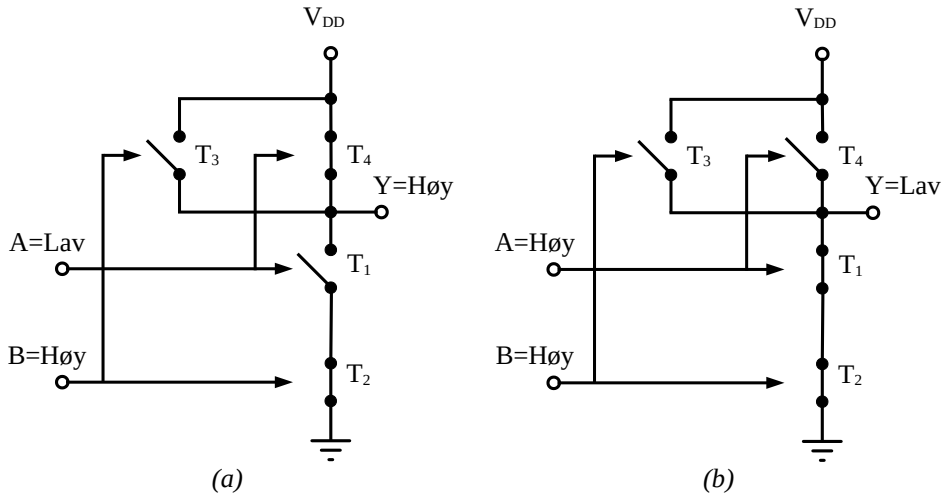
*Figur 3.25. CMOS: Inverterer (a), NAND-port (b), NOR-port (c).*

## 3.10.2. NAND-port

I figur 3.25b er vist en NAND-port. Her benyttes to PMOS- og to NMOS-transistorer. NMOS-transistorene er koplet i serie og PMOS-transistorene er koplet i parallell.

For å forstå virkemåten for NAND-porten, kan denne illustreres med styrte brytere istedenfor transistorer som vist i figur 3.26. I figur 3.26a er inngangen A lav (logisk 0) mens inngangen B er høy (logisk 1). Dette slår på transistoren $T_4$ slik at $V_{DD}$ føres til utgangen Y, som dermed blir høy (logisk 1). Legg merke til at $T_1$ er av slik at vi ikke får noen forbindelse til 0 V.

I figur 3.26b er begge inngangene høye (logisk 1). Dette slår på transistorene $T_1$ og $T_2$ slik at 0 V føres til utgangen Y, som dermed blir lav (logisk 0). Legg merke til at både $T_3$ og $T_4$ er av slik at vi ikke får noen forbindelse til $V_{DD}$. Siden $T_1$ og $T_2$ er koplet i serie, er A = B = Høy (logisk 1) det eneste tilfellet der utgangen blir lav (logisk 0). Dette stemmer med sannhetstabellen for NAND-porten.



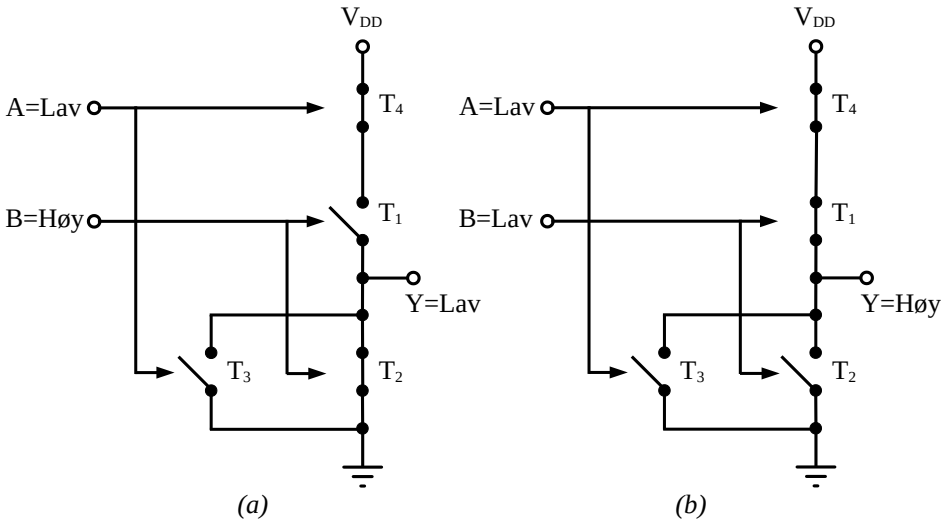

*Figur 3.26. NAND-port med utgang høy (a) og lav (b).*

## 3.10.3. NOR-port

I figur 3.25c er vist en NOR-port. Her benyttes igjen to PMOS- og to NMOS-transistorer. NMOS-transistorene er nå koplet i parallell og PMOS-transistorene er koplet i serie, altså motsatt av koplingen for NAND-porten.

For å forstå virkemåten for NOR-porten, kan denne illustreres med styrte brytere istedenfor transistorer som vist i figur 3.27. I figur 3.27a er inngangen A lav (logisk 0) mens inngangen B er høy (logisk 1). Dette slår på transistoren $T_2$ slik at 0 V føres til utgangen Y, som dermed blir lav (logisk 0). Legg merke til at $T_1$ er av slik at vi ikke får noen forbindelse til $V_{DD}$.

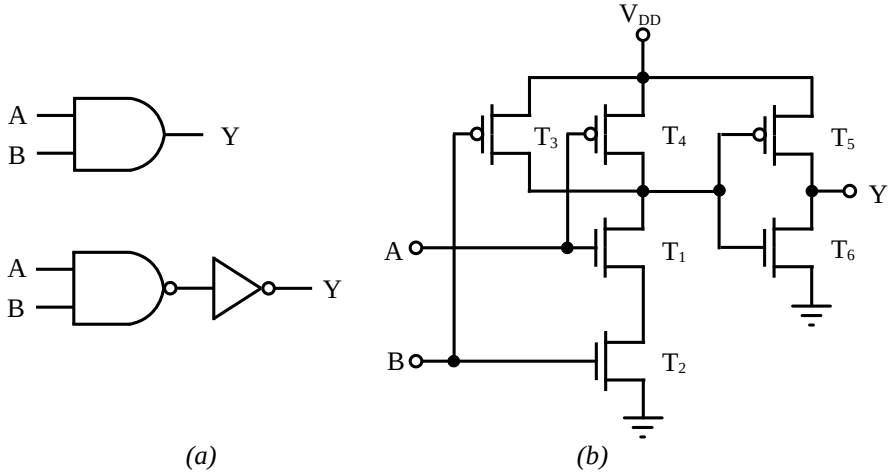
I figur 3.27b er begge inngangene lave (logisk 0). Dette slår på transistorene $T_1$ og $T_4$ slik at $V_{DD}$ føres til utgangen Y, som dermed blir høy (logisk 1). Legg merke til at både $T_2$ og $T_3$ er av slik at vi ikke får noen forbindelse til 0 V. Siden $T_1$ og $T_4$ er koplet i serie, er A = B = Lav (logisk 0) det eneste tilfellet der utgangen blir høy (logisk 1). Dette stemmer med sannhetstabellen for NOR-porten.




*Figur 3.27. NOR-port med utgang lav (a) og høy (b).*

## 3.10.4. OG-port

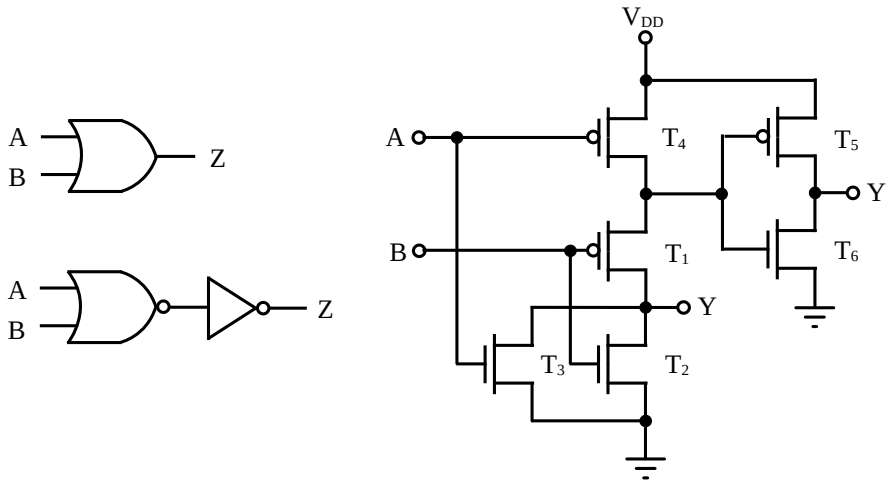
I CMOS lages OG- porten enklest ved først å realisere NAND-porten og så invertere utgangen. I figur 3.28a er vist den logiske ekvivalenten, mens det i figur 3.28b er vist en mulig praktisk kretsrealisering. Transistorene $T_5$ og $T_6$ er invertereren fra figur 3.25a mens transistorene $T_1$ - $T_4$ utgjør NAND-porten fra figur 3.25b. Det skjønnes fra dette at tiden det tar fra en endring på inngangen til en endring på utgangen (transport-forsinkelsen) er kortere med NAND-porten enn med OG-porten.




*Figur 3.28. Realisering av OG-port.*

## 3.10.5. ELLER-port

Tilsvarende lages ELLER-porten i CMOS ved først å realisere NOR-porten for så å invertere utgangen. I figur 3.29a er vist den logiske ekvivalenten, mens det i figur 3.29b er vist en mulig praktisk kretsrealisering. Transistorene $T_5$ og $T_6$ er invertereren fra figur 3.25a mens transistorene $T_1$ - $T_4$ utgjør NOR-porten fra figur 3.25c.




*(a)* *(b)*

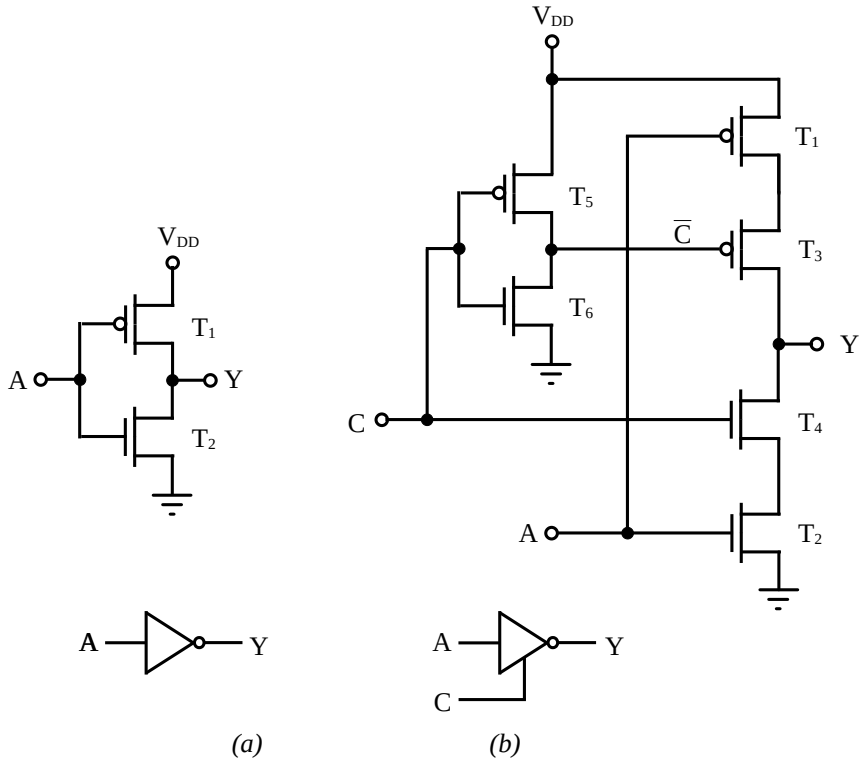
*Figur 3.29. Realisering av ELLER-port.*

Som for OG-porten og NAND-porten vil det være slik at transport-forsinkelsen blir større med ELLER-porten enn med NOR-porten.

## 3.10.6. 3-nivå Buffer

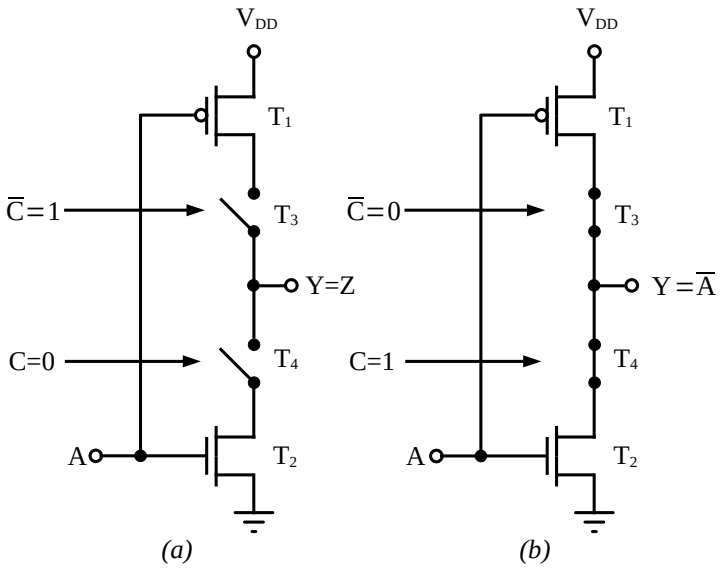
I figur 3.30 er vist en inverterer og en inverterende 3-nivå buffer. For realiseringen av sistnevnte i figur 3.30b sørger transistorene $T_3$ og $T_4$ for å frakople utgangen når styreinngangen C er lav (logisk 0). Legg merke til at det benyttes en inverterer (transistorene $T_5$ og $T_6$) for å invertere styresignalet C til transistoren $T_3$.

Dersom vi ønsker en ikke-inverterende 3-nivå buffer kan vi kople en inverterer før denne (kaskadekople kretsene i figur 3.30a og 3.30b).



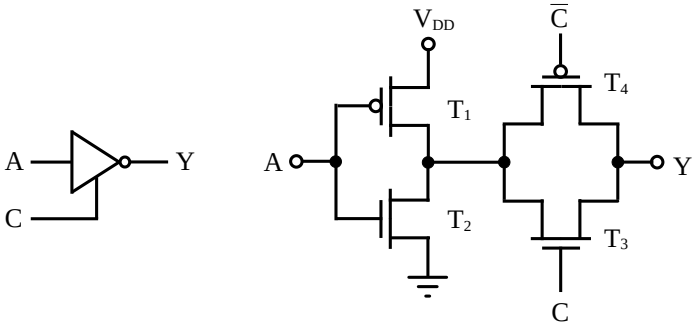

*Figur 3.30. Inverterer (a) og 3-nivå inverterende buffer (b).*

Virkemåten for 3-nivå bufferen kan forklares ut fra figur 3.31. I figur 3.31a er vist situasjonen når C = 0. Det ses at transistorene $T_3$ og $T_4$ er av, og da er utgangen frakoplet resten av kretsen. I figur 3.31b er C = 1 og transistorene $T_3$ og $T_4$ er begge på. Dermed har vi en vanlig inverterer.




*Figur 3.31. 3-nivå buffer for C=0 (a) og C=1 (b).*

En alternativ realisering av 3-nivå bufferen er vist i figur 3.32. Her brukes en transmisjonsport (’Transmission Gate’) på utgangen av en inverterer. Transmisjonsporten fungerer slik at når C = 0, vil begge transistorene $T_3$ og $T_4$ være av. Det betyr at utgangen Y = Z (høy impedans). Når C = 1, vil transistorene $T_3$ og $T_4$ være på slik at $Y = \overline{A}$.




*Figur 3.32. 3-nivå buffer med transmisjonsport.*

Vanligvis koples substratet på $T_4$ til $V_{DD}$ mens substratet på $T_3$ koples til 0 V. At det er to komplementære transistorer i transmisjonsporten, sikrer derfor god ledning selv om spenningen på inngangen ikke er helt ned mot 0 V eller opp mot $V_{DD}$.

# Kapittel 4
# Boole’sk algebra

## 4.1. Innledning

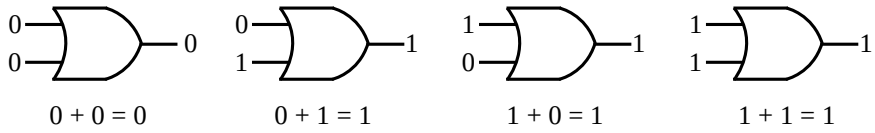
Det var så langt tilbake som i 1854 at George Boole publiserte det som ble kalt logisk algebra. Men det var ikke før i 1938 at arbeidet fikk praktisk betydning da Claude Shannon tok teorien i bruk for logiske kretser.

I dag er boole'sk algebra grunnleggende for analyse og konstruksjon av kombinatoriske og sekvensielle kretser.

Innen boole'sk algebra kan en variabel ha verdien 0 eller 1. Komplementet er det inverse, slik at en variabel A har komplementet $\overline{A}$. Hvis A = 1, er $\overline{A}$ = 0. Og hvis A = 0, er $\overline{A}$ = 1. En variabel og dens komplement kan også kalles litteral.

## 4.2. Sum og produkt

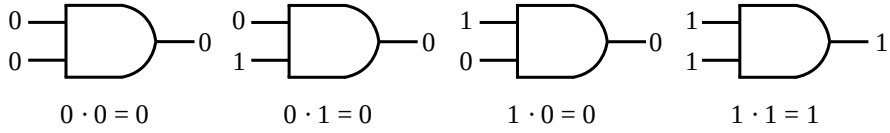
Boole'sk addisjon er det samme som å utføre en ELLER-operasjon og kan illustreres som vist i figur 4.1 med ELLER-porter.




*Figur 4.1. Boole'sk sum.*

En boole'sk sum er med andre ord lik 0 bare når alle variable er lik 0, ellers er den lik 1. En sum kan generelt skrives A + B, $\overline{A}$ + B, A + B + C etc.

Boole'sk multiplikasjon er det samme som å utføre en OG-operasjon og kan illustreres som vist i figur 4.2 med OG-porter.




*Figur 4.2. Boole'sk produkt.*

Et produkt er med andre ord lik 1 bare når alle variable er lik 1, ellers er den lik 0. En boole'sk multiplikasjon kan generelt skrives som produktene $A \cdot B$, $\overline{A} \cdot B$, $A \cdot B \cdot C$ etc.

# 4.3. Lover i boole'sk algebra

## 4.3.1. Kommutative lov

Den kommutative lov for boole'sk addisjon er:

$$A + B = B + A \tag{4.1}$$

Den kommutative lov for boole'sk multiplikasjon er:

$$A \cdot B = B \cdot A \tag{4.2}$$

Lovene er i tråd med vanlig algebra og er selvforklarende.

## 4.3.2. Assosiative lov

Den assosiative lov for boole'sk addisjon (med tre variable) er:

$$A + (B + C) = (A + B) + C \tag{4.3}$$

Den assosiative lov for boole'sk multiplikasjon er:

$$A \cdot (B \cdot C) = (A \cdot B) \cdot C \tag{4.4}$$

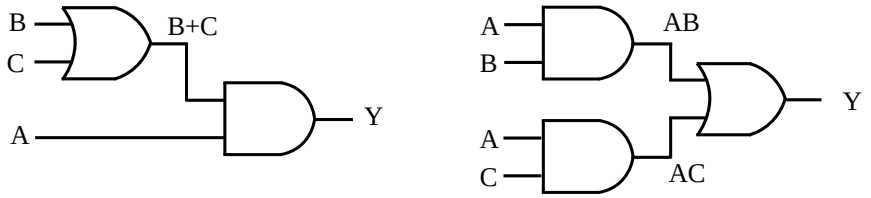
Også disse lovene er i tråd med vanlig algebra og er selvforklarende.

## 4.3.3. Distributive lov

Den distributive lov for boole'sk addisjon (med tre variable) er:

$$A \cdot (B + C) = AB + AC \tag{4.5}$$

Denne loven er også i tråd med vanlig algebra og er illustrert i figur 4.3.



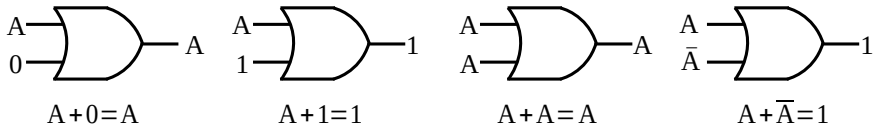

*Figur 4.3. Den distributive lov.*

## 4.3.4. Regler for Boole’sk algebra

Tabell 4.1 viser de grunnleggende reglene for boole’sk algebra.

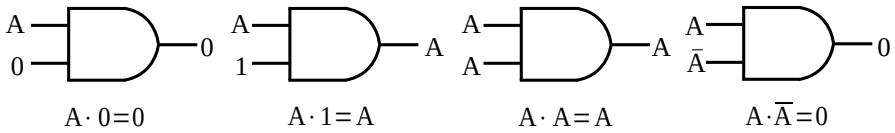
| | | | | |
|---|---|---|---|---|
| 1. | $A + 0 = A$ | 7. | $A \cdot A = A$ | |
| 2. | $A + 1 = 1$ | 8. | $A \cdot \overline{A} = 0$ | |
| 3. | $A + A = A$ | 9. | $\overline{\overline{A}} = A$ | *Tabell 4.1* |
| 4. | $A + \overline{A} = 1$ | 10. | $A + AB = A$ | *Boole’sk* |
| 5. | $A \cdot 0 = 0$ | 11. | $A + \overline{A}B = A + B$ | *algebra* |
| 6. | $A \cdot 1 = A$ | 12. | $(A + B)(A + C) = A + BC$ | |

I figur 4.4 er illustrert reglene 1-4 med ELLER-porter. Reglene verifiseres enkelt ved å la den variable innta verdien 0 eller 1.



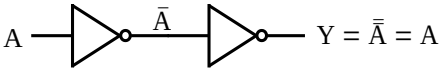

*Figur 4.4. Boole'ske regler 1-4 illustrert med ELLER-porter.*

I figur er illustrert reglene 5-8 med OG-porter. Også her kan reglene verifiseres enkelt ved å la den variable verdien 0 eller 1.



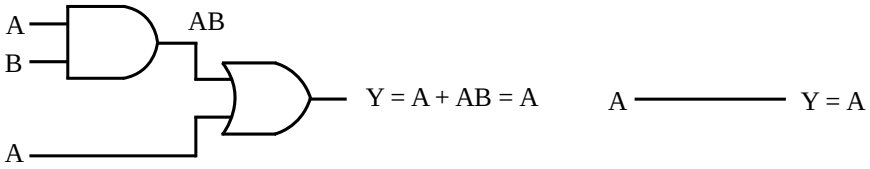

*Figur 4.5. Boole'ske regler 5-8 illustrert med OG-porter.*

I figur 4.6 er illustrert regel 9 med inverterere. Siden den inverterte av 0 er lik 1 og den inverterte av 1 er lik 0, må en dobbelinvertering gi oss den opprinnelige variable.



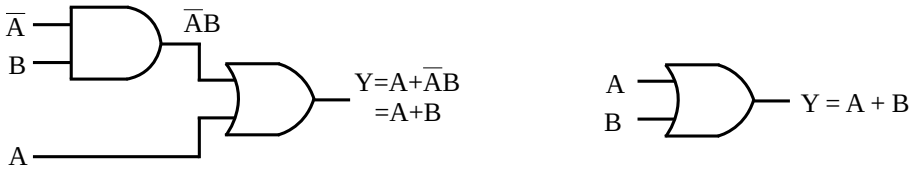

*Figur 4.6. Boole'sk regel 9 illustrert med inverterere.*

Regel 10 kan skrives $A + AB = A(1 + B) = A$ ved å bruke den distributive lov og regel 2. I figur 4.7 er regel 10 illustrert.




*Figur 4.7. Boole'sk regel 10.*

For regel 11 ses at $A + \overline{A}B = A + AB + \overline{A}B = A + (A + \overline{A})B = A + B$, der vi først har brukt regel 10 og så regel 4 (etter å ha brukt den distributive lov). I figur 4.8 er denne regelen illustrert.




*Figur 4.8. Boole'sk regel 11.*

Gyldigheten av regel 12 er kanskje enklest å se ved bruk av sannhetstabellen vist nedenfor. Det ses at (A + B)(A + C) = A + BC i henhold til regelen.

| A | B | C | A + B | A + C | (A + B)(A + C) | BC | A + BC |
|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 0 | 1 | 0 | 0 | 0 |
| 0 | 1 | 0 | 1 | 0 | 0 | 0 | 0 |
| 0 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| 1 | 0 | 0 | 1 | 1 | 1 | 0 | 1 |
| 1 | 0 | 1 | 1 | 1 | 1 | 0 | 1 |
| 1 | 1 | 0 | 1 | 1 | 1 | 0 | 1 |
| 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |

### 4.3.5. De Morgan's teoremer

Augustus De Morgan var en matematiker som kjente til Boole's arbeid. Han kompletterte Boole's arbeid med to teoremer. Det første er gitt ved:

$$\overline{A \cdot B} = \overline{A} + \overline{B} \tag{4.6}$$

Dette teoremet er illustrert med logiske porter i figur 4.9.


*Figur 4.9. De Morgan's teorem 1.*

Gyldigheten av dette teoremet kan vises ved hjelp av sannhetstabellen vist nedenfor.

| A | B | $\overline{A}$ | $\overline{B}$ | $\overline{A \cdot B}$ | $\overline{A} + \overline{B}$ |
|---|---|---|---|---|---|
| 0 | 0 | 1 | 1 | 1 | 1 |
| 0 | 1 | 1 | 0 | 1 | 1 |
| 1 | 0 | 0 | 1 | 1 | 1 |
| 1 | 1 | 0 | 0 | 0 | 0 |

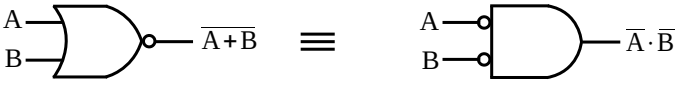
Teoremet gjelder også for flere variable. For tre variable er således teoremet gitt som:

$$\overline{A \cdot B \cdot C} = \overline{A} + \overline{B} + \overline{C} \tag{4.7}$$

Det andre teoremet er gitt ved:

$$\overline{A + B} = \overline{A} \cdot \overline{B} \tag{4.8}$$

Dette teoremet er illustrert med logiske porter i figur 4.10.




*Figur 4.10. De Morgan's teorem 2.*

Gyldigheten av dette teoremet kan vises ved hjelp av sannhetstabellen vist nedenfor.

| A | B | $\overline{A}$ | $\overline{B}$ | $\overline{A+B}$ | $\overline{A}\cdot\overline{B}$ |
|---|---|---|---|---|---|
| 0 | 0 | 1 | 1 | 1 | 1 |
| 0 | 1 | 1 | 0 | 0 | 0 |
| 1 | 0 | 0 | 1 | 0 | 0 |
| 1 | 1 | 0 | 0 | 0 | 0 |

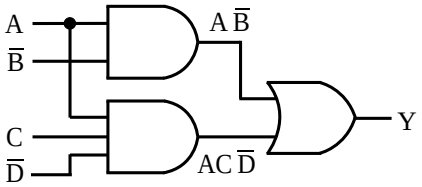
Teoremet gjelder også for flere variable. For tre variable er således teoremet gitt som:

$$\overline{A + B + C} = \overline{A} \cdot \overline{B} \cdot \overline{C} \tag{4.9}$$

# 4.4. SOP og POS

## 4.4.1. Sum av produkter, SOP

Anta Y = A$\overline{B}$ + AC$\overline{D}$, se figur 4.11. Dette er et uttrykk med sum av produkter (Sum Of Products, SOP). I dette eksemplet er det to produkter og en sum. Uttrykket inneholder også fire litteraler A, B, C og D. Denne typen uttrykk egner seg godt for realisering med bare NAND-porter som vi skal se senere.




*Figur 4.11. Eksempel på sum av produkter, SOP.*

Ofte opereres med standard SOP-form. Det betyr at alle litteraler skal være med i hvert produkt. Skal Y skrives på standard form, kan vi benytte at B + $\overline{B}$ = 1, C + $\overline{C}$ = 1 og D + $\overline{D}$ = 1:

$$A\overline{B} = A\overline{B}(C+\overline{C}) = A\overline{B}C + A\overline{B}\overline{C} \quad \text{der}$$
$$A\overline{B}C = A\overline{B}C(D+\overline{D}) = A\overline{B}C\,D + A\overline{B}C\,\overline{D}$$
$$A\overline{B}\overline{C} = A\overline{B}\overline{C}(D+\overline{D}) = A\overline{B}\overline{C}\,D + A\overline{B}\overline{C}\,\overline{D} \Rightarrow$$
$$A\overline{B} = A\overline{B}CD + A\overline{B}C\overline{D} + A\overline{B}\overline{C}D + A\overline{B}\overline{C}\,\overline{D}$$

Videre fås:

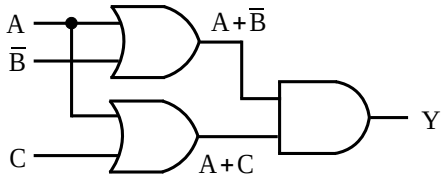
$$AC\overline{D} = A\,C\overline{D}(B+\overline{B}) = A\,B\,C\overline{D} + A\,\overline{B}\,C\,\overline{D}$$

Dette gir da:

$$Y = A\,\overline{B} + AC\overline{D} = A\,\overline{B}\,CD + A\,\overline{B}\,C\,\overline{D} + A\,\overline{B}\,\overline{C}\,D + A\,\overline{B}\,\overline{C}\,\overline{D} + ABC\,\overline{D} + A\,\overline{B}\,C\,\overline{D}$$

### 4.4.2. Produkt av summer, POS

Anta Y = (A + $\overline{B}$)·(A + C), se figur 4.12. Dette er et uttrykk med produkt av summer (Products Of Sums, POS). I dette eksemplet er det to summer og ett produkt. Uttrykket inneholder også tre litteraler A, $\overline{B}$ og C. Denne typen uttrykk egner seg godt for realisering med bare NOR-porter som vi skal se senere.




*Figur 4.12. Eksempel på produkt av summer, POS.*

Ofte opereres med standard POS-form. Det betyr at alle litteraler skal være med i hvert produkt. Skal Y ovenfor skrives på standard form, kan vi benytte at B$\overline{B}$=0 og C$\overline{C}$=0:

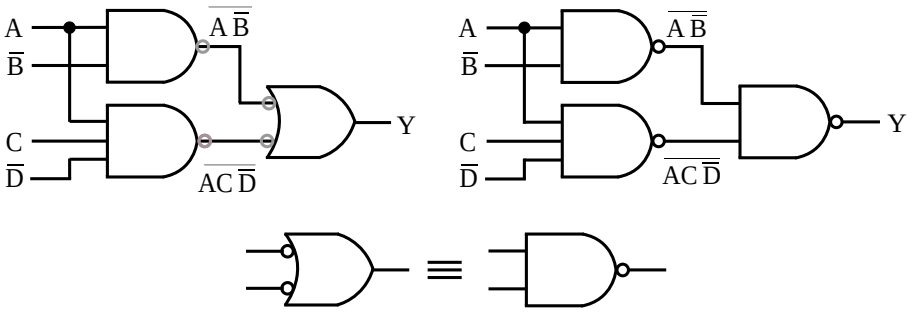
$$A + \overline{B} = A + \overline{B} + C\,\overline{C} = (A + \overline{B} + C)(A + \overline{B} + \overline{C}) \quad (\text{Regel } 12)$$
$$A + C = A + B\,\overline{B} + C = (A + B + C)(A + \overline{B} + C) \quad (\text{Regel } 12) \Rightarrow$$
$$Y = (A + \overline{B} + C)(A + \overline{B} + \overline{C})(A + B + C)(A + \overline{B} + C)$$

## 4.5. Realisering med bare NAND- eller NOR-porter

### 4.5.1. Bare NAND-porter

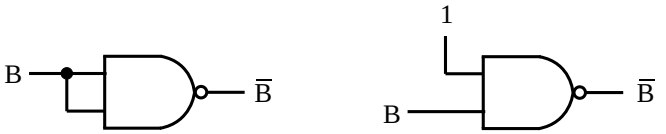
Realiseringen av uttrykket Y = A$\overline{B}$ + AC$\overline{D}$ var vist i figur 4.11. Dette er et uttrykk med sum av produkter, SOP. Til høyre i figur 4.13 er vist realiseringen ved hjelp av bare NAND-porter.

Til venstre i figuren er vist hvordan en kan tenke seg overgangen fra figur 4.11 med å bruke dobbelinvertering. Nederst i figuren ses hvordan De Morgans teorem 1 kan illustreres for å gå fra ELLER-port med inverterte innganger til en NAND-port.




*Figur 4.13. Realisering ved hjelp av bare NAND-porter.*

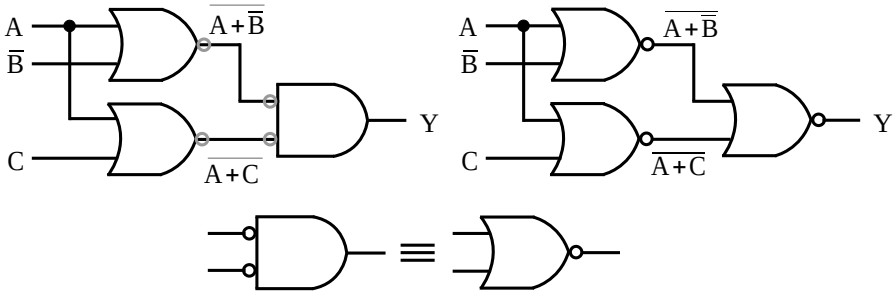
Dersom den inverterte (av B og D) ikke finnes, kan en også bruke en NAND-port for å realisere en inverterer, se figur 4.14.




*Figur 4.14. NAND-port som inverterer.*

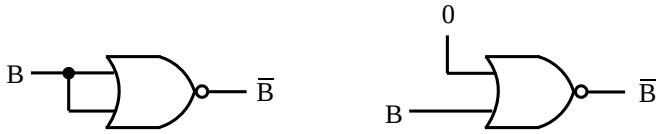
## 4.5.2. Bare NOR-porter

Realiseringen av uttrykket Y = (A + B)(A + C) var vist i figur 4.12. Dette er et uttrykk med produkt av summer, POS. Til høyre i figur 4.15 er vist realiseringen ved hjelp av bare NOR-porter. Til venstre i figuren er vist hvordan en kan tenke seg overgangen fra figur 4.12 ved å bruke dobbelinvertering. Nederst i figuren ses hvordan De Morgans teorem 2 kan illustreres for å gå fra OG-port med inverterte innganger til en NOR-port.




*Figur 4.15. Realisering ved hjelp av bare NOR-porter.*

Dersom den inverterte (av B) ikke finnes, kan en også bruke en NOR-port for å realisere en inverterer som illustrert i figur 4.16.

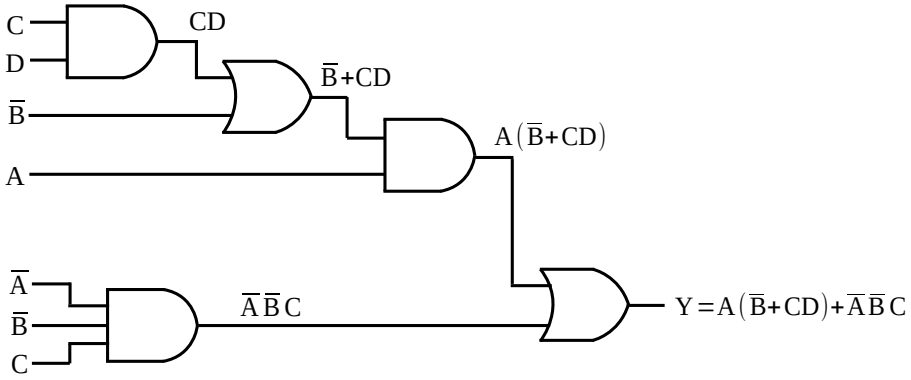



*Figur 4.16. NOR-port som inverterer.*

# 4.6. Forenkling av logiske uttrykk

## 4.6.1. Fra logisk krets

I figur 4.17 er vist en logisk krets som ikke er optimal fordi det er brukt flere porter enn nødvendig.

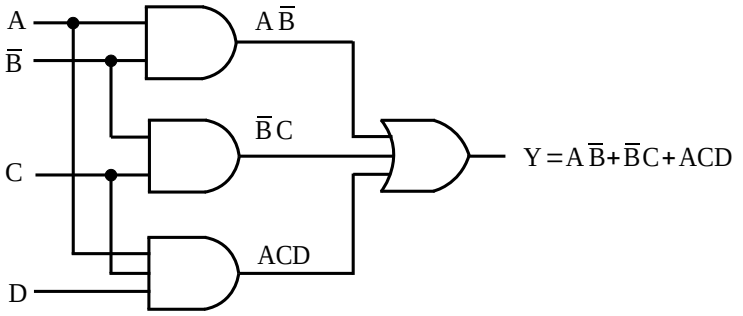



*Figur 4.17. Krets som ønskes forenklet.*

Forenklingen kan da gjøres slik:

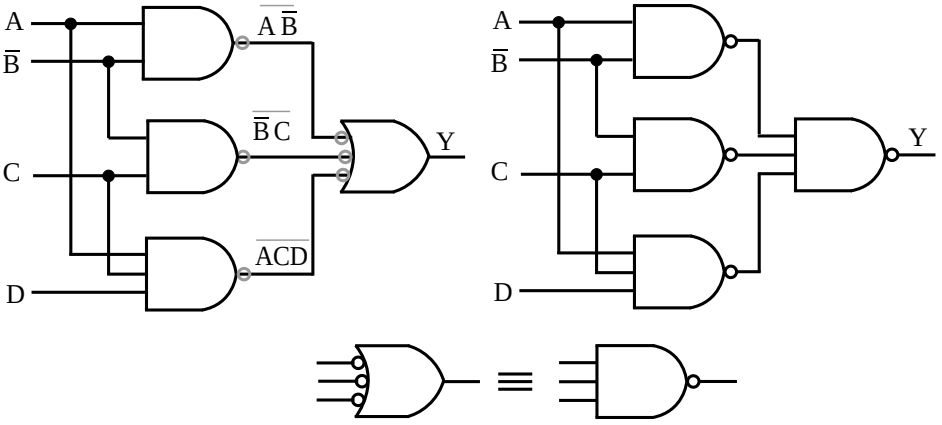
$$Y = A(\overline{B}+CD)+\overline{A}\,\overline{B}\,C = A\,\overline{B}+\overline{A}\,\overline{B}\,C+ACD$$
$$Y = \overline{B}(A+\overline{A}\,C)+ACD$$
$$Y = \overline{B}(A+C)+ACD \quad (\text{Regel } 11)$$
$$Y = \overline{B}\,A+\overline{B}\,C+ACD \quad \text{Forenklet SOP}$$

Realiseringen av dette uttrykket er vist i figur 4.18. Det ses at det benyttes færre porter.




*Figur 4.18. Skjema for redusert uttrykk.*

Vi kan også tenke oss en realisering med bare NAND-porter. Dette er vist til høyre i figur 4.19. Til venstre i figuren er vist hvordan en kan tenke seg overgangen fra figur 4.18 med å bruke dobbel-invertering. Nederst i figuren ses hvordan De Morgans teorem 1 kan illustreres for å gå fra ELLER-port med inverterte innganger til en NAND-port.



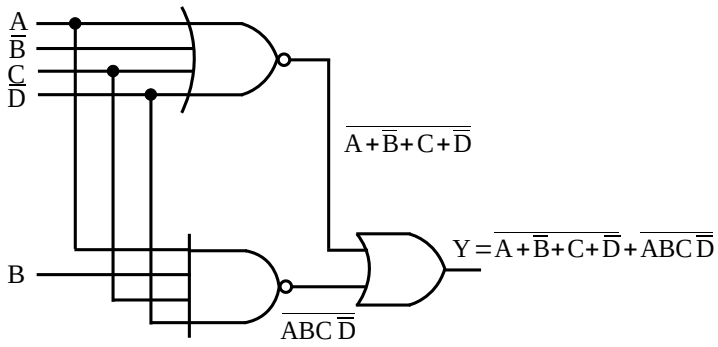

*Figur 4.19. Realisering ved hjelp av bare NAND-porter.*

## 4.6.2. Fra logisk uttrykk

Gitt følgende logiske uttrykk:

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

En realisering av uttrykket er vist i figur 4.20.



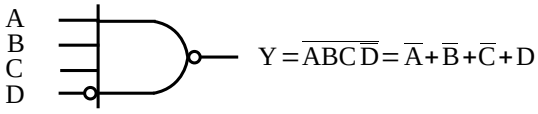

*Figur 4.20. Krets som skal forenkles.*

Her kan vi først benytte De Morgan:

$$Y=\overline{A+\overline{B}+C+\overline{D}}+\overline{A\,B\,C\,\overline{D}}=\overline{A}\,B\,\overline{C}\,D+\overline{A}+\overline{B}+\overline{C}+D=\overline{A}\,B\,\overline{C}\,D+\overline{A}\cdot 1+\overline{B}+\overline{C}+D$$

$$Y=\overline{A}(B\overline{C}D+1)+\overline{B}+\overline{C}+D=\overline{A}+\overline{B}+\overline{C}+D$$

Her har vi til slutt benyttet den distributive lov og Regel 2. Realiseringen av dette uttrykket er meget enkelt med en NAND-port med fire innganger, der den ene er invertert, se figur 4.21. Det ses at forenklingen fra figur 4.20 er ganske omfattende.




*Figur 4.21. Realisering med NAND-port.*

# 4.7. Karnaugh-diagram

## 4.7.1. Innledning

Karnaugh-diagrammet er et hendig hjelpemiddel for å redusere logiske Boole'ske uttrykk. Dette diagrammet er sannhetstabellen på en spesiell matriseform. En rute i Karnaugh-diagrammet er en linje i sannhetstabellen. Naboruter i diagrammet har bare en forskjell i en litteral. Det benyttes følgelig Gray-kode.

## 4.7.2. Diagram for tre variable

La oss se på sannhetstabellen nedenfor som eksempel. Vi ser at Y = 1 for A = 1 og B = 1 samt for A = 1 og C = 1. Vi ser også at Y = 0 for A = 0 og for B+C = 0.

| A | B | C | Y | |
|---|---|---|---|---|
| 0 | 0 | 0 | 0 | $A+B+C=0$ |
| 0 | 0 | 1 | 0 | $A+B+\bar{C}=0$ |
| 0 | 1 | 0 | 0 | $A+\bar{B}+C=0$ |
| 0 | 1 | 1 | 0 | $A+\bar{B}+\bar{C}=0$ |
| 1 | 0 | 0 | 0 | $\bar{A}+B+C=0$ |
| 1 | 0 | 1 | 1 | $A\,\bar{B}\,C=1$ |
| 1 | 1 | 0 | 1 | $AB\,\bar{C}=1$ |
| 1 | 1 | 1 | 1 | $ABC=1$ |

Fra sannhetstabellen ser vi at standard SOP-form er:

$Y = A\overline{B}C + AB\overline{C} + ABC$

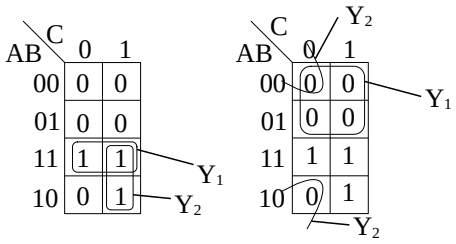
Likeledes ser vi at standard POS-form er:

$Y = (A+B+C)(A+B+\overline{C})(A+\overline{B}+C)(A+\overline{B}+\overline{C})(\overline{A}+B+C)$

Med utgangspunkt i sannhetstabellen kan vi tegne Karnaugh-diagrammet vist i figur 4.22. Legg merke til rekkefølgen for AB som er i henhold til Gray-koden.

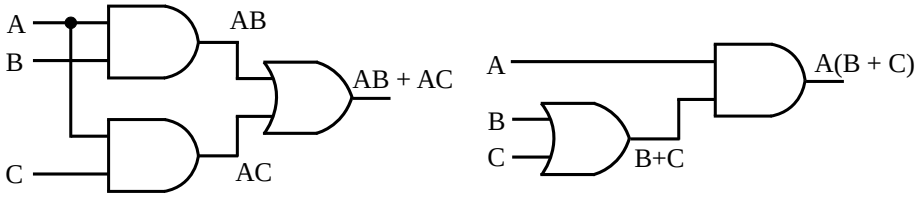
Til venstre i figuren ser vi at $Y_1 = 1$ for A = 1 og B = 1. Dette gir $Y_1 = AB$. Videre ser vi at $Y_2 = 1$ for A = 1 og C = 1. Dette gir $Y_2 = AC$. Følgelig er forenklingen på SOP-form: $Y = Y_1 + Y_2 = AB+AC$

Til høyre i figur 4.22 ser vi at $Y_1 = 0$ for A = 0. Dette gir $Y_1 = A$. Videre ser vi at $Y_2 = 0$ for B = 0 og C = 0. Dette gir $Y_2 = B+C$. Legg forøvrig merke til hva som er naborutene i $Y_2$. Dette gir da forenklingen på POS-form: $Y = Y_1 \cdot Y_2 = A(B+C)$.




*Figur 4.22. Karnaugh-diagram for 3 variable.*

Skjema for SOP- og POS-form er vist i figur 4.23. Det ses at i dette tilfellet gir POS-formen færrest antall porter.




*(a)* *(b)*

*Figur 4.23. Skjema for SOP-form (a) og POS-form (b).*

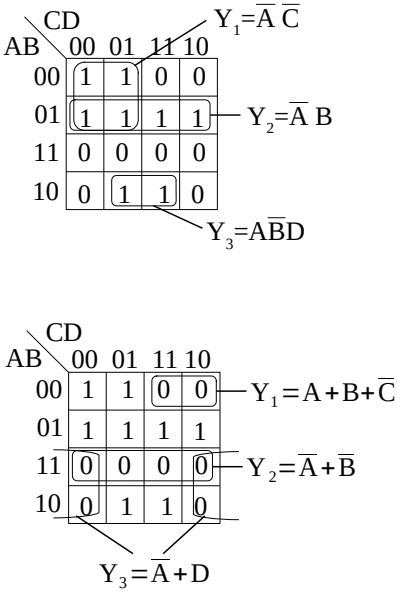
## 4.7.3. Diagram for fire variable

Et eksempel med sannhetstabell og Karnaugh-diagram for fire variable A, B, C og D er vist i figur 4.24.

Vi kan velge om vi vil redusere slik at vi står igjen med et uttrykk på SOP- eller POS-form. Fra øverst i figuren får vi på SOP-form:

$Y = Y_1 + Y_2 + Y_3 = \overline{A}\,\overline{C} + \overline{A}B + A\overline{B}D$

| A | B | C | D | Y |
|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 1 |
| 0 | 0 | 0 | 1 | 1 |
| 0 | 0 | 1 | 0 | 0 |
| 0 | 0 | 1 | 1 | 0 |
| 0 | 1 | 0 | 0 | 1 |
| 0 | 1 | 0 | 1 | 1 |
| 0 | 1 | 1 | 0 | 1 |
| 0 | 1 | 1 | 1 | 1 |
| 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 | 0 |
| 1 | 1 | 1 | 0 | 0 |
| 1 | 1 | 1 | 1 | 0 |



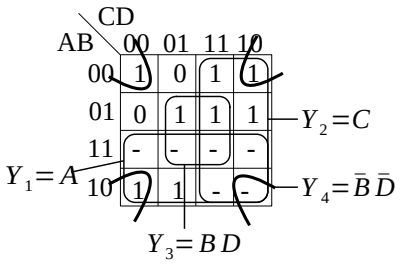

*Figur 4.24. Sannhetstabell og Karnaugh-diagram for fire variable.*

Ellers ser vi at det reduserte uttrykket på POS-form er:

$Y = Y_1 \cdot Y_2 \cdot Y_3 = (A+B+\overline{C})(\overline{A}+\overline{B})(\overline{A}+D)$

Disse uttrykkene vil selvfølgelig gi samme utgang for alle kombinasjonene i sannhetstabellen. Vi kan velge hvilken realisering vi vil velge. Og ønsker vi å realisere uttrykket med bare NAND-porter, vil vi velge Y på SOP-form. Tilsvarende vil vi velge Y på POS-form dersom vi ønsker å realisere uttrykket med bare NOR-porter.

I noen tilfeller forekommer det at noen inngangskombinasjoner ikke kan forekomme eller er valgfrie. Dette er illustrert i Karnaugh-diagrammet i figur 4.25. Vi bruker bindestrek (-) for å angi ruter som kan velges lik 0 eller 1.

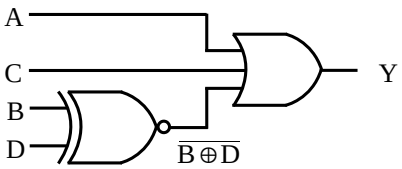



*Figur 4.25. Karnaugh-diagram med valgfrie ruter.*

I figuren er de valgfrie valgt som enere for å få størst mulige energrupper. Uttrykket på SOP-form er da:

$$Y = Y_1 + Y_2 + Y_3 + Y_4 = A + C + BD + \overline{B}\,\overline{D} = A + C + \overline{B \oplus D}$$

I figur 4.26 er vist en mulig realisering av Y.




*Figur 4.26. Realisering av* $Y = A + C + \overline{B \oplus D}$ .

Generelt kan de valgfrie velges lik 1 når det logiske uttrykket skal foreligge på SOP-form eller velges lik 0 når det logiske uttrykket skal foreligge på POS-form for å få logiske uttrykk som er mest mulig redusert.

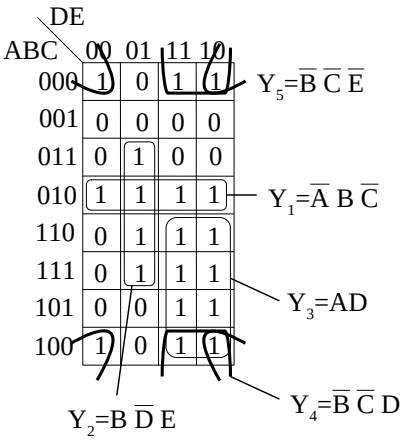
## 4.7.4. Diagram for fem variable

Et ferdig utfylt Karnaugh-diagram for fem variable er vist i figur 4.27. Dette svarer til en sannhetstabell med 32 kombinasjoner.

Vi ser at det reduserte uttrykket på SOP-form er:

$$Y = Y_1 + Y_2 + Y_3 + Y_4 + Y_5 = \overline{A}\,B\,\overline{C} + B\,\overline{D}\,E + AD + \overline{B}\,\overline{C}\,D + \overline{B}\,\overline{C}\,\overline{E}$$

Det er selvsagt også mulig å slå sammen nullere for å få det reduserte uttrykket på POS-form.




*Figur 4.27. Karnaugh-diagram for fem variable.*

Til slutt skal nevnes at det også kan benyttes Karnaugh-diagram for seks variable, men utover dette benyttes Karnaugh-diagram stort sett ikke på grunn av kompleksiteten.

# Kapittel 5
# Kombinatoriske funksjoner

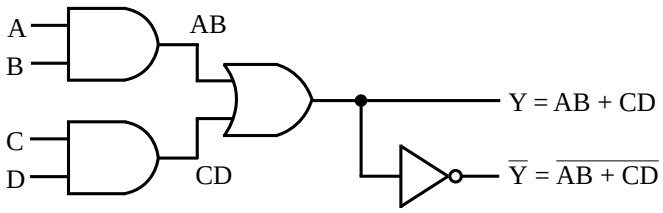
## 5.1. Innledning

Det finnes kombinatoriske funksjoner som er satt sammen av primitive logiske porter. Disse kan også refereres til som komponenter. Flere komponenter kan så benyttes til å sette sammen større digitale systemer.

Vi skal her først se på logisk realisering av de forskjellige kombinatoriske funksjonene før vi til slutt ser på noen CMOS-realiseringer.

## 5.2. OG-ELLER-port

En av de enkleste funksjonene er OG-ELLER-porten, som også kan kalles en komponent siden den finnes tilgjengelig som en ferdigbygd krets, se figur 5.1.



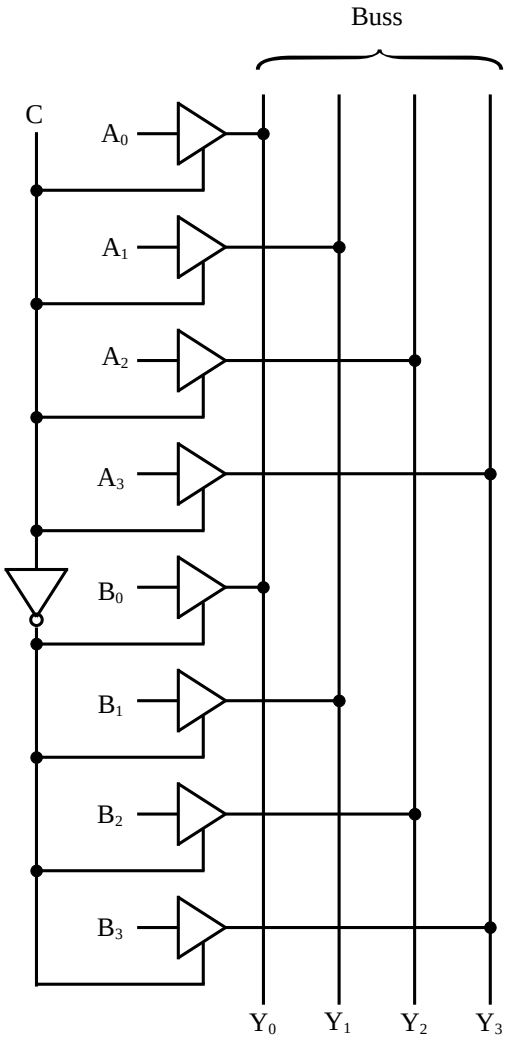

*Figur 5.1. OG-ELLER-port samt OG-NOR-port.*

Det ses at utgangen Y er 1 når både A og B er 1 eller når både C og D er 1. En OG-NOR-port er også tilgjengelig ved å invertere Y som vist i figuren.

## 5.3. Buss-driver

Vi har i kapittel 3.7 sett på 3-nivå buffere. Som nevnt kan slike benyttes ved at utgangene kan være koplet til samme leder i en buss. I noen tilfeller er dette bedre enn å bruke OG-ELLER-porter eller andre løsninger. Det er vist et eksempel med ikke-inverterende 3-nivå buffere i figur 5.2.

Det er to fire bits ord A og B tilkoplet bussen. Styresignalet til bufferne er C. Når C = 1 koples $A_0$-$A_3$ til bussen slik at $Y_0 = A_0$, $Y_1 = A_1$ etc., se sannhetstabell for buffer i samme figur. Da er samtidig ord B frakoplet bussen siden utgangen på disse fire bufferne er høyimpedante.




Sannhetstabell for buffer

| Inngang | | Utgang |
|---|---|---|
| C | X | Y |
| 0 | 0 | Z |
| 0 | 1 | Z |
| 1 | 0 | 0 |
| 1 | 1 | 1 |

Z - Høyimpedant

*Figur 5.2. Buss tilkoplet to fire bits ord*

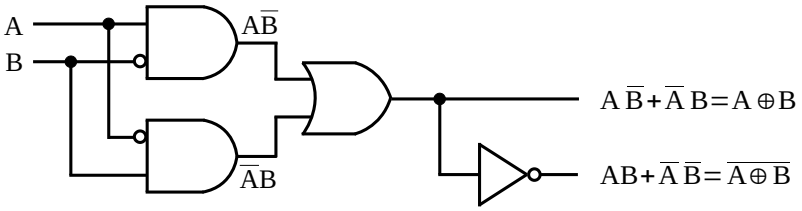
Når C = 0 koples $B_0$-$B_3$ til bussen slik at $Y_0 = B_0$, $Y_1 = B_1$ etc. Da er samtidig ord A frakoplet bussen siden utgangen på disse fire bufferne er høyimpedante.

I eksemplet er det brukt bare to ord. Imidlertid kan vi tenke oss å «henge» flere ord på bussen ved å bruke flere 3-nivå buffere og flere kontrollsignaler. Ordlengden kan selvfølgelig også økes (eller minskes) hvis dette er ønskelig.

## 5.4. Eksklusiv ELLER-port

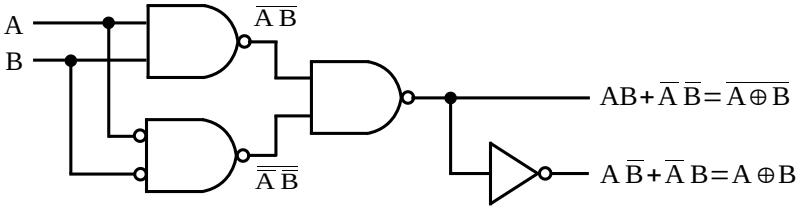
Som tidligere nevnt finnes Eksklusiv ELLER-porten som egen komponent. Det finnes også flere måter å realisere denne funksjonen på.

I figur 5.3 er vist en realisering med OG-ELLER-port. Det er også tatt med en ekstra inverting slik at Eksklusiv NOR også realiseres.




*Figur 5.3. Eksklusiv ELLER og Eksklusiv NOR.*

I figur 5.4 er vist realisering av Eksklusiv NOR med NAND-porter. Det er også tatt med en invertering for å få med realiseringen av Eksklusiv ELLER.

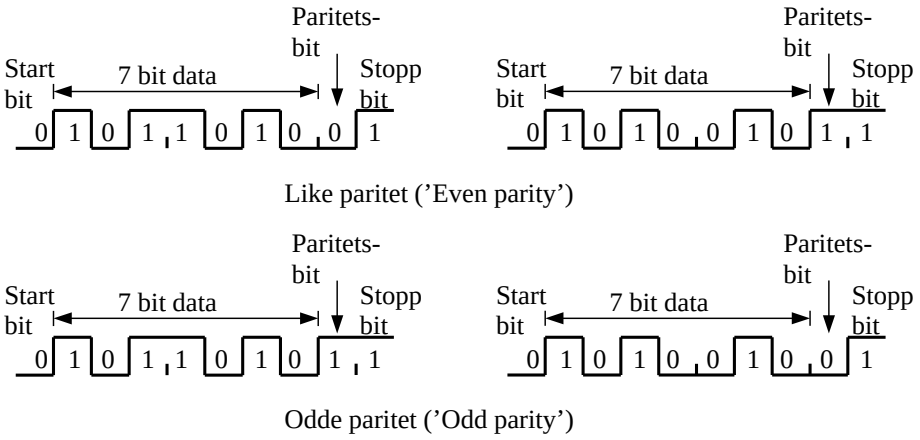



*Figur 5.4. Eksklusiv NOR og Eksklusiv ELLER med NAND-porter.*

## 5.5. Paritetsbit-generator

Ved overføring av digitale signaler er det ønskelig å kunne detektere feil i mottatt data. Den enkleste teknikken for å detektere feil er ved bruk av paritetsbit. Metoden er for eksempel i utstrakt bruk ved lavhastighets seriell kommunikasjon med UART, Universal Asynchronous Receiver-Transmitter.

Paritetsbit (1 eller 0) legges da til slutten på et dataord slik at antall enere i det nye dataordet (med paritetsbit) er enten like (lik paritet) eller odde (odde paritet). Dette er vist i figur 5.5. Det er her benyttet et format som benyttes for UART med 7 bit dataord.




*Figur 5.5. Paritetsbit for like og odde paritet.*

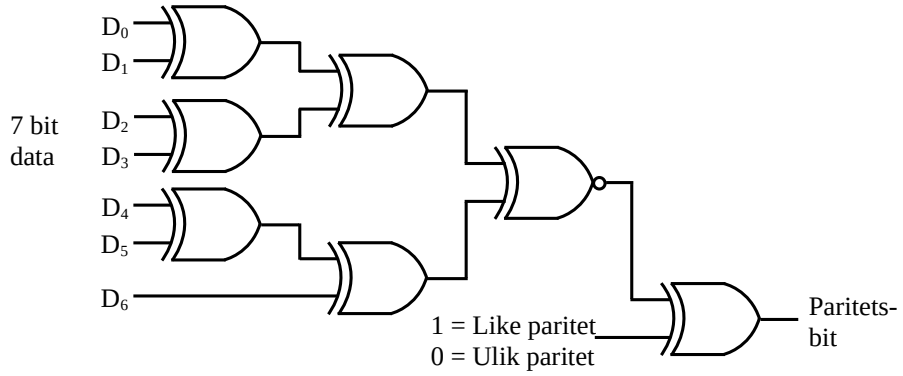
Overføringen starter med et start-bit, som alltid er logisk 0, og avsluttes med ett (eller flere) stoppbit, som alltid er logisk 1. Fra figuren ses at paritetsbit settes inn før stopp-bit. Sender og mottaker må begge settes til å ha enten like eller odde paritet. Det bør også nevnes at bruk av paritetsbit for UART er opsjonell.

I mottakeren telles hvor mange enere det er mottatt (med paritetsbit). Dersom paritetsbit er feil, kan det konkluderes med at overføringen er feil. Normalt må da mottatt data forkastes, og i de fleste tilfeller vil en da be om å få sendt data på nytt. Bemerk at feil mottatt paritetsbit kan bety at bare dette bit er feil og at data er riktig såvel som at det er feil i data som er mottatt.

Det medfører også at riktig paritetsbit ikke nødvendigvis betyr at mottatt data er riktig siden det kan ha oppstått dobbeltfeil ved overføringen. Denne type feildeteksjon egner seg således best der en vet at overføringskanalen er relativt feilfri. Der det ønskes større robusthet mot feil, må det innføres større redundans, det vil si at det må innføres flere ekstra bit for å være i stand til å oppdage feilen.

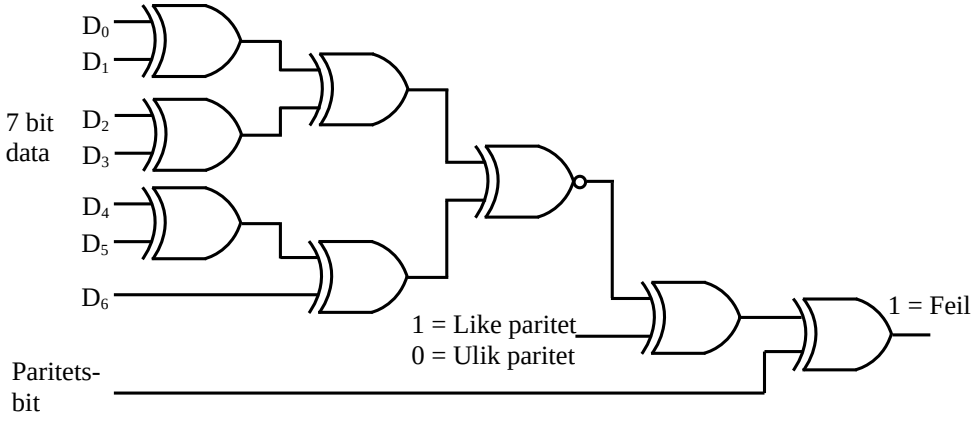
I figur 5.6 er vist en paritetsbit-generator for 7 bit data, benevnt $D_0$ - $D_6$. Det benyttes Eksklusiv-ELLER-porter og en Eksklusiv-NOR-port, der sistnevnte brukes siden vi har et ulike antall databit. Utgangsporten i figuren bestemmer om det skal være like eller ulike paritet.

For å få  den serielle bitstrømmen i figur 5.5, kan det benyttes et skiftregister med parallell innlesing og seriell utgang, se kapittel 7.4.




*Figur 5.6. Paritetsbit-generator for 7 bit data.*

Skjemaet i figur 5.6 kan også brukes i mottakeren for å realisere en paritetsbit-kontroller, se figur 5.7. De mottatte 7 bit data brukes til å generere et paritetsbit (som i paritetsbit-generatoren). Dette bit sammenlignes med det mottatte paritetsbit i en Eksklusiv-ELLER-port. Resultatet vil da være lik 0 dersom resultatet er riktig, men lik 1 hvis en feil har oppstått.



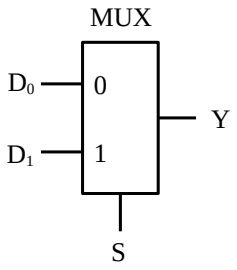

*Figur 5.7. Paritetsbit-kontroller.*

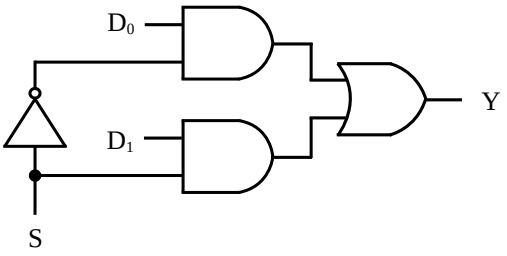
I mottakeren må den serielle bitstrømmen fås over på parallell form. Det kan da benyttes et skiftregister med seriell innlesing og parallell utgang, se kapittel 7.3.

## 5.6. Multipleksere

En multiplekser (MUX) lager forbindelse mellom én av flere innganger til én utgang. I figur 5.8 er vist en enkel multiplekser med to data-innganger og en logisk realisering med OG-ELLER-porten og en inverterer.

Inngangene er benevnt $D_0$ og $D_1$. Inngangen S (Select) velger mellom disse inngangene. Sannhetstabellen for denne multiplekseren er vist til høyre.



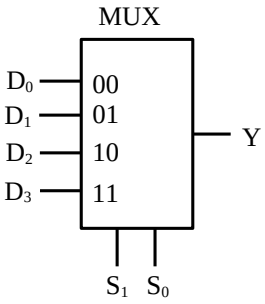

| S | Y |
|---|---|
| 0 | $D_0$ |
| 1 | $D_1$ |

*Figur 5.8. Enkel multiplekser.*

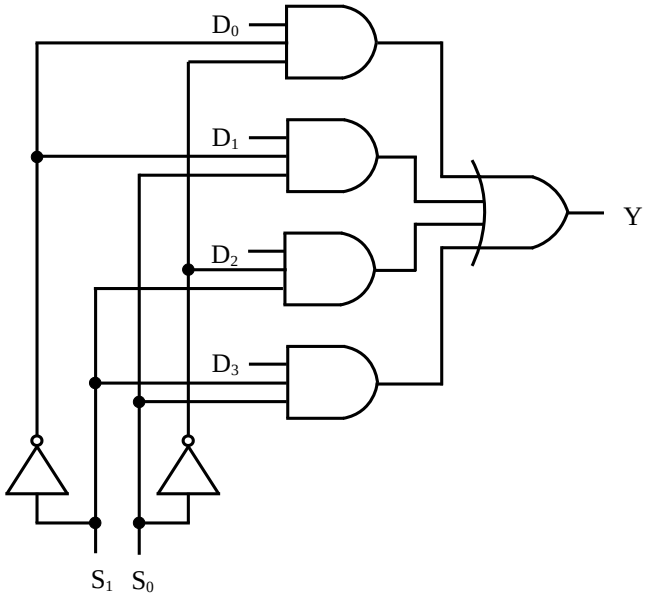
Utgangen $Y = D_0$ når $S = 0$ og $Y = D_1$ når $S = 1$. Logisk uttrykk for denne multiplekseren er:

$$Y = D_0 \bar{S} + D_1 S \tag{5.1}$$

En multiplekser med fire innganger er vist til venstre i figur 5.9 sammen med en logisk realisering til høyre i figuren.




| $S_1$ | $S_0$ | Y |
|---|---|---|
| 0 | 0 | $D_0$ |
| 0 | 1 | $D_1$ |
| 1 | 0 | $D_2$ |
| 1 | 1 | $D_3$ |

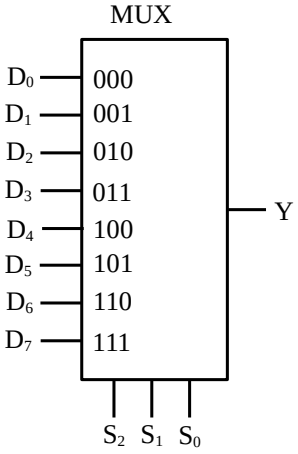



*Figur 5.9. Fire inngangers multiplekser.*

Inngangene er benevnt $D_0$ - $D_3$. Inngangene $S_0$ og $S_1$ velger mellom disse inngangene, se sannhetstabellen vist i figuren. Det ses at utgangen $Y = D_0$ eller $D_1$ når $S_1 = 0$ og $Y = D_2$ eller $D_3$ når $S_1 = 1$. Et logisk uttrykk for denne multiplekseren kan skrives:

$$Y = D_0 \overline{S_1}\,\overline{S_0} + D_1 \overline{S_1} S_0 + D_2 S_1 \overline{S_0} + D_3 S_1 S_0 \tag{5.2}$$

Med åtte innganger får vi multiplekseren med sannhetstabell som vist i figur 5.10.




| $S_2$ | $S_1$ | $S_0$ | Y |
|---|---|---|---|
| 0 | 0 | 0 | $D_0$ |
| 0 | 0 | 1 | $D_1$ |
| 0 | 1 | 0 | $D_2$ |
| 0 | 1 | 1 | $D_3$ |
| 1 | 0 | 0 | $D_4$ |
| 1 | 0 | 1 | $D_5$ |
| 1 | 1 | 0 | $D_6$ |
| 1 | 1 | 1 | $D_7$ |

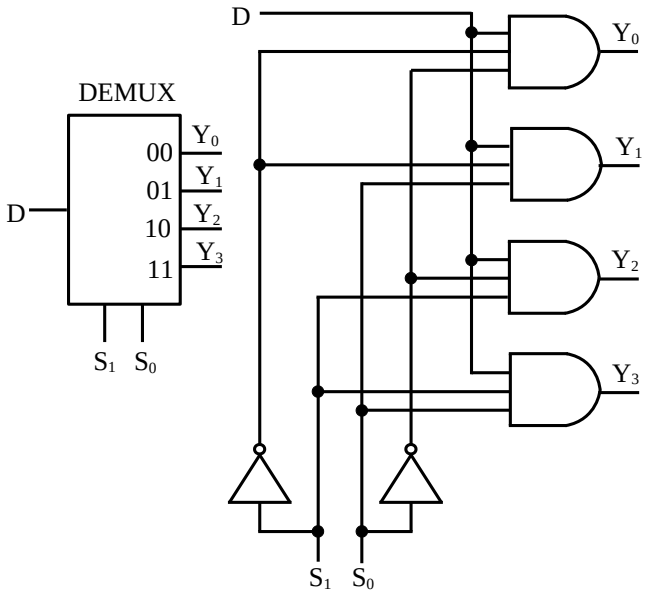
*Figur 5.10. Åtte inngangers multiplekser.*

## 5.7. Demultipleksere

En demultiplekser (DEMUX) utfører den motsatte operasjonen av en multiplekser: den inngangen som velges av styresignalene, føres til utgangen.

Med fire utganger kan en demultiplekser vist til venstre i figur 5.11 realiseres logisk som vist midt i figuren. Utgangene er benevnt $Y_0$ - $Y_3$. Inngangene $S_0$ og $S_1$ velger hvilken utgang inngangen D skal føres til, se sannhetstabellen vist til høyre. De logiske uttrykk for denne demultiplekseren kan da skrives:

$$\begin{aligned} Y_0 &= D\,\overline{S_1}\,\overline{S_0} \\ Y_1 &= D\,\overline{S_1} S_0 \\ Y_2 &= D\,S_1\,\overline{S_0} \\ Y_3 &= D\,S_1 S_0 \end{aligned} \tag{5.3}$$




| D | $S_1$ | $S_0$ | $Y_0$ | $Y_1$ | $Y_2$ | $Y_3$ |
|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 0 | 0 | 0 | 0 |
| 0 | 1 | 0 | 0 | 0 | 0 | 0 |
| 0 | 1 | 1 | 0 | 0 | 0 | 0 |
| 1 | 0 | 0 | 1 | 0 | 0 | 0 |
| 1 | 0 | 1 | 0 | 1 | 0 | 0 |
| 1 | 1 | 0 | 0 | 0 | 1 | 0 |
| 1 | 1 | 1 | 0 | 0 | 0 | 1 |

*Figur 5.11. Demultiplekser med fire utganger.*

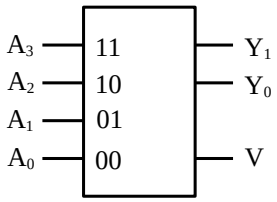
# 5.8. Prioritetskodere

## 5.8.1. Innledning

På utgangen av en prioritetskoder finnes en binær kode som avhenger av inngangenes tilstand på en slik måte at det foretas en prioritering mellom disse. Prioritetskodere brukes for eksempel til avbruddsprioritering for mikrokontrollere. Mikrokontrolleren kan da «hoppe ut» av sitt vanlige program og kjøre en avbruddsrutine. Det kan finnes flere slike rutiner, og denne koderen kan prioritere mellom disse.

## 5.8.2. 4-2 prioritetskoder

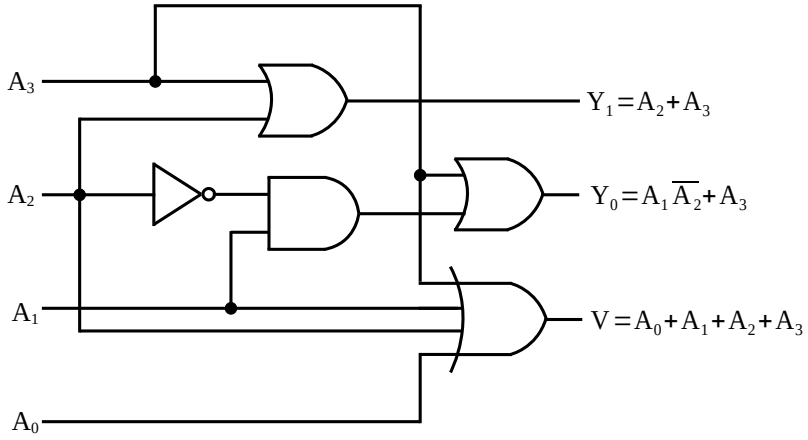
Et blokkdiagram og sannhetstabell for en 4-2 prioritetskoder er vist i figur 5.12. Fra sannhetstabellen fremgår det at inngang $A_3$ har høyeste prioritet. Når denne er 1, har det ingen betydning hva de andre inngangene er. Dette er vist med tegnet -, som betyr vilkårlig (Don't care). Siden $A_0 = 1$ gir både $Y_1$ og $Y_0$ lik 0, brukes en ekstra utgang for å skille dette resultatet fra det tilfellet at alle inngangene er lik 0. Denne ekstra utgangen er kalt V (Validate).




| $A_3$ | $A_2$ | $A_1$ | $A_0$ | $Y_1$ | $Y_0$ | V |
|---|---|---|---|---|---|---|
| 1 | - | - | - | 1 | 1 | 1 |
| 0 | 1 | - | - | 1 | 0 | 1 |
| 0 | 0 | 1 | - | 0 | 1 | 1 |
| 0 | 0 | 0 | 1 | 0 | 0 | 1 |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 |

*Figur 5.12. Blokkdiagram og sannhetstabell for 4-2 prioritetskoder.*

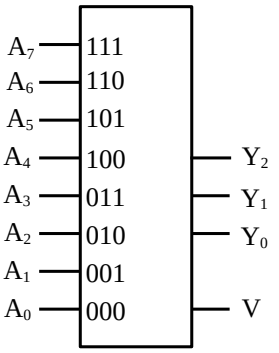
En mulig logisk realisering av denne prioritetskoderen er vist i figur 5.13. Den baserer seg på at alle utganger er lik 0 for alle innganger lik 0 som vist i sannhetstabellen i figur 5.12.




*Figur 5.13. Logisk realisering av 4-2 prioritetskoder.*

## 5.8.3. 8-3 prioritetskoder

Et blokkdiagram og sannhetstabell for en 8-3 prioritetskoder er vist i figur 5.14.



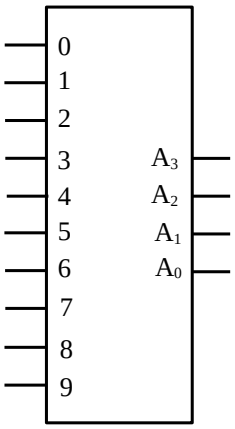

| $A_7$ | $A_6$ | $A_5$ | $A_4$ | $A_3$ | $A_2$ | $A_1$ | $A_0$ | $Y_2$ | $Y_1$ | $Y_0$ | V |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | - | - | - | - | - | - | - | 1 | 1 | 1 | 1 |
| 0 | 1 | - | - | - | - | - | - | 1 | 1 | 0 | 1 |
| 0 | 0 | 1 | - | - | - | - | - | 1 | 0 | 1 | 1 |
| 0 | 0 | 0 | 1 | - | - | - | - | 1 | 0 | 0 | 1 |
| 0 | 0 | 0 | 0 | 1 | - | - | - | 0 | 1 | 1 | 1 |
| 0 | 0 | 0 | 0 | 0 | 1 | - | - | 0 | 1 | 0 | 1 |
| 0 | 0 | 0 | 0 | 0 | 0 | 1 | - | 0 | 0 | 1 | 1 |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 1 |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

*Figur 5.14. Blokkdiagram og sannhetstabell for 8-3-prioritetskoder.*

Virkemåten er som for 4-2 prioritetskoderen. Her er det inngang $A_7$ som har høyeste prioritet. Også her benyttes en ekstra utgang for å sette utgangen V (Validate) lik 0 når alle inngangene er lik 0.

## 5.9. Desimal til BCD koder

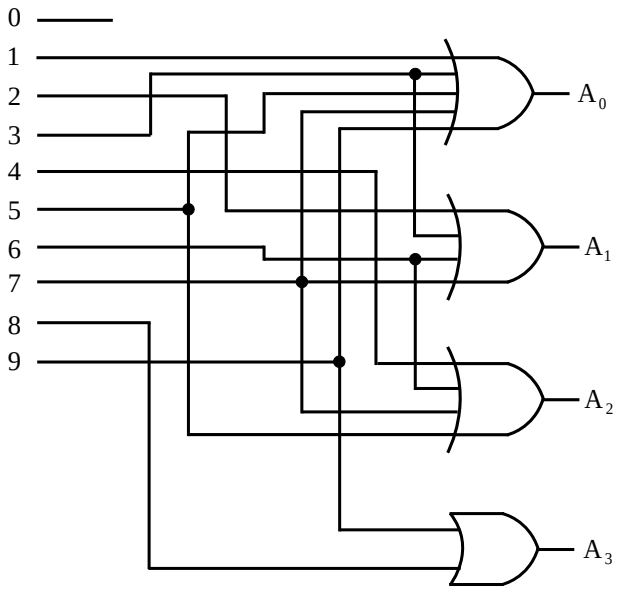
Denne koderen har 10 innganger, en for hvert desimale siffer, mens utgangen er BCD-kode (4 bit). Bare en av inngangene er høy av gangen, med andre ord er dette ingen prioritetskoder. I figur 5.15 er vist blokkskjema og sannhetstabell for denne koderen.




| Siffer | BCD-kode | | | |
|---|---|---|---|---|
| | $A_3$ | $A_2$ | $A_1$ | $A_0$ |
| 0 | 0 | 0 | 0 | 0 |
| 1 | 0 | 0 | 0 | 1 |
| 2 | 0 | 0 | 1 | 0 |
| 3 | 0 | 0 | 1 | 1 |
| 4 | 0 | 1 | 0 | 0 |
| 5 | 0 | 1 | 0 | 1 |
| 6 | 0 | 1 | 1 | 0 |
| 7 | 0 | 1 | 1 | 1 |
| 8 | 1 | 0 | 0 | 0 |
| 9 | 1 | 0 | 0 | 1 |

*Figur 5.15. Blokkdiagram og sannhetstabell for desimal til BCD koder.*

I figur 5.16 er vist en mulig logisk realisering av desimal til BCD-koderen med ELLER-porter. En kan også tenke seg denne koderen med inverterte innganger. Da er bare en av inngangene lav i gangen. Denne koderen kan også ha inverterte utganger, da vil utgangen være for eksempel lik 1111 for siffer 0.




*Figur 5.16. Logisk realisering av desimal til BCD koder.*

# 5.10. 2-4 og 3-8 dekodere

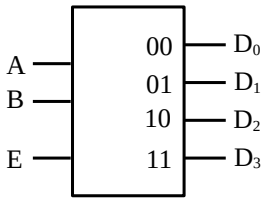
## 5.10.1. Innledning

Disse dekoderne kan sies å foreta den motsatte prosessen av prioritetskoderne. De bruker et binært mønster på inngangen til å aktivere en utgang av gangen. Ofte har dekoderen også en Enable-inngang.

Grunnen til at det ofte benyttes en Enable-inngang er at det kan tenkes tilfeller der en ønsker å ha 0 på utgangen. Det finnes også dekodere av typen 3-nivå der utgangene koples fra når E = 0.

## 5.10.2. 2-4 dekoder

I figur 5.17 er vist blokkdiagram og sannhetstabell for en 2-4 dekoder, der E er Enable-inngangen. Når E = 1, vil en av utgangene $D_0$-$D_3$ gå høy, som vist i figuren. Når E = 0, vil alle utgangene legges til 0.

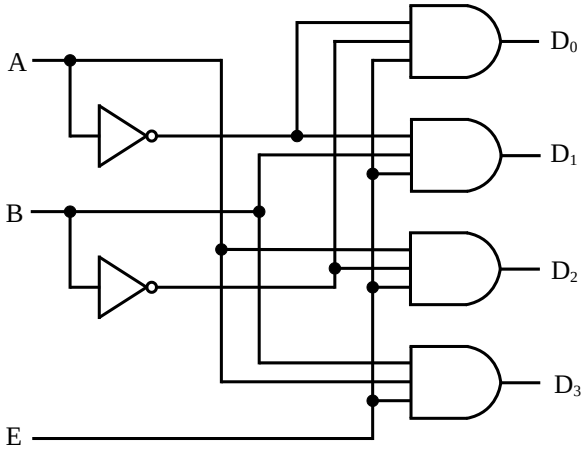



| E = 1 | | | | | |
|---|---|---|---|---|---|
| A | B | $D_0$ | $D_1$ | $D_2$ | $D_3$ |
| 0 | 0 | 1 | 0 | 0 | 0 |
| 0 | 1 | 0 | 1 | 0 | 0 |
| 1 | 0 | 0 | 0 | 1 | 0 |
| 1 | 1 | 0 | 0 | 0 | 1 |

| E = 0 | | | | | |
|---|---|---|---|---|---|
| A | B | $D_0$ | $D_1$ | $D_2$ | $D_3$ |
| 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 1 | 0 | 0 | 0 | 0 |
| 1 | 0 | 0 | 0 | 0 | 0 |
| 1 | 1 | 0 | 0 | 0 | 0 |

*Figur 5.17. Blokkdiagram og sannhetstabell for 2-4 dekoder.*

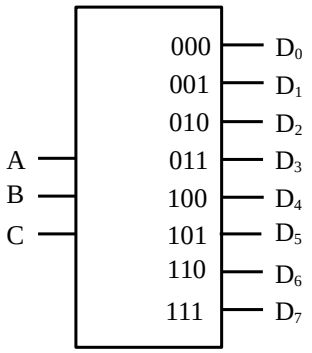
Det finnes også dekodere der utgangene er inverterte. I figur 5.18 er vist en mulig realisering av 2-4-dekoderen med blokkdiagrammet i figur 5.17.




*Figur 5.18. Logisk realisering av 2-4-dekoder.*

## 5.10.3. 3-8 dekoder

I figur 5.19 er vist blokkdiagram og sannhetstabell for en 3-8 dekoder. Den er vist uten Enable-inngang, men det er vanlig å ha en slik inngang også for 3-8 dekodere. Det kan også nevnes at antall bit på inngangen kan økes til 4, dermed fås en 4-16 dekoder der det er 16 utganger. Det fås dermed blokkdiagram og sannhetstabell tilsvarende figur 5.19 med 4 innganger og 16 utganger med denne utvidelsen.

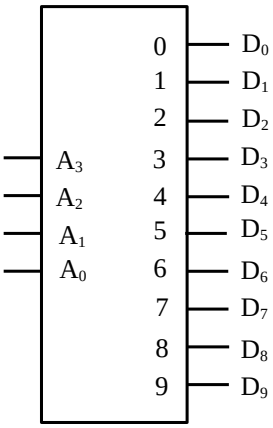



| A | B | C | $D_0$ | $D_1$ | $D_2$ | $D_3$ | $D_4$ | $D_5$ | $D_6$ | $D_7$ |
|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 1 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 |
| 0 | 1 | 1 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 |
| 1 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 |
| 1 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 |
| 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 |
| 1 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 |

*Figur 5.19. Blokkdiagram og sannhetstabell for 3-8 dekoder.*

## 5.11. BCD til desimal dekoder

I figur 5.20 er vist blokkskjema og sannhetstabell for BCD til desimal dekoder.




| BCD-kode | | | | Siffer | Funksjon |
|---|---|---|---|---|---|
| $A_3$ | $A_2$ | $A_1$ | $A_0$ | | |
| 0 | 0 | 0 | 0 | 0 | $D_0 = \overline{A}_3\overline{A}_2\overline{A}_1\overline{A}_0$ |
| 0 | 0 | 0 | 1 | 1 | $D_1 = \overline{A}_3\overline{A}_2\overline{A}_1A_0$ |
| 0 | 0 | 1 | 0 | 2 | $D_2 = \overline{A}_3\overline{A}_2A_1\overline{A}_0$ |
| 0 | 0 | 1 | 1 | 3 | $D_3 = \overline{A}_3\overline{A}_2A_1A_0$ |
| 0 | 1 | 0 | 0 | 4 | $D_4 = \overline{A}_3A_2\overline{A}_1\overline{A}_0$ |
| 0 | 1 | 0 | 1 | 5 | $D_5 = \overline{A}_3A_2\overline{A}_1A_0$ |
| 0 | 1 | 1 | 0 | 6 | $D_6 = \overline{A}_3A_2A_1\overline{A}_0$ |
| 0 | 1 | 1 | 1 | 7 | $D_7 = \overline{A}_3A_2A_1A_0$ |
| 1 | 0 | 0 | 0 | 8 | $D_3 = A_3\overline{A}_2\overline{A}_1\overline{A}_0$ |
| 1 | 0 | 0 | 1 | 9 | $D_3 = A_3\overline{A}_2\overline{A}_1A_0$ |

*Figur 5.20. Blokkdiagram og sannhetstabell for BCD til desimal dekoder.*

Funksjonskodene ($D_0$ - $D_9$) er også angitt. Legg merke til at bare en av utgangene er aktiv, det vil si legges høy, til enhver tid. Det finnes også slike dekodere med inverterte utganger.

## 5.12. Binær/Gray dekoder/koder

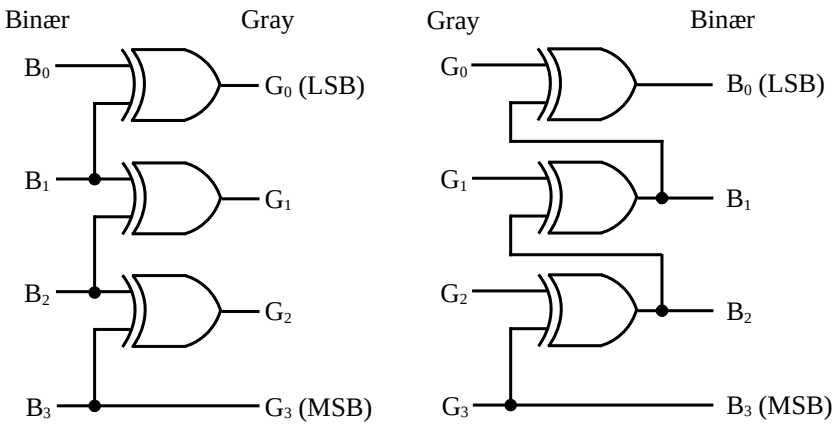
I tabell 5.1 er vist omformingstabellen for omforming mellom binærkode og Gray-kode.

| $B_3$ | $B_2$ | $B_1$ | $B_0$ | $G_3$ | $G_2$ | $G_1$ | $G_0$ |
|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 1 | 0 | 0 | 0 | 1 |
| 0 | 0 | 1 | 0 | 0 | 0 | 1 | 1 |
| 0 | 0 | 1 | 1 | 0 | 0 | 1 | 0 |
| 0 | 1 | 0 | 0 | 0 | 1 | 1 | 0 |
| 0 | 1 | 0 | 1 | 0 | 1 | 1 | 1 |
| 0 | 1 | 1 | 0 | 0 | 1 | 0 | 1 |
| 0 | 1 | 1 | 1 | 0 | 1 | 0 | 0 |
| 1 | 0 | 0 | 0 | 1 | 1 | 0 | 0 |
| 1 | 0 | 0 | 1 | 1 | 1 | 0 | 1 |
| 1 | 0 | 1 | 0 | 1 | 1 | 1 | 1 |
| 1 | 0 | 1 | 1 | 1 | 1 | 1 | 0 |
| 1 | 1 | 0 | 0 | 1 | 0 | 1 | 0 |
| 1 | 1 | 0 | 1 | 1 | 0 | 1 | 1 |
| 1 | 1 | 1 | 0 | 1 | 0 | 0 | 1 |
| 1 | 1 | 1 | 1 | 1 | 0 | 0 | 0 |

*Tabell 5.1. Binær-Gray dekoding.*

Eksklusiv ELLER-porter brukes for omforming mellom binærkode og Gray-kode og vice versa.

I figur 5.21 er vist en 4 bit binær-Gray dekoder og en 4 bit Gray-binær koder. Bemerk at for hvert bit økning i ordlengde benyttes en ekstra Eksklusiv-ELLER-port for både dekoder og koder.

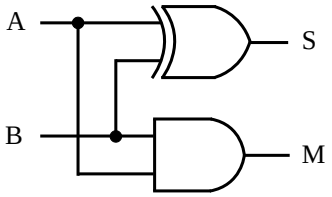



*Figur 5.21. Binær-Gray dekoder og Gray-binær koder.*

## 5.13. Adderere

Når vi adderer to én bits tall A og B, fås resultatet vist i tabellen i figur 5.22, angitt som sum S og mente M. Det ses at summen dannes ved en Eksklusiv-ELLER mellom A og B mens menten dannes ved en OG-funksjon mellom A og B.

| Tall A | Tall B | Sum S | Mente M |
|---|---|---|---|
| 0 | 0 | 0 | 0 |
| 0 | 1 | 1 | 0 |
| 1 | 0 | 1 | 0 |
| 1 | 1 | 0 | 1 |

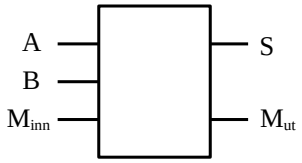



*Figur 5.22. Halv-adderer.*

Til høyre i figuren er vist en logisk realisering av denne addereren, som kalles halv-adderer (half-adder).

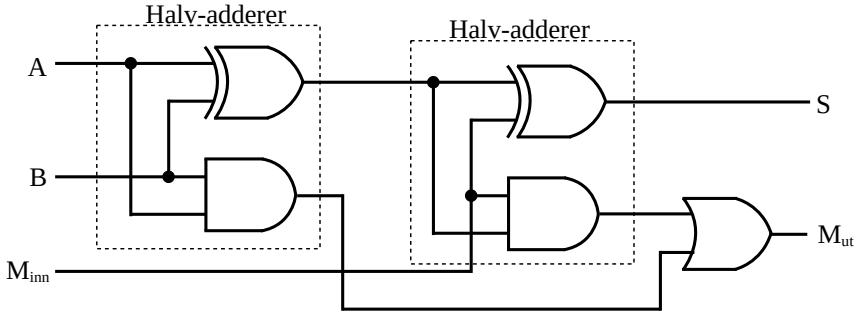
For å muliggjøre utvidelse til mer enn ett bit, må addereren også ha en inngang for mente fra summen av mindre signifikante bit. Dermed fås sannhetstabellen til en hel-adderer (full-adder) som vist i figur 5.23. Det ses at summen nå er Eksklusiv-ELLER mellom de tre inngangene. Et blokkskjema for en hel-adderer er vist i samme figur.

| $M_{inn}$ | A | B | S | $M_{ut}$ |
|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 1 | 0 |
| 0 | 1 | 0 | 1 | 0 |
| 0 | 1 | 1 | 0 | 1 |
| 1 | 0 | 0 | 1 | 0 |
| 1 | 0 | 1 | 0 | 1 |
| 1 | 1 | 0 | 0 | 1 |
| 1 | 1 | 1 | 1 | 1 |



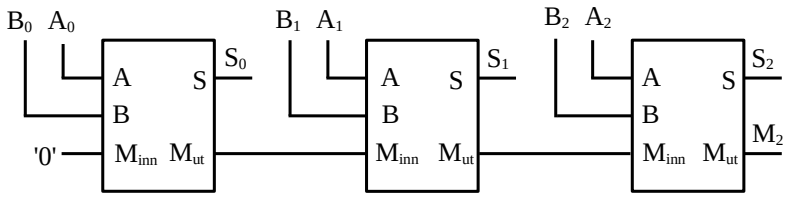

*Figur 5.23. Hel-adderer.*

I figur 5.24 er vist en logisk realisering av hel-addereren. Den er i dette tilfellet bygd opp av to halv-adderere pluss en ELLER-port.




*Figur 5.24. Logisk realisering av hel-adderer.*

Når ordlengden er større enn ett bit, kan flere hel-adderere koples som i figur 5.25. I dette eksemplet er det vist et tilfelle med 3 bit ordlengde. Vi får da en 3 bit rippel-adderer (Ripple-carry adder). Første adderer er koplet som en halv-adderer siden mente inn er lik 0. $A_0$ og $B_0$ representerer minst signifikante bit i henholdsvis ord A og B. $S_0$, $S_1$ og $S_2$ er således summen med $S_0$ som minst signifikante bit. $M_2$ er menten ut etter addisjonen av de to tallene.




*Figur 5.25. 3 bit rippel-adderer.*

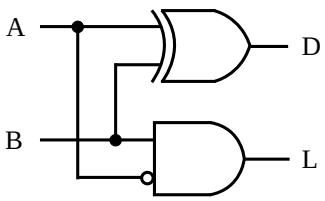
Etterhvert som ordlengden øker, vil transportforsinkelsen øke. For større ordlengder kan denne bli uakseptabel. Det finnes derfor realiseringer som bygger på å kunne predikere menten for å kunne redusere transportforsinkelsen (såkalt Carry Look Ahead Adder).

## 5.14. Subtraktorer

Subtraksjon kan utføres mye på samme måte som addisjon. Vi kan trekke subtrahenden fra minuenden (subtraktoren) slik at vi står igjen med differansen. Når vi subtraherer to én bits tall A og B, fås resultatet vist i tabellen i figur 5.26, angitt som differanse D og lån B.

Det ses at differansen dannes ved en Eksklusiv-ELLER mellom minuenden A og subtrahenden B, mens lånebit dannes ved en OG-funksjon mellom $\overline{A}$ og B. Til høyre i figuren er vist en logisk realisering av denne addereren, som kalles halv-subtraktor. Bemerk at differanseutgangen er den samme som for halv-addereren.

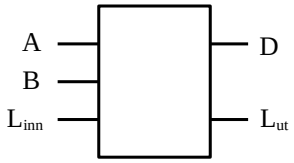
| Minuend A | Subtrahend B | Differanse D | Lån L |
|---|---|---|---|
| 0 | 0 | 0 | 0 |
| 0 | 1 | 1 | 1 |
| 1 | 0 | 1 | 0 |
| 1 | 1 | 0 | 0 |




*Figur 5.26. Halv-subtraktor.*

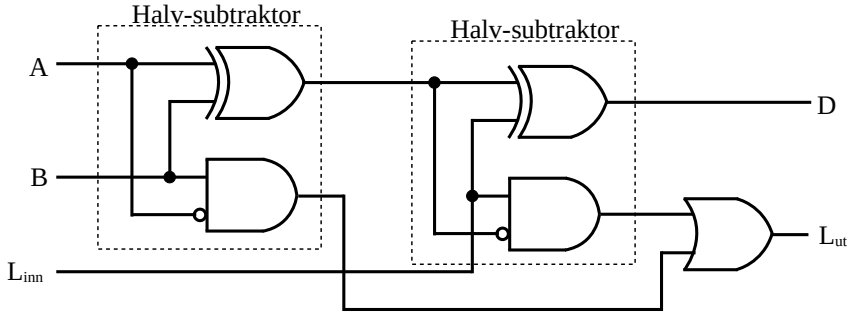
Er subtrahenden større enn minuenden eller det må lånes fra et foregående siffer, må et lån føres til neste mer signifikante siffer. For å muliggjøre utvidelse til mer enn ett bit, må subtraktoren følgelig ha en låne-inngang kalt $L_{inn}$ i sannhetstabellen for en hel-subtraktor som vist i figur 5.27. Et blokkskjema er vist i samme figur.

| $L_{inn}$ | A | B | D | $L_{ut}$ |
|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 1 | 1 |
| 0 | 1 | 0 | 1 | 0 |
| 0 | 1 | 1 | 0 | 0 |
| 1 | 0 | 0 | 1 | 1 |
| 1 | 0 | 1 | 0 | 1 |
| 1 | 1 | 0 | 0 | 0 |
| 1 | 1 | 1 | 1 | 1 |



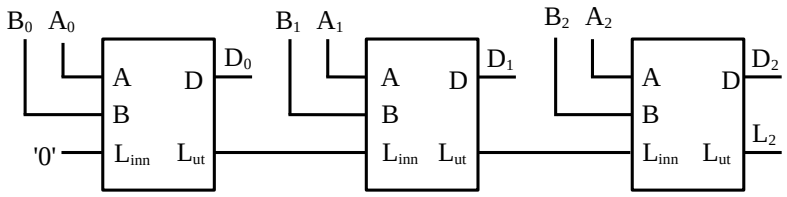

*Figur 5.27. Hel-subtraktor.*

Minuenden er igjen betegnet A mens subtrahenden er betegnet B. Låne-utgangen er benevnt $L_{ut}$. I figur 5.28 er vist en logisk realisering av hel-subtraktoren. Den er i dette tilfellet bygd opp av to halv-subtraktorer pluss en ELLER-port. Bemerk forøvrig at differanseutgangen er den samme som for hel-addereren.




*Figur 5.28. Logisk realisering av hel-subtraktor.*

Når ordlengden er større enn ett bit, kan flere hel-subtraktorer koples som i figur 5.29. I dette eksemplet er det vist et tilfelle med 3 bit ordlengde. Vi får da en 3 bit rippel-subtraktor. Første subtraktor er koplet som en halv-subtraktor siden lånebit inn er lik 0. $A_0$ og $B_0$ representerer minst signifikante bit i henholdsvis minuend A og subtrahend B. $D_0$, $D_1$ og $D_2$ er således differansen med $D_0$ som minst signifikante bit. $L_2$ er lånebit ut etter subtraksjon av de to tallene.



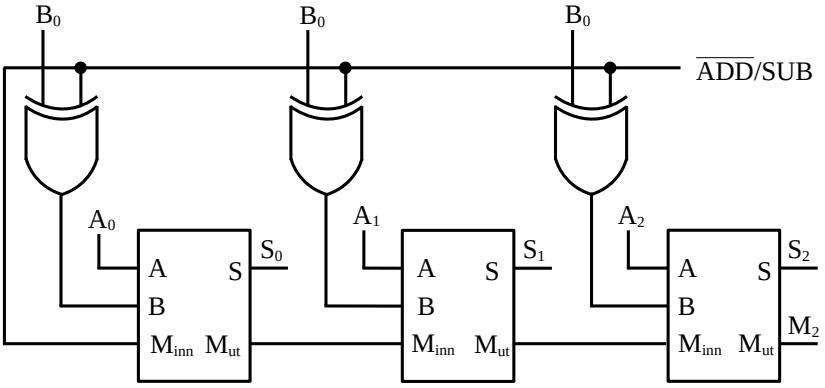

*Figur 5.29. 3 bit rippel-subtraktor.*

Etterhvert som ordlengden øker, vil transportforsinkelsen øke som for hel-addereren. For større ordlengder kan denne bli uakseptabel.

Som nevnt i kapittel 2 er et godt alternativ å realisere subtraksjon på å ta toer-komplement av subtrahenden og addere den til minuenden.

## 5.15. Adderer/Subtraktor

I figur 5.30 er vist en krets som både adderer og subtraherer to trebits tall, der tallet B inverteres når subtraksjon skal utføres. Når ADD er ’1’, utføres addisjon ved at tallet B ikke inverteres og mente inn, $M_{inn}$, legges til ’0’. Når SUB er lik ’1’, utføres subtraksjon ved at det tas enerkomplement av tallet B med Eksklusiv-ELLER-portene. Toerkomplementet av B fås da ved å legge menteinngangen lik ’1’. Sum eller differanse mellom de to tallene A og B fås da med tallet S mens $M_2$ er mentebit ved addisjon og lånebit ved subtraksjon.




*Figur 5.30. Kombinert addisjons- og subtraksjonskrets.*

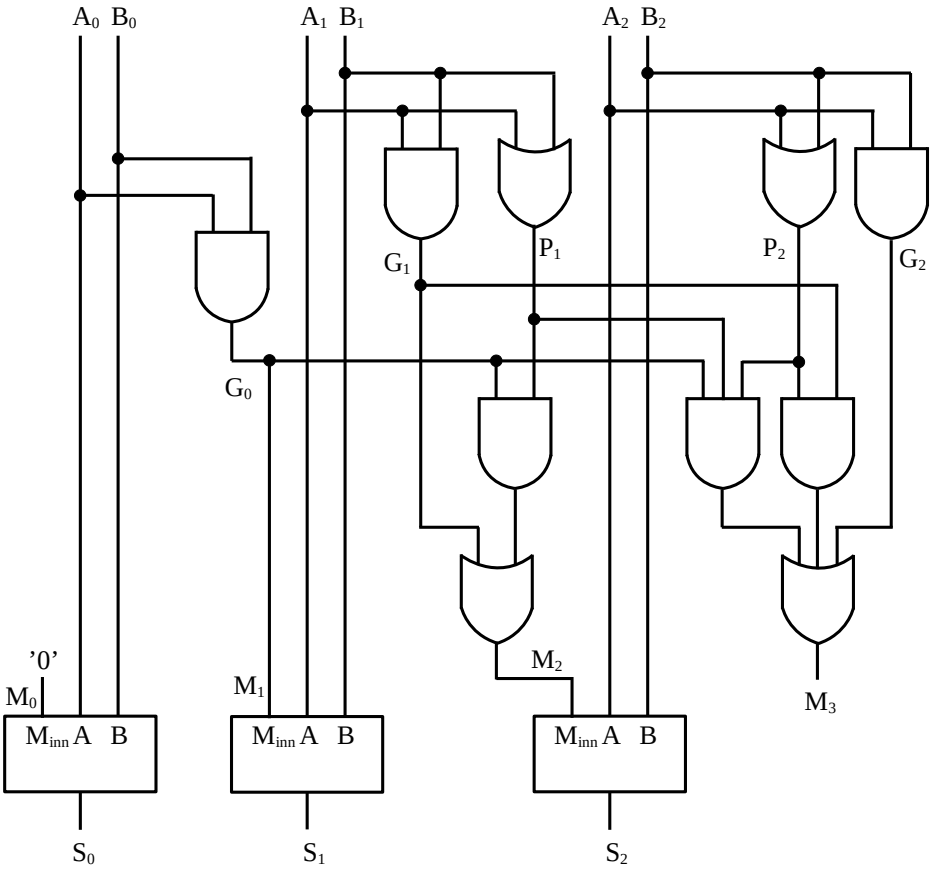
## 5.16. Carry Look Ahead

Rippel-addereren vil medføre for stor forsinkelse ved større ordlengder. Summasjonen av ett og ett bit er forholdsvis rask, men mente fra mindre signifikante bit brukes av mer signifikante bit.

Summasjonen for hver bitposisjon kan følgelig ikke utføres før menten foreligger fra mindre signifikante trinn. Det betyr at forplantningshastigheten er proporsjonal med antall bit i operandene. Dette er lite ønskelig siden det medfører at tiden det tar for summasjonen blir lengre og lengre jo større ordlengden blir.

For å redusere forplantningshastigheten til hvert mentebit, kan det brukes en metode kjent som Carry Look Ahead. Referert til figur 5.31 fås følgende uttrykk for sum $S_i$ og mente ut $M_{i+1}$:

$S_i = \overline{A}_i\overline{B}_iM_i + \overline{A}_iB_i\overline{M}_i + A_i\overline{B}_i\overline{M}_i + A_iB_iM_i$
$M_{i+1} = A_iB_i + A_iM_i + B_iM_i = A_iB_i + (A_i + B_i)M_i$




*Figur 5.31. Carry Look Ahead adderer.*

Mente ut kan da skrives som:

$M_{i+1} = G_i + P_iM_i$ der $G_i = A_iB_i$ og $P_i = A_i + B_i$.

Leddene $G_i$ og $P_i$ kalles henholdsvis genererings- og forplantnings (propagerings)-funksjonen. Med $G_i = 1$ genereres en mente i trinn i. Med $P_i = 1$ forplantes (propageres) en mente gjennom trinn i når $A_i = 1$ eller $B_i = 1$. Legg merke til at $G_i$ og $P_i$ kan dannes ved bare et logisk nivå (OG-funksjon og ELLER-funksjon), se figur 5.31.

Menten $M_1$ ut av trinn 0 (med $A_0$ og $B_0$) er $G_0 + P_0M_0$, men siden $M_0 = 0$, er $M_1 = G_0$. Menten $M_2$ ut av trinn 1 (med $A_1$ og $B_1$) er $G_1 + P_1M_1$. Siden $M_1 = G_0$ kan denne skrives som $M_2 = G_1 + P_1G_0$. Menten $M_3$ ut av trinn 2 (med $A_2$ og $B_2$) er $G_2 + P_2M_2$. Siden $M_2 = G_1 + P_1G_0$ kan denne skrives som $M_3 = G_2 + P_2G_1 + P_2P_1G_0$.

Uten videre regning er det lett å se at for fire bit (en nibbel) må menten $M_4$ ut av trinn 3 (med $A_3$ og $B_3$) være lik $M_4 = G_3 + P_3G_2 + P_3P_2G_1 + P_3P_2P_1G_0$.

Bemerk at vi fortsatt har forsinkelsen gjennom hel-addererne som for rippel-addereren, men mentekjeden er brutt opp til å kreve en portforsinkelse for $G_i$ og $P_i$ og to flere portforsinkelser for å generere $M_{i+1}$. Følgelig legges det til tre portforsinkelser, men mentekjeden fjernes.

Dersom vi antar at hver hel-adderer har en portforsinkelse på to, vil en fire bit Carry Look-ahead-adderer ha en maksimal portforsinkelse på fem, mens en 4 bit rippel-adderer vil ha en maksimal portforsinkelse på åtte.

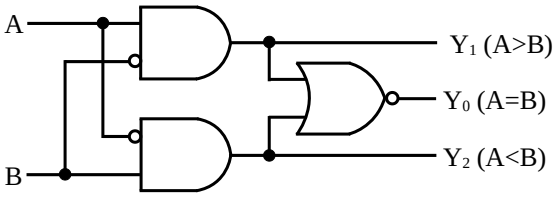
Denne forskjellen øker i favør av Carry Lookahead-addereren for større ordlengder. I praksis er det imidlertid vanskelig å realisere Carry Lookahead-adderere for mer enn åtte bit (en byte) på grunn av fan-in-begrensninger.

To åtte bit Carry Lookahead-adderere kan imidlertid kombineres med noe ekstra kombinatorikk til å realisere 16 bit adderer som er adskillig raskere enn en tilsvarende 16 bit rippel-adderer.

Til slutt skal nevnes at det finnes flere algoritmer for rask addisjon, Kogge-Stone-addereren og Brent-Kung-addereren er to eksempler.

## 5.17. Komparatorer

Som nevnt kan Eksklusiv-NOR-porten realisere en én-bits komparator. Vi kan utvide funksjonen til også å finne om et tall er større eller mindre enn et annet. Da får vi det logiske skjemaet vist i figur 5.32.




*Figur 5.32. 1 bit komparator.*

Utgangen $Y_0$ (A=B) er da gitt som:

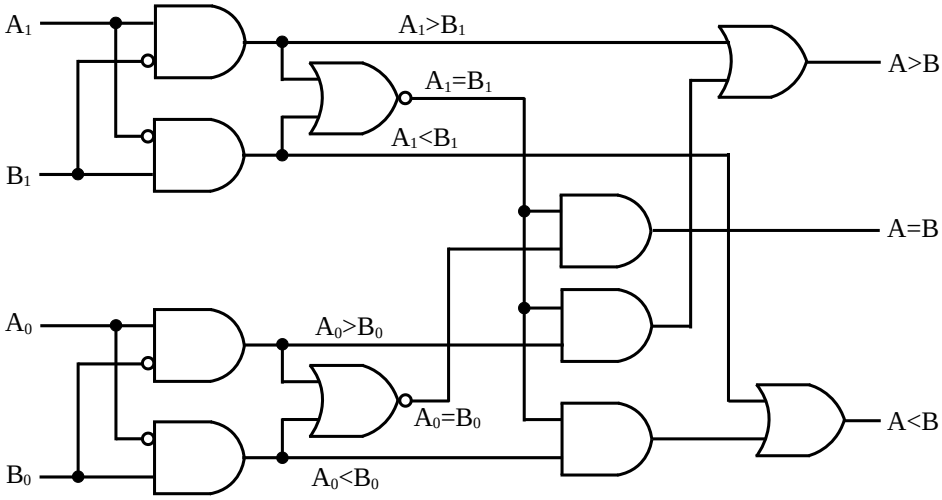
$$Y_0 = A \cdot B + \overline{A} \cdot \overline{B} = \overline{A \oplus B} \tag{5.4}$$

Utgangen $Y_1$ (A>B) er gitt som:

$$Y_1 = A \cdot \overline{B} \tag{5.5}$$

Utgangen $Y_2$ (A<B) er da gitt som:

$$Y_2 = \overline{A} \cdot B \tag{5.6}$$

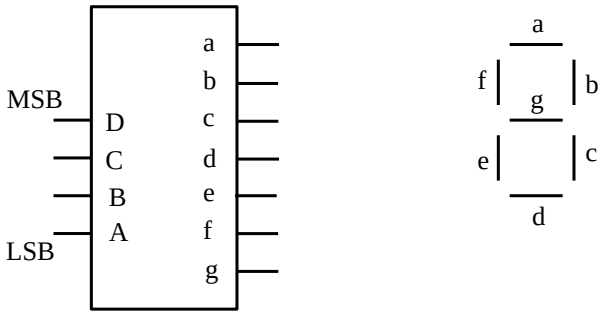



*Figur 5.33. 2 bit komparator.*

Vi kan utvide til flere bit ved gjentatt bruk av denne komparatoren. I figur 5.33 er vist et eksempel for to bit ordlengde. 4 bit komparatorer finnes som ferdig krets. Ved å ha ekstra innganger for A>B, A=B og A>B kan to eller flere komparatorer kaskadekoples for større ordlengde.

## 5.18. BCD til 7-segment dekoder

Et blokkskjema og et 7-segment display med angivelse av de forskjellige segmentene er vist i figur 5.34.




*Figur 5.34. 7-segment dekoder og 7-segment display.*

En BCD til 7-segment dekoder vil drive de tilhørende segmentene i henhold til BCD-koden på inngangen. Dette er vist i tabell 5.2.

| D | C | B | A | a | b | c | d | e | f | g |
|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 0 |
| 0 | 0 | 0 | 1 | 0 | 1 | 1 | 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 0 | 1 | 1 | 0 | 1 | 1 | 0 | 1 |
| 0 | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 0 | 0 | 1 |
| 0 | 1 | 0 | 0 | 0 | 1 | 1 | 0 | 0 | 1 | 1 |
| 0 | 1 | 0 | 1 | 1 | 0 | 1 | 1 | 0 | 1 | 1 |
| 0 | 1 | 1 | 0 | 1 | 0 | 1 | 1 | 1 | 1 | 1 |
| 0 | 1 | 1 | 1 | 1 | 1 | 1 | 0 | 0 | 0 | 0 |
| 1 | 0 | 0 | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| 1 | 0 | 0 | 1 | 1 | 1 | 1 | 1 | 0 | 1 | 1 |
| 1 | 0 | 1 | 0 - | | | | | | | |
| 1 | 1 | 1 | 1 | - | - | - | - | - | - | - |

*Tabell 5.2.*

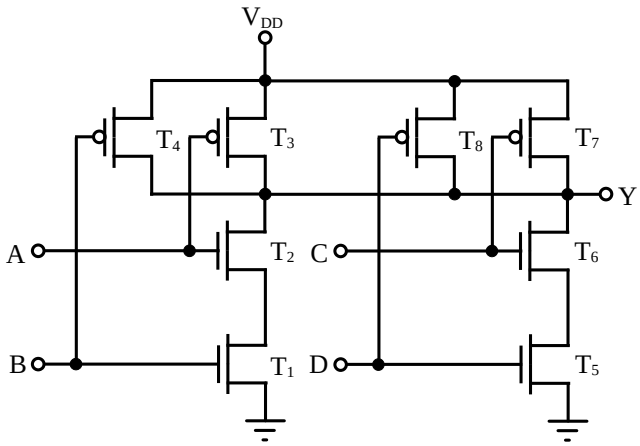
*BCD til 7-segment dekoder.*

Legg merke til at koden 1010-1111 ikke er tillatt, denne kan da velges vilkårlig. I noen kommersielt tilgjengelige dekodere brukes disse kodene for å angi annet enn et siffer for displayet. Det mest vanlige er kanskje å vise E (for Error).

# 5.19. CMOS-Realisering

## 5.19.1. OG-NOR-port

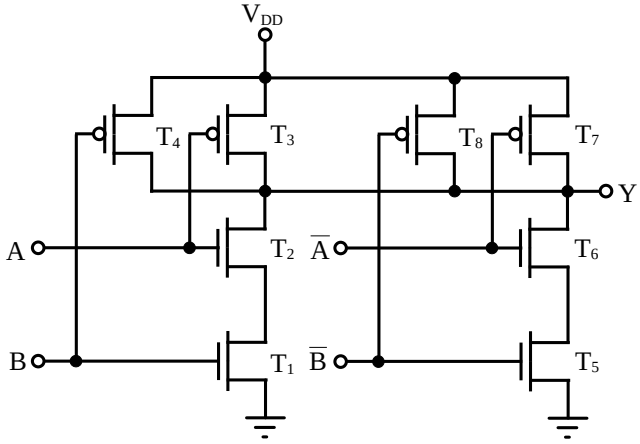
OG-NOR-porten i figur 5.1 kan realiseres i CMOS ved å kople sammen utgangene fra to NAND-porter til utgangen $Y = \overline{AB+CD}$, se figur 5.35. OG-ELLER-port fås med Y invertert.




*Figur 5.35. CMOS OG-NOR-port.*

## 5.19.2. Eksklusiv ELLER

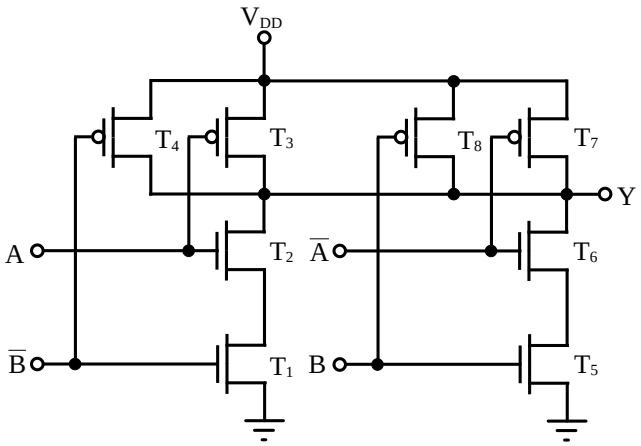
I figur 5.36 er vist CMOS-realisering av en Eksklusiv ELLER-port. Denne realiseringen er da egentlig et spesialtilfelle av OG-NOR-porten i figur 5.35.




*Figur 5.36. Eksklusiv ELLER realisering.*

### 5.19.3. Eksklusiv NOR

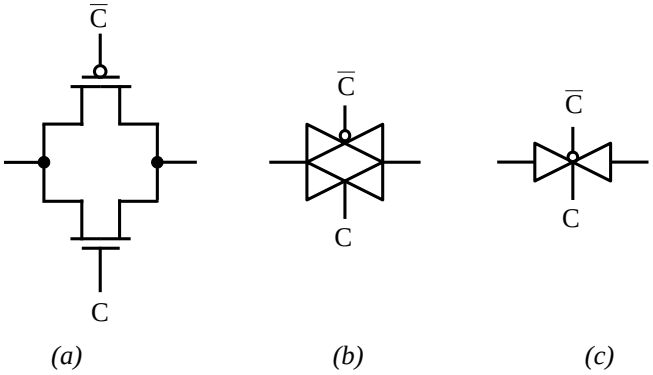
Eksklusiv NOR kan realiseres ved å invertere utgangen i figur 5.36. Imidlertid kan den realiseres med like mange transistorer ved å benytte løsningen vist i figur 5.37. Denne realiseringen er egentlig også et spesialtilfeller av OG-NOR-porten i figur 5.35.




*Figur 5.37. Eksklusiv NOR realisering.*

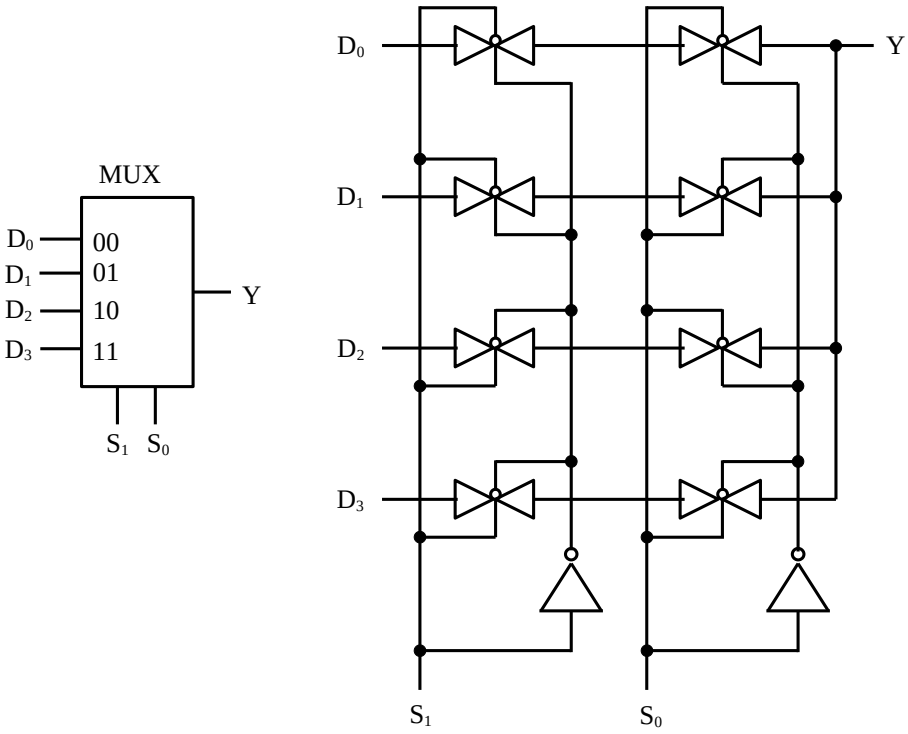
### 5.19.4. Multipleksere

I figur 5.38a er vist transmisjonsporten fra kapittel 3 med mest brukte symboler i figur 5.38b og c. Denne kan brukes til å realisere multipleksere.




*Figur 5.38. Transmisjonsport med transistorer (a ) og symbol (b og c).*

Siden transmisjonsporten har høy impedans når transistorene er slått av, kan utgangene koples sammen. Dette er vist i figur 5.39 med en CMOS-realisering av multiplekseren i figur 5.9.




*Figur 5.39. 4-1 multiplekser med transmisjonsporter.*

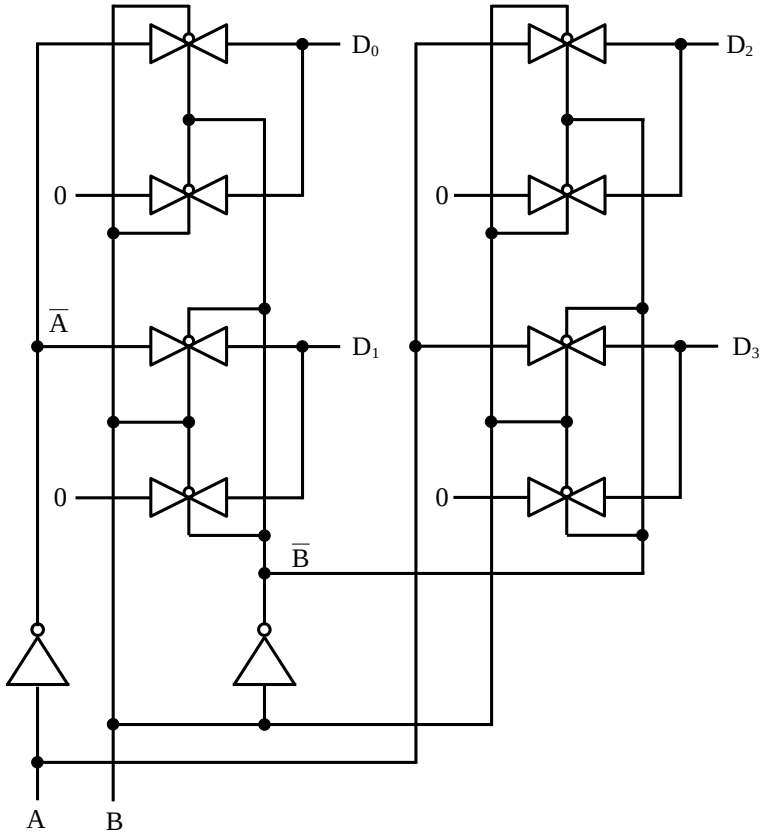
Multiplekseren i figur 5.9 realisert ved hjelp av logiske porter har et antall transistorer på 36 (med fem NAND-porter og to inverterere).

For realiseringen med transmisjonsporter er antall transistorer nå bare 20 (med 4 transistorer til inverterere). At antall transistorer kan reduseres der transmisjonsporter kan benyttes, gjelder forøvrig også andre CMOS-realiseringer.

## 5.19.5. 2-4 dekoder

I figur 5.17 er vist blokkdiagram og sannhetstabell for 2-4 dekoder. I figur 5.40 er vist dekoderen realisert med transmisjonsporter for E = 1. Dersom E også skal være inngang, kreves et tillegg på 4 transmisjonsporter.

Bemerk forøvrig at det benyttes en ekstra transmisjonsport for hver utgang. Dette gjøres for å legge hver utgang til logisk 0 i stedet for til høy impedans når utgangen ikke skal være lik logisk 1.



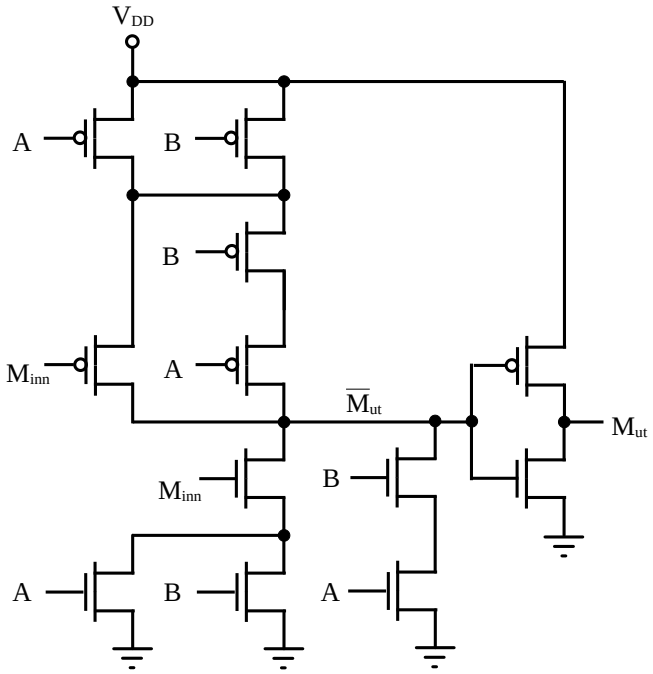

*Figur 5.40. 2-4 dekoder med transmisjonsporter.*

## 5.19.6. Hel-adderer

I figur 5.23 er vist sannhetstabellen for hel-addereren. Det ses at mente ut kan skrives som:

$M_{ut} = (A+B)\cdot M_{inn} + A\cdot B\cdot M_{inn}$

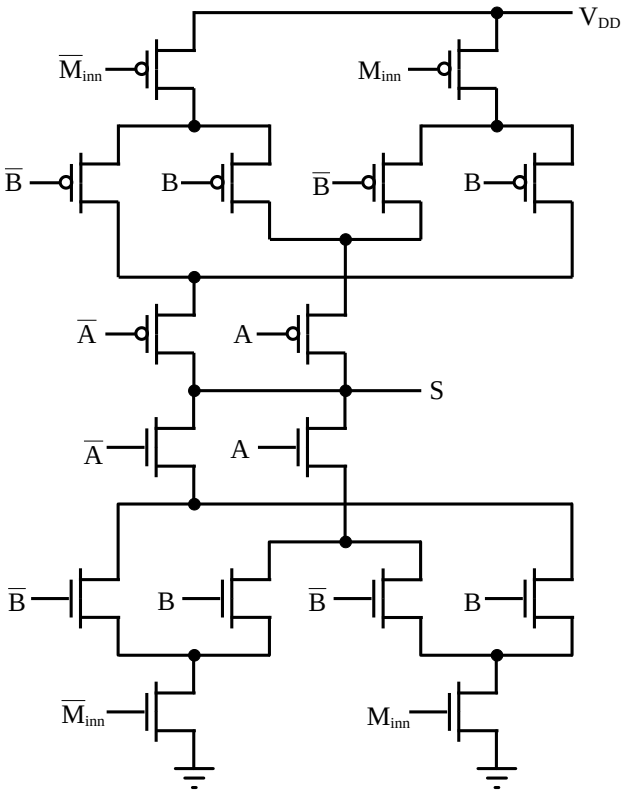
Dette uttrykket kan realiseres som vist i figur 5.41 ved først å realisere leddene $(A+B)\cdot M_{inn}$ og $A\cdot B\cdot M_{inn}$ for så å ta NOR-funksjonen av disse før den endelige inverteringen gir $M_{ut}$.




*Figur 5.41. Generering av mente ut for hel-adderer.*

Sum-utgangen for hel-addereren kan for eksempel realiseres ved å la A og B være innganger til en eksklusiv-ELLER-port, realisert for eksempel som i figur 5.36. Utgangen fra denne kan så føres til den ene inngangen til en ny eksklusiv-ELLER-port, der mente inn utgjør den andre inngangen, se figur 5.24.

En mer elegant løsning er å realisere en Eksklusiv ELLER med 3 innganger A, B og $M_{inn}$. Dette er vist i figur 5.42. Denne løsningen gir adskillig færre transistorer enn ved




*Figur 5.42. Generering av sum for hel-adderer.*

# Kapittel 6
# Låsekretser og vipper

## 6.1. Innledning

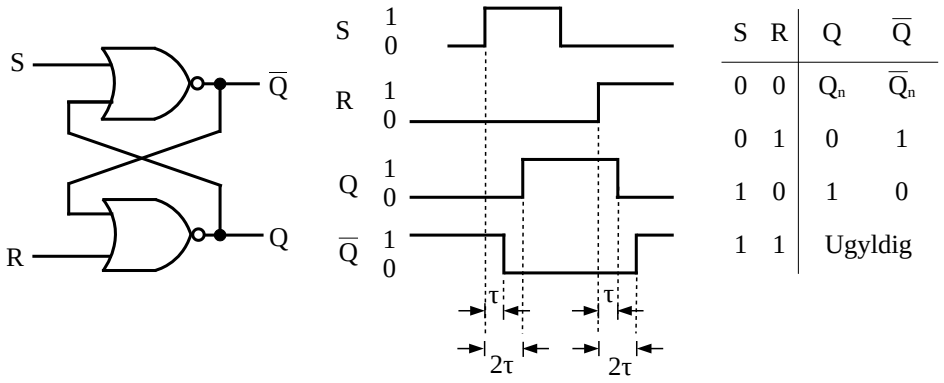
Vi har tidligere sett på kombinatoriske funksjoner. Da er utgangen bare avhengig av kombinasjonen av gjeldende innganger. Tidligere tilstander til inngangene har ingen betydning for hva som foreligger på utgangen.

Imidlertid er det mange tilfeller der kretsene må ha «minne», det vil si huske tidligere inngangstilstander. Da vil utgangstilstanden også være et resultat av hva som tidligere forelå på inngangene. Kretser der utgangen ikke bare er avhengig av nåværende tilstand, men også av tidligere tilstander, kalles sekvensielle.

Ved sekvensiell logikk er følgelig ny utgangstilstand definert av nåværende tilstand i tillegg til tilstanden til inngangene. Dette er den største forskjellen i forhold til kombinatoriske funksjoner. Dette er meget viktig når vi skal lage minnekretser og kontrollkretser. Sekvensiell logikk realiseres ved hjelp av låsekretser, vipper og (eventuelt også) kombinatorisk logikk.

## 6.2. SR-Lås (SR Latch)

En SR-lås realisert ved hjelp av to NOR-porter er vist i figur 6.1. Betegnelsene S og R står for henholdsvis Sett og Resett. Ved S=1 resettes utgangen $\overline{Q}$ til 0 mens utgangen Q settes til 1. Tilsvarende ses at R=1 resetter Q til 0 mens utgangen $\overline{Q}$ settes til 1. Vi benytter betegnelsen $Q_n$ for å angi at vippen blir stående i samme tilstand (uten endring).



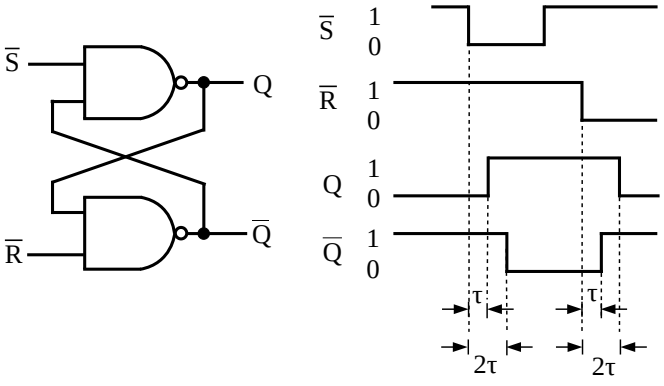

| S | R | Q | $\overline{Q}$ |
|---|---|---|---|
| 0 | 0 | $Q_n$ | $\overline{Q}_n$ |
| 0 | 1 | 0 | 1 |
| 1 | 0 | 1 | 0 |
| 1 | 1 | Ugyldig | |

*Figur 6.1. NOR-port SR-lås med tidsdiagram og sannhetstabell.*

Ideelt sett skal utgangene Q og $\overline{Q}$ være inverterte, men på grunn av tidsforsinkelsene ses at det er et lite tidsintervall der begge disse utgangene er 0 samtidig. Som tidligere nevnt tar det en viss tid for signalet å forplante seg fra inngang til utgang. Dette er vist i figur 6.1, der vi har kalt denne forsinkelsen for τ. I figuren er det også antatt at det tar like lang tid om utgangen på NOR-porten skal gå fra 0 til 1 som fra 1 til 0.

I sannhetstabellen til høyre i figuren ses at kombinasjonen av S=1 og R=1 samtidig ikke er tillatt. Denne kombinasjonen gir begge utgangene lik 0 samtidig. Kombinasjonen er heller ikke stabil. Det fremgår av sannhetstabellen at en SR-lås kan brukes som en ett bit minneelement ved først å la R eller S være lik 1 for så å sette begge lik 0.

Det finnes flere måter å realisere en SR-lås på. Figur 6.2 viser en realisering med NAND-porter som medfører at S- og R-inngangene er aktivt lave. Sannhetstabellen er den samme som vist i figur 6.1.



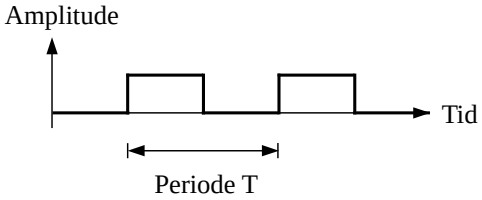

*Figur 6.2. NAND-port SR-lås med tidsdiagram.*

I figur 6.3 er vist logiske logiske symboler for SR-lås med aktiv høy inngang (til venstre) og aktiv lav inngang (til høyre).


*Figur 6.3. Logiske symboler for SR-lås og $\overline{S}$ $\overline{R}$-lås.*

## 6.3. SR-vippe (SR Flip Flop)

For å synkronisere flere hendelser er det ofte benyttet et klokkesignal, se figur 6.4. vanligvis vil amplituden være like lenge lav og høy for klokkesignalet.

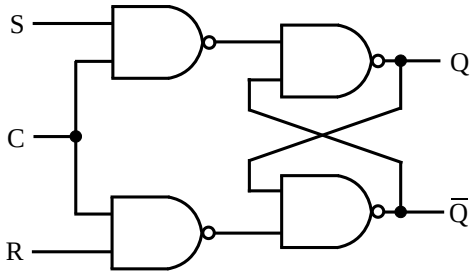



*Figur 6.4. Klokkesignal.*

Klokkefrekvensen $f_C$ er det inverse av klokkeperioden T:

$$f_C = \frac{1}{T} \tag{6.1}$$

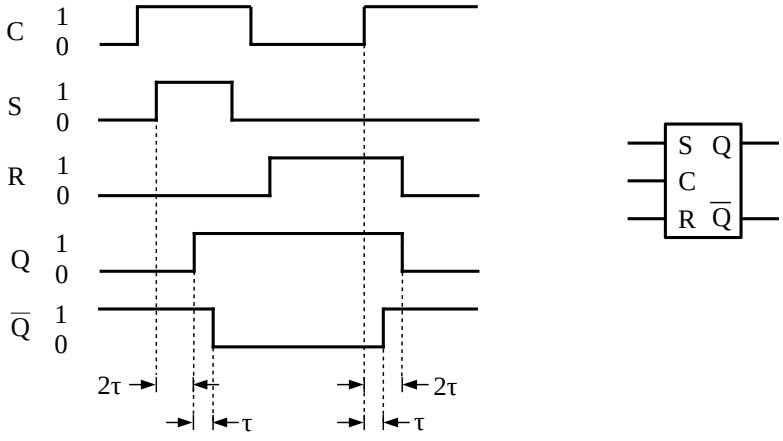
Det kan gjøres bruk av en klokkeinngang på en SR-lås. Da får vi SR-vippen vist i figur 6.5. Av sannhetstabellen ses at klokkeinngangen C må være 1 for at S og R skal henholdsvis sette og resette vippen. Vi benytter - (bindestrek) i de tilfeller det ikke spiller noen rolle om inngangen settes til 0 eller 1.




| C | S | R | Q | $\overline{Q}$ |
|---|---|---|---|---|
| 0 | - | - | $Q_n$ | $\overline{Q}_n$ |
| 1 | 0 | 0 | $Q_n$ | $\overline{Q}_n$ |
| 1 | 0 | 1 | 0 | 1 |
| 1 | 1 | 0 | 1 | 0 |
| 1 | 1 | 1 | Ugyldig | |

*Figur 6.5. SR-vippe (med klokkeinngang C).*

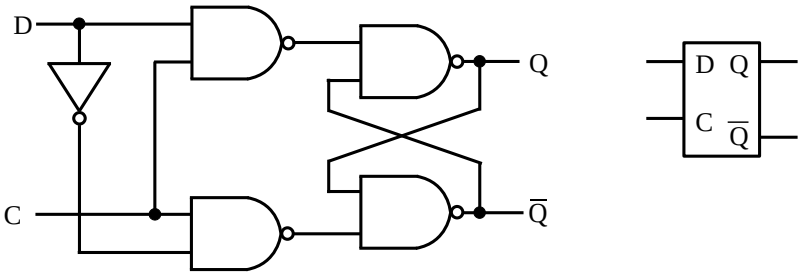
Tidsdiagrammet er vist i figur 6.6. Det ses blant annet at selv om R går til 1 vil det ikke skje noe før C blir 1. I samme figur er vist logisk symbol for denne SR-vippen.




*Figur 6.6. SR-vippe: Tidsdiagram (til venstre) og logisk symbol (til høyre).*

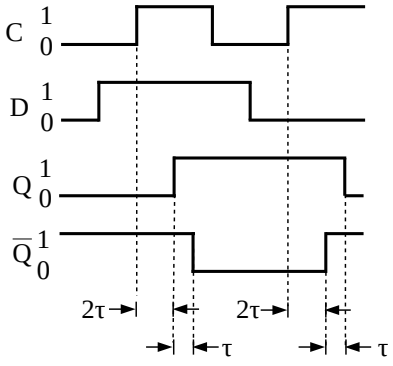
## 6.4. D-vippe (D Flip Flop)

En ulempe med SR-vippen er at vi for å lagre en 0 eller 1, må bruke forskjellige innganger (R eller S) avhengig av hva vi ønsker å lagre. Et alternativ er D-vippen der vi tilfører 0 eller 1 på bare en inngang: datainngangen D. Denne vippen er vist i figur 6.7 med symbol.




*Figur 6.7. D-vippe med tidsdiagram og symbol.*

Fra figuren ses at modifiseringen av SR-vippen bare består i å bruke en inverterer fra S-inngangen til R-inngangen i SR-vippen. Tidsdiagrammet og sannhetstabellen er vist i figur 6.8. Som for SR-vippen er det bare når klokken C er lik 1 at endringer kan skje. Når C er 1, vil Q settes til 1 ellet 0 når D er henholdsvis 1 eller 0. Vi benytter - (bindestrek) i de tilfeller det ikke spiller noen rolle om inngangen settes til 0 eller 1.

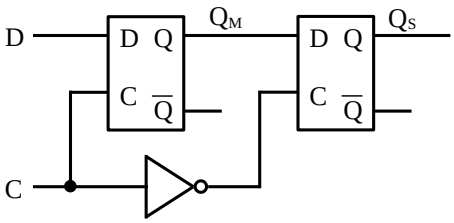



| C | D | Q | $\overline{Q}$ |
|---|---|---|---|
| 0 | - | $Q_n$ | $\overline{Q}_n$ |
| 1 | 0 | 0 | 1 |
| 1 | 1 | 1 | 0 |

*Figur 6.8. Tidsdiagram og sannhetstabell for D-vippe.*

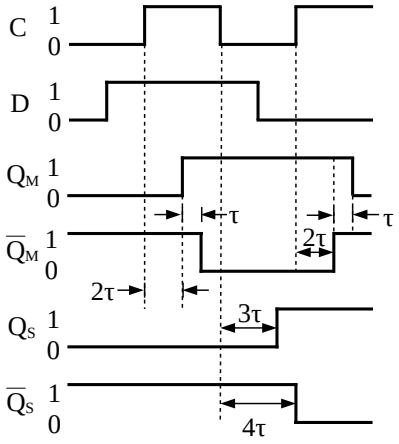
## 6.5. Master Slave D-vippe

For å gardere seg mot at D-vippen skifter tilstand mer enn én gang når klokkesignalet er høyt, kan to vipper, for eksempel som i figur 6.7, koples i kaskade: som «herre og slave». En slik kopling er vist i figur 6.9.




*Figur 6.9. Master Slave D-vippe*

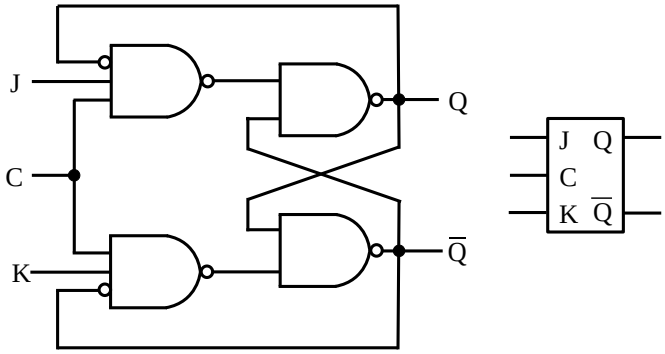
Master-vippen (med utgang $Q_M$) kan skifte tilstand når klokken C = 1 mens slave-vippen inntar tilstanden til master-vippen når C = 0, se figur 6.10. Følgelig må C gå høy og så lav før inngangen D er klokket videre til slaveutgangen $Q_S$.




*Figur 6.10. Tidsdiagram for Master Slave D-vippe.*

## 6.6. JK-vippe (JK Flip Flop)

Som nevnt var en ulempe med SR-vippen at de to inngangene ikke kunne være 1 samtidig. En JK-vippe tillater dette. Dette oppnås ved å innføre ekstra tilbakekopling fra utgangen til inngangen. Realisering av en enkel JK-vippe med symbol er vist i figur 6.11.




| J | K | Q | $\overline{Q}$ |
|---|---|---|---|
| 0 | 0 | $Q_n$ | $\overline{Q}_n$ |
| 0 | 1 | 0 | 1 |
| 1 | 0 | 1 | 0 |
| 1 | 1 | Toggler | |

*Figur 6.11. JK-vippe med symbol og sannhetstabell.*

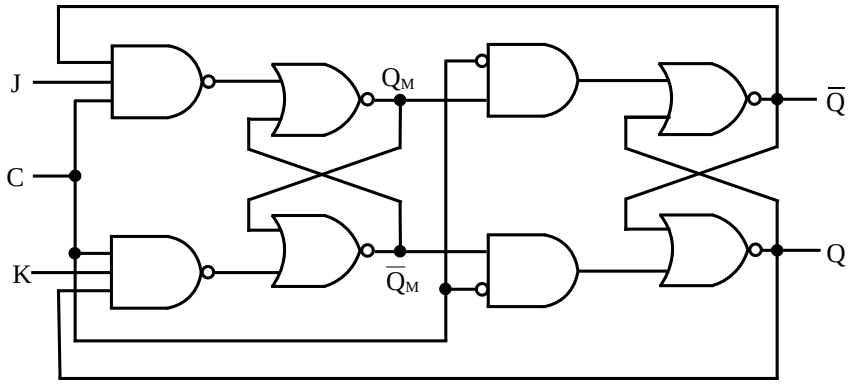
Dersom klokken C = 0, ses at et eventuelt nytt påtrykk på J- og K-inngangen ikke vil føre til endring av utgangene, det vil si $Q = Q_n$ (og $\overline{Q} = \overline{Q}_n$). Til høyre i figur 6.11 er vist sannhetstabellen når klokken C = 1. Det ses at den følger sannhetstabellen for SR-vippen med unntak av at begge inngangene er lik 1 samtidig. I dette tilfellet vil vippen toggle, det betyr at den vil skifte fra 0 til 1 hvis den står i 0, eller fra 1 til 0 hvis den står i 1. Med J- og K-inngangen koplet permanent til 1 får vi også en vippe som da kalles T-vippe. Denne vil da skifte (toggle) for hver klokkepuls med T = J = K = 1.

## 6.7. Master Slave JK-vippe

Et problem med JK-vippen i forrige avsnitt kan oppstå dersom J = K = 1 samtidig som klokken C = 1. Vippen kan da skifte (toggle) flere ganger. Dette problemet kan løses ved hjelp av en Master Slave JK-vippe. Prinsippet er det samme som for Master Slave D-vippen. En Master Slave JK-vippe er vist i figur 6.12.

Når C = 1 vil slave-vippen være sperret slik at utgangene Q og $\overline{Q}$ ikke endres. Samtidig vil master-vippen (med utgang $Q_M$) settes til ønsket tilstand, avhengig av innholdet på J- og K-inngangene.

Når C = 0 vil slave-vippen settes til verdien i master-vippen mens denne vil være sperret for eventuell endring på J- og K-inngangen. Følgelig må C gå høy og så lav før det eventuelt skjer en endring, gitt av innholdet på J- og K-inngangen, på utgangene Q og $\overline{Q}$.

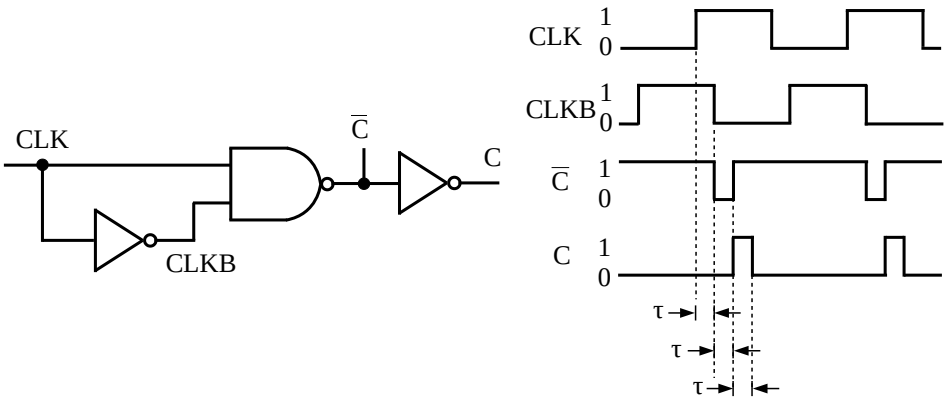



*Figur 6.12. Master Slave JK-vippe.*

## 6.8. Generering av klokkesignal

Det kan oppstå et problem med Master Slave-vippen i forrige avsnitt: Hvis en inngang er høy mens klokken er høy, og dersom inngangen er i ferd med å skifte tilstand, vil vippen «se» en 1 som om det var en gyldig inngang.

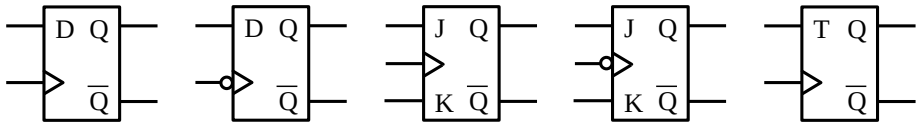
Dette problemet kan unngås ved å bruke et klokkesignal som varer bare en kort tid. Et eksempel på generering av et slikt klokkesignal er vist i figur 6.13.




*Figur 6.13. Generering av klokkepuls.*

Fra figuren fremgår at klokkesignalet CLK er forsinket med tiden τ og tilføres en NAND-port sammen med det opprinnelige klokkesignalet CLK. Det betyr at vi etter tiden τ får en kort puls (fra 1 til 0) som varer tiden τ. Etter tiden τ får vi så en positiv puls (fra 0 til 1) som også varer tiden τ. Det er antatt at transportforsinkelsen er like lang for inverterere og NAND-porten. Det er benyttet én inverterer. For lengre pulser kan for eksempel tre inverterere benyttes.

Vanligvis benyttes en trekant som symbol for klokkeinngangen for vipper som benytter denne korte klokkepulsen. Vi sier at vippene klokkes på oppadgående flanke når klokkesignalet C benyttes, mens de klokkes på nedadgående flanke når $\overline{C}$ benyttes. I sistnevnte tilfelle er dette markert med en ring foran trekanten. I figur 6.14 er vist symboler for flanketriggede data-, JK- og T-vippe. Sistnevnte er en JK-vippe der J- og K-inngangen er bundet sammen. Dette betyr at T-vippen vil skifte (toggle) for hver positivgående klokkepuls når T = 1.




*Figur 6.14. Symboler for flanketrigget data-, JK- og T-vippe.*

I figur 6.15 er vist sannhetstabeller for data- og JK-vippe som trigges på oppadgående klokkeflanke. Bemerk at en eventuell endring på utgangene bare skjer ved denne transisjonen. Sannhetstabellene er tilsvarende for nedadgående flanker på klokken. Vi benytter - (bindestrek) i de tilfeller det ikke spiller noen rolle om inngangen settes til 0 eller 1.

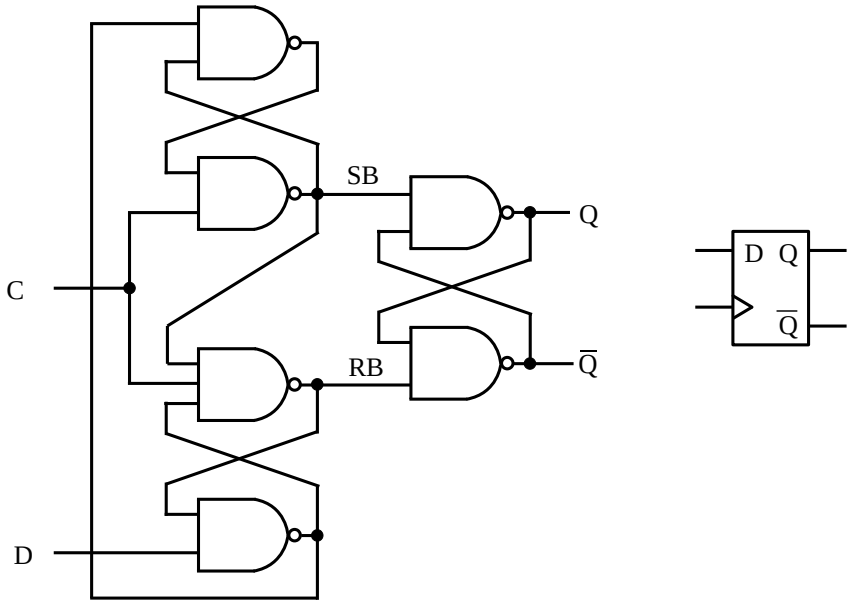
| Klokke | D | Q | $\overline{Q}$ |
|---|---|---|---|
| 0/1 | - | $Q_n$ | $\overline{Q}_n$ |
| ↑ | 0 | 0 | 1 |
| ↑ | 1 | 1 | 0 |

| Klokke | J | K | Q | $\overline{Q}$ |
|---|---|---|---|---|
| 0/1 | - | - | $Q_n$ | $\overline{Q}_n$ |
| ↑ | 0 | 0 | $Q_n$ | $\overline{Q}_n$ |
| ↑ | 0 | 1 | 0 | 1 |
| ↑ | 1 | 0 | 1 | 0 |

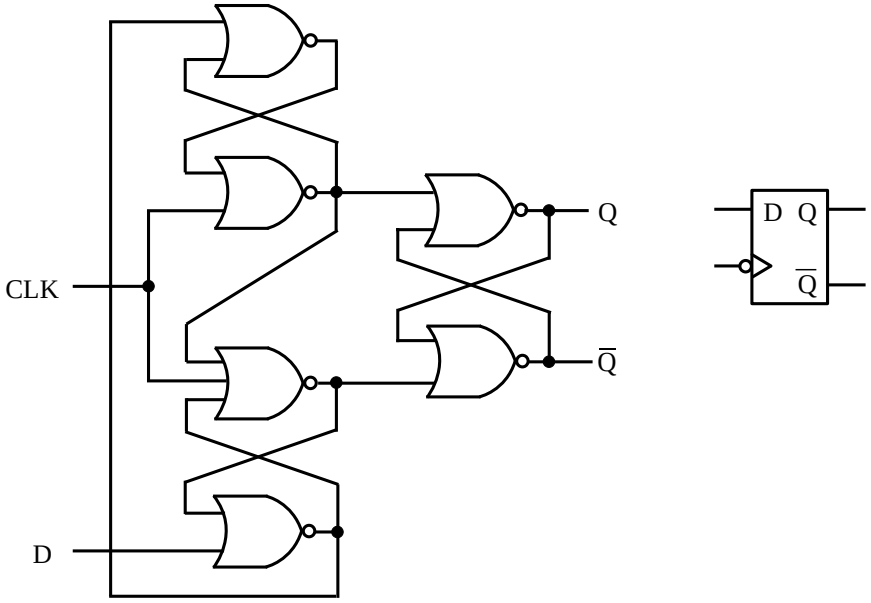
*Figur 6.15. Sannhetstabell for flanketrigget data- og JK-vippe.*

En alternativ metode for å få til flanketrigging for en D-vippe er vist i figur 6.16 (med symbol til høyre). Når klokkesignalet C er lavt, vil SB = RB = 1, og utgangslåsen (med utganger Q og $\overline{Q}$) vil ikke forandre verdi. Samtidig kan datainngangen D forandres uten at utgangene påvirkes. Når klokken går høy, vil bare verdien som topp- og bunn-låsen har, når dette skjer, påvirke utgangslåsen. Når klokken er høy, vil ikke endring i D påvirke utgangslåsen.

Ved å bruke NOR-porter kan en D-vippe som trigger på nedadgående flanke designes, se figur 6.17 (med symbol til høyre). Virkemåten er forøvrig som i vippen i figur 6.16.



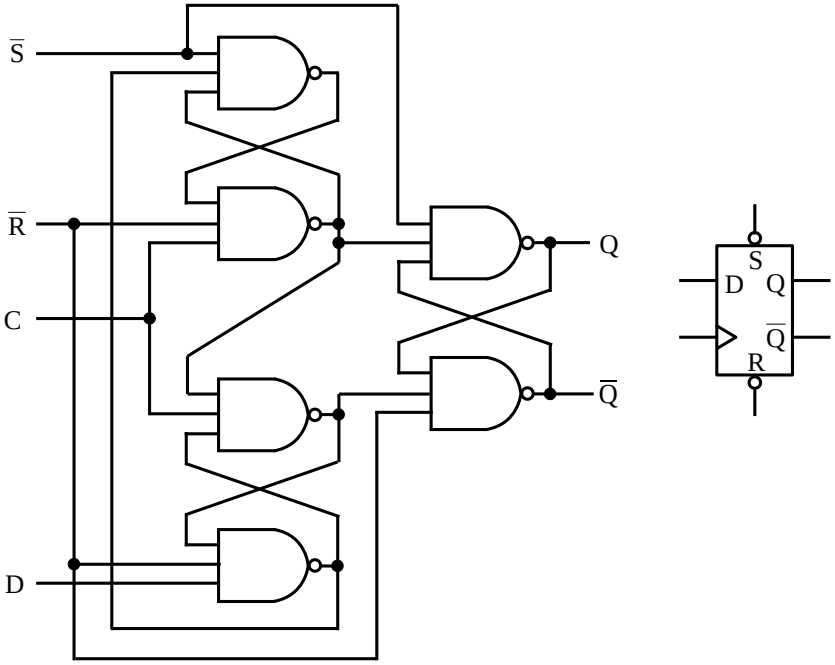

*Figur 6.16. Oppadgående flanketrigget D-vippe.*




*Figur 6.17. Nedadgående flanketrigget D-vippe.*

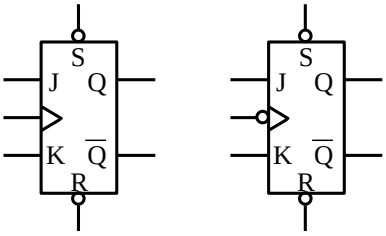
## 6.9. Asynkron setting og resetting

Mange D-vipper og JK-vipper har egen inngang for setting og/eller resetting. D-vippen i figur 6.16 kan modifiseres som vist i figur 6.18 (med symbol til høyre). Det ses at dersom $\overline{S}$-inngangen er lav, vil Q legges høy og $\overline{Q}$ legges lav, uavhengig av klokke C. Tilsvarende ses at dersom $\overline{R}$-inngangen er lav, vil Q legges lav og $\overline{Q}$ legges høy, uavhengig av klokke.




*Figur 6.18. Flanketrigget D-vippe med asynkron setting og resetting.*

Også JK-vipper kan ha slike asynkrone sett- og resett-innganger. Symbolene for positivt- og negativtgående klokke er vist i figur 6.19.




*Figur 6.19. JK-vipper med asynkron setting og resetting.*

I figur 6.20 er vist sannhetstabeller for data- og JK-vippe med asynkrone sett- og resett-innganger som trigges på oppadgående klokkeflanke. Bemerk at de aktivt lave sett- og resett-inngangene overstyrer klokkeinngangen. Sannhetstabellene er tilsvarende for disse vippene med asynkrone sett- og resett-innganger som trigges på nedadgående flanker på klokken. Bemerk forøvrig at sett- og resett-inngangene ikke kan være aktive samtidig. Vi benytter - (bindestrek) i de tilfeller det ikke spiller noen rolle om inngangen settes til 0 eller 1.

| Klokke | D | $\overline{R}$ | $\overline{S}$ | Q | $\overline{Q}$ |
|---|---|---|---|---|---|
| - | - | 0 | 1 | 0 | 1 |
| - | - | 1 | 0 | 1 | 0 |
| 0/1 | - | 1 | 1 | $Q_n$ | $\overline{Q}_n$ |
| ↑ | 0 | 1 | 1 | 0 | 1 |
| ↑ | 1 | 1 | 1 | 1 | 0 |

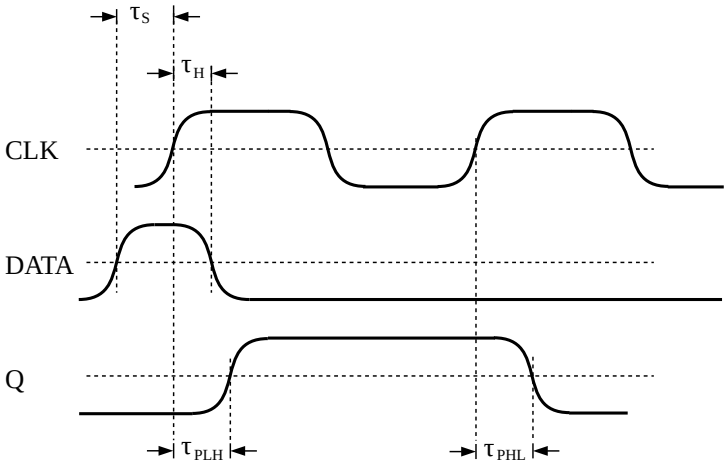
| Klokke | $\overline{R}$ | $\overline{S}$ | J | K | Q | $\overline{Q}$ |
|---|---|---|---|---|---|---|
| - | 0 | 1 | - | - | 0 | 1 |
| - | 1 | 0 | - | - | 1 | 0 |
| 0/1 | 1 | 1 | - | - | $Q_n$ | $\overline{Q}_n$ |
| ↑ | 1 | 1 | 0 | 0 | $Q_n$ | $\overline{Q}_n$ |
| ↑ | 1 | 1 | 0 | 1 | 0 | 1 |
| ↑ | 1 | 1 | 1 | 0 | 1 | 0 |

*Figur 6.20. Sannhetstabell for flanketrigget data- og JK-vippe med asynkron sett og resett.*

## 6.10. Tidskarakteristika for vipper

For vipper det viktig å å ta høyde for tiden før og etter en klokketransisjon, positivtgående eller negativtgående. Vi skal her begrense oss til D-vipper, men det følgende gjelder også for flanketriggede JK-vipper. Dersom to innganger, for eksempel data og klokke eller klokke og resett, endrer seg samtidig, sies tilstanden å være metastabil. Dette betyr at datautgangen ikke er forutsigbar og kan ta en lang tid for å stabilisere seg til en fast verdi. Oscillasjoner kan også forekomme.

Metastabilitet kan unngås ved å forsikre seg om at data, klokke, resett og sett-inngangene er gyldige og med konstant verdi for en nærmere spesifisert tid. For data og klokke er dette spesifisert som tiden før og etter klokketransisjonen, kalt henholdsvis oppsettings-tid (Setup time) og holde-tid (Hold time), som vist i figur 6.21.




*Figur 6.21. Tidskarakteristika for D-vippe.*

Oppsettings-tiden, $\tau_S$, er minimum tid som data må være stabil før klokketransisjonen. Holde-tiden, $\tau_H$, er minimum tid som data må være stabil etter klokketransisjonen. Med andre ord må data være stabil og gyldig i tiden $\tau_S + \tau_H$. Hvor lang oppsettings- og holde-tiden er, avhenger av hvordan vippen er bygd opp. Det er mulig å lage vipper med $\tau_H = 0$, eller for den slags skyld $\tau_S = 0$, men åpningstiden $\tau_S + \tau_H$ kan ikke bli null. Vanligvis vil disse tidene ligge i området rundt 1 ns for rimelig hurtige vipper.

Transportforsinkelsen i vipper regnes som tiden fra klokketransisjonen skjer til Q-utgangen endres som vist i figuren. Som for porter opereres også her med $\tau_{PLH}$ og $\tau_{PHL}$. Disse kan være ulike, og verdiene til disse ligger normalt i ns-området.

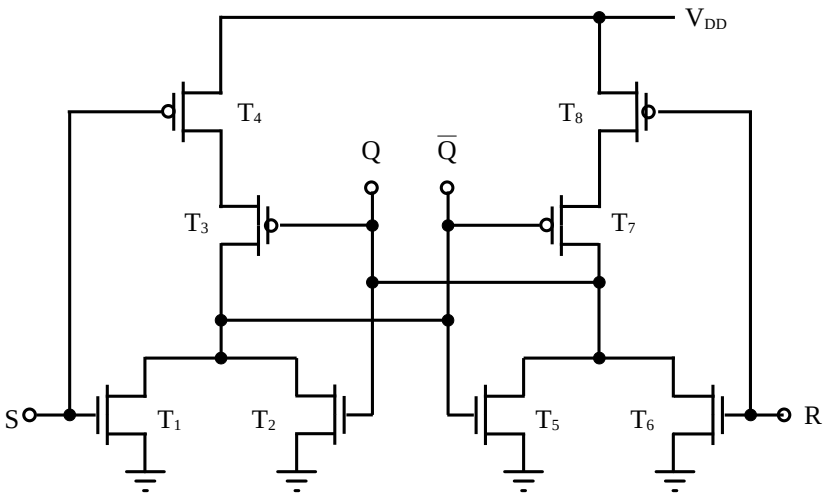
Til slutt skal nevnes at karakteristika presentert her for D-vipper også gjelder for JK-vipper. For asynkrone sett- og resett-innganger gjelder at disse ikke må skje slik at metastabilitet opptrer. Ellers vil transportforsinkelsen fra disse inngangene til Q-utgangen være forskjellig fra transportforsinkelsen etter klokketransisjonen.

## 6.11. CMOS-realisering

### 6.11.1. SR-lås med NOR-porter

Et eksempel på en CMOS-realisering av en SR-lås med NOR-porter, der S- og R-inngangene er aktivt høye (figur 6.1), er vist i figur 6.22.

Det ses at $T_1$-$T_4$ og $T_5$-$T_8$ utgjør de to NOR-portene. Dersom S går høy, vil $T_1$ slås på og $\overline{Q}$ gå lav. Siden R må være lav, er både $T_5$ og $T_6$ av. Da vil $T_7$ og $T_8$ være på og følgelig er Q høy. Dersom R går høy og S er lav, vil de to NOR-portene bytte rolle og $\overline{Q}$ gå høy og Q gå lav.



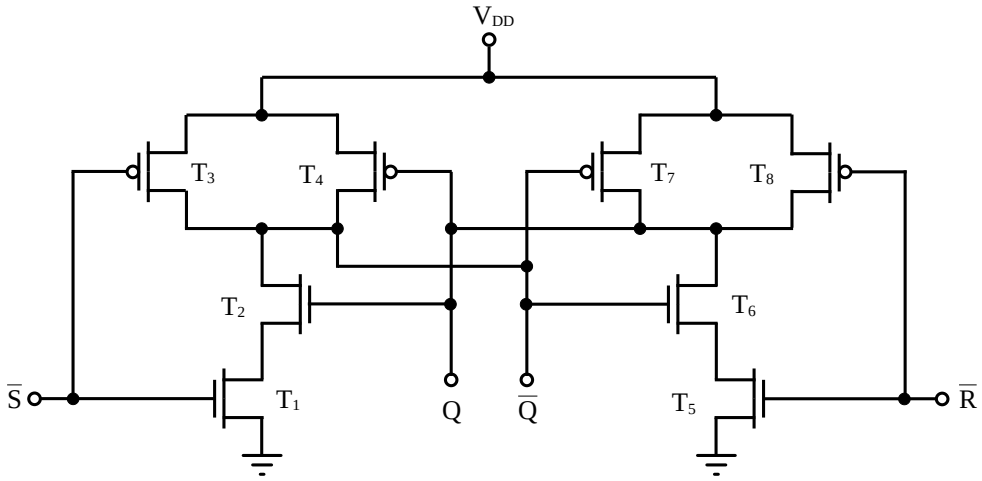

*Figur 6.22. SR-lås med NOR-porter.*

Dersom Q er høy, vil S lav etterpå sørge for at $T_1$ slås av. Men siden $T_7$ og $T_8$ fortsatt er på, vil fortsatt Q være høy og $\overline{Q}$ lav siden $T_2$ er på. Det fås heller ingen endring av utgangene dersom R legges lav etter at $\overline{Q}$ er lagt høy. Dersom derimot R og S legges høy samtidig, vil både Q og $\overline{Q}$ bli lav, og det fås en ustabil og ikke tillatt tilstand.

## 6.11.2. SR-lås med NAND-porter

Et eksempel på en CMOS-realisering av en SR-lås med NAND-porter, der S- og R-inngangene er aktivt lave, er vist i figur 6.23. Det ses at $T_1$-$T_4$ og $T_5$-$T_8$ utgjør de to NAND-portene.

Dersom $\overline{S}$ går lav, vil $T_1$ slås av, $T_3$ slås på og Q gå høy. Siden $\overline{R}$ må være høy, er både $T_5$ og $T_6$ på. Da vil $T_7$ og $T_8$ være av og følgelig er $\overline{Q}$ lav. Dersom $\overline{R}$ går lav og $\overline{S}$ er høy, vil de to NAND-portene bytte rolle og $\overline{Q}$ går høy mens Q går lav.



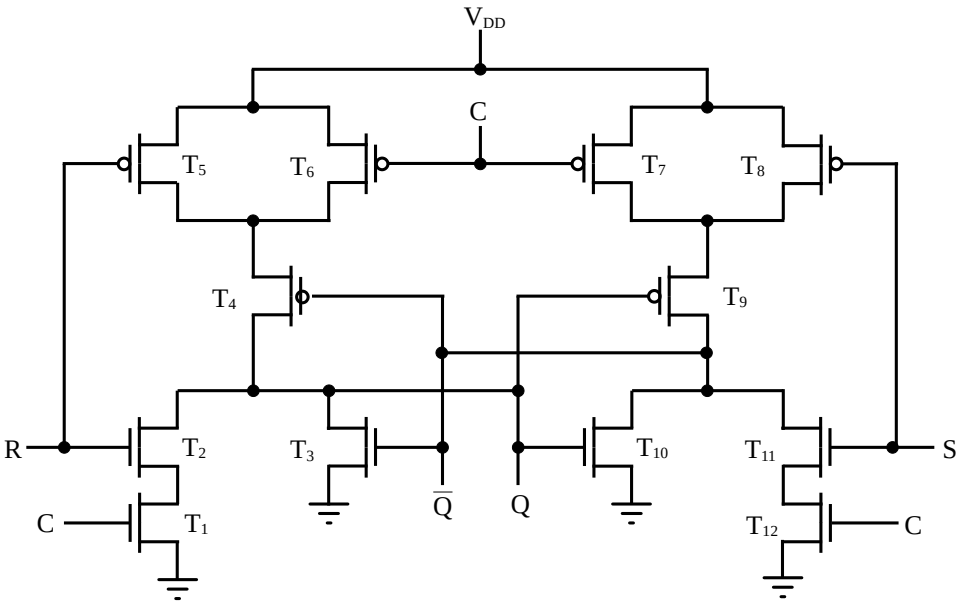

*Figur 6.23. SR-lås med NAND-porter.*

Dersom Q er lagt høy ved å la $\overline{S}$ være lav (mens $\overline{R}$ er høy), vil $\overline{S}$ høy etterpå sørge for at $T_4$ fortsatt er på (mens $T_1$-$T_3$ er av). Q vil da fortsatt være høy og $\overline{Q}$ lav (siden $T_5$ og $T_6$ er på). Det fås heller ingen endring av utgangene dersom $\overline{R}$ legges høy etter at $\overline{Q}$ er lagt høy. Dersom $\overline{R}$ og $\overline{S}$ derimot legges lav samtidig, vil både Q og $\overline{Q}$ bli høy, og det fås en ustabil og ikke tillatt tilstand.

## 6.11.3. SR-vippe

Et eksempel på en CMOS-realisering av en SR-vippe med klokke C (figur 6.5), er vist i figur 6.24. Når klokken C er lav, er transistorene $T_1$, $T_2$, $T_{11}$ og $T_{12}$ av samtidig som transistorene $T_6$ og $T_7$ er på. Dette sikrer at utgangene Q og $\overline{Q}$ forblir i deres tilstand. Dersom Q = 1 vil da $T_4$ være på mens $T_9$ vil være av. Dersom Q = 0 vil da $T_4$ være av mens $T_9$ vil være på.

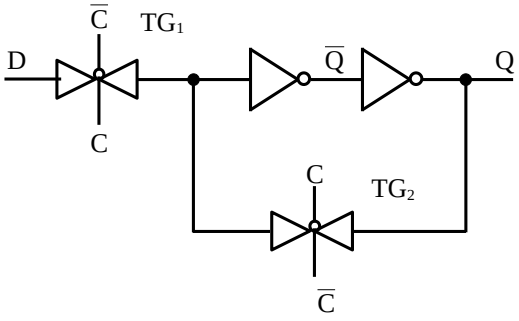
Når klokken C er høy, er transistorene $T_6$ og $T_7$ av samtidig som transistorene $T_1$ og $T_{12}$ er på. Vi får da kretsen i figur 6.22 slik at kretsen ganske enkelt være en CMOS NOR lås som responderer på hva som legges på inngangene S og R.




*Figur 6.24. SR-vippe med klokkeinngang C.*

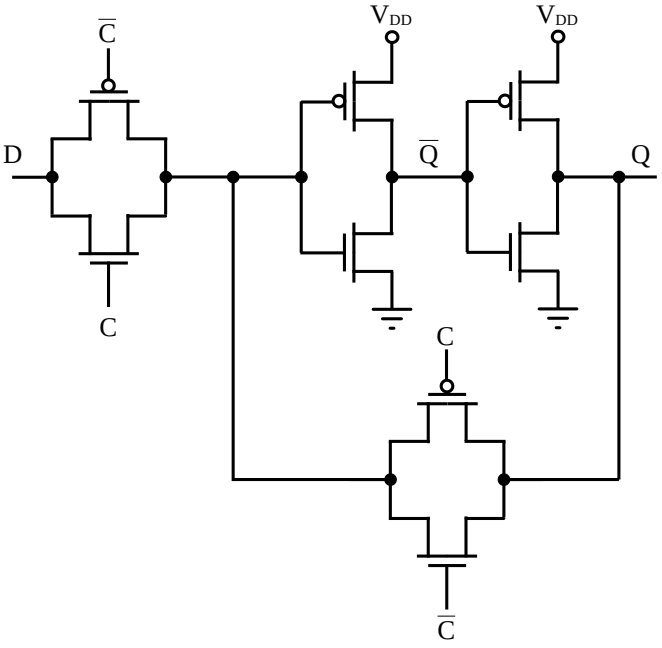
## 6.11.4. D-vippe

Et eksempel på en CMOS-realisering av en D-vippe med klokke C (figur 6.7) kan realiseres ved hjelp av transmisjonsporter som vist i figur 6.25. Datainngangen er som tidligere benevnt D. Dersom C = 0, vil D ikke influere på utgangen Q og transmisjonsporten $TG_2$ sørger for at utgangen holdes i uendret tilstand.



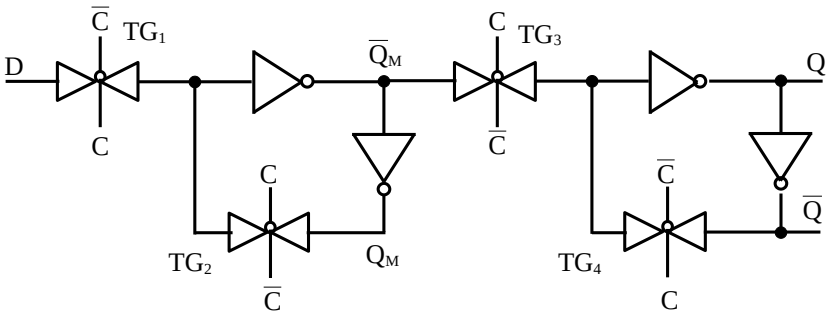

*Figur 6.25. D-vippe med transmisjonsporter.*

Dersom C = 1, vil transmisjonsporten $TG_1$ sørge for at innholdet på D føres til utgangen Q samtidig som transmisjonsporten $TG_2$ legges i høy impedans og ikke påvirker denne transporten. I figur 6.26 er figur 6.25 vist med transistorer. Det benyttes åtte transistorer. I tillegg kommer 1-2 transistorer for klokke. Følgelig er realisering ved bruk av transmisjonsporter mer effektiv med hensyn på transistorantall.




*Figur 6.26. D-vippe med transmisjonsporter, vist med transistorer.*

I figur 6.27 er vist en realisering av en Master Slave D-vippe (figur 6.9) med bruk av transmisjonsporter. Legg merke til at klokkene til transmisjonsportene $TG_1/TG_2$ og $TG_3/TG_4$ er koplet i motfase. Dette gjør at når C = 1 føres D til masterutgangen ($\overline{Q}_M/Q_M$), men ikke til Q-utgangen. Når C = 0 føres data på masterutgangen $\overline{Q}_M$ videre til Q-utgangen. Igjen ses at realiseringen med bruk av transmisjonsporter er mer effektiv med hensyn på transistorantall.




*Figur 6.27. Master Slave D-vippe med transmisjonsporter.*

Dersom det ønskes asynkrone setting- og resetting-innganger, kan dette forholdsvis enkelt realiseres ved å kombinere transmisjonsportene (og inverterne) med for eksempel NAND-porter.

Til slutt skal nevnes at også JK-vipper egner seg godt for CMOS-realisering med transmisjons-porter.

# Kapittel 7
# Skiftregistre

## 7.1. Innledning

Ved sekvensiell logikk er, som tidligere nevnt, utgangstilstanden ikke bare avhengig av nåværende tilstand til inngangene, men også avhengig av tidligere inngangstilstander. Skiftregistre og tellere er den største gruppen som realiserer slik type logikk.

Skiftregistre kan realiseres ved at vipper koples i kaskade slik at utgangen fra en vippe blir inngangen til neste vippe. Vi får da et serielt skiftregister. Skiftingen skjer da ved at vippene har en felles klokke som sørger for skiftingen. Vippene kan også settes eller resettes samtidig.

Et firetrinns serielt skiftregister sørger således for en forsinkelse på en klokkeperiode for hvert klokkeskift, og det skal da fire klokkeperioder til for at data inn skiftes ut.

Skiftregistre realiseres normalt ved hjelp av datavipper eller JK-vipper, men SR-vipper benyttes også.

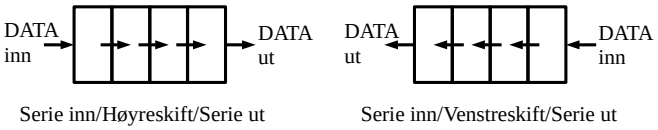
Skiftregistre kan for eksempel brukes til seriell dataoverføring over lengre eller kortere avstander. Da kan disse på sendersiden benyttes for å omgjøre data på parallell form til seriell form. Data sendes så serielt, og på mottakersiden benyttes så skiftregistre for å omforme fra seriell form til parallell form. Seriell datakommunikasjon bruker således raske høyhastighets overføringsmedier (for eksempel koaksial- eller fiber-kabel) istedenfor mange parallelle lavhastighets kabler.

Ved seriell dataoverføring over kortere avstander kan skiftregistre også brukes for å redusere antallet signalledninger. Typiske eksempler er her å overføre data inn og ut av analog/digital-omformere, skjerm-drivere, minnekretser og mikrokontrollere.

Skiftregistre kan klassifiseres i følgende typer:

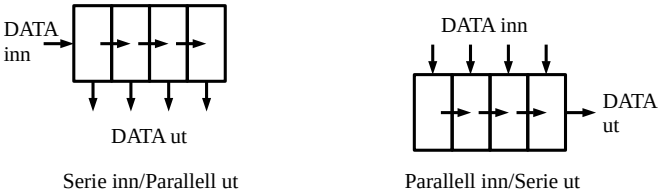
- Serie inn/serie ut
- Serie inn /parallell ut
- Parallell inn/serie ut
- Parallell inn/parallell ut
- Tellere og mønstergeneratorer

Vi snakker også om skifting til høyre og skifting til venstre. I mikrokontrollere er slike skiftregistre spesielt nyttige. I figur 7.1 er vist et fire bit serie inn/serie ut skiftregister på blokkskjematisk form. Til venstre i figuren skjer skiftingen til høyre, mens venstreskiftingen er vist til høyre i figuren.




*Figur 7.1. Serie inn /Serie ut skiftregister.*

I figur 7.2 er på blokkskjematisk form vist et fire bit skiftregister til venstre som realiserer serie inn/parallell ut. Til høyre er vist på blokkskjematisk form et 4 bit skiftregister som realiserer parallell inn/serie ut.

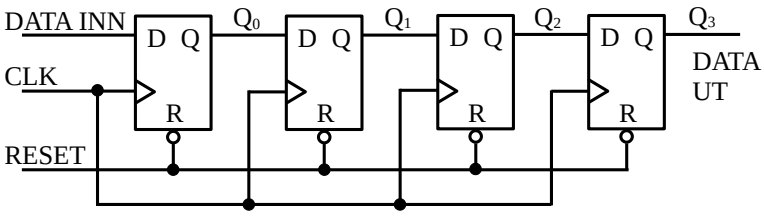



*Figur 7.2. Serie/parallell skiftregister.*

Det finnes flere varianter som realiserer kombinasjoner av disse skiftregistrene. Vi skal her se på noen av dem.

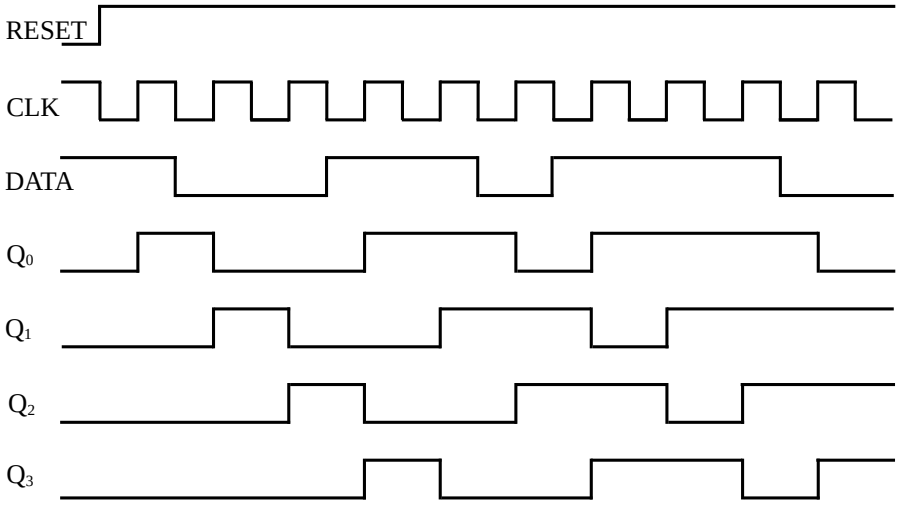
## 7.2. Serie inn/serie ut skiftregister

I figur 7.3 er vist et fire bit serie inn/serie ut skiftregister realisert ved hjelp av datavipper. Alle vippene klokkes med den samme klokken CLK. Data inn tilføres første vippe mens data ut foreligger på den siste vippen. Det er også vist hvordan en felles asynkron resetting kan gjøres. Dette er opsjonelt og har ikke noe med selve skiftingen å gjøre.




*Figur 7.3. Serie inn/Serie ut skiftregister.*

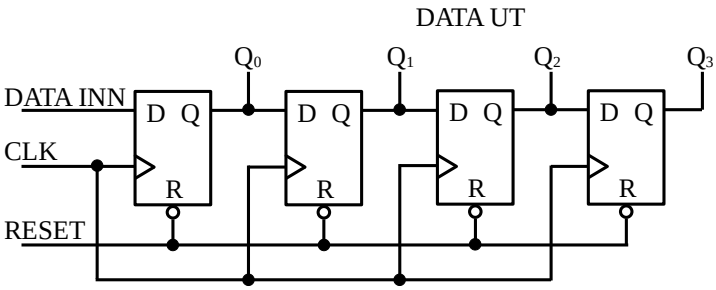
Tidsdiagrammet er vist i figur 7.4. Alle vippene er resatt til å begynne med. Det ses at data på inngangen skiftes fra inngangsvippen til den foreligger på utgangsvippen fire klokkeperioder senere.




*Figur 7.4. Tidsdiagram for serie inn/serie ut og serie inn/parallell ut skiftregister.*

## 7.3. Serie inn/parallell ut skiftregister

I figur 7.5 er vist realiseringen av et fire bit serie inn/parallell ut skiftregister. Det ses at oppbyggingen i prinsippet er det samme som serie inn /serie ut skiftregisteret i figur 7.3. Forskjellen er at alle vippeutgangene er tilgjengelig. Tidsforløpet blir da som vist i figur 7.4.

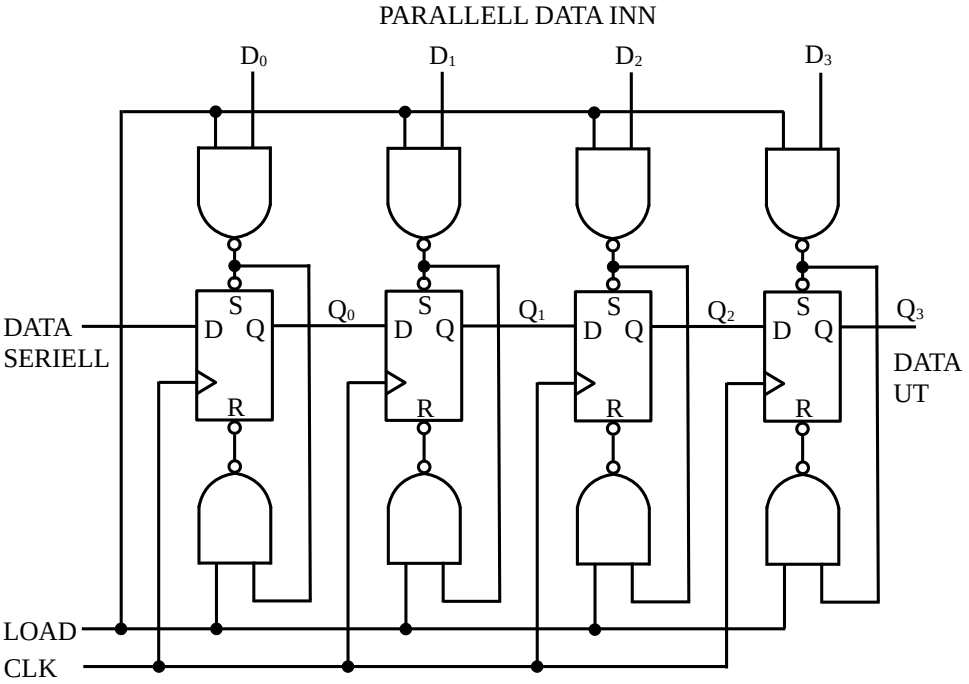



*Figur 7.5. Serie inn /parallell ut skiftregister.*

## 7.4. Parallell inn /serie ut skiftregister

I figur 7.6 er vist realiseringen av et fire bit parallell inn/serie ut skiftregister. Når LOAD = 1 vil data lastes inn til de enkelte vippene. Dette skjer uavhengig av klokken, slik at vippene settes/resettes asynkront. Legg merke til at for det enkelte databit lik 0 vil vippens resett-inngang (R) brukes, mens for det enkelte databit lik 1 vil vippens sett-inngang (S) brukes.

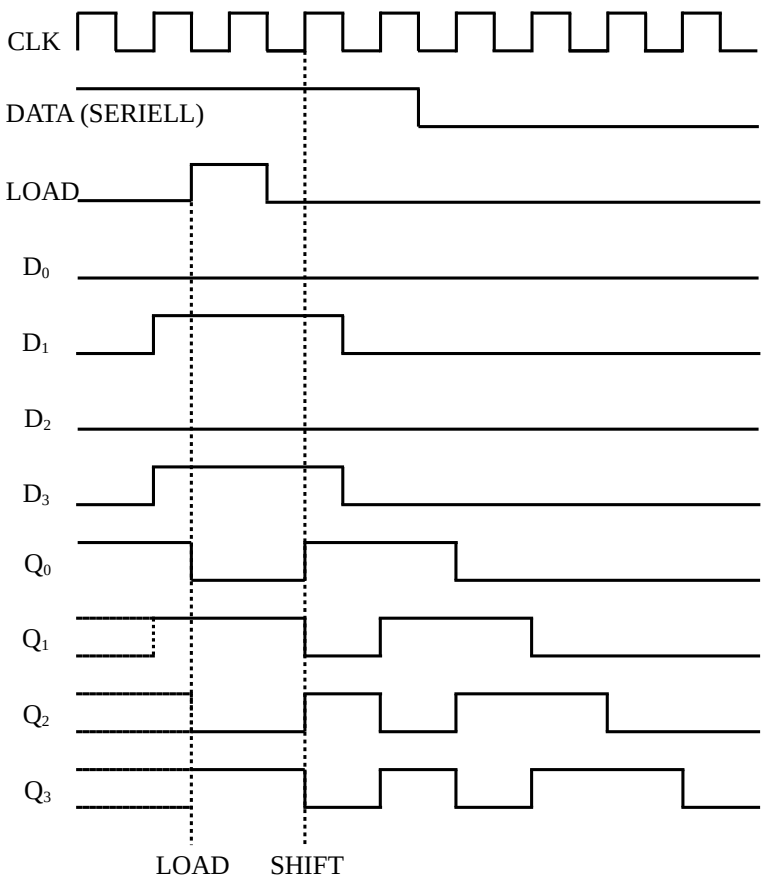
Når LOAD = 0 vil vippene ikke settes/resettes og skifting vil skje på positivtgående klokkepuls som for de andre skiftregistrene (med seriell datainngang). Det er her også tatt med en seriell datainngang, men denne er opsjonell.




*Figur 7.6. Parallell inn/serie ut skiftregister.*

I figur 7.7 er vist tidsdiagram for dette skiftregisteret. Det ses at LOAD = 1 vil sette/resette de enkelte vippene mens for LOAD = 0 vil data skiftes på positivtgående klokkepulser.

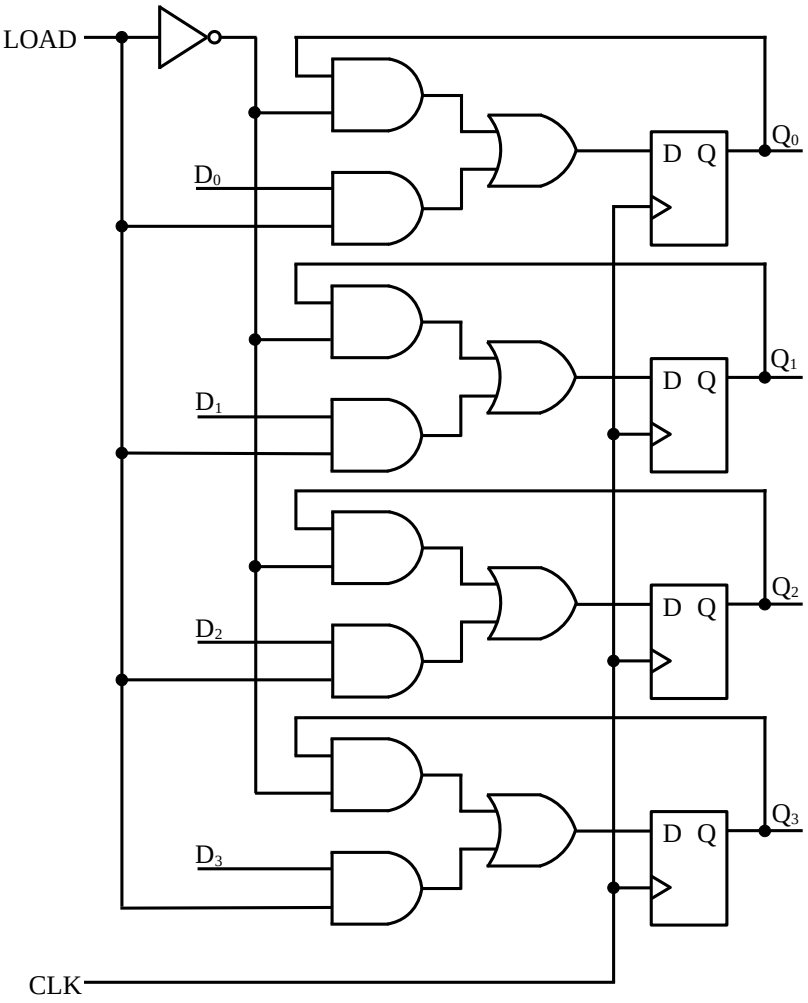
Det finnes også skiftregistre der parallell data lastes inn synkront med klokken. Oppbyggingen av et slikt register er vist i kapittel 7.6.




*Figur 7.7. Tidsdiagram for parallell inn/serie ut skiftregister.*

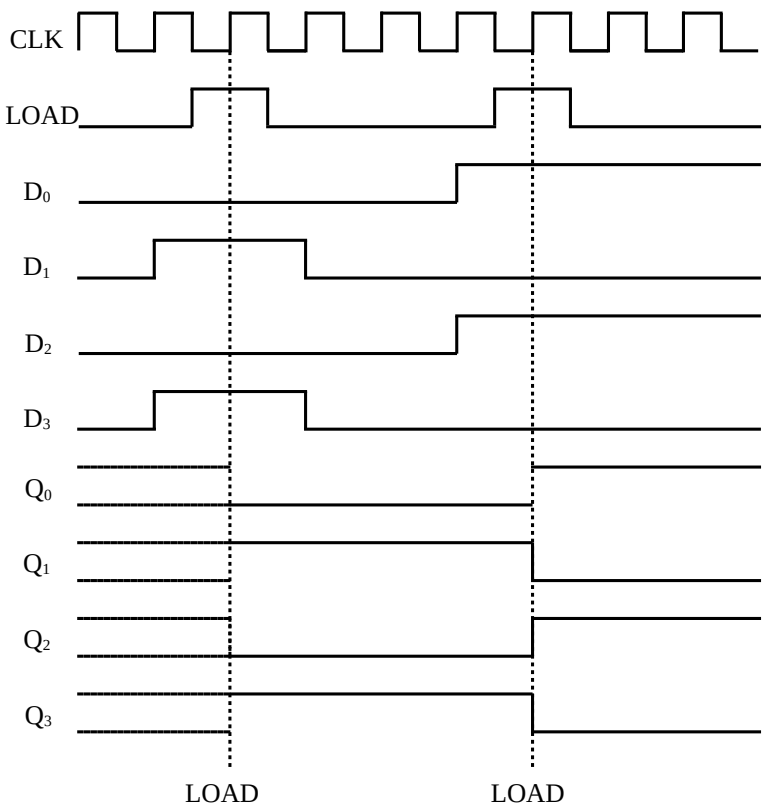
## 7.5. Parallell inn /parallell ut register

I figur 7.8 er vist skjema for et fire bit parallell inn/parallell ut register. Legg merke til at dette ikke er et skiftregister siden det ikke er noen kaskadekopling mellom de enkelte vippene. Det som er viktig å merke seg, er at data lastes inn synkront med klokken når LOAD = 1. Når LOAD = 0, blir data uendret stående i vippen. Dette er med andre ord et register som representerer et 4 bit minne.




*Figur 7.8. Parallell inn/parallell ut register.*

Tidsdiagrammet er vist i figur 7.9. Vi har antatt at inneholdet i de enkelte vipper er ukjent før registeret lastes første gang. Legg merke til at vippene inneholder den innlastede verdien når LOAD = 0 og vippene klokkes. Først når LOAD = 1 på nytt vil vippenes innhold eventuelt endres.

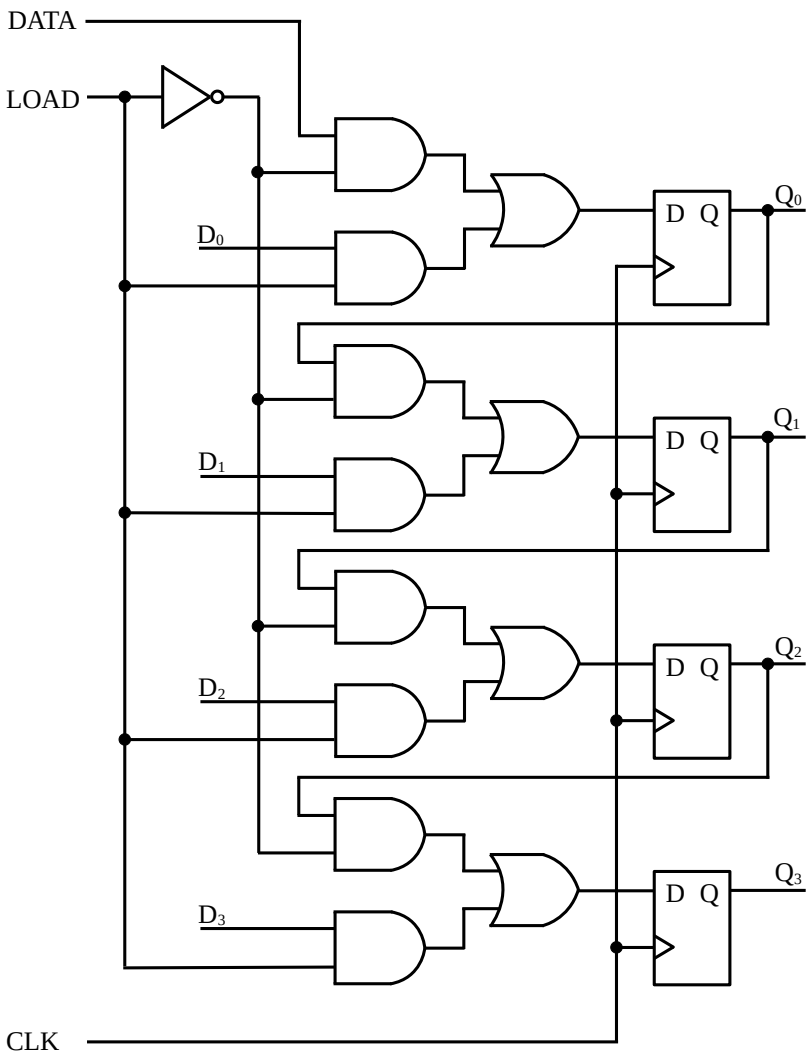



*Figur 7.9. Tidsdiagram for parallell inn/parallell ut register.*

## 7.6. Parallell inn /serie ut skiftregister

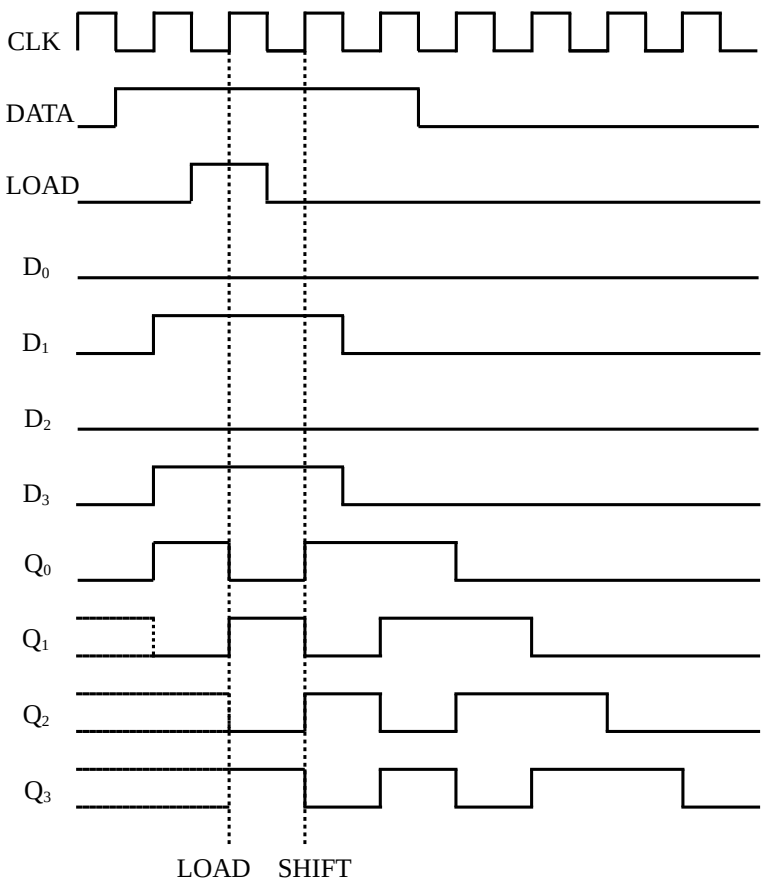
Registeret i figur 7.8 kan enkelt modifiseres til et parallell inn/serie ut skiftregister. Dette er vist i figur 7.10. Istedenfor at utgangen på vippen føres tilbake til dennes inngang når LOAD = 0, føres den til neste vippes inngang. Dermed får vi samme virkemåte for dette skiftregisteret som for det skissert i figur 7.6.

Imidlertid skjer nå innlastingen synkront med klokken når LOAD = 1. Når LOAD = 0, skiftes data mellom vippene. Skiftregisteret har også en opsjonell seriedatainngang. Det ses også at skiftregisteret kan opereres som et parallell inn/parallell ut skiftregister når alle vippeutgangene er tilgjengelige.




*Figur 7.10. Parallell inn/serie ut/parallell ut skiftregister.*

Av tidsdiagrammet i figur 7.11 ses at innlastingen av parallell data skjer synkront med klokken. Sammenlignet med tidsdiagrammet i figur 7.7, ses at dette skiller de to typene skiftregister.



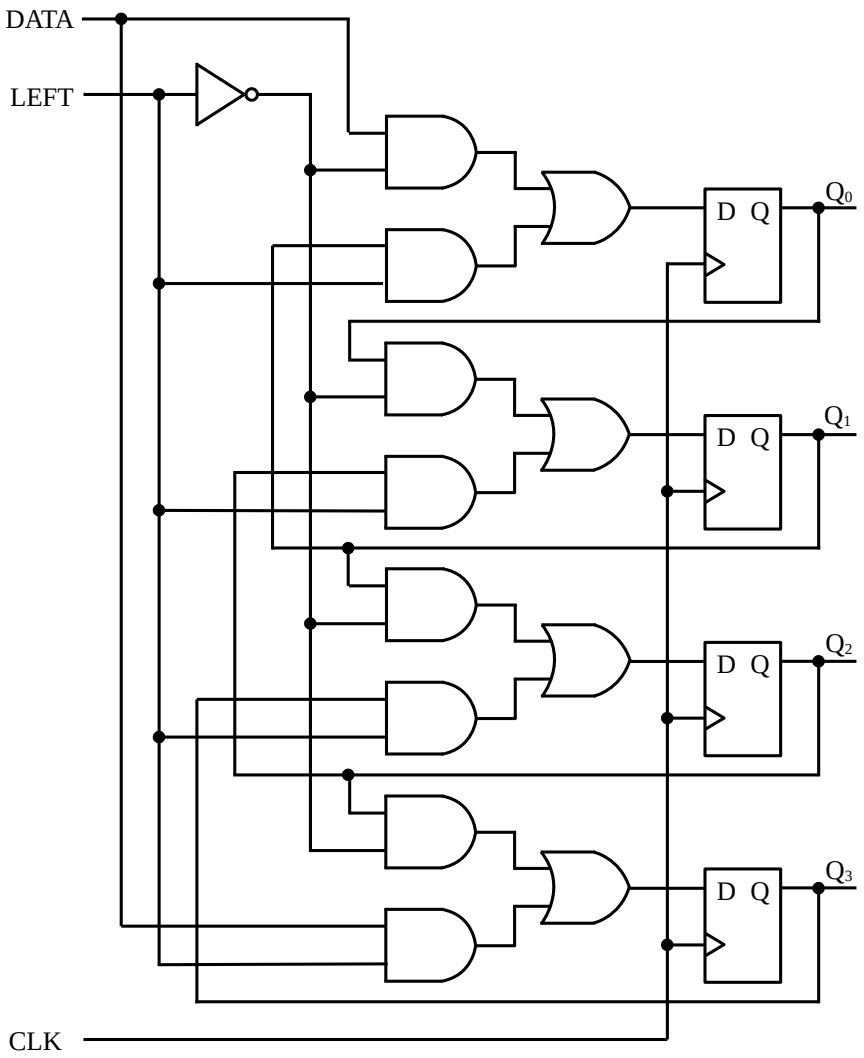

*Figur 7.11. Parallell inn/serie ut (og parallell ut) skiftregister.*

## 7.7. Bidireksjonalt skiftregister

Med utgangspunkt i registeret i figur 7.10 kan vi lage et bidireksjonalt register. Dette er vist i figur 7.12. Når LEFT = 0 føres seriell data inn på den øverste vippen (med utgang $Q_0$), dens utgang føres til neste vippe (med utgang $Q_1$) og så videre. Vi har da samme virkemåte som skiftregisteret i figur 7.3 (serie inn/serie ut skiftregister) og 7.5 (serie inn/parallell ut skiftregister), alt ettersom de parallelle utgangene er tilgjengelige.

Når LEFT = 1, føres seriell data inn på den nederste vippen (med utgang $Q_3$), dens utgang føres til vippen ovenfor (med utgang $Q_2$) og så videre. Vi får da et skiftregister der data skiftes i motsatt retning.

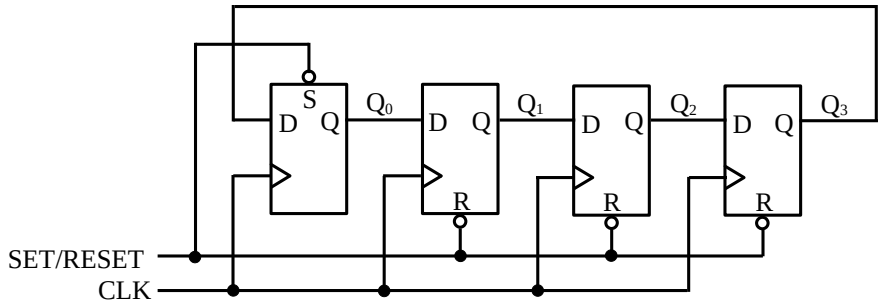
For å utvide bruksområdet ytterligere, er det vanlig å kombinere denne funksjonen med også å ha parallellinnganger som i figur 7.10. Da fås såkalte universelle skiftregistre. Slike skiftregistre er for eksempel typiske i mikrokontrollere.




*Figur 7.12. Bidireksjonalt skiftregister.*

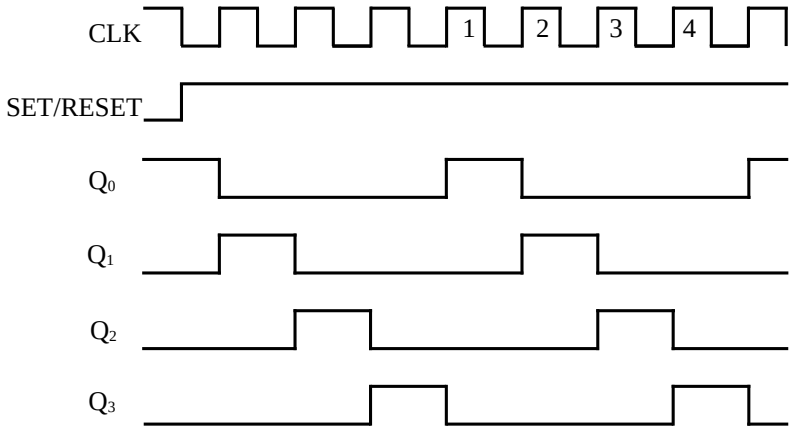
## 7.8. Ringteller

En fire bit ringteller er vist i figur 7.13. Det ses at utgangen fra siste vippe (med utgang $Q_3$) er koplet tilbake til inngangen på den første vippen (med utgang $Q_0$). Utgangen fra hver vippe føres til inngangen på neste. For at denne ringtelleren skal virke, må første vippe settes til 1 og de andre vippene settes til 0 i begynnelsen.




*Figur 7.13. Ringteller.*

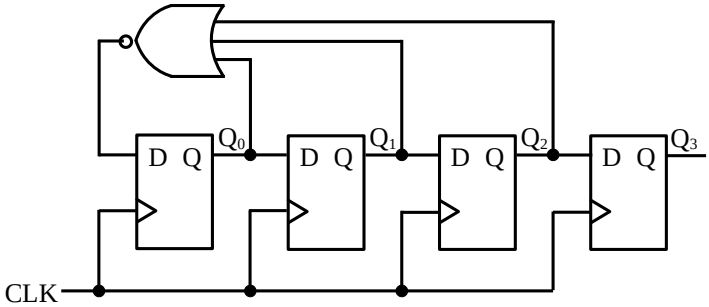
Tidsdiagrammet er vist i figur 7.14. Det ses at utgangene av vippene danner en sekvens der hver vippe går til 1 etter tur. Siden vi får en sekvens av tilstander betraktes denne koplingen som en teller. Ringtelleren er således et eksempel der skiftregistre benyttes for å realisere tellerfunksjoner.




*Figur 7.14. Tidsdiagram for ringteller.*

Siden vi har fire tilstander, kan vi betrakte denne telleren som en mod-4-teller. Generelt vil en n bit ringteller være en mod-n-teller. Siden bare fire av i alt 16 muligheter er benyttet, er telleren ikke noe særlig effektiv. Til gjengjeld trenger ikke denne telleren noen dekoding for å avgjøre hvilken tilstand den er i.

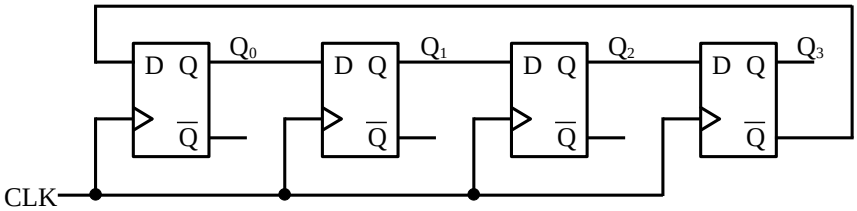
I figur 7.15 er vist en alternativ realisering av ringtelleren som ikke krever setting og resetting ved oppstart. Til gjengjeld kreves ekstra logikk, i figuren vist som en NOR-port. Tidsdiagrammet i figur 7.14 viser forløpet også for denne ringtelleren.




*Figur 7.15. Alternativ ringteller.*

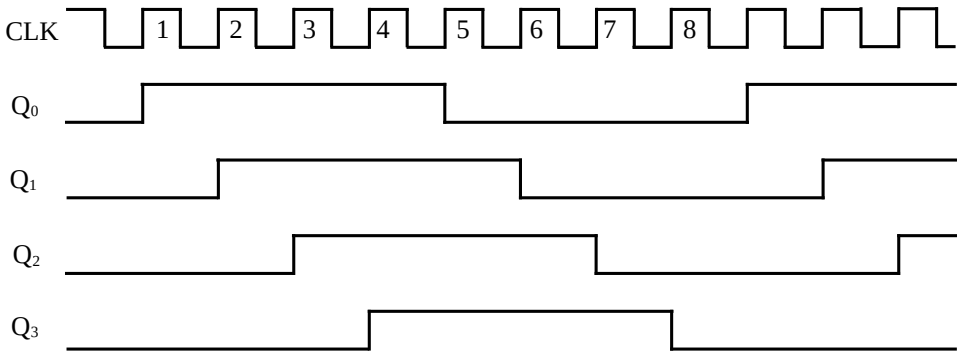
## 7.9. Johnson-teller

Johnson-telleren er en variant av ringtelleren ved at den inverterte utgangen fra den siste vippen (med utgang $Q_3$) er koplet tilbake til inngangen på den første vippen (med utgang $Q_0$). Johson-telleren i figur 7.16 er på fire bit.




*Figur 7.16. Johnson-teller.*

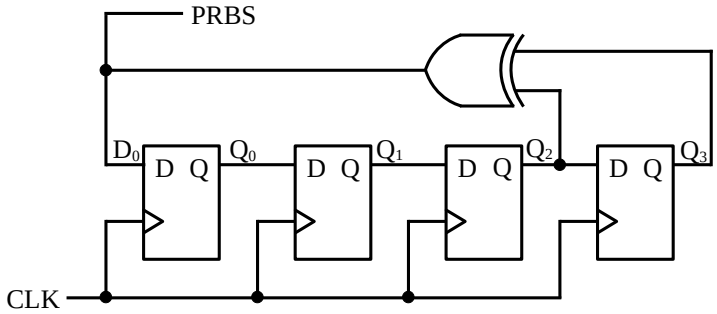
Det ses av tidsdiagrammet i figur 7.17 at vi da får en mod-8 teller. Generelt vil en n bit Johnson-teller være en mod-2n-teller. Siden bare åtte av i alt 16 muligheter er benyttet, er heller ikke denne telleren noe særlig effektiv. For å få den rette tellesekvensen, er det en fordel at alle vippene er resatte til å begynne med.




*Figur 7.17. Tidsdiagram for Johnson-teller.*

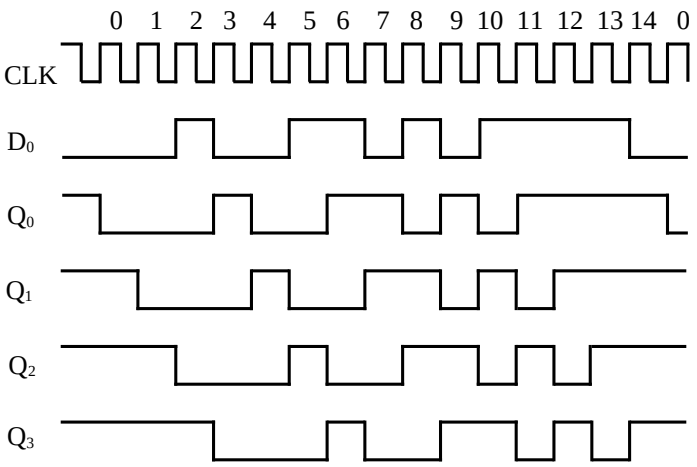
## 7.10. Mønstergenerator

I figur 7.18 er vist en 4 bit mønstergenerator. Den er realisert ved hjelp av et 4 bit skiftregister og en Eksklusiv ELLER-port. Mønstergeneratoren er et eksempel på generatorer som går under navnet pseudovilkårlige sekvensgeneratorer. Den engelske betegnelsen er ’Pseudo Random Binary Sequence’, PRBS, generator. Slike mønstergeneratorer brukes ofte til å teste høy-hastighets serielt utstyr for feil. Ved å bruke skiftregistre av forskjellig lengde og en eller flere Eksklusiv ELLER-porter, kan pseudovilkårlige sekvenser av varierende lengde realiseres.




*Figur 7.18. 4 bit mønstergenerator.*

Et tidsdiagram for mønstergeneratoren er vist i figur 7.19. Utgangen fra generatoren kan egentlig tas fra alle vippene eller fra porten. Det er viktig å observere at alle vippene er antatt satt til 1 til å begynne med. Dersom alle vippene står i 0, vil det ikke skje noe når registeret klokkes. Det genereres $2^4$-1 = 15 tilstander, som gjentas kontinuerlig. Det sies at ’PRBS’-lengden (perioden) er 15 bit. Mønsteret inneholder alle kombinasjoner unntatt bare 0-ere.




*Figur 7.19. Tidsdiagram for 4 bit mønstergenerator.*

Tilsvarende vil en mønstergenerator på L bit ha en 'PRBS'-lengde på $2^L-1$ bit. Mønstergeneratoren i figuren inneholder $2^{L-1} = 2^{4-1} = 8$ enere og $2^{L-1} -1 = 2^{4-1} -1 = 7$ nullere. Er for eksempel L lik 7 bit, vil 'PRBS'-lengden være lik $2^7-1 = 127$ bit. Generatoren vil da inneholde $2^{L-1} = 2^{7-1} = 64$ enere og $2^{L-1} -1 = 2^{7-1} -1 = 63$ nullere.

Inngangene til Eksklusiv-ELLER-porten i generatoren i figur 7.18 er tatt fra tredje og fjerde D-vippe. Denne generatoren vil da generelt uttrykkes matematisk som:

$$p(x) = x^3 + x^4 + 1 \tag{7.1}$$

Tallet 1 i polynomet svarer til inngangen av første vippe ($x^0 = 1$). Bruken av x er ganske vanlig, men x er egentlig ikke en variabel. Bruken gjør det imidlertid mulig å benytte polynomer i matematikken for slike generatorer, og som nevnt i kapittel 10.2, også for såkalte scramblere og descramblere. Et polynom som for eksempel er gitt ved $p(x) = x^9 + x^5 + 1$ vil således hente inngangene til Eksklusiv-ELLER-porten fra niende og femte D-vippe (siden skiftregisteret er på 9 bit). Perioden på denne generatoren vil da være $2^L-1 = 2^9-1 = 511$.

Dersom oppgaven til mønstergeneratoren er å teste utstyr med flest mulige bitkombinasjoner, er det en fordel at perioden er lang samtidig som polynomet er mest mulig effektivt. Det finnes derfor flere standardiserte polynomer til dette formålet.

Til slutt skal nevnes at slike mønstergeneratorer som dette kan brukes som tellere der ikke-binære tellesekvenser kan aksepteres. Fordelen er at disse tellerne vil være raske siden logikken som kreves er forholdsvis enkel og fordi skiftregistre i seg selv er meget raske og kan operere ved høye klokkefrekvenser.

# Kapittel 8
# Tellere

## 8.1. Innledning

Kretser med tellere brukes ofte i digitale systemer. Siden en teller må holde rede på tidligere tilstander, må den være oppbygd med minne. Tellere vil da vanligvis være oppbygd av vipper. De vanligste vil da være laget med hjelp av JK- eller D-vipper. Antall vipper og hvordan de er koplet sammen, vil da bestemme antall telletilstander og sekvensen som telleren gjennomløper for hver tellesyklus.

Tellere kan grovt klassifiseres i to grupper etter hvordan telleren klokkes:

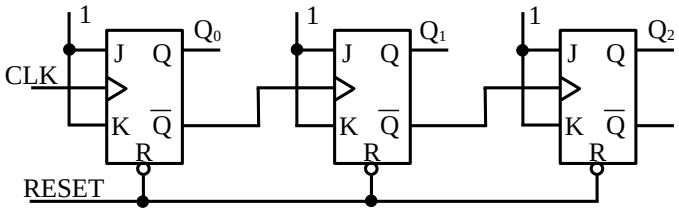
1. Asynkrone tellere: Den første vippen klokkes av et eksternt klokkesignal mens hver påfølgende vippe klokkes av $Q/\overline{Q}$ -utgangen til foregående vippe.
2. Synkrone tellere: Alle vippene klokkes av samme eksterne klokkesignal.

Vi skal her se på begge gruppene der flere typer tellere realiseres. Som eksempel kan nevnes binærtellere og dekadetellere. Vi vil også se på opp/ned-tellere.

## 8.2. Asynkron 3 bit binærteller

En 3 bit asynkron binærteller er vist i figur 8.1. Den eksterne klokken CLK er koplet til klokkeinngangen på den første vippen (med utgang $Q_0$). Denne skifter tilstand for hver positivtgående klokkepuls. Dette skjer siden J- og K-inngangene er koplet til 1 (logisk høy). Dette svarer til en vippe som kalles T-vippe siden den skifter tilstand (’toggler’) for hver klokkepuls.

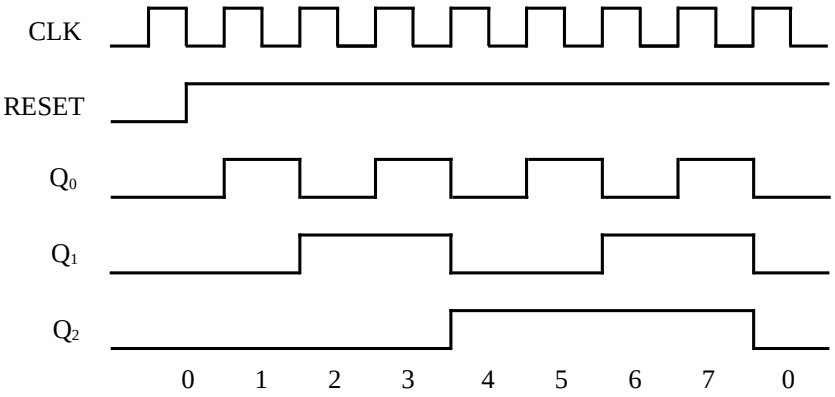
Den neste vippen (med utgang $Q_1$) skifter tilstand bare når utgang $\overline{Q}_0$ går fra lav (0) til høy (1). På grunn av transportforsinkelsen i vippen, vil aldri transisjonen av klokken og utgangen $\overline{Q}_0$ skje samtidig. Den neste vippen (med utgang $Q_2$) skifter tilstand bare når utgang $\overline{Q}_1$ går fra lav til høy. Det skjønnes at vippene ikke kan trigges samtidig, og vi har en asynkron teller.




*Figur 8.1. Asynkron 3 bit binærteller.*

For enkelhets skyld er transisjonene i tidsdiagrammet i figur 8.2 vist å skje samtidig. I virkeligheten vil det være en liten tidsforsinkelse mellom transisjonene til CLK, $Q_0$, $Q_1$ og $Q_2$. I skjemaet i figur 8.1 er også tatt med en asynkron (samtidig) resetting av alle vippene, som skjer når RESET er lav (0). Denne resettingen er uavhengig av klokkesignalet CLK.

Denne typen asynkrone tellere kalles også rippel-tellere siden transisjonene brer seg fra den ene vippen til den neste.




*Figur 8.2. Tidsforløp for 3 bit asynkron binærteller.*

Denne 3 bit telleren har 8 forskjellige tilstander (0-7) som svarer til de 8 forskjellige tellerverdiene. Dette er vist i tidsdiagrammet i figur 8.2 og i tabell 8.1.

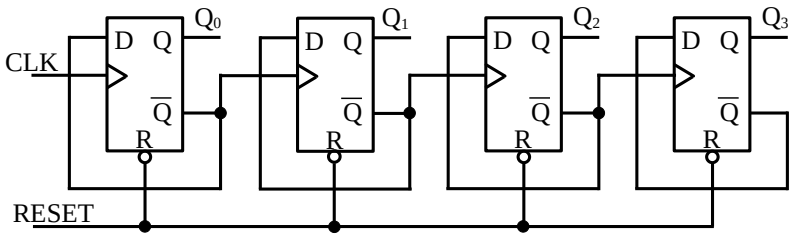
| Klokke | $Q_2$ | $Q_1$ | $Q_0$ |
|---|---|---|---|
| 0 | 0 | 0 | 0 |
| 1 | 0 | 0 | 1 |
| 2 | 0 | 1 | 0 |
| 3 | 0 | 1 | 1 |
| 4 | 1 | 0 | 0 |
| 5 | 1 | 0 | 1 |
| 6 | 1 | 1 | 0 |
| 7 | 1 | 1 | 1 |

*Tabell 8.1.Tilstander 0-7.*

Tilsvarende kan en teller med n vipper ha $2^n$ forskjellige tilstander (tellerverdier). Antall tilstander i en teller er kjent som modulus (mod)-tallet. En 3 bit teller som denne er da for eksempel en mod-8 teller.

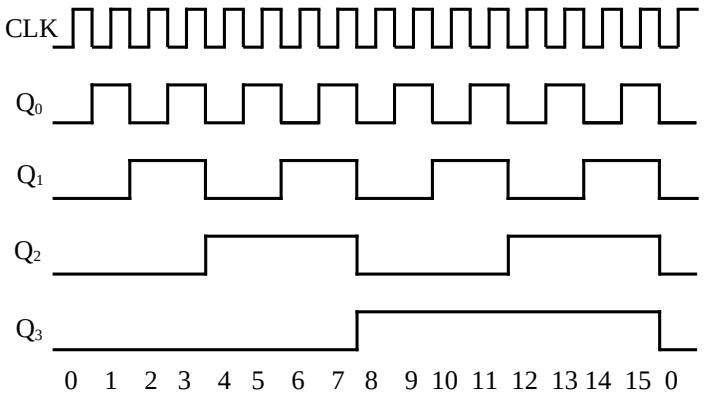
En mod-n teller kan også benevnes en ’del på n’-teller. Dette er fordi den mest signifikante vippen ($Q_2$ i figur 8.1) har en periode som er n ganger klokkeperioden. Frekvensen til den mest signifikante vippen er dermed n’te-delen av klokkefrekvensen. Følgelig er denne 3 bit telleren en ’del på 8’-teller.

Dersom vi føyer til en ekstra vippe i figur 8.1, som klokkes fra den inverterte utgangen $Q_2$, får vi en 4 bit asynkron binærteller, som følgelig gjennomløper $2^4 = 16$ forskjellige tilstander (tellerverdier). Dette er vist i figur 8.3, der vi har benyttet D-vipper istedenfor JK-vipper.




*Figur 8.3. 4 bit asynkron binærteller.*

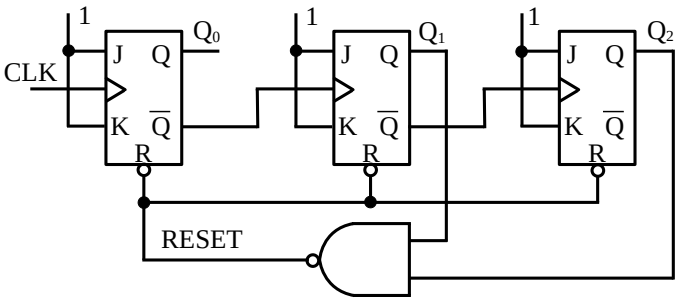
Denne 4 bit telleren har som nevnt 16 forskjellige tilstander (0-15) som svarer til de 16 forskjellige tellerverdiene. Dette er vist i tidsdiagrammet i figur 8.4.




*Figur 8.4. Tidsforløp for 4 bit asynkron binærteller.*

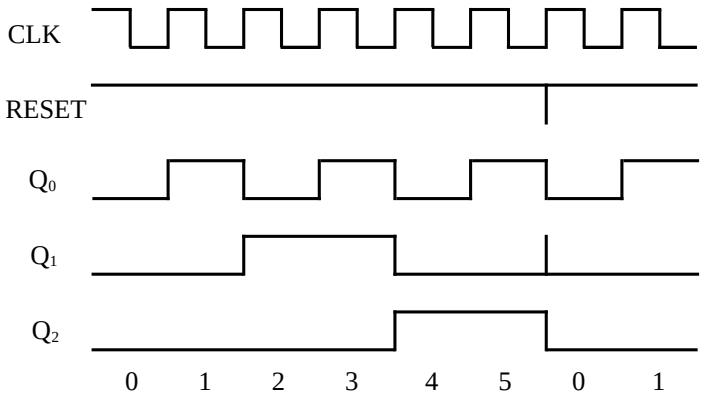
## 8.3. Asynkron mod-6 teller

Dersom vi modifiserer en 3 bit asynkron binærteller, kan vi få en asynkron mod-6 teller som vist i figur 8.5. Her har vi igjen benyttet JK-vipper.




*Figur 8.5. Mod-6 asynkron teller.*

Istedenfor å la telleren gjennomløpe alle de 8 tilstandene, tvinges den til å resettes når verdien 6 (binært 110) nås. Tidsdiagrammet er vist i figur 8.6. Det ses at RESET settes lavt lenge nok til at alle vippene resettes. Ulempen er at telleren kortvarig er innom telleverdien 6 (binært 110). I mange sammenhenger trenger dette ikke å ha noen betydning i praksis.

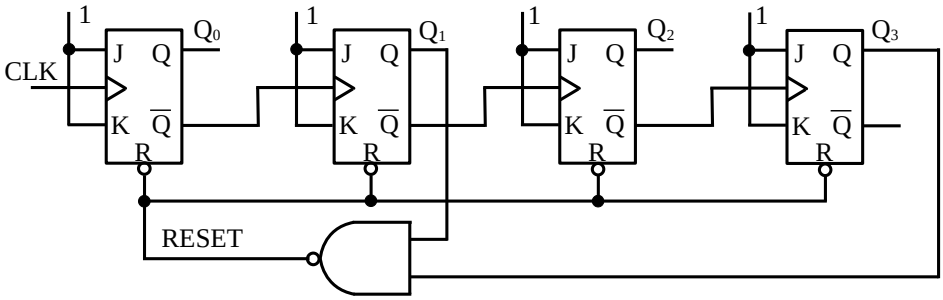



*Figur 8.6. Tidsdiagram for mod-6 asynkron teller.*

Bemerk forøvrig at det bare er utgangene $Q_1$ og $Q_2$ som benyttes for å lage resett-pulsen. Dette kalles delvis dekoding. Grunnen til at vi kan gjøre det på denne måten, er at ingen andre tilstander (for verdiene 0 til 5) har utgangene $Q_1$ og $Q_2$ høye samtidig.

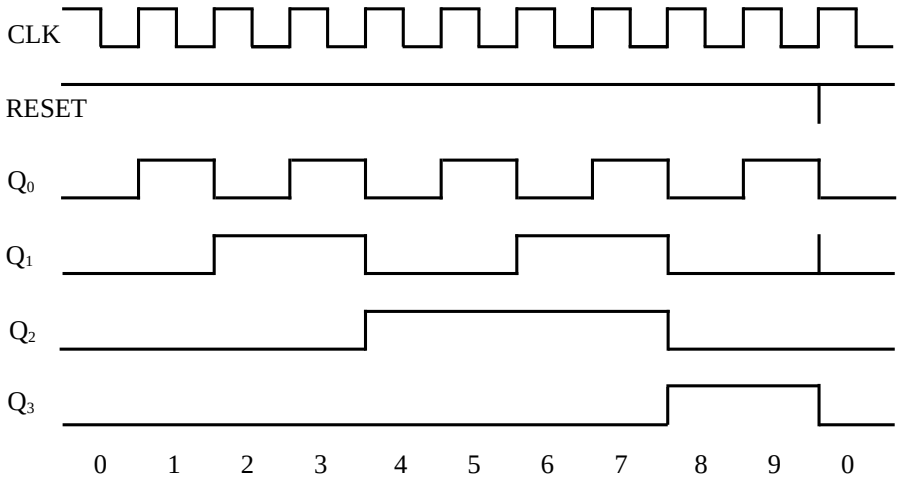
## 8.4. Asynkron dekadeteller

Dersom vi modifiserer en 4 bit asynkron binærteller, kan vi få en asynkron dekadeteller som vist i figur Feil: Fant ikke kilden til referansen. Istedenfor å la telleren gjennomløpe alle de 16 tilstandene, tvinges den til å resettes når verdien 10 (binært 1010) nås.




*Figur 8.7. Asynkron dekadeteller.*

Tidsdiagrammet er vist i figur 8.8. Det ses at RESET settes lavt lenge nok til at alle vippene resettes. Ulempen er at telleren kortvarig er innom telleverdien 1010.

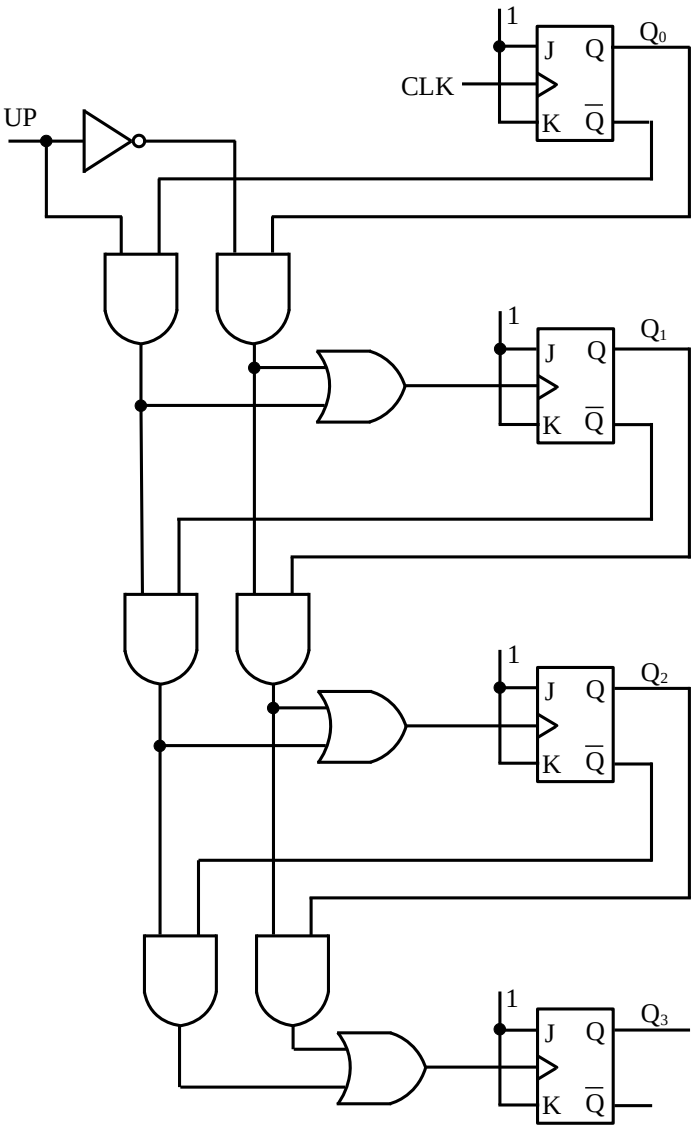



*Figur 8.8. Tidsdiagram for asynkron dekadeteller.*

Bemerk forøvrig at det bare er utgangene $Q_1$ og $Q_3$ som benyttes for å lage resett-pulsen. Grunnen til at vi kan gjøre det på denne måten, er at ingen andre tilstander (for verdiene 0 til 9) har utgangene $Q_1$ og $Q_3$ høye samtidig.

## 8.5. Asynkrone opp/ned-tellere

For enkelte anvendelser er det ønskelig at tellere kan både telle opp og ned. Kretsen vist i figur 8.9 er en 4 bit asynkron opp/ned-teller.




*Figur 8.9. Asynkron opp/ned-teller.*

Den teller opp når styresignalet UP er logisk 1 og ned i det motsatte tilfellet. For UP lik logisk 0 vil vippen med utgang $Q_1$ skifte tilstand når vippen med utgang $Q_0$ går høy.

Tilsvarende vil vippen med utgangen $Q_2$ skifte tilstand når vippen med utgang $Q_1$ går høy. Til slutt ses at at vippen med utgangen $Q_3$ vil skifte tilstand når vippen med utgang $Q_2$ går høy. Dette vil føre til at telleren virker som en nedteller.

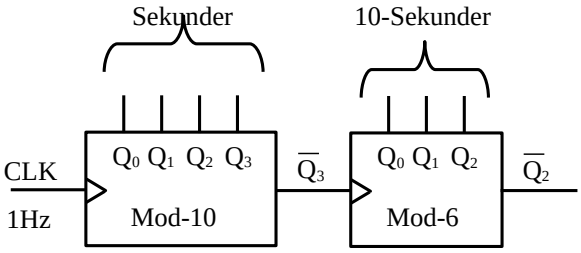
Tellesekvensen vil da (desimalt) være: 15, 14, 13, 12, 11, 10, 9, 8, 7, 6, 5, 4, 3, 2, 1, 0, 15, 14,...

Tellesekvensen for en opp-teller vil være som før: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 0, 1,...

Det bør bemerkes at en opp/ned-teller vil være mindre rask enn en tilsvarende opp- eller ned-teller på grunn av den ekstra logikken som kreves, siden denne medfører ekstra transportforsinkelse mellom portene.

## 8.6. Kaskadekopling av tellere

De enkelte tellerne kan koples i kaskade. I figur 8.10 er vist et eksempel med en mod-10-teller (dekadeteller) og en mod-6-teller fra henholdsvis figur Feil: Fant ikke kilden til referansen og 8.5. Vi får da en mod-60-teller som teller fra desimalt 00 til 59. Som klokkefrekvens er det tatt et eksempel der frekvensen er 1 Hz. Ut fra dekadetelleren får vi da sekunder mens vi får 10-sekunder fra mod-6-telleren (som deler med 6).




*Figur 8.10. Kaskadekopling av mod-10 og mod-6-teller.*

Den sistnevnte telleren får sin klokke fra dekadetelleren. Dersom vi kopler en ny mod-60-teller etter kaskaden vist i figur 8.10, kan vi få en minutt-teller og en 10-minutt-teller. Klokkeinngangen til denne minutt-telleren kan få sin klokke fra mod-6-telleren i figur 8.10.

## 8.7. Ulemper med asynkrone tellere

Et problem med asynkrone tellere er som nevnt at ikke alle vippeutganger skifter samtidig. Dette betyr at vippene tidligere i kjeden skifter før de siste vippene på grunn av forsinkelsen i vippene. Resultatet er at det fås falske tellertilstander. Jo flere bit, jo mer alvorlig er den samlede forsinkelsen. Tar vi en 4 bit binærteller som eksempel (se figur 8.4), vil ikke den telle fra 0111 til 1000 som den skal. I stedet går den fra 0111 til 0110, 0100, 0000 og til 1000 (fra 7 til 6, 4, 0 og til 8).

I mange tilfeller kan effekten tolereres. Hvis vi for eksempel bare ønsket å vise tiden på en skjerm fra telleren, ville det spille liten rolle om vi hadde denne feilen, siden vi ikke ville kunne se mellomtilstandene.

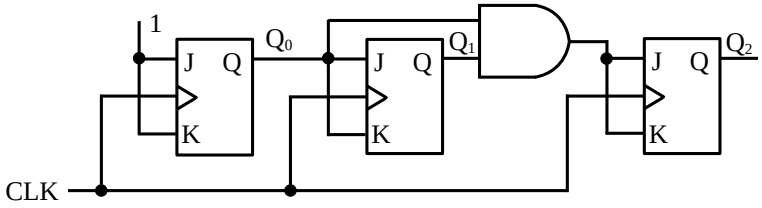
Skulle vi derimot bruke telleren til å drive valg-inngangene til en multiplekser eller til en adressepeker, kan feilen få fatale konsekvenser. Derfor vil en i mange tilfeller foretrekke synkrone tellere.

En annen ulempe med asynkrone tellere er at forsinkelsen er additiv. Med mange trinn vil dette begrense hvor høy hastighet telleren kan opereres med. Med synkrone tellere vil vi ikke få en slik additiv forsinkelse mellom vippene. I synkrone tellere kan derimot kompleksiteten av logikken som kreves utenom vippene, bidra til å redusere maksimal klokkehastighet.

## 8.8. Synkron 3 bit binærteller

En 3 bit synkron binærteller er vist i figur 8.11. Det er vist en realisering med JK-vipper. Legg merke til at alle vippene klokkes med samme klokke (CLK). Den første vippen (med utgang $Q_0$) skifter tilstand for hver positivtgående klokkepuls. Dette skjer siden J- og K-inngangene er koplet til 1 (logisk høy). Den neste vippen (med utgang $Q_1$) skifter tilstand bare når utgang $Q_0$ er høy og klokken går høy. Den neste vippen (med utgang $Q_2$) skifter tilstand bare når utgang $Q_0$ og utgang $Q_1$ samtidig er høy og klokken går høy.

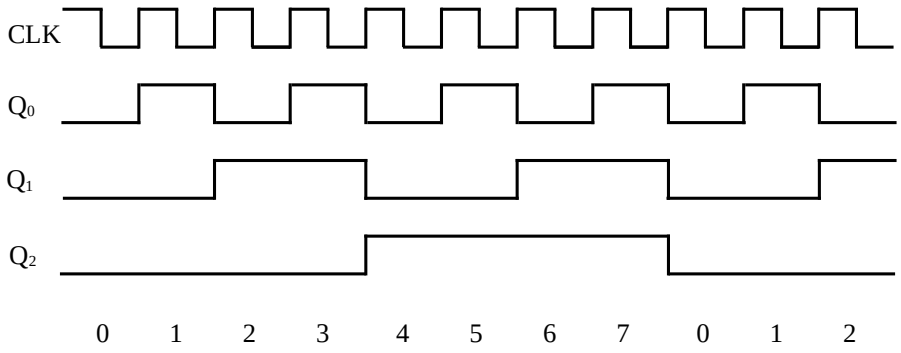
I skjemaet er det ikke tatt med noen asynkron (samtidig) resetting av alle vippene, men dette er det selvfølgelig mulig å gjøre. Denne resettingen er da uavhengig av klokkesignalet CLK. Det er også mulig å få denne resettingen til å skje synkront med klokken, men da blir telleren adskillig mer komplisert.




*Figur 8.11. Synkron 3 bit binærteller*

Ideelt sett skal transisjonene i tidsdiagrammet i figur 8.12 skje samtidig, og i praksis vil det være liten forskjell, i særlig grad dersom vippene er laget på samme brikke.

Siden maksimal klokkefrekvens i denne telleren er bestemt av transportforsinkelsen i JK-vippene, vil denne synkrone telleren være raskere enn tilsvarende asynkrone teller. Dette er en generell regel for alle synkrone tellere som sammenlignes med asynkrone tellere.




*Figur 8.12. Tidsdiagram for synkron 3 bit binærteller.*

## 8.9. Gray-teller

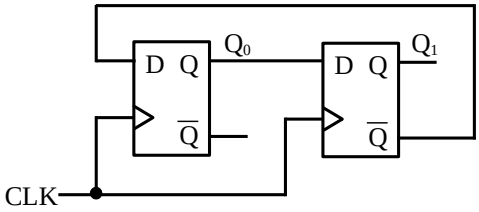
Gray-telleren karakteriseres ved at kun ett bit endrer seg fra en tellerstilling til neste. I tabell 8.2 er vist tellerforløpet for 2 og 3 bit Gray-teller.

| Klokke | $Q_1$ | $Q_0$ |
|---|---|---|
| 0 | 0 | 0 |
| 1 | 0 | 1 |
| 2 | 1 | 1 |
| 3 | 1 | 0 |
| 0 | 0 | 0 |

| Klokke | $Q_2$ | $Q_1$ | $Q_0$ |
|---|---|---|---|
| 0 | 0 | 0 | 0 |
| 1 | 0 | 0 | 1 |
| 2 | 0 | 1 | 1 |
| 3 | 0 | 1 | 0 |
| 4 | 1 | 1 | 0 |
| 5 | 1 | 1 | 1 |
| 6 | 1 | 0 | 1 |
| 7 | 1 | 0 | 0 |
| 0 | 0 | 0 | 0 |

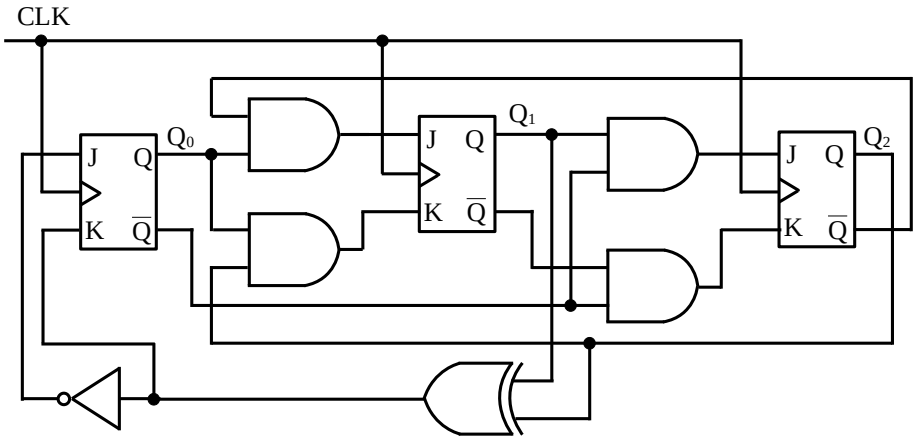
*Tabell 8.2. Gray-teller (2 og 3 bit).*

Tellerforløpet er 00, 01, 11, 10 for 2 bit telleren. For 3 bit telleren er forløpet 000, 001, 011, 010, 110, 111, 101, 100. Realiseringen av en 2 bit Gray-teller ved hjelp av datavipper er vist i figur 8.13.




*Figur 8.13. Synkron 2 bit Gray-teller.*

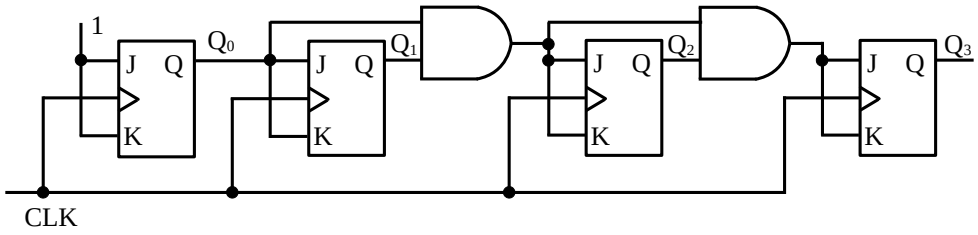
Realiseringen av 3 bit Gray-tellere er noe mer komplisert. I figur 8.14 er vist skjema for en slik teller realisert ved hjelp av JK-vipper.




*Figur 8.14. Synkron 3 bit Gray-teller.*

## 8.10. Synkron 4 bit binærteller

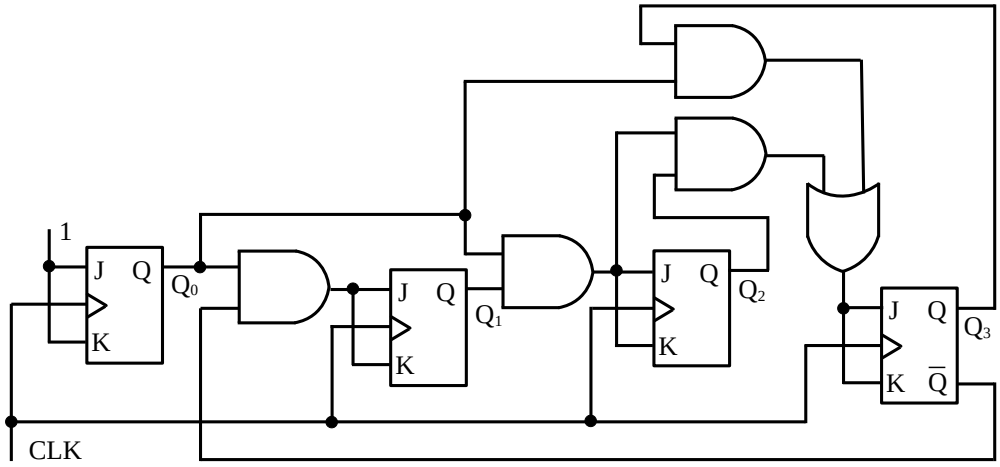
En 4 bit synkron binærteller er vist i figur 8.15. Det er igjen vist en realisering med JK-vipper, der alle vippene klokkes med samme klokke (CLK). Sammenlignet med den synkrone 3 bit telleren i figur 8.11 er det lagt til en OG-port og en JK-vippe. Dermed vil denne vippen (med utgang $Q_3$) kun skifte tilstand når utgangene $Q_0$, $Q_1$ og $Q_2$ er høye samtidig og klokken går høy. Dermed fås tellesekvensen 0 til 15 desimalt (0000, 0001, 0010 ...1111 binært).




*Figur 8.15. Synkron 4 bit binærteller.*

## 8.11. Synkron dekadeteller

Dersom vi modifiserer en 4 bit synkron binærteller, som vist i figur 8.15, kan det fås en synkron dekadeteller som vist i figur 8.16. En dekadeteller har de 10 forskjellige tilstandene (0-9) som svarer til de 10 forskjellige tellerverdiene vist i tabell 8.3.




*Figur 8.16. Synkron dekadeteller.*

Det ses at $Q_0$ skifter for hver klokkepuls. $Q_1$ endrer verdi på neste klokkepuls hver gang $Q_0 = 1$ og $Q_3 = 0$. $Q_2$ endrer verdi på neste klokkepuls hver gang $Q_0 = Q_1 = 1$. $Q_3$ endrer verdi på neste klokkepuls hver gang tellerstilling 7 og 9 has.

Denne informasjonen kan vi bruke for å realisere dekadetelleren som vist i figur 8.16. Det bør ellers bemerkes at det er flere alternative måter å realisere en dekadeteller på. I kapittel 9.5.4 er vist en alternativ realisering.

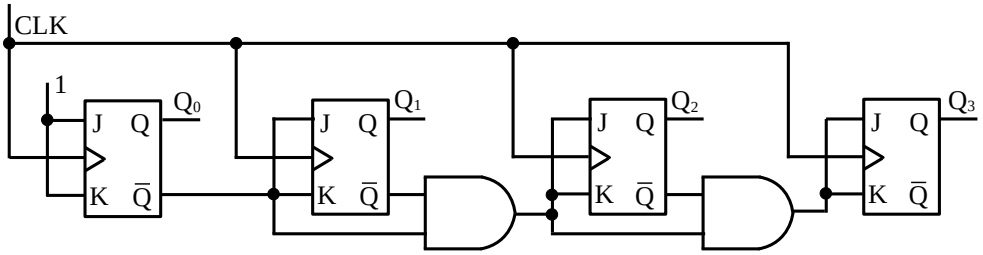
| Klokke | $Q_3$ | $Q_2$ | $Q_1$ | $Q_0$ |
|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 |
| 1 | 0 | 0 | 0 | 1 |
| 2 | 0 | 0 | 1 | 0 |
| 3 | 0 | 0 | 1 | 1 |
| 4 | 0 | 1 | 0 | 0 |
| 5 | 0 | 1 | 0 | 1 |
| 6 | 0 | 1 | 1 | 0 |
| 7 | 0 | 1 | 1 | 1 |
| 8 | 1 | 0 | 0 | 0 |
| 9 | 1 | 0 | 0 | 1 |

*Tabell 8.3. Dekadeteller.*

## 8.12. Synkron 4 bit binær nedteller

En 4 bit synkron binær nedteller er vist i figur 8.17. Sammenlignet med den synkrone 4 bit binærtelleren i figur 8.15 ses at det nå er de inverterte utgangene som styrer påfølgende vipper. Det ses at JK-vippen med utgang $Q_1$ skifter når $Q_0 = 0$, JK-vippen med utgang $Q_2$ skifter når både $Q_0 = 0$ og $Q_1 = 0$, JK-vippen med utgang $Q_3$ skifter når $Q_0 = Q_1 = Q_2 = 0$.

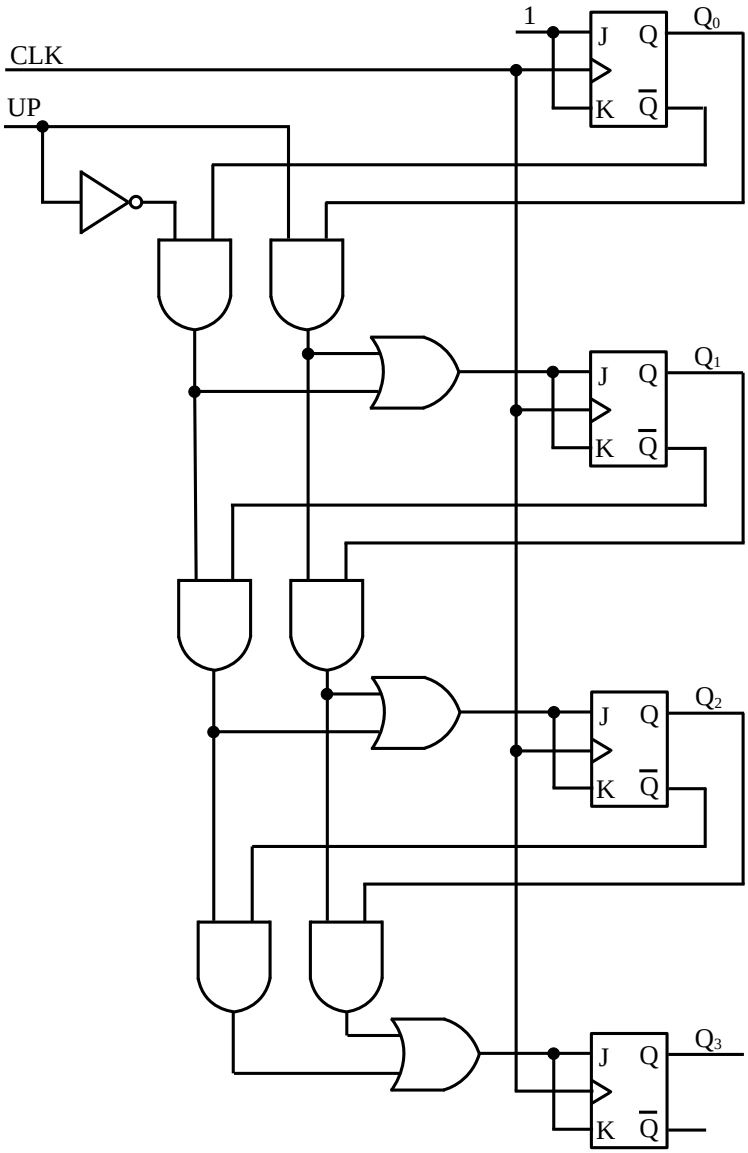
Alle transisjoner skjer ved positiv klokkepuls. Ta som eksempel at telleren er lik 0000. Det ses at da er alle JK-innganger lik 1. Da vil telleren gå til 1111 på neste klokkepuls. Dermed fås tellesekvensen: 15, 14, 13, 12, 11, 10, 9, 8, 7, 6, 5, 4, 3, 2, 1, 0, 15, 14,...




*Figur 8.17. Synkron 4 bit binær nedteller.*

## 8.13. Synkrone opp/ned-tellere

Det er også mulig å lage synkrone opp/ned-tellere. Kretsen vist i figur 8.18 er en 4 bit synkron opp/ned-teller som fås ved å kombinere tellerne i figur 8.15 og 8.17.




*Figur 8.18. Synkron 4 bit opp/ned-teller.*

Den teller opp når styresignalet UP er logisk 1 og ned når UP er logisk 0. For UP lik logisk 1 ses at telleren vil være lik den synkrone 4 bit binærtelleren. For UP lik logisk 0 ses at de inverterte vippeutgangene benyttes som innganger til vippene.

# Kapittel 9
# Sekvensielle kretser

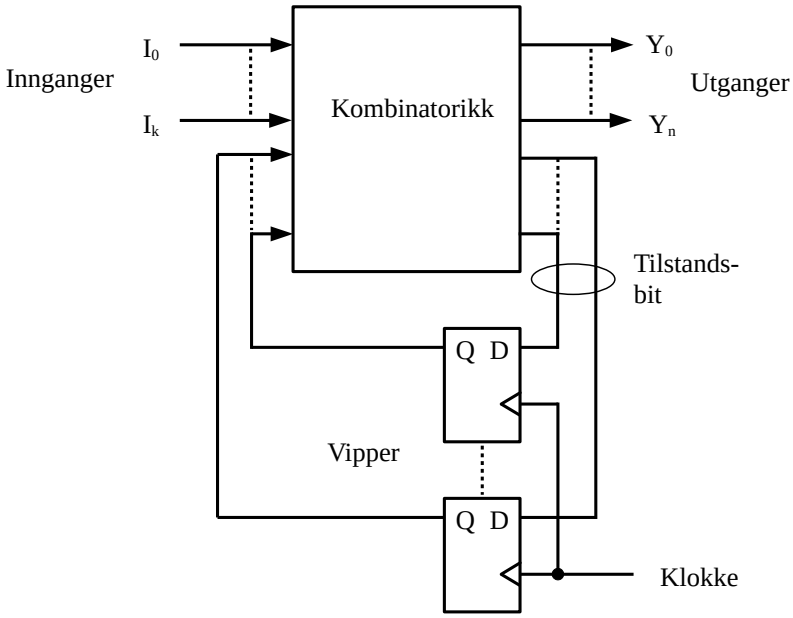
## 9.1. Innledning

Vi har tidligere sett på kombinatoriske funksjoner, der utgangene var fullstendig definert ved en kombinasjon av inngangene. Når inngangene forandres, tapes informasjonen gitt av tidligere inngangssignaler. Kombinatoriske funksjoner har med andre ord ingen hukommelse.

I mange tilfeller må vi ha tilgang til tidligere inngangsverdier for senere bruk. Selv om så å si alle digitale systemer inneholder kombinatorikk, må de sannsynligvis også ha hukommelse.

Slike digitale kretser der utgangen i tillegg til nåværende inngangsverdier avhenger av tidligere inngangsverdier, kaller vi for sekvensielle. Som eksempler på slike kretser har vi sett skift-registre og tellere i de foregående kapitlene.

Den matematiske modellen av en sekvensiell krets kaller vi for sekvensiell tilstandsmaskin, der den engelske betegnelsen er Finite State Machines, FSM. En modell er vist i figur 9.1.




*Figur 9.1. Sekvensiell tilstandsmaskin.*

Den største forskjellen i forhold til kombinatoriske funksjoner er med andre ord at tidligere inngangstilstander påvirker nåværende tilstand og utgangene. Dette er meget viktig når vi skal lage minnekretser og kontrollkretser.

Sekvensiell logikk realiseres ved hjelp av låsekretser, vipper og eventuelt også kombinatorisk logikk. Synkrone tellere som presentert i kapittel 8 er typiske eksempler på forholdsvis enkle tilstandsmaskiner.

I blokkdiagrammet i figur 9.1 ses at en blokk med kombinatorikk er koplet til minne-elementer (i skjemaet vist som datavipper) som danner en tilbakekoplingsvei. Minne-elementene kan lagre binær informasjon mens kombinatorikken som inngangssignal har verdiene fra utenomverdenen i tillegg til verdiene fra minne-elementene. Minne-elementenes verdier, ofte benevnt som tilstandsvariable, kalles også nåværende tilstand til kretsen.

Blokkdiagrammet viser også at utgangene fra blokken ikke bare er en funksjon av eksterne innganger, men også av nåværende tilstand til minne-elementene. Neste tilstand til minne-elementene er også en funksjon av eksterne innganger og nåværende tilstand. Følgelig er en sekvensiell krets spesifisert av tidsforløpet til innganger, utganger og interne tilstander.

Blokkdiagrammet i figur 9.1 viser en Moore tilstandsmaskin, der dens utganger er funksjon av nåværende tilstand og ikke dens nåværende innganger. I en Mealy tilstandsmaskin vil derimot utgangene også være en funksjon av inngangene. I det følgende vil de fleste eksemplene kunne kategoriseres som Moore tilstandsmaskiner.

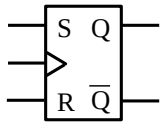
## 9.2. Egenskaper for vipper

Siden minne-elementene i sekvenskretser i hovedsak er bygd opp av vipper, vil vi begynne med å oppsummere egenskapene til disse. Vi velger her å dele inn vippene i de tre typene SR-vipper, JK-vipper og D-vipper. Vi viser da symbol, sannhetstabell og transisjonstabell for disse tre typene.

Transisjonstabellen forteller hva vippeinngangene må være for at vi skal gå fra en gitt nå-tilstand til en ny tilstand, for eksempel for at en Q-utgang lik 1 skal bli 0. Vi benytter betegnelsen $Q_n$ for nåværende tilstand og betegnelsen $Q_{n+1}$ for neste tilstand. Dette er under forutsetning av at vi har et aktivt klokkesignal, vanligvis en klokke med oppad- eller nedad-gående flanke.

Tabell 9.1 viser symbol, sannhetstabell og transisjonstabell for SR-vippen som trigges på oppadgående flanke. Det er samme sannhetstabell og samme transisjonstabell dersom vippen trigges på nedadgående flanke. Av transisjonstabellen ses for eksempel at dersom vippen står i 1 ($Q_n = 1$) og vi ønsker at utgangen skal være 0 ($Q_{n+1} = 0$), må vi sette S = 0 og R = 1.

Vi benytter - (bindestrek) i de tilfeller det ikke spiller noen rolle om inngangen settes til 0 eller 1. For eksempel ses at dersom vippen står i 1 ($Q_n = 1$) og vi ønsker at utgangen fortsatt skal være 1 ($Q_{n+1} = 1$), må vi sette R = 0 mens S kan være både 0 og 1, vist som - (bindestrek).

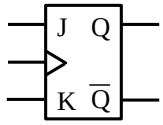
| Symbol |
|---|
|   |

| Sannhetstabell | | | |
|---|---|---|---|
| S | R | Q | $\overline{Q}$ |
| 0 | 0 | $Q_n$ | $\overline{Q}_n$ |
| 0 | 1 | 0 | 1 |
| 1 | 0 | 1 | 0 |
| 1 | 1 | Ugyldig | |

| Transisjonstabell | | | |
|---|---|---|---|
| $Q_n$ | $Q_{n+1}$ | S | R |
| 0 | 0 | 0 | - |
| 0 | 1 | 1 | 0 |
| 1 | 0 | 0 | 1 |
| 1 | 1 | - | 0 |

*Tabell 9.1. Symbol, sannhetstabell og transisjonstabell for SR-vippe.*

Tabell 9.2 viser symbol, sannhetstabell og transisjonstabell for JK-vippen som trigges på oppadgående flanke. Også her er det samme sannhetstabell og samme transisjonstabell dersom vippen trigges på nedadgående flanke. Av transisjonstabellen ses for eksempel at dersom vippen står i 1 ($Q_n = 1$) og vi ønsker at utgangen skal være 0 ($Q_{n+1} = 0$), må vi sette K = 1 mens J kan være både 0 og 1, vist som - (bindestrek).

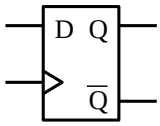
| Symbol |
|---|
|   |

| Sannhetstabell | | | |
|---|---|---|---|
| J | K | Q | $\overline{Q}$ |
| 0 | 0 | $Q_n$ | $\overline{Q}_n$ |
| 0 | 1 | 0 | 1 |
| 1 | 0 | 1 | 0 |
| 1 | 1 | Veksler | |

| Transisjonstabell | | | |
|---|---|---|---|
| $Q_n$ | $Q_{n+1}$ | J | K |
| 0 | 0 | 0 | - |
| 0 | 1 | 1 | - |
| 1 | 0 | - | 1 |
| 1 | 1 | - | 0 |

*Tabell 9.2. Symbol, sannhetstabell og transisjonstabell for JK-vippe.*

Tabell 9.3 viser symbol, sannhetstabell og transisjonstabell for D-vippen som trigges på oppadgående flanke. Denne har bare en inngang (i tillegg til klokkeinngangen). Vi har også her samme sannhetstabell og samme transisjonstabell dersom vippen trigges på nedadgående flanke. For denne vippen ses av transisjonstabellen at D-inngangen alltid skal være lik neste utgangsverdi, $D = Q_{n+1}$.

| Symbol | Sannhetstabell | | | Transisjonstabell | | |
|---|---|---|---|---|---|---|
|  | D | Q | $\overline{Q}$ | $Q_n$ | $Q_{n+1}$ | D |
| | 0 | 0 | 1 | 0 | 0 | 0 |
| | 1 | 1 | 0 | 0 | 1 | 1 |
| | | | | 1 | 0 | 0 |
| | | | | 1 | 1 | 1 |



*Tabell 9.3. Symbol, sannhetstabell og transisjonstabell for D-vippe.*

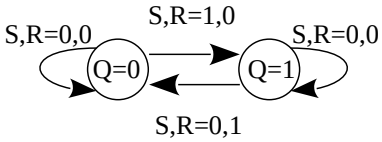
Tabell 9.1, 9.2 og 9.3 er nyttig både for å analysere sekvenskretser og for design av slike kretser der vippene benyttes.

## 9.3. Tilstandsdiagram

Tilstandsdiagram er en god metode for visualisering av sekvensielle hendelser. Her skal vi først benytte det til å visualisere virkemåten til vippene i forrige avsnitt.

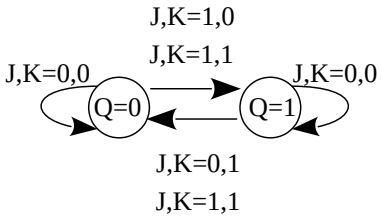
### 9.3.1. Tilstandsdiagram for vipper

I figur 9.2 er vist tilstandsdiagrammet for SR-vippen. Det ses at vi har to tilstander: Q = 0 og Q = 1. Den blir stående i Q = 0 til S = 1 og R = 0. Dette medfører skifte fra Q = 0 til Q = 1, mens S = 0 og R = 1 medfører skifte fra Q = 1 til Q = 0. Det ses videre at S = 0 og R = 0 ikke medfører noe skifte, verken for tilstanden Q = 0 eller for Q = 1. Bemerk at vi forutsetter at vippen er klokkestyrt, selv om dette ikke direkte fremgår av tilstandsdiagrammet.



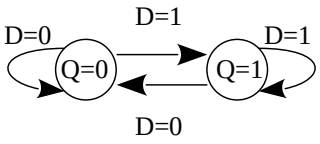

*Figur 9.2. Tilstandsdiagram for SR-vippe.*

I figur 9.3 er vist tilstandsdiagrammet for JK-vippen. Igjen har vi to tilstander: Q = 0 og Q = 1. J = 1 og K = 0 eller 1 medfører skifte fra Q = 0 til Q = 1, mens K = 1 og J = 0 eller 1 medfører skifte fra Q = 1 til Q = 0. Det ses videre at J = 0 og K = 0 ikke medfører noe skifte, verken for tilstanden Q = 0 eller for Q = 1.




*Figur 9.3. Tilstandsdiagram for JK-vippe.*

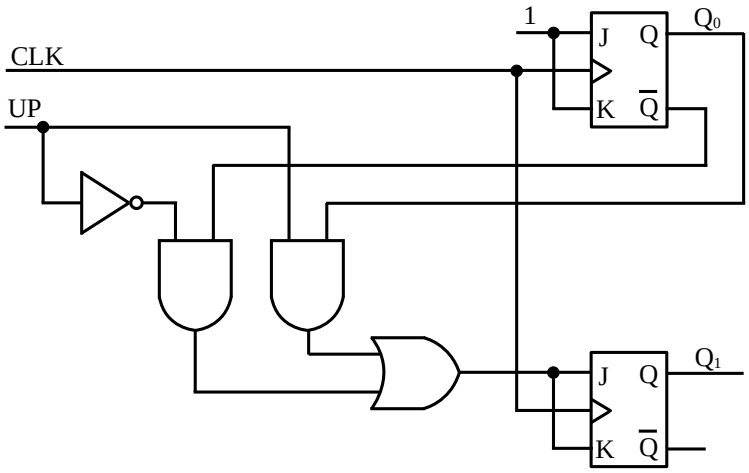
I figur 9.4 er vist tilstandsdiagrammet for D-vippen. Igjen har vi to tilstander: Q = 0 og Q = 1. D = 1 medfører skifte fra Q = 0 til Q = 1 og D = 0 medfører skifte fra Q = 1 til Q = 0. Videre ses at D = 0 ikke gir noe skifte når Q = 0. Tilsvarende vil D = 1 ikke gi noe skifte når Q = 1.




*Figur 9.4. Tilstandsdiagram for D-vippe.*

## 9.3.2. Skjema, sannhetstabell og tilstandsdiagram

For å illustrere sammenhengen mellom skjema, sannhetstabell og tilstandsdiagram kan vi ta utgangspunkt i skjemaet i figur 9.5.




*Figur 9.5. 2 bit binær opp/ned-teller.*

Dette er en 2 bit binær opp/ned-teller, bygd opp som den tilsvarende synkrone 4 bit binær opp/ned-telleren i kapittel 8.3, der UP = 1 skal få telleren til å telle opp, mens UP = 0 vil få den til å telle ned.

For å vise at dette er en 2 bit binær opp/ned-teller, kan vi for eksempel sette opp de logiske uttrykk for JK-inngangene. For den øverste vippen er $J_0 = K_0 = 1$, som betyr at denne vippen vil skifte for hver positivtgående klokkepuls. For den nederste vippen er $J_1 = K_1 = UP \cdot Q_0 + \overline{UP} \cdot \overline{Q_0}$.

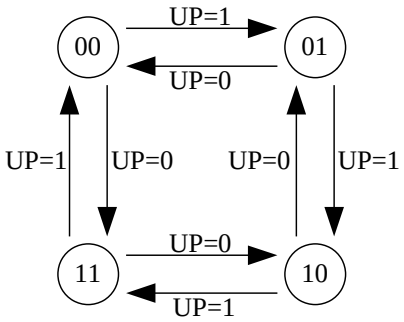
Siden J = K = 0 ikke medfører skifte og J = K = 1 medfører skifte, kan vi sette opp nå/neste-tabellen som vist i tabell 9.4.

| Tilstand | Nå-tilstand | | Inngang | Vippe | Neste tilstand | |
|---|---|---|---|---|---|---|
| | $Q_1$ | $Q_0$ | UP | $J_1 = K_1$ | $Q_1$ | $Q_0$ |
| 0 | 0 | 0 | 0 | 1 | 1 | 1 |
| | 0 | 0 | 1 | 0 | 0 | 1 |
| 1 | 0 | 1 | 0 | 0 | 0 | 0 |
| | 0 | 1 | 1 | 1 | 1 | 0 |
| 2 | 1 | 0 | 0 | 1 | 0 | 1 |
| | 1 | 0 | 1 | 0 | 1 | 1 |
| 3 | 1 | 1 | 0 | 0 | 1 | 0 |
| | 1 | 1 | 1 | 1 | 0 | 0 |

*Tabell 9.4. Nå/neste-tabell for 2 bit binær opp/ned-teller.*

Det er fire tilstander for denne telleren, benevnt desimalt 0-3. Tabellen viser skifte mellom de forskjellige tilstandene. Er for eksempel tilstanden lik 2, er nå-tilstanden $Q_1 = 1$ og $Q_0 = 0$. Er da UP = 0, vil neste tilstand være 1, med andre ord $Q_1 = 0$ og $Q_0 = 1$. Også her forutsetter vi at klokketransisjon må skje for å få til skifte av tilstand.

Tilstandsdiagrammet er vist i figur 9.6. Heller ikke her har vi tatt med noe klokkesignal, men forutsatt at klokketransisjoner er tilstede. Tilstandsdiagrammet viser ikke noe mer informasjon enn det som står i nå/neste-tabellen, imidlertid er oversikten bedre.




*Figur 9.6. Tilstandsdiagram for 2 bit binær opp/ned-teller.*

## 9.4. Design av sekvensielle kretser

Design av synkrone sekvensielle kretser begynner med et sett spesifikasjoner og ender med et logisk diagram eller en liste med boole'ske funksjoner som kan gi et logisk diagram. I motsetning til ren kombinatorikk som er fullt spesifisert ved en sannhetstabell, kreves en tilstandstabell, eller en ekvivalent representasjon slik som tilstandsdiagram, for å spesifisere sekvensielle kretser.

En sekvensiell krets realiseres ved hjelp av vipper og kombinatoriske porter. Designet av kretsene består i å velge hvilken type vipper vi vil benytte og så finne den kombinatoriske strukturen som sammen med vippene kan realisere kretsen som vil oppfylle kravspesifikasjonen. Antall vipper bestemmes av antall tilstander som trenges i kretsen.

Et forløp for et design kan være:

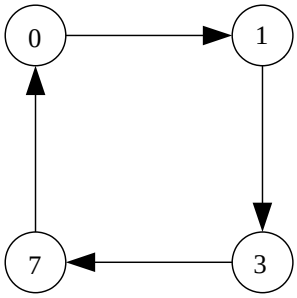
- Start med en kravspesifikasjon
- Tegn tilstandsdiagram
- Finn nå/neste tilstandstabell
- Reduser om mulig antall tilstander
- Bestem antall vipper som er nødvendig
- Velg hvilken type vippe som skal brukes
- Finn ligningene for realisering
- Utled funksjonsuttrykk for vippenes inn- og utganger (bruk Karnaugh-diagram)
- Tegn logisk skjema

I det følgende er det tatt med noen eksempler der hovedvekten er lagt på design av tellere. Til slutt er det tatt med et par andre eksempler på tilstandsmaskiner. Det bør imidlertid poengteres at bruk av syntheseverktøy, for eksempel med kombinasjon av VHDL, vil forenkle designprosedyren vesentlig. Da vil vi kunne hoppe over de fleste punktene listet ovenfor.

# 9.5. Design av tellere

## 9.5.1. 3 bit teller med D-vipper

Som et eksempel på design av synkrone tellere kan vi ta for oss en enkel 3 bit teller med D-vipper. Tilstandsdiagrammet til telleren vi ønsker å realisere, er vist i figur 9.7.




*Figur 9.7. Tilstandsdiagram for 3 bit teller.*

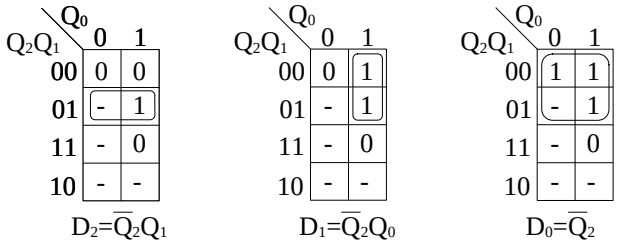
Denne telleren har bare en klokkeinngang. Den skal ha tre utganger og gjennomløpe tilstandene 0,1,3 og 7, angitt desimalt. Dette svarer da til binærverdiene 000, 001, 011 og 111.

Nå/neste-tabellen for denne telleren er vist i tabell 9.5. Med neste tilstand menes hva telleren skal gå til når alle vippene tilføres samme klokke. Vi har benyttet betegnelsene $Q_2$, $Q_1$ og $Q_0$ for vippeutgangene, der $Q_2$ er mest signifikante bit. I tabellen er også vist hva datainngangene på de forskjellige vippene må være for at vi skal kunne nå neste tilstand for telleren. Dette følger transisjonstabellen for D-vippen presentert i tabell 9.3. Datainngang $D_2$ svarer til vippen med utgang $Q_2$ og så videre.

| Tilstand | Nå-tilstand | | | Neste tilstand | | | Vippe-innganger | | |
|---|---|---|---|---|---|---|---|---|---|
| | $Q_2$ | $Q_1$ | $Q_0$ | $Q_2$ | $Q_1$ | $Q_0$ | $D_2$ | $D_1$ | $D_0$ |
| 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 1 |
| 1 | 0 | 0 | 1 | 0 | 1 | 1 | 0 | 1 | 1 |
| 3 | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| 7 | 1 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 |

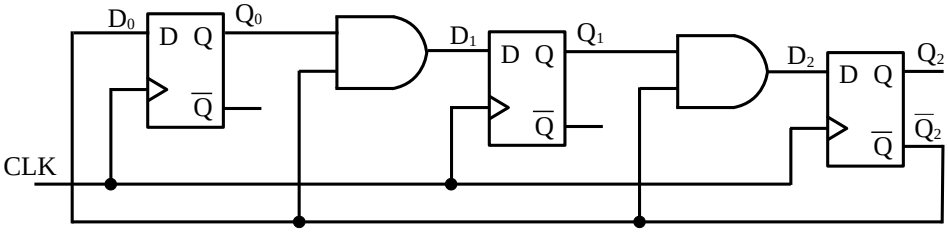
*Tabell 9.5. Nå/neste tilstandstabell for 3 bit teller.*

Vi kan bruke Karnaugh-diagram for å finne forenklede logiske uttrykk for de forskjellige datainngangene. Hver celle i Karnaugh-diagrammet svarer til nå-tilstanden i nå/neste tilstandstabellen. Igjen bruker vi - (bindestrek) der det ikke er av betydning om vi har 0 eller 1. I figur 9.8 er vist Karnaugh-diagrammene og de logiske uttrykkene for hver av datainngangene.




*Figur 9.8. Karnaugh-diagrammer og logiske uttrykk for 3 bit teller.*

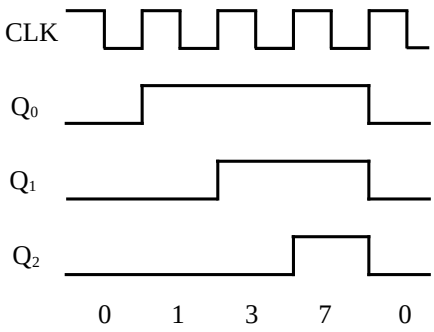
Når vi har funnet de logiske uttrykk for datainngangene, er neste trinn å tegne skjema med vippene og nødvendige logiske porter. Kretsskjemaet er vist i figur 9.9.




*Figur 9.9. Kretsdiagram for 3 bit teller.*

I skjemaet har vi brukt OG-porter. Vi kunne selvfølgelig valgt å bruke andre typer porter, for eksempel NOR- eller NAND-porter. Legg merke til at telleren skal være synkron. Det betyr at alle datavippene har samme klokke, kalt CLK.

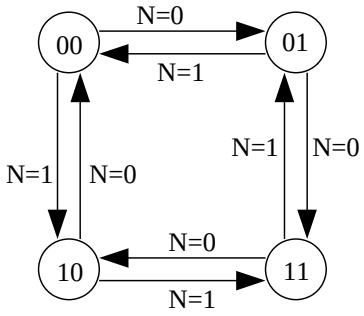
Det er alltid en god regel å verifisere et design for å kunne luke ut eventuelle feil. Til dette formål egner en simulator seg godt. I figur 9.10 er vist et tidsdiagram for telleren. Det ses at telleren gjennomløper 0 (000), 1 (001), 3 (011) og 7 (111) som ønsket.




*Figur 9.10. Tidsdiagram for 3 bit teller.*

## 9.5.2. 2 bit Gray opp/ned-teller

Vi ønsker å realisere en 2 bit synkron Gray opp/ned-teller bygd opp med JK-vipper som har tilstandsdiagrammet vist i figur 9.11.



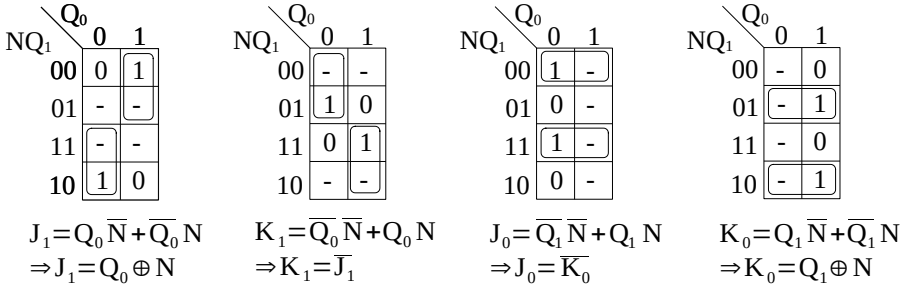

*Figur 9.11. Tilstandsdiagram for 2 bit Gray opp/ned-teller.*

Vi ønsker å telle opp for N = 0 og ned for N = 1. I tabell 9.6 er vist nå/neste tilstandstabell for denne Gray-telleren. I tabellen er også vist hva JK-inngangene på de to vippene må være for at vi skal kunne nå neste tilstand for telleren. Dette følger transisjonstabellen for JK-vippen presentert i tabell 9.2. Det fås to muligheter for neste tilstand avhengig av verdien på N.

| Tilstand | Nå-tilstand | | Inngang | Neste tilstand | | Vippe-innganger | | | |
|---|---|---|---|---|---|---|---|---|---|
| | $Q_1$ | $Q_0$ | N | $Q_1$ | $Q_0$ | $J_1$ | $K_1$ | $J_0$ | $K_0$ |
| 00 | 0 | 0 | 0 | 0 | 1 | 0 | - | 1 | - |
| | 0 | 0 | 1 | 1 | 0 | 1 | - | 0 | - |
| 01 | 0 | 1 | 0 | 1 | 1 | 1 | - | - | 0 |
| | 0 | 1 | 1 | 0 | 0 | 0 | - | - | 1 |
| 11 | 1 | 1 | 0 | 1 | 0 | - | 0 | - | 1 |
| | 1 | 1 | 1 | 0 | 1 | - | 1 | - | 0 |
| 10 | 1 | 0 | 0 | 0 | 0 | - | 1 | 0 | - |
| | 1 | 0 | 1 | 1 | 1 | - | 0 | 1 | - |

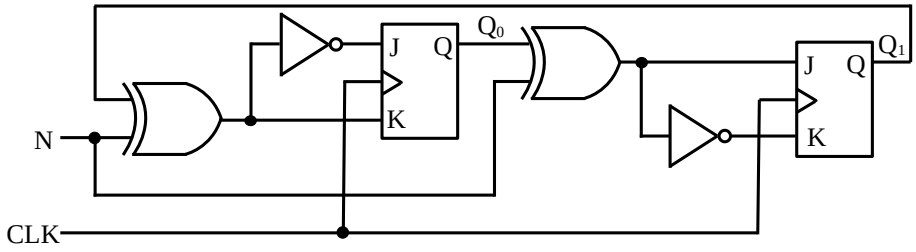
*Tabell 9.6. Nå/neste tilstandstabell for 2 bit Gray opp/ned-teller.*

Vi kan igjen bruke Karnaugh-diagram for å finne forenklede logiske uttrykk for de to JK-inngangene. Hver celle i Karnaugh-diagrammet svarer som før til nå-tilstanden i nå/neste tilstandstabellen. Igjen bruker vi - (bindestrek) der det ikke er av betydning om vi har 0 eller 1. I figur 9.12 er vist Karnaugh-diagrammene og de logiske uttrykkene for de to JK-inngangene.




*Figur 9.12. Karnaugh-diagrammer og logiske uttrykk for Gray-teller.*

Når vi har funnet de logiske uttrykk for JK-inngangene, gjenstår å tegne skjema med vippene og nødvendige logiske porter. Dette er gjort i figur 9.13.

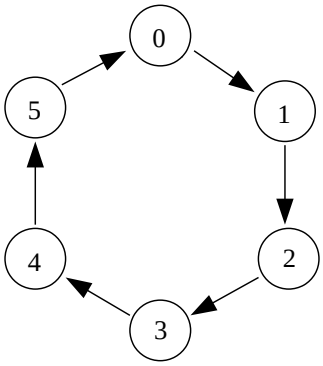



*Figur 9.13. Kretsdiagram for Gray-teller.*

I skjemaet har vi brukt Eksklusiv-ELLER-porter. Siden telleren skal være synkron, må begge JK-vippene ha samme klokke, i skjemaet kalt CLK.

## 9.5.3. Synkron mod-6 teller

Dersom vi modifiserer en 3 bit synkron binærteller, kan vi få en synkron mod-6 teller. Tilstandsdiagrammet til telleren vi ønsker å realisere med JK-vipper, er vist i figur 9.14.




*Figur 9.14. Tilstandsdiagram for mod-6 teller.*

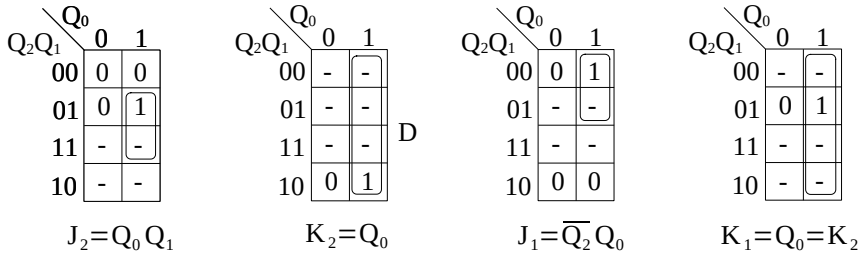
Denne telleren har bare en klokkeinngang. Den skal ha tre utganger og gjennomløpe tilstandene 0 til 5, angitt desimalt. Dette svarer da til binærverdiene 000 til 101.

Nå/neste-tabellen for denne telleren er vist i tabell 9.7. Vi har benyttet betegnelsene $Q_2$, $Q_1$ og $Q_0$ for vippeutgangene, der $Q_2$ er mest signifikante bit. I tabellen er også vist hva inngangene på de forskjellige JK-vippene må være for at vi skal kunne nå neste tilstand for telleren. Dette følger transisjonstabellen for JK-vippen presentert i tabell 9.2. Legg merke til at vi ikke har tatt med $J_0$ og $K_0$. Dette er fordi $Q_0$ skifter for hver klokkepuls, og det skjer dersom $J_0 = K_0 = 1$.

| Tilstand | Nå-tilstand | | | Neste tilstand | | | Vippe-innganger | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| | $Q_2$ | $Q_1$ | $Q_0$ | $Q_2$ | $Q_1$ | $Q_0$ | $J_2$ | $K_2$ | $J_1$ | $K_1$ |
| 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | - | 0 | - |
| 1 | 0 | 0 | 1 | 0 | 1 | 0 | 0 | - | 1 | - |
| 2 | 0 | 1 | 0 | 0 | 1 | 1 | 0 | - | - | 0 |
| 3 | 0 | 1 | 1 | 1 | 0 | 0 | 1 | - | - | 1 |
| 4 | 1 | 0 | 0 | 1 | 0 | 1 | - | 0 | 0 | - |
| 5 | 1 | 0 | 1 | 0 | 0 | 0 | - | 1 | 0 | - |

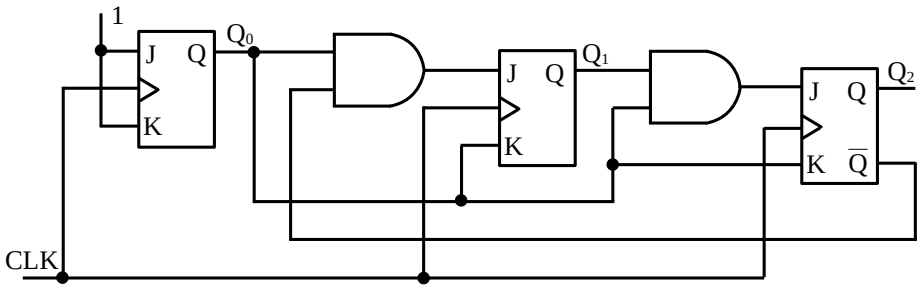
*Tabell 9.7. Nå/neste tilstandstabell for mod-6 teller.*

Igjen bruker vi Karnaugh-diagram for å finne logiske uttrykk for de forskjellige JK-inngangene. I figur 9.15 er vist Karnaugh-diagrammene og de logiske uttrykkene for JK-inngangene $J_2$, $K_2$, $J_1$ og $K_1$.




*Figur 9.15. Karnaugh-diagrammer og logiske uttrykk for mod-6 teller.*

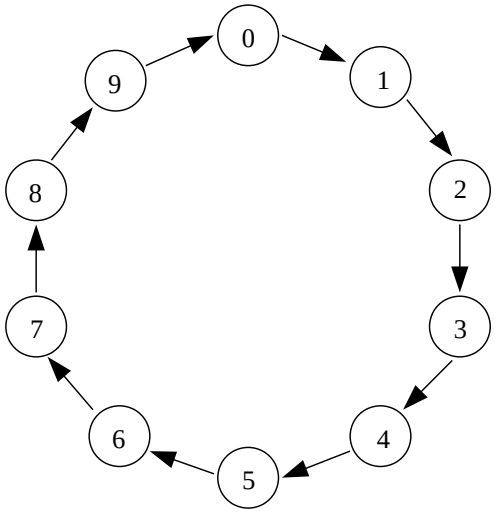
Skjema for mod-6 telleren er vist i figur 9.16. Legg merke til at $J_0 = K_0 = 1$. Siden telleren skal være synkron, har alle JK-vippene samme klokke, kalt CLK.




*Figur 9.16. Kretsdiagram for mod-6 teller.*

## 9.5.4. Synkron dekadeteller

Tilstandsdiagrammet til en synkron dekadeteller er vist i figur 9.17. Telleren skal ha en klokkeinngang, og vi ønsker å realisere denne telleren med JK-vipper.




*Figur 9.17. Tilstandsdiagram for dekadeteller.*

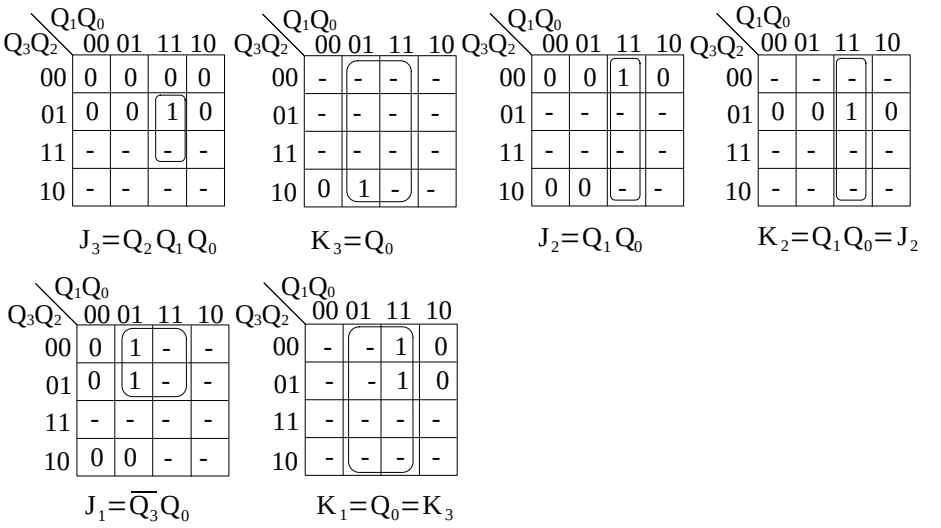
Nå/neste-tabellen for denne telleren er vist i tabell 9.8. Vi har benyttet betegnelsene $Q_3$, $Q_2$, $Q_1$ og $Q_0$ for vippeutgangene, der $Q_3$ er mest signifikante bit. I tabellen er også vist hva inngangene på de forskjellige JK-vippene må være for at vi skal kunne nå neste tilstand for telleren.

Dette følger transisjonstabellen for JK-vippen presentert i tabell 9.2. Legg igjen merke til at vi ikke har tatt med $J_0$ og $K_0$. Dette er fordi $Q_0$ skifter for hver klokkepuls, og det skjer dersom $J_0 = K_0 = 1$.

| Tilstand | Nå-tilstand | | | | Neste tilstand | | | | Vippe-innganger | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | $Q_3$ | $Q_2$ | $Q_1$ | $Q_0$ | $Q_3$ | $Q_2$ | $Q_1$ | $Q_0$ | $J_3$ | $K_3$ | $J_2$ | $K_2$ | $J_1$ | $K_1$ |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | - | 0 | - | 0 | - |
| 1 | 0 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | - | 0 | - | 1 | - |
| 2 | 0 | 0 | 1 | 0 | 0 | 0 | 1 | 1 | 0 | - | 0 | - | - | 0 |
| 3 | 0 | 0 | 1 | 1 | 0 | 1 | 0 | 0 | 0 | - | 1 | - | - | 1 |
| 4 | 0 | 1 | 0 | 0 | 0 | 1 | 0 | 1 | 0 | - | - | 0 | 0 | - |
| 5 | 0 | 1 | 0 | 1 | 0 | 1 | 1 | 0 | 0 | - | - | 0 | 1 | - |
| 6 | 0 | 1 | 1 | 0 | 0 | 1 | 1 | 1 | 0 | - | - | 0 | - | 0 |
| 7 | 0 | 1 | 1 | 1 | 1 | 0 | 0 | 0 | 1 | - | - | 1 | - | 1 |
| 8 | 1 | 0 | 0 | 0 | 1 | 0 | 0 | 1 | - | 0 | 0 | - | 0 | - |
| 9 | 1 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | - | 1 | 0 | - | 0 | - |

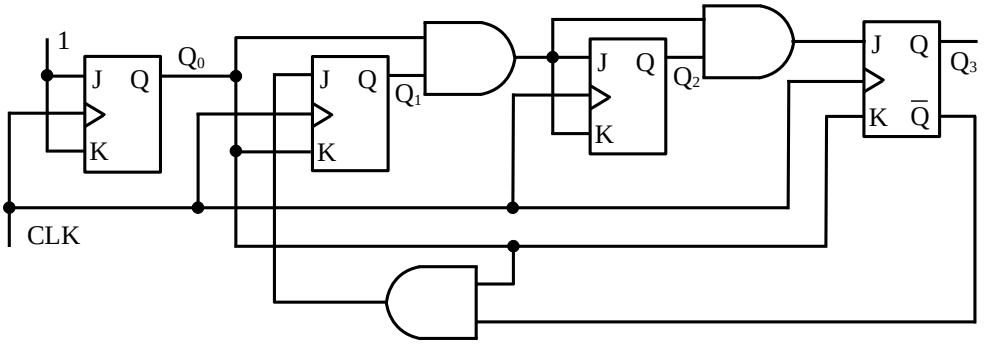
*Tabell 9.8. Nå/neste tilstandstabell for dekadeteller.*

Igjen bruker vi Karnaugh-diagram for å finne logiske uttrykk for de forskjellige JK-inngangene. I figur 9.15 er vist Karnaugh-diagrammene og de logiske uttrykkene for JK-inngangene $J_3$, $K_3$, $J_2$, $K_2$, $J_1$ og $K_1$.




*Figur 9.18. Karnaugh-diagrammer og logiske uttrykk for dekadeteller.*

Skjemaet for dekadetelleren er vist i figur 9.19. Legg igjen merke til at $J_0 = K_0 = 1$. Siden telleren skal være synkron, har alle JK-vippene samme klokke, kalt CLK.



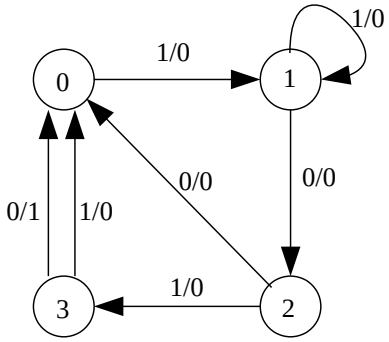

*Figur 9.19. Kretsskjema for synkron dekadeteller.*

# 9.6. Design av tilstandsmaskiner

## 9.6.1. Mønstergjenkjenner

I figur 9.20 er vist tilstandsdiagram for realisering av en mønstergjenkjenner. Mønstergjenkjennere benyttes mye i seriell kommunikasjon for å kunne finne fram til starten av en datasekvens. I praksis vil bitmønsteret være på ganske mange bit. Det vil også repeteres med jevne mellomrom. Her vil vi se på et meget enkelt eksempel der vi bare ønsker å detektere bitmønsteret 1010. Når dette mønsteret er funnet, skal vi la utgangen fra mønstergjenkjenneren bli lik 1. Vi ønsker her å benytte JK-vipper for realiseringen av denne mønstergjenkjenneren, men realisering med D-vipper er selvfølgelig også mulig.

Vi starter i tilstand 0, se tilstandsdiagrammet i figur 9.20. Når første ener opptrer i data vi mottar, skifter vi over til tilstand 1. I tilstandsdiagrammet er dette vist som 1/0, der 0 i 1/0 betyr at utgangen fra tilstandsmaskinen er lik 0. Dersom vi mottar en ny ener, vil vi bli stående i tilstand 1 mens utgangen fortsatt er lik 0. Mottar vi 0 (når vi står i tilstand 1), vil vi skifte over til tilstand 2 mens utgangen fortsatt er lik 0.




*Figur 9.20. Tilstandsdiagram for mønstergjenkjenner.*

Mottar vi så en 0, må vi starte nytt søk etter mønsteret, vi går da tilbake til tilstand 0 (og utgangen er selvfølgelig lik 0). Mottar vi derimot en ener, skifter vi over til tilstand 3 (og utgangen er fortsatt lik 0). Mottar vi så en ny ener, har vi ikke mottatt mønsteret, og vi må gå da tilbake til tilstand 0 (og utgangen er selvfølgelig lik 0).

Mottar vi derimot en 0, har vi detektert mønsteret og vi setter utgangen lik 1. Vi går så tilbake til tilstand 0 for å påbegynne et nytt søk.

I tabell 9.9 er vist nå/neste tilstandstabell for mønstergjenkjenneren. Vi har benyttet betegnelsene $Q_1$ og $Q_0$ for vippeutgangene, der $Q_1$ er mest signifikante bit. I tabellen er også vist hva inngangene på de to JK-vippene må være for at vi skal kunne nå neste tilstand for mønstergjenkjenneren. Dette følger transisjonstabellen for JK-vippen presentert i tabell 9.2.

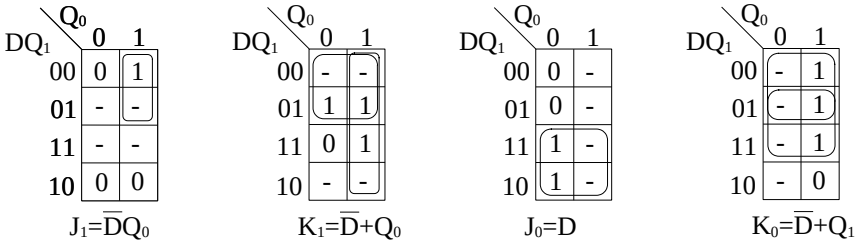
Utgangen fra mønstergjenkjenneren har vi kalt L som skal være lik 1 når riktig mønster i inn-data, kalt D, er funnet.

| Tilstand | Nå-tilstand | | Data inn | Neste tilstand | | Vippe-innganger | | | | Utgang |
|---|---|---|---|---|---|---|---|---|---|---|
| | $Q_1$ | $Q_0$ | D | $Q_1$ | $Q_0$ | $J_1$ | $K_1$ | $J_0$ | $K_0$ | $L_M$ |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | - | 0 | - | 0 |
| | 0 | 0 | 1 | 0 | 1 | 0 | - | 1 | - | 0 |
| 1 | 0 | 1 | 0 | 1 | 0 | 1 | - | - | 1 | 0 |
| | 0 | 1 | 1 | 0 | 1 | 0 | - | - | 0 | 0 |
| 2 | 1 | 0 | 0 | 0 | 0 | - | 1 | 0 | - | 0 |
| | 1 | 0 | 1 | 1 | 1 | - | 0 | 1 | - | 0 |
| 3 | 1 | 1 | 0 | 0 | 0 | - | 1 | - | 1 | 1 |
| | 1 | 1 | 1 | 0 | 0 | - | 1 | - | 1 | 0 |

*Tabell 9.9. Nå/neste tilstandstabell for mønstergjenkjenner.*

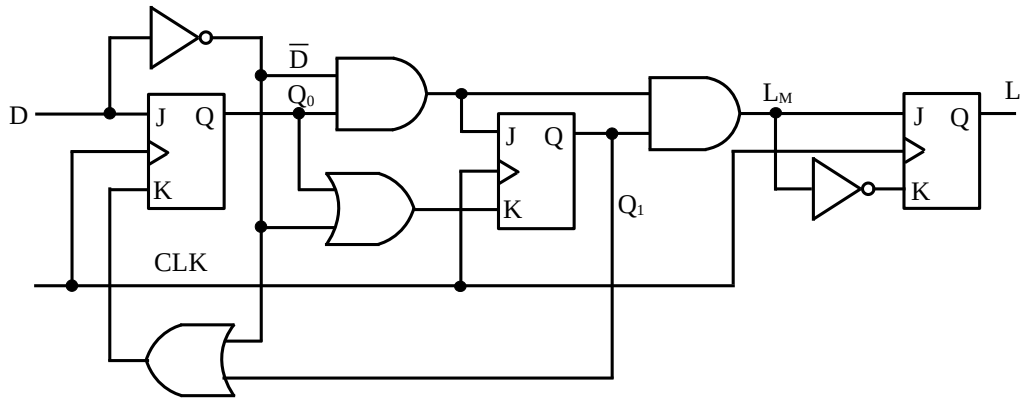
Det ses at $L_M = Q_1Q_0\overline{D}$ fra tabellen. Ved neste klokkepuls fås da L. Ellers kan vi igjen bruke Karnaugh-diagram for å finne forenklede logiske uttrykk for de to JK-inngangene. Hver celle i Karnaugh-diagrammet svarer som før til nå-tilstanden i nå/neste tilstandstabellen. Igjen bruker vi - (bindestrek) der det ikke er av betydning om vi har 0 eller 1.

I figur 9.21 er vist Karnaugh-diagrammene og de logiske uttrykkene for de to JK-inngangene.



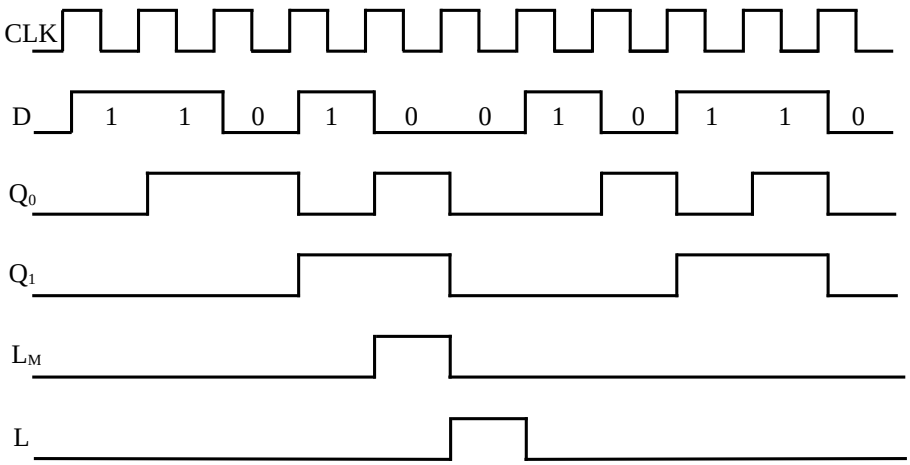

*Figur 9.21. Karnaugh-diagrammer for mønstergjenkjenner.*

Skjemaet for mønstergjenkjenneren er vist i figur 9.22. Bemerk at JK-vippene klokkes synkront med samme klokke, her kalt CLK.




*Figur 9.22. Realisering av mønstergjenkjenner.*

I figur 9.23 er vist tidsdiagram for mønstergjenkjenneren med et datasignal D som først inneholder mønsteret slik at vi får utgangen L = 1, men der mønsteret 1011 senere ikke gir deteksjon av mønsteret (L = 0). I tidsdiagrammet er forøvrig D antatt å ligge etter positiv flanke på CLK.




*Figur 9.23. Tidsdiagram for mønstergjenkjenner.*

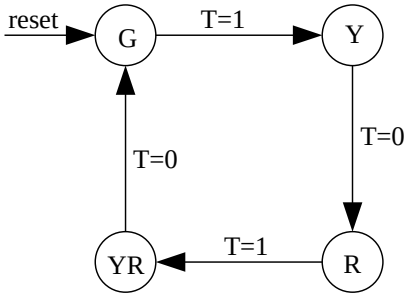
Forøvrig kan denne mønstergjenkjenneren kategoriseres som en Mealy tilstandsmaskin siden utgangen L også avhenger av datainngangen D.

Vi har her benyttet to vipper for de fire tilstandene. Det er også mulig å benytte en vippe for hver tilstand, dette kalles på engelsk for 'one hot' koding. Siden det er mange ubrukte tilstander, kan det gi enklere logikkrealiseringer i noen tilfeller.

Det bør bemerkes at det ikke er alle mønstre som lar seg realisere med bare fire tilstander selv om dataordet bare er på fire bit. Dette gjelder for eksempel mønsteret 1111.

### 9.6.2. Trafikklys-styring

Som et siste eksempel på tilstandsmaskiner, har vi tatt med et eksempel med trafikklys-styring. For å begrense kompleksiteten, har vi begrenset oppgaven til bare ett trafikkfyr. I figur 9.24 er vist tilstandsdiagrammet. Betegnelsen G, Y og R brukes for henholdsvis grønt, gult og rødt lys. Et resettings-signal sørger for at vi starter i tilstand G. Det benyttes et klokkesignal T for tidsstyringen av trafikkfyret.




*Figur 9.24. Tilstandsdiagram for trafikklys-styring.*

Når T = 1 skiftes fra G til Y. Signalet T forutsettes være 1 bare kort tid (i praksis ca 3 sekunder), se figur 9.27 for et eksempel på utseendet på dette klokkesignalet. Fra tilstand Y skiftes til tilstand R, som vi har forutsatt her skal vare like lenge som tilstand G. Etter tilstand R skiftes til tilstand YR i en kort tid før tilstand G igjen nås.

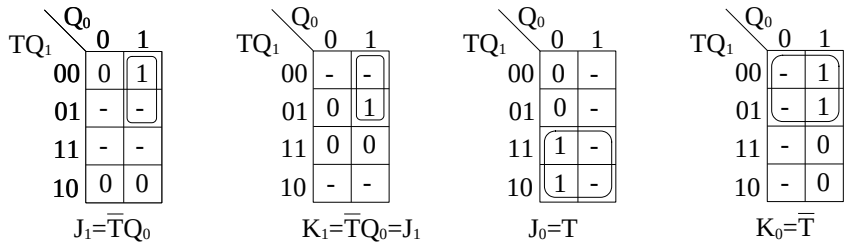
I tabell 9.10 er vist nå/neste tilstandstabell for trafikklys-styringen. Vi har benyttet betegnelsene $Q_1$ og $Q_0$ for vippeutgangene, der $Q_1Q_0 = 00$ svarer til tilstand G, $Q_1Q_0 = 01$ svarer til tilstand Y, $Q_1Q_0 = 10$ svarer til tilstand R og $Q_1Q_0 = 11$ svarer til tilstand YR.

Siden vi ønsker å bruke JK-vipper til realiseringen av trafikklys-styringen, har vi i tabellen også vist hva inngangene på de to JK-vippene må være for at vi skal kunne nå neste tilstand fra en gitt tilstand. Dette følger transisjonstabellen for JK-vippen presentert i tabell 9.2. Vi har tre utganger fra trafikklys-styringen: G, R og Y som vi setter lik 1 når lyset skal være på.

| Tilstand | Nå-tilstand | | Tid | Neste tilstand | | Vippe-innganger | | | | Utganger | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | $Q_1$ | $Q_0$ | T | $Q_1$ | $Q_0$ | $J_1$ | $K_1$ | $J_0$ | $K_0$ | G | Y | R |
| G | 0 | 0 | 0 | 0 | 0 | 0 | - | 0 | - | 1 | 0 | 0 |
| | 0 | 0 | 1 | 0 | 1 | 0 | - | 1 | - | 1 | 0 | 0 |
| Y | 0 | 1 | 0 | 1 | 0 | 1 | - | - | 1 | 0 | 1 | 0 |
| | 0 | 1 | 1 | 0 | 1 | 0 | - | - | 0 | 0 | 1 | 0 |
| R | 1 | 0 | 0 | 1 | 0 | - | 0 | 0 | - | 0 | 0 | 1 |
| | 1 | 0 | 1 | 1 | 1 | - | 0 | 1 | - | 0 | 0 | 1 |
| YR | 1 | 1 | 0 | 0 | 0 | - | 1 | - | 1 | 0 | 1 | 1 |
| | 1 | 1 | 1 | 1 | 1 | - | 0 | - | 0 | 0 | 1 | 1 |

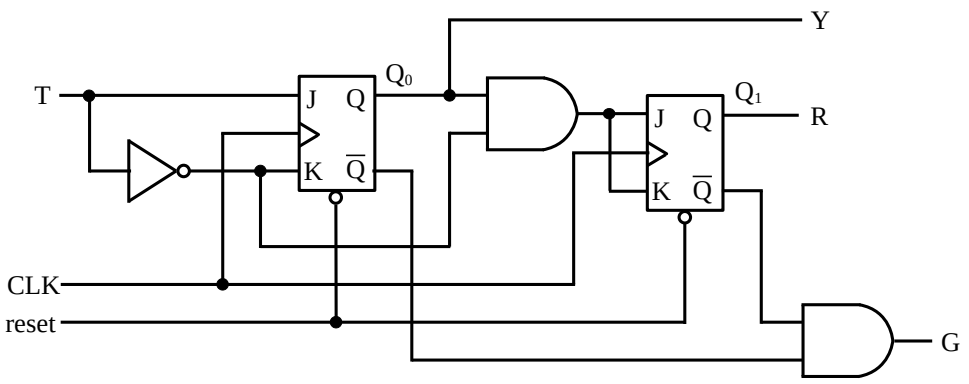
*Tabell 9.10. Nå/neste tilstandstabell for trafikklys-styring.*

Vi kan igjen bruke Karnaugh-diagram for å finne forenklede logiske uttrykk for de to JK-inngangene. Hver celle i Karnaugh-diagrammet svarer som før til nå-tilstanden i nå/neste tilstandstabellen. Igjen bruker vi - (bindestrek) der det ikke er av betydning om vi har 0 eller 1. I figur 9.25 er vist Karnaugh-diagrammene og de logiske uttrykkene for de to JK-inngangene.



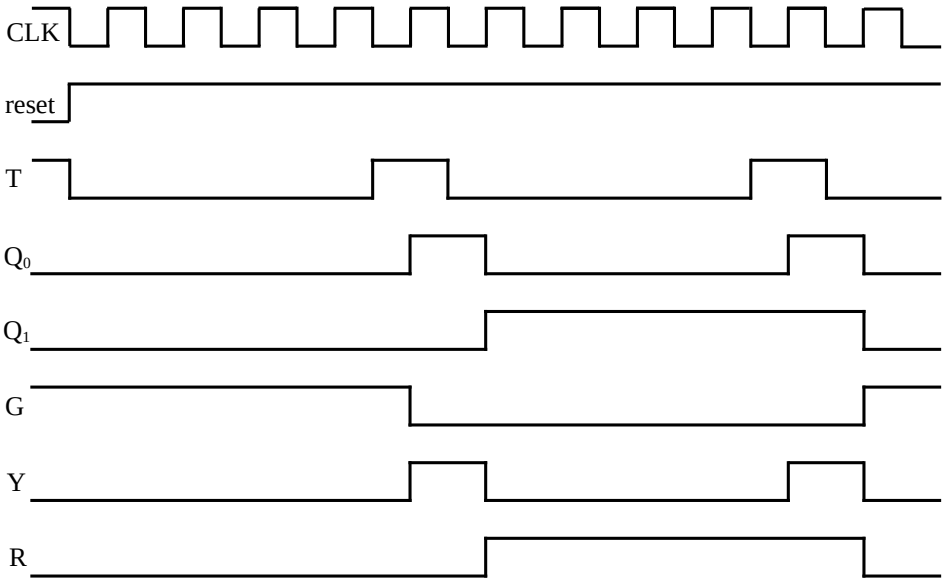

*Figur 9.25. Karnaugh-diagrammer for trafikklys-styring.*

Skjemaet for trafikklys-styringen er vist i figur 9.26. Bemerk at JK-vippene klokkes synkront med samme klokke, her igjen kalt CLK.




*Figur 9.26. Skjema for trafikklys-styring.*

I figur 9.27 er vist tidsdiagram for trafikklys-styringen. Resettingssignalet RST er antatt aktivt lavt slik at vi er garantert å starte i tilstand G. Klokkesignalet T er lik 1 mye kortere tid enn lik 0 for at tilstand Y og YR skal vare mye kortere enn tilstand G og R. Utgangene G, Y og R er som nevnt aktivt høye.




*Figur 9.27. Tidsdiagram for trafikklys-styring.*

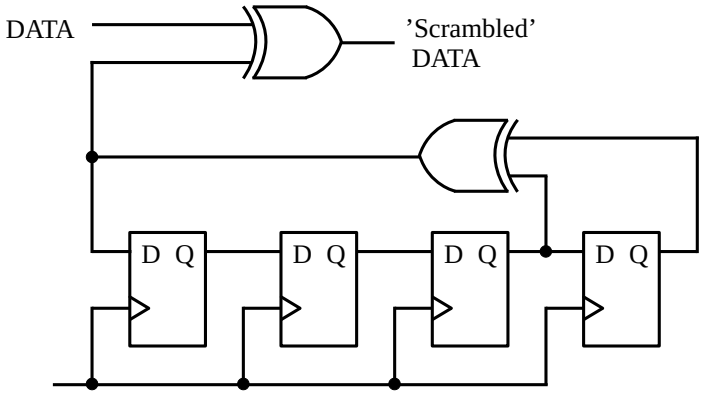
# Kapittel 10
# Spesialkretser

## 10.1. Innledning

I dette kapitlet beskrives noen kretser for realisering av forskjellige funksjoner. Vi kan referere til disse som byggeklosser for å lage digitale systemer. Karakteristisk for disse spesialkretsene er at de vil være oppbygd som kombinatoriske og sekvensielle kretser (med hukommelse).

## 10.2. Scrambler og Descrambler

I kapittel 7.10 ble en mønstergenerator basert på et tilbakekoplet skiftregister beskrevet. En slik mønstergenerator kan brukes til å modifisere en seriell datastrøm. Dette er vist i figur 10.1, der mønstergeneratoren føder en eksklusiv-ELLER-port som på sin andre inngang har en seriell datastrøm.




*Figur 10.1. Data Scrambler.*

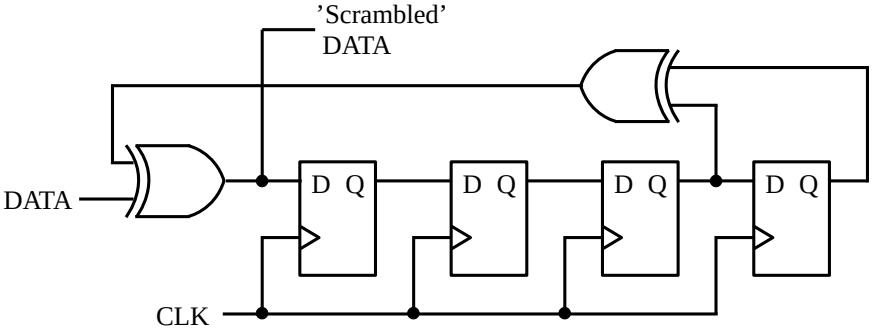
Denne typen kalles additiv scrambler. Det skjønnes at data som er repetitiv, vil bli mer vilkårlig. Dermed forhindres gjentakelse av spesielle mønstre som kan være et problem ved en overføring av serielle datastrømmer. Som eksempel kan nevnes store effekt-topper ved trådløs overføring som kan føre til interferens, uheldig påvirkning, av andre signaler. En fordel er også at lange strømmer med bare enere eller nullere fjernes. Dette er til stor hjelp når en en mottaker bruker mottatt datastrøm til klokke-regenerering, det vil si gjenskape klokken som ble brukt på sendersiden.

En descrambler i mottakeren vil ha samme skjema som vist i figur 10.1, men inngangen vil nå være 'Scrambled' DATA istedenfor DATA, og det som finnes på utgangen av øverste Eksklusiv-ELLER-port vil være sendte DATA istedenfor 'Scrambled' DATA.

En slik scrambler som dette krever at mottaker og sender er synkroniserte. Det kan for eksempel oppnås ved å sette alle vippene lik 1 i starten av en ramme. En ramme kan så bygges opp ved at begynnelsen på denne er et synkroniseringsord, se figur 10.2. Det kan være forskjellig innhold i slike rammer, og foruten data kan de inneholde for eksempel senderadresse, mottakeradresse, type data og ekstra felt som kan brukes til feildetektering med videre.

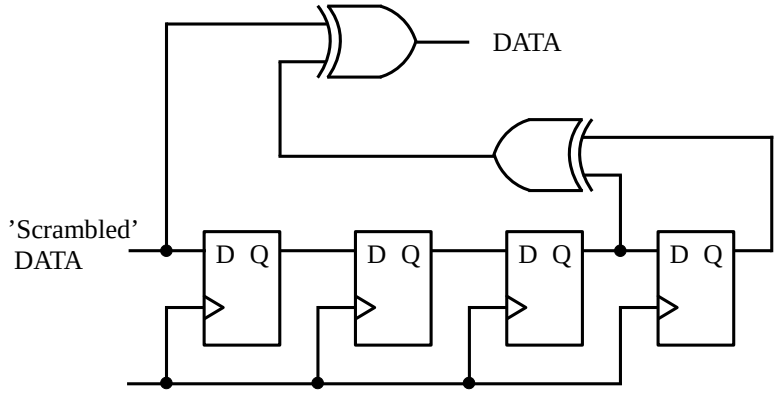

*Figur 10.2. Rammeoppbygging.*

Det finnes også selvsynkroniserende scramblere. Ved å omarrangere figur 10.1, får vi en slik i figur 10.3. Det ses at 'scrambled' DATA nå også føres inn på skiftregisteret.




*Figur 10.3. Selvsynkroniserende scrambler.*

En selvsynkroniserende descrambler er vist i figur 10.4. Det ses at mottatt 'Scrambled' DATA føres direkte til skiftregisteret mens DATA tas ut av den øverste porten.




*Figur 10.4. Selvsynkroniserende descrambler.*

Også ved scramblere og descramblere er det vanlig å benytte polynomer for å beskrive virkemåten. For scrambleren i figur 10.3 has følgelig:

$$p(x) = 1 + x^{-3} + x^{-4} \tag{10.1}$$

Da svarer 3 i eksponenten til at 'scrambled' DATA er forsinket tre perioder mens 4 i eksponenten svarer til fire perioders forsinkelse. Tallet 1 i polynomet svarer til ingen forsinkelse ($x^0 = 1$).

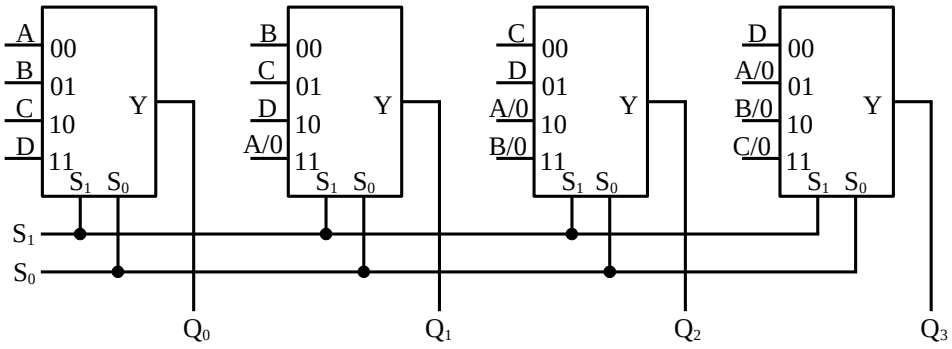
Til slutt skal nevnes at et problem med selvsynkroniserende scramblere er at en feil i mottatte data kan føre til flere feil fra descrambleren. Det kan også fås inndata som fører til at alle vippeutganger blir lik 0 og scramblingen opphører. Det må da has egne deteksjonskretser for å rette på dette.

## 10.3. Barrel Shifter

En Barrel Shifter kan skifte et dataord med et spesifisert antall bit uten å bruke sekvensiell logikk. Ofte ses den som en nyttig komponent i en aritmetisk logisk enhet, ALU ('Arithmetic Logic Unit', se kapittel 10.8) i mikrokontrolleren (se kapittel 11). Da kan den brukes til å skifte og rotere et programmert antall bit, gjerne innenfor bare en klokkeperiode.

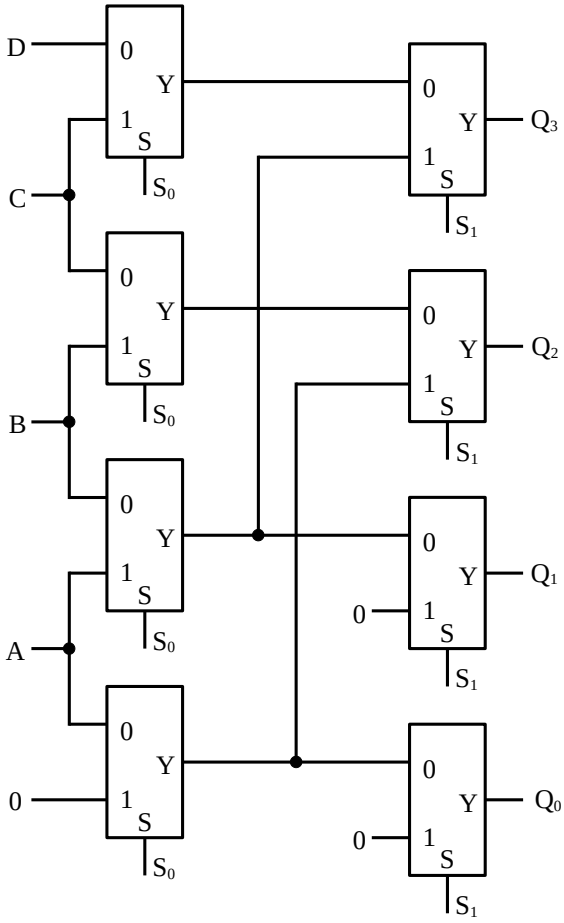
For eksempel kan en fire bit Barrel Shifter med inngangene A, B, C og D endre rekkefølgen til DABC, CDAB eller BCDA. Da benyttes høyre-rotering uten tap av data. Det kan også være aktuelt med venstre-rotering til BCDA, CDAB eller DABC. Noen ganger vil en bare være interessert i skifting og ikke rotering. En vanlig måte å realisere kretsen på, er å bruke multipleksere.

Et eksempel på en 4 bit Barrel Shifter er vist i figur 10.5 med bruk av fire 4-1 multipleksere. I figuren er vist venstre-skift/rotering, der skift medfører at bit «etterfylles» med 0. Valg av antall skift gjøres med styresignalene $S_1$ og $S_0$ slik at 00 gir ut ABCD for $Q_0Q_1Q_2Q_3$ mens 01 gir ut BCD og A/0 avhengig om en ønsker skifte eller rotering. Tilsvarende fås CDAB eller CD00 for $S_1S_0$ lik 10 og DABC eller D000 for $S_1S_0$ lik 11.




*Figur 10.5. 4 bit Barrell Shifter med venstreskift.*

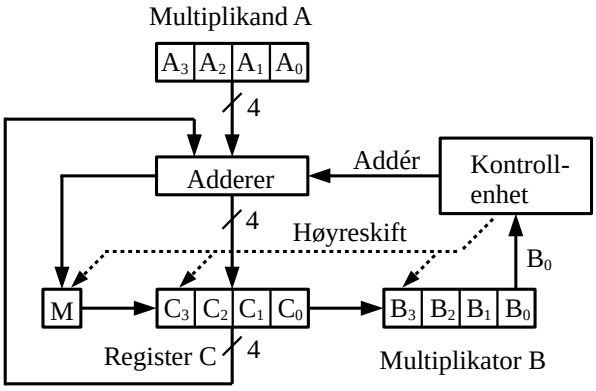
I figur 10.6 er vist et eksempel med en fire bit Barrel Shifter der det benyttes 2-1 multipleksere koplet i kaskade. Det ses at valget med $S_1S_0$ lik 00 gir ut ABCD for $Q_0Q_1Q_2Q_3$ mens $S_1S_0$ lik 01 gir ut 0ABC. Videre fås 00AB og 000A for $S_1S_0$ henholdsvis lik 10 og 11. Det er forøvrig forholdsvis vanlig med kaskadekopling av multipleksere for større ordlengder.




*Figur 10.6. 4 bit Barrel Shifter med høyreskift.*

## 10.4. Multiplikasjons-krets

I figur 10.7 er vist realisering av en seriell multiplikasjons-krets for multiplikasjon av to heltall (uten fortegn). Kretsen skal multiplisere to fire bit ord, multiplikanden A og multiplikatoren B. Den inneholder videre en adderer for to fire bit ord som kan gi ut et mente-bit M, hvis påkrevet.




*Figur 10.7. Seriell multiplikasjons-krets.*

Det er også et fire bit register C og en kontrollenhet som er «hjernen» i multiplikasjons-kretsen. Før multiplikasjonen resettes M-bit og C-registeret. Under multiplikasjonsprosessen vil bit $B_0$ (minst signifikante bit i multiplikatoren B) bestemme om multiplikanden skal adderes til produktet (i C-registeret) for hvert trinn.

Takten bestemmes av kontrollenheten som styres av en klokke (ikke vist i figuren). Etter at multiplikanden eventuelt er addert til produktet, skiftes registeret (samt mente) og multiplikatoren en plass til høyre. Dette svarer til at multiplikanden skiftes en plass til venstre.

I tabell 10.1 er vist et eksempel på en multiplikasjonsprosess der multiplikanden er 1011 (desimalt 11) og multiplikatoren er 1101 (desimalt 13). Initielt er mente-bit M og C-registeret nullstilt mens multiplikatoren er lagt i B-registeret.

Minst signifikante bit i multiplikatoren ($B_0$) er 1 slik at er multiplikanden A er addert til C-registeret. Så skiftes registrene M, C og B til høyre. Nå er nest minst signifikante bit i B-registeret ($B_1$) lik 0 slik at det ikke blir noen addisjon, men bare høyreskift.

Neste gang er nest mest signifikante bit i B-registeret ($B_2$) lik 1 slik at vi får addisjon med påfølgende høyreskift. Dette gjentas en gang til før produktet på 8 bit lik 10001111 (desimalt 143) blir stående i C- og B-registeret, der de fire mest signifikante bit står i C-registeret og de fire minst signifikante bit står i B-registeret. Antall trinn i multiplikasjonsprosessen bestemmes av hvor mange bit det er i multiplikatoren (her fire).

Multiplikand A: 1 0 1 1

| M | C | B | |
|---|---|---|---|
| 0 | 0 0 0 0 | 1 1 0 1 | Initialverdier |
| 0 | 1 0 1 1 | 1 1 0 1 | Addér A til C |
| 0 | 0 1 0 1 | 1 1 1 0 | Høyreskift |
| 0 | 0 0 1 0 | 1 1 1 1 | Høyreskift uten addisjon |
| 0 | 1 1 0 1 | 1 1 1 1 | Addér A til C |
| 0 | 0 1 1 0 | 1 1 1 1 | Høyreskift |
| 1 | 0 0 0 1 | 1 1 1 1 | Addér A til C |
| 0 | 1 0 0 0 | 1 1 1 1 | Høyreskift |
| | 1 0 0 0 | 1 1 1 1 | Produkt |

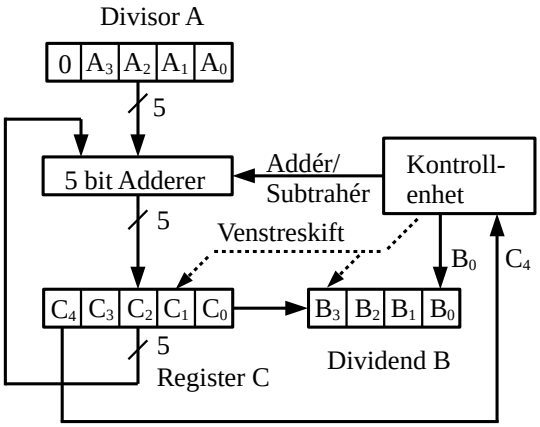
*Tabell 10.1. Multiplikasjonsprosess med seriell multiplikasjons-krets.*

## 10.5. Divisjons-krets

I figur 10.8 er vist realisering av en seriell divisjons-krets. Denne deler en dividend med en divisor, der begge er heltall (uten fortegn). I motsetning til multiplikasjons-kretsen skal divisjonskretsen subtrahere istedenfor å addere. Ellers skal kretsen operere med venstreskift istedenfor høyreskift som for multiplikasjonskretsen.

Kretsen i figuren inneholder en fem bit adderer og et fire bit register (B) for dividenden. Kretsen inneholder også et fem bits register (A) for divisor og et fem bits rest-register (C). Det er nødvendig med fem bit fordi det trenges et ekstra bit for å angi fortegnet til mellomresultatet siden rest etter subtraksjonen kan være negativ.

Dersom mest signifikante bit ($C_4$) i rest-registeret er lik 1, betyr det at subtraksjonen mellom C-registeret og divisoren er negativ. Da må C-registeret tilbakestilles, gjennomrettes, ved at divisor adderes til dette registeret.




*Figur 10.8. Seriell divisjons-krets.*

Dersom mest signifikante bit ($C_4$) i rest-registeret er lik 0, betyr det at subtraksjonen mellom C-registeret og divisoren er positiv. Da lykkes divisjonen og minst signifikante bit i dividenden ($B_0$) kan settes lik 1.

I tabell 10.2 er vist et eksempel på en divisjonsprosess der divisor er 0011 (desimalt 3) og dividenden er 1010 (desimalt 10). Initielt er C-registeret nullstilt og kontrollenheten styrer prosessen til å foregå over fire trinn (lik ordlengden).

Prosessen begynner med venstreskift. Deretter subtraheres divisor (A) fra C. Det benyttes toer-komplement, der $-3_{10} = 11101_2$. Resultatet er negativt idet $C_4 = 1$. Da må C-registeret gjenopprettes og dette gjøres ved å addere divisoren til registeret. Det foretas et nytt venstreskift. Igjen er resultatet negativt, og C-registeret må også nå gjenopprettes.

Etter neste venstreskift og subtraksjon, er innholdet i C-registeret positivt. Da blir det ingen gjenoppretting, men minst signifikante bit i dividenden ($B_0$) settes lik 1. Ved siste venstreskift gir subtraksjonen igjen et positivt resultat og $B_0$ settes lik 1. Vi står da igjen med dividenden (B) lik 0011 (3 desimalt) og en rest på 0001 i $C_3$-$C_0$.

Divisor A: 0 0 0 1 1

| C | | | | | B | | | | |
|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 1 | 0 | 1 | 0 | Initialverdier |
| 0 | 0 | 0 | 0 | 1 | 0 | 1 | 0 | 0 | Venstreskift |
| 1 | 1 | 1 | 1 | 0 | 0 | 1 | 0 | 0 | Subtrahér A fra C |
| 0 | 0 | 0 | 0 | 1 | 0 | 1 | 0 | 0 | Gjenopprett C (Addér A til C) |
| 0 | 0 | 0 | 0 | 1 | 0 | 1 | 0 | 0 | Resett $B_0$ |
| 0 | 0 | 0 | 1 | 0 | 1 | 0 | 0 | 0 | Venstreskift |
| 1 | 1 | 1 | 1 | 1 | 1 | 0 | 0 | 0 | Subtrahér A fra C |
| 0 | 0 | 0 | 1 | 0 | 1 | 0 | 0 | 0 | Gjenopprett C (Addér A til C) |
| 0 | 0 | 0 | 1 | 0 | 1 | 0 | 0 | 0 | Resett $B_0$ |
| 0 | 0 | 1 | 0 | 1 | 0 | 0 | 0 | 0 | Venstreskift |
| 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | Subtrahér A fra C |
| 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 1 | Sett $B_0$ |
| 0 | 0 | 1 | 0 | 0 | 0 | 0 | 1 | 0 | Venstreskift |
| 0 | 0 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | Subtrahér A fra C |
| 0 | 0 | 0 | 0 | 1 | 0 | 0 | 1 | 1 | Sett $B_0$ |
| | Rest | | | | | Kvotient | | | |

*Tabell 10.2. Divisjonsprosess med seriell divisjons-krets.*

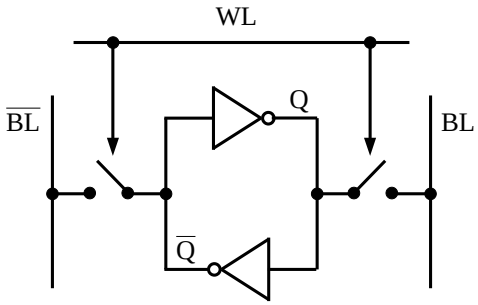
## 10.6. RAM

### 10.6.1. Innledning

RAM er akronym for Random Access Memory. RAM benyttes for lagring av data i for eksempel mikrokontrollere og datamaskiner. Data kan skrives til og leses fra en hvilken som helst valgt adresse til et vilkårlig tidspunkt. Dette er en type halvlederbasert minne som er avhengig av strømtilførsel. Når spenningen forsvinner, forsvinner også data.

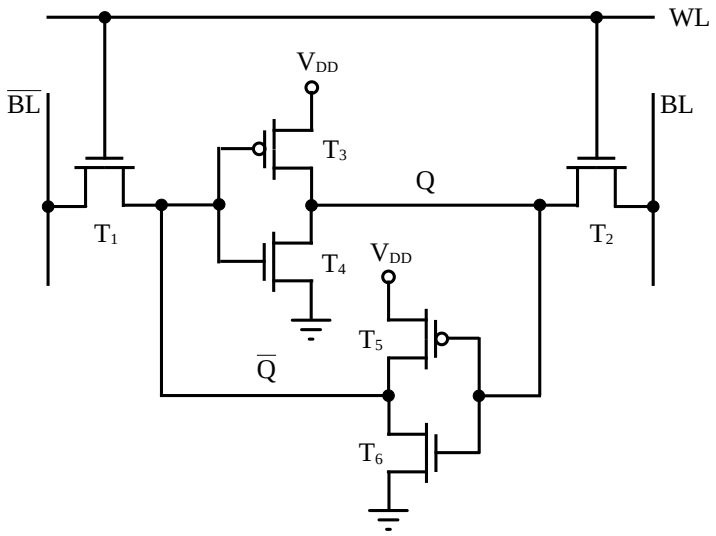
## 10.6.2. SRAM

SRAM er akronym for 'Static RAM'. Figur 10.9 viser prinsipp for en SRAM-celle. Den består av to inverterere samt to brytere. Cellen er tilkoplet en adressebuss (Word line, WL) og en databuss (Bit Line, BL). Dersom cellen ikke er valgt (med brytere som vist), vil den beholde sin verdi. Dersom cellen velges (brytere legges inn), kan cellen skrives til (settes til 0 eller 1) eller leses fra. Verdien vil da ligge på databussen. (med verdien til BL).



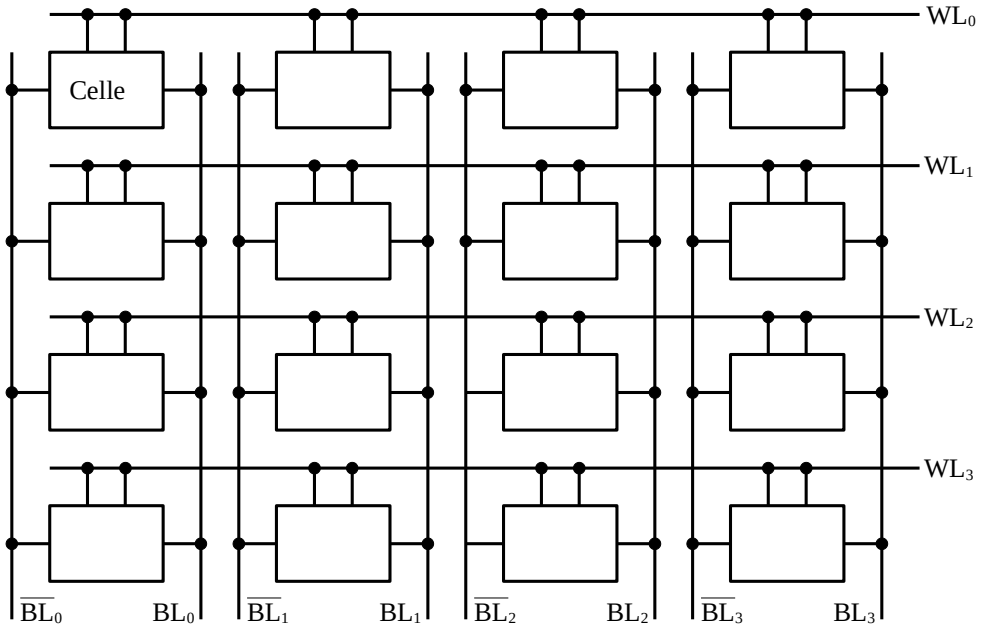

*Figur 10.9. Prinsipp for SRAM-celle.*

I figur 10.10 er vist en praktisk realisering av cellen ovenfor med 6 transistorer. Inverterne er realisert med transistorene $T_3/T_4$ og $T_5/T_6$. Bryterne er realisert med $T_1$ og $T_2$ som slås på når WL = 1 og av når WL = 0.




*Figur 10.10. Oppbygging av SRAM-celle.*

I figur 10.11 er vist organisering av en 4x4 matrise med cellen ovenfor. Adressebussen er benevnt $WL_0$-$WL_3$ mens databussen er benevnt $BL_0$-$BL_3$. I praksis vil antallet celler være 256x256 (8 bit data og adresse) eller mer.




*Figur 10.11. 4x4 SRAM matrise.*

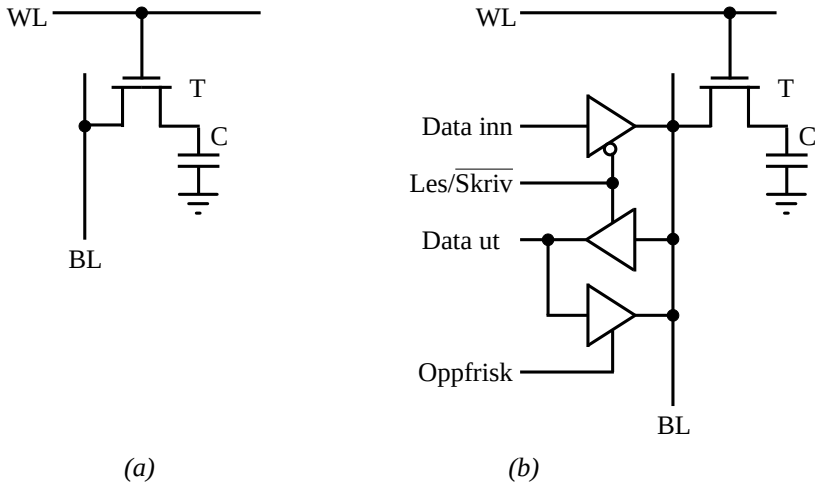
## 10.6.3. DRAM

DRAM er akronym for 'Dynamic RAM'. En DRAM-celle bruker en kondensator til lagring og en transistor som bryter, se figur 10.12a. Enkelheten gjør at en DRAM-celle kan gjøres mindre enn en SRAM-celle. Ulempen er at opp- og utladning av kondensatoren tar en viss tid slik at DRAM blir tregere enn SRAM.

Med referanse til figur 10.12 ses at adresseringen er med WL mens data vil foreligge på BL, som for SRAM-cellen. Når WL er høy, vil transistoren T slås på. Skal cellen skrives til, legges Les/$\overline{\text{Skriv}}$ lav, se figur 10.12b. Er data lik 1, kan kondensatoren lades opp (for eksempel til $V_{DD}$). Er data lik 0, kan kondensatoren lades ut (til 0 V). Når transistoren slås av, vil kondensatoren holde den spenningen den da var opp/ut-ladet til.

Tilsvarende kan cellen leses fra ved å legge WL høy og Les/$\overline{\text{Skriv}}$ høy. Det brukes 3-nivå buffer for kopling til BL for Data inn og Data ut.

Problemet er imidlertid at det vil være små lekkstrømmer slik at kondensatoren må «oppfriskes» slik at data ikke går tapt. Dette er illustrert i figur 10.12b med en oppfriskings-kommando. Da leses verdien på kondensatoren (Data ut) med Les/$\overline{\text{Skriv}}$ høy, spenningen forsterkes og kondensatoren etterfylles med denne oppfriskede spenningen.




*Figur 10.12. DRAM-celle (a) med databuss-styring (b).*

Nødvendigheten med oppfrisking gjør at DRAM-minner får en mer kompleks oppbygging enn SRAM-minner. Det finnes flere metoder for å kunne oppfriske cellene uten at dette går ut over aksess-tiden. Imidlertid gjør størrelsen på hver DRAM-celle at arealeffektiviteten er større for DRAM enn for SRAM, og DRAM benyttes derfor i utstrakt grad. Som nevnt er ulempen for både SRAM og DRAM at disse minnekretsene er flyktige: Når spenningen forsvinner, mistes også data. Denne typen minne benyttes derfor til korttidsminne.

## 10.7. ROM

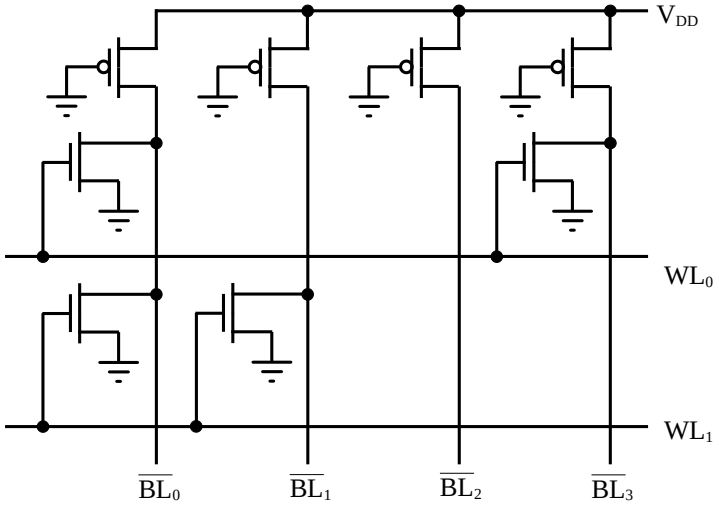
### 10.7.1. Innledning

ROM er akronym for Read Only Memory. I motsetning til RAM kan ROM bare skrives til et begrenset antall ganger. Den vanligste formen er ROM som enten prefabrikkeres med et fast innhold i hver celle eller programmeres én gang av brukeren (Programmable ROM, PROM). EPROM (Erasable PROM) er en type som kan nullstilles og programmeres igjen. Ultrafiolett lys ble tidligere brukt til å nullstille koblingspunkter. EEPROM (Electrical EPROM) bruker elektrisk spenning istedenfor ultrafiolett lys for å nullstille koblingspunkter.

## 10.7.2. ROM-celle

Mask ROM (MROM) er kretser som blir programmert under produksjonen. Produsenten maskerer et område på kretsen under prosessen, derav Mask-navnet. En ROM-celle programmeres permanent under tilvirkningsprosessen og kan ikke endres senere. Bruken begrenses normalt til standard eller brukerspesifikke funksjoner i en eller annen form.

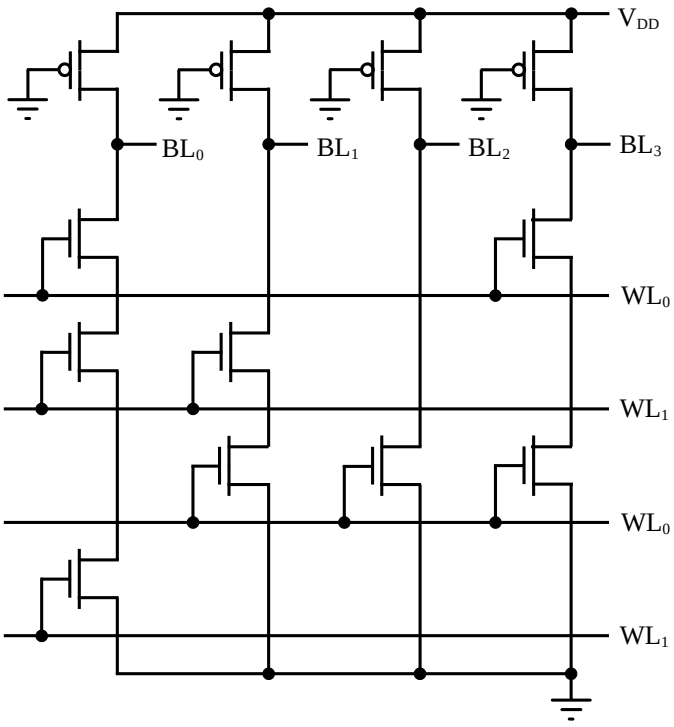
Et eksempel på en minnematrise er vist i figur 10.13. Dette er minne-celler med NOR-arkitektur siden minne-transistorene er koplet i parallell. De fire øverste transistorene utgjør aktiv last for de underliggende transistorene. Ved programmeringen fjernes forbindelsen til bitlinjen BL dersom det skal lagres en logisk 0 i cellen. Lagring av 1 realiseres ved at drain på transistoren er tilkoplet bitlinjen.




*Figur 10.13. Matrise med ROM-celler (NOR-arkitektur).*

Når en slik transistor velges med ordlinjen WL, slås den på og drain går lav. Bitlinjen vil følgelig inneholde den inverterte verdien av programmeringen. I eksemplet i figuren ses at adresse 0 ($WL_0$) lagrer data 1001 mens adresse 1 ($WL_1$) lagrer data 1100.

Et eksempel på en minnematrise bestående av minneceller med NAND-arkitektur er vist i figur 10.14. Dette er NAND-arkitektur siden minne-transistorene er koplet i serie. Ved programmeringen kortsluttes transistoren når det skal lagres en logisk 1 i cellen. Lagring av 0 realiseres ved at transistoren utgjør en del av bitlinjen.




*Figur 10.14. Matrise med ROM-celler (NAND-arkitektur).*

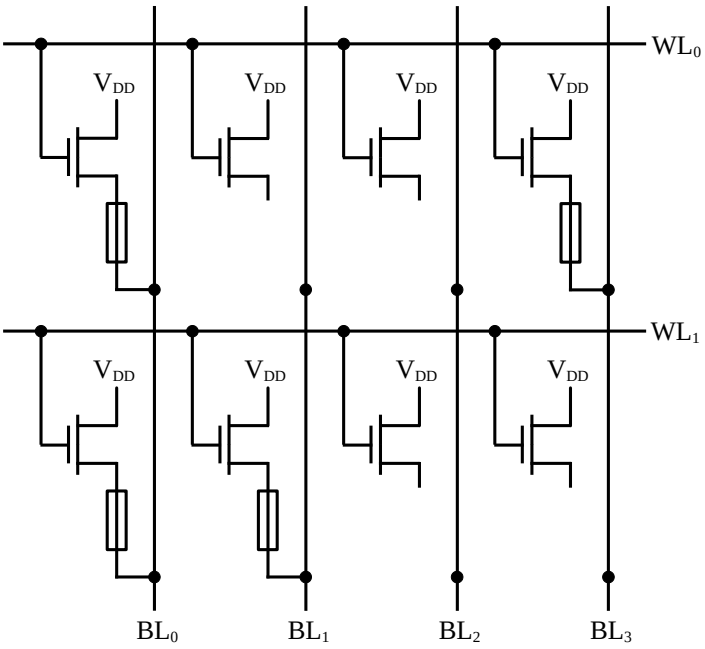
Vanligvis ligger ordlinjene høye slik at transistorene er på. Når en transistor skal velges med ordlinjen WL, legges gate lav og transistoren slås av. Bitlinjen går da høy og vil følgelig inneholde verdien av programmeringen. Dersom det er programmert en logisk 0 i adressen, ses at bitlinjen vil forbli høy siden de andre transistorene er på. I eksemplet i figuren ses at adresse 0 ($WL_0$) lagrer data 1001 mens adresse 1 ($WL_1$) lagrer data 1100.

## 10.7.3. PROM

En PROM kan fabrikkeres slik at brukeren kan programmere minnet ved å la en forbindelse være intakt eller bli brutt. Dette oppnås ved å bruke en form for sikring som kan realiseres på forskjellige måter. Denne programmeringsmetoden er irreversibel; er minnet programmert, kan det ikke forandres. Ved programmering sørges for tilstrekkelig strøm slik at sikringen «smelter» og bryter forbindelsen. Det finnes forøvrig også en metode der en forbindelse opprettes eller umuliggjøres ved bruk av dioder.

I figur 10.15 er vist et eksempel på en matrise med PROM-celler. Det vanlige vil være at en intakt sikring representerer en logisk 1 mens en brutt forbindelse representerer en logisk 0.

Lagring av 1 er her realisert ved at source på transistoren er tilkoplet bitlinjen BL. Når en slik transistor velges med ordlinjen WL, slås den på og source går høy. Bitlinjen vil følgelig inneholde verdien av programmeringen. I eksemplet i figuren ses at adresse 0 ($WL_0$) lagrer data 1001 mens adresse 1 ($WL_1$) lagrer data 1100.




*Figur 10.15. Matrise med PROM-celler.*

## 10.7.4. EPROM

EPROM er en type minnekretser som består av en matrise med transistorer som blir individuelt programmert. En EPROM kan reprogrammeres et visst antall ganger. Det benyttes en metode der tidligere programmerte data slettes før minnet programmeres på nytt, derav navnet Erasable PROM.

Det er to hovedtyper av EPROM, UV EPROM og EEPROM. Førstnevnte bruker ultrafiolett (UV) lys til sletting. Her er kretsene lett kjennbare ved at de har et gjennomsiktig kvartsglass rett over silisiumkretsen. EEPROM kan både slettes og programmeres med elektriske pulser. Mens sletting med ultrafiolett lys kan ta forholdsvis lang tid, kan programmering av EEPROM gjøres ganske raskt. Denne har derfor overtatt for UV EPROM.

Det finnes to hovedtyper av EEPROM: bruk av den eldre MNOS (Metall-Nitrid-Oksyd-Silisium)-transistoren og den nyere MOS-transistoren med «flytende» gate. Sistnevnte brukes med stort hell i FLASH-minner.

Figur 10.16 viser prinsippet for MOS-transistoren med «flytende» gate. Den består av en kontroll-gate og en «flytende» gate. Sistnevnte er vist med prikker. Disse representerer elektronladninger. En logisk 0 er representert med stort antall elektronladninger, mens en logisk 1 er representert ved ingen eller få elektronladninger. Antallet elektronladninger vil bestemme om transistoren er på eller av.


Lagring av 1 Lagring av 0

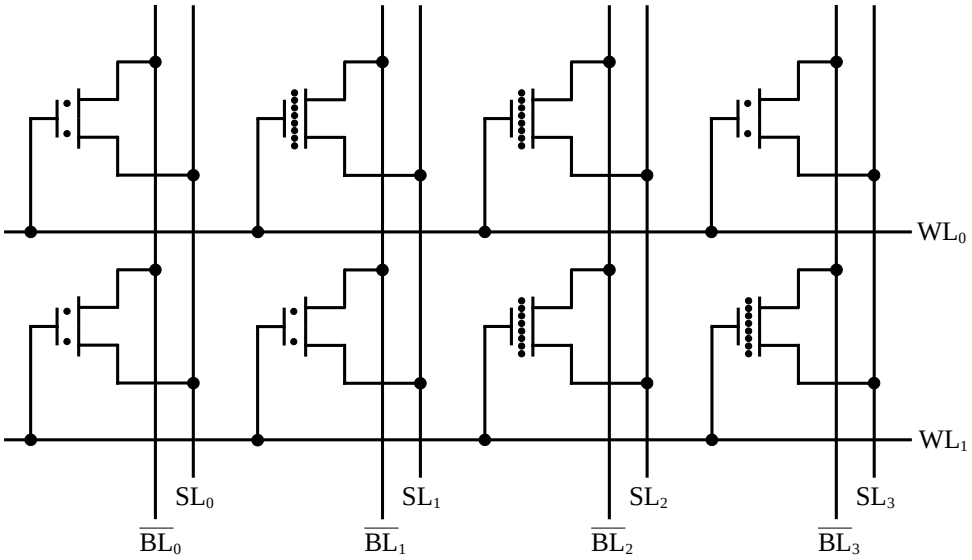
*Figur 10.16. EEPROM-celle med «flytende» gate.*

Sletteoperasjonen går ut på tømme «flytende» gate for elektronladninger. Dette gjøres ved å legge kontroll-gate på 0 V og source på en positiv spenning. Dersom en ønsker at transistoren skal lagre 0, tilføres elektroner til den «flytende» gate-en. Dette gjøres ved å legge kontroll-gate på en positiv spenning og «injisere» elektroner på source. Dersom en ønsker å lagre 1, lar en cellen forbli i slettet tilstand.

Når data skal leses, velges transistoren ved å legge kontroll-gate høy. Dersom «flytende» gate er tømt for elektroner (vi lagrer en logisk 1), slås transistoren på. Dersom «flytende» gate inneholder en stor elektronladning (vi lagrer en logisk 0), vil transistoren forbli av selv om kontroll-gate legges høy.

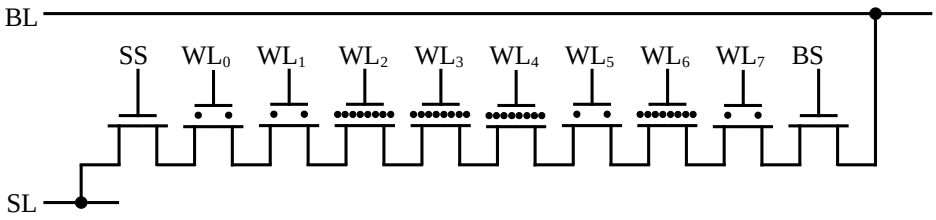
Flash-minner er kompakte minner med stor lagringskapasitet. Lagringskapasiteten er etterhvert blitt så stor at de kan erstatte konvensjonelle mindre hard-disker. Det er to typer arkitektur for flashminner: NOR og NAND. Mens en med NOR-typen kan skrive til individuelle byte, må en med NAND-typen aksessere informasjonen sekvensielt i blokker. I tillegg til å være billigere, er NAND-flash mer kompakt og har større lagringskapasitet enn NOR-typen.

I figur 10.17 er vist et eksempel på en enkel flash-minne-matrise med NOR-arkitektur. Som nevnt må source være tilgjengelig. Denne bussen er benevnt $SL_0$-$SL_3$ mens databussen er benevnt $\overline{BL_0}$-$\overline{BL_3}$. Når en transistor velges med ordlinjen WL, slås den på og drain går lav. Bitlinjen vil følgelig inneholde den inverterte verdien av programmeringen. I eksemplet i figuren ses at adresse 0 ($WL_0$) lagrer data 1001 mens adresse 1 ($WL_1$) lagrer data 1100.




*Figur 10.17. Flash minne-matrise (NOR-arkitektur).*

I figur 10.18 er vist et eksempel på flash celle med NAND-arkitektur. Her er transistorene med «flytende» gate koplet i serie, dette gir opphav til betegnelsen NAND-arkitektur. Vi skal ikke gå inn på virkemåten her, bare kort nevne at når data skal leses, velges adresse som en av ord-linjene WL. De andre ordlinjene legges på en spenning som gjør at de tilhørende transistorene slås på.



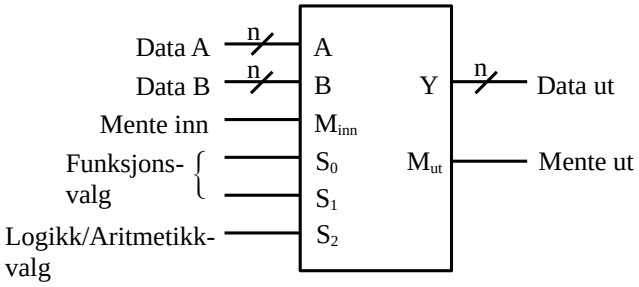

*Figur 10.18. Flash celle NAND-arkitektur*

Den valgte transistoren tilføres en gate-spenning akkurat over terskelspenningen. Dermed vil bitlinjen gå lav dersom den valgte transistoren er programmert til logisk 1, ellers vil bitlinjen forbli høy. Det brukes to transistorer til bitvalget, BS og SS. Sourcelinjen SL vil normalt koples til jord. På tross av økt antall transistorer i forhold til NOR-arkitekturen, vil reduksjonen i antall jordlinjer og bitlinjer muliggjøre et tettere utlegg og større lagringskapasitet for NAND-arkitekturen.

# 10.8. ALU

En Aritmetisk Logisk Enhet, ALU ('Arithmetic Logic Unit'), utgjør en integrert del av prosessorer. Den utgjør en viktig del i mikrokontrollere, se kapittel 11. En ALU utfører en rekke aritmetiske operasjoner som for eksempel addisjon, subtraksjon, multiplikasjon, divisjon, inkrementering, dekrementering og sammenligning. Den utfører også en rekke logiske operasjoner som for eksempel OG, ELLER, Eksklusiv ELLER og komplementering. I tillegg kommer for eksempel venstre- og høyre-bitskift med eller uten rotering.

Et forslag til et blokkskjema for en ALU er vist i figur 10.19. Som innganger er det to n bits dataord A og B. Datautgangen Y er også på n bit. Det er her tenkt at enheten er delt i en aritmetisk enhet og en logisk enhet. For hver av enhetene foretas et funksjonsvalg med $S_1$ og $S_0$. Valget mellom den logiske delen og den aritmetiske delen foretas av $S_2$. For den aritmetiske delen er det en mente inngang $M_{inn}$ og en mente utgang $M_{ut}$.

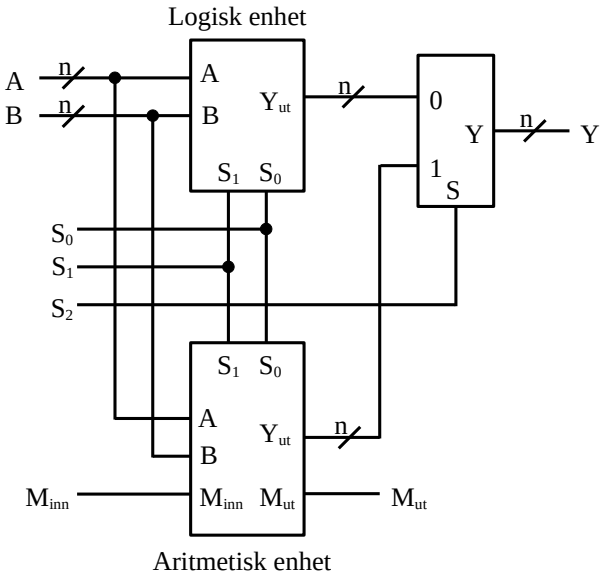



*Figur 10.19. Aritmetisk Logisk Enhet.*

Et blokkskjema for en ALU som er delt i en aritmetisk enhet og en logisk enhet, kan være som vist i figur 10.20. Den logiske enheten har da datainngangene A og B, hver på n bit, samt valginngangene $S_1$ og $S_0$. Utgangen er benevnt $Y_{ut}$, som også er på n bit.

Den aritmetiske enheten har også datainngangene A og B (på n bit) med de samme valginngangene $S_1$ og $S_0$. I tillegg has en menteinngang $M_{inn}$. Datautgangen er her $Y_{ut}$, som er på n bit. I tillegg has en menteutgang $M_{ut}$. Bruk av mente gir en forholdsvis enkel utvidelse av aritmetiske funksjoner. For eksempel kan inkrementering av data gjøres ved å sette menten inn, $M_{inn} = 1$.

I det følgende vil vi holde oss til kun ett bit (n = 1), men beskrivelsen kan forholdsvis enkelt utvides til flere bit.



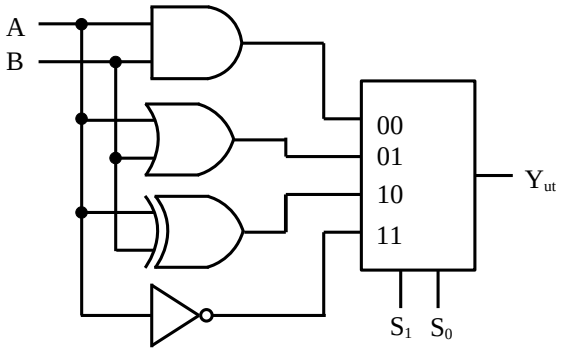

*Figur 10.20. Blokkskjema for n bit ALU.*

En Logisk Enhet for ett bit kan for eksempel ha funksjonene vist i tabell 10.3. Av sannhetstabellen fremgår at de logiske funksjonene er OG, ELLER, Eksklusiv ELLER og invertering.

| $S_1$ | $S_0$ | $Y_{ut}$ | Funksjon |
|---|---|---|---|
| 0 | 0 | $A \cdot B$ | OG |
| 0 | 1 | $A+B$ | ELLER |
| 1 | 0 | $A \oplus B$ | Eksklusiv ELLER |
| 1 | 1 | $\overline{A}$ | Invertering |

*Tabell 10.3. Sannhetstabell for Logisk enhet.*

En realisering av en Logisk Enhet med multiplekser kan være som vist i figur 10.21. Det ses at funksjonene i sannhetstabellen velges ved hjelp av 4-1 multiplekseren.



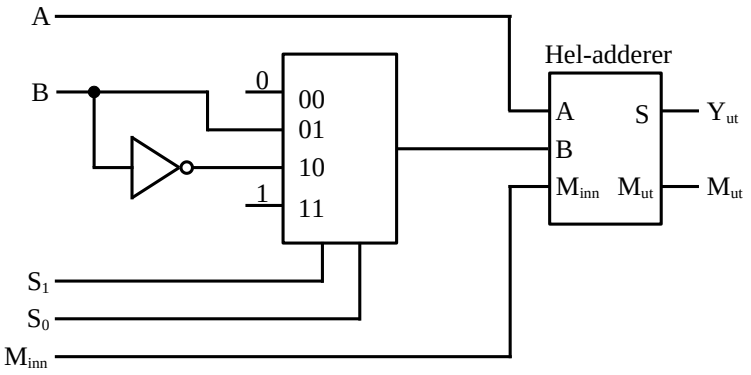

*Figur 10.21. Blokkskjema for Logisk Enhet.*

En Aritmetisk Enhet for ett bit kan for eksempel ha funksjonene vist i tabell 10.4. Av sannhetstabellen fremgår at det blant annet fås de aritmetiske funksjonene addisjon, subtraksjon, inkrementering og dekrementering.

| $M_{inn}$ | $S_1$ | $S_0$ | $Y_{ut}$ | Funksjon |
|---|---|---|---|---|
| 0 | 0 | 0 | A | |
| 0 | 0 | 1 | A+B | Addisjon |
| 0 | 1 | 0 | $A+\overline{B}$ | Addisjon med enerkomplement |
| 0 | 1 | 1 | A−1 | Dekrementering |
| 1 | 0 | 0 | A+1 | Inkrementering |
| 1 | 0 | 1 | A+B+1 | Addisjon med mente |
| 1 | 1 | 0 | $A+\overline{B}+1$ | Subtraksjon |
| 1 | 1 | 1 | A | |

*Tabell 10.4. Sannhetstabell for Aritmetisk enhet.*

En realisering av en Aritmetisk Enhet med multiplekser kan være som vist i figur 10.22. Det ses at 0, B, $\overline{B}$ og 1 velges ved hjelp av 4-1 multiplekseren. Ellers er det benyttet en hel-adderer for å realisere de aritmetiske funksjonene.




*Figur 10.22. Blokkskjema for Aritmetisk Enhet.*

For å realisere en n bit ALU kan det benyttes n Logiske enheter og n Aritmetiske enheter i henhold til figur 10.20. Bemerk at styresignalene $S_0$, $S_1$ og $S_2$ hver er på et bit. Mente inn, $M_{inn}$, tilføres da minst signifikante bit mens mente ut, $M_{ut}$, tas fra mest signifikante bit. Det vil da også være naturlig å bytte ut hel-addereren med en Carry Look Ahead adderer (se kapittel 5.16) for å minske transportforsinkelsen ved store ordlengder.

# Kapittel 11
# Mikrokontrollere

## 11.1. Innledning

En mikrokontroller kan enkelt sies å være en datamaskin på en IC-brikke. Den kan programmeres til å gjøre spesielle oppgaver. Den har sørget for at vi kan omgi oss med stadig flere og mer avanserte hjelpemidler, instrumenter og leketøy.

Hjernen i mikrokontrolleren kalles CPU, Central Processing Unit, og er en mikroprosessor. Mikroprosessoren er også en integrert krets som inneholder datamaskinens funksjoner. Den mangler hukommelse og inn- og ut-porter. Mikroprosessorer stammer fra 1971, da den første mikroprosessoren fra Intel så dagens lys. I dag kjenner vi mikroprosessoren blant annet som CPU-enheten i personlige datamaskiner.

Dersom vi kobler en mikroprosessor sammen med et minne og inn- og ut-porter via en systembuss, får vi en mikrokontroller. Som betegnelse for mikrokontrollere er det også vanlig å benytte betegnelsen MCU, som står for Mikrocontroller Unit.

De første mikrokontrollerne kom på markedet ikke lenge etter mikroprosessorene. De første mikrokontrollerne var imidlertid langt fra å være så avanserte som dagens. De var heller ikke så enkle å programmere som en kan si at dagens mikrokontrollere er.

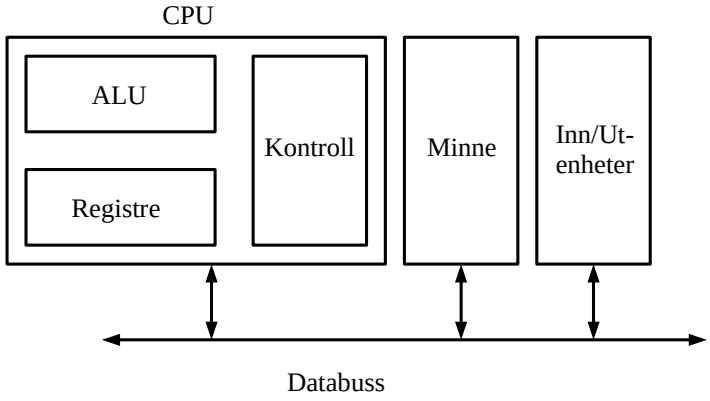
Mikrokontrolleren har som nevnt et stort anvendelsesområde. Den programmeres for bestemte anvendelser og monteres inn i utstyret den skal brukes til. Vi får da en slags «innebygd styring» (embedded control) av utstyret. Et innebygd system (embedded system) framstår ikke som en datamaskin for brukeren. I hverdagslivet omgis vi av «ting» som er styrt av mikrokontrollere uten å tenke på at de styres av en liten datamaskin.

## 11.2. Arkitektur

En datamaskin som sies å bestå av de tre hovedelementene CPU, minne og inn/ut-enheter, se figur 11.1, kalles von Neumann-maskiner. Det benyttes da en databuss som forbinder alle enhetene i maskinen. CPU kan utveksle data med de andre enhetene som etter tur kan benytte databussen.

Data som sendes ut på databussen, kan ses av de andre enhetene som er tilkoblet bussen. Det benyttes adressering for å sikre at data bare sendes til den ønskede enheten. Bussene fører derfor både adresse- og datasignaler.

CPU utfører alle beregninger til mikrokontrolleren. CPU bestemmer også hvordan og i hvilken rekkefølge programmer skal utføres. Den tar alle nødvendige avgjørelser for at mikrokontrolleren skal virke. Som vist i figur 11.1, kan CPU sies å bestå av enhetene ALU (aritmetisk logisk enhet), Kontroll og Registre. Det kan derfor sies at CPU utfører aritmetiske og logiske operasjoner.

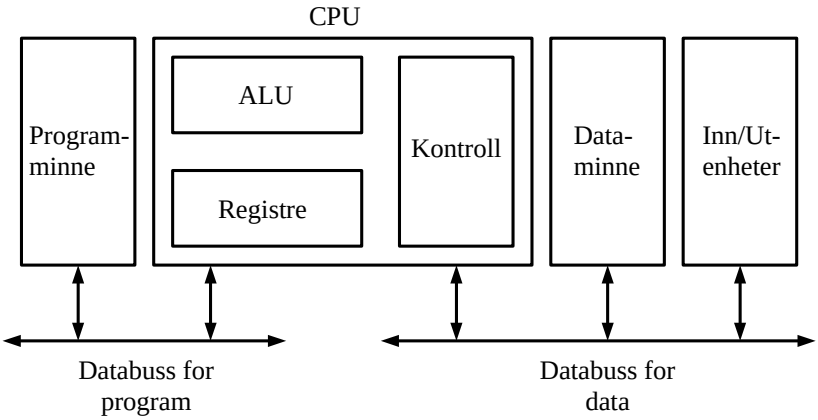



*Figur 11.1. Oppbygging av mikrokontroller.*

Alle programmer som skal utføres og alle data som behandles, må lagres i minnet. Også minnet er derfor tilkoplet databussen.

Inn/ut-enhetene har som oppgave å mate mikrokontrolleren med informasjon og å presentere informasjon fra mikrokontrolleren. Mikrokontrollerens informasjon kan utveksles direkte med brukere eller den kan utveksles med for eksempel måle- eller kontroll-utstyr. Informasjon kan selvfølgelig også utveksles med andre mikrokontrollere.

Istedenfor å ha et felles buss-system for program og data, kan det benyttes et system med to separate busser og to separate minner som vist i figur 11.2. Denne organiseringen er kjent som Harvard arkitektur. Denne arkitekturen gjør at CPU kan lese program og data samtidig, noe som øker hurtigheten.




*Figur 11.2. Mikrokontroller med Harvard-arkitektur.*

## 11.3. AVR-arkitektur

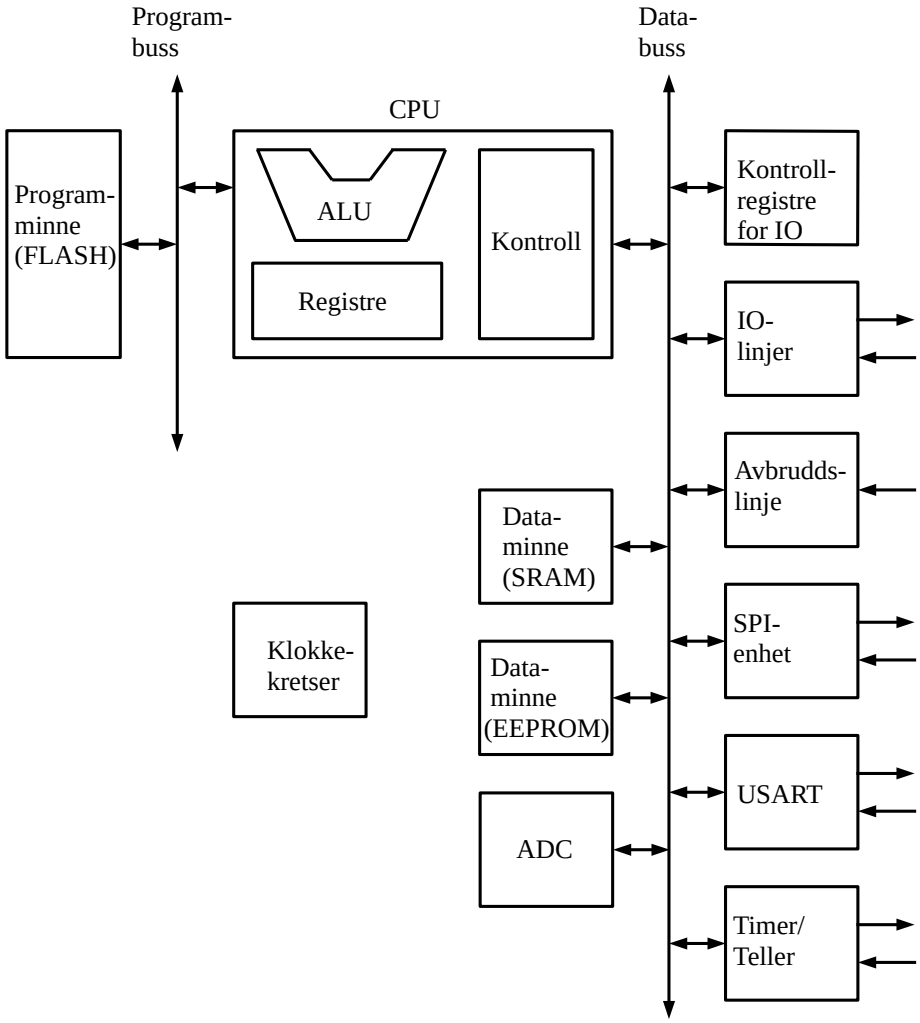
Den originale AVR MCU ble utviklet av to NTNU-studenter i tilknytning til ASIC-firmaet Nordic VLSI (nå Nordic Semiconductor). Teknologien ble solgt til Atmel i 1996. AVR 8-bit mikrokontroller-arkitekturen ble introdusert i 1997. AVR er i dag kjent som en familie mikrokontrollere. Disse er modifiserte Harvard-arkitektur 8-bit RISC mikrokontrollere på en brikke, der RISC står for Reduced Instruction Set Computer.

I praksis betyr dette at de aller fleste maskininstruksjoner utføres i løpet av en enkelt klokkeperiode. RISC-arkitekturen medfører også at alle maskininstruksjoner holdes like lange av effektivitetsgrunner. Ordlengden for instruksjoner er 16 bit. De andre ordlengdene er for det meste 8 bit. Motstykket til RISC er CISC-maskiner, der CISC står for Complex Instruction Set Computer. CISC er vanlig for mikroprosessorer som brukes i personlige datamaskiner.

AVR har en modifisert Harvard arkitektur der program og data lagres i separate fysiske minnesystemer. AVR finnes i forskjellige familier. AVR har som andre mikrokontrollere mange anvendelsesområder og er også kjent som mikrokontrollerne som brukes til Arduino.

AVR var også blant de første til å bruke FLASH programminne. Flash, EEPROM og SRAM er integrert på samme brikke slik at behovet for eksternt minne i de fleste tilfeller er unødvendig. Noen typer kan legge til ekstra dataminne ved hjelp av ekstern parallell buss. De fleste har serielle grensesnitt som kan brukes til å forbinde større serielle EEPROM- eller FLASH-brikker.

AVR-kontrollerne benytter FLASH (16 bit) programminne og SRAM (8 bit) dataminne. For permanent lager av parametre og andre størrelser som må huskes, benyttes EEPROM (8 bit), se figur 11.3. Antall ord som inngår i de forskjellige hukommelsestypene, vil variere mellom de ulike typer AVR mikrokontrollere.



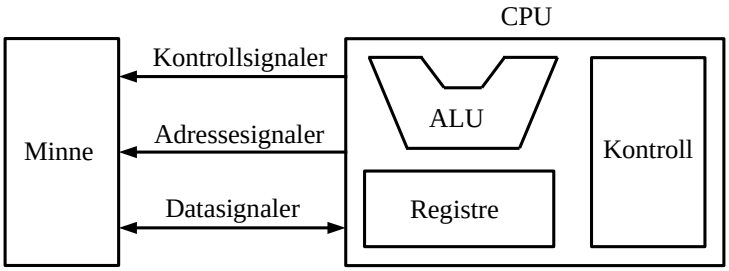

*Figur 11.3. AVR mikrokontrollerarkitektur.*

I tillegg til CPU og minne har AVR mikrokontrollerne flere integrerte inn/ut-enheter. Som det ses av figur 11.3, har mikrokontrolleren interne buss-systemer. For at det skal være mulig å utvide kapasiteten til mikrokontrolleren, er det også mulig å forlenge de interne bussene til eksterne. På denne måten blir det for eksempel mulig å koble til ekstra SRAM og ekstra inn/ut-funksjoner, dette er imidlertid ikke vist i figuren.

Antall inn/ut-funksjoner vil også variere. Slike funksjoner være parallellporter for digital inn/ut, analog/digital-omformere, klokker og tidsmålere (timere) samt SPI-enheter. SPI står forøvrig for Serial Peripheral Interface, som er en buss for synkron seriekommunikasjon.

Internt i mikrokontrolleren er det 2 hovedbuss-systemer, et for transport av programinstruksjoner og et for transport av data. Disse deles igjen opp i 3 deler, se figur 11.4:

1. Kontrollbuss-instruksjon: Buss som fører kontrollsignaler for det som går på instruksjonsbussen.
2. Kontrollbuss-data: Buss som fører kontrollsignaler for lesing og skriving av data over databussen.
3. Adressebuss: Buss som fører adresse til data i dataminnet, til registre i CPU eller i inn/ut-enhetene.
4. Programadressebuss: Buss som fører adresser til instruksjoner som leses fra programminnet.
5. Instruksjonsbuss: Buss som fører instruksjoner som leses fra programbussen for instruksjonsregisteret.
6. Databuss: Buss som fører data mellom registre og dataminnet og mellom registre og inn/ut (IO)-enheter.




*Figur 11.4. AVR buss-system.*

# 11.4. CPU

CPU kan sies å deles opp i de tre enhetene ALU, generelle arbeidsregistre og kontrollenhet, se figur 11.4.

## 11.4.1. ALU

Aritmetisk, Logisk Enhet, ALU, er tidligere beskrevet i kapittel 10.8. For AVR mikrokontrollere henter ALU operander fra arbeidsregistrene og leverer resultatet til databussen. Statusinformasjon fra ALU-operasjonene lagres i statusregisteret.

## 11.4.2. Generelle registre

AVR mikrokontrollere har 32 stk. 8 bits generelle registre for mellomlagring av data som behandles av mikrokontrollerprogrammet. Disse arbeidsregistrene går under navnet GPR, General Purpose Register og er adresserbare fra 0 til 31 desimalt.

Verdier kan skrives inn og leses ut av registrene via databussen. Noen av arbeidsregistrene kan inneholde adresser (indirekte adressering) for å peke på data i dataminnet eller maskininstruksjoner i programminnet.

### 11.4.3. Kontrollenhet

Kontrollenheten henter inn og tolker instruksjoner fra hukommelsen. Dermed sørges for at ALU får stadig nye oppgaver. Den styrer også innskrivning og utlesning av data for hukommelsen. Kontrollenheten henter instruksjonene fra hukommelsen fortløpende. Imidlertid kan mikrokontrolleren ved såkalte hoppinstruksjoner avvike fra denne sekvensielle gangen.

### 11.4.4. Funksjonsenheter i kontrollenheten

En oversikt over funksjonsenheter som er typisk for AVR mikrokontrollere, er vist i figur 11.5.

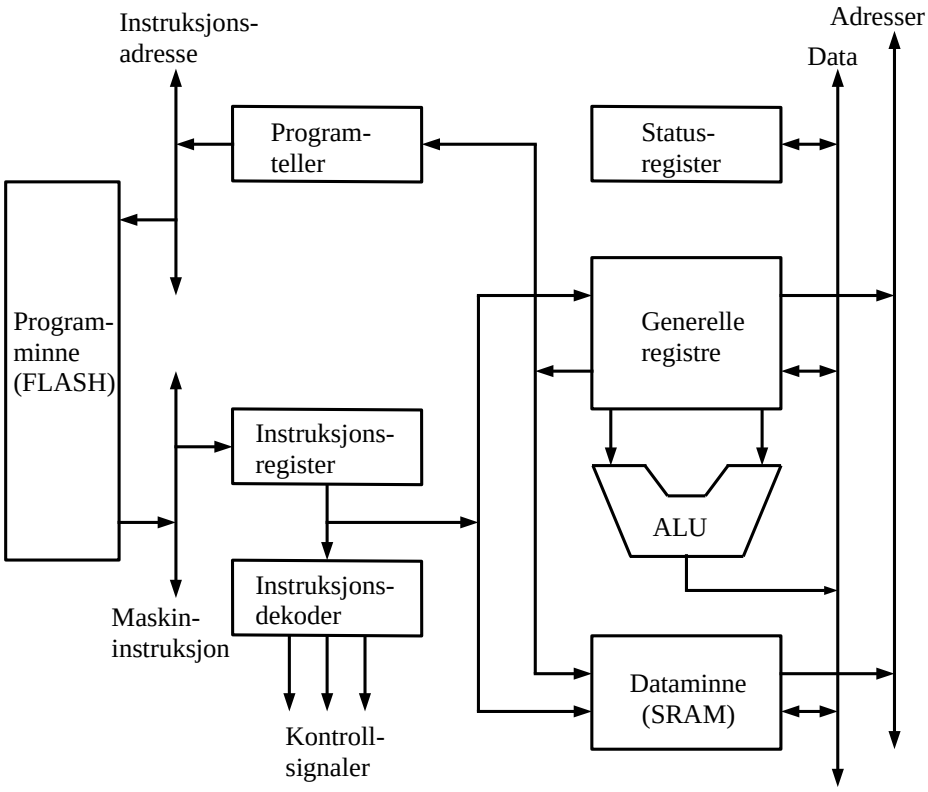
Følgende funksjonsenheter finnes i kontrollenheten for en AVR mikrokontroller:

1. Statusregister: Dette er et register som inneholder statusinformasjon blant annet fra ALU. Hvert av de åtte bit kalles flagg og hvert flagg har spesiell betydning. Et eksempel på et slikt flagg er Z (Zero), som er null-flagget. Dette flagget settes for eksempel dersom resultatet av en addisjon er lik 0. Et eksempel på et annet flagg er C (Carry), som gir ut 1 dersom mente genereres ved addisjon. Det finnes også flagget V, som er et Overflow-flagg.

2. Programteller: Programteller-registeret holder orden på hvor i programminnet neste instruksjon befinner seg. Neste instruksjon hentes fra den adressen som angis av innholdet i programtelleren. Når en instruksjon er utført, inkrementeres programtelleren ved at adressen økes med 1 eller mer avhengig av hvor mange linjer instruksjonen er på. For andre typer instruksjoner (for eksempel hoppinstruksjoner) settes programtellerens nye adresse av selve instruksjonen.

3. Instruksjonsregister: I instruksjonsregisteret lagres instruksjonene som tolkes og utføres av CPU. Noen instruksjoner kan inneholde direkte referanser til dataminnet eller til registre.

4. Instruksjonsdekoder: Instruksjonsdekoderen tolker hver enkelt instruksjon og lager kontrollsignaler som styrer utførelsen av instruksjonene i ALU. Den sørger også for at data flyttes fra register til register eller mellom registre og dataminnet.

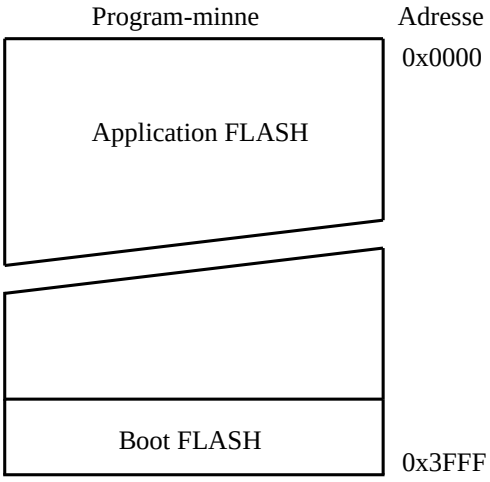



*Figur 11.5. AVR arkitektur med funksjonsenheter.*

## 11.5. Internhukommelse

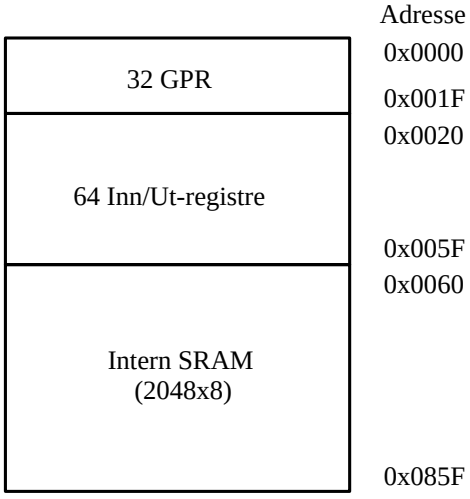
Hos AVR-kontrollerne har generelt dataminnet 8-bits ordlengde, mens programminnet har 16-bits ordlengde. Data på 16 bit kan settes sammen ved hjelp av to ord. Instruksjoner er på 16 eller 32 bit. For instruksjoner er ordlengden 2 byte, 16 bit. Adresser oppgis vanligvis på heksadesimal form. Dette kan angis ved å sette en h bak adressen. Et annet alternativ for angivelse av heksadesimale adresser er å benytte prefikset 0x, se figur 11.6. Eksemplet som vises, er for mikrokontrolleren ATmega32.

Fra figur 11.6 ser vi at i vårt eksempel adresseres programminnet 0x0000 til 0x3FFF. Dette gir 32K byte reprogrammerbar FLASH-minne for lagring av program. Størrelsen på programminnet vil være forskjellig fra mikrokontroller til mikrokontroller.




*Figur 11.6. AVR Program-minne.*

For ATmega32 kan eksempelvis dataminnet adresseres fra 0x00 til 0x085F, som tilsvarer et adresseringsrom på 2144 adresser, se figur 11.7. Av disse benyttes 32 adresser til arbeidsregistrene (GPR), 64 til inn/ut-registrene og 2048 til SRAM dataminne, sistnevnte kan følgelig lagre 2 KB (2048 byte). Figur 11.8 viser strukturen til de 32 generelle arbeidsregistrene. Disse registrene nummereres som regel desimalt, fra R0 til R31.




*Figur 11.7. AVR Dataminne.*

| Bit 7 0 | Adresse | |
|---|---|---|
| R0 | 0x00 | |
| R1 | 0x01 | |
| R2 | 0x02 | |
| --- | | |
| R13 | 0x0D | |
| R14 | 0x0E | |
| R15 | 0x0F | |
| R16 | 0x10 | |
| R17 | 0x11 | |
| --- | | |
| R26 | 0x1A | X-register Low Byte |
| R27 | 0x1B | X-register High Byte |
| R28 | 0x1C | Y-register Low Byte |
| R29 | 0x1D | Y-register High Byte |
| R30 | 0x1E | Z-register Low Byte |
| R31 | 0x1F | Z-register High Byte |

*Figur 11.8. AVR General Purpose Register*

I tillegg til å være generelle lagringsplasser for data, har registrene R26 - R31 også en tilleggsfunksjon. Disse registrene kan 2 og 2 sammen benyttes til å lagre adresser. De dobbelte registrene kalles henholdsvis X, Y og Z. Disse er satt sammen som vist i figur 11.9.

| Bit 15 8 | 7 0 | |
|---|---|---|
| 7 R27 0 | 7 R26 0 | X-register |
| 7 R29 0 | 7 R28 0 | Y-register |
| 7 R31 0 | 7 R30 0 | Z-register |

*Figur 11.9. Adresseregistre R26-R31.*

X, Y og Z kalles adresseregistre eller peker-registre. De benyttes i forbindelse med indirekte adressering av datarommet, for eksempel for lesing eller skriving av data til SRAM (intern eller ekstern). Det er spesielle instruksjoner i mikrokontrollerens instruksjons-sett som benytter adresseregistrene. Disse instruksjonene gir også muligheter for å inkrementere eller dekrementere innholdet i registrene automatisk etter lesing eller skriving. For å kunne forandre innholdet i et adresseregister, må nye verdier i mest signifikante byte og minst signifikante byte lastes hver for seg.

De fleste instruksjoner bruker to byte i programminnet mens noen bruker det dobbelte. Sistnevnte gjelder først og fremst instruksjoner som angir direkte adresse i dataområdet.

ATmega32 inneholder 1 KB (1024 byte) med EEPROM dataminne. Dette minnet er organisert i et eget adresserom adskilt fra programminnet og dataminnet for øvrig. Det er mulig å lese og skrive dataord (på 8 bit) for dette dataområdet.

De 64 8-bits inn/ut-registrene benyttes for å styre og kontrollere de fleste inn/ut-funksjonsenhetene i mikrokontrolleren. For å utføre en inn/ut-operasjon, skrives en kommando til det registeret som styrer denne. Status kan leses fra statusbit i registrene. Det er følgelig enkelt å overføre (lese eller skrive) data mellom R0-R31 og inn/ut-registre.

## 11.6. Instruksjonsutførelse

Med referanse til figur 11.5 skal vi i det følgende vise en forenklet punktvis beskrivelse av hva som skjer når mikrokontrolleren utfører en instruksjon. Som eksempel kan instruksjonen ha som oppgave å hente en databyte fra generelt register R16 og kopiere innholdet til register R20.

- Adressen fra programtelleren legges på instruksjonsadressebussen.
- Instruksjonen hentes ut fra programminnet og kopieres til instruksjonsregisteret.
- Programtelleren inkrementeres med 1.
- Instruksjonen sendes videre til instruksjonsdekoderen der instruksjonen dekodes.
- Adressen til de generelle registrene R16 og R20 hentes fra instruksjonsregisteret.
- Instruksjonsdekoderen genererer kontrollsignaler for kopiering.
- Data i register R16 legges ut på databuss.
- Innholdet på databussen kopieres til register R20.

## 11.7. Programmering

### 11.7.1. Innledning

Mikrokontrolleren må programmeres for å utføre de oppgavene som ønskes. Vi skriver da et program som lagres i programminnet. Programmet består av instruksjoner. Jo flere instruksjoner, jo mer komplisert oppgave vil vi at mikrokontrolleren skal utføre. I mikrokontrollerens programminne lagres disse instruksjonene i binærkode, kalt maskinkode. Det ligger i sakens natur at maskinkoden vil variere fra mikrokontroller til mikrokontroller, selv imellom forskjellige AVR mikrokontrollere.

Å skrive et program i maskinkode vil være en nesten uoverkommelig oppgave med stor risiko for å gjøre feil. Derfor kan vi benytte høynivå programmeringsspråk, for eksempel C, C++ eller Java. Et program skrevet i høynivåspråk må kompileres, det vil si oversettes til maskinkode før det overføres til mikrokontrolleren. For de fleste oppgaver vil et program skrevet i for eksempel C, være den raskeste og greieste måten å programmere mikrokontrolleren på.

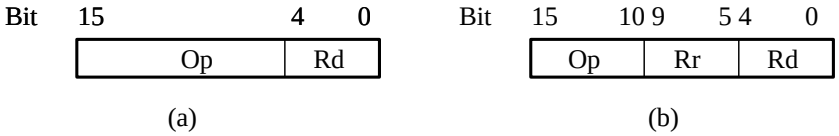
For mer tidskritiske anvendelser kan det hende at vi må skrive hele eller deler av programmet i såkalt assemblerspråk. Dette er egentlig en form for kodet maskinkode.

Det sier seg selv at jo mer kompliserte oppgaver mikrokontrolleren skal utføre, jo mer egner et høynivåspråk seg til programmeringen. I det følgende vil vi imidlertid vise noen eksempler i assemblerspråk siden disse kan illustrere virkemåten for mikrokontrolleren.

## 11.7.2. Maskininstruksjoner

Summen av alle maskininstruksjoner som kan forstås av en mikrokontroller, kalles instruksjons-settet. Det kan bestå av alt fra ca. 100 instruksjoner til opp mot 500. ATmega32 har 131 maskininstruksjoner, de fleste utføres i løpet av en klokkeperiode. De fleste maskininstruksjonene er som nevnt på to byte.

Antall bit som går med til operasjonskode, register-referanser og dataminne-referanser vises ofte som i figur 11.10 der det er vist to eksempler. I figur 11.10a er vist Direct Single Register Addressing og i figur 11.10b Direct Register Addressing, Two Registers. Her er Op operasjonskoden i instruksjonsordet mens Rd er Destination (and source) register in the Register File og Rr er Source register in the Register File.




*Figur 11.10. eksempel på instruksjonsoppbygging.*

Source-registeret er det registeret som dataverdien hentes fra. Med andre ord inneholder Rd instruksjonens (ene) operand. Destination-registeret angir det registeret et resultat skal skrives til. Det er ikke mulig å angi mer enn to registre for hver instruksjon. I de tilfellene der det trenges to registre til å holde operander, må den ene operanden ligge i samme register som resultatet skal legges i. Rd angir i de tilfellene både source og destination.

Operasjonskoden angir hva instruksjonen skal utføre, som for eksempel addisjon, subtraksjon, kopiering av data og så videre.

Mikrokontrollerens instruksjons-sett kan inndeles i fire grupper:

1. Aritmetiske og logiske instruksjoner
2. Forgreningsinstruksjoner (for eksempel relative hopp-instruksjoner)
3. Dataoverføringsfunksjoner (for eksempel overføring av data fra port til generelt register)
4. Bit- og Bit-test-instruksjoner (for eksempel setting et bit i et inn/ut-register)

Instruksjonsoppbyggingen kan være forskjellig fra figur 11.10, men det henvises til den særskilte mikrokontrollerens dokumentasjonen for detaljer.

## 11.7.3. Assemblerinstruksjoner

Det å forholde seg til bitkombinasjonene som danner maskininstruksjonene, er svært arbeidskrevende og lite effektivt. Derfor programmerer ingen direkte i maskinkode. Maskininstruksjoner skrevet på symbolsk form kalles for assemblerinstruksjoner. Hver operasjonskode i maskin-instruksjons-settet har et symbolsk navn. Dette navnet er en forkortelse som beskriver hva instruksjonen går ut på. Disse forkortelsene kalles mnemonics (huskesymboler) og er laget for at det skal være ganske enkelt å huske hva instruksjonene gjør.

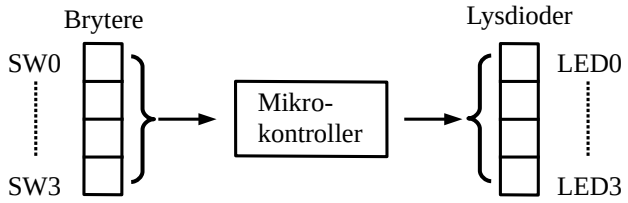
Eksempler på typiske navn er MOV og ADD som står for henholdsvis flytt- og addisjonsinstruksjon. Direkte adresser til dataminnet kan også gis symbolske adresser. Som nevnt blir ofte adresser angitt heksadesimalt. For å angi numeriske verdier, kan det brukes alt fra desimalt og heksadesimalt til binær representasjon. I det siste tilfellet kan prefikset 0b benyttes, eksempelvis vil 0b1001 svare til 9 desimalt.

Et eksempel på et lite program i assembler med noen få instruksjoner er:

```
.def A=R16
.def B=R17
.def SUM=R20
mov SUM,A       (0010111101000000)
add SUM,B       (0000111101000001)
loop: rjmp loop (1100111111111111)
```

I begynnelsen ses direktivene som definerer ***A***, ***B*** og ***SUM*** til å gjelde registrene R16, R17 og R20. Programmet kopierer data i R16 til R20. Instruksjonsutførelsen er som beskrevet i kapittel 11.6. Data i R17 adderes så til data som allerede er i R20. Med andre ord har vi utført operasjonen SUM = A + B. Instruksjonen ***rjmp*** står for ***relative jump***. Dette programmet vil med andre ord gå i en evig sløyfe. Kodet maskinkode for ATmega32 er vist i parentes.

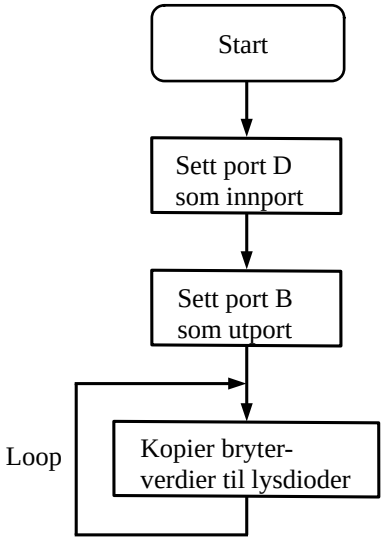
Som et annet eksempel på et relativt enkelt program, vil vi her vise hvordan vi kan programmere mikrokontrolleren slik at hver gang vi trykker en bryter, vil tilsvarende lysdiode tennes: Bryter SW0 tenner lysdiode LED0, bryter SW1 tenner lysdiode LED1 osv. Vi har da et mikrokontrollersystem for fire brytere og lysdioder som vist i figuren under.




*Figur 11.11. Mikrokontrollersystem med brytere og lysdioder.*

På mikrokontrolleren må vi definere inn- og ut-porter for henholdsvis bryterne og lysdiodene. Her kan vi bruke port D som inngangsport og port B som utgangsport.

For vår mikrokontroller defineres inn/ut-porter ved å sette verdier i portenes dataretnings-registre. Dataretningsregisterne er DDRB for PortD og DDRB for PortB, der 0 = innport og 1 = utport. Først kan vi lage oss et flytskjema for programmet, som vist nedenfor.




*Figur 11.12.Flytdiagram.*

Etter dette kan vi lage et assemblerprogram basert på flytskjemaet. Programmet som er vist på neste side, inneholder seks assemblerinstruksjoner som er vist i tabellen nedenfor. I tabellen er også vist betydningen av disse.

| **Instruksjon** | **Betydning** |
|---|---|
| rjmp start | Relative jump. Hopp til ’label’ kalt start |
| ldi temp, 0xff | Load immedeate. Overfører verdien 0xff til temp |
| out ddrb, temp | Overfører innholdet i temp til DDRB |
| in temp, pind | Overfører innholdet i port-inngangene for PortD til temp |
| out portb, temp | Overfører innholdet i temp til PortB |
| rjmp loop | Relative jump. Hopp til label kalt loop |

*Tabell 11.1.Assembler-instruksjoner med betydning.*

Programmet kan da være som vist nedenfor.

```
; Lysdioder og trykknapper
.nolist
.include "m32def.inc"
.list
;-----------------------------------------------------
.def temp=R16
;----------------------------HOPP-tabell----------------
.org 0
reset; rjmp start
.org 20
;-----------------------------------------------------
start:
;----------------------PORTB=utport-----------------
ldi temp,0xff
out ddrb,temp
;-----------------------------------------------------
; Evig sløyfe. PORTB=PIND
;-----------------------------------------------------
loop:
in temp,PIND
out PORT,temp
rjmp loop
;-----------------------------------------------------
```

*Figur 11.13. Assemblerprogram.*

Programmet har to etiketter (labels), avsluttet med kolon: loop og start. I tillegg til de 6 instruksjonene inneholder programmet kommentarlinjer og assembler-direktiver. Kommentarer defineres ved et semikolon ( ; ) som start-tegn. Kommentaren går ut linjen. Et assemblerdirektiv starter med et punktum og er en kommando, «beskjed», til assembleren. Det er 6 assemblerdirektiver i dette programmet, som vist nedenfor. Merk forøvrig: Assembler skiller ikke mellom små og store bokstaver; 0xff betyr $FF_{16}$ = $1111\ 1111_2$.

| **Assembler-direktiv** | **Betydning** |
|---|---|
| .nolist | Ingen utskrift av det etterfølgende |
| .include ”m32def.inc” | Inkluder filen m32def.inc i programmet |
| .list | Utskrift av det etterfølgende |
| .def temp=R16 | Register R16 defineres ved navnet TEMP |
| .org 0 | Etterfølgende instruksjon legges i adresse 0 |
| .org 20 | Etterfølgende instruksjon legges i adresse 20 |

*Tabell 11.2.Assembler-direktiver med betydning.*

# Kapittel 12
# VHDL

## 12.1. Innledning

VHDL er et akronym som står for VHSIC Hardware Description Language. VHSIC er et annet akronym som står for Very High Speed Integrated Circuits. VHDL er et maskinvarebeskrivende programmeringsspråk. VHDL er en standard under IEEE. Standarden har vært revidert flere ganger. VHDL'87 og VHDL'93 er standarder som ofte brukes.

Et maskinvarebeskrivende språk kan brukes på forskjellige måter. Det kan være en alternativ måte å representere et kretsskjema på eller det kan være som et høynivå programmeringsspråk som løser et gitt problem. Et maskinvarebeskrivende språk kan brukes for å høynivå-programmere kretser uten å måtte behøve å bry seg så mye om hvordan designet ser ut på portnivå og uten å måtte kople sammen porter. Med VHDL kan det abstraheres mye, men muligheten for å være relativt maskin-nær er også til stede.

VHDL kan brukes til dokumentasjon, verifisering og syntese av store digitale design. Dette er faktisk en meget viktig egenskap til VHDL siden samme VHDL-kode kan brukes for å oppnå alle disse tre målene. Dette forenkler design og minsker risikoen for feil på gangen mellom spesifikasjon og realisering.

I tillegg kan VHDL brukes til å beskrive maskinvare strukturelt, funksjonelt og ved hjelp av dataflyt. Vanligvis benyttes alle tre metodene, og et fullstendig digitalt design vil ha en blanding av alle disse, der hver del kan beskrives på forskjellige måter. Det betyr også at det alltid vil være flere metoder og løsninger i VHDL.

## 12.2. Strukturell beskrivelse

For å gjøre design mer forståelige, lettere å vedlikeholde og lettere å bruke (helt eller delvis) på nytt, deles det vanligvis opp i blokker. Disse blokkene er så bundet sammen til et komplett design. Et VHDL-design kan beskrives ved hjelp av en enkel blokk eller kan dekomponeres i flere små blokker.

Hver blokk i VHDL kalles en **entitet**. Entiteten beskriver grensesnittet til blokken. En separat del til blokken beskriver hvordan blokken opererer. Grensesnittbeskrivelsen er som en pinnebeskrivelse på en integrert krets der inn- og utganger spesifiseres. Beskrivelsen av hva blokken gjør, kan sammenlignes med et skjema for blokken.

En blokk kan være et design, og mange blokker som er bundet sammen, kan være et fullstendig design. Når vi skal beskrive en blokk i VHDL, er det som nevnt flere muligheter.

Et typisk VHDL-program har følgende struktur:

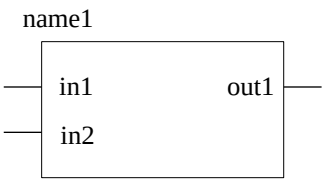
```
entity name1 is  -- Doble bindestrek innleder kommentarer
-- Beskrivelse av grensesnittet, inn- og utganger, følger så her
end name1; -- Blokken, entiteten, har navnet name1

architecture name2 of name1 is  -- Beskrivelse av hvordan blokken name1 opererer
-- Beskrivelse av eventuelle interne signaler følger her
begin
-- Beskrivelse av funksjon følger her
end name2; -- Kan også skrive bare end eller end entity
```

En komplett entitet kan for eksempel se slik ut:

```
entity name1 is
port (in1, in2: in bit;
      out1: out bit);
end name1;
```

Entiteten kan illustreres med et blokkskjemasymbol som vist nedenfor.




*Figur 12.1. Blokkskjema for name1.*

En tilhørende arkitektur kan for eksempel se slik ut:

```
architecture name2 of name1 is
begin
out1 <= not (in1 and in2); -- Funksjonsuttrykk
end name2; -- Kan også skrive bare end eller end architecture
```

I portbeskrivelsen ses at vi har to innganger og en utgang. Vi ser at signalene er av type ***bit***. Signaler eksisterer mellom blokker og er mer komplekse enn variable i tradisjonelle programmeringsspråk. Signaler er tidsavhengige og trenger følgelig en viss tid for å kunne endres.

Variable finnes også i VHDL, men kan bare finnes innen prosesser, prosedyrer eller funksjoner. Variable forandres umiddelbart som i andre programmeringsspråk. I arkitekturen ovenfor er for eksempel out1 et signal. Derfor benyttes <= istedenfor likhetstegn.

Alle signalene i eksemplet ovenfor er som nevnt av type ***bit***. Da er signalet begrenset til å bare kunne ha to verdier: 0 og 1. En mer kompleks datatype vil kunne ha verdiene 0, 1, Z (høy impedans) og til og med X (ukjent). For å kunne representere virkelige verdier brukes et ekstra bibliotek, ***std_logic*** er det mest brukte.

I eksemplet ovenfor vil VHDL-arkitekturen beskrive en NAND-port med to innganger. Dette ses av OG-funksjonen (***and***) mellom in1 og in2 i arkitekturbeskrivelsen samt en invertering (***not***).

Som nevnt ble entiteten beskrevet med to innganger og en utgang. Med andre ord kan vi bruke en entitet til å beskrive en blokk med et innhold som kan være den enkle NAND-porten, men også til å beskrive alt fra et komplisert digitalt design til enkle porter.

Ønsker vi at blokken skal inneholde en NOR-port med to innganger, kan vi følgelig beholde entitetsbeskrivelsen, men lage oss en ny arkitekturbeskrivelse, som for eksempel:

```
architecture name3 of name1 is
begin
out1 <= not (in1 or in2);
-- out1 <= in1 nor in2; -- Alternativ
end architecture;
```

Til slutt må nevnes at mens andre programmeringsspråk, som for eksempel Java og C/C++, er designet for sekvensiell eksekvering, er VHDL helt forskjellig. Ettersom VHDL beskriver hvordan en programmerbar krets skal oppføre seg, beskriver dette språket parallelle forløp.

## 12.3. Forbindelse mellom blokker

La oss si at vi ønsker å lage en RS datalås ved hjelp av to NOR-porter. Da kan entiteten for datalåsen se slik ut:

```
entity latch1 is
port (set, reset: in std_logic;
      q, nq: out std_logic); -- nq: invertert Q-utgang
end entity;
```

Legg merke til at vi har byttet ut type ***bit*** med type ***std_logic***. Det er ikke mulig å bruke forskjellige typer om hverandre, slik at for eksempel **in *bit*** og **in *std_logic*** brukt samtidig vil gi feil. Skal vi gå fra en type til en annen, må vi ta i bruk spesielle funksjoner.

Da kan vi la entiteten for NOR-porten være:

```
entity two_one is
port (in1, in2: in std_logic;
      out1: out std_logic);
end;
```

Og realiseringen av NOR-funksjonen kan vi skrive:

```
architecture nor_port of two_one is
begin
out1 <= in1 nor in2;
end;
```

Arkitektur-beskrivelsen av datalåsen kan da skrives:

```
architecture netlist of latch1 is
component two_one -- component erstatter entity for denne blokken  (NOR-porten)
port (in1, in2: in std_logic;
       out1: out std_logic);
end component;
signal x,y: std_logic; -- Interne signaler
begin
n1: two_one port map (reset, y, x); -- Samme rekkefølgen som i portbeskrivelsen
-- Vi trenger to NOR-porter:
n2: two_one port map (in1 => set, in2 => x, out1 => y); -- Alternativ mappingmetode
q <= x;   -- Interne signaler x og y føres til utgangene q og nq
nq <= y;
end;
```

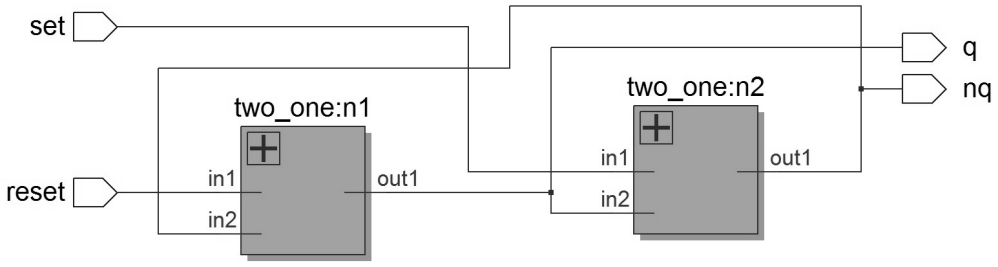
Dette ser jo ut til å være relativt komplisert bare for å lage en datalås, men her er eksemplet bare ment å vise prinsippet for å lage forbindelse mellom blokker. Bemerk at vi må bruke interne signaler x og y siden utgangssignalene ikke kan brukes internt i VHDL.

Legg merke til at det benyttes **component** istedenfor **entity** når vi skal bruke NOR-porten som en blokk to ganger her. Port-beskrivelsen for **component** er den samme som for **entity**. Mens **entity** beskriver grensesnittet til verden på utsiden, beskriver **component** bare grensesnittet til blokken som brukes som en del av entiteten. En liste av komponenter og deres forbindelser kalles vanligvis en nettliste.

Vi bruker komponenten kalt two_one to ganger ved hjelp av **port map**. Legg merke til at mappingen kan foretas på to forskjellige måter. Ved mappingen kalt n1 er det er viktig at rekkefølgen er det samme som i portbeskrivelsen, mens ved mappingen kalt n2 er det ikke så nøye med rekkefølgen siden vi bruker pilsymbol.

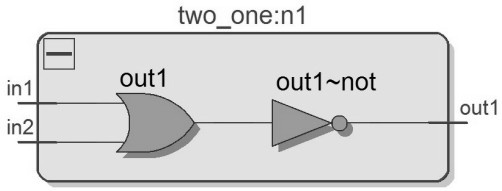
Det bør også bemerkes at slike komponenter som latch finnes i biblioteket til maskinvareleverandørenes programvare.

Dersom vi kompilerer VHDL-beskrivelsen ovenfor med en egnet programvare, kan vi få visualisert hva dette har ledet til. For betegnelsen dataflyt brukes ofte betegnelsen RTL (Register Transfer Language). I et RTL-skjema ser det ut som i figur 12.2 etter kompileringen.




*Figur 12.2. RTL-skjema for latch1.*

Innholdet av two_one vil da se ut som i figur 12.3. Vi ser at dette er NOR-porten. Vår kode beskriver følgelig en datalås.



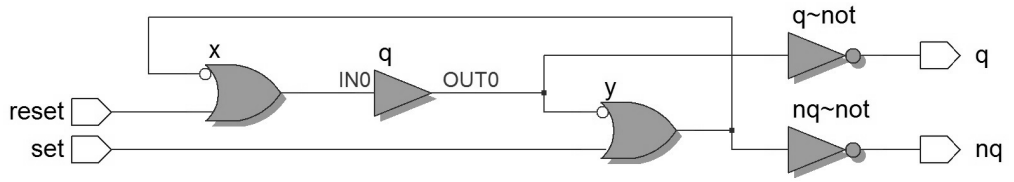

*Figur 12.3. RTL-skjema for two_one.*

## 12.4. Dataflytbeskrivelse

Vi kan tenke oss å lage en datalås med en dataflyt-beskrivelse. Med dataflytbeskrivelse mener vi hvordan primitive logiske komponenter (for eksempel NOR-porter) eller rene kombinatoriske blokker er koplet sammen. Med andre ord beskriver vi hvordan signaler flyter gjennom kretsen. Med samme entitet som ovenfor, kan vi tenke oss å ha en arkitektur-beskrivelse av en datalås som ser slik ut:

```
architecture dataflow of latch1 is
signal x,y: std_logic; -- Interne signaler
begin
x <= reset nor y; -- Den ene NOR-porten
y <= set nor x; -- Den andre NOR-porten
q <= x; -- Interne signaler x og y føres til utgangene q og nq
nq <= y;
end dataflow;
```

I et RTL-skjema ser det ut som i figur 12.4. Vi ser at det er benyttet ELLER-porter med inverteringer. Legg merke til at de to forskjellige måtene å beskrive samme funksjonen på, leder til ulik realisering.




*Figur 12.4. RTL-skjema for latch med dataflyt.*

Noen ganger ser en også betegnelsen dataflytbeskrivelse om kombinatoriske blokker som klokkes av en enkelt klokke, men her vil betegnelsen RTL kanskje passe bedre.

# 12.5. Definisjoner

I VHDL er det ikke forskjell på store og små bokstaver (og heller ikke om det benyttes fet skrifttype), **Begin**, **BEGIN**, **begin** og begin gir samme resultat. Det er en god regel å bruke kommentartegnet (--), både for egen del og for å lette forståelsen for andre.

I VHDL kan vi som nevnt ikke blande forskjellige datatyper, som eksempel ble det påpekt at en blanding av ***bit*** og ***standard_logic*** ikke godtas. I VHDL er det i noen tilfeller ikke enkelt å gå fra en datatype til en annen, slik at det er en god regel å holde seg til ***std_logic*** og ***std_logic_vector*** (se avsnitt 12.6.2).

Følgende regler for identifikatorer i VHDL gjelder:

- En identifikator består av bokstaver, sifre og/eller understrekingstegn
- Første tegn må være en bokstav
- Siste tegn kan ikke være understrekingstegn
- To påfølgende understrekingstegn er ikke tillatt
- Reserverte ord (for eksempel **architecture**, **begin**) må ikke brukes som identifikatorer

VHDL skiller mellom **signal** og **variable**. En variabel er bare definert innenfor en **process**, og behøver ikke egentlig å finnes i det ferdige designet. Det er noen ganger det kan være hensiktsmessig å bruke **variable**, men det er en god hovedregel å bruke **signal**.

# 12.6. Datatyper

## 12.6.1. Bit og std_logic

Det mest brukte biblioteket som beskriver datatyper er ***ieee.std_logic_1164***. Vi har tidligere sett datatypen ***bit***, som bare kan ta verdien 0 og 1.

Datatypen ***boolean*** er i prinsippet det samme, men med verdien FALSE og TRUE. Som nevnt er dette ofte begrenset, selv på bitnivå. Adresse- og data-busser lar seg ikke realisere med denne datatypen siden vi ikke kan bruke tre-nivå (høy impedans). Det får vi som nevnt med ***std_logic***, en datatype som finnes i biblioteket ***std_logic_1164***.

Det er en standard datatype som det anbefales å bruke istedenfor ***bit***. Et signal av typen ***std_logic*** kan ha følgende verdier:

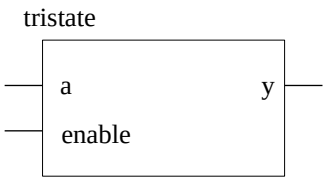
| | |
|---|---|
| ’U’: | Uninitialized |
| ’X’: | Forcing Unknown |
| ’0’: | Forcing 0 |
| ’1’: | Forcing 1 |
| ’Z’: | High impedance |
| ’W’: | Weak Unknown |
| ’L’: | Weak 0 |
| ’H’: | Weak 1 |
| ’-’: | Don’t care |

U og X vil vi (ofte) se ved simuleringer mens Z er for tre-nivå-utganger. De andre typene uten-om 0 og 1 vil sjelden være aktuelle.

Tre-nivå buffer er mye brukt både for data- og adresse-busser. Eksempel på en entitet for en slik kan være:

```
entity tristate is
      port (a, enable: in std_logic;
            y: out std_logic);
end tristate;
```

Entiteten kan illustreres med et blokkskjemasymbol som vist nedenfor.




*Figur 12.5. Blokkskjema for tristate.*

Arkitekturbeskrivelsen kan for eksempel være:

```
architecture when_else of tristate is
begin
      y <= a when enable = '1' else 'Z';
      -- y <= not(a) when enable = '1' else 'Z'; -- Alternativ med inverterende buffer
end when_else;
```

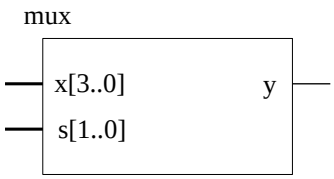
Det er benyttet **when else** der vi lar inngangs- og utgangsdata være av datatypen ***std_logic***. Når enable er lik 0, vil utgangen på bufferen være Z, altså høyimpedant.

## 12.6.2. Bit_vector og std_logic_vector

Ofte benyttes flere bit for å representere et binært tall i et design. I VHDL finnes som i andre programmeringsspråk datatypen ***vector*** (array). En forhåndsdefinert type er ***bit_vector*** som representerer en samling av bit. En 4-1 multiplekser kan ha følgende entitet:

```
entity mux is
  port (x: in bit_vector(3 downto 0); -- 4 innsignaler
        s: in bit_vector(1 downto 0); -- Velger innsignal
        y: out bit); -- Utsignal
end mux;
```

Her er vektoren x av lengde 4 nummerert fra x(3) til x(0), der x(3) er mest signifikante bit. Dette er de fire inngangene til multiplekseren som velges til utgangen y med vektoren s på to bit. Entiteten kan illustreres med et blokkskjemasymbol som vist nedenfor. Legg forøvrig merke til den litt spesielle buss-notasjonen som brukes for vektorangivelse.




*Figur 12.6. Blokkskjema for mux.*

På samme måte som ***bit*** kan ***bit_vector*** ikke operere på tre-nivå-signaler. Derfor benyttes normalt ***std_logic_vector*** istedenfor. Da kan en 4-1 multiplekseren ha følgende entitet:

```
entity mux is
port (x: in std_logic_vector(3 downto 0); -- 4 innsignaler
      s: in std_logic_vector(1 downto 0); -- Velger innsignal
      y: out std_logic); -- Utsignal
end mux;
```

En arkitekturbeskrivelse kan for eksempel se slik ut:

```
architecture with_sel of mux is
begin
    with s select
        y <= x(3) when "11", -- s = 11 velger x(3) til utgangen
             x(2) when "10", -- s = 10 velger x(2) til utgangen
             x(1) when "01", -- s = 01 velger x(1) til utgangen
             x(0) when "00", -- s = 00 velger x(0) til utgangen
             x(0) when others; -- Denne er obligatorisk
end with_sel;
```

Koden bør være selvforklarende. Det er benyttet **with select** for å rute den riktige inngangen til utgangen. Legg merke til at **when others** er obligatorisk selv om vi har dekket alle 4 mulighetene i vår arkitekturbeskrivelse. Vi kunne selvsagt også bare tatt de tre øverste mulighetene og så deretter skrevet:

```
x(0) when others;
```

Dersom s var ukjent, kunne vi for eksempel ha skrevet:

```
'1' when others; -- Alternativ
```

Da har vi en mulighet for eksempelvis å kunne finne feil. Vi kan også la vektoren gå fra minst signifikante bit til mest signifikante bit, for eksempel:

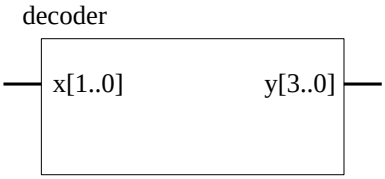
```
bit_vector(1 to 3) eller std_logic_vector(1 to 3)
```

Når vi utfører operasjoner på hele vektorer, må de ha samme lengde. Hvis ikke, vil kompilatoren protestere. Dersom vi skal operere på vektorer med forskjellig lengde, er det forholdsvis greit å øke lengden på den korteste vektoren (se kapittel 12.6.4).

Et alternativ til **with select** er **when else**. Entitetsbeskrivelsen for en 2-4 dekoder kan for eksempel være:

```
entity decoder is
port (x: in std_logic_vector(1 downto 0);
      y: out std_logic_vector(3 downto 0));
end decoder;
```

Entiteten kan illustreres med et blokkskjemasymbol som vist nedenfor.




*Figur 12.7. Blokkskjema for decoder.*

Arkitekturbeskrivelsen med **when else** kan da for eksempel være:

```
architecture when_else of decoder is
begin
      y <= “0001” when x = “00” else
            “0010” when x = “01” else
            “0100” when x = “10” else
            “1000” when x = “11”;
end when_else;
```

Legg merke til at denne koden er mer oversiktlig enn om vi skulle bruke boole’ske uttrykk for alle kombinasjoner. En 3-8 dekoder kan selvfølgelig lages på tilsvarende måte.

Både **with select** og **when else** er ofte brukt for implementering av kombinatoriske funksjoner. Dersom disse funksjonene er omfattende, vil de ofte gi så kompliserte boole’ske uttrykk at vi i realiteten ikke har noe valg utenom **with select**, **when else** eller bruk av prosess-instruksjonen (se kapittel 12.7.2).

### 12.6.3. Integer og natural

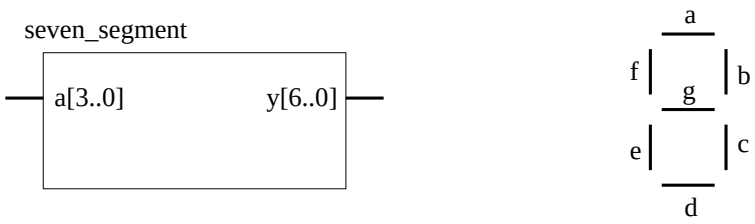
***Integer*** opererer stort sett som i andre programmeringsspråk. Med denne kan vi operere med positive og negative heltall. De mest åpenbare operasjonene på numeriske datatyper er typiske algebraiske som addisjon, subtraksjon, multiplikasjon og divisjon.

Et problem med ***integer*** er at den ikke kan indekseres på bitnivå, her vil ***std_logic_vector*** være et naturlig valg. For kun positive heltall kan vi bruke ***natural*** istedenfor ***integer***.

Dersom vi ønsker å lage en syvsegment-dekoder som viser sifrene 0-9 og er blank for desimal-verdiene 10-15, kan vi tenke oss følgende entitet:

```
entity seven_segment is
port (a : in integer range 0 to 15; -- Merk bruken av range
       y : out std_logic_vector(6 downto 0));
end seven_segment;
```

Her kunne vi brukt ***natural*** istedenfor ***integer***. Legg forøvrig merke til bruken av **range**. Entiteten kan illustreres med et blokkskjemasymbol som vist til venstre nedenfor.




*Figur 12.8. Blokkskjema og segmentbetegnelser for syvsegment-dekoder.*

Arkitekturbeskrivelsen kan for eksempel være:

```
architecture when_else of seven_segment is
begin
 y <= "1111110"   when a=0 else -- Segmentrekkefølge a,b,c,d,e,f,g
      "0110000"   when a=1 else -- Segment på for logisk 1, av for logisk 0
      "1101101"   when a=2 else
      "1111001"   when a=3 else
      "0110011"   when a=4 else
      "1011011"   when a=5 else
      "1011111"   when a=6 else
      "1110000"   when a=7 else
      "1111111"   when a=8 else
      "1111011"   when a=9 else
      "0000000"; -- Segmentene slokkes for a > 9
end when_else;
```

Koden bør være selvforklarende. Det er benyttet **when else** der vi lar inngangsdata være av datatypen ***integer***. Også her kunne vi benyttet **with select**. Istedenfor ***integer*** kunne vi selvfølgelig også benyttet ***std_logic_vector***.

### 12.6.4. Andre operatorer

Typiske algebraiske operatorer er addisjon (+), subtraksjon (-), multiplikasjon (*) og divisjon (/); der operatorene er vist i parentes.

Vanlige sammenlignings-operatorer som lik (=), ulik (/=) , mindre enn (< ), større enn (>), mindre enn eller lik (<=) og større enn eller lik (>=) er også tilgjengelig. Resultatet av alle disse operatorene er TRUE eller FALSE.

Argumentene for operatorene = og /= kan være alle typer. Argumentene for operatorene <, <=, > og >= kan være skalar type (***integer***, ***real*** og fysiske typer) eller vektortype. Dersom argumentene er vektortype, må argumentene være av samme lengde, og resultatet blir TRUE bare dersom hvert korresponderende element i matrisen er TRUE.

Operatoren & utfører sammenkjeding av vektorer. Ta for eksempel følgende deklareringer:

```
signal a: bit_vector (1 to 4);
signal b: bit_vector (1 to 8);
```

Følgende instruksjon vil legge a til høyre halvdel av b og la venstre del av b være konstant 0:

```
b <= "0000" & a;
```

Operatoren & føyer a til slutten av "0000" for å gi et resultat som er på 8 bit.

## 12.7. Forløpsbeskrivelse

### 12.7.1. Innledning

Metoden med forløpsbeskrivelse for å modellere maskinvare er forskjellig fra strukturell metode og dataflyt-metode. Dette skyldes at vi med forløpsbeskrivelsen ikke sier noe om hvordan designet implementeres. Vi opererer med en blokk med definerte inn- og utganger. Metoden modellerer hva som skjer på disse inn- og utgangene mens innholdet i boksen betraktes som irrelevant.

Metoden kan brukes i VHDL for å designe komplekse komponenter eller funksjoner som ellers ville bli for omfattende til å kunne designes med de to andre metodene. Dersom vi for eksempel skulle designe styringen av en automat, ville denne typen modellering være et naturlig valg. Ellers kan denne metoden være bedre og mer effektiv for noen design. Metoden vil kanskje innebære en strukturering av designet. Metoden egner seg godt til å evaluere en algoritme med tilhørende simulering, der passende inndata kan brukes for å verifisere designet.

Metoden egner seg godt for hierarkisk design idet vi kan definere blokker som kan forbindes. Hver blokk kan modelleres med denne metoden, og så lenge grensesnittene er definerte, kan også hver blokk designes av forskjellige personer uten problemer.

### 12.7.2. Prosess-instruksjonen

Forløpsbeskrivelsen krever en prosess-instruksjon. Prosess-instruksjonen kan legges i en arkitekturbeskrivelse på samme måte som signaldefinisjoner. Innholdet i prosess-instruksjonen vil ofte inneholde sekvensinstruksjoner som vi møter i andre programmeringsspråk. Disse instruksjonene brukes til å beregne utgangsdata fra tilførte inngangsdata.

Sekvensielle instruksjoner kan være kraftfulle, men har ofte ingen direkte sammenheng med maskinvare-implementeringen. Prosess-instruksjonene kan også inneholde signaltilordninger for å spesifisere prosessens utgangsdata. En gruppe av prosesser vil utføres samtidig siden prosess-instruksjonen er sekvensiell. Vi kan også gruppere en blokk med samtidige instruksjoner sammen og spesifisere når de skal utføres, dette kan være effektivt.

Nedenfor er et eksempel som egentlig ikke krever prosess-instruksjonen, men som kan brukes som eksempel for å illustrere metoden. Arkitekturbeskrivelsen kan være:

```
architecture behavioural of half_adder is -- Inngangene er a og b
begin
    compute_ha: process(a,b) -- Prosessen kan om ønskelig gis et navn (her: compute_ha)
    begin
        sum <= a xor b; -- Eksklusiv ELLER gir sum ut
        carry <= a and b; -- OG gir mente ut
    end process;
end behavioural;
```

Denne prosessen inneholder to signal-instruksjoner (***xor*** og ***and***). I motsetning til slike instruksjoner som ikke er innenfor en prosess, vil disse instruksjonene ikke gi en endret utgangsverdi medmindre a og/eller b endrer seg. Vi sier at a og b utgjør sensitivitetslisten. Det er således viktig å føre opp signaler vi ønsker skal påvirke prosessresultatet i sensitivitetslisten.

Instruksjonene innen prosess-instruksjonen utføres i rekkefølge fra første til siste. Når siste instruksjon er utført, avsluttes prosessen. Når det skjer en endring på signalene i sensitivitetslisten, gjenopptas prosessen, og instruksjonene innen prosess-instruksjonen utføres fra første til siste på nytt med forskjellig inndata.

### 12.7.3. Variable

Det er to forskjellige objekter som brukes for data. Signaler brukes for det meste i strukturell beskrivelse og dataflytbeskrivelse. Signaler brukes også til å forbinde komponenter, blokker og prosesser. Det andre objektet som brukes, kalles variabel. Oppførselen for variable er den samme som kjennes fra andre programmeringsspråk, men er i VHDL forskjellig fra signaler. Et eksempel på tilordning av variable kan være:

```
a:=b;
```

Her tilordnes verdien av b til a. Verdien av b kopieres umiddelbart direkte til a. Variable kan bare brukes i prosesser. Følgelig kan tilordninger bare finnes innen prosesser. Tilordninger gjøres når prosessen utføres. En arkitekturbeskrivelse kan for eksempel være:

```
architecture behavioural of counter is
begin
    count_up: process (clock) -- Prosess utføres når clock endres (x må være definert)
    variable cnt : integer := 0; -- Deklarering med initialisering
        begin
        if rising_edge(clock) then   -- Positiv klokketransisjon
        cnt:=cnt+1; -- Inkrementering av variabel cnt
        end if;
        count <= conv_std_logic_vector(cnt,4); -- Signalet count må være definert i entity
    end process;
end;
```

Legg merke til at deklarering av variabel cnt gjøres før **begin** i **process**. Her er også tatt med initialisering, men dette er opsjonelt (selv om det kan være en god regel). Den variable cnt er her deklarert til å være av datatype ***integer***.

Prosessen inneholder en instruksjon: tilordningen cnt:=cnt+1, der verdien av cnt økes med verdien 1 og umiddelbart lagres med den nye verdien som variabelen cnt. Siden cnt er initialisert til 0, vil første verdien av cnt være lik 1 når prosessen starter. For at inkrementering skal skje, må klokkesignalet gå fra 0 til 1.

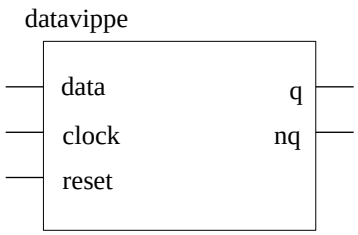
For at resultatet skal være tilgjengelig som et fysisk signal, må variabel cnt konverteres til signalet count. Her har vi antatt at dette signalet er på 4 bit, som må være med i konverteringsinstruksjonen. Gjøres ikke dette, vil ikke prosessen ha noen utgangsdata, med andre ord vil vi da ikke få noen signaler ut av blokken som er beskrevet av denne prosessen.

## 12.7.4. Sekvensielle instruksjoner

Det er flere typer instruksjoner som bare kan brukes innen prosess-instruksjonen. Dette er sekvensielle instruksjoner som utføres fra start til slutt i rekkefølgen de står i prosess-instruksjonen. Et eksempel på en slikt sekvensiell instruksjon er **if**. La oss ta en datavippe som eksempel. En entitetsbeskrivelse kan være:

```
entity datavippe is
port(data, clock, reset : in std_logic;
      q, nq : out std_logic); -- nq: invertert Q-utgang
end;
```

Entiteten kan illustreres med et blokkskjemasymbol som vist nedenfor.




*Figur 12.9. Blokkskjema for datavippe.*

En arkitekturbeskrivelse kan for eksempel være:

```
architecture use_if of datavippe is
begin
p0: process (reset, clock) is -- Navn p0 på process er opsjonelt
     begin
       if (reset = '0') then -- Asynkron reset er aktiv lav
         q <= '0';
         nq <= '1';
      elsif rising_edge(clock) then -- Stigende flanke på klokke
         q <= data; -- Data på inngang føres til Q-utgang
        nq <= not(data);
    end if;
  end process p0;
end architecture;
```

Foruten **if then** og **elsif** kan også benyttes **if then else**. Det ses at resettingen er asynkron siden denne er lagt før stigende flanke til clock. Legg forøvrig merke til at data føres til utgangen på stigende flanke til clock med ***rising_edge***(clock). Ønsket vi i stedet at dette skulle skje på fallende flanke, kunne vi skrive: ***falling_edge***(clock). Det finnes også en attributt som heter **event**, slik at vi i stedet kunne ha skrevet:

```
elsif (clock='1' and clock'event) then -- Stigende flanke på klokke
```

Bemerk forøvrig at prosessen ikke utføres uten at klokke eller resetting endrer verdi som det ses av **process** (reset, clock).

## 12.7.5. Registre

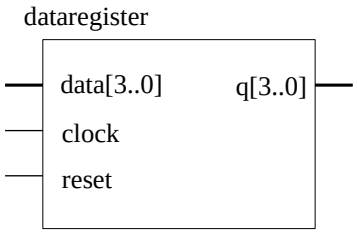
### *Dataregister*

Som et annet eksempel på **process**- og **if**-instruksjonene kan vi ta et parallell inn/parallell ut dataregister. En entitetsbeskrivelse kan være som følgende:

```
entity dataregister is
generic (n : natural := 4); -- 4 bit brukt her
port (data: in std_logic_vector(n-1 downto 0);
       clock, reset : in std_logic;
       q : out std_logic_vector(n-1 downto 0));
end entity;
```

Legg merke til bruken av **generic**. Fordelen med denne er at vi kan definere lengden av registeret bare ved å endre verdien på n. Ellers har vi også tatt med en resett-inngang foruten klokkeinngangen. Inngangen og utgangen er vektorer med lengder bestemt av n. Resettingen kan være asynkron eller synkron.

Entiteten kan illustreres med et blokkskjemasymbol som vist nedenfor.




*Figur 12.10. Blokkskjema for dataregister (n = 4).*

I arkitekturbeskrivelsen nedenfor har vi brukt asynkron resetting.

```
architecture process_descript of dataregister is
begin
p0: process (clock, reset) is
      begin
        if (reset = '1') then -- Aktiv høy asynkron resetting
         q <= (others => '0');
         elsif rising_edge(clock) then
         q <= data;
         end if;
      end process p0;
end architecture;
```

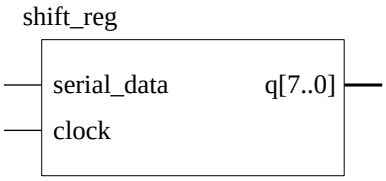
Legg igjen merke til instruksjonen ***q <= (others => '0')***. Dette betyr at alle Q-utgangene resettes til 0 (når reset er lik 1 her). Dette er uavhengig av klokkesignalet. For hver positivgående klokkeflanke vil data på inngangen føres til Q-utgangene. Legg forøvrig merke til at n = 1 i **generic** gjør at vi får en enkel D-vippe.

## *Skiftregister*

En annen type register er serie inn/parallell ut skiftregister. Entitetsbeskrivelsen blir nesten som for foregående register:

```
entity shift_reg is
generic (n : natural := 8); -- 8 bit brukt her
port (serial_data: in std_logic;
       clock : in std_logic;
       q : out std_logic_vector(n-1 downto 0));
end;
```

Også her har vi benyttet **generic**. Entiteten kan illustreres med et blokkskjemasymbol som vist nedenfor.




*Figur 12.11. Blokkskjema for shift_reg (n = 8).*

En arkitekturbeskrivelse kan være som følgende:

```
architecture process_descript of shift_reg is
signal q_temp: std_logic_vector(n-1 downto 0);
begin
      process(clock)
      begin
         if rising_edge(clock) then
         q_temp((n-1) downto 1) <= q_temp((n-2) downto 0);
         q_temp(0) <= serial_data;
         end if;
      end process;
  q <= q_temp;
end;
```

Koden bør være selvforklarende. De 7 minst signifikante bit (for n = 8), $q_0$-$q_6$, føres til de øverste utgangene $q_1$-$q_7$ mens seriell data (serial_data) føres til utgangen av minst signifikante bit $q_0$. Det benyttes temporær q i prosessen siden utgangene ikke kan benyttes internt.

## 12.7.6. Tellere

### *Binærteller*

Som et annet eksempel på **process**- og **if**-instruksjonene kan vi ta en binærteller som eksempel. En entitetsbeskrivelse kan være:

```
entity counter is
generic(n : natural := 8); -- 8 bit valgt
port(clk : in std_logic;
        count : out std_logic_vector(n-1 downto 0) );
end entity;
```

Legg igjen merke til bruken av **generic**. Fordelen med denne er at vi kan definere hvor mange bit telleren skal være på, bare ved å endre verdien på n.

Entiteten kan illustreres med et blokkskjemasymbol som vist nedenfor.


*Figur 12.12. Blokkskjema for counter (n = 8).*

En arkitekturbeskrivelse kan for eksempel være:

```
architecture behave_if of counter is
begin
    count_up: process (clk) is -- Navn count_up på process er opsjonelt
    variable cnt : integer :=0 ; -- Variabel initialiseres til 0
        begin
          if rising_edge(clk) then -- Stigende flanke på clk
          cnt:=cnt+1; -- Inkrementering
          end if;
        count <= conv_std_logic_vector(cnt, n); -- Er plassert på innsiden av process
    end process;
end architecture;
```

Legg merke til at inkrementeringen av cnt skjer på stigende flanke til clk med ***rising_edge***(clk). Ønsket vi i stedet at inkrementeringen skulle skje på fallende flanke, kunne vi som nevnt skrive: ***falling_edge***(clk). I alle tilfeller vil **if**-instruksjonen bare utføres når clk endres. Vi vil med andre ord inkrementere cnt hver gang clk går fra 0 til 1 (eventuelt fra 1 til 0 for ***falling_edge***(clk)). Legg også merke til at den variable ***cnt*** med størrelse ***n*** må konverteres til ***std_logic_vector*** inne i prosess-beskrivelsen.

En arkitekturbeskrivelse der det benyttes temporært ***signal*** istedenfor ***variable***, kan for eksempel se slik ut:

```
architecture behave_if of counter is
signal cnt: std_logic_vector (n-1 downto 0) :=(others => '0') ; -- Signal initialiseres til 0
begin
    count_up: process (clk) is -- Navn count_up på process er opsjonelt
        begin
          if rising_edge(clk) then -- Stigende flanke på clk
          cnt <= cnt+1; -- Inkrementering
          end if;
    end process;
    count <= cnt; -- Er plassert på utsiden av process
end architecture;
```

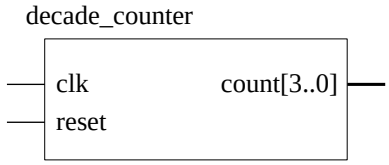
Legg igjen merke til bruken av instruksjonen q <= (**others** => ’0’) som betyr at alle utgangene resettes til 0. I de fleste tilfeller vil imidlertid initialisering før tellingen starter være unødvendig både for den variable ***cnt*** og signalet ***cnt***.

## *Dekadeteller*

En dekadeteller med resetting-inngang kan ha følgende entitetsbeskrivelse:

```
entity decade_counter is
port(clk, reset : in std_logic;
        count : out std_logic_vector(3 downto 0));
end entity;
```

Entiteten kan illustreres med blokkskjemasymbolet vist nedenfor.




*Figur 12.13. Blokkskjema for decade_counter.*

En arkitekturbeskrivelse der det benyttes temporært ***signal***, kan for eksempel se slik ut:

```
architecture behave of decade_counter is
  signal cnt: std_logic_vector (3 downto 0);
  begin
      process (clk, reset) is
        begin
          if rising_edge(clk) then -- Stigende flanke på clk
          if (reset = '0' or cnt="1001") then -- Synkron resetting og nullstilling hvis teller er lik 9
            cnt <= (others => '0');
            else
            cnt <= cnt+1; -- Inkrementering
         end if;
      end if;
    end process;
 count <= cnt; -- Er plassert på utsiden av process
end architecture;
```

Det ses at resettingen er synkron siden den er lagt inn etter ***rising_edge***(clk). Tilsvarende vil telleren nullstilles når telleren har nådd verdien 9 desimalt. Telleforløpet er følgelig 0, 1, 2...,9, 0, 1 etc. Legg igjen merke til bruken av instruksjonen ***q <= (others => '0')*** som betyr at alle utgangene resettes til 0.

### *Timer*

Det er ikke noe i veien for å ha flere prosesser i en arkitekturbeskrivelse. Her kan vi illustrere det med en 4 bit binærteller der vi ønsker å gi ut logisk 1 i de to tilfellene der telleren har verdiene 3 og 9 desimalt. Entiteten og arkitekturen kan da for eksempel være:

```
entity timer is
port(clk, reset : in std_logic;
        t1, t2: out std_logic;
        count : out std_logic_vector(3 downto 0));
end entity;

architecture behave of timer is
signal cnt: std_logic_vector(3 downto 0);
begin
p1: process (clk, reset) is
     begin
       if reset = '0' then
          cnt <= (others => '0');
          elsif rising_edge(clk) then
          cnt <= cnt + 1;
       end if;
     end process p1;
p2: process (cnt) is
     begin
       t1 <= '0';
       t2 <= '0';
       case cnt is
         when "0011" => t1 <= '1';
         when "1001" => t2 <= '1';
         when others =>
         t1 <= '0';
         t2 <= '0';
       end case;
   end process p2;
      count <= cnt;
end architecture;
```

Her lar vi ***t1*** være 1 når telleren har verdien 3 mens ***t2*** er 1 når telleren har verdien 9. Instruksjonen **case** har mye til felles med tilsvarende instruksjon i enkelte andre programmeringsspråk.

Entiteten kan illustreres med et blokkskjemasymbol som vist nedenfor.


*Figur 12.14. Blokkskjema for timer.*

## 12.7.7. Loop-instruksjonen

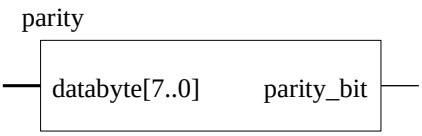
Det finnes flere varianter av **loop**-instruksjonen. En variant er **for**-instruksjonen. I det første eksemplet brukes denne for å finne lik paritet til en byte (8 bit vektor).

### *Paritetsgenerator*

Når antallet enere i de 8 bit er et likt tall, settes paritetsbit lik 0, ellers lik 1. Entitets-beskrivelsen kan for eksempel være som vist nedenfor. Her er paritetsbit kalt **parity_bit**.

```
entity parity is
    port (databyte: in std_logic_vector(7 downto 0);
          parity_bit: out std_logic);
end entity;
```

Entiteten kan illustreres med et blokkskjemasymbol som vist nedenfor.




*Figur 12.15. Blokkskjema for parity.*

Arkitektur-beskrivelsen kan for eksempel være som vist nedenfor.

```
architecture behave of parity is
  begin
     process(databyte)
     variable par: std_logic;
       begin
          par:='0';  -- Må initialiseres til 0
          for j in 7 downto 0 loop
             par := par xor databyte(j);
          end loop;
       parity_bit <= par;
     end process;
end architecture;
```

Instruksjonen ***for j in 7 downto 0*** kalles parameterspesifikasjonen. Her spesifiseres hvor mange ganger den påfølgende instruksjonen utføres og definerer j som så løper fra 7 ned til 0. Siden det brukes eksklusiv ELLER, vil den variable ***par*** være 0 dersom to og to bit begge er 0 eller 1, og 1 ellers. Sløyfen gjennomløpes det antallet ganger som er spesifisert av den variable j. Til slutt kopieres verdien av den variable ***par*** til paritetsbit ***parity_bit***. Bemerk at den variable ***par*** må settes lik 0 før loop-instruksjonen.

## *Binær til BCD-omformer*

I det andre eksemplet med **loop**- og **for**-instruksjonen er det tatt med en binær til BCD-omformer ved bruk av den såkalte «Skift og addér 3»-algoritmen. Eksemplet her tar utgangspunkt i et ord på 8 bit, se tabell 12.1. I tabellen er antatt «worst Case» ved å la det binære tallet være bare enere.

I tabellen er det kolonner for både det opprinnelige binære tallet og BCD-koden, sistnevnte delt inn i En, Ti og Hundre. Sistnevnte er på bare to bit, mens de to andre er på 4 bit hver. Algoritmen starter med at BCD-koden initialiseres til bare nullere. Deretter foretas tilsammen 8 venstreskift (lik antall bit i den binære verdien). Først foretas tre venstreskift. Før neste venstreskift testes kolonnen i BCD-koden om størrelsen desimalt er større enn 4.

Anta at En-kolonnen har verdien 0100. Etter et venstreskift vil kolonnen ha verdien 1001, som er en gyldig BCD-kode. Er verdien i kolonnen imidlertid lik 0101, vil den etter et venstreskift ha verdien 1011 (desimalt 11), som riktignok er 2 mer enn 1001, men som ikke er gyldig BCD-kode. Dersom vi legger 0011 (desimalt 3) til 0101, fås 1000 som etter et venstreskift gir 0001 i Ti-kolonnen og 0001 i En-kolonnen. Dette er det riktige resultatet i BCD-kode.

Algoritmen vil følgelig teste om verdien i BCD-kode-kolonnene er større enn 4. Hvis så, legges til desimalt 3. Fra tabellen ses at En-kolonnen har verdien 0111 etter tre venstreskift. Ved å addere 3, fås 1010, som etter et venstreskift blir 0001 i Ti-kolonnen og 0101 i En-kolonnen, altså verdien 15 (desimalt) som ventet.

| | BCD-kode | | | Binær |
|---|---|---|---|---|
| | Hundre | Ti | En | 8 siffer |
| Posisjon | 17 16 | 15 14 13 12 | 11 10 9 8 | 7 6 5 4 3 2 1 0 |
| Initialisering | 0 0 | 0 0 0 0 | 0 0 0 0 | 1 1 1 1 1 1 1 1 |
| 3 venstreskift | 0 0 | 0 0 0 0 | 0 1 1 1 | 1 1 1 1 1 |
| Adder 3 | 0 0 | 0 0 0 0 | 1 0 1 0 | 1 1 1 1 1 |
| 4. venstreskift | 0 0 | 0 0 0 1 | 0 1 0 1 | 1 1 1 1 |
| Adder 3 | 0 0 | 0 0 0 1 | 1 0 0 0 | 1 1 1 1 |
| 5. venstreskift | 0 0 | 0 0 1 1 | 0 0 0 1 | 1 1 1 |
| 6. venstreskift | 0 0 | 0 1 1 0 | 0 0 1 1 | 1 1 |
| Adder 3 | 0 0 | 1 0 0 1 | 0 0 1 1 | 1 1 |
| 7. venstreskift | 0 1 | 0 0 1 0 | 0 1 1 1 | 1 |
| Adder 3 | 0 1 | 0 0 1 0 | 1 0 1 0 | 1 |
| 8. venstreskift | 1 0 | 0 1 0 1 | 0 1 0 1 | |
| Desimalverdi | 2 | 5 | 5 | 255 |

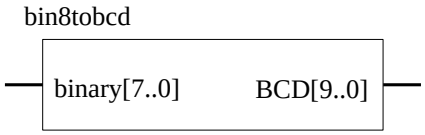
*Tabell 12.1. Binær til BCD-kode med skifting og addering.*

Fra tabellen ses også at etter 6. venstreskift er verdien i ***Ti***-kolonnen lik 0110. Etter å ha lagt til 3, fås 1001 i denne kolonnen. Etter et venstreskift fås 01 i ***Hundre***-kolonnen og 0010 i ***Ti***-kolonnen som er det riktige resultatet (siden 0110 skiftet en plass til venstre gir 1100, desimalt 12 som vi jo har fått i BCD-koden). Etter 8 venstreskift er Binær-kolonnen tom, mens verdien av det binære tallet er tilgjengelig i BCD-kode.

Entiteten kan da være som vist nedenfor, der vi har kalt det binære tallet ***binary***, men BCD-utgangen er kalt ***BCD***. Førstnevnte er på 8 bit mens sistnevnte er på 10 bit.

```
entity bin8tobcd is
   port (binary: in  std_logic_vector (7 downto 0);
         BCD: out std_logic_vector (9 downto 0));
end entity;
```

Entiteten kan illustreres med et blokkskjemasymbol som vist nedenfor.




*Figur 12.16. Blokkskjema for bin8tobcd.*

Arkitekturbeskrivelsen kan da være som vist nedenfor. Den starter med deklareringen av y som er på 18 bit, summen av binærtallet og BCD-verdien. Deretter følger initialiseringen, nullstilling som er nødvendig ved **loop**, og tre venstreskift. De fem neste venstreskiftene (av totalt åtte) følger så. I både En-kolonne og Ti-kolonne testes om verdien er større enn 4, i så fall legges til 3. Etter åtte venstreskift foreligger BCD-koden i posisjon 8 til 17.

```
architecture bin8tobcd_arch of bin8tobcd is
begin
  process (binary)
  variable y: std_logic_vector (17 downto 0);  -- Løpende variabel y
  begin
     for i in 0 to 17 loop
       y(i) := '0';  -- Initialisering (Nullstilling)
     end loop;
     y(10 downto 3) := binary;  -- Tre venstreskift
     for i in 0 to 4 loop  -- Fem venstreskift (av 8)
       if y(11 downto 8) > 4 then  -- Test i En-kolonne
         y(11 downto 8) := y(11 downto 8) + 3;  -- Legg til 3
       end if;
       if y(15 downto 12) > 4 then  -- Test i Ti-kolonne
         y(15 downto 12) := y(15 downto 12) + 3;
       end if;
     y(17 downto 1) := y(16 downto 0);   -- Venstreskift etter test og eventuell addisjon
     end loop;
     BCD <= y(17 downto 8);   -- Ferdig BCD-kode
  end process;
end architecture;
```

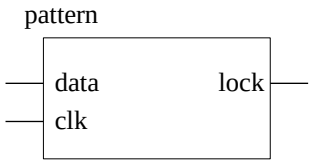
## 12.7.8. Tilstandsmaskiner

### *Mønstergjenkjenner*

Instruksjonen **case** har som nevnt mye til felles med tilsvarende instruksjon i enkelte andre programmeringsspråk. Den er et naturlig valg ved tilstandsmaskiner. Her kan vi bruke eksemplet med å gi ut en ener når vi finner tre påfølgende enere i innkommende data. Entiteten kan for eksempel være:

```
entity pattern is
port(data, clk: in std_logic;
       lock: out std_logic);
end pattern;
```

Her er clk klokkesignalet mens lock er utgangssignalet. Entiteten kan illustreres med et blokk-skjemasymbol som vist nedenfor.




*Figur 12.17. Blokkskjema for pattern.*

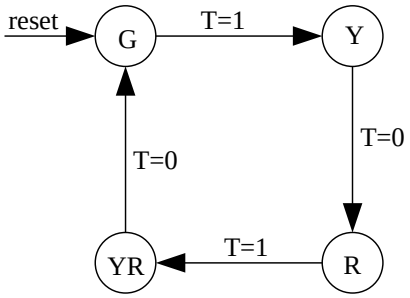
En arkitekturbeskrivelse med **case** kan være:

```
architecture case_when of pattern is
signal q: std_logic_vector(1 downto 0);
begin
  process (data,clk)
    begin
    if rising_edge(clk) then
      if (data = '0') then
        q <= "00";
      else
      case q is
        when "00" => q <= "01";
        when "01" => q <= "10";
        when "10" => q <= "11";
        when others => q <= "11";
      end case;
      end if;
    end if;
  end process;
lock <= '1' when q = "11" else '0';
end;
```

Det er ialt fire tilstander (q lik 0 til 3 desimalt). Det ses at data og clk utgjør sensitivitetslisten. Først på stigende flanke til klokkesignalet clk testes innholdet i innkommende data. Hvis databit er lik 0, blir vi stående i tilstand 0 (q lik 0). Dersom databit er lik 1, skiftes tilstand til tilstand 1 (q lik 1). Dersom neste databit er lik 0, vil vi gå tilbake til tilstand 0. Følgelig må vi ha tre påfølgende enere før vi når tilstand 3 (q lik 3). Da settes utgangen lock lik 1. Ny søking etter tre enere skjer ikke før data inn endrer seg.

## *Trafikklys-styring*

I kapittel 9.6.2 ble en enkel trafikklys-styring presentert. Dersom en lar synteseverktøy ta utgangspunkt i tilstandsdiagrammet presentert i figur 9.24, vist på nytt i figur 12.18, fås resultatet som gitt i det følgende. Betegnelsen G, Y og R brukes for henholdsvis grønt, gult og rødt lys. Et resettings-signal sørger for at vi starter i tilstand G. Det benyttes et klokkesignal T for tidsstyringen av trafikkfyret.

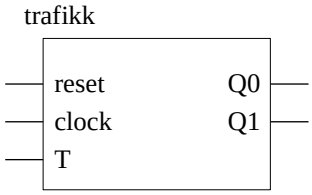



*Figur 12.18. Tilstandsdiagram for trafikklys-styring.*

Entiteten har tre innganger: T, reset og clock. For enkelhets skyld har vi bare tatt med utgangene $Q_0$ og $Q_1$.

```
entity trafikk is
port(reset: in std_logic :='0';
     clock: in std_logic;
     T: in std_logic :='0';
     Q0: out std_logic;
     Q1: out std_logic);
end trafikk;
```

Entiteten kan illustreres med et blokkskjemasymbol som vist nedenfor.




*Figur 12.19. Blokkskjema for trafikk.*

I arkitekturbeskrivelsen opereres med en type i VHDL som har en symbolsk verdi. Dette er typisk for tilstandsmaskiner. Det fås da signaler av denne typen. Legg også merke til at det benyttes to prosesser i denne arkitekturbeskrivelsen. Dette er også en vanlig måte å beskrive tilstandsmaskiner på.

Den første prosessen brukes til å modellere tilstandsregistrene mens den andre brukes til modellering av tilstandsendringer og utgangslogikk. Legg forøvrig merke til bruken av **case**.

```
architecture behavior of trafikk is
  type type_fstate is (G,Y,R,YR); -- Symbolsk type type_fstate
  signal fstate : type_fstate; -- Signal av type type_fstate
  signal reg_fstate : type_fstate;
begin
   process (clock,reg_fstate) -- Modellering av tilstandsregistre
   begin
      if (clock='1' and clock'event) then
         fstate <= reg_fstate;
      end if;
   end process;
   process (fstate,reset,T) -- Modellering av tilstandsendringer og utgangslogikk
   begin
      if (reset='1') then
         reg_fstate <= G; -- Tilstand G er initialtilstand
         Q0 <= '0';
         Q1 <= '0';
      else
         Q0 <= '0';
         Q1 <= '0';
         case fstate is
            when G =>
               if ((T = '1')) then
                  reg_fstate <= Y; -- Bytte til tilstand Y hvis T = 1
               else
                  reg_fstate <= G;
               end if;
               Q0 <= '0';
               Q1 <= '0';
            when Y =>
               if (not((T = '1'))) then
                  reg_fstate <= R; -- Bytte til tilstand R hvis T = 0
               else
                  reg_fstate <= Y;
               end if;
               Q0 <= '1';
               Q1 <= '0';
            when R =>
               if ((T = '1')) then
                  reg_fstate <= YR; -- Bytte til tilstand YR hvis T = 1
               else
                  reg_fstate <= R;
```

```
            end if;
            Q0 <= '0';
            Q1 <= '1';
          when YR =>
            if (not((T = '1'))) then
               reg_fstate <= G; -- Bytte til tilstand G hvis T = 0
            else
               reg_fstate <= YR;
            end if;
            Q0 <= '1';
            Q1 <= '1';
          when others =>
            Q0 <= 'X';
            Q1 <= 'X';
        end case;
      end if;
   end process;
end behavior;
```